

2013-04-25

Ultra-Low Power Sub-Threshold Nanoscale CMOS Path Planning Cores

Wu, Ryan

Wu, R. (2013). Ultra-Low Power Sub-Threshold Nanoscale CMOS Path Planning Cores
(Master's thesis, University of Calgary, Calgary, Canada). Retrieved from
<https://prism.ucalgary.ca>. doi:10.11575/PRISM/28545
<http://hdl.handle.net/11023/626>

Downloaded from PRISM Repository, University of Calgary

UNIVERSITY OF CALGARY

Ultra-Low Power Sub-Threshold Nanoscale CMOS

Path Planning Cores

by

Ryan Chung-Yen Wu

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

CALGARY, ALBERTA

APRIL, 2013

© Ryan Chung-Yen Wu 2013

UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled “Ultra-Low Power Sub-Threshold Nanoscale CMOS Path Planning Cores” submitted by Ryan Chung-Yen Wu in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE.

Abstract

This project aims to develop a specialized processor that is optimized for power and algorithm efficiency. The optimization would be targeted for a path-planning algorithm for micro-robots such as insect-bots, UAVs (unmanned aerial vehicles) and for nano-medicine.

This processor would provide a computation platform that is smaller, lighter and more power efficient than conventional general-purpose processors to allow these micro-bots to reach a high level of intelligence. These micro-bots would then be able to navigate themselves to their destination, and perform simple tasks and data processing.

To achieve this goal, the processor was designed using a custom computing architecture implemented in RTL, then synthesized and detailed designed to layouts that are then fabricated into a physical processor. The processor design target was to operate at 100kHz, while consuming only tens of microwatts and weighing only milligrams.

Acknowledgements

I would like to thank my supervisor, Dr. Sebastian Magierowski, for his constant academic support. His friendliness, encouragements and vast knowledge in life and engineering has inspired me for higher achievements and has guided me through my degree. Also, I would like to thank my co-supervisor, for his generous financial support throughout my degree.

Secondly, I would like to thank my colleagues, Zhixing Zhao and everyone else in F.I.S.H. Lab for their technical support, and providing a comforting and motivating work environment.

Lastly, and most importantly, I want to thank my family and friends for their continuous encouragements.

Above mentioned, are people whom I am truly grateful for to have in my life. Whom have made significant changes in my achievements and career, and to whom, I would like to dedicate this work to.

Table of Contents

Abstract	ii
Acknowledgements	iii
Table of Contents	iv
List of Tables	vii
List of Figures and Illustrations	viii
List of Symbols, Abbreviations and Nomenclature	x
1 INTRODUCTION	1
1.1 Motivation	1
1.2 Background	3
1.2.1 Nonlinear Path Planning Algorithm	4
1.2.2 Previous Work	7
1.2.3 The MIPS Instruction Set Architecture	8
1.3 Thesis Contributions	10
1.4 Thesis Outline	11
2 METHODOLOGIES	13
2.1 Planning Algorithm Translation	13
2.1.1 Simplification	13
2.1.2 Customization	14
2.1.3 Verification	15
2.2 Architecture	16
2.2.1 Digital vs. Analog	18
2.2.2 Multicycle Processor	20
2.2.3 Word-Addressable Memory	21
2.2.4 Instruction Set Architecture	21
2.3 Sub-threshold Transistors	27
3 PROCESSOR & MEMORIES	32
3.1 Arithmetic Logic Unit	32
3.2 Static Random Access Memory (SRAM)	37
3.3 Memory System	45
3.3.1 Memory Organization	48
3.3.2 Caching Scheme	49
3.3.3 Cache Controller	52
3.4 Look-Up-Table (LUT)	54
3.5 Analog Components	55
3.6 Test Circuits	59
4 FULL CHIP INTEGRATION	62
4.1 IBM 130-nm CMOS Technology	62
4.2 TSMC 90-nm Technology	64
4.2.1 ADC and DAC Integration	66

5	SIMULATIONS AND TEST RESULTS	69
5.1	MATLAB Simulations	69
5.2	Verilog Simulations	70
5.2.1	Core Simulations	70
5.2.2	Look-Up-Table Simulations	73
5.2.3	Final Verification in Verilog	75
5.3	Analog Design Environment (ADE)	75
5.4	SPICE	77
5.5	FPGA Tests	78
5.6	Detailed Final Simulation	79
5.7	Testing	80
6	CONCLUSION	83
6.1	Accomplishments	83
6.2	Future Work	84
6.2.1	Customized Library	85
6.2.2	Dynamic Tuning	85
6.2.3	Integration	86
6.2.4	Architecture	86
	Bibliography	87
A	Algorithms	91
A.1	Original Algorithm	91
A.2	Original Algorithm in Matlab	93
A.3	Algorithm in Assembly Language	94
A.4	Algorithm in Machine Code	96
A.5	Machine Code in Verilog Byte-Addressable Memory	97
B	Scripts and Programs	99
B.1	MATLAB Processor	99
B.2	Perl Assembler	99
B.3	Synopsys Compiler Script	102
B.4	Encounter Top-Level Command Script	106
B.5	Layout Synthesis Configurations	106
B.6	Synthesis Pin Placement	108
B.7	Floor Planning	111
B.8	Power Planning	111
B.9	Cell Placement	112
B.10	Clock-Tree Synthesis	112
B.11	Routing	113
B.12	Verification	113
C	Verilog Codes	115
C.1	Core	115
C.2	MIPS	116
C.3	Memory Controller	132
C.4	Test Signal Routing	137
C.5	Look-up Tables	141
C.6	Memory	146

C.7	ADC Controller	158
D	Additional Supporting Material	160
D.1	Look-Up-Table Test Results	160

List of Tables

2.1	Supported Instruction List for the Modified-MIPS Processor	23
3.1	LUT function codes	57
4.1	IBM P13 Fabrication Detail	62
4.2	TSMC 90-nm Fabrication Detail	66

List of Figures and Illustrations

1.1	A “micro-UAV” in the form of a robot fly [3].	2
1.2	A feedback representation of a robot path planner.	4
1.3	An oblique view of the feedback mechanism applied recursively to handle more sophisticated operations.	6
1.4	MIPS instruction structure	10
2.1	Original model of the path planning simulation [9].	15
2.2	New Simulink model of the path planning simulation.	16
2.3	An example calculation of the algorithm in Simulink. The solid line represents the robot’s path through a 2D landscape containing obstacles denoted by \times	17
2.4	Schematic of the modified-MIPS processor design.	19
2.5	Expected processor performance at different voltage levels. The dashed and solid curves refer to instruction fetching from the cache and external memory respectively.	26
2.6	Voltage scaling delay in TSMC’s 90-nm standard cell digital library inverter circuit ($FO = 4$).	28
2.7	Voltage scaling delay in TSMC’s 90-nm standard cell digital library inverter circuit ($FO = 4$).	29
2.8	Dynamic and static energy/cycle tradeoff in a 65-nm CMOS ALU [28].	30
2.9	Total energy/cycle consumption as a function of V_{DD} for various CMOS technology generations [28].	31
3.1	Schematic of the modified-MIPS processor design.	33
3.2	Schematic of the arithmetic logic unit.	35
3.3	10-transistor (10T) SRAM cell.	39
3.4	Word line control circuit.	40
3.5	36-bit word line arrangement.	41
3.6	32-word block signal conditioner.	41
3.7	Schematic of the Full 1-kByte SRAM.	43
3.8	Layout of the 10T SRAM cell using TSMC 90-nm technology.	44
3.9	WL and rWL Buffers (left 1/4); BL rBL read/write logic (right 3/4), in TSMC 90-nm CMOS.	45
3.10	All of the SRAM control circuits for a 32-word block in TSMC 90-nm CMOS.	46
3.11	Layout view of a 32-word SRAM block in TSMC 90-nm CMOS.	47
3.12	Full layout of the 256-word SRAM in TSMC 90-nm CMOS.	48
3.13	16-bit memory address structure.	49
3.14	Structure of the cache memory.	52
3.15	Cache controller states	53
3.16	Comparison of LUT outputs vs Real values of $\sin(x)$	55
3.17	The system arrangement including the location of the data converters.	56
3.18	Architecture of the successive-approximation ADC. From [34].	57
3.19	Architecture of the DAC.	58

3.20	Layout of the successive-approximation ADC.	59
3.21	Layout of the binary capacitive weighted DAC.	60
4.1	Design flow for VLSI Circuits using Synopsys [®] and Cadence [®] toolsets. . . .	63
4.2	Final layout of the IBM p13 chip (without metal fill).	65
4.3	Final layout of the TSMC 90-nm chip (without metal fill).	67
5.1	Result of translated algorithm (finite arithmetic, machine commands, etc.) simulation in Simulink	70
5.2	Block diagram of the MATLAB simulation.	71
5.3	Result of the original algorithm simulation in Simulink.	72
5.4	Results of the I/O test bench using NC_Verilog.	72
5.5	Look-Up-Table outputs of a sine function.	73
5.6	Look-Up-Table outputs of a negative hyperbolic-sine function	74
5.7	Look-Up-Table outputs of a hyperbolic-secant function	74
5.8	Single-Bit SRAM cell test setup	76
5.9	Single-Bit SRAM cell test simulation results.	77
5.10	Power simulation of the 90-nm chip in operation at 100 kHz.	80
5.11	Schematic of the power simulation for the 90-nm chip.	80
5.12	ICRCYSUP Chip's Processor Power Consumption vs. Supply Voltage	81
5.13	Hardware Measurement Test Setup	82
D.1	Look-Up-Table outputs of a sine function	161
D.2	Look-Up-Table outputs of a cosine function	161
D.3	Look-Up-Table outputs of a cosecant function	162
D.4	Look-Up-Table outputs of a secant function	162
D.5	Look-Up-Table outputs of a negative sine function	163
D.6	Look-Up-Table outputs of a negative cosine function	163
D.7	Look-Up-Table outputs of a negative cosecant function	164
D.8	Look-Up-Table outputs of a negative secant function	164
D.9	Look-Up-Table outputs of a hyperbolic-sine function	165
D.10	Look-Up-Table outputs of a hyperbolic-cosine function	165
D.11	Look-Up-Table outputs of a hyperbolic-cosecant function	166
D.12	Look-Up-Table outputs of a hyperbolic-secant function	166
D.13	Look-Up-Table outputs of a negative hyperbolic-sine function	167
D.14	Look-Up-Table outputs of a negative hyperbolic-cosine function	167
D.15	Look-Up-Table outputs of a negative hyperbolic-cosecant function	168
D.16	Look-Up-Table outputs of a negative hyperbolic-secant function	168

List of Symbols, Abbreviations and Nomenclature

Symbol

Definition

U of C

University of Calgary

Chapter 1

INTRODUCTION

1.1 Motivation

This project aims to develop a specialized processor intended to provide adequate computational resources for mobile micro-robots (for example, insect-bots) with severely constrained weight and power allowances. The processor considered in this thesis adapts the so-called *MIPS* architecture outlined briefly in Section 1.2.3. In terms of emerging technical parlance the broader context of the work described herein is that of the “conservation core”, a term referring to specialized processors intended to minimize the power consumption of specific tasks [1]. Such cores are intended to serve as the constituents of a larger computing system that, as a whole, achieve a complex functional behaviour with minimal power requirements.

The specialization of the processor discussed in this thesis is for autonomous navigation, its purpose is to implement a controller capable of calculating a path that a robot can follow towards some physical target. The sensory information about the environment and the robot’s state are assumed to be given. The circuitry needed to actuate the machine is not considered. Of course, a controller capable of carrying out such path planning calculations can be generalized to handle a larger set of problems as long as those problems can be formulated in an appropriate goal-seeking manner. For example, other potential applications in addition to insect-bots include unmanned aerial vehicles (UAVs), devices for nano-medicine and situations wherein non-computationally intensive applications require low weight, size and power. In general, it seems that the research community itself has expanded its view of planning to coincide with the pursuit of automated problem solving techniques in general [2].

An example of a micro-robot platform is the robotic fly described in [3]. A photograph of this robot is shown in Fig. 1.1. In this case the design measures roughly 2 cm between

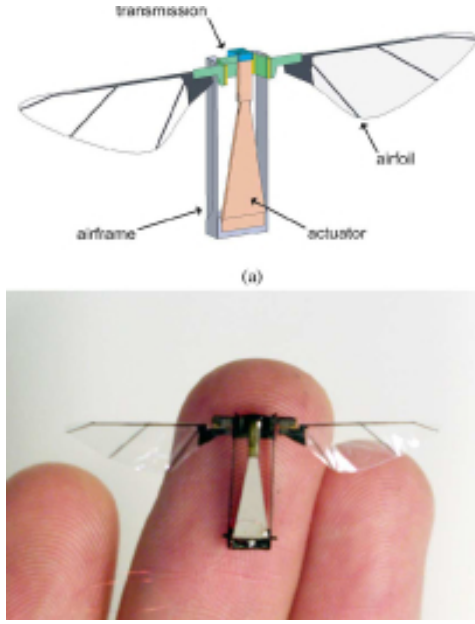


Figure 1.1: A “micro-UAV” in the form of a robot fly [3].

wingtips and is expected to weigh in the range of 120 mg. Only 12 mg of this design are set aside for the electronics which is about the weight of a $4 \times 4 \text{ mm}^2$ silicon chip. Accounting for packaging, pads, as well as power supply circuitry, signal conditioning, driving circuitry, and memory perhaps only a $1 \times 1 \text{ mm}^2$ area would be available for the raw compute circuitry of the proposed robot fly. A rough analysis of Intel CMOS microprocessor technology over the last 20 years indicates a transistor density of roughly 5×10^{-3} transistors per chip area normalized to the square of the minimum MOSFET gate length L_{gmin}^2 for a particular technology. For example, a 130-nm Intel CMOS technology used in the context of a microprocessor fits about $5 \times 10^{-3} / (130 \times 10^{-6})^2 \approx 3 \times 10^6$ transistors/ mm^2 . This number of transistors is slightly less than the amount used by the $0.8\text{-}\mu\text{m}$ Pentium chip released in 1993 [4]. Thus, it would seem that even this challenging micro-robot example may leave enough space for the realization of an adequate computer (at least for primitive operations).

Because of the limited weight and size capacity (by definition) of micro-robots, they also have very limited access to power with which to achieve an acceptable level of computational intelligence. For example, to maintain hover, the robot fly outlined in [3] requires approxi-

mately 5 mW. A Pentium processor allowed to run at 10 MHz requires roughly $1000\times$ this power level. Thus although a seemingly reasonable transistor count may be achievable for even an extreme contemporary example of micro-robotics, power consumption looms as a potentially challenging problem.

At 10-MHz clock speeds the Pentium can operate at roughly 20 million instructions per second (MIPS). Based on Moravec’s approximation of 50,000 MIPS/g of biological neural matter [5], the suitability of a Pentium for calculations capable of emulating an animal’s (admittedly complex) behaviour seems limited as this microprocessor achieves roughly 100 MIPS/g. Accounting for power (again, about 5 W for the 10-MHz Pentium circa 1993) the Pentium achieves 26 MIPS/mg/W, a measure we will refer to as cognitive efficiency. For comparison, a honeybee with a 20 mg neural component and a metabolic rate of roughly 10 mW manages an approximate computational output of 1000 MIPS and a cognitive efficiency of 5000 MIPS/mg/W.

Much better cognitive efficiencies from processor technologies are possible. For example in [6] 24,200 MIPS/mg/W of cognitive efficiency was achieved in a custom designed 0.25- μ m CMOS processor. The architecture uses a number of simplifications (e.g. no pipelining), controls (e.g. clock gating) plus low supply voltage (1.0 V) and low frequency of operation (100-kHz) to achieve its performance. Of course this also limits its peak output to only 0.5 MIPS. Clearly achieving both peak and efficient performance is a challenging objective.

1.2 Background

This work uses a Lyapunov-based non-linear path-planning algorithm, a MIPS architecture, and the results of [7, 8, 9, 10] as the theoretical and functional components underpinning the presented CMOS processor. Results reported in previous work on this algorithm that are critical to the design presented in this thesis have been retested and verified.

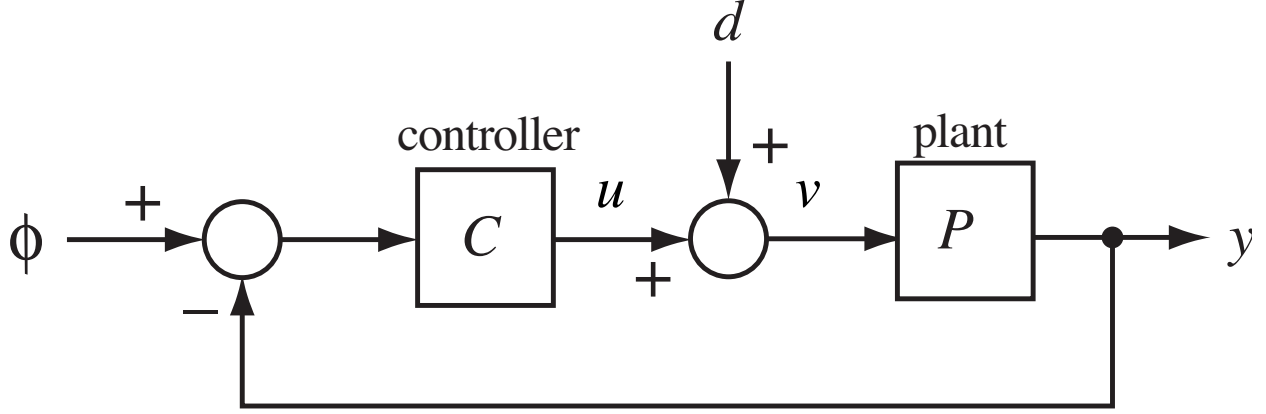


Figure 1.2: A feedback representation of a robot path planner.

1.2.1 Nonlinear Path Planning Algorithm

The algorithm that this processor is optimized for is a Lyapunov-based non-linear path-planner [7]. The robot motion induced by this controller¹ looks similar to the movement generally ascribed to insects and hence is sometimes referred to as insect-like in its characteristics [10]. The goal of this approach however was not to necessarily produce an insect-like locomotion, but rather to develop a more flexible and scalable means of autonomous problem solving that simultaneously promised an efficient hardware implementation. A thorough justification of this point is beyond the scope of this thesis (which is focused on a suitable translation of the idea into integrated circuit form), but is detailed in [7, 8, 9, 10].

The path-planning idea implemented in this thesis abstracts the search problem to a feedback mechanism as illustrated in Fig. 1.2. In control system terms, this picture represents a scenario wherein we seek to produce an output, y , from some entity (a plant), P , that matches an input ϕ . The output y produced by P is a result of the stimulus, v , that P receives. This stimulus is (partly) generated by a controller, C , in the form of the signal u . In turn, u is the response of C to the difference, $\phi - y$. In any realistic situation we need to account for the presence of disturbances, d , beyond our direct control and this is shown in Fig. 1.2 as well. For each y produced, C adjusts u , in the best way it can to minimize $|\phi - y|$

¹In this thesis we use the terms controller and path-planner interchangeably.

(or perhaps some more sophisticated measure of distance between two vector signals).

In the path-planning context P represents an aspect of the robot and the environment within which it operates. For example P could be as simple as a differential steering mechanism operating on a floor with some coefficient of friction. The input, v , to P could be the differential steering commands consisting of some signals proportional to the intended translational and rotational velocity desired of the device. The “environmental” output, y , could be a measure of the device’s location and orientation. Similarly, ϕ , could be a measure of the goal’s location in terms of Cartesian coordinates. When y and ϕ match or are “close” to some suitable degree the machine’s goal has been reached and its problem is considered “solved”.

The reader may be able to imagine that such a scheme has the potential to scale towards more sophisticated levels of operation. One approach is to nest feedback models into a recursive hierarchical pattern as illustrated in Fig. 1.3. Herein, the system is split into three levels with some ostensibly complex goal defined by ϕ_3 fed in at the top level and achieved through the combined operation of lower-level behaviours. Such a scheme follows the controller hierarchy imagined by Brooks 20 years ago, a proposal that launched significant activity into reactive robotics [11], a contrast to the deliberative systems characteristics of classical artificial intelligence approaches.

The computational heart of the robot is located within C . In the Lyapunov-based algorithm under consideration, this block is simply a nonlinear ordinary differential equation solver implementing a state-space description of the machine’s path planning ability. In general, its function can be expressed with

$$\dot{\mathbf{u}} = \mathbf{A}\mathbf{u} + \mathbf{B}\mathbf{e} \tag{1.1}$$

where \mathbf{e} is the error value generated by $\phi - \mathbf{y}$. The lower case symbols used in (1.1) are simply vector versions of those presented in Fig. 1.2. The matrices \mathbf{A} and \mathbf{B} effectively describe the system (the robot and its environment), their derivation is the purview of

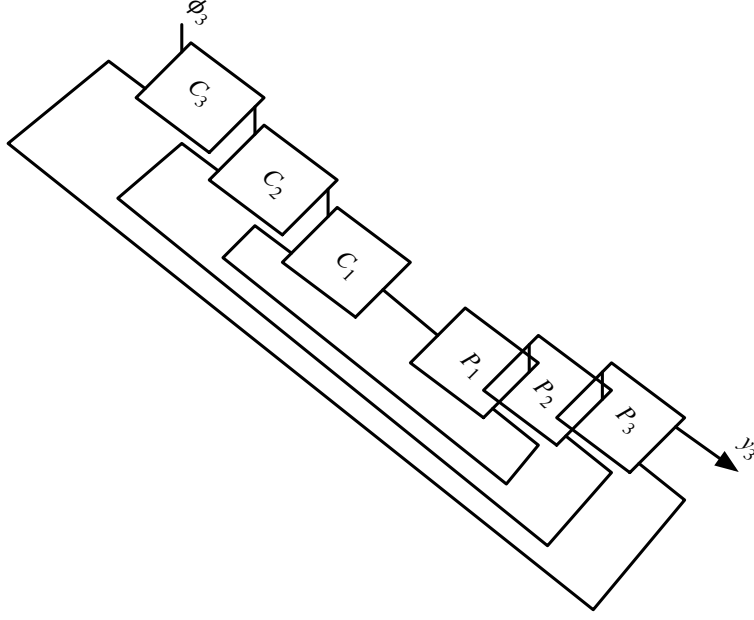


Figure 1.3: An oblique view of the feedback mechanism applied recursively to handle more sophisticated operations.

nonlinear control theory as outlined in [7]. We now briefly elaborate on the implications of achieving this manner of controller.

The state-space formalism for generating commands for mobile robots (i.e. the control signal, \mathbf{u}) obviously differs from the classical scripting technique that relies on such constructs as **if-then**. In some sense, we can think of this as another programming language, albeit one “compiled” by the techniques of nonlinear control theory which are responsible for finding the appropriate expressions for \mathbf{A} and \mathbf{B} while working under the constraints imposed by P , d (see Fig. 1.2) and the type of convergence desired between \mathbf{y} and ϕ . In the particular algorithm under consideration in this thesis the so-called Lyapunov synthesis technique was employed in conjunction with the backstepping approach [12] to obtain \mathbf{A} and \mathbf{B} .

Considerations of the nonlinear methods behind the details of (1.1) are (far) beyond the scope of this thesis. In short, the Lyapunov technique serves as an efficient means of deciding on the stability of a dynamical system. This stems from the fact that it needs only to solve for a scalar description of the system rather than its full set of differential

equation expressions. Run “backwards”, the Lyapunov analysis approach can serve as a means of synthesis, by deciding what parameters are required to achieve stability (i.e. to have \mathbf{y} ultimately approach ϕ and achieve a solution). The backstepping approach is a means of retaining the solution derived from the Lyapunov synthesis while including extra filtering into the system to fine-tune the machine’s response. These actions don’t change the structure of (1.1), but they do modify the final expressions for \mathbf{A} and \mathbf{B} . As mentioned however, this part of the system design is not the concern of this thesis. Rather the focus is on presenting an IC realization of the algorithm that is compact, power efficient, scalable, and capable of supporting a suitable array of \mathbf{A} and \mathbf{B} as called upon by a variety of plants and nonlinear derivation schemes.

In its base implementation, the algorithm uses 25 matrix multiplications and some non-linear functions to generate a new vector that instructs the robot to adjust its direction and speed. The algorithm assumes knowledge of its current position and location of its destination. Using sensors, it detects obstacles in proximity and finds ways to navigate around those obstacles towards its destination. Tested in [8, 9, 10], the algorithm proves to be effective in obstacle avoidance, navigating to a destination, and is scalable and “light-weight” in terms of its power and computational requirements. It can be made to perform coarse calculations or more precise path calculations using bigger matrices; navigate in a 2-dimensional vector environment or possibly a 3-dimensional vector space; or it can be designed to run recursively for bigger applications. As already indicated, the work in this thesis is not only designed to run this algorithm efficiently, but is also designed to be flexible and scalable to take full benefit of the potential advantages of this approach.

1.2.2 Previous Work

A contribution to the general area of low-power micro-robot IC design in the form of a simplified microprocessor [6] has been outlined above. In the context of the particular algorithm under consideration in this thesis, a mixed-signal processor (i.e. a combination of analog and

digital parts) has been designed by Mihir Naik [8]. The mixed-signal implementation differs from the one presented in this thesis in that it is less flexible and possibly consumes less power (a thorough system-wide power analysis was not provided in [8]). It uses an analog multiplier, a digital controller and three separate memory blocks to execute the hard-coded path-planning algorithm (see page 36 of [8] for a block diagram). By attempting to compress the potentially power hungry multiplier realization into a specialized analog component such a mixed-signal approach suggests the possibility of very low power operation. However, workload distribution is best done by digital circuitry and the need to interface the analog and digital portions of a mixed-signal system require data conversion blocks which start to compromise the power savings. Still, [8] serves as a special-purpose path-planner and an IC-design proof of concept where the application of a Lyapunov-based path-planning algorithm can achieve exceptionally low power.

1.2.3 The MIPS Instruction Set Architecture

When a program gets compiled, the compiler converts a high-level programming language (e.g. C++) into assembly instructions, in which, each instruction represents a simple elementary task defined by the architecture. Each of these instructions directly corresponds to a 32-bit machine code² which a processor can read and execute.

MIPS, short for *Microprocessor without Interlocked Pipeline Stages*, is a computing architecture, which defines an instruction set designed to execute complex software using basic commands. Baseline versions of this architecture, MIPS32 and MIPS64, are published by MIPS Technologies Inc. [14] Other popular architectures include IA-32, IA-64, SPARC, PowerPC, etc. MIPS is one of the most common architectures found in embedded applications and is one of the best known examples of the *Reduced Instruction Set Computers* (RISC) class of architectures.

²Machine code word size typically varies between 8 and 64-bits depending on the microprocessor. Generally, most embedded processors favour a 32-bit design [13].

In a RISC instruction set architecture, each instruction carries out a simple task and instructions are fixed-length. The technique began to gain support in the computing field in the 70's with the work of John Cocke at IBM but is most commonly attributed to the efforts of David Patterson (University of California Berkeley) and John Hennessy (Stanford University) to construct efficiently pipelined machines. In fact, MIPS was a direct outgrowth of the Stanford efforts into RISC. Besides MIPS the ARM (Advanced RISC Machines) architecture is another extremely popular example of this design approach. Because of the simpler tasks and fixed-length instructions, RISC architectures allow the processor design to be simpler, more efficient, and amenable to scaling and customization. For embedded designs, the RISC approach is by far the dominant paradigm [15].

Each of the MIPS instructions is 32 bits in length, where the first 6 bits represent the op-code field that defines the type of instruction being processed [16]. Shown in Fig. 1.4 are the three most common types of instructions expressed in terms of their key constituent fields.

R-type instructions operate on three operands each read from or written to one of three registers in the processor's *register file* (or *register set*). The variables, *rs*, *rt*, and *rd*, denote the address of the register within the register file where each of the operands referenced by the instruction are located or are to be placed.

I-type instructions also operate on three operands. Two of these are again associated with locations (*rs* and *rt*) in the register file and a third, an *immediate value* is contained directly in the instruction itself.

J-type instructions operate with a 26-bit immediate field. As with I-type commands, immediate values are defined within the instruction, which could be 16 bits or 26 bits and be used for additions, comparisons or defining addresses. As noted, register locations are defined using 5 bit addresses (see Fig. 1.4) in the instruction and hence provide up to $2^5 = 32$ register locations for instructions to operate on.

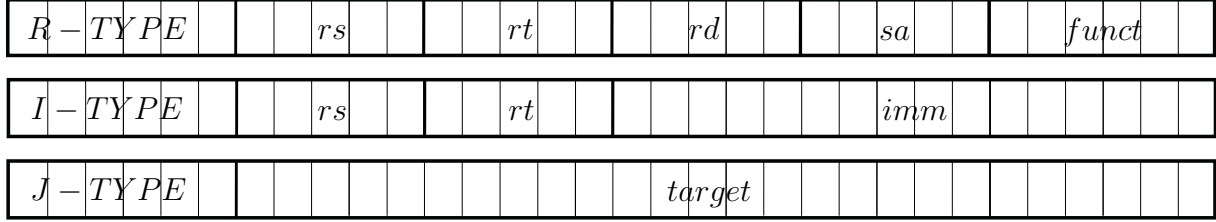


Figure 1.4: MIPS instruction structure

Finally, in addition to the op-code, there are the *sa* and *funct* fields that define more specific tasks for R-type instructions [4].

1.3 Thesis Contributions

This thesis presents a different approach to low-power IC computing specifically optimized for the non-linear path-planning algorithm introduced in [7]. As noted above an IC design of this algorithm was also presented in [8] (although not fabricated). As opposed to [8], the sub-threshold processor presented in this report is very flexible in accommodating changes in the algorithm, tuning for different environments, and tuning the data processing. Using the MIPS architecture as its foundation, unnecessary functionalities are removed while new functionalities are added to support and improve the efficiency of the algorithm. Paired with a custom designed SRAM (static random access memory), this processor is capable of caching used instructions while providing expandability to accommodate a bigger and more complex variation of the algorithm by using cache as data memory.

The non-linear path-planning algorithm used, proven in [9, 10] to behave with exceptional stability while processing a signal corrupted by high levels of noise (e.g. from the sensor inputs), has been re-tested in this work to verify that it navigates to a destination around obstacle(s). The algorithm has then been simplified and broken down into machine code for a custom architecture presented in this thesis using a custom built assembler (without changing the algorithm itself).

After carefully examining the algorithm for repeated patterns and instructions, the archi-

itecture presented in this thesis was optimized to run these patterns efficiently. This includes careful selection and design of the architecture and micro-architecture. The architecture is similar to MIPS, but much simpler with support for only elementary instructions necessary for the algorithm. Additional support for multiply-and-add and several non-linear functions has been added in order to run the algorithm as well as to provide support for possible variations in the algorithm. Memory space has also been modified to work with the custom designed memory system with cache controller, cache memory, external flash, sensor inputs, and motor outputs. The memory system designed is a significant part of this thesis as it is critical for such flexibility and expandability.

Micro-architecture, another significant part of the work in this thesis included the design of an efficient ALU that can execute all the instructions in this architecture, and ways of executing non-linear functions efficiently and reliably. Micro-architecture design greatly influences the power efficiency, reliability and computational power of the processor.

With all the above parts depending upon each other, it was critical that they all work in sync and are optimally designed to work with one another. This thesis presents the full design from algorithm to transistor, with two iterations of the design fabricated in a 130-nm and a 90-nm CMOS technology. Simulation results as well as test results show that the design is further scalable to variable complexity of the algorithm and also scalable to different (more advanced) technologies.

In comparison to a general purpose processor, the processor presented in this thesis consumes less space, less power and weighs less. Hence making it exceptionally suitable for small robots (micro-bots) that require navigation capability and light computational needs.

1.4 Thesis Outline

This thesis will explain the different hierarchies of design and the results. The following Chapters are organized as follows:

- Chapter 2: How the methodologies were selected and developed along with the proposed architecture.
- Chapter 3: Implementation details of the processor and each of its customized components to achieve the proposed functionality.
- Chapter 4: How the processor was integrated into the full chip and fabricated in different technologies.
- Chapter 5: The simulation results of each component and the full design. Also, the test results of the fabricated chips.
- Chapter 6: Reflect on the results of the chips, summary of accomplishments in this work, and how the design can be further improved.

Chapter 2

METHODOLOGIES

The planning algorithm's implementation in silicon followed a design methodology consisting of three main steps. These were carried out in an iterative fashion and consisted of: 1.) the translation of the algorithm into a form optimized for execution on a low-power von Neumann machine, 2.) the design of a suitable computational instruction set architecture for the algorithm, 3.) the design of the physical (transistor-based) make-up of the computing machine. These steps are elaborated in the following sections of this chapter.

2.1 Planning Algorithm Translation

The translation of the algorithm into a form suitable for a low-power von Neumann machine consisted of three main elements. The simplification of the core algorithm's computations, the customization of the algorithm to the needs of an instruction-based machine and simulation-driven verification of the resulting machine-level program.

2.1.1 Simplification

The Lyapunov-based non-linear path-planning algorithm implemented in this work includes 25 matrix multiplications, 4 non-linear functions, 12 addition/subtractions and an inverse. A summary of the functions executed as part of this algorithm is given in Appendix A.1. All of these operations can be either reduced, simplified or pre-calculated.

For example, the inverse can be pre-calculated, all multiplications by 0 or 1 can be eliminated, and some calculated values such as ***BtP*** (outlined in Appendix A.1) can be saved and reused in future calculations. Also, because of the inherent characteristics of this algorithm, there are many repeating values within the matrices all of which can be reduced

when broken down into elementary functions.

Using this method and a customized computing architecture, the 40 line MATLAB code (executing 61 calculations) capable of describing the controller (see Appendix A.2 for a copy of the controller MATLAB code) can be translated into 110 lines of assembly instructions as shown in Appendix A.3. Given that the MATLAB code consists of non-linear functions and matrix arithmetic, 110 lines of assembly instructions represent a significant reduction in complexity. By comparison, the mixed-mode architectural formulation of this controller presented in [8] implemented the algorithm with a state machine consisting of roughly 200 states.

2.1.2 Customization

One of the benefits of the algorithm employed for the implementation of the path planning function is the constancy of its structure. That is, the scale and numerical settings of the approach adapt to problem details, but the form of the algorithm (i.e. the calculations required of the Lyapunov and backstepping schemes and their sequence) remains constant. This property made it practical to manually identify the computational optimizations applicable to the MATLAB code (as noted in the previous section) and then to manually implement these in machine instructions for the custom processor (a discussion of the core's instruction set appears below). For the realization of larger problems (e.g. planning over a number of behavioural levels like physical location search, resource gathering, group behaviour, etc.) a custom compiler may be a useful tool.

Following manual translation to assembly code an assembler written in Perl (Appendix B.2), was used to convert the assembly code to machine code and/or a verilog memory block. This assembler has proven to be very useful as test programs are much easier to write in assembly than machine code, and also, assembly is significantly easier to debug.

In addition to the algorithm itself, the simulator (written in MATLAB/Simulink) has also been modified to enhance the testing phase of the project. It has been made more

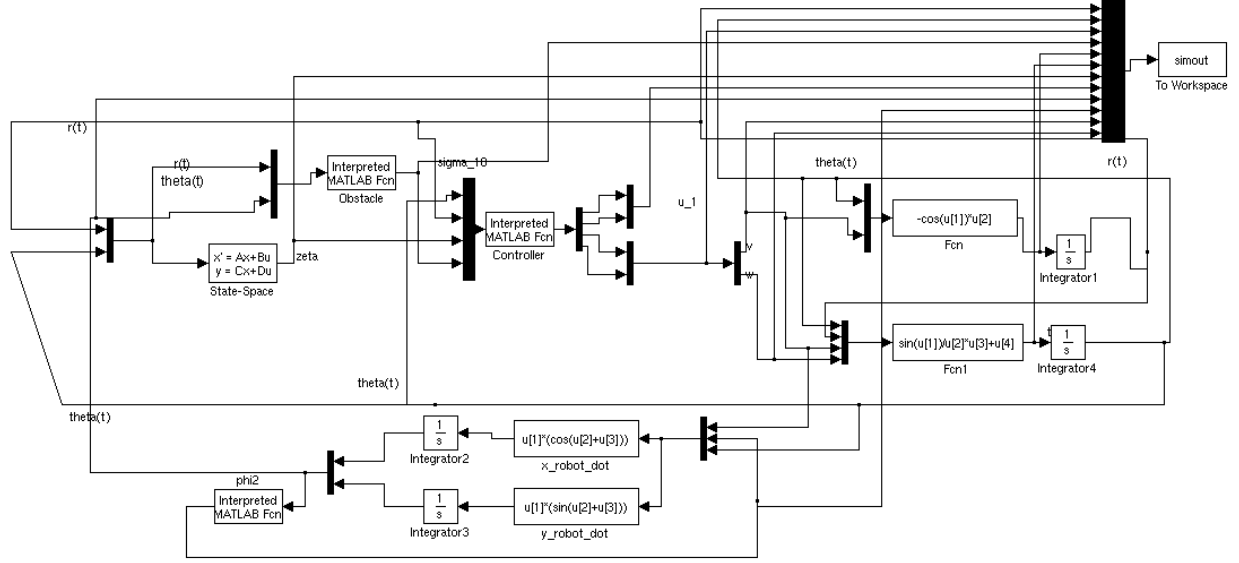


Figure 2.1: Original model of the path planning simulation [9].

organized, easier to read, has data recording capability on all the important variables, and has maximum/minimum limits on the motor for a more realistic simulation. Also, a clock has been added to the controller that runs each instruction sequentially (rather than a full loop each time within which each calculation is completed). As an additional feature, the revised simulator displays the simulated path in real-time so it is no longer necessary to wait for the entire simulation to complete. For easy comparison, the original Simulink model presented in [9] is shown in Fig. 2.1, and the new one is shown in Fig. 2.2. Please see Chapter 5.1 for more details on the MATLAB simulation.

2.1.3 Verification

As a final verification step before the main hardware design, the modified algorithm in machine code (see Appendix A.4) was put through a MATLAB simulated processor (see Appendix B.1), that runs through the full path planning simulation the same way [8] and [10] did.

A typical output of the simulation is shown in Fig. 2.3. The graph represents a 2-D map of the simulation environment, where the x's represent obstacles and the line represents

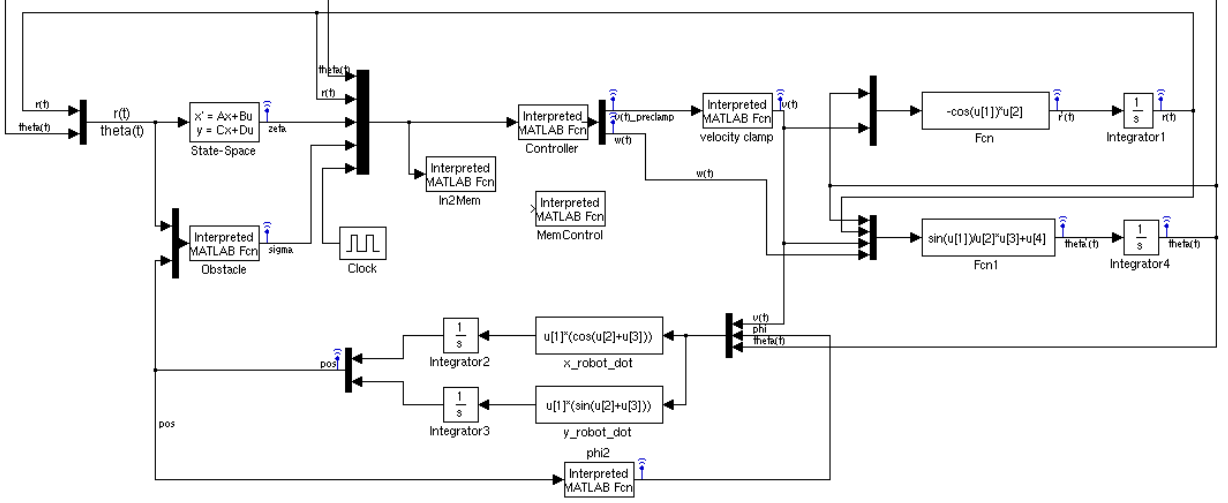


Figure 2.2: New Simulink model of the path planning simulation.

the calculated navigation path of the robot from coordinate (0,0) to (10,10). This result matches the prediction made using the original MATLAB description of the algorithm and demonstrates that after all the computational simplifications and modifications applied to it the algorithm still works to navigate to destination and avoid obstacles. There are more details regarding the simulation in Chapter 5.1.

2.2 Architecture

The architecture¹ of the processor described in this thesis is the result of many assessments and iterative design choices that took into account different methodologies useful for the achievement of flexibility, expandability and power. While Chapter 3 discusses the memory and microarchitecture components more closely, this section primarily describes the instruction set architecture of a modified-MIPS system and the constraints that shaped it.

For the reader's reference a schematic of the microarchitecture used for the processor is

¹Following generally used contemporary terminology, we use “architecture” as an overarching term covering instruction set, organization, and hardware components of a computing system. As noted in [17] the term took on such a general flavour with the increasing integration of computers within one semiconductor substrate and should be distinguished from the more specific “instruction set architecture” and “microarchitecture” designations.

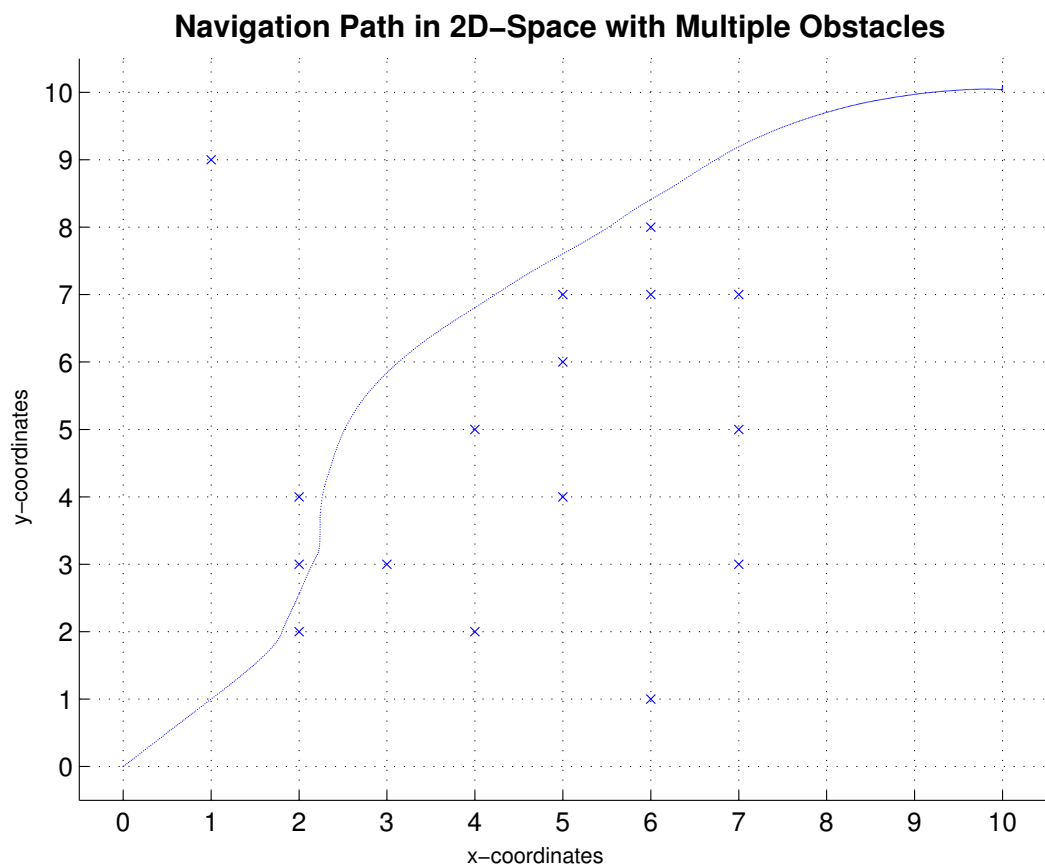


Figure 2.3: An example calculation of the algorithm in Simulink. The solid line represents the robot's path through a 2D landscape containing obstacles denoted by \times .

shown in Fig. 3.1. The design conforms to the classic von Neumann arrangement and is coarsely partitioned into a control module (Controller plus ALU Control blocks in Fig. 3.1) and a datapath module (the remaining assortment of blocks in Fig. 3.1), a very common means of partitioning digital systems [17]. The datapath is responsible for carrying out operations on the input data while the control determines the sequence in which these operations are executed.

2.2.1 Digital vs. Analog

The first choice in architectural design was actually quite a primal one: a decision between an analog or a digital realization of the computer. The choice for a digital implementation and the decision to apply it towards a modified-MIPS system was made to achieve a larger degree of operational flexibility. In its application of general control theory techniques, the path-planning algorithm proposed in [7] is inherently capable of addressing many different environments and problem settings; it's potential for scaling to more complex and higher-dimensional problems in a recursive fashion, although not yet rigorously proven, shows promise through earlier case studies [10]. It was obviously incumbent on the implementation to somehow preserve this inherent capability within the physical constraints posed by micro-robotic systems.

In initial attempts at implementing the path planning algorithm an analog approach was followed [8]. Indeed, such a realization was an initial motivation for the design of the algorithm itself given the general performance advantages of analog systems for processing algorithms sufficiently narrow in scope (e.g. high-frequency amplification as a simple form of signal processing). In [8] the approach was hard-coded to reduce area and took advantage of the algorithm's resistance to noise to reduce power consumption.

However this approach greatly constrained its ability to address a wider array of problems. Thus, despite the repetitive calculations required by the algorithm, enough potential for scaling exists to make an analog realization quite restrictive. A digital implementation,

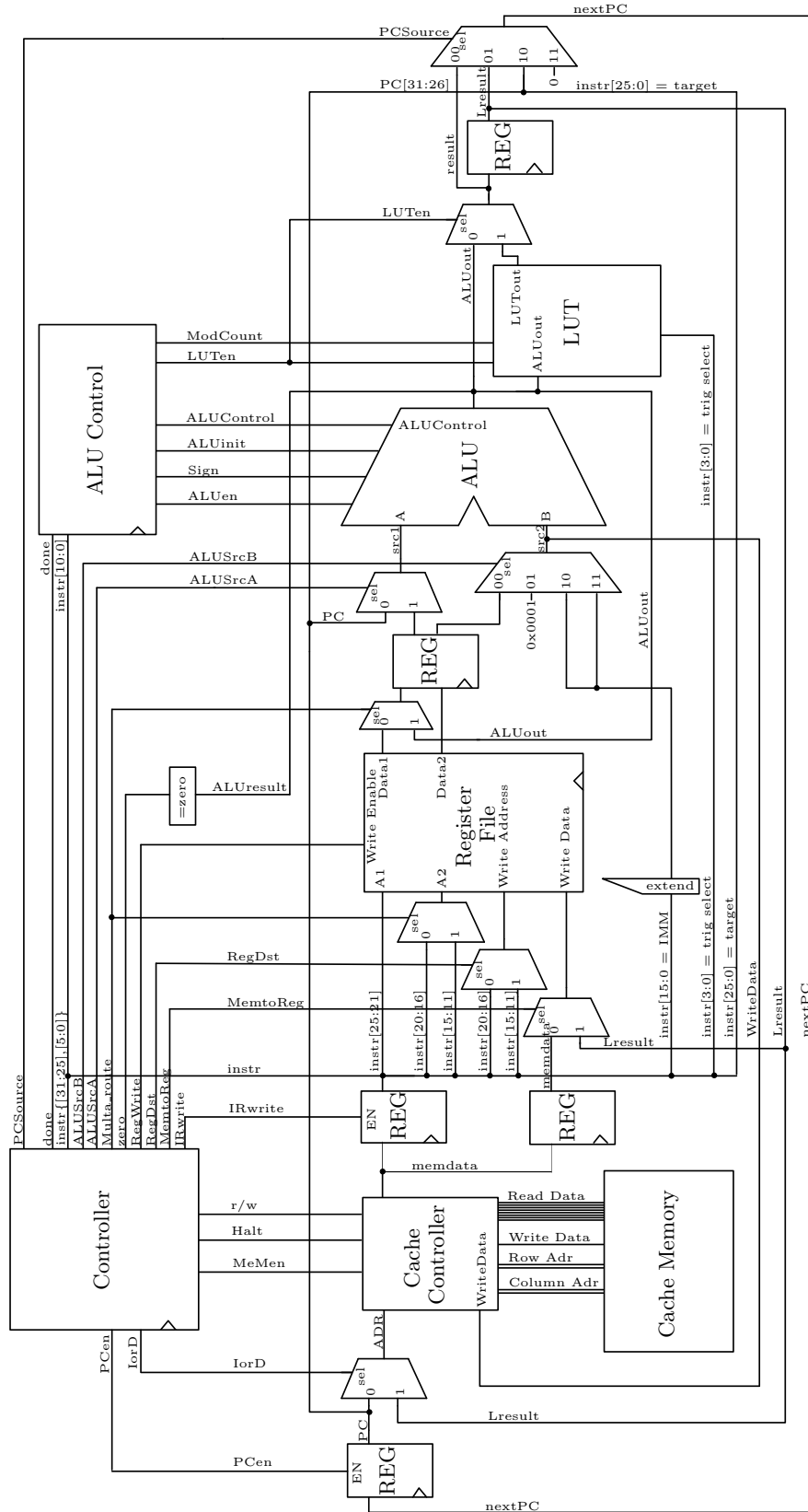


Figure 2.4: Schematic of the modified-MIPS processor design.

though not as small, can take the algorithm to its full potential in flexibility and expandability, and still achieve small size and lower power in comparison to general purpose processors.

2.2.2 Multicycle Processor

The modified-MIPS processor uses a multicycle microarchitecture (Fig. 3.1) to minimize space and complexity [4]. A multicycle microarchitecture refers to a processor arrangement wherein instruction execution is dissected into multiple steps depending on the instruction. This dissection is not used to achieve any form of parallelism as would be the case in a pipelined processor, but rather allows the computer to allocate an optimum number of cycles per instruction each of which are still executed without overlap.

While a pipelined system can boost the performance of the processor and allow higher clock speeds, such capability is not immediately necessary for running the path-planning algorithm under consideration. Since the algorithm is intended to operate directly on real-time sensor data that in general rarely encounters dynamics exceeding roughly 100 Hz, its 100 instruction execution cycle would seem as not too great a demand on computing resources, especially for nanoscale CMOS technologies. For these reasons this work chose to err on the side of simplicity and did not deeply consider a pipelined implementation. Still, it may be the case that a pipelined microarchitecture presents an even more efficient means of utilizing computing resource. This may especially be so given the lower incidence of control hazards for the Lyapunov-based planning approach.

A single cycle microarchitecture allows similar performance using a lower clock speed, but to complete each instruction in a single clock, much more hardware is required. In the case of this multicycle architecture, only one shared memory is used and only one ALU (arithmetic logic unit) is needed for all the calculations. This minimizes the size of the processor as well as its power consumption.

2.2.3 Word-Addressable Memory

Since most of the memory access consists of instruction fetches, where each instruction is a word, the processor uses word-addressable memory. This simplifies the memory access and expands the address space. There is a memory controller (discussed in greater detail in Chapter 3.2) where the word-addressable memory is translated to byte-addressable for greater compatibility with external memory and to reduce the number of pins on the chip.

2.2.4 Instruction Set Architecture

The modified-MIPS processor takes instructions and variables from a reprogrammable memory, like all general-purpose processors. As with the majority of processors today the design employs a three-address instruction scheme, that allows it to explicitly reference the locations of up to three operands in one instruction. This scheme is preferred by many compilers that employ a three-address intermediate representation and hence map very easily to a similarly structured instruction set [18].

Further, the design employs a register-to-register arrangement that only allows one memory address directly to main memory within an instruction (these being the *load* or a *store* instructions) and otherwise allows instructions to only access operands directly within a register file. For the obvious reason of reducing the number of (resource intensive) accesses to main memory this paradigm has been dominant for RISC architectures released since the 1980's. For the purpose of studying a design with maximum flexibility the 32-register file size common to standard MIPS has been retained in the modified-MIPS processor.

When main memory reads are required, to save power and boost performance, the processor reads instructions from the reprogrammable external memory and caches everything in its internal SRAM (Static Random Access Memory) upon first execution. Therefore, after the first loop of the algorithm, the processor would rarely, if ever, access the external memory. Once again, this ability to localize execution over many instructions is a fundamental

property of the underlying algorithm being implemented.

The full instruction set shown in Table 2.1 is modified from a standard MIPS processor. With more than 60% of the original instructions stripped away (i.e. 40 out of 63) the design underwent a significant alteration. This optimization saves instruction memory by reducing the instruction count by 57%. The reduction in instruction count not only reduces memory usage, it also makes the entire processor more efficient because it needs to fetch and decode less instructions.

The new architecture provides added support for 16 trigonometric functions which allow ready means of implementing spatial rotation calculations as well as the type of saturating and pulse functions utilized by the algorithm’s so-called sliding-mode control strategy (again, not detailed in this thesis) [19] to regulate the system based on gross variable measures (e.g. controlling the dynamics of state variables using sign values rather than actual magnitudes). As discussed in Chapter 3 the functions are computed using a look-up-table for which the *rs* setting denotes either a periodic function ($rs = \pi$) or an aperiodic one ($rs = 0$).

Naturally, with the load word (**lw**) and store word (**sw**) commands, the instruction set retains the ability to exchange information between the datapath registers and main memory. Besides enabling the feeding of instructions through the processor these commands are a critical component to scaling. They allow the storage of the state-variables in one layer of a multi-tiered control hierarchy as discussed in Chapter 1 while processing the calculations native to another layer. Such flexibility is a well known advantage of stored-program machines. In its present incarnation, the implemented processor allows for the storage of 256 variables in its cache.

To further the flexibility of the system, the custom architecture retained branch (**beq**) and jump (**j**) instructions. Such functions readily allow the algorithm to access alternate routines in the stack and hence give the controller the ability to efficiently operate over different levels of hierarchy running potentially alternate versions of the controller (e.g. systems

Table 2.1: Supported Instruction List for the Modified-MIPS Processor

R-type, (op = 000000)		
Instruction	Opcode/Function	Operation
sll	000000	$rd = rt \ll sa$
srl	000010	$rd = rt \gg sa$
sra	000011	$rd = rt \gg sa$
mfhi	010000	$rd = \{rs * rt\}[63 : 32]$ loads previous result, does not calculate
mflo	010010	$rd = \{rs * rt\}[31 : 0]$ loads previous result, does not calculate
mult	011000	$rd = \{rs * rt\}[52 : 21]$
multu	011001	$rd = \text{unsigned}\{rs * rt\}[52 : 21]$
multa	011100	$rd = rd + \{rs * rt\}[52 : 21]$
add	100000	$rd = rs + rt$
sub	100010	$rd = rt - rs$
and	100100	$rd = rs \& rt$
or	100101	$rd = rs rt$
slt	101010	$rd = (rs < rt)$
cos	110001	$rd = \cos(rt), rs = \pi$
cosh	110101	$rd = \cosh(rt), rs = 0$
ncos	111001	$rd = -\cos(rt), rs = \pi$
ncosh	111101	$rd = -\cosh(rt), rs = 0$
sin	110000	$rd = \sin(rt), rs = \pi$
sinh	110100	$rd = \sinh(rt), rs = 0$
nsin	111000	$rd = -\sin(rt), rs = \pi$
nsinh	111100	$rd = -\sinh(rt), rs = 0$
sec	110011	$rd = \sec(rt), rs = \pi$
sech	110111	$rd = \text{sech}(rt), rs = 0$
nsec	111011	$rd = -\sec(rt), rs = \pi$
nsech	111111	$rd = -\text{sech}(rt), rs = 0$
csc	110010	$rd = \csc(rt), rs = \pi$
csch	110110	$rd = \text{csch}(rt), rs = 0$
ncsc	111010	$rd = -\csc(rt), rs = \pi$
ncsch	111110	$rd = -\text{csch}(rt), rs = 0$
I-type, (op != 000000, 00001x, 0100xx)		
addi	001000	$rt = rs + \text{immediate}$
andi	001100	$rt = rs \& \text{immediate}$
ori	001101	$rt = rs \text{immediate}$
slti	001010	$rt = (rs < \text{immediate})$
beq	000100	$\text{if}(rs = rt), PC + \text{immediate}$
lw	100011	$rt = \text{memory}(rs + \text{immediate})$
sw	101011	$\text{memory}(rs + \text{immediate}) = rt$
J-type, (op != 00001x)		
j	000010	$PC = \text{immediate}$

derived using an alternate Lyapunov candidate, or differing gain functions in a sliding-mode calculation or even employing entirely new control strategies). Should a multi-level implementation of the algorithm be sought, this is an obvious means of supporting such requirements.

At the same time, the custom implementation removed some convenient, but not essential MIPS instructions related to procedure calls and routines. These would be the jump and link (**jal**) and jump register (**jr**) commands. Their absence requires the programmer to explicitly specify and store addresses. It is precisely the minimal dependence on conditionals of the planning algorithm in question that minimizes the inconvenience associated with the removal of these instructions.

In addition to the memory, conditional, and non-linear trigonometric function support special instructions for multiplication operations was included. In particular the **multa** (multiply-and-accumulate) function was added to the instruction set and the **mult** (multiply) functions was modified. This was done because the algorithm depends heavily on multiply and add instructions as a result of the matrix multiplications it regularly invokes.

The **mult** function was modified such that it automatically shifts and saves the result into a destination register, and the **multa** function shifts and adds the result to the destination register. In the MIPS32 instruction set architecture [14], the multiply and multiply-accumulate functions require additional instructions to fetch the resultant higher-order and lower-order product bits and manually shift the result to the correct place followed by saving or adding the result into some user selected destination register of the register file. These extra steps are a result of the fact that a separate multiplication unit, outside the pipelined datapath, is assumed in the MIPS32 architecture [15]. As such, the standard MIPS architecture allows multiplication functions to be carried out in parallel with other instructions being fed through the pipelined datapath. However, this requires extra instructions to seamlessly reintegrate the result of the multiplication, which is effectively asynchronous with the central

processor, into the main datapath.

No such limitations need be imposed on the customized processor under consideration here, especially since any issues with hazards or interlocks are avoided by foregoing a pipelined approach for the multicycle implementation. Thus, completely self-contained multiplication functions have been implemented for the planning processor. As a result, every time the `mult` instruction is used, only one instruction is executed (instead of a sequence of five as required by the standard MIPS32 architecture), and every time the `multa` instruction is used, only one instruction is executed (instead of six).

Of course standard arithmetic, logical, and shift functions are retained as well. These include addition (`add`), subtraction (`sub`), logical and (`and`), logical or (`or`), set on less than (`slt`), shift left logical (`sll`, left shifts that fill the least significant bits with zero), shift right logical (`srl`, right shifts that fill the most significant bits with zero) and shift right arithmetic (`sra`, right shifts that preserve the sign bit).

In summary, with the above modifications, simulations indicate that the processor can be expected to operate at up to 1.25 million-instructions-per-second (MIPS) from a 400-mV supply and up to 12 MIPS from a 1-V supply. Fig. 2.5 graphs the data for two versions of the operation, one where instructions are fetched from the cache and one where they are fetched from external memory. Nine extra clock cycles are needed for an external instruction fetch obviously reducing the computational performance of the processor in this case.

The MIPS performance numbers of Fig. 2.5 were extracted based on the worst case delay through the processor. In this design, the worst case delay was that experienced by the progress of the multiplication functions through the ALU. Using the Synopsys[®] Design Compiler, a digital circuit synthesizer, it was determined that the delay through the ALU was equivalent to 132 fanout-of-4 (FO4) inverter delays. This effectively determines the minimum clock period (in a single-edge clocking scheme) required to process the instructions inside the processor. Alternatively, it is a statement on the maximum clock frequency sustainable

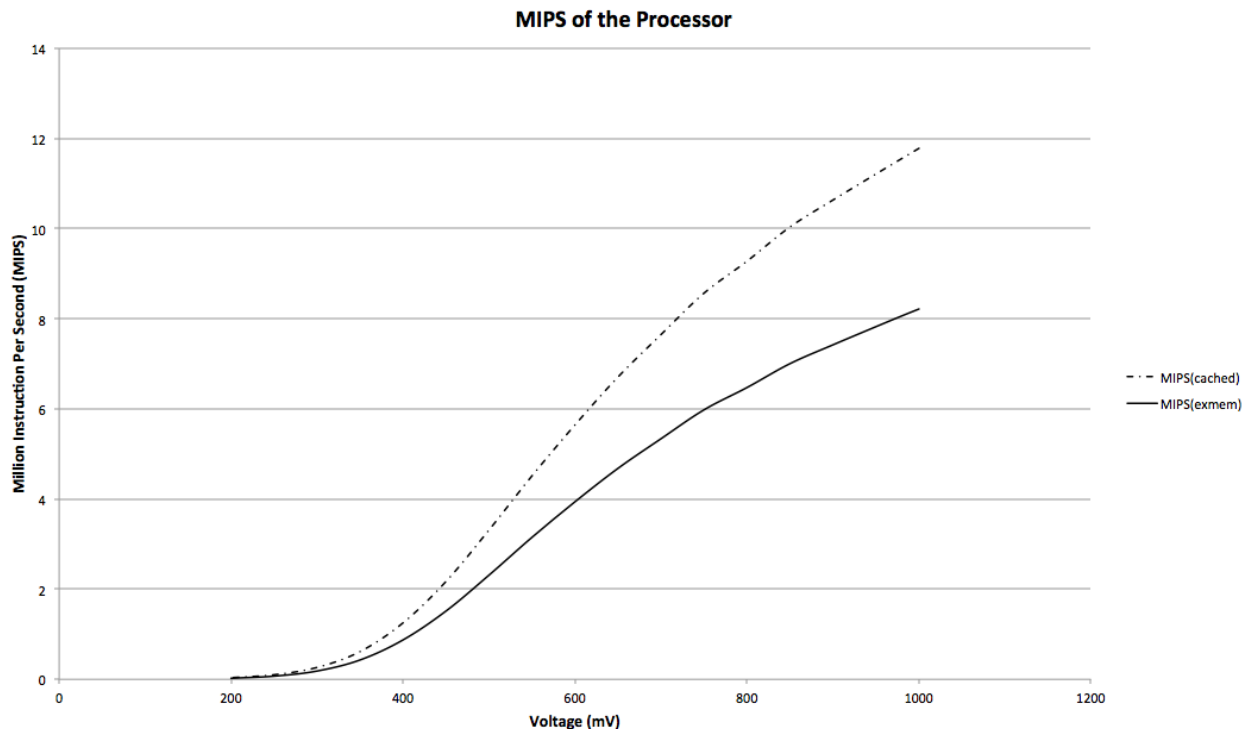


Figure 2.5: Expected processor performance at different voltage levels. The dashed and solid curves refer to instruction fetching from the cache and external memory respectively.

by the design.

The ALU delay was extracted using the 90-nm CMOS standard cell library available from the Taiwan Semiconductor Manufacturing Corporation (TSMC). One of the silicon versions of this design was ultimately fabricated in this CMOS technology. The actual inverter FO4 delay as a function of the supply voltage, V_{DD} , was determined using the Spectre circuit simulator available from Cadence[®]. At $V_{DD} = 400$ mV the FO4 delay was about 180 ps (implying a maximum clock frequency of 42 MHz). These results are discussed further in the following section 2.3.

With a maximum clocking frequency established it only remained to calculate the average number of cycles per instruction (CPI) to attain a MIPS performance estimate. The CPI is calculated by finding the weighted average of the clock cycles needed to complete an iteration of the Lyapunov algorithm. The weighting is based on the type of instruction and the number of times that instruction occurs during an iteration. For this design the average

CPI was 19.3 for the cache-based operation and 27.7 for a processor operating from external memory. Finally, dividing the processor’s operational clocking frequency of 100 kHz by the CPI led to the approximate MIPS performance curves shown in Fig. 2.5.

2.3 Sub-threshold Transistors

Due to its suitability for low-power digital operation (i.e. high density of interconnect and relatively low parasitic leakage current) a complementary metal-oxide-semiconductor (CMOS) technology was the obvious choice for processor implementation. Further, running the CMOS devices at voltages below the device’s threshold voltage (i.e. the voltage at which a fully inverted channel is formed under the gate of the transistor and the MOSFET switch is generally considered “closed”) can help achieve even lower power operation [20].

MOS transistors made to operate in such a fashion are noted as operating in the “sub-threshold” regime. It is a technique that was first indirectly employed in wristwatch applications over 40 years ago where attempts to operate MOS transistors as “normally” as possible under low supply voltage effectively implemented sub-threshold operation [21, 22]. The technique was more clearly articulated shortly thereafter by Swanson and Meindl [23, 24] and has been steadily but slowly gaining more relevance in general purpose computing over the last 20 years [25, 26]. Devices made to operate in this mode are also referred to as being in weak inversion, a reference to the relatively small amount of mobile charge introduced for the purpose of current conduction through the transistor. Devices operating above threshold are said to be strongly inverted.

Although they can achieve substantially lower operating power levels, sub-threshold circuits suffer from long delays relative to their strongly inverted counterparts (because they conduct less current and hence require more time to charge up a given capacitance to a given voltage) and are sensitive to noise. In Fig. 2.6, Spectre simulations of inverter delay as a function of supply voltage (V_{DD}) make it clear that voltages below 300 mV increase the

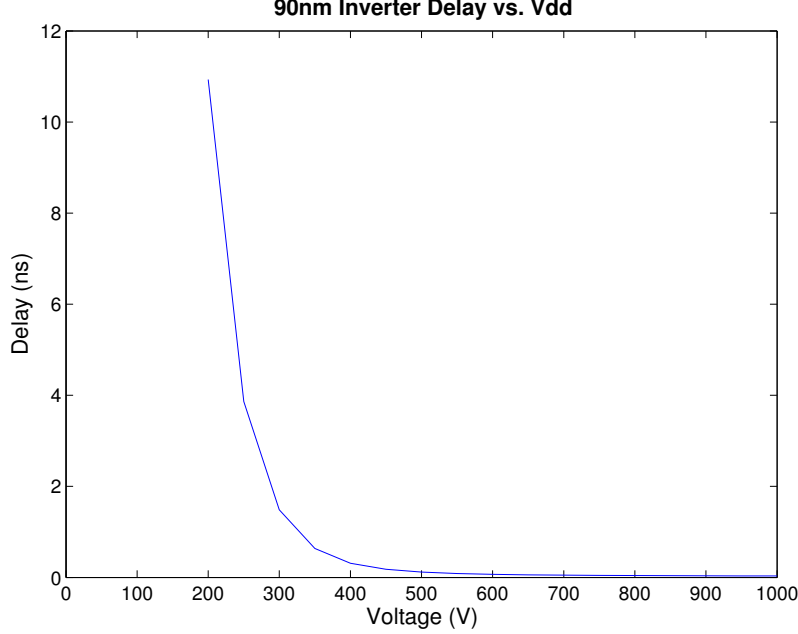


Figure 2.6: Voltage scaling delay in TSMC’s 90-nm standard cell digital library inverter circuit (FO = 4).

delay exponentially. Fig. 2.7 is the same graph in log scale, where the extreme values are shown more clearly. Generally speaking, the ideal operating voltage for low power and speed balance (i.e. minimum energy per operation) is determined by an ideal balance of leakage and dynamic power as discussed below [27].

In [28], contemporary viewpoints on the ideal operating voltage for digital CMOS circuits are discussed. Regular, above-threshold, operating voltages are wasteful and consume high active dynamic power, P_{ACT} . The dynamic power refers to the power lost in charging and discharging digital circuits during information transfer from one block to the next. For CMOS this is generally expressed with

$$P_{ACT} = \alpha \cdot C \cdot V_{DD}^2 \cdot f_{clk} \quad (2.1)$$

where C refers to the capacitance being charged and discharged by a digital circuit, V_{DD} is the supply voltage of the digital circuit (and hence the maximum voltage to which the capacitive load is charged), f_{clk} is the underlying clock frequency to which information flow

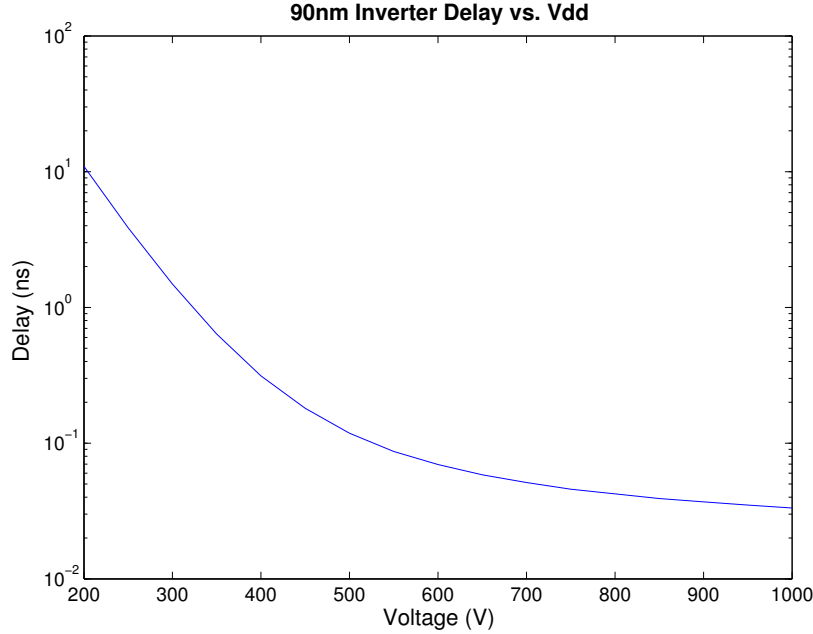


Figure 2.7: Voltage scaling delay in TSMC’s 90-nm standard cell digital library inverter circuit (FO = 4).

through the digital system is subject and α is the proportion of the clock’s transitions during which the digital circuit is actually made to charge a capacitance.

As V_{DD} and f_{clk} are lowered P_{ACT} can obviously drop substantially. As the dynamic power drops, another loss component, the static leakage power, P_{LEAK} , starts to become an important power contributor to account for. P_{LEAK} is simply the power consumed by a digital circuit when it is not switching. Ideally it should be zero for CMOS, but current, I_{LEAK} , leaks through the circuit from V_{DD} to ground and hence induces the power drain

$$P_{LEAK} = I_{LEAK} \cdot V_{DD}. \quad (2.2)$$

As V_{DD} is reduced therefore increasing delay and reducing f_{clk} the amount of time needed to complete a whole calculation increases, thus draining more energy due to leakage in the interim. As already shown in Fig. 2.6 the delay of a digital circuit increases exponentially as V_{DD} drops performance deep into sub-threshold.

Fig. 2.8 clearly demonstrates the tradeoff between the active and static consumption

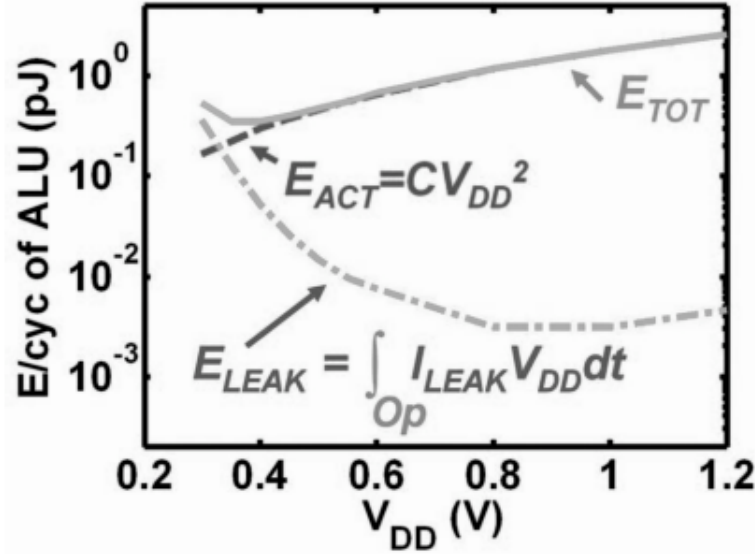


Figure 2.8: Dynamic and static energy/cycle tradeoff in a 65-nm CMOS ALU [28].

elements, by sketching the average energy consumed per cycle of a 65-nm CMOS ALU as a function of V_{DD} . As shown therein, an optimum energy per cycle performance is reached at $V_{DD} \approx 400$ mV.

Fig. 2.9 further shows the minimum energy point in different technologies. It is apparent that this *minimum energy point* exists in different technologies around similar voltages, and most papers in low power digital circuits or SRAM designs operate in this voltage range. Since the path-planning algorithm has a very low computing performance requirement, the operating voltage can be reduced to the *minimum energy point* to reduce the power consumption by an order of magnitude.

In light of these studies plus the desire to implement an optimal power solution, a minimum supply of 400-mV was targeted for the processor in question and hence near-threshold operation adopted as a suitable design choice.

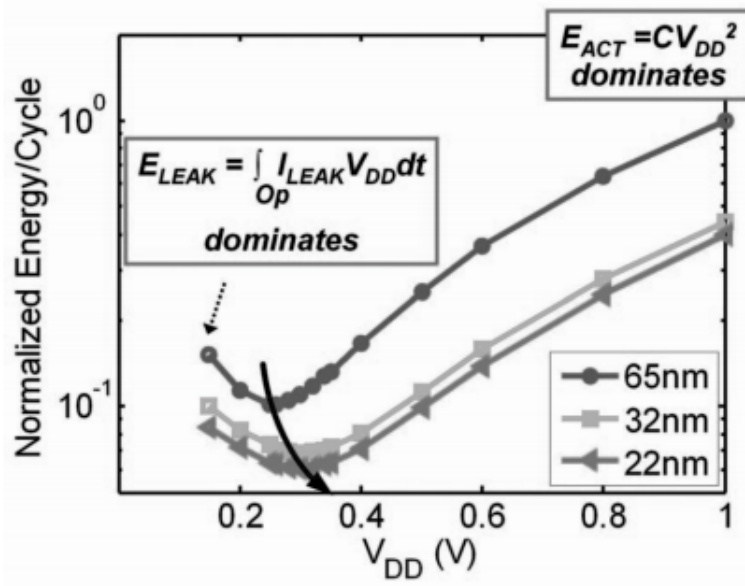


Figure 2.9: Total energy/cycle consumption as a function of V_{DD} for various CMOS technology generations [28].

Chapter 3

PROCESSOR & MEMORIES

The modified-MIPS processor was built according to methods discussed in Chapter 2. In the sections below, the implementation of the processor, the memory, specialized command implementation, analog components and test are discussed in further detail. In Figure 3.1, the schematic of the processor is shown without the test circuits, analog components, and external connections. This illustrated the relation between the core components, and how they interact. The full Verilog code for this work is included in Appendix D for reference.

3.1 Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) was designed to be small and power efficient. To achieve this, the ALU was made using a minimum number of components while retaining all the required functionality. Multiplications and additions are both done serially, using a 32-bit ripple adder, a 32-bit register (for the multiplier operand) and a 64-bit loop shift register (also called a rotator and used for the multiplicand and product operands). The direct integration of the multiplication function with the processor's datapath is a break from the traditional MIPS arrangement which does not merge this block directly with the main pipeline. The reason for this choice in the standard MIPS design was due to the excessive interruption that such a long-delay unit would cause to a pipeline, a problem not encountered in the multi-cycle design under consideration here [15]. Together with some combinational logic and multiplexers, the ALU is capable of executing: addition, subtraction, signed and unsigned multiplication, logical shifts and arithmetic shifts, modulus, AND, OR, and SLT.

The ALU controller (i.e. the *ALU Control* block in Fig. 3.1) has also been heavily modified from the simple combinational decoder considered in typical MIPS systems [4, 16] to a custom

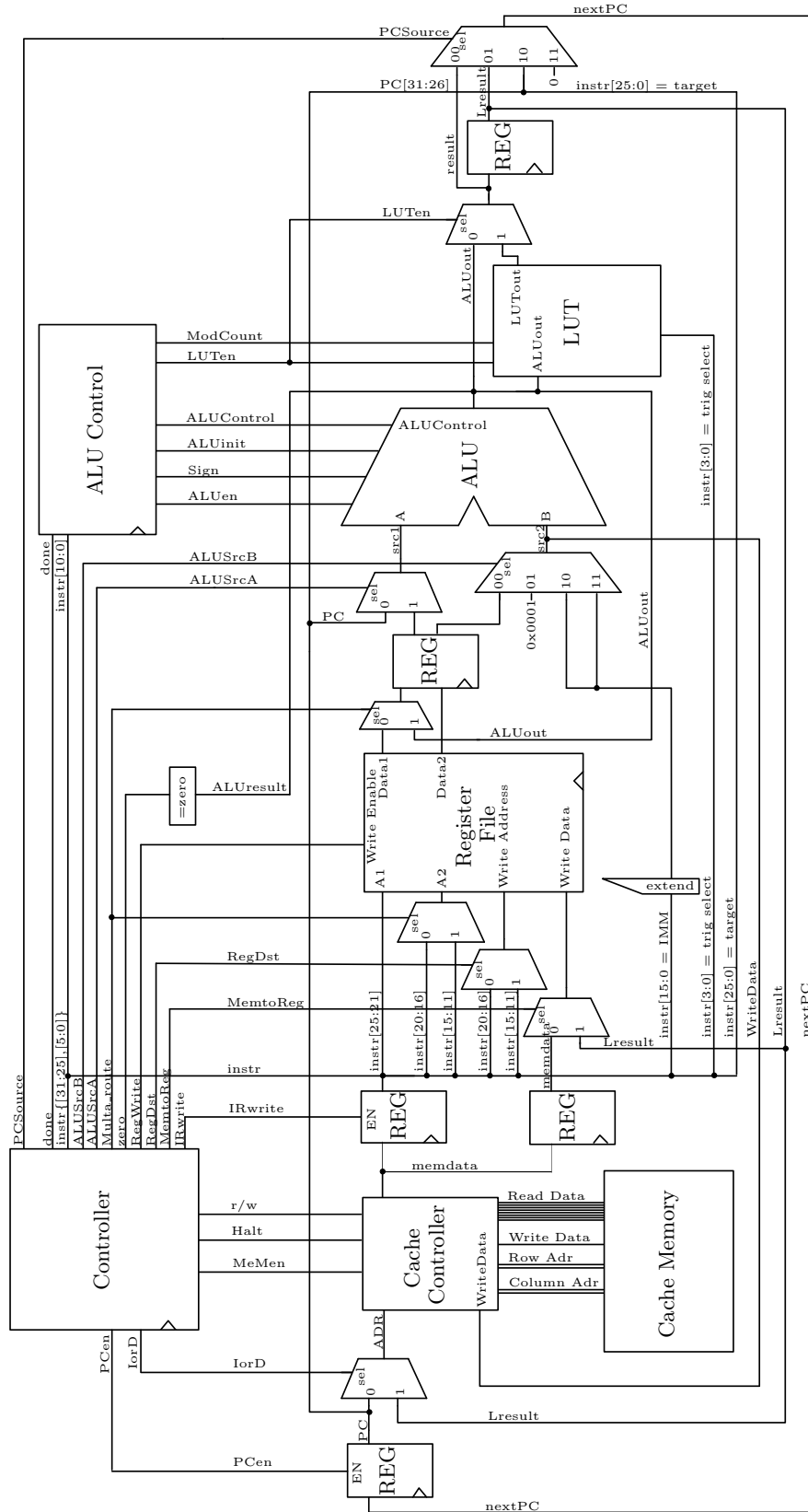


Figure 3.1: Schematic of the modified-MIPS processor design.

state-machine. This allows components in the ALU to be reused for different purposes. The ALU Control generates a 4-bit signal that gives the ALU its broad functionalities by activating different parts of the ALU or by sequencing through a series of different functions (e.g. multiplication requires three different operations to complete).

Fig. 3.2 illustrates the schematic of the ALU and the control signals needed for each function. The ALU Control works like a second stage instruction decode (following the core *Controller* in Fig. 3.1) where, in the initial state, it decodes all the instructions and initiates the multi-cycle instruction processing through the datapath’s combinational logic.

In a standard single-cycle calculation, the processor would take the result from the ALU and proceed to subsequent actions without interruption. In a multi-cycle calculation, the ALU Control block cycles the ALU through the appropriate sequence of calculations, while pausing all other processor datapath operations until the calculation is complete. The full Verilog implementations of the ALU Control and the ALU itself are given in Appendix C.2 and are contained in modules *alucontrol* and *alu*.

The significance of this ALU is that multiplication support only requires the inclusion of a 64-bit register within the datapath, and that it can carry out both signed and unsigned multiplication. The inclusion of such a register is not in itself a compromise relative to the standard MIPS architecture, which effectively accommodated such a structure by the inclusion of two extra 32-bit registers for storing the multiplier’s product [14].

Although signed multiplication could be accomplished using a series of comparisons, inversions and unsigned multiplications, native (i.e. direct) hardware support for such a sequence of operations is much more difficult. Eq. (3.1) shows the arithmetic for unsigned multiplications. In comparison, the modified Baugh-Wooley signed multiplication arithmetic shown in (3.2) includes the irregularities of additional constants and some inversions [29]. The ALU Control block routes input signals and intermediate signals through different existing components (shifter, inverters, multiplexers, etc.) to accomplish this intricate task, a clear

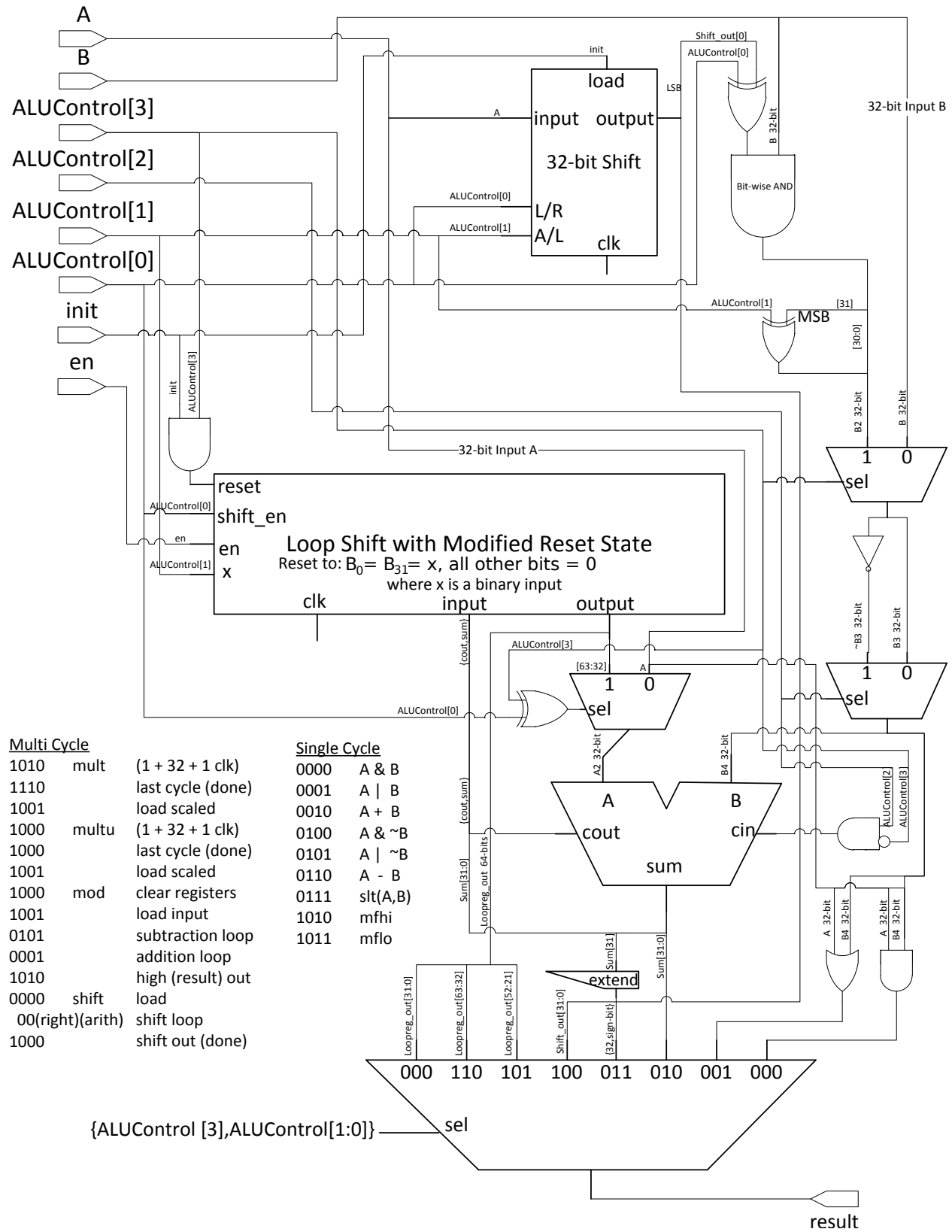


Figure 3.2: Schematic of the arithmetic logic unit.

motivation for a state-machine realization of such a block as discussed above. The 32-bit ripple adder and the 64-bit register then work together as an accumulator to complete the serial multiplication.

$$\begin{array}{r}
\begin{array}{cccccccc}
& & & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 \\
\times & & & b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0
\end{array} \\
\hline
& & & a_0b_7 & a_0b_6 & a_0b_5 & a_0b_4 & a_0b_3 & a_0b_2 & a_0b_1 & a_0b_0 \\
& & a_1b_7 & a_1b_6 & a_1b_5 & a_1b_4 & a_1b_3 & a_1b_2 & a_1b_1 & a_1b_0 & \\
& a_2b_7 & a_2b_6 & a_2b_5 & a_2b_4 & a_2b_3 & a_2b_2 & a_2b_1 & a_2b_0 & & \\
& & a_3b_7 & a_3b_6 & a_3b_5 & a_3b_4 & a_3b_3 & a_3b_2 & a_3b_1 & a_3b_0 & \\
& & & a_4b_7 & a_4b_6 & a_4b_5 & a_4b_4 & a_4b_3 & a_4b_2 & a_4b_1 & a_4b_0 \\
& & & & a_5b_7 & a_5b_6 & a_5b_5 & a_5b_4 & a_5b_3 & a_5b_2 & a_5b_1 & a_5b_0 \\
& & & & & a_6b_7 & a_6b_6 & a_6b_5 & a_6b_4 & a_6b_3 & a_6b_2 & a_6b_1 & a_6b_0 \\
& & & & & & a_7b_7 & a_7b_6 & a_7b_5 & a_7b_4 & a_7b_3 & a_7b_2 & a_7b_1 & a_7b_0
\end{array}
\tag{3.1}$$

$$\begin{array}{r}
\begin{array}{cccccccc}
& & & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 \\
\times & & & b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0
\end{array} \\
\hline
& & & 1 & \overline{a_0b_7} & a_0b_6 & a_0b_5 & a_0b_4 & a_0b_3 & a_0b_2 & a_0b_1 & a_0b_0 \\
& & \overline{a_1b_7} & a_1b_6 & a_1b_5 & a_1b_4 & a_1b_3 & a_1b_2 & a_1b_1 & a_1b_0 & & \\
& & & \overline{a_2b_7} & a_2b_6 & a_2b_5 & a_2b_4 & a_2b_3 & a_2b_2 & a_2b_1 & a_2b_0 & \\
& & & & \overline{a_3b_7} & a_3b_6 & a_3b_5 & a_3b_4 & a_3b_3 & a_3b_2 & a_3b_1 & a_3b_0 \\
& & & & & \overline{a_4b_7} & a_4b_6 & a_4b_5 & a_4b_4 & a_4b_3 & a_4b_2 & a_4b_1 & a_4b_0 \\
& & & & & & \overline{a_5b_7} & a_5b_6 & a_5b_5 & a_5b_4 & a_5b_3 & a_5b_2 & a_5b_1 & a_5b_0 \\
& & & & & & & \overline{a_6b_7} & a_6b_6 & a_6b_5 & a_6b_4 & a_6b_3 & a_6b_2 & a_6b_1 & a_6b_0 \\
& & & & & & & & 1 & \overline{a_7b_7} & \overline{a_7b_6} & \overline{a_7b_5} & \overline{a_7b_4} & \overline{a_7b_3} & \overline{a_7b_2} & \overline{a_7b_1} & \overline{a_7b_0}
\end{array}
\tag{3.2}$$

In greater detail, in the first stage of the multiplication sequence, the 64-bit register is cleared and initialized to 0x0000 0000 0000 0000 for unsigned or 0x0000 0000 8000 0001 for signed multiplication. This accounts for the two constants that will later appear in partial product accumulations [i.e. the 1s in the partial product sequence of (3.2)]. The 64-bit register is a loop shift register (i.e. a rotator) that rotates right, incrementally pushing out the multiplicand (to the right) as it constructs the product bit-by-bit (in the 64-bit loop shift register). Therefore, the adder has continual access to the 32 most significant bits of the intermediate products as input and, after 32 calculations, results in the final product are correctly positioned within the 64-bit register. Constants are loaded automatically when the calculation loops around to the most significant bit (MSB). Using the loop shift register eliminates the need for a huge 32-bit by 32-input multiplexer.

Also, in the top-right corner of Fig. 3.2, input A gets loaded into the 32-bit shifter (that also doubles as a means of realizing arithmetic and logical shift operations), where input A is shifted right, and the LSB of the output will sequence from A_{LSB} to A_{MSB} for each step of the multiplication. With the addition of some small logic, multiplexers and the existing 32-bit inverter, the ALU controller is capable of making appropriate inversions when needed for signed multiplications.

3.2 Static Random Access Memory (SRAM)

Many research papers have explored implementations of sub-threshold SRAM cells. Many of these, however, have not been implemented into a usable memory block complete with control circuits, and lack consideration of the control circuit's complexity. Reports with fully realized memory structures that seemed suitable for integration into the modified-MIPS processor included [30, 31, 32]. The method described in [32] was adopted for this work because of its promising result and complete design. It offers better read and write margins because of the

virtual V_{DD} (VV_{DD}) and the pull-up transistor in the read (output) circuit.

Figure 3.3 shows the schematic for the single-bit 10T (10 transistor) SRAM cell. It consists of a double inverter loop (transistors M1-M4) for storing the information whose power supply connection, VV_{DD} , is made to float briefly during write operations. Disengaging the supply in this way prevents the PMOS pull-up transistors in the inverter loop from opposing a write through the access transistors (M5 and M6) which conduct write signals through the bit lines (BL) when they (the access transistors) are engaged by a high word line signal, WL . This greatly improves the *write margin* (i.e. the minimum amount of noise voltage required to prevent an access transistor from overpowering a pull-up transistor) of the cell because the write signal can drive over the stored signal (now with weak floating voltage) more easily.

The reader will notice that the read network (transistors M7-M10) employs a separate, isolated (via M7-M9), read bit line rBL . This buffer circuit is necessary to prevent the possibility of the read signal overpowering the cell's contents as maintained by M1-M4, especially at low operating voltage levels.

In the cell's read circuitry, both a pull-up (M7) and a pull-down transistor (M8) are used although only M8 serves to implement the reading operation as rBL is pre-charged (thus negating the effect of a pull-up). However M7 does serve to reduce leakage, an important consideration for sub-threshold operation. When the gate voltage of M7 is low the transistor charges up the source of M10, reducing its leakage current and hence load on rBL . Even when the gate voltage of M7 is high, the leakage through it is prevented from sinking to ground by M9 and hence tends to raise the source of M10 again attenuating its leakage load on rBL .

Next, Fig. 3.4 shows the schematic of a *word line control* that controls the VV_{DD} in response to a word-line input $WLin$. The circuit simply implements an OR logic function that only engages the floating supply through an activation PMOS transistor when both



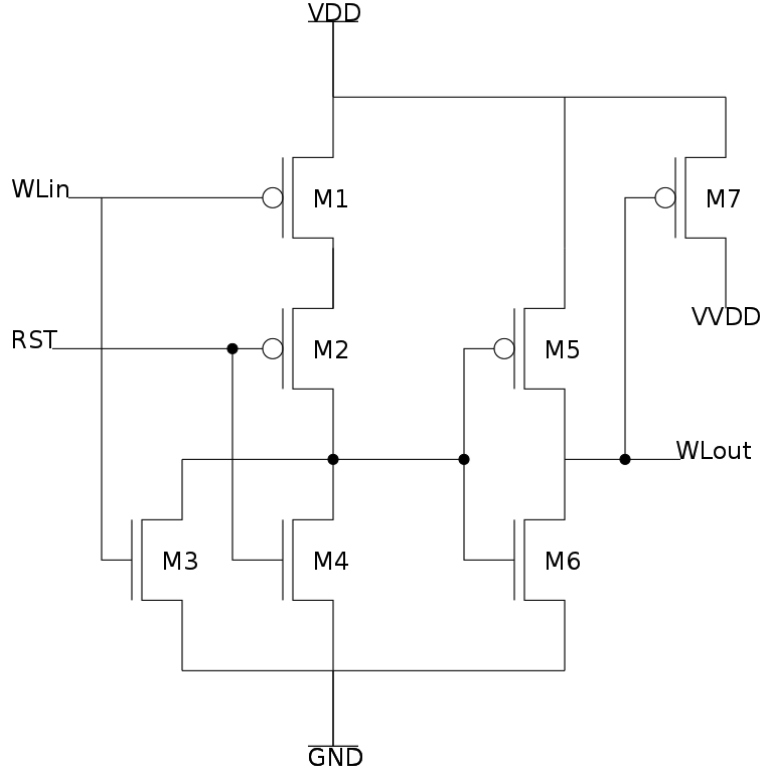


Figure 3.4: Word line control circuit.

$WLin$ and RST are low. During this setting the SRAM cell obviously retains the stored data. When $WLin$ goes high, $WLout$ follows thus turning on the access transistors for a write while simultaneously disconnecting the SRAM cell from V_{DD} (i.e. V_{VDD} goes low). The RST pin is intended to allow the SRAM to be initialized to a known state.

Fig. 3.5 shows the block diagram for a single SRAM word-line, with a representation of the word-line controller given by the block on the left and a 36-cell instantiation of the SRAM cell represented by the block on the right (`bit<35:0>`). The word-line consists of the stored 32-bit word as well as 4 extra condition bits (described below).

Thirty-two words are then cascaded together in rows to produce a 32-word block that connects to a memory I/O controller (discussed below) that regulates the connection between the processor and activities in the SRAM. The schematic of the signal conditioning circuitry between the memory controller and the core SRAM circuitry is shown in Fig. 3.6.

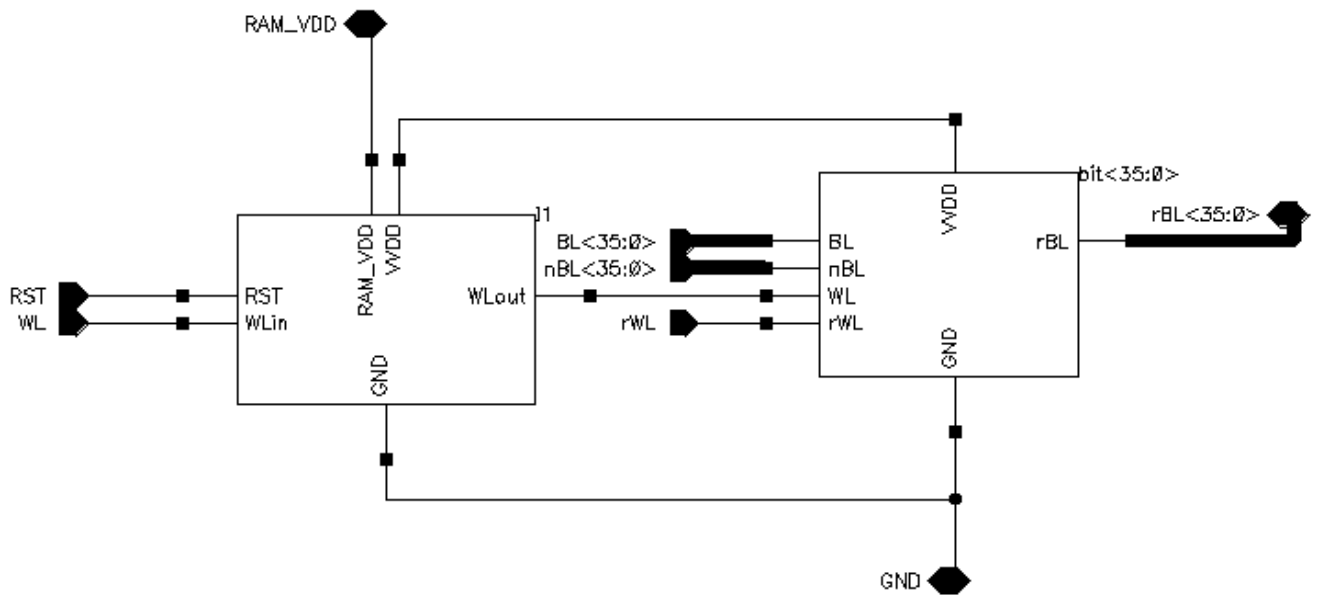


Figure 3.5: 36-bit word line arrangement.

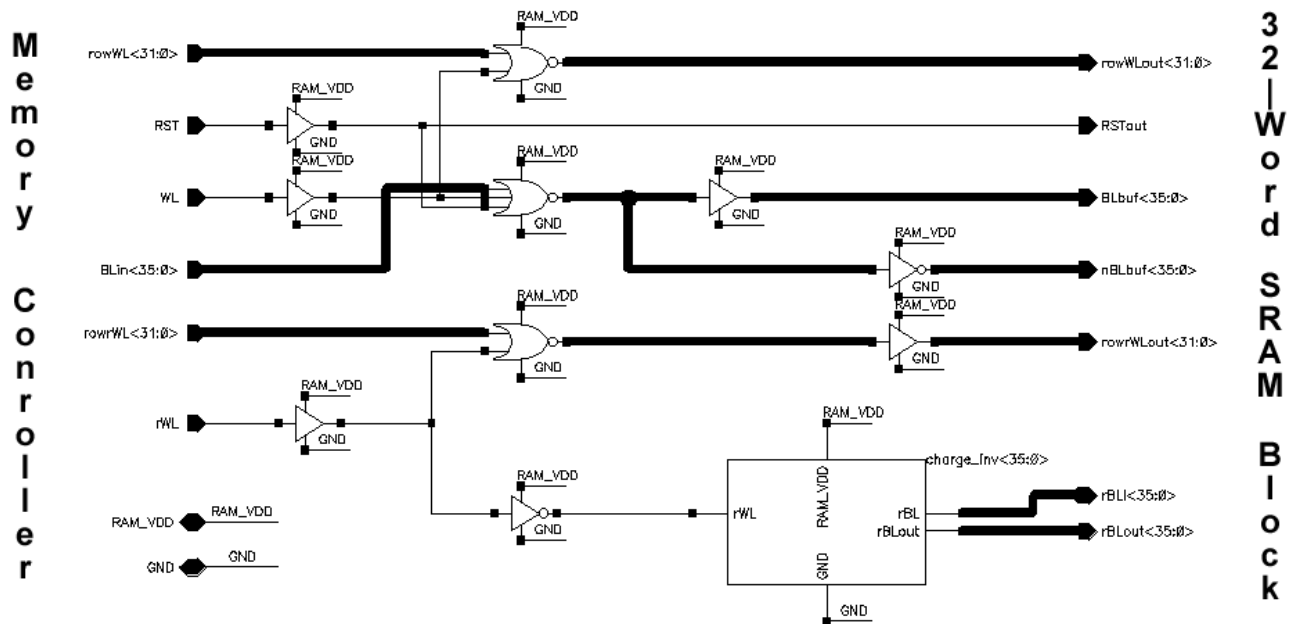


Figure 3.6: 32-word block signal conditioner.

The signal conditioning circuitry consists of buffers, line sense, line pre-charge, and block enable signals. The *rowWLout* line activates the word to be written to in the memory block (i.e. it turns on the access transistors M5 and M6 of Fig. 3.3 in each SRAM cell of the word of interest), while the *rowrWLout* activates the read circuitry (i.e. the *rWL* line in Fig. 3.3) of each cell. The write bit lines are pre-charged with set with *BLbuf* and *nBLbuf* while the read bit line is precharged with *rBLout*.

Fig. 3.7 shows a schematic of the SRAM memory arrangement. It is folded into 8 columns (although the schematic in Fig. 3.7 arranges the 32-word memory blocks in a row format, these constituents are physically arranged as columns in the chip layout) of 32-word blocks. A column is selected for write with the signal *colWL* and a column is selected for read with *colrWL*. The signal's *rowWL* and *rowrWL* engage the write and read word lines, respectively, in the selected columns (i.e. the signals are shared between the 8 memory blocks). The signal *BL* provides the command to pre-charge the write bit lines in the selected column.

The 1-kB SRAM (32 bit words \times 32 rows \times 8 columns) consumes a significant amount of chip space due to the high transistor count and the large amount of wiring. Therefore consideration of the physical layout was a very important part in designing the SRAM.

The most important layout consideration was how each component would be connected to each other, preferably by grid placement of the component rather than individual wiring. The dimensions of each cell, the placement of each transistor, direction of each wire and spacing of metal rails were all important contributing factors.

Fig. 3.8 shows the hand layout settled on for a 10T SRAM cell where by simply placing cells side by side, top to top and bottom to bottom, results in an elegantly designed SRAM array that has minimum spacing between cells and requires no additional internal wiring. The dimensions of a single 10T cell are $2.6 \times 2.635 \mu\text{m}^2$.

Similarly, the signal conditioner for each 32-word SRAM block must be pitch

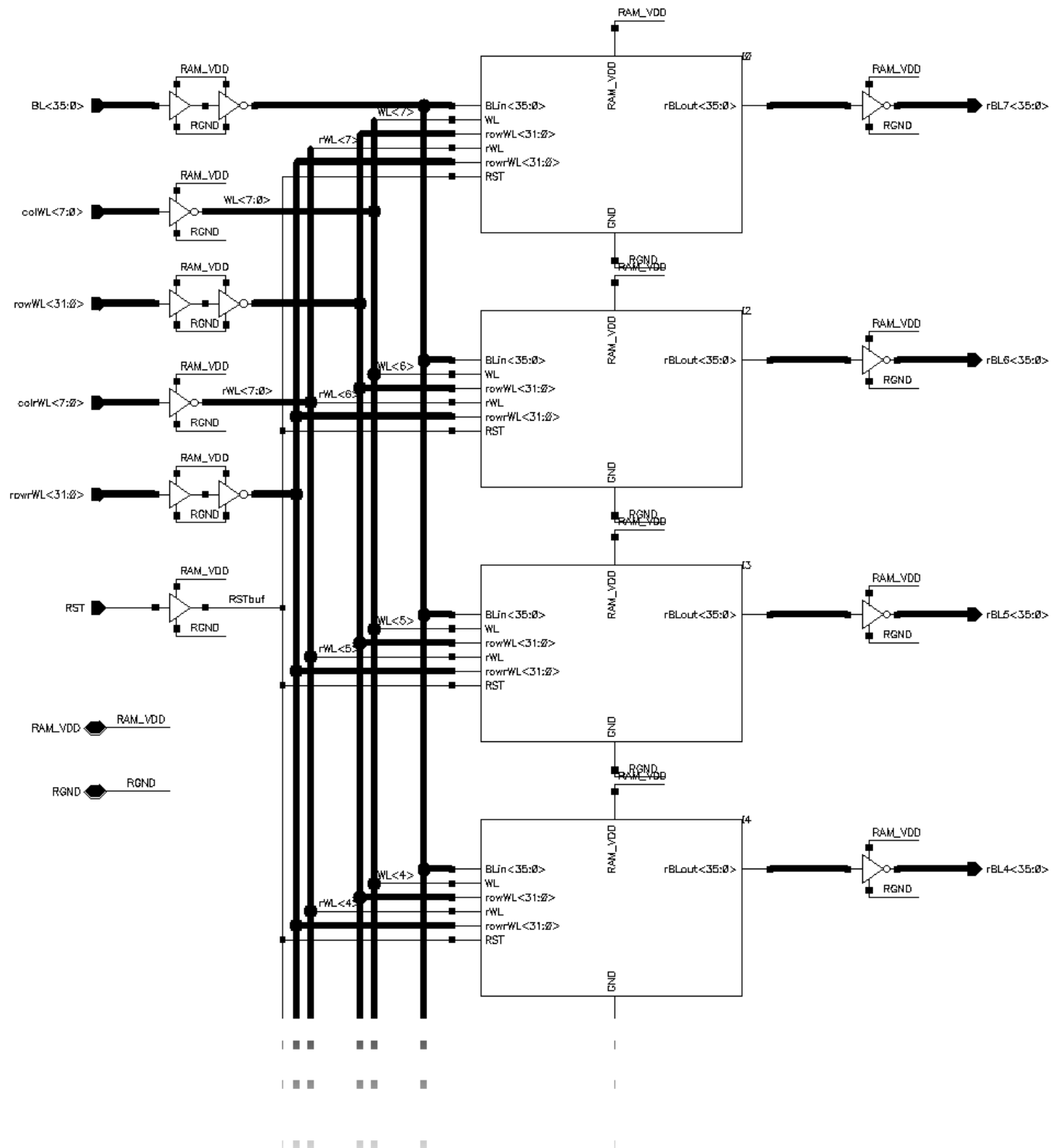


Figure 3.7: Schematic of the Full 1-kByte SRAM.

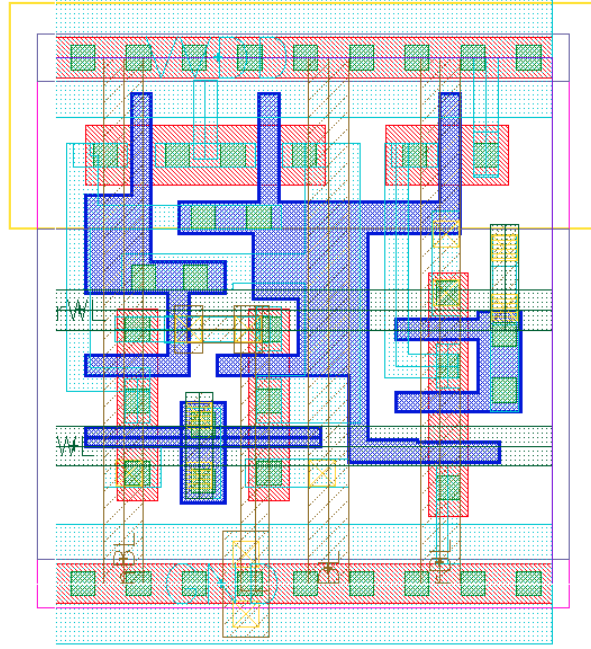


Figure 3.8: Layout of the 10T SRAM cell using TSMC 90-nm technology.

matched to the core memory as well for an effective on-chip arrangement. Relative to the SRAM call, the signal conditioner contains many more signals that must be wired and uses a broader variety of different digital standard cells. Adequately laying out this component, proved to be a more challenging task because of the varying cell widths of the myriad constituent circuits used and the large amount of signals being funnelled through the signal conditioner.

Fig. 3.9 shows the layout of the signal conditioning circuit (which would be placed at the bottom of an SRAM block). The design dimensions are $10.08 \times 100.86 \mu\text{m}^2$. The long horizontal row shown in Fig. 3.9 consists of four rows of standard cell digital circuits each of which span the whole $100.86 \mu\text{m}$ length of the design. The relative location of key signalling ports in the signal conditioner are labelled in Fig. 3.9.

Along the left side of each 32-word SRAM block, there is a column of 32 word line control circuits (see Fig. 3.4 for a schematic of each control circuit). Fig. 3.10 shows all of the control circuits around the SRAM block together with the signal conditioner, word line control and

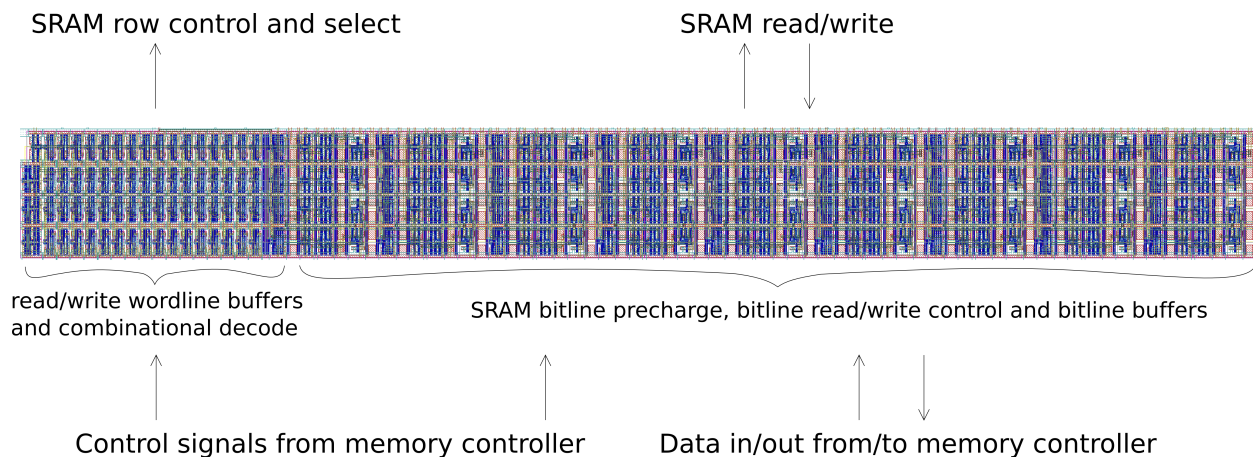


Figure 3.9: WL and rWL Buffers (left 1/4); BL rBL read/write logic (right 3/4), in TSMC 90-nm CMOS.

wiring.

Finally, Fig. 3.11 shows the complete 32-word SRAM block, and Fig. 3.12 shows the entire SRAM built with 8 32-word blocks measuring $845 \times 141 \mu\text{m}^2$, at density of $465 \mu\text{m}^2/\text{word}$.

To verify this final design, SPICE simulations that read and write to random SRAM blocks were used to make sure that the circuitry behaves as expected. Also, a Verilog netlist was extracted and a full verilog simulation was run with the memory controller and the processor. (see Appendix C.6 for the Verilog netlist)

3.3 Memory System

The memory architecture design is driven by considerations of the MIPS processor, SRAM, and algorithm's expandability. Between the MIPS processor, SRAM and external memory (or memories) and/or I/O(s), is the cache controller (see Fig. 3.1). After all the memory components and operations are defined, the cache controller connects them together and lets them operate independently. From the processor's perspective, the cache controller and all the memory elements and memory-mapped I/O connected to it are seen as a single unit. Simultaneously each memory element (internal and external) and I/O also see themselves as independently operating components. In the subsections below, the realization of each

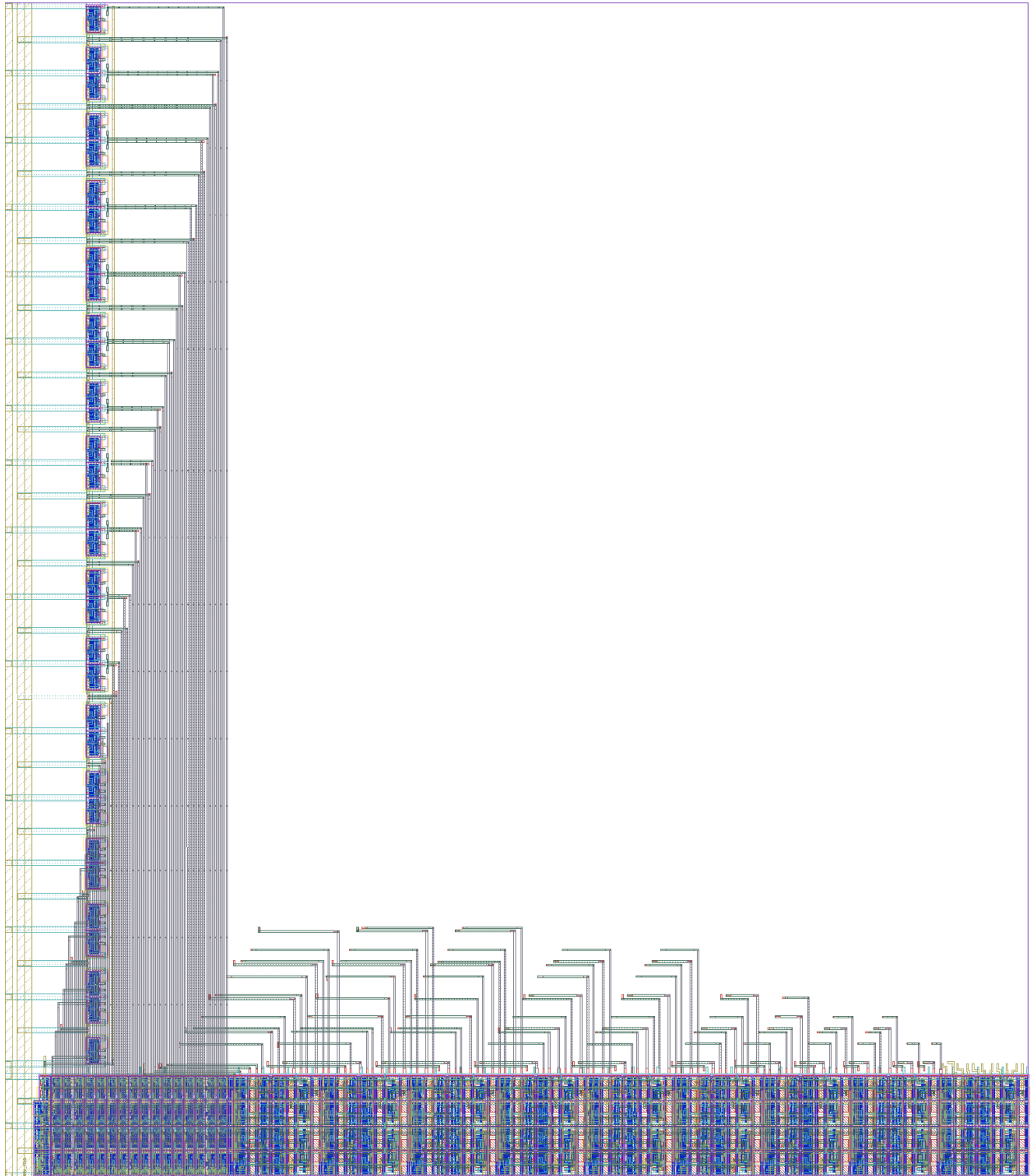


Figure 3.10: All of the SRAM control circuits for a 32-word block in TSMC 90-nm CMOS.

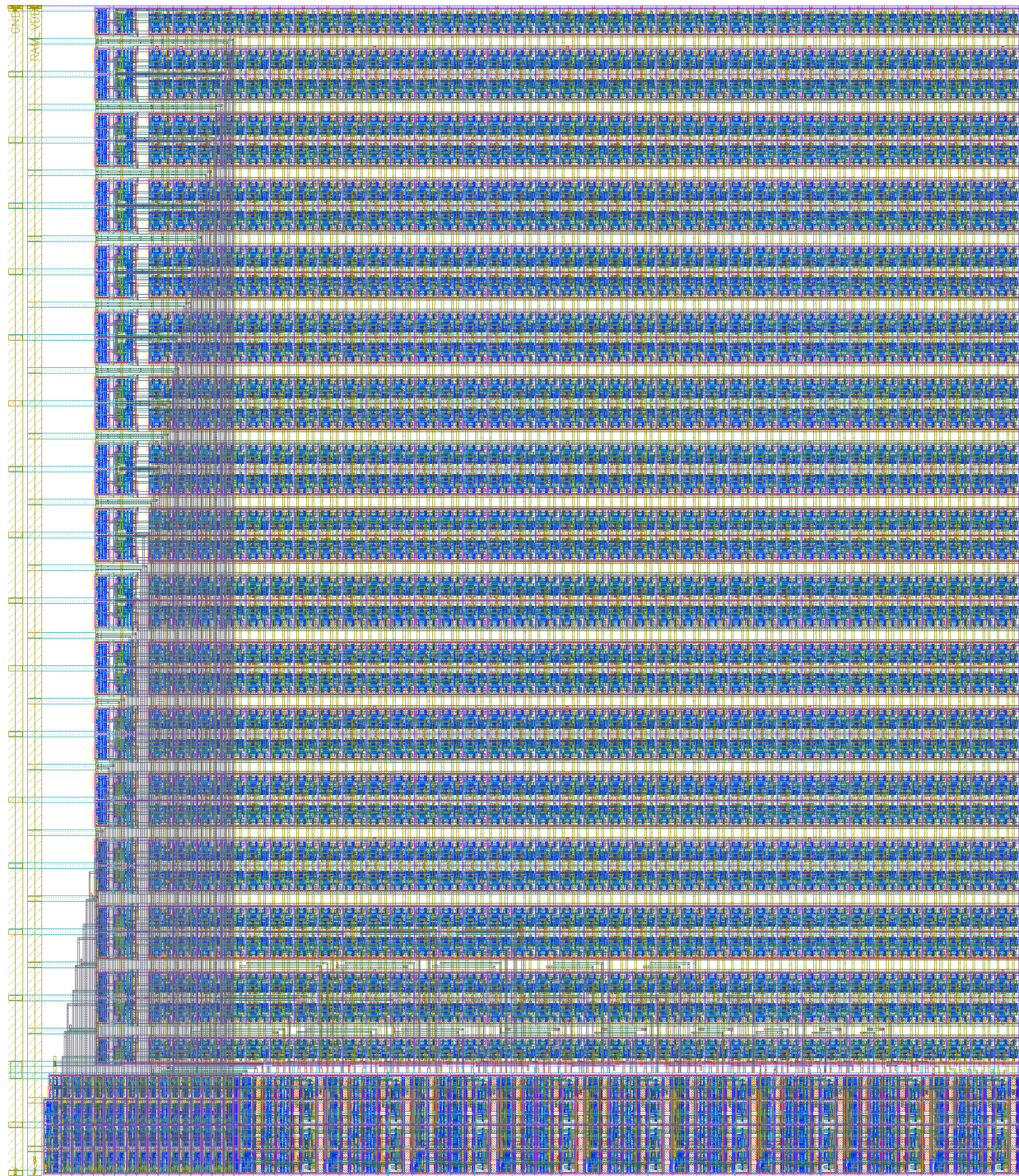


Figure 3.11: Layout view of a 32-word SRAM block in TSMC 90-nm CMOS.

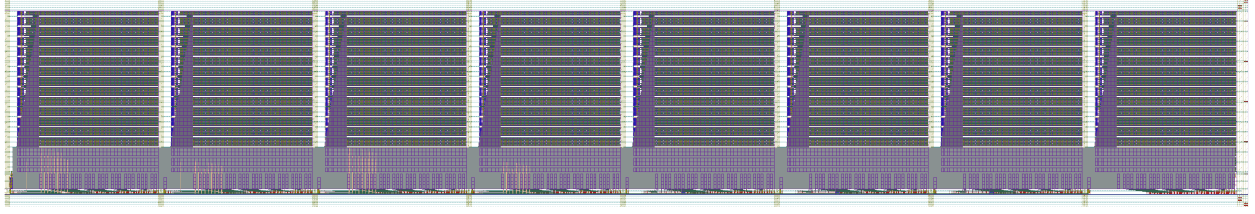


Figure 3.12: Full layout of the 256-word SRAM in TSMC 90-nm CMOS.

memory component and the cache controller are discussed in more detail.

3.3.1 Memory Organization

The modified MIPS processor is geared to use a 16-bit address space. This matches the size of the *immediate* field present in the **lw** (load word) and **sw** (save word) instructions. A *base addressing* procedure is used to access main memory. The 16-bit address can be used to access up to 64 kB of memory space (of which 1 kB is implemented in the 256 word SRAM cache described in section 3.2). This provides more than sufficient address space for instructions, data and I/Os required for the path-planning algorithm. In fact, for the purpose of running the algorithm used in this thesis (2×2 vectors), 440 bytes would be sufficient and can thus fit entirely in the cache, a circumstance that bears influence on its design as discussed in section 3.3.2. The 16 address bits allotted in this design simply allow the processor to be scalable to a much larger algorithm of higher complexity using existing hardware.

Also, the memory structure is made *word addressable* only (in contrast to a *byte addressable* realization) because all of the instructions and data are 32 bits and there is no direct need to address byte-wide sub-components of these instructions. Further, by increasing the data bus width between the memory controller and processor such that only word-sized segments are exchanged (unlike standard MIPS which moves words in byte-sized segments over a byte-sized bus), memory accesses become four times faster compared to byte addressable memory, and require a lower number of instructions. Given the local nature of this exchange (i.e. the word addressing is confined entirely to the chip, between the processor and the

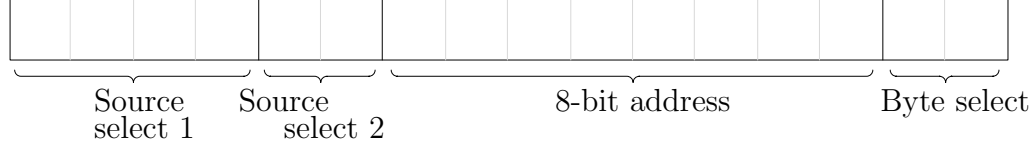


Figure 3.13: 16-bit memory address structure.

cache) the area penalty is minimal.

Shown in Fig. 3.13, is the 16-bit address interpretation. The 2 least significant address bits effectively describe the location of bytes in each word. Since the processor is designed to interact with the cache on a word-by-word basis, these two bits are obviously not considered in the cache. However, they remain a part of the addressing scheme to allow broader flexibility in corresponding with external memory and I/Os which may only be byte addressable.

The remaining fields of the 16-bit memory address structure are elaborated on in the following sub-section. In short, the *Source select 2* field represent the cache “tag”, allowing the processor to correctly identify data in the cache by its absolute main memory address. *Source select 1* is employed to allow the processor to communicate with external memory and memory-mapped I/O.

3.3.2 Caching Scheme

The caching scheme refers to how the 256-word, 1-kB cache memory is to be used. Key actions related to its use include: when and what data is to be written into the cache, when that data should be over-written, and when to read from the cache instead of using external memory.

The capacity of the cache memory is sufficient to contain the entire algorithm (which, as mentioned above, occupies only 440 bytes in its current form). Further, in its current form, the algorithm’s variables can be (and are) entirely retained and operated on in the processor’s register file during execution with no need for data fetch. Thus, having a cache memory large enough to store the entire algorithm can significantly reduce power consumption as it could completely eliminate the need for external memory access (slow and power consuming) after

each instruction has been executed.

The relatively large cache size also simplifies its design. Since the program fits entirely within the cache no conflicts can occur in the course of a memory access from the processor and the information stored in the cache. Thus a single block (i.e. *direct mapped*) cache organization can be employed. This provides an immediate savings in complexity and power as the control circuitry normally needed to manage associative cache structures is not required.

Thus, since the cache block size, b , is just set to unity the number of sets in the cache, S , is equal to the cache capacity, C which refers to the number of words (256) stored in it. As a result the 256 cache sets are referenced by the 8-bit *set* field in the address structure of Fig. 3.13.

To further reduce hardware requirements, the address *tag* field (i.e. the *Source select 2* in Fig. 3.13) is set to only two bits. This 2-bit tag thus allows the 1-kB cache to reference up to 4 kB from main memory. Addresses after 0x0FFF (i.e. with *SourceSelect1* > 0) will never be cached by the processor in order to reduce possible conflicts in larger algorithms and to avoid caching I/Os. This does not affect the total addressing space, just the “cacheable addressing space”. That is, the processor is still allowed to operate with a 64-kB main memory structure, but will attempt to cache data addressed between 0x0000 and 0x0FFF of main memory; data addressed outside this range will be communicated between the processor and one of external memory or some choice of I/O (as determined by some corresponding *Source Select 1* code). Therefore, this limitation does not impact the processors scalability to the algorithm since the entire cache space is still available and the non-cacheable address space remains accessible for external memory modules and I/Os.

Further modifications from the traditional caching scheme in the modified MIPS include a replacement of the standard “dirty bit” indicator with a “variable bit”. The dirty bit (or “used bit”) is a conditional bit stored for each *set* of data in the cache. It indicates that the data within the cache has been changed since last being loaded from the cache and hence

should be written back to external memory before being overwritten with any other content to keep all memory stores consistent [4]. Such functionality is not required in the case of the path-planning algorithm because all variables are contained inside the algorithm instructions, and write operations to cache memory are unnecessary (write to external memory function still exists).

In the case of the 2×2 vector algorithm, the 32 registers available in the processor are more than sufficient to contain all intermediate variables. However, if a larger algorithm needs extra variable space, the cache memory can be used and this is where the “variable bit” comes in. The entire cache (up to 1024 bytes) is available for variable space if needed. To use the cache as variable space, the processor simply uses the **sw** instruction to an available address (use different tag from instruction if there is a conflict); the variable will be saved and be accessible later. The **sw** instruction will automatically activate the variable bit indicating that a variable was stored and should not be over written as instruction memory cache.

The cache memory is organized as shown in Fig. 3.14. The valid bit, variable bit and tag bits are invisible to the processor, and are read by the cache controller to determine whether to use the data from cache memory or not. The tag component acts as an extension to the address (address bits 9 and 10) to ensure that it is the correct data. If the data is both valid and correct, the cache controller gives the 32-bit data to the processor (i.e. if the cache was invoked using a **lw** instruction), and if not, it seeks the original data from external memory. If an external memory is accessed within the cache-able address range (again cache-able addresses exist in the range 0x0000 and 0x0FFF), the cache controller saves this data to the SRAM cache if it isn’t occupied by variable data.

An efficient way to use this cache is to have instructions use the lowest address (0x0000) to higher addresses and vice versa for variable memory. In the ideal case, there would be sufficient memory such that instruction cache memory and variable memory never overlap. If they do happen to overlap however (in cache memory with a different tag, actual 16-bit

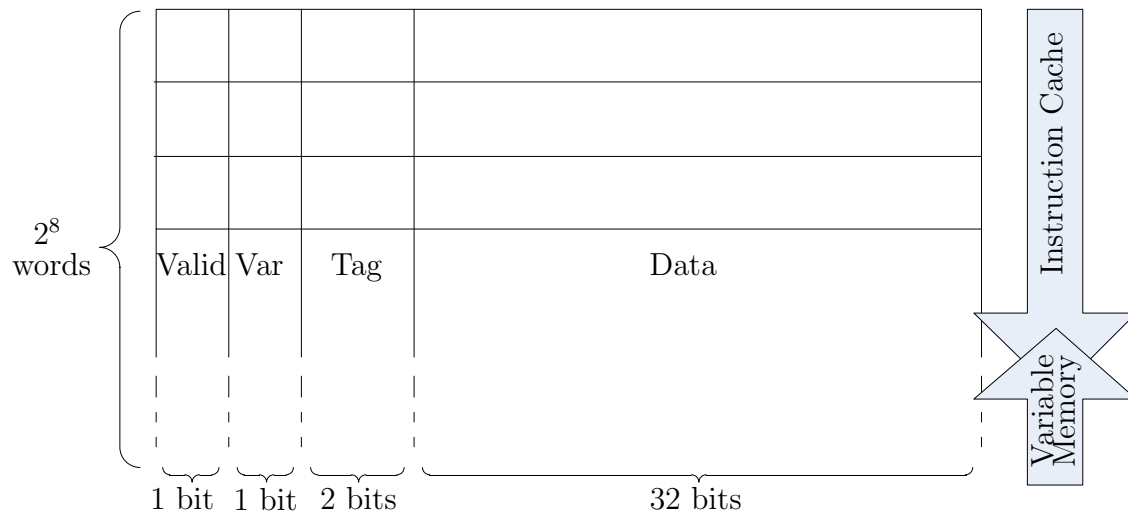


Figure 3.14: Structure of the cache memory.

memory address should never overlap), variables will take precedence and future instruction accesses from such address will be fetched externally.

3.3.3 Cache Controller

The cache controller manages interactions between cache memory, external memory(ies), external I/Os and the processor as described in previous sections. To do so, it is an one-hot state machine that sends out control signals to every component it is connected to. It controls different address lines, data buses, read/write operations, it determines the validity of the data, and it is able to halt the processor when waiting for memory access.

Fig. 3.15 is a state diagram that illustrates the operations of the cache controller. To simplify the diagram, each state is named by the operation(s) it carries out, rather than providing a whole list of its control signals. For more detail, see Appendix C.3 for the Verilog implementation of the cache controller. When the cache controller receives a read/write operation from the processor, it exits the *Stand By* state and halts the processor. Then, it carries out the read/write operation using the correct component(s), caches the data and/or prepares data for the processor where appropriate, then resumes processor operation before returning to the *Stand By* state.

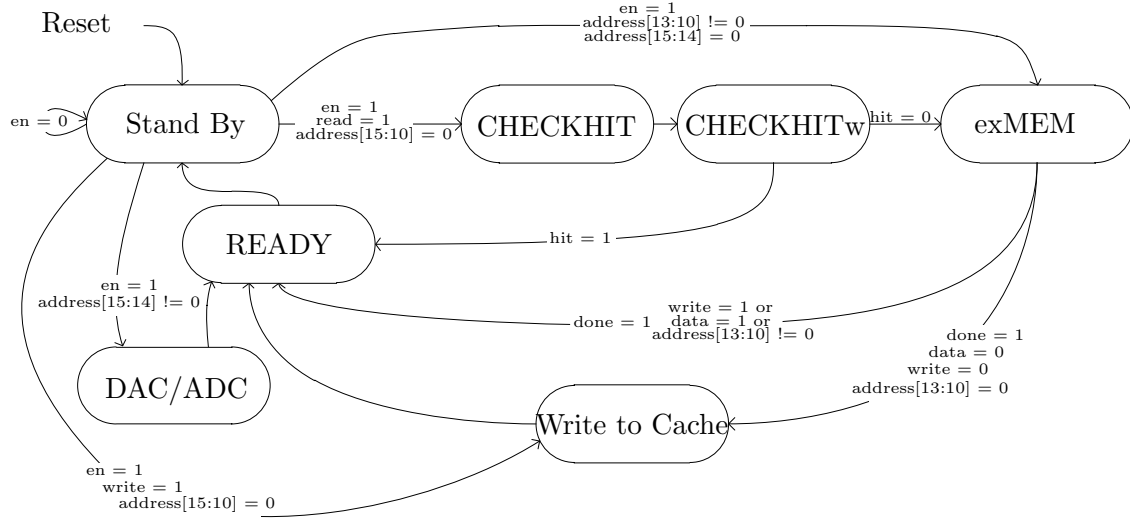


Figure 3.15: Cache controller states

It is important to note that most memory modules have 8-bit data buses and are byte-addressable. Because of this and the pin count consideration, the cache controller assumes that all the external components and analog components are 8 bits and are byte-addressable. The cache controller manages these byte-addressable components by latching each byte separately (8-bits/2-clocks), while word-addressable components such as the SRAM and the processor receive the entire 36-bit and 32-bit data in 2 clocks. Since the components will be using low voltages there are possible speed compatibility problems with external components. To address this, the cache controller generates a signal *exclk* that's half of the processor clock speed, and toggles when data is ready.

Although it is troublesome to manage both byte-addressable and word-addressable components, the SRAM and the processor are made to be word-addressable for better efficiency, and byte-addressable components are rarely accessed after the first loop. Each instruction would be cached upon first execution, therefore, access to external memory should be rare if any, after the first loop of the algorithm (out of thousands of loops).

3.4 Look-Up-Table (LUT)

The Look-Up-Table (LUT) is also a major modification specific to this processor. It acts as a second-stage ALU, where it takes the result of the ALU to a pre-mapped table that contains results for certain non-linear functions. In the case of this path-planning algorithm, it requires low-precision non-linear functions including trigonometric and hyperbolic functions. As an example, Fig. 3.16 is the LUT output vs. input compared with the real values for a sine function. Using only a 12-element LUT, the sine function could be calculated over the full processor's range (32-bit fixed precision). The processor uses the ALU to perform a modulus- π function. During the serial modulus calculation, there is an internal counter that keeps count of which of the 4 quadrants the input is at. The quadrant determines if the LUT values should be reversed, and if the output should be negated. The modulus output is then sent to the LUT and an output is produced.

Similar to the sine function, all the trigonometric functions are periodic and repeat values from 0 to $\pi/2$, some times in reversed order, and sometimes negated. Further, tangent is not required because it can be obtained by multiplying sine and secant. By utilizing 4 low-precision LUTs for *sin*, *cos*, *sec*, *csc*, results of all trigonometric functions including *tan* and *cot* and the negation of each function could be obtained.

Using similar techniques for hyperbolic functions, and by truncating extreme values, *sinh*, *cosh*, *tanh*, *csch*, *sech*, *coth*, $-\sinh$, $-\cosh$, $-\tanh$, $-\operatorname{csch}$, $-\operatorname{sech}$ and $-\operatorname{coth}$ could also be calculated.

The quantization noise produced by the LUT, shown in Fig. 3.16, would be similar for all LUT functions. Such precision is more than sufficient for the path-planning algorithm, and the broad support for all the trigonometry and hyperbolic functions would provide flexibility for all possible variations of the algorithm. The outputs of all the LUT functions could be found in section 5.2.2 and the verilog implementation in Appendix C.5.

The above operations are all done automatically by the processor. As a user, LUT

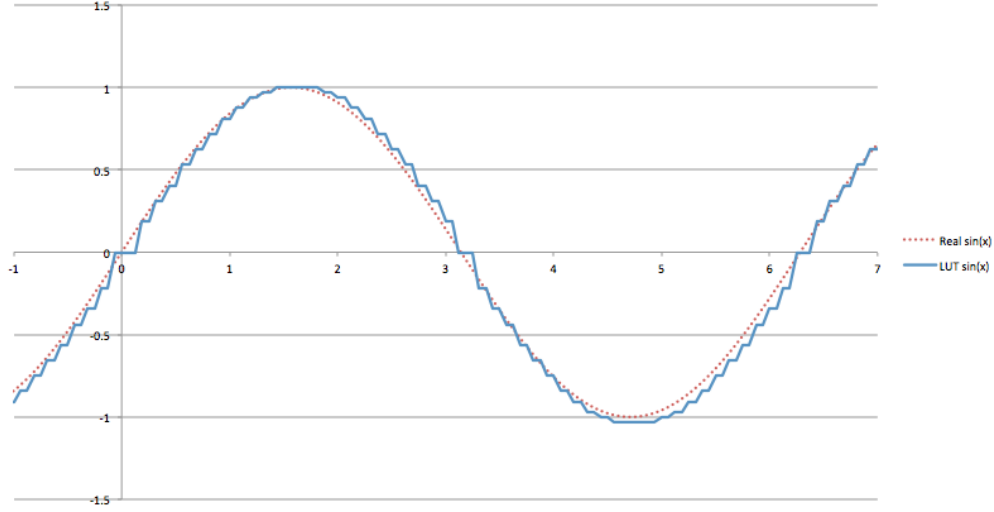


Figure 3.16: Comparison of LUT outputs vs Real values of $\sin(x)$

functions are simply executed by an R-type instruction like all other arithmetic functions. The function code $11X_3X_2X_1X_0$ would be decoded as an LUT instruction, where X_0 selects whether it is a sine or cosine, X_1 inverts the output, X_2 represents hyperbolic functions and X_3 negates the output. Each LUT function and their respective function code is shown in table 3.1

3.5 Analog Components

The 90-nm CMOS design of the modified-MIPS processor included analog-to-digital (A/D) and digital-to-analog (D/A) converters (i.e. ADC and DAC) contributed by Zhixing Zhao and Arshan Naji respectively. The placement of the data converters within the processor is indicated in Fig. 3.17. Two versions of the DAC are included to simultaneously produce signals for both the translational and rotational motion of the robot. For convenience these units are interfaced with the MIPS processor through a test block which is discussed in section 3.6.

A schematic of the ADC architecture is shown in Fig. 3.18. The design adheres to a successive-approximation (SAR) structure, an approach well know for its ability to operate

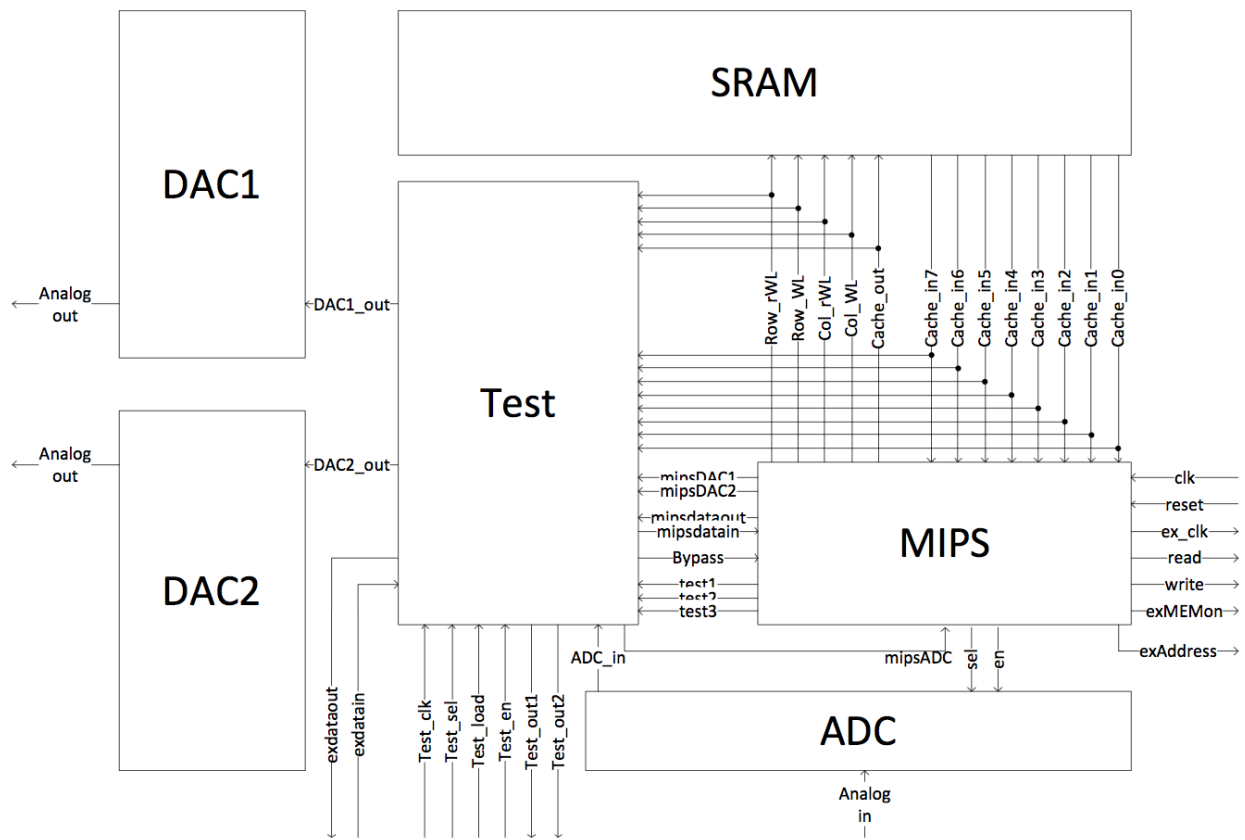


Figure 3.17: The system arrangement including the location of the data converters.

Table 3.1: LUT function codes

Function Code [5:0]	LUT operation
110000	$output = \sin(input)$
110001	$output = \cos(input)$
110010	$output = \csc(input)$
110011	$output = \sec(input)$
110100	$output = \sinh(input)$
110101	$output = \cosh(input)$
110110	$output = \operatorname{csch}(input)$
110111	$output = \operatorname{sech}(input)$
111000	$output = -\sin(input)$
111001	$output = -\cos(input)$
111010	$output = -\csc(input)$
111011	$output = -\sec(input)$
111100	$output = -\sinh(input)$
111101	$output = -\cosh(input)$
111110	$output = -\operatorname{csch}(input)$
111111	$output = -\operatorname{sech}(input)$

with low-power consumption (albeit low to moderate speed) [33]. It follows the specific implementation details reported in [34] and is designed to achieve a resolution of 8 bits.

The basic idea of the SAR structure is to obtain the digital representation of a sample (from a sample-and-hold, S&H, block) one bit at a time, from the MSB to the LSB, by asking the hardware to determine whether the sample is above or below half the available signal. In this way an N -bit representation can be obtain in N steps.

A typical SAR ADC implementation employs a comparator to decide on the relative value

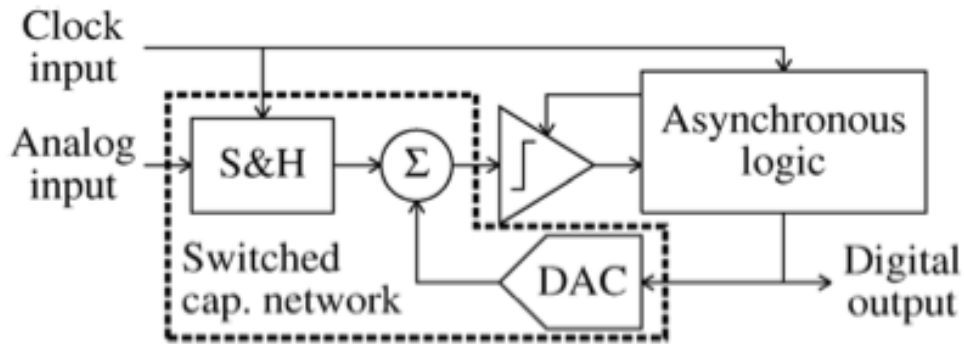


Figure 3.18: Architecture of the successive-approximation ADC. From [34].

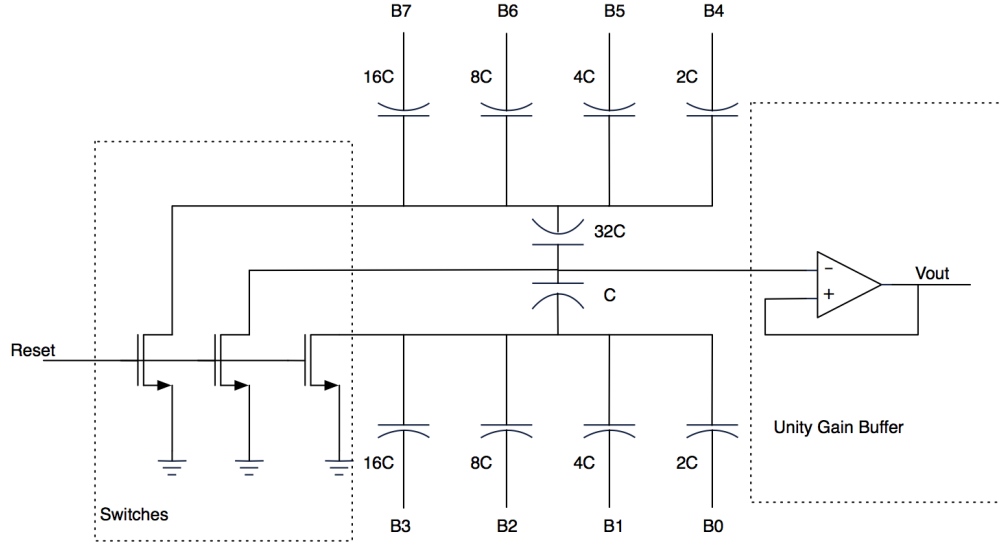


Figure 3.19: Architecture of the DAC.

of a signal and digital logic to decide on the bit whose value is determined by the comparator's action. Based on this information the digital logic creates an offset word that is converted to an analog signal by a simple DAC which then offsets the net value of the sample (appearing after a summing circuit) that's fed into the comparator for the determination of the ensuing bit. In the design, the comparator clock signal is generated inside the ADC, which is not synchronized to the system clock. This asynchronous clock simplifies the circuit design and saves the power consumption.

For the DAC design pictured in Fig. 3.19, a capacitive-ladder architecture is used to realize an 8-bit converter. In this case the current funnelled through the switches on the left is integrated by the weighted capacitive elements to achieve a binary summation that produces a net analog voltage proportional to the magnitude of the digital input signal, $[B0 \dots B7]$. The unity gain buffer, buffers the capacitor's output such that the operation of the capacitive elements is not degraded by the load being driven. As with any functioning 8-bit DAC the relationship between the output voltage and the binary setting is

$$V_{out} = \frac{B0}{2^8} + \frac{B1}{2^7} + \dots + \frac{B7}{2^1}. \quad (3.3)$$

The advantage of this topology over binary weighted capacitors is in its substantial

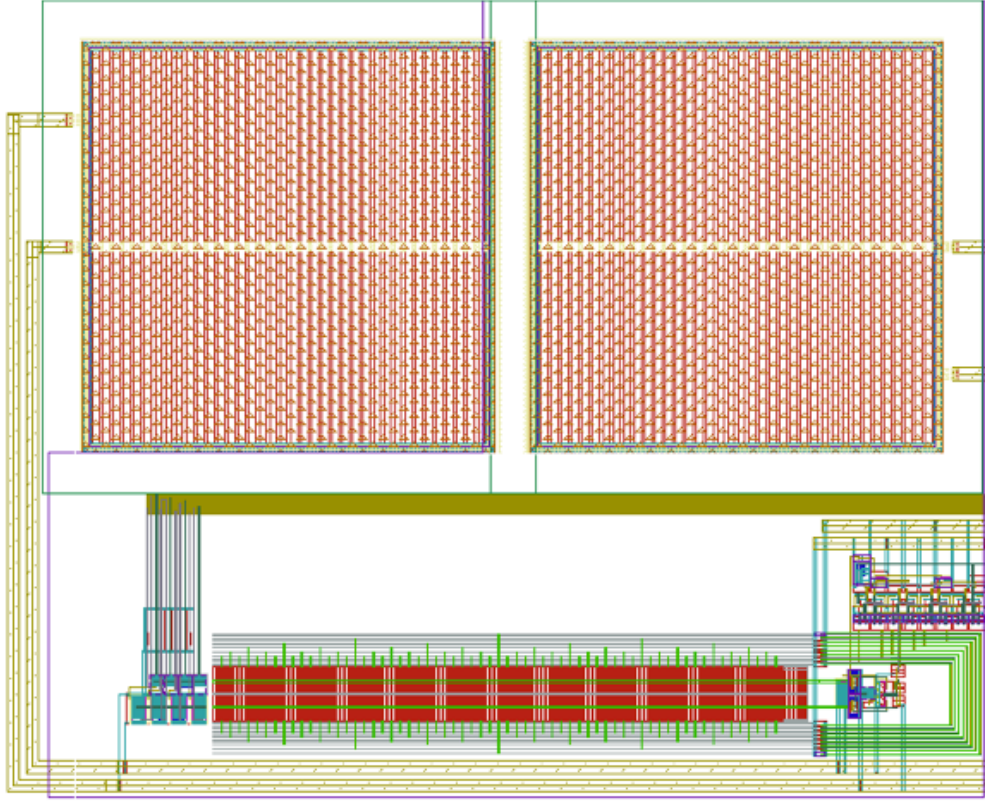


Figure 3.20: Layout of the successive-approximation ADC.

savings in layout area requirement. The switches on the left side of the schematic in Fig. 3.19 provide a path to ground for the floating nodes of the capacitors. After a couple of cycles, and depending on the frequency of operation, this circuit must reset to avoid accumulation of extra charges on the floating nodes of the capacitor. A reset means all the inputs must be zero and switches have to ground all the floating nodes.

Layouts of the ADC and DAC in 90-nm CMOS are shown in Figs. 3.20 and 3.21, respectively. The ADC layout measures $200 \times 245 \mu\text{m}^2$ while the DAC layout measures $264 \times 226 \mu\text{m}^2$.

3.6 Test Circuits

Most fabricated digital circuits require some sort of test circuitry. Because of the complexity of digital circuits, it is extremely difficult to debug without a well-designed test circuit [35].

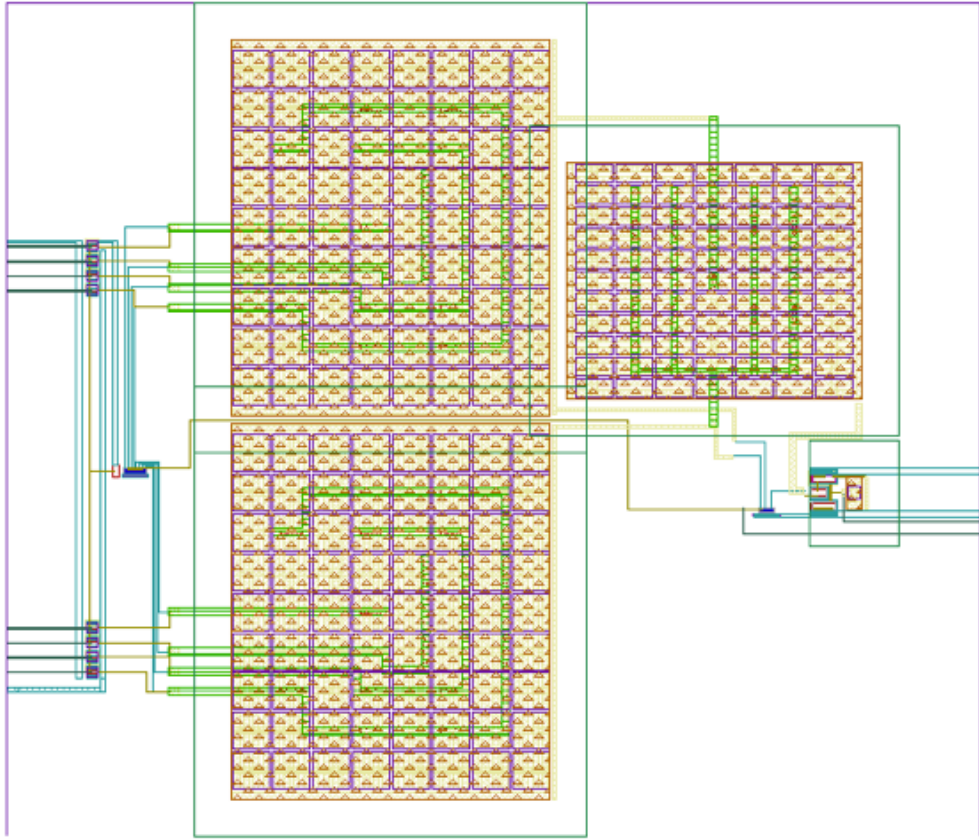


Figure 3.21: Layout of the binary capacitive weighted DAC.

Such test circuits usually consist of scan chains where test data are read serially from a test pin that replaces selected internal signal(s) and/or pins while the output is observed. To do so, the test structure consists of a series of registers to propagate the serial input and some latches at the target signals to isolate and re-route the signal. Similarly, to read an internal signal, latches reroute the signal to be loaded into the register chain, and the sequence outputs to an output pin serially. Read/write scan chains allows the tester to sample and/or replace an internal signal while the processor is running.

In this work, the test circuit design also included component isolation and bypass in addition to read/write scan chains. Component isolation allows each component to be tested individually. This includes the processor, SRAM, and both the A/D and D/A converters. The component isolation feature is especially important for a design this complex so that faults can be identified. The component bypass function is just as important so the potentially faulty component could be bypassed while the rest of the circuits are being tested.

This cooperative work also required careful component isolation so that each designer could test their own component(s). Cooperative work may also be more error-prone, hence the importance of the component bypass feature as well. To further improve the reliability and testability of this project, each major component has its dedicated power and ground pads to be connected externally.

Implementation detail could be found in Appendix C.4.

Chapter 4

FULL CHIP INTEGRATION

The work in this thesis led to the fabrication of two processor designs in silicon, one in IBM’s 130-nm mixed-mode CMOS technology and the other in TSMC’s 90-nm CMOS technology. Both submissions included months of Verilog coding, simulations, synthesis, and layout. In the sections below, Section 4.1 focuses on the entire design flow while Section 4.2 focuses on the enhancements and differences made in the TSMC 90-nm design which followed the processor’s initial implementation in IBM’s process.

4.1 IBM 130-nm CMOS Technology

The first instantiation of this chip was fabricated by IBM using its p13 technology (formally known as its 0.13- μm or 130-nm CMOS technology). Table 4.1 includes the fabrication details of this submission.

Table 4.1: IBM P13 Fabrication Detail

Design Run	1201CG
Design Name	ICGCYSUP
Exact Dimensions	2.42 mm \times 2.42 mm
Principal Designer	Ryan Wu
Sponsorships	Dr. Sebastian Magierowski Dr. Jim Haslett CMC Microsystems

The simplified design flow demonstrated in Fig. 4.1 shows only the critical parts of the design; understandably, there was a substantial amount of simulation and verification involved during and after each step. This work relied heavily on the IC design software available from Cadence[®] and Synopsys[®] and the flow employed many of their tools including: Synopsys Compiler, SOC Encounter, Virtuoso Schematic, Virtuoso Layout, Virtuoso Layout-XL,

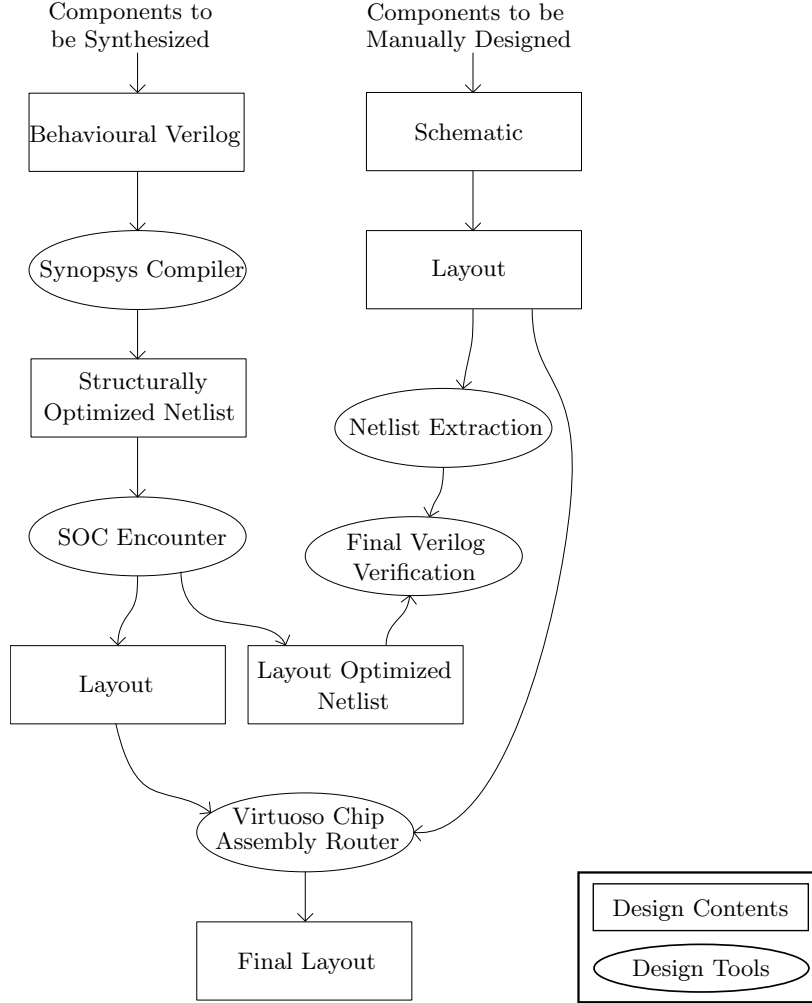


Figure 4.1: Design flow for VLSI Circuits using Synopsys® and Cadence® toolsets.

Virtuoso Chip Assembly Router (VCAR), Calibre DRC/LVS, Assura DRC/LVS, Analog Environment, NC-Verilog, PSPICE, HSPICE, etc. Though troublesome, this distributed tool set is presently the de-facto industry standard design flow used in VLSI design and is similar to the ones described in [36] and [37].

There were many optimization parameters in the compilation and synthesis process that required careful tuning. It is important to note that these parameters contribute to the final size, power and performance of the chip. In addition, because of some incompatibilities between the tools, configurations of the tools were also critical. The scripts for setting up and running the tools are attached in appendices B.3, B.4, B.5, B.6, B.7, B.8, B.9, B.10, B.11

and B.12. However, test scripts and benchmark source codes are not included because the specific implementation detail does not contribute to the final design.

Shown in Fig. 4.2 is the final layout of the chip in p13 technology. The very thin ring with beveled corners around the chip is the guard ring that prevents unwanted static discharge and cracks during fabrication. The thick square ring of circuits are the 64 I/O pads that supply multiple power levels, multiple grounds, and input/output connections to the core. Along the top of the core is the 256-word SRAM, and the block in the centre is the processor with the test circuits to its left. It is apparent that the size of the chip is limited by the number of pads used (so-called *pad-limited design*), and there is plenty of unused space inside the pad ring. Therefore, all the optimization parameters during synthesis were for power optimization rather than size. It is also why the design tried to minimize the number of pads.

4.2 TSMC 90-nm Technology

The second chip included a lot more features and improvements over the p13 implementation. Considering the technology alone, 90 nm CMOS naturally provides less delay (i.e. faster switches) and less dynamic power consumption (lower device capacitance to charge). This means that with the same design and performance requirement, the chip in 90-nm technology would operate more reliably while consuming less power. The chip area used was 40% smaller, despite the mass number of pins and the increased number of functionalities and components. In other words, the decrease in chip size was largely due to the more efficient design in addition to the smaller technology. This is demonstrated clearly when comparing the size of SRAM to the size of the MIPS processor. Table 4.2 includes the fabrication detail and in Fig. 4.3 is the final layout.

It is visible that both the MIPS core and the SRAM are significantly smaller than the previous design. The new features include analog component support in the memory con-

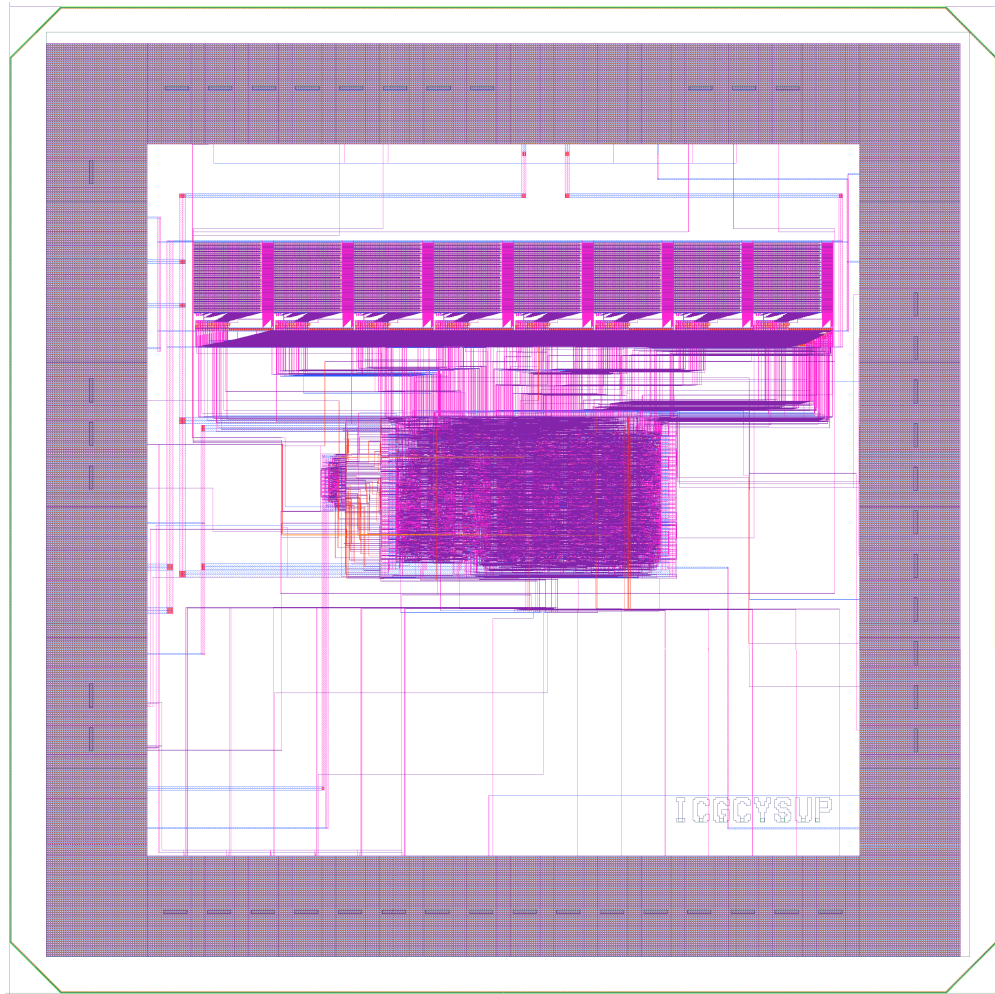


Figure 4.2: Final layout of the IBM p13 chip (without metal fill).

Table 4.2: TSMC 90-nm Fabrication Detail

Design Run	1201CR
Design Name	ICRCYSUP
Exact Dimensions	2.08 mm \times 1.73 mm
Principal Designer	Ryan Wu
Other Contributors	Zhixing Zhao Arshan Naji
Sponsorships	Dr. Sebastian Magierowski Dr. Jim Haslett CMC Microsystems

troller and higher testability to handle extra components. The 2 D/A converters located to the left of the processor, and the A/D converter placed below the processor are also new. Similar to the previous chip, the size is limited by the number of I/O pads. It is also visible in Fig. 4.3 that most of the pads have ESD protection and buffers to protect the chip (the thin strips of circuits attached directly to the pads), as well as to increase performance. Hidden inside the processor, are 2-stage buffers to help step the voltage up and down. These are the major differences between the p13 chip and the 90-nm chip.

The design process also differs slightly in that it is more automated with higher levels of optimization. Every single step of synthesis is automated by scripts allowing more detailed tuning and modifications.

4.2.1 ADC and DAC Integration

This is another step towards full integration of a robotics controller. It shows that the architecture developed in this work is capable of accommodating further integration and is flexible to potential modifications.

Each of these analog components has an 8-bit digital interface to the memory controller. The ADC serves as the analog input to the processor; assuming that the robot's sensor has an analog output, the processor will be able to read its analog signal and determine the correct path. Also, the ADC has a 4-to-1 analog multiplexer attached to it that allows it to sample 4 different analog signals (or from 4 different sensors). The 2 DACs would be

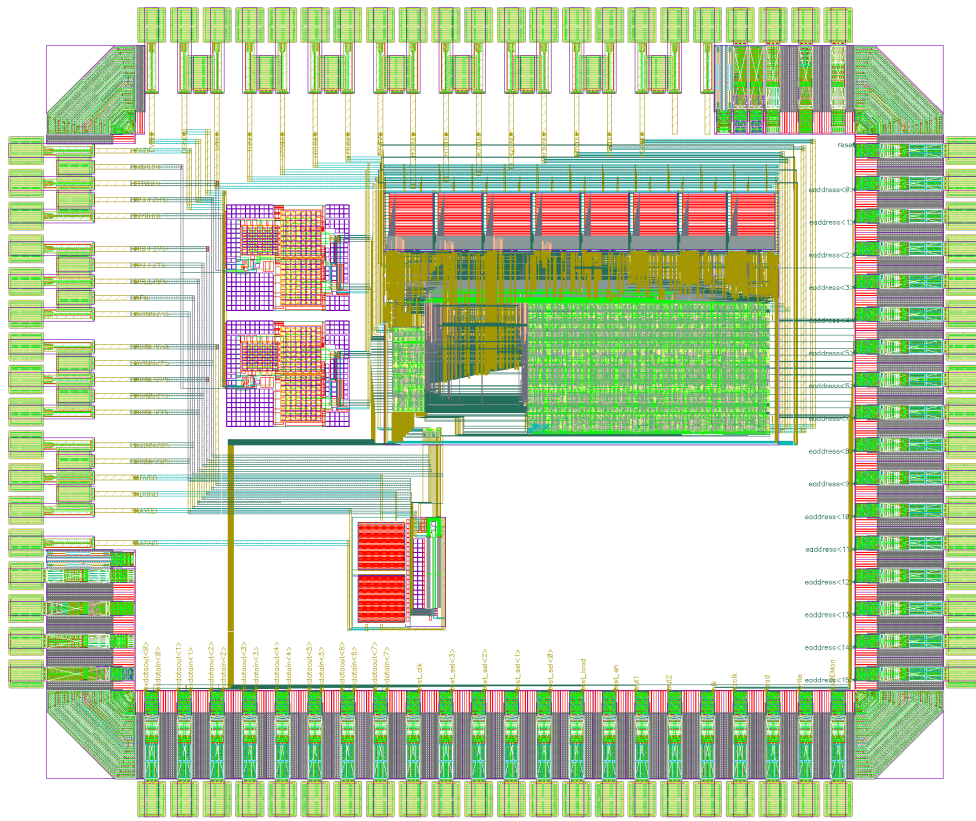


Figure 4.3: Final layout of the TSMC 90-nm chip (without metal fill).

used for the 2 analog output for motor control. Assuming that the motor controller takes an analog input, these outputs would represent the robot's speed and rotation.

The analog components were designed by Zhixing Zhao and Arshan Naji at the University of Calgary. Between the processor and these analog components, there are 2-stage buffers to step between different voltage levels as well as to isolate interference. Testability functions included here are component isolation and bypass so that the analog-to-digital converter (ADC) and digital-to-analog converters (DAC) could be tested independently and be bypassed if they fail.

Similar to the other components in this project, these analog components each have their dedicated power and ground supply for independent power measurements, and for device isolation purposes.

Chapter 5

SIMULATIONS AND TEST RESULTS

Divided into five sections here, are summaries of the major tests used to verify the functionality of the final design of this work. Although the implementation details for verification are not included here because they are very lengthy and these specifics do not contribute to the final design of this work, its process is important to prove the validity of this work. Included in the following sections are the steps taken to test the algorithm, analog circuits, digital circuits and the final simulation.

5.1 MATLAB Simulations

In addition to Fig. 2.3 which demonstrated successful navigation around a multi-obstacle course, Fig. 5.1 summarizes the results of a different verification test for the algorithm that utilized only a single-obstacle test scenario. In both cases, the modified (optimized) algorithm with additive quantization noise resulted in nearly identical paths compared to the original algorithm result shown in Fig. 5.3.

With several more test scenarios and comparisons to the original algorithm, each stage of algorithm modification was tested. The final algorithm in machine code (Appendix A.4) was tested using a processor emulated in MATLAB that translates each instruction into a mathematical command. Within the emulator the range and domain of all calculations in the algorithm is limited to $[-1024, 1023]$ with data employing Q21 format (more specifically Q11.21 format) in alignment with the fixed-precision characteristics [38] of the modified-MIPS hardware. This system is represented in Fig. 5.2 as a block diagram.

Using the final product, a full simulation run was recorded. The recording included all inputs, outputs and register values, and was later used for circuit verification.

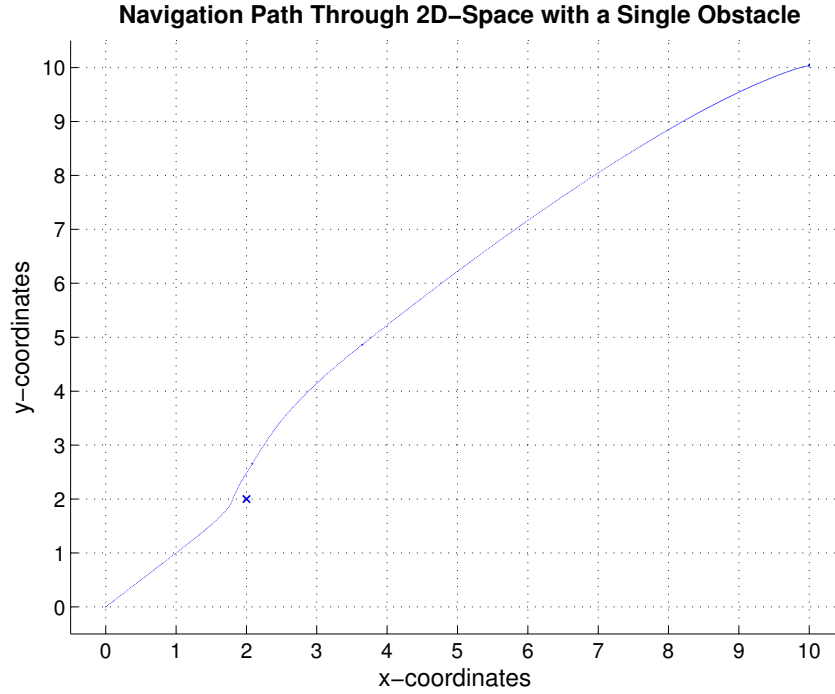


Figure 5.1: Result of translated algorithm (finite arithmetic, machine commands, etc.) simulation in Simulink .

5.2 Verilog Simulations

The Verilog level simulation spans from the initial design stage to the verification of the final processor design and a full FPGA implementation. This includes all of the digital circuits, SRAM, dummy analog components and all their interconnects.

5.2.1 Core Simulations

“Core Simulations” here represent simulations of the digital (synthesized) components at their top level, called the “core”. The core includes the MIPS processor and the test circuits. To test this, several different test algorithms were used to verify the correctness of the core’s output in different scenarios. Together, these tests covered the entire list of supported instructions and all of the test circuit functionalities.

These tests consist of creating a Verilog test bench that triggers a pre-defined series of inputs, then comparing the outputs and relevant intermediate signals. As an example,

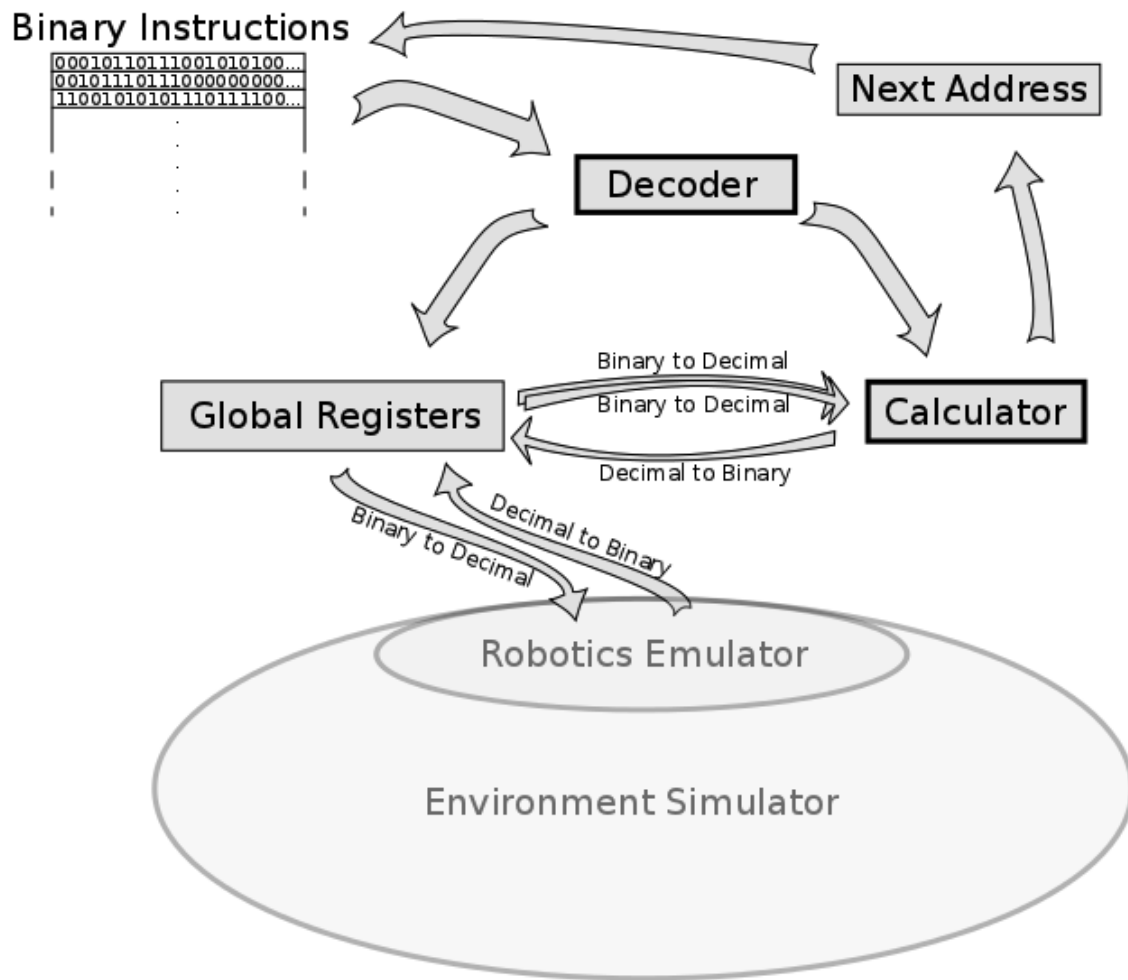


Figure 5.2: Block diagram of the MATLAB simulation.

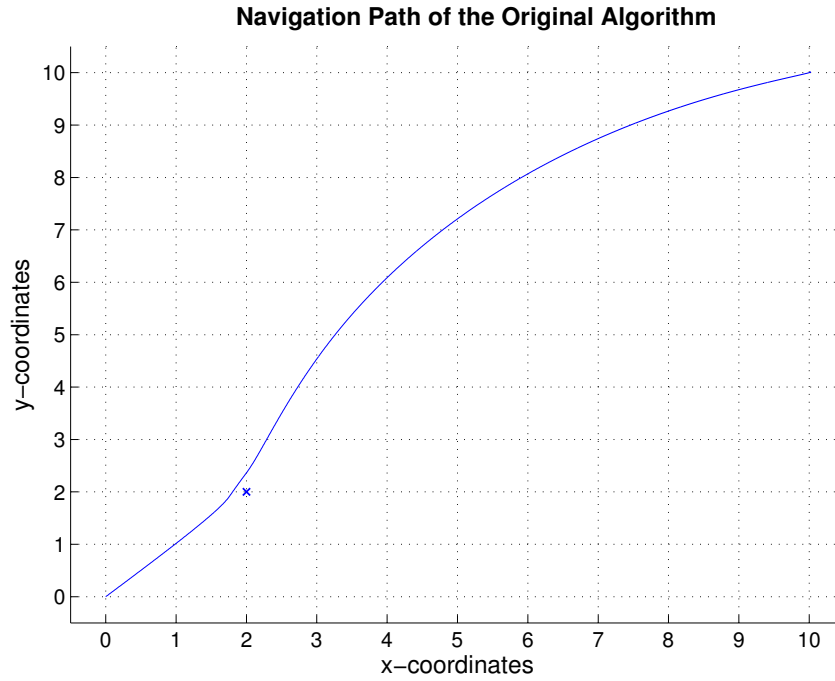


Figure 5.3: Result of the original algorithm simulation in Simulink.

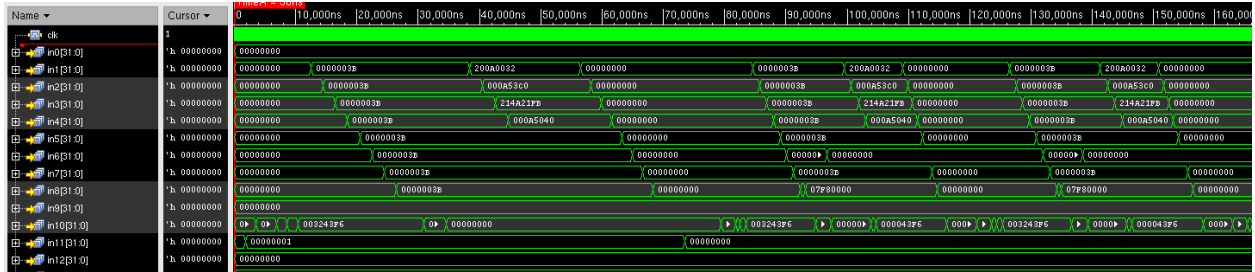


Figure 5.4: Results of the I/O test bench using NC_Verilog.

Fig. 5.4 is the result of a test bench that tests the read/write functions of the processor with external addresses, add and shift arithmetic, and the switch for SRAM bypass. All intermediate values demonstrate that correct values are being loaded, saved and calculated, and that there is a significant speed increase about halfway into the simulation, when bypass is turned off.

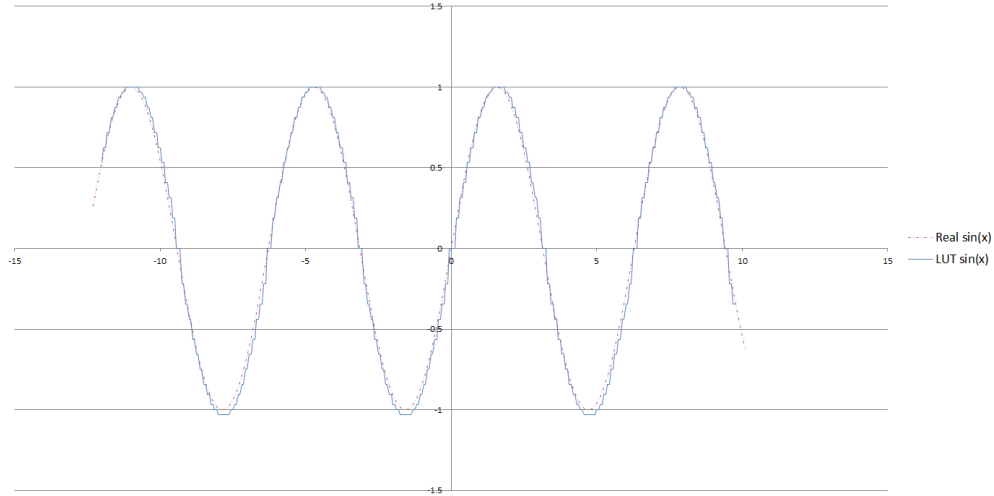


Figure 5.5: Look-Up-Table outputs of a sine function.

5.2.2 Look-Up-Table Simulations

Tests for the look-up-table were a bit more interesting. Because there were a vast amount of possible inputs to test in order to verify the shape of the waveform, it is impossible to toggle each input and record each output individually. Instead, a short test algorithm was used, that automatically increments the input (using the *add* function of the processor) over a period of time. All the intermediate outputs are recorded, and the useful parts are then extracted using a PERL script.

Shown in figure 5.5 is a great demonstration of the sine function produced by the look-up-table compared to the real sine wave. Limited by the range of the fixed decimal system employed by this processor, the input and output values are limited within $[-1024, 1023]$. Figs. 5.6 and 5.7 further show the output of a negative hyperbolic-sine and a hyperbolic-secant function respectively. Similarly, all variations of non-linear functions are simulated and graphed to verify their accuracy and precision.

The full results of every look-up-table function are listed in Appendix D.1.

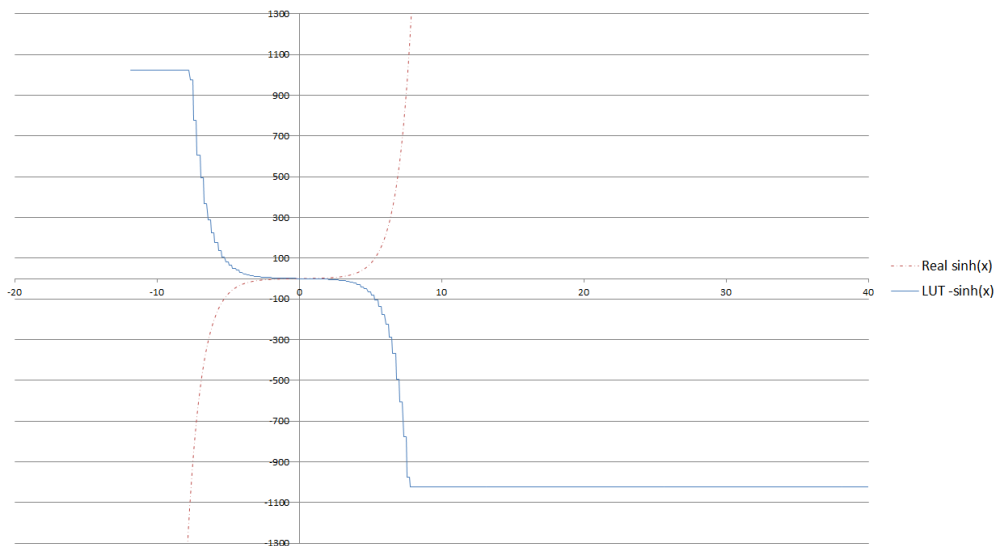


Figure 5.6: Look-Up-Table outputs of a negative hyperbolic-sine function

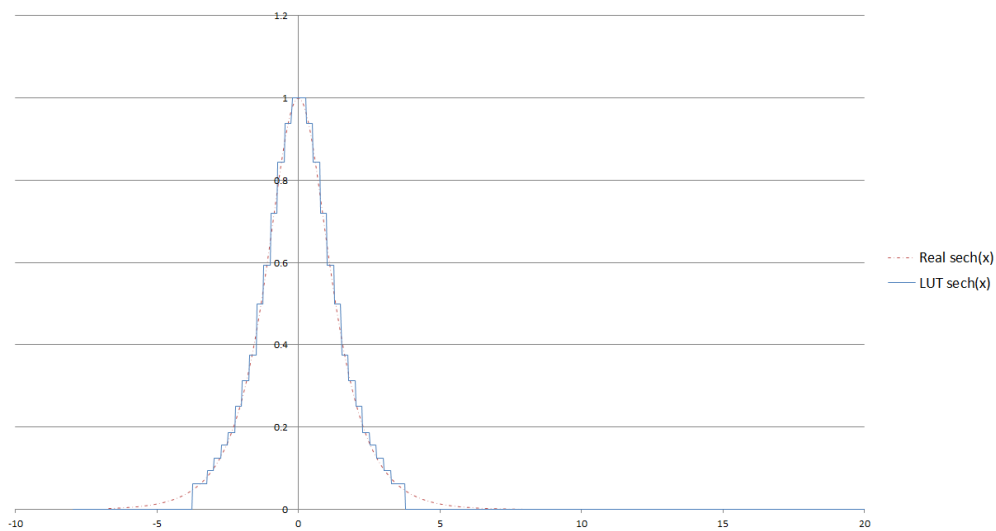


Figure 5.7: Look-Up-Table outputs of a hyperbolic-secant function

5.2.3 Final Verification in Verilog

Although the automatic place & route tools available from Cadence and Synopsys are extremely convenient compared to drawing a design that consists of hundreds of thousands of transistors, they occasionally create errors that are extremely difficult to debug. These errors include missing VIAs, missing connections and shorting wires during routing and export/imports between programs. Other errors are more common and they come in vast quantities. These errors are specific to the technology and include special-case area limitations, inter-components bus wire spacings, thick wire spacings, inter-layer rules, antenna rules, and density rules.

This final verification (after DRC — the “design rule check” — and LVS — the “layout vs. schematic” check) is extremely important in order to make sure that all the internal connections and logic are still correct. Because of the design complexity, it runs very slowly. Therefore, an efficient test that accurately captures the operation of the design is important.

In this test, dummy components made of registers are used to simulate the latched input/outputs of analog parts. The full netlist of the processor, test circuits, and SRAM are extracted from the layout. Then, using the libraries’ Verilog definitions, a full simulation was run with the actual algorithm. The inputs of this simulation are from the recorded MATLAB simulation, and all intermediate and final results are compared against the MATLAB results.

5.3 Analog Design Environment (ADE)

The Analog Design Environment (ADE) from the Cadence tool suite was used to test the operation of smaller components with simple behaviours. ADE is a graphically-driven front-end for Spectre, a SPICE-like integrated circuit simulator also within the Cadence tool suite. Among the things studied using ADE were the electrical characteristics of a single-bit 10T SRAM cell and the row-reset module for the SRAM. Spectre (using physics-based BSIM4 MOS device model) provides much more detailed electrical performance results compared to

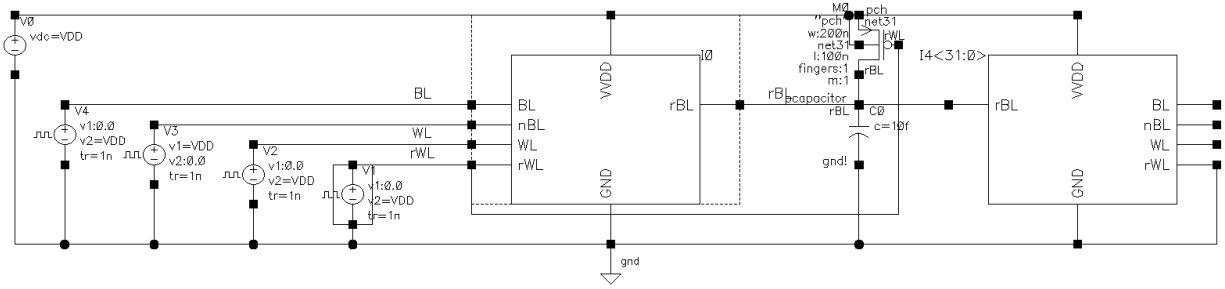


Figure 5.8: Single-Bit SRAM cell test setup

Verilog, and is necessary for the simulated evaluation of non-digital components.

Taking the single bit SRAM cell for example, we have a 7 port (including power) structure to characterize. Shown in Fig. 5.8 is the test setup, where the centre block is the device-under-test (DUT), the SRAM cell. All input ports are toggled using pulse generators and the output is loaded with expected components for leakage, parasitics and pre-charge. This is the most detailed analog verification in this work because the chip will be expected to operate at 100 kHz or within the low frequency (LF) range of 30 - 300 kHz. The circuit response in the higher frequency range is irrelevant and not considered. Also, voltage sources and the operating environment are assumed to be ideal in order to eliminated further complications.

Fig. 5.9 shows the results of the read/write test that demonstrate the 10T design can successfully store information from a 400-mV power supply and that the stored information can be reliably read. The 5 rows of signals from top to bottom are bit line (BL), word line (WL), stored bit, read word line (rWL) and the read bit line (rBL). In the centre row, the stored bit is toggled repeatedly, and exemplifies a very clean and fast response to write operations. The read bit line (bottom row), shows a longer delay because of the large parasitics of the long line shared between multiple cells. Nevertheless, because of the signal isolation in the 10T design and the line recharge, the output produced was very accurate.

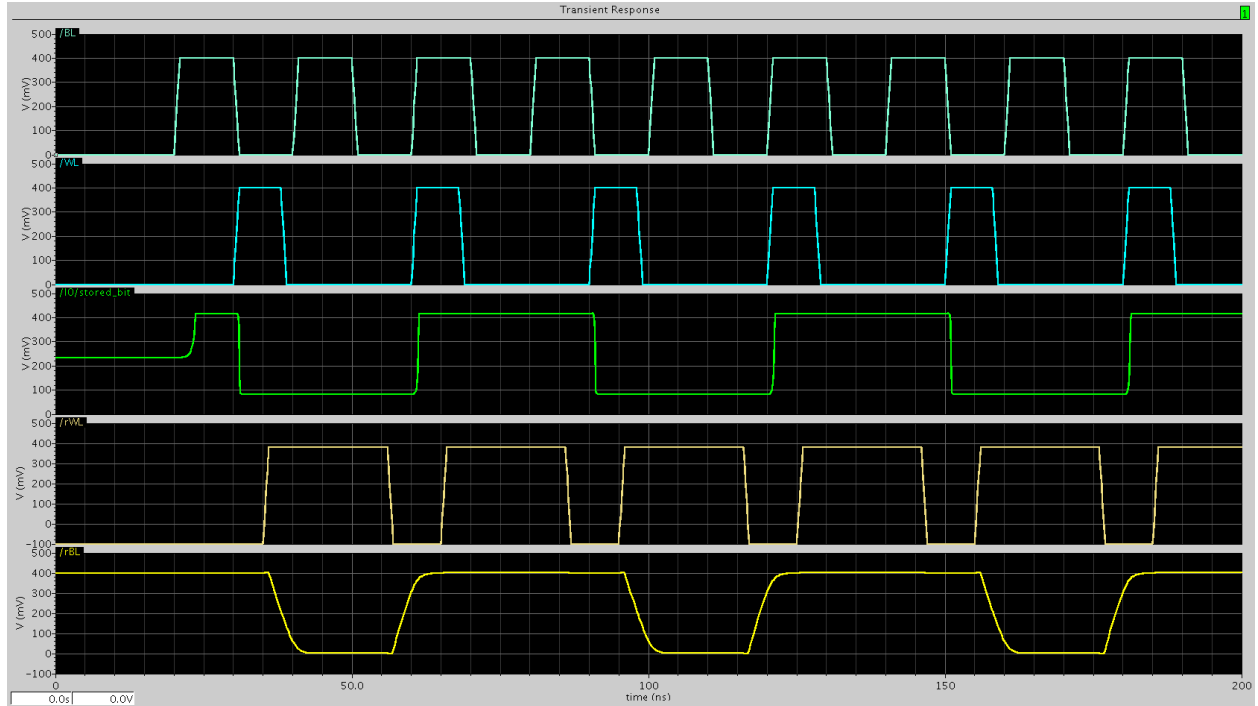


Figure 5.9: Single-Bit SRAM cell test simulation results.

5.4 SPICE

An automated (using scripts) test using HSPICE (from the Synopsys tool suite) was conducted during different design stages of the SRAM. Although Spectre could also be used for these simulations, the HSPICE interface proved more convenient for this task. As with Spectre, the HSPICE simulations utilized physics-based BSIM4 models of the transistors.

The SRAM tests included verifications of the row controller, column controller, 10T cell, the word row, the 32-word block and the entire 256-word SRAM. Like the ADE tests, the SPICE script toggles various input signals and verifies the outputs against the expected output after a defined delay time. Because of the exponential increase in design complexity, ADE simulations were no longer suitable, and the detailed analog response was less important. HSPICE became the choice of tool for analog simulation for these more complex circuits.

In the case of the 256-word SRAM simulation, it was a 48 step input sequence that reads

and writes to same addresses, different addresses, adjacent addresses and addresses in different blocks. The output is then compared to the expected output to produce an error file for debugging. In the final design, the error file contained 0 errors. This simulation demonstrates that under the assumed conditions, the analog behaviour of the SRAM produced correct logic, the design is valid, and that the connections are correct.

5.5 FPGA Tests

To validate the actual operation of the entire design, several tests were performed on the FPGA that include the exact designs used for fabrication. The FPGAs used included the Xilinx Virtex 2, Spartan 6 and Virtex 6. Each board had different functionality and availability, hence several different boards were used. To start off, each component was individually tested on the Virtex 2 (the Virtex 2 board had a large number of I/Os, but small amount of memory), where it was possible to trigger and probe each signal of interest separately. These components included the simulated external memory, SRAM, MIPS processor, LUTs and the ALU.

Bigger tests that included the entire design were run on the Spartan 6 board that had bigger memory space to fit the entire design. The Spartan 6 board has very limited I/O's, and therefore, only a few results could be checked. Because the entire processor was used, the only required inputs are the reset and clock signals, while the test bench is saved in the simulated external memory. The tests included functionality verifications that check the operation of different instructions and test functions.

Lastly, using the Virtex 6 board, a real-time simulation of the entire design was checked using the onboard LCD to constantly display internal register data. Using the push buttons as data and instruction triggers, and the pre-recorded MATLAB simulation as simulation data, the entire algorithm was executed using the processor on the FPGA. Then, using the LCD display, the register data were verified after each instruction with the final result

matching the recorded MATLAB simulation. This, at the very least serves as a proof of concept validation of the design in a hardware instantiation.

5.6 Detailed Final Simulation

In this final test, it is already known that the design works and that its internal interconnect and logic functions are both correct. The purpose of this test is to obtain some data regarding its actual analog operation and power consumption.

Since the p13 project used a black box library that doesn't allow simulation, only the 90-nm project could be tested for this part. Using numerous pulse generators, a short series of instructions are generated to initiate a multiply operation, and the processor's operations are monitored in Analog Design Environment. It generated the correct logic results as expected, and had power consumption within expectation. Fig. 5.10 illustrates the power consumption curve showing the decrease in power as the operating voltage is scaled down. Also, at the target operating range of around 400 mV to 500 mV, the power consumption was approximately 10 μ W, which is orders of magnitudes lower than off-the-shelf general purpose processors, while also being small, light, and versatile. Comparing processing throughput in millions of instructions per second to power and weight (accounting for the full 2.08×1.73 mm² chip area) simulations indicate that this processor can achieve approximately 35,000 MIPS/mg/W. This exceeds (by 44%) the cognitive efficiency reported in [6] with twice the peak MIPS potential (1 MIPS vs. 0.5 MIPS). Further the present design includes 4x more on-chip memory, plus the space to include three data converters capable of producing two analog outputs and handle four analog inputs (the power consumption of the analog components is not included in accessing the cognitive efficiency).

Fig. 5.11 shows the schematics of this test setup, where, on the right-hand-side, is placed the MIPS processor. And to its left, are the pulse generators and the VDD voltage supply. While having frequency and VDD as simulation variables, the power consumption at 100-

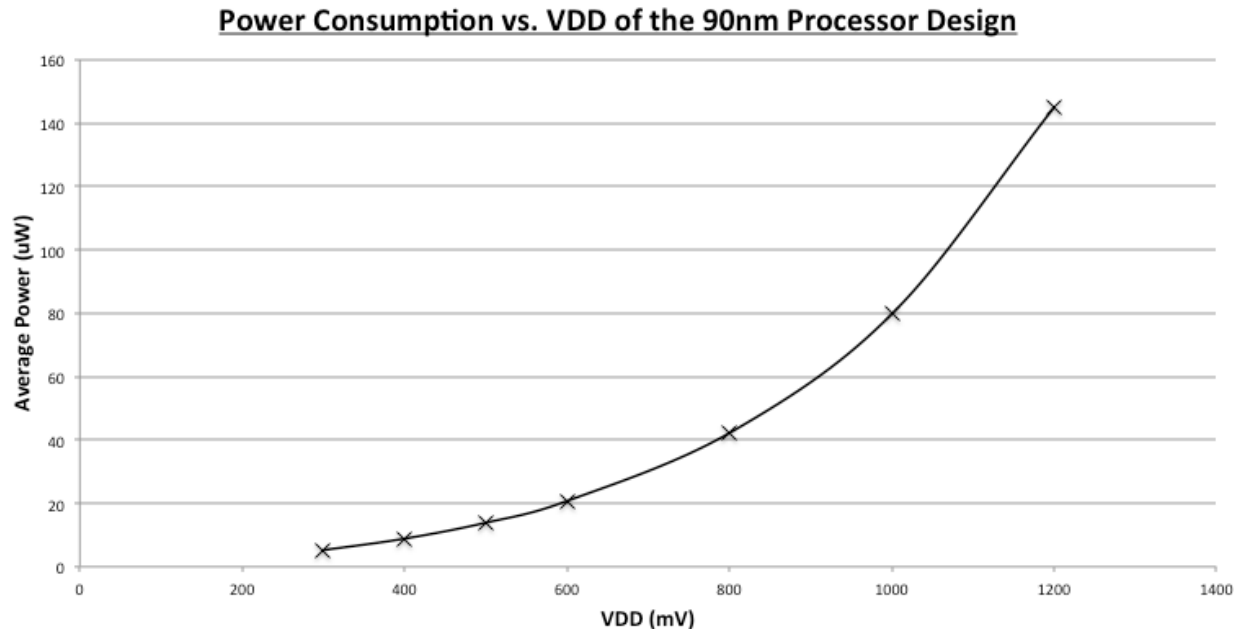


Figure 5.10: Power simulation of the 90-nm chip in operation at 100 kHz.

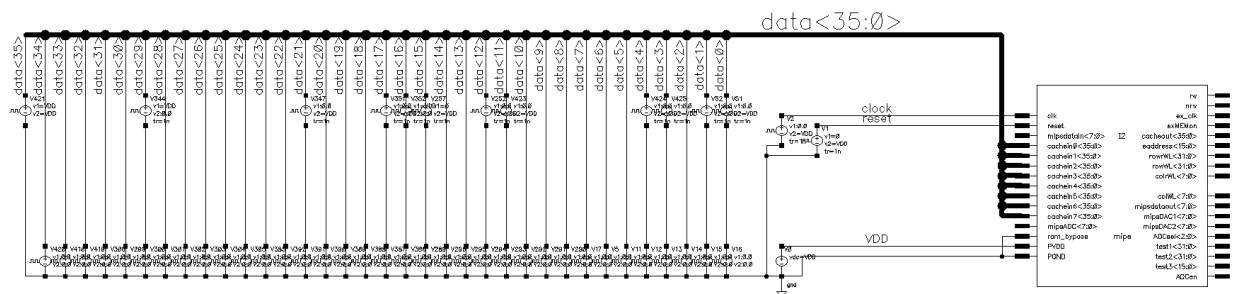


Figure 5.11: Schematic of the power simulation for the 90-nm chip.

kHz and various VDDs is obtained. Due to the complexity of the MIPS design in this analog simulation, each data point takes 1 to 3 days to obtain. Therefore, only a few data points are used in figure 5.10 to represent the curve.

5.7 Testing

The actual physical testing done to support this thesis is limited due to the constraints in time and test complexity. However, the complete simulations and FPGA tests sufficiently show the success of this project, and demonstrates the benefits of a customized ASIC for path-

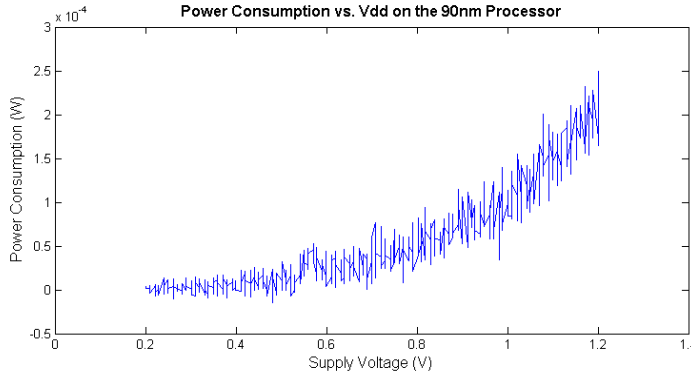


Figure 5.12: ICRCYSUP Chip’s Processor Power Consumption vs. Supply Voltage

planning purpose in achieving superior power, weight and size when compared to general-purpose counter-parts. Figure 5.12, is the actual power measurement on the ICRCYSUP (90nm) chip, where the power consumption curve matches the simulation result very closely.

This result was achieved using the Xilinx Vertex-II Pro FPGA board as a pattern generator, AD8001 600MHz amplifiers to step the voltage down from the FPGA to the chip (and vice versa). The processor is powered using an Agilent 66321B power supply that is able to measure the current draw to $\pm 2.5\mu A$ precision. Other components on the chip and the amplifier array are powered separately using other power supplies. Figure 5.13 shows the general setup for this test, where to the left, are verification equipments to read the outputs as well as to verify and fine-tune the inputs. Beside them, are two power supplies that power components that are not being measured. In the middle, is the FPGA (signal generator), the chip (device-under-test), the amplifier array and all the wiring that routes signals between them. On the right side, is the most important component, which is the power supply that powers the processor and measures its power consumption. The Agilent 66321B power supply is being controlled by MatLab through GPIB interface so that its voltage can be adjusted in thousands of small increments, while taking dozens of measurements at each voltage to produce figure 5.12.

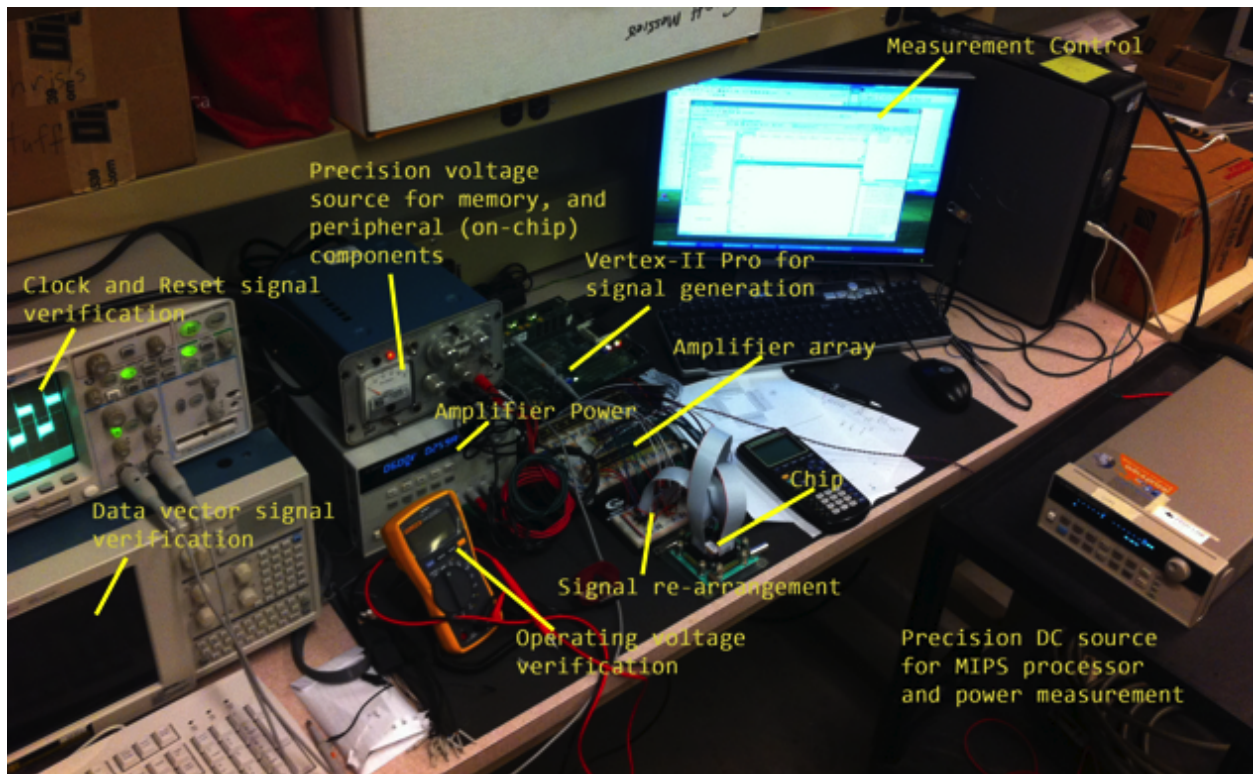


Figure 5.13: Hardware Measurement Test Setup

Chapter 6

CONCLUSION

6.1 Accomplishments

This work has served as a proof of concept CMOS IC implementation of a control algorithm for autonomous system navigation. It considered a unique (and previously described) means of robot navigation embracing control theory and constructed a special purpose silicon processor capable of running it efficiently (i.e. with low power consumption). The algorithm itself presents a means of achieving superior autonomous problem solving ability with minimal need for control flow statements. This suggests the possibility of implementation in a small, low-power machine applicable to physically small agents (e.g. micro-robots).

The realization of this processor required the application of a substantial number of ideas. These included:

1. The translation of the algorithm's operations into a form optimized for digital machine calculation.
2. The verification of the algorithm in a fixed-point arithmetic context.
3. The design of an instruction-set architecture suitable for the efficient implementation of the algorithm with the functionality and the capacity to scale to "bigger" navigation problems.
4. The design of a micro-architecture suitable for the algorithm and the context in which it is expected to run (i.e. real-time navigation of "small" machines).
5. The customization of the processor's datapath (ALU optimization for matrix manipulation, efficient arithmetic decoding, etc.).

6. The design of a suitable memory scheme (caching, memory mapped I/O, etc.).
7. The inclusion of mixed-signal components for interface to sensing and actuation components.
8. The design of on-chip test hardware.
9. Physical integrated circuit design of sub-threshold logic and on-chip cache including custom hand layout.
10. The realization of a simple assembler.
11. The application of a sophisticated VLSI integration methodology using a customized design flow working over a number of disparate IC design tools. This flow further allows the design to be readily translated to more advanced CMOS technologies.
12. Design test and verification over a number of levels of abstraction (high-level programming language, hardware description language, charge-level circuit simulation, FPGA verification).

The size and weight achieved in the 90-nm CMOS chip are $2.08 \times 1.73 \text{ mm}^2$ and 2.86 mg, respectively. The simulated power consumption of this machine is $10 \text{ } \mu\text{W}$ from a 400-mV supply with an estimated computational speed of 1 MIPS leading to a cognitive efficiency of 35,000 MIPS/mg/W. A better than 40% improvement over the state-of-the-art [6].

6.2 Future Work

This work has demonstrated the potential advantages and viability of having a low-power optimized processor for robotics path-planning. From here, there are numerous advancements that could be explored. In the following sections are outlined some major topics worth further investigation.

6.2.1 Customized Library

The chip in this work was constructed using the standard cell CMOS library targeted to operate at regular voltages (1.0 V to 1.2 V). This decision was to minimize the risk of a customized library failing, and to speed up the design process.

A chip re-synthesized using a customized standard cell CMOS library could be expected to have lower minimum operating voltage, higher operating frequency and, most importantly, more energy efficient behaviour.

A library customized for sub-threshold operation should have all the multi-level combinational gates and transmission gates removed from the library because they cause significant delays in sub-threshold operation. Also, sub-threshold characterization of the library is necessary for the synthesizer to optimize the design more efficiently for the targeted operating voltage. There could also be some optimization in transistor sizing and ratioing for optimal, symmetric logic.

6.2.2 Dynamic Tuning

To further reduce energy consumption, several dynamic tuning methods could be employed. These methods include voltage stepping, frequency stepping and sleep mode, and they are often used in modern processors.

Voltage stepping and frequency stepping are the most common methods to reduce power while retaining the capability of high computing performance. This has been proven to be effective in sub-threshold as well in [39]. During times that require higher computing power, the V_{DD} is raised to allow higher operating frequency; and when the processor is doing less work, V_{DD} and frequency can be reduced.

Sleep mode is to reduce the voltage and frequency to its lowest state while retaining necessary data. This requires a software trigger or some intelligent detection, and it is less effective because the chip is already running at a very low voltage. However, it would be

useful to temporarily cut off power to large components such as LUTs when they are not used.

6.2.3 Integration

While a low power processor could increase robotics intelligence and decrease payload, a higher level of integration has similar benefits. Although it makes the resulting design less flexible, it could further reduce size, weight and power. The integration of the A/D converter and D/A converters are an example of this. Future integrations could include flash memories, sensors, voltage dividers, voltage controllers, and even an energy source.

6.2.4 Architecture

The relatively short time horizon for this Master's research work, forced a quick convergence on the question of micro-architecture. However a number of opportunities exist for enhancements. An immediate possibility is the employment of a pipelined machine. Given the nature of the algorithm employed the overhead normally encountered in dealing with hazards should be minimal. As a result, the machine should be able to operate at progressively slower clocks as the number of stages is boosted.

Bibliography

- [1] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, “Conservation cores: Reducing the energy of mature computations,” in *ASPLOS 2010: Architectural Support for Programming Languages and Operating Systems*, Mar. 2010, pp. 1–14.
- [2] S. M. LaValle, *Planning Algorithms*. Cambridge University Press, 2006.
- [3] R. J. Wood, “The first takeoff of a biologically-inspired at-scale robotic insect,” *IEEE Trans. on Robotics*, vol. 24, no. 2, pp. 341–347, Feb. 2008.
- [4] D. M. Harris and S. L. Harris, *Digital Design and Computing Architecture*. Morgan Kaufmann, 2007.
- [5] H. Moravec, “Robots after all,” *Communications of the ACM*, vol. 46, no. 10, pp. 90–97, Oct. 2003.
- [6] B. A. Warneke and K. S. J. Pister, “An ultra-low energy microcontroller for smart dust wireless sensor networks,” in *ISSCC Dig. Tech. Papers*, Feb. 2004, pp. 316–317.
- [7] T. Zourntos, N. J. Mathai, S. Magierowski, and D. Kundur, “A bio-inspired layered analog scheme for navigational control of lightweight autonomous agents,” in *Proc. IEEE Int. Conf. Robo. Autom.*, May 2008, pp. 1132–1137.
- [8] M. Naik, “Mixed mode IC implementation for a robot path planner,” Master’s thesis, University of Calgary, Canada, January 2011.
- [9] T. ur Rehman Butt, “Intelligent navigation for autonomous robots using neural network and fuzzy logic techniques,” Master’s thesis, University of Calgary, Canada, January 2010.

- [10] T. Lei, “An insect like navigation mobile robot based on lyapunov control and fuzzy logic technique,” Master’s thesis, University of Calgary, Canada, January 2012.
- [11] R. A. Brooks, “A robust layered control system for a mobile robot,” *IEEE J. Robotics Automat.*, vol. 2, no. 1, pp. 14–23, Mar. 1986.
- [12] J. A. Farrell and M. M. Polycarpou, *Adaptive Approximation Based Control: Unifying Neural, Fuzzy and Traditional Adaptive Approximation Approaches*. Wiley-Interscience, 2006.
- [13] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools*. Morgan Kaufmann, 2005.
- [14] *MIPS32 Architecture for Programmers Volume II: The MIPS32 Instruction Set*, MIPS Technologies, June 2003, rev. 2.00.
- [15] D. Sweetman, *See MIPS Run*. Morgan Kaufmann, 2007.
- [16] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 4th ed. Morgan Kaufmann, 2008.
- [17] M. M. Mano and C. R. Kime, *Logic and Computer Design Fundamentals*, 4th ed. Prentice Hall, 2008.
- [18] K. D. Cooper and L. Torczon, *Engineering a Compiler*, 2nd ed. Morgan Kaufmann, 2012.
- [19] T. Zourntos and N. J. Mathai, “A beam-inspired lyapunov-based strategy for obstacle avoidance and target-seeking,” in *Proc. American Control Conf.*, July 2007, pp. 5302–5309.
- [20] Y. P. Tsividis and C. McAndrew, *Operation and Modeling of the MOS Transistor*, 3rd ed. New York, NY: Oxford University Press, 2010.

- [21] F. Leuenberger and E. Vittoz, “Complementary-MOS low-power low-voltage integrated binary counter,” *Proc. IEEE*, vol. 57, no. 9, pp. 1528–1532, Sept. 1969.
- [22] R. G. Daniels and R. R. Burgess, “The electronic wristwatch: An application for Si gate CMOS-ICs,” in *ISSCC Dig. Tech. Papers*, Feb. 1971, pp. 62–63.
- [23] J. D. Meindl and R. N. Swanson, “Potential improvements in power-speed performance of digital circuits,” *Proc. IEEE*, vol. 59, no. 5, pp. 815–816, May 1971.
- [24] R. N. Swanson and J. D. Meindl, “Ion-implanted complementary MOS transistors in low-voltage circuits,” *IEEE J. Solid-State Circuits*, vol. 7, no. 2, pp. 146–153, April 1972.
- [25] J. B. Burr and J. Shott, “A 200-mV self-testing encoder/decoder using Stanford ultra-low power CMOS,” in *ISSCC Dig. Tech. Papers*, Feb. 1994, pp. 84–85.
- [26] A. Wang and A. Chandrakasan, “A 180-mV FFT processor using sub-threshold circuit techniques,” in *ISSCC Dig. Tech. Papers*, Feb. 2004, pp. 292–293.
- [27] B. H. Calhoun, A. Wang, and A. Chandrakasan, “Modeling and sizing for minimum energy operation in subthreshold circuits,” *IEEE J. Solid-State Circuits*, vol. 40, no. 9, pp. 1778–1786, Sept. 2005.
- [28] A. C. Joyce Kwong, “Advances in ultra-low-voltage design,” *Solid-State-Circuits Newsletter, IEEE*, vol. 13, no. 44, pp. 20–27, Fall 2008.
- [29] M. Hatamian and G. Cash, “A 70-MHz 8-bit \times 8-bit parallel pipelined multiplier in 2.5- μ m CMOS,” *IEEE J. Solid-State Circuits*, vol. 21, no. 4, pp. 505–513, Aug. 1986.
- [30] C.-H. Lo and S.-Y. Huang, “P-P-N based 10T SRAM cell for low-leakage and resilient subthreshold operation,” *IEEE J. Solid-State Circuits*, vol. 46, no. 3, pp. 695–704, Mar. 2011.

- [31] K. R. Jaydeep P. Kulkarni, Keejong Kim, “A 160mV, fully differential, robust Schmitt trigger based sub-threshold SRAM,” in *Low Power Electronics and Design (ISLPED)*, 2007 ACM/IEEE International Symposium on, Aug. 2007, pp. 171–176.
- [32] B. H. Calhoun and A. P. Chandrakasan, “A 256-kb 65-nm sub-threshold SRAM design for ultra-low-voltage operation,” *IEEE J. Solid-State Circuits*, vol. 42, no. 3, pp. 680–688, Mar. 2007.
- [33] P. E. Allen and D. R. Holberg, *CMOS Analog Circuit Design*, 2nd ed. Oxford University Press, 2002.
- [34] P. J. A. Harpe, C. Zhou, Y. Bi, N. P. van der Meijs, X. Wang, K. Philips, G. Dolmans, and H. de Groot, “A 26 μ W 8 bit 10 MS/s asynchronous SAR ADC for low energy radios,” *IEEE J. Solid-State Circuits*, vol. 47, no. 7, pp. 1585–1595, Jul. 2011.
- [35] L. Wang, C. Stroud, and N. Touba, Eds., *System-on-Chip Test Architectures: Nanometer Design for Testability*. Elsevier, 2008.
- [36] E. Brunvand, *Digital VLSI Chip Design with Cadence and Synopsys CAD Tools*. School of Computing, University of Utah: Addison-Wesley, 2010.
- [37] *Application Note: Backend Digital Design Flow*, Electronics Group, Department of Electrical and Computer Engineering, University of Toronto, 2010.
- [38] W. T. Padgett and D. V. Anderson, *Fixed-Point Signal Processing*. Morgan & Claypool, 2009.
- [39] U. R. Matthias Blesken, Sven Lutkemeier, “Multiobjective optimization for transistor sizing of sub-threshold CMOS logic standard cells,” in *Circuits and Systems (ISCAS), Proceeding of 2010 IEEE International Symposium on*, May 2010, pp. 1480–1483.

Appendix A

Algorithms

A.1 Original Algorithm

The key components of the path planning control algorithm and their transformations are shown below. The *inputs* to the controller consist of its estimated distance and orientation relative to the goal target, r_t and θ_t , respectively. Also required is the robot's measure of its distance and orientation from the nearest obstacle or obstacles, d_r and d_θ , respectively. The elements of ζ are the state variables of a state-space filter preceding the controller. As indicated below, only a 4-state filter was used in this specific example, although larger state-spaces can be used if the application demands it. The purpose of this preceding filter is to smooth-out the response of the machine. Including the filter complicates the design of the Lyapunov controller itself however. Fortunately, a technique known as backstepping allows these two components to be introduced separately. The expressions below include the equations for the combined Lyapunov and backstepping technique. They culminate in the expression for the desired translational and rotational velocity (u_r and u_θ , respectively) commands needed to get the robot to its target.

$$\begin{aligned} input &= \begin{bmatrix} \theta_t & r_t & \zeta_1 & \zeta_2 & \zeta_3 & \zeta_4 & d_r & d_\theta \end{bmatrix}^T \\ \boldsymbol{\eta} &= \begin{bmatrix} r_t & \theta_t \end{bmatrix}^T \\ \boldsymbol{\zeta} &= \begin{bmatrix} \zeta_1 & \zeta_2 & \zeta_3 & \zeta_4 \end{bmatrix}^T \\ \mathbf{B}_{1\boldsymbol{\eta}} &= \begin{bmatrix} -\cos \theta_t & 0 \\ \sin \theta_t / r_t & 1 \end{bmatrix} \end{aligned}$$

$$\mathbf{d} = \begin{bmatrix} \text{rand}(1) \cdot 0.1 \\ \text{rand}(1) \cdot \pi/20 \end{bmatrix}$$

$$k = 1$$

$$k_1 = 0.1$$

$$k_2 = 0.1$$

$$d_{max} = 1$$

$$\dot{\boldsymbol{\zeta}} = \begin{bmatrix} -14 & -100 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & -14 & -100 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \boldsymbol{\zeta} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \cdot (-\boldsymbol{\eta} + \mathbf{d})$$

$$\mathbf{BtP} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}^T \cdot \begin{bmatrix} 0.0361 & 0.0050 & 0 & 0 \\ 0.0050 & 3.6771 & 0 & 0 \\ 0 & 0 & 0.0361 & 0.0050 \\ 0 & 0 & 0.0050 & 3.6771 \end{bmatrix}^T$$

$$\mathbf{BtPz} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}^T \cdot \begin{bmatrix} 0.0361 & 0.0050 & 0 & 0 \\ 0.0050 & 3.6771 & 0 & 0 \\ 0 & 0 & 0.0361 & 0.0050 \\ 0 & 0 & 0.0050 & 3.6771 \end{bmatrix}^T \cdot \boldsymbol{\zeta}$$

$$\mathbf{BtPzd} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}^T \cdot \begin{bmatrix} 0.0361 & 0.0050 & 0 & 0 \\ 0.0050 & 3.6771 & 0 & 0 \\ 0 & 0 & 0.0361 & 0.0050 \\ 0 & 0 & 0.0050 & 3.6771 \end{bmatrix}^T \cdot \dot{\boldsymbol{\zeta}}$$

$$\mathbf{j} = \text{sech}((\mathbf{BtPz})^2) \cdot \mathbf{BtPzd}$$

$$\dot{\boldsymbol{\eta}}_d = k_1 \cdot \mathbf{BtPzd} - \mathbf{j}$$

$$\boldsymbol{\eta}_d = k_1 \cdot \mathbf{BtPz} - \tanh(\mathbf{BtPz}) = \begin{bmatrix} \alpha_r & \alpha_\theta \end{bmatrix}^T$$

$$\begin{aligned}
\epsilon &= \eta - \eta_d \\
\mathbf{dist} &= \begin{bmatrix} d_r & d_\theta \end{bmatrix}^T \\
\mathbf{u} &= \begin{bmatrix} u_r & u_\theta \end{bmatrix}^T = \mathbf{B}_1 \eta^{-1} \cdot (\mathbf{B} \mathbf{t} \mathbf{P} \mathbf{z} - k_2 \epsilon + \dot{\eta}_d) + \mathbf{dist} \\
\mathbf{output} &= \begin{bmatrix} \alpha_r & \alpha_\theta & u_r & u_\theta \end{bmatrix}^T
\end{aligned}$$

A.2 Original Algorithm in Matlab

A MATLAB script implementing the controller calculations outlined above is shown below.

```

function uout = Controller (input);
global zeta;
theta_t = input(1); %---- The robot's relative angle to the target.
r_t     = input(2); %---- The robot's relative distance to the target.
zeta     = input(3:6); %-- The smoothing filter's state variables.
d_r      = input(7); %---- The robot's relative distance to obstacles.
d_theta  = input(8); %---- The robot's relative angle to obstacles.
global eta;
eta = [r_t; theta_t];
global B P A;
k    = 1; %----- Gain factor.
k_1  = 0.1; %----- Gain factor.
k_2  = 0.1; %----- Gain factor.
inv_B_1eta = [-1/cos(theta_t) 0; sin(theta_t)/(cos(theta_t)*r_t) 1]; %----
%----- Inverse of polar-velocity to polar-differential-location
%----- transformation matrix.
global C;
global zetadot
d      = [(rand(1)*0.1); (rand(1)*0.1*pi/2)]; %---- Model of unpredictable sensory
%----- disturbances
zetadot = (A*zeta) + (B*(-eta+d)); %----- Filter state-variable
%----- differential
BtP     = B'*P';
BtPz    = BtP*zeta;
BtPzd   = BtP*zetadot;
dmax    = 1;
global eta_d eta_ddot;
j1 = (sech(BtPz(1))^2)*BtPzd(1);
j2 = (sech(BtPz(2))^2)*BtPzd(2);
j   = [j1;j2];
eta_ddot = k_1*B'*P'*zetadot-(dmax*j);
eta_d    = k_1*B'*P'*zeta-(dmax*tanh(B'*P'*zeta));
alpha_r  = eta_d(1);
alpha_theta = eta_d(2);
global epsilon_epsilon_array;
epsilon   = eta - eta_d;
epsilon_size = size(epsilon_array);
epsilon_array(epsilon_size(1)+1,1) = epsilon(1);
epsilon_array(epsilon_size(1)+1,2) = epsilon(2);
dist      = [d_r ; d_theta]; %----- Robot's polar distance to
%----- obstacles
u         = inv_B_1eta * ( (B'*P*zeta) - (k_2*epsilon) + eta_ddot );
u         = u+dist; %----- Robot's motion polar velocity
%----- commands.
u_r       = u(1);

```

```

u_theta = u(2);
uout=[alpha_r alpha_theta u_r u_theta];

```

A.3 Algorithm in Assembly Language

The assembly language implementation of the controller algorithm is shown below.

0x0000	addi	\$k0	\$0	0032	#k0 = pi/2
0x0001	sll	\$k0	\$k0	000F	
0x0002	addi	\$k0	\$k0	21fb	
0x0003	sll	\$k0	\$k0	0001	
0x0004	lw	\$a0		\$0(00FC)	#load inputs
0x0005	lw	\$a1		\$0(00FD)	
0x0006	lw	\$a6		\$0(00FE)	
0x0007	lw	\$a7		\$0(00FF)	
0x0008	addi	\$t0	\$0	FE40	#t0 = -14
0x0009	sll	\$t0	\$t0	0010	
0x000A	addi	\$t1	\$0	0C80	#t1 = 100
0x000B	sll	\$t1	\$t1	0010	
0x000C	addi	\$t2	\$0	0020	#t2 = 1
0x000D	sll	\$t2	\$t2	0010	
0x000E	addi	\$t3	\$0	0001	#t3 = 0.0361
0x000F	sll	\$t3	\$t3	0010	
0x0010	addi	\$t3	\$t3	27BB	
0x0011	addi	\$t4	\$0	28F6	#t4 = 0.005
0x0012	sec	\$s0	\$k0	\$a0	#s0 = sec(a0)
0x0013	add	\$s1	\$0	\$0	#s1 = 0
0x0014	sin	\$t5	\$k0	\$a0	#t5 = sin(a0)
0x0015	mult	\$t5	\$s0	\$t5	#t5 = sin(a0) * sec(a0) = tan(a0)
0x0016	sub	\$s0	\$0	\$s0	#s0 = -sec(a0)
0x0017	add	\$s2	\$t5	\$0	#s2 = t5/a1 (replaced with binary shifts)
0x0018	add	\$t6	\$a1	\$0	
0x0019	srl	\$t6	\$t6	0010	
0x001A	sll	\$s2	\$s2	0004	#(*64) *4
0x001B	beq	\$t6	\$0	0026	
0x001C	srl	\$t6	\$t6	0001	#(*32) *4
0x001D	srl	\$s2	\$s2	0001	
0x001E	beq	\$t6	\$0	0023	
0x001F	srl	\$t6	\$t6	0001	#(*16) *2
0x0020	srl	\$s2	\$s2	0001	
0x0021	beq	\$t6	\$0	0020	
0x0022	srl	\$t6	\$t6	0001	#(*8) *2
0x0023	srl	\$s2	\$s2	0001	
0x0024	beq	\$t6	\$0	0017	
0x0025	srl	\$t6	\$t6	0001	#(*4)
0x0026	srl	\$s2	\$s2	0001	
0x0027	beq	\$t6	\$0	0014	
0x0028	srl	\$t6	\$t6	0001	#(*2)
0x0029	srl	\$s2	\$s2	0001	
0x002A	beq	\$t6	\$0	0011	
0x002B	srl	\$t6	\$t6	0001	#(1)
0x002C	srl	\$s2	\$s2	0001	
0x002D	beq	\$t6	\$0	0008	

0x002E	srl	\$t6	\$t6	0001	#(/2)
0x002F	srl	\$s2	\$s2	0001	
0x0031	beq	\$t6	\$0	0005	#(/4)
0x0032	srl	\$t6	\$t6	0001	
0x0033	srl	\$s2	\$s2	0001	#(/8)
0x0034	beq	\$t6	\$0	0002	
0x0035	srl	\$t6	\$t6	0001	#s3 = 1
0x0036	srl	\$s2	\$s2	0001	
0x0037	add	\$s3	\$t2	\$0	#s4 = t0*a2 + t1*a3 + t2*a1
0x0038	mult	\$s4	\$t0	\$a2	
0x0039	multa	\$s4	\$t1	\$a3	#s5 = t2*a2
0x003A	multa	\$s4	\$t2	\$a1	
0x003B	mult	\$s5	\$t2	\$a2	#s6 = t0*a4 + t1*a5 + t2*a0
0x003C	mult	\$s6	\$t0	\$a4	
0x003D	multa	\$s6	\$t1	\$a5	#s7 = t2*a4
0x003E	multa	\$s6	\$t2	\$a0	
0x003F	mult	\$s7	\$t2	\$a4	#t5 = t3*a2 + t4*a3
0x0040	mult	\$t5	\$t3	\$a2	
0x0041	multa	\$t5	\$t4	\$a3	#t6 = t3*a4 + t4*a5
0x0042	mult	\$t6	\$t3	\$a4	
0x0043	multa	\$t6	\$t4	\$a5	#s4 = t3*s4 + t4*s5
0x0044	mult	\$s4	\$t3	\$s4	
0x0045	multa	\$s4	\$t4	\$s5	#s5 = t3*s6 + t4*s7
0x0046	mult	\$s5	\$t3	\$s6	
0x0047	multa	\$s5	\$t4	\$s7	#t0 = sech(t5)
0x0048	sech	\$t0	\$0	\$t5	
0x0049	mult	\$t0	\$t0	\$t0	#t1 = t0*s4
0x004A	mult	\$t1	\$t0	\$s4	
0x004B	sech	\$t0	\$0	\$t6	#t0 = sech(t6)
0x004C	mult	\$t0	\$t0	\$t0	
0x004D	mult	\$t2	\$t0	\$s5	#t0 = t0*t0
0x004E	addi	\$t0	\$0	0004	
0x004F	sll	\$t0	\$t0	0010	#t2 = t0*s5
0x0050	mult	\$s6	\$t0	\$s4	
0x0051	sub	\$s6	\$s6	\$t1	#t0 = k1(0.125)
0x0052	mult	\$s7	\$t0	\$s5	
0x0053	sub	\$s7	\$s7	\$t2	#s6 = t0*s4-t1
0x0054	sinh	\$t7	\$0	\$t5	
0x0055	sech	\$t3	\$0	\$t5	#s7 = t0*s5-t2
0x0056	mult	\$t7	\$t7	\$t3	
0x0057	sub	\$t1	\$t5	\$t7	#t7 = tanh(t5)
0x0058	mult	\$t1	\$t1	\$t0	
0x0059	sinh	\$t7	\$0	\$t6	#t1 = t5-t7
0x005A	sech	\$t3	\$0	\$t6	
0x005B	mult	\$t7	\$t7	\$t3	#t1 = t1*t0 (t0 = k1)
					#t7 = tanh(t6)

0x005C	sub	\$t2	\$t6	\$t7	#t2 = t6-t7
0x005D	mult	\$t2	\$t2	\$t0	#t2 = t2*t0 (t0 = k1)
0x005E	sub	\$t3	\$a1	\$t1	#t3 = (a1-t1)*t0
0x005F	mult	\$t3	\$t3	\$t0	
0x0060	sub	\$t4	\$a0	\$t2	#t4 = (a0-t2)*t0
0x0061	mult	\$t4	\$t4	\$t0	
0x0062	sub	\$t1	\$t5	\$t3	#t1 = t5-t3+s6
0x0063	add	\$t1	\$t1	\$s6	
0x0064	sub	\$t2	\$t6	\$t4	#t2 = t6-t4+s7
0x0065	add	\$t2	\$t2	\$s7	
0x0066	mult	\$v0	\$s0	\$t1	#v0 = s0*t1 + s1*t2 + a6
0x0067	multa	\$v0	\$s1	\$t2	
0x0068	add	\$v0	\$v0	\$a6	
0x0069	mult	\$v1	\$s2	\$t1	#v1 = s2*t1 + s3*t2 + a7
0x006A	multa	\$v1	\$s3	\$t2	
0x006B	add	\$v1	\$v1	\$a7	
0x006C	sw	\$v0		\$0(00FA)	
0x006D	sw	\$v1		\$0(00FB)	
0x006E	j			4	

A.4 Algorithm in Machine Code

The machine code description of the controller algorithm is shown below.

```

001000000000111000000000000110010      00000000000100101001000001000010      00000000000100100110000000110111
000000000000111001110001111000000      00000000000101101011000001000010      000000011000110001100000000011000
00100011100111000010000111111011      0001001001000000000000000000010100      00000001100110010111000000011000
00000000000011100111000001000000      00000000000100101001000001000010      00100000000011000000000000000100
10001100000001000000000011111100      00000000000101101011000001000010      000000000001100011001000000000
10001100000001010000000011111101      0001001001000000000000000000010001      00000001100110001101000000011000
10001100000001010000000011111110      00000000000100101001000001000010      00000001101110101101000000100010
10001100000001011000000011111111      00000000000101101011000001000010      00000001100110011101100000011000
0010000000001100111111001000000      0001001001000000000000000000010000      0000000111011011101100000100010
0000000000001100110010000000000      00000000000100101001000001000010      0000000111011011101100000100010
0000000000001100110010000000111111      0001001001000000000000000000010000      00000001110110011101100000011000
0010000000001101000001001000000      00000000000101101011000001000010      00000001001101111001100000011011
00000000000011010110110000000000      00010010010000000000000000000101      0000001001101111001100000011000
0010000000001110000000000100000      00000000000100101001000001000010      00000010011100010110100000100010
0000000000001110011010000000000      00000000000101101011000001000010      00000001101011000110100000011000
00100000000011100000000000000001      0001001001000000000000000000010000      00000000000100101001100000011000
0000000000001110011010000000000      00000000000101101011000001000010      000000011101100111100000011000
0010000111101111001001111011011      0000000000010110100001000000100010      0000000100110111000100000011000
00100000000100000010100011110110      00000001110000001011100000100000      00000001001100100111000000100010
000001110000100101000000110011      00000001100001101100000000011000      0000000111001100011100000011000
0000000000000001010100000100000      0000000110100111100000000011100      0000000110100101011100000100010
00000011100001001000100000110000      00000001110001011100000000011100      0000000111011000111100000011000
0000010100100011000100000011000      0000000110000101100000000011000      0000000111000100100000000100010
0000010010000000101000000100010      00000001101010011101000000011100      000000010010011000000000010000
00000100010000001011000000100000      00000001101010011101000000011100      0000000101010110000100000001100
00000001010000001001000000100000      00000001100001001101000000011100      000000010110100110100000100000
00000000000100101001010000000010      000000011100100011010000000011000      00000001000001001001100000011000
000000000001010101100001000010      0000000111110101100100000011000      0000000101010110000100000001100
000000000001010101100001000010      000000010000010011001000000011100      0000000000010010000100000010000
0001001001000000000000000100011      0000000111110001100000000011000      000000010110011010001100000011000
00000000000100101001000001000010      000000010000110011100000000011100      0000000101101110000010000001100
00000000000101101011000001000010      0000000111110101100100000011000      0000000000011010110000100000010000
00010010010000000000000000100110      0000001000000111000100000011100      000000010100011010001000000011000
00000000000100101001000001000010      0000000111000000000000000011100      00000000000100000000000000000100
00000000000101101011000001000010      000000010000010000101000000011011      00000000000100000000000000000100
00000000000101101011000001000010      00000001100011000110000000011000      00000001100110000110100000011000
0001001001000000000000000010111      000000011000100001101000000011000      00000000000100000000000000000100
00010010010000000000000000010111      00000001100110000110100000011000      00000000000100000000000000000100
00000001100110000110100000011000      00000001100110000110100000011000      00000000000100000000000000000100

```

A.5 Machine Code in Verilog Byte-Addressable Memory

```
external_memory[0] <= 8'b000110010;  
external_memory[1] <= 8'b000000000;  
external_memory[2] <= 8'b000011100;  
external_memory[3] <= 8'b001000000;  
external_memory[4] <= 8'b111000000;  
external_memory[5] <= 8'b111100011;  
external_memory[6] <= 8'b000111100;  
external_memory[7] <= 8'b000000000;  
external_memory[8] <= 8'b111111011;  
external_memory[9] <= 8'b001000011;  
external_memory[10] <= 8'b100111100;  
external_memory[11] <= 8'b001000011;  
external_memory[12] <= 8'b101000000;  
external_memory[13] <= 8'b111100000;  
external_memory[14] <= 8'b000111100;  
external_memory[15] <= 8'b000000000;  
external_memory[16] <= 8'b111111100;  
external_memory[17] <= 8'b000000000;  
external_memory[18] <= 8'b000001010;  
external_memory[19] <= 8'b100011100;  
external_memory[20] <= 8'b111111101;  
external_memory[21] <= 8'b000000000;  
external_memory[22] <= 8'b000001011;  
external_memory[23] <= 8'b100011100;  
external_memory[24] <= 8'b111111110;  
external_memory[25] <= 8'b000000000;  
external_memory[26] <= 8'b000010101;  
external_memory[27] <= 8'b100011100;  
external_memory[28] <= 8'b111111111;  
external_memory[29] <= 8'b000000000;  
external_memory[30] <= 8'b000010111;  
external_memory[31] <= 8'b100011100;  
external_memory[32] <= 8'b010000000;  
external_memory[33] <= 8'b111111110;  
external_memory[34] <= 8'b000011100;  
external_memory[35] <= 8'b001000000;  
external_memory[36] <= 8'b000000000;  
external_memory[37] <= 8'b011001010;  
external_memory[38] <= 8'b000011100;  
external_memory[39] <= 8'b000000000;  
external_memory[40] <= 8'b100000000;  
external_memory[41] <= 8'b000011100;  
external_memory[42] <= 8'b000011011;  
external_memory[43] <= 8'b001000000;  
external_memory[44] <= 8'b000000000;  
external_memory[45] <= 8'b011011100;  
external_memory[46] <= 8'b000011101;  
external_memory[47] <= 8'b000000000;  
external_memory[48] <= 8'b001000000;  
external_memory[49] <= 8'b000000000;  
external_memory[50] <= 8'b000011100;  
external_memory[51] <= 8'b001000000;  
external_memory[52] <= 8'b000000000;  
external_memory[53] <= 8'b011110100;  
external_memory[54] <= 8'b000011110;  
external_memory[55] <= 8'b000000000;  
external_memory[56] <= 8'b000000001;  
external_memory[57] <= 8'b000000000;  
external_memory[58] <= 8'b000011111;  
external_memory[59] <= 8'b001000000;  
external_memory[60] <= 8'b000000000;  
external_memory[61] <= 8'b011111100;  
external_memory[62] <= 8'b000011111;  
external_memory[63] <= 8'b000000000;  
external_memory[64] <= 8'b101111011;  
external_memory[65] <= 8'b001001111;  
external_memory[66] <= 8'b111101111;  
external_memory[67] <= 8'b001000011;  
external_memory[68] <= 8'b111101110;  
external_memory[69] <= 8'b001010000;  
external_memory[70] <= 8'b000100000;  
external_memory[71] <= 8'b001000000;  
external_memory[72] <= 8'b001100111;  
external_memory[73] <= 8'b101000000;  
external_memory[74] <= 8'b100001010;  
external_memory[75] <= 8'b000000011;  
external_memory[76] <= 8'b001000000;  
external_memory[77] <= 8'b101010000;  
external_memory[78] <= 8'b000000000;  
external_memory[79] <= 8'b000000000;  
external_memory[80] <= 8'b001100000;  
external_memory[81] <= 8'b100010000;  
external_memory[82] <= 8'b100001010;  
external_memory[83] <= 8'b000000011;  
external_memory[84] <= 8'b000110000;  
external_memory[85] <= 8'b100010000;  
external_memory[86] <= 8'b100100001;  
external_memory[87] <= 8'b000000010;  
external_memory[88] <= 8'b001000010;  
external_memory[89] <= 8'b101000000;  
external_memory[90] <= 8'b100000000;  
external_memory[91] <= 8'b000000010;  
external_memory[92] <= 8'b001000000;  
external_memory[93] <= 8'b101100000;  
external_memory[94] <= 8'b001000000;  
external_memory[95] <= 8'b000000010;  
external_memory[96] <= 8'b001000000;  
external_memory[97] <= 8'b100100000;  
external_memory[98] <= 8'b101000000;  
external_memory[99] <= 8'b000000000;  
external_memory[100] <= 8'b000000010;  
external_memory[101] <= 8'b100101010;  
external_memory[102] <= 8'b000100101;  
external_memory[103] <= 8'b000000000;  
external_memory[104] <= 8'b000000000;  
external_memory[105] <= 8'b101100011;  
external_memory[106] <= 8'b000101110;  
external_memory[107] <= 8'b000000000;  
external_memory[108] <= 8'b001001101;  
external_memory[109] <= 8'b000000000;  
external_memory[110] <= 8'b010000000;  
external_memory[111] <= 8'b000100101;  
external_memory[112] <= 8'b010000010;  
external_memory[113] <= 8'b100101000;  
external_memory[114] <= 8'b000100101;  
external_memory[115] <= 8'b000000000;  
external_memory[116] <= 8'b010000010;  
external_memory[117] <= 8'b101100000;  
external_memory[118] <= 8'b000101110;  
external_memory[119] <= 8'b000000000;  
external_memory[120] <= 8'b001000011;  
external_memory[121] <= 8'b000000000;  
external_memory[122] <= 8'b010000000;  
external_memory[123] <= 8'b000100101;  
external_memory[124] <= 8'b010000010;  
external_memory[125] <= 8'b100101000;  
external_memory[126] <= 8'b000100101;  
external_memory[127] <= 8'b000000000;  
external_memory[128] <= 8'b010000010;  
external_memory[129] <= 8'b101100000;  
external_memory[130] <= 8'b000101110;  
external_memory[131] <= 8'b000000000;  
external_memory[132] <= 8'b001000000;  
external_memory[133] <= 8'b000000000;  
external_memory[134] <= 8'b010000000;  
external_memory[135] <= 8'b000100101;  
external_memory[136] <= 8'b010000010;  
external_memory[137] <= 8'b100101000;  
external_memory[138] <= 8'b000100101;  
external_memory[139] <= 8'b000000000;  
external_memory[140] <= 8'b010000010;  
external_memory[141] <= 8'b101100000;  
external_memory[142] <= 8'b000101110;  
external_memory[143] <= 8'b000000000;  
external_memory[144] <= 8'b000101111;  
external_memory[145] <= 8'b000000000;  
external_memory[146] <= 8'b010000000;  
external_memory[147] <= 8'b000100101;  
external_memory[148] <= 8'b010000010;  
external_memory[149] <= 8'b100101000;  
external_memory[150] <= 8'b000100101;  
external_memory[151] <= 8'b000000000;  
external_memory[152] <= 8'b010000010;  
external_memory[153] <= 8'b101100000;  
external_memory[154] <= 8'b000101110;  
external_memory[155] <= 8'b000000000;  
external_memory[156] <= 8'b000101010;  
external_memory[157] <= 8'b000000000;  
external_memory[158] <= 8'b010000000;  
external_memory[159] <= 8'b000100101;  
external_memory[160] <= 8'b010000010;  
external_memory[161] <= 8'b1001
```


Appendix B

Scripts and Programs

B.1 MATLAB Processor

```
function uout = Controller (input)
global memory reg regfile;
global accumulator i_addr clk pclk;
p1 = 10;
p2 = 21;
clk = input(9);

%input arguments to the function
a0 = qno(input(1),2,5);%theta_t
a1 = qno(input(2),2,5);%r_t
a2 = 0;%qno(input(3),0,7);%zeta(1)
a3 = 0;%qno(input(4),0,7);%zeta(2)
a4 = 0;%qno(input(5),0,7);%zeta(3)
a5 = 0;%qno(input(6),0,7);%zeta(4)
a6 = qno(input(7),3,4);%d_r
a7 = qno(input(8),3,4);%d_theta

reg(5) = a0*2^21;
reg(6) = a1*2^21;
reg(7) = a2*2^21;
reg(8) = a3*2^21;
reg(9) = a4*2^21;
reg(10) = a5*2^21;
reg(11) = a6*2^21;
reg(12) = a7*2^21;

% bin2dec(mem(100,0,'read'))
% bin2dec(condense(memory(101,:)))
% bin2dec(condense(memory(102,:)))
% bin2dec(condense(memory(103,:)))
% bin2dec(condense(memory(104,:)))
% bin2dec(condense(memory(105,:)))
% bin2dec(condense(memory(106,:)))
% bin2dec(condense(memory(107,:)))

if (clk && ~pclk)
    %i = 0;
    %while i_addr <= 107
    %i = i+1;
    %fetch instruction
    instruction = condense(memory(i_addr,:));
    %PC increment
    i_addr = mod(i_addr,99)+1;
    %decode
    op = instruction(1:6);
    rs = bin2dec(instruction(7:11));
    rt = bin2dec(instruction(12:16));
    rd = bin2dec(instruction(17:21));
    shamt = bin2dec(instruction(22:26));
    funct = instruction(27:32);
    imm = b2d(instruction(17:32));

    %decimal calculation with 2^21 offset
    switch op
        %R-Type
        case '000000' %op, rs, rt, rd, sa func
            switch funct
                case '100000' %add
                    reg(rd) = reg(rs) + reg(rt);
                case '100010' %sub
                    reg(rd) = reg(rs) - reg(rt);
                case '011000' %mult
                    reg(rd) = reg(rs) * reg(rt)/2^21;
            case '011100' %multa
                    reg(rd) = reg(rs) * reg(rt) * reg(rt)/2^21;
            case '000000' %sll
                    reg(rd) = reg(rt) * 2^shamt;
            case '000010' %srl
                    reg(rd) = fix(reg(rt) / 2^shamt);
            case '110000'%sin
                    reg(rd) = LUT_sin(reg(rt));
            case '110001'%cos
                    reg(rd) = LUT_cos(reg(rt));
            case '110010'%csc = 1/sin
                    reg(rd) = LUT_csc(reg(rt));
            case '110011'%sec = 1/cos
                    reg(rd) = LUT_sec(reg(rt));
            case '110100'%sinh
                    reg(rd) = LUT_sinh(reg(rt));
            case '110101'%cosh
                    reg(rd) = LUT_cosh(reg(rt));
            case '110110'%csch = 1/sinh
                    reg(rd) = LUT_csch(reg(rt));
            case '110111'%sech = 1/cosh
                    reg(rd) = LUT_sech(reg(rt));
            case '111000'%
                    reg(rd) = -LUT_sin(reg(rt));
            case '111001'%
                    reg(rd) = -LUT_cos(reg(rt));
            case '111010'%
                    reg(rd) = -LUT_csc(reg(rt));
            case '111011'%
                    reg(rd) = -LUT_sec(reg(rt));
            case '111100'%
                    reg(rd) = -LUT_sinh(reg(rt));
            case '111101'%
                    reg(rd) = -LUT_cosh(reg(rt));
            case '111110'%
                    reg(rd) = -LUT_csch(reg(rt));
            case '111111'%
                    reg(rd) = -LUT_sech(reg(rt));
            end
        end
    end

    %I-Type
    case '001000' %addi
        reg(rt) = reg(rs) + imm;
    case '000100' %beq
        if reg(rs) == reg(rt)
            i_addr = i_addr+imm;
        end
    end

    end

    clc;

    v0 = qno(bin2dec1021(dec2bin1021(reg(3)/2^21)),1,6);
    v1 = qno(bin2dec1021(dec2bin1021(reg(4)/2^21)),2,5);

    regindex = size(regfile);
    regindex = regindex(1,1);
    regfile(regindex+i,:) = reg;

    uout=[v0 v1];
    pclk = clk;
```

B.2 Perl Assembler

```
#!/usr/local/bin/perl
```

```

%funct = ( "add"   => "100000",
           "and"   => "100100",
           "or"    => "100101",
           "mult"  => "011000",
           "multu" => "011001",
           "multa" => "011100",
           "mfhi"  => "010000",
           "mflo"  => "010010",
           "sll"   => "000000",
           "srl"   => "000010",
           "sra"   => "000011",
           "addi"  => "000000",
           "sub"   => "100010",
           "sin"   => "110000",
           "cos"   => "110001",
           "csc"   => "110010",
           "sec"   => "110011",
           "sinh"  => "110100",
           "cosh"  => "110101",
           "csch"  => "110110",
           "sech"  => "110111",
           "nsin"  => "111000",
           "ncos"  => "111001",
           "ncsc"  => "111010",
           "nsec"  => "111011",
           "nsinh" => "111100",
           "ncosh" => "111101",
           "ncsch" => "111110",
           "nsech" => "111111",

           "beq"   => "000000",
           "lw"    => "000000",
           "sw"    => "000000",

           "j"     => "000000");

```

```

%opcode = ( "add"   => "000000",
            "and"   => "000000",
            "or"    => "000000",
            "mult"  => "000000",
            "multu" => "000000",
            "multa" => "000000",
            "mfhi"  => "000000",
            "mflo"  => "000000",
            "sll"   => "000000",
            "srl"   => "000000",
            "sra"   => "000000",
            "addi"  => "001000",
            "sub"   => "000000",
            "sin"   => "000000",
            "cos"   => "000000",
            "csc"   => "000000",
            "sec"   => "000000",
            "sinh"  => "000000",
            "cosh"  => "000000",
            "csch"  => "000000",
            "sech"  => "000000",
            "nsin"  => "000000",
            "ncos"  => "000000",
            "ncsc"  => "000000",
            "nsec"  => "000000",
            "nsinh" => "000000",
            "ncosh" => "000000",
            "ncsch" => "000000",
            "nsech" => "000000",

            "beq"   => "000100",
            "lw"    => "100011",
            "sw"    => "101011",

            "j"     => "000010");

```

```

%reg = ( "\$0"   => "00000",
         "\$1"   => "00001",
         "\$2"   => "00010",
         "\$3"   => "00011",
         "\$4"   => "00100",
         "\$5"   => "00101",
         "\$6"   => "00110",
         "\$7"   => "00111",
         "\$8"   => "01000",
         "\$9"   => "01001",
         "\$10"  => "01010",
         "\$11"  => "01011",
         "\$12"  => "01100",
         "\$13"  => "01101",
         "\$14"  => "01110",
         "\$15"  => "01111",
         "\$16"  => "10000",
         "\$17"  => "10001",

```



```

        print wFILE "$reg{$instruction[2]}";
        print wFILE "$reg{$instruction[1]}";
        printf wFILE "%05b", hex($instruction[3]);
        print wFILE "$funct{$instruction[0]}\n";
    }
    elsif (($instruction[0] eq "mfhi") or ($instruction[0] eq "mflo")){
        print wFILE "00000";
        print wFILE "00000";
        print wFILE "$reg{$instruction[1]}";
        print wFILE "00000";
        print wFILE "$funct{$instruction[0]}\n";
    }
}
}
close(wFILE);
close(rFILE);

open rFILE, "bench.tmp" or die $!;
open wFILE, ">bench.dat" or die $!;
while (my $line = <rFILE>) {
    my @instruction = split(undef,$line);
    # print "$line";

    my $dec = @instruction[24]*8 + @instruction[25]*4 + @instruction[26]*2 + @instruction[27]*1;
    printf wFILE "%x",$dec;
    $dec = @instruction[28]*8 + @instruction[29]*4 + @instruction[30]*2 + @instruction[31]*1;
    printf wFILE "%x\n",$dec;
    my $dec = @instruction[16]*8 + @instruction[17]*4 + @instruction[18]*2 + @instruction[19]*1;
    printf wFILE "%x",$dec;
    $dec = @instruction[20]*8 + @instruction[21]*4 + @instruction[22]*2 + @instruction[23]*1;
    printf wFILE "%x\n",$dec;
    my $dec = @instruction[8]*8 + @instruction[9]*4 + @instruction[10]*2 + @instruction[11]*1;
    printf wFILE "%x",$dec;
    $dec = @instruction[12]*8 + @instruction[13]*4 + @instruction[14]*2 + @instruction[15]*1;
    printf wFILE "%x\n",$dec;
    my $dec = @instruction[0]*8 + @instruction[1]*4 + @instruction[2]*2 + @instruction[3]*1;
    printf wFILE "%x",$dec;
    $dec = @instruction[4]*8 + @instruction[5]*4 + @instruction[6]*2 + @instruction[7]*1;
    printf wFILE "%x\n",$dec;
}
close(wFILE);
close(rFILE);

open rFILE, "bench.tmp" or die $!;
open wFILE, ">bench.data" or die $!;
$adr = 0;
while (my $line = <rFILE>) {
    my @instruction = split(undef,$line);
    print "$line";

    print wFILE "        external_memory[$adr] <= 8'b";
    printf wFILE "%i%i%i%i",@instruction[24],@instruction[25],@instruction[26],@instruction[27];
    printf wFILE "%i%i%i%i;\n",@instruction[28],@instruction[29],@instruction[30],@instruction[31];
    $adr = $adr + 1;
    print wFILE "        external_memory[$adr] <= 8'b";
    printf wFILE "%i%i%i%i",@instruction[16],@instruction[17],@instruction[18],@instruction[19];
    printf wFILE "%i%i%i%i;\n",@instruction[20],@instruction[21],@instruction[22],@instruction[23];
    $adr = $adr + 1;
    print wFILE "        external_memory[$adr] <= 8'b";
    printf wFILE "%i%i%i%i",@instruction[8],@instruction[9],@instruction[10],@instruction[11];
    printf wFILE "%i%i%i%i;\n",@instruction[12],@instruction[13],@instruction[14],@instruction[15];
    $adr = $adr + 1;
    print wFILE "        external_memory[$adr] <= 8'b";
    printf wFILE "%i%i%i%i",@instruction[0],@instruction[1],@instruction[2],@instruction[3];
    printf wFILE "%i%i%i%i;\n",@instruction[4],@instruction[5],@instruction[6],@instruction[7];
    $adr = $adr + 1;
}
close(wFILE);
close(rFILE);

```

B.3 Synopsys Compiler Script

```

*****
** By: Ryan Wu University of Calgary          *
** Modified from: Erik Brunvand              *
** syn-script                                *
*****
#Environment Setup (.synopsys_dc.setup content)
set company {University of Calgary}
set designer {Ryan Wu}

```

```

set SynopsysInstall [getenv "SYNOPSYS"]
define_design_lib WORK -path ./WORK

#####
#Design specific Parameters
#####
set myFiles [list ./NC_Verilog/lut.v ./NC_Verilog/mips.v ./NC_Verilog/memcontrol.v ./NC_Verilog/core.v ./NC_Verilog/dft.v];# all
set fileFormat verilog; # verilog or VHDL
set basename mips; # Top level module name
set myClk clk; # The name of your clk
set virtual 0; # 1 if virtual clock (for design w/ no clk), 0 if real clock

#####Timing and loading information#####
set myPeriod_ns 100; # desired clock period in ns (sets speed goal)
set myInDelay_ns 0.25; # delay from clock to input valid
set myOutDelay_ns 0.25; # delay from clock to output valid
set myInputBuf INVX4TF; # name of the cell driving the inputs
set myLoadLibrary scx3_cmos8rf_rvt_tt_1p2v_25c; # name of the library the cell comes from
set myLoadPin A; # name of pin that the outputs drive
set myDrivePin Y; # name of the that drives the inputs

#####Control the writing of result files#####
set runname struct; #name appended to output files

#the following control which output files you want. They should be set to 1 if you want the file, 0 if not
set write_v 1; # compiled structural Verilog file
set write_ddc 1; # compiled file in ddc format
set write_sdf 0; # sdf file for back-annotated timing sum
set write_sdc 1; # sdc time constraint file for place and route
set write_rep 1; # report file for compilation
set write_pow 1; # report file for power generation

#####compiler switches#####
set useUltra 1; # 1 for compile_ultra, 0 for compile
#mapEffort, use Ungroup are for non-ultra compile
set mapEffort1 medium; # First pass - low, medium, or high
set mapEffort2 medium; # second pass - low, medium or high
set useUngroup 1; # 0, if no flatten, 1 if flatten

#####
# Set some system-level things #
# #
# Your library path may be empty if your library will be in #
# your synthesis directory because "." is already on the path #
#####
set search_path [list . \
./NC_Verilog \
[format "%s%s" $SynopsysInstall /libraries/syn] \
[format "%s%s" $SynopsysInstall /dw/sim_ver] \
]

#Target library list should include all target .db files
#Library names separated by spaces if more than one
set target_library [list ./TECH/scx3_cmos8rf_rvt_tt_1p2v_25c.db]

set synthetic_library [list dw_foundation.sldb]

set symbol_library [list generic.sdb]

set link_library [concat \
[concat "*" $target_library] $synthetic_library]

```

```
#####
# Below here should not be changed... #
#####

#analyse and elaborate the files
analyze -format $fileFormat -lib WORK $myFiles
elaborate $basename -lib WORK -update
current_design $basename

# The link command makes sure that all the required design
#parts are linked together
# The uniquify command makes unique copies of replicated
#modules
link
uniquify

# now you can create clocks for the design
#and set other constraints
if { $virtual == 0 } {
    create_clock -period $myPeriod_ns $myClk
} else {
    create_clock -period $myPeriod_ns -name $myClk
}

# Set the driving cell for all inputs except the clock
# The clock has infinite drive by default. This is usually
# what you want for synthesis because you will use other
#tools (like SOC Encounter) to build the clock tree
# (or define it by hand)
if { $virtual == 0 } {
    set_driving_cell -library $myLoadLibrary -lib_cell $myInputBuf -pin $myDrivePin [all_inputs] \
} else {
    set_driving_cell -library $myLoadLibrary -lib_cell $myInputBuf -pin $myDrivePin \
        [remove_from_collection [all_inputs] $myClk]
}

#set the input and output delay relative to myClk
if { $virtual == 0 } {
    set_input_delay $myInDelay_ns -clock $myClk [all_inputs] \
} else {
    set_input_delay $myInDelay_ns -clock $myClk \
        [remove_from_collection [all_inputs] $myClk]
}
set_output_delay $myOutDelay_ns -clock $myClk [all_outputs]

#set the load of the circuit outputs in terms of load
#of the next cell that they will drive, also try to fix
#hold time issues
set_load [load_of [format "%s%s%s%s" $myLoadLibrary "/" $myInputBuf "/" $myLoadPin]] [all_outputs]
set_fix_hold $myClk

#This command will fix the problem of having assign statements left in your structural file
# But, it will insert pairs of inverters for feedthroughs!
set_fix_multiple_port_nets -all -buffer_constants

#now compile the design with given mapping effort
#and do a second compile with incremental mapping
# or use the the compile_ultra meta-command
if { $useUltra == 1 } {
    compile_ultra
} else {
    if { $useUngroup == 1 } {
        compile -ungroup_all -map_effort $mapEffort1 #The instruction in the book is ungoup_all; typo?
        compile -incremental_mapping -map_effort $mapEffort2
    } else {
        compile -map_effort $mapEffort1
    }
}
```

```

    compile -incremental_mapping -map_effort $mapEffort2
}
}

#
check_design
report_constraint -all_violators

#####
# now write out the results                                     #
#####

set filebase [format "%s%s" [format "%s%s" $basename "_" $runname]

# structural (synthesised) file as verilog
if { $write_v == 1 } {
    set filename [format "%s%s" $filebase ".v"]
    write -format verilog -hierarchy -output $filename
    redirect change_names \
{ change_names -rules verilog -hierarchy -verbose }
    write -format verilog -hierarchy -output [format "%s%s" $filebase ".sdf"]
}

# write out the sdf file for back-annotated verilog sim
# This file can be large!
if { $write_sdf == 1 } {
    set filename [format "%s%s" $filebase ".sdf"]
    write_sdf -version 1.0 $filename
}

# This is the timings constraints file generated from the
# conditions above -used in the place and route program
if { $write_sdc == 1 } {
    set filename [format "%s%s" $filebase ".sdc"]
    write_sdc $filename
}

# synopsys database format in case you want to read this
# synthesized result back in to synopsys later in XG mode (ddc format)
if { $write_ddc == 1 } {
    set filename [format "%s%s" $filebase ".ddc"]
    write -format ddc -hierarchy -o $filename
}

# report on the results from synthesis
# note that > makes a new file and >> appends to a file
if { $write_rep == 1 } {
    set filename [format "%s%s" $filebase ".rep"]
    redirect $filename { report_timing }
    redirect -append $filename { report_area }
}

# report the power estimate from synthesis
if { $write_pow == 1 } {
    set filename [format "%s%s" $filebase ".pow"]
    redirect $filename { report_power }
}

quit

```

B.4 Encounter Top-Level Command Script

```
#####
# Encounter Top-Level Script for Layout Synthesis      #
# By: Ryan Wu                                         #
# Modified from Erik Brunvand                         #
#####

# set the BASENAME for the config files. This will also
# be used for the .lib, .lef, .v, and .spef files
# that are generated by this script
set BASENAME "mips"
set PROCESS 130;

# set the name of the filler cells - you don't need a list
# if you only have one
set fillerCells [list FILL8TF FILL64TF FILL4TF FILL32TF FILL2TF FILL1TF FILL16TF]

# choose numbers that make sense for you
set usepct 0.50 ;# percent utilization in placing cells
set rowgap 3    ;# gap between pairs of std cell rows
set aspect 0.5 ;# aspect ratio of overall cell (1.0 is square)
# less than 1.0 is landscape, greater than 1.0 is portrait

#####
# You may not have to change things below this line - but check!
#####
set clockBufName <INVX1TF> ;# Footprint of inverter in .lib file

# Set some values that define the power rings and stripes.
# use these defaults, or choose your own.
# Note that all these numbers should be divisible by 3 so
# that they fit on the lambda grid
set pwidth 15 ;# power rail width
set pspace 3  ;# power rail space
set swidth 6  ;# power stripe width
set sspace 2  ;# power stripe spacing
set sset 7    ;# number of power stripe
set soffset 60 ;# power stripe offset to first stripe
set coregap 39.0 ;# gap between the core and the power rails

# Import design and floorplan
# If the config file is not named $BASENAME.conf, edit this line.
loadConfig $BASENAME.conf 0
commitConfig

# source the files that operate on the circuit
source fplan.tcl ;# create the floorplan (might be done by hand...)
win
source pplan.tcl ;# create the power rings and stripes
win
source place.tcl ;# Place the cells and optimize (pre-CTS)
win
source cts.tcl ;# Create the clock tree, and optimize (post-CTS)
win
source route.tcl ;# Route the design using nanoRoute
win
source verify.tcl ;# Verify the design and produce output files
#exit
```

B.5 Layout Synthesis Configurations

```
#####
# MIPS Synthesis Configuration file                  #
```

```

# By: Ryan Wu                                     #
# Modifications from Erik Brunvand                #
#####
global rda_Input
#
#####
# Here are the parts you need to update for your design
#####
#
# Your input is structural verilog. Set the top module name
# and also give the .sdc file you used in synthesis for the
# clock timing constraints.
set rda_Input(ui_netlist)      {mips_struct.v}
set rda_Input(ui_topcell)      {mips}
set rda_Input(ui_timingcon_file) {mips_struct.sdc}
set rda_Input(ui_io_file)      {mips.io}
#
# Leave min and max empty if you have only one timing library
# (space-separated if you have more than one)
set rda_Input(ui_timelib)      {./TECH/scx3_cmos8rf_rvt_tt_1p2v_25c.lib ./TECH/iogpil_cmrf8sf_rvt_tt_1p2v_2p5v_25c.lib}
set rda_Input(ui_timelib,min)  {./TECH/scx3_cmos8rf_rvt_ff_1p65v_125c.lib ./TECH/iogpil_cmrf8sf_rvt_ff_1p65v_2p7v_125c.lib}
set rda_Input(ui_timelib,max)  {./TECH/scx3_cmos8rf_rvt_ss_1p08v_m55c.lib ./TECH/iogpil_cmrf8sf_rvt_ss_1p08v_2p3v_m55c.lib}
#
# Set the name of your lef file or files
# (space-separated if you have more than one).
set rda_Input(ui_leffile) {./TECH/ibm13_8lm_2thick_3rf_tech.lef ./TECH/ibm13rfrvt_macros.lef ./TECH/iogpil_cmrf8sf_rvt_M2_3_1e}
#
# Include the footprints of your cells that fit these uses. Delay
# can be an inverter or a buffer. Leave buf blank if you don't
# have a non-inverting buffer. These are the "footprints" in
# the .lib file, not the cell names.
#buf/inv are now automatically identified, use "loadFootPrint" command to override
##set rda_Input(ui_buf_footprint)      {BUF2TF}
##set rda_Input(ui_delay_footprint)     {DLY1X1TF}
##set rda_Input(ui_inv_footprint)       {INV1X1TF}
##set rda_Input(ui_cts_cell_footprint) {}
#
#####
# You might want to set core utilization and core_to spacing
# or you can leave these defaults...
#####
#
set rda_Input(ui_core_util) {0.5}
set rda_Input(ui_core_to_left) {30.0}
set rda_Input(ui_core_to_right) {30.0}
set rda_Input(ui_core_to_top) {30.0}
set rda_Input(ui_core_to_bottom) {30.0}
#
#####
# Below here you should be able to leave alone...
#####
set rda_Input(import_mode) {-treatUndefinedCellAsBbox 0 -keepEmptyModule 1 }
set rda_Input(ui_netlisttype) {Verilog}
set rda_Input(ui_settop) {1}
set rda_Input(ui_core_cntl) {aspect}
set rda_Input(ui_aspect_ratio) {0.4}
set rda_Input(ui_isVerTrackHalfPitch) {1}
set rda_Input(ui_ioOri) {R180}
set rda_Input(ui_delay_limit) {1000}
set rda_Input(ui_net_delay) {1000.0ps}
set rda_Input(ui_net_load) {0.5pf}
set rda_Input(ui_in_tran_delay) {120.0ps}
set rda_Input(ui_defcap_scale) {1.0}
set rda_Input(ui_detcap_scale) {1.0}
set rda_Input(ui_xcap_scale) {1.0}
set rda_Input(ui_preRoute_res) {1.0}
set rda_Input(ui_postRoute_res) {1.0}
set rda_Input(ui_shr_scale) {1.0}

```

```

set rda_Input(ui_time_unit) {none}
set rda_Input(flip_first) {1}
set rda_Input(double_back) {1}
set rda_Input(assign_buffer) {1}
set rda_Input(ui_pwrnet) {VDD}
set rda_Input(ui_gndnet) {VSS}
set rda_Input(ui_pg_connections) [list \
    {PIN:VSS:} \
    {PIN:VDD:} \
    {NET:VSS:} \
    {NET:VDD:} \
    {TIEHI::} \
    {TIELO::} \
]
set rda_Input(PIN:VSS:) {VSS}
set rda_Input(PIN:VDD:) {VDD}
set rda_Input(NET:VSS:) {VSS}
set rda_Input(NET:VDD:) {VDD}
set rda_Input(TIEHI::) {VDD}
set rda_Input(TIELO::) {VSS}

```

B.6 Synthesis Pin Placement

```

#Pin Assignments and Placement for Synthesis
#By Ryan Wu, Feb. 2011

# W = West|Left,    bottom to top
#
# E = East|right,   bottom to top
#
# N = North|top,    left to right
#
# S = South|bottom, left to right
#

Pin: clk          W
Pin: reset        W
Pin: rw           W
Pin: ex_clk       W
Pin: ex_MEMon     W
Pin: ram_bypass   W
Pin: exdatain[0]  S
Pin: exdatain[1]  S
Pin: exdatain[2]  S
Pin: exdatain[3]  S
Pin: exdatain[4]  S
Pin: exdatain[5]  S
Pin: exdatain[6]  S
Pin: exdatain[7]  S
Pin: exdataout[0] S
Pin: exdataout[1] S
Pin: exdataout[2] S
Pin: exdataout[3] S
Pin: exdataout[4] S
Pin: exdataout[5] S
Pin: exdataout[6] S
Pin: exdataout[7] S

Pin: test1[0]     E
Pin: test1[1]     E
Pin: test1[2]     E
Pin: test1[3]     E
Pin: test1[4]     E

Pin: test1[5]     E
Pin: test1[6]     E
Pin: test1[7]     E
Pin: test1[8]     E
Pin: test1[9]     E
Pin: test1[10]    E
Pin: test1[11]    E
Pin: test1[12]    E
Pin: test1[13]    E
Pin: test1[14]    E
Pin: test1[15]    E
Pin: test1[16]    E
Pin: test1[17]    E
Pin: test1[18]    E
Pin: test1[19]    E
Pin: test1[20]    E
Pin: test1[21]    E
Pin: test1[22]    E
Pin: test1[23]    E
Pin: test1[24]    E
Pin: test1[25]    E
Pin: test1[26]    E
Pin: test1[27]    E
Pin: test1[28]    E
Pin: test1[29]    E
Pin: test1[30]    E
Pin: test1[31]    E
Pin: test2[0]     E
Pin: test2[1]     E
Pin: test2[2]     E
Pin: test2[3]     E
Pin: test2[4]     E
Pin: test2[5]     E
Pin: test2[6]     E
Pin: test2[7]     E
Pin: test2[8]     E
Pin: test2[9]     E
Pin: test2[10]    E
Pin: test2[11]    E
Pin: test2[12]    E

Pin: test2[13]    E
Pin: test2[14]    E
Pin: test2[15]    E
Pin: test2[16]    E
Pin: test2[17]    E
Pin: test2[18]    E
Pin: test2[19]    E
Pin: test2[20]    E
Pin: test2[21]    E
Pin: test2[22]    E
Pin: test2[23]    E
Pin: test2[24]    E
Pin: test2[25]    E
Pin: test2[26]    E
Pin: test2[27]    E
Pin: test2[28]    E
Pin: test2[29]    E
Pin: test2[30]    E
Pin: test2[31]    E
Pin: test3[0]     E
Pin: test3[1]     E
Pin: test3[2]     E
Pin: test3[3]     E
Pin: test3[4]     E
Pin: test3[5]     E
Pin: test3[6]     E
Pin: test3[7]     E
Pin: test3[8]     E
Pin: test3[9]     E
Pin: test3[10]    E
Pin: test3[11]    E
Pin: test3[12]    E
Pin: test3[13]    E
Pin: test3[14]    E
Pin: test3[15]    E

Pin: colWL[0]     N
Pin: colrWL[0]    N
Pin: rowrWL[0]    N
Pin: rowWL[0]     N

```


Pin: rowrWL[1]	N	Pin: cacheout[20]	N	Pin: colWL[1]	N
Pin: rowrWL[1]	N	Pin: cacheout[3]	N	Pin: colrWL[1]	N
Pin: rowrWL[2]	N	Pin: cacheout[21]	N	Pin: cachein1[0]	N
Pin: rowrWL[2]	N	Pin: cacheout[4]	N	Pin: cachein1[18]	N
Pin: rowrWL[3]	N	Pin: cacheout[22]	N	Pin: cachein1[1]	N
Pin: rowrWL[3]	N	Pin: cacheout[5]	N	Pin: cachein1[19]	N
Pin: rowrWL[4]	N	Pin: cacheout[23]	N	Pin: cachein1[2]	N
Pin: rowrWL[4]	N	Pin: cacheout[6]	N	Pin: cachein1[20]	N
Pin: rowrWL[5]	N	Pin: cacheout[24]	N	Pin: cachein1[3]	N
Pin: rowrWL[5]	N	Pin: cacheout[7]	N	Pin: cachein1[21]	N
Pin: rowrWL[6]	N	Pin: cacheout[25]	N	Pin: cachein1[4]	N
Pin: rowrWL[6]	N	Pin: cacheout[8]	N	Pin: cachein1[22]	N
Pin: rowrWL[7]	N	Pin: cacheout[26]	N	Pin: cachein1[5]	N
Pin: rowrWL[7]	N	Pin: cacheout[9]	N	Pin: cachein1[23]	N
Pin: rowrWL[8]	N	Pin: cacheout[27]	N	Pin: cachein1[6]	N
Pin: rowrWL[8]	N	Pin: cacheout[10]	N	Pin: cachein1[24]	N
Pin: rowrWL[9]	N	Pin: cacheout[28]	N	Pin: cachein1[7]	N
Pin: rowrWL[9]	N	Pin: cacheout[11]	N	Pin: cachein1[25]	N
Pin: rowrWL[10]	N	Pin: cacheout[29]	N	Pin: cachein1[8]	N
Pin: rowrWL[10]	N	Pin: cacheout[12]	N	Pin: cachein1[26]	N
Pin: rowrWL[11]	N	Pin: cacheout[30]	N	Pin: cachein1[9]	N
Pin: rowrWL[11]	N	Pin: cacheout[13]	N	Pin: cachein1[27]	N
Pin: rowrWL[12]	N	Pin: cacheout[31]	N	Pin: cachein1[10]	N
Pin: rowrWL[12]	N	Pin: cacheout[14]	N	Pin: cachein1[28]	N
Pin: rowrWL[13]	N	Pin: cacheout[32]	N	Pin: cachein1[11]	N
Pin: rowrWL[13]	N	Pin: cacheout[15]	N	Pin: cachein1[29]	N
Pin: rowrWL[14]	N	Pin: cacheout[33]	N	Pin: cachein1[12]	N
Pin: rowrWL[14]	N	Pin: cacheout[16]	N	Pin: cachein1[30]	N
Pin: rowrWL[15]	N	Pin: cacheout[34]	N	Pin: cachein1[13]	N
Pin: rowrWL[15]	N	Pin: cacheout[17]	N	Pin: cachein1[31]	N
Pin: rowrWL[16]	N	Pin: cacheout[35]	N	Pin: cachein1[14]	N
Pin: rowrWL[16]	N			Pin: cachein1[32]	N
Pin: rowrWL[17]	N	Pin: cachein0[0]	N	Pin: cachein1[15]	N
Pin: rowrWL[17]	N	Pin: cachein0[18]	N	Pin: cachein1[33]	N
Pin: rowrWL[18]	N	Pin: cachein0[1]	N	Pin: cachein1[16]	N
Pin: rowrWL[18]	N	Pin: cachein0[19]	N	Pin: cachein1[34]	N
Pin: rowrWL[19]	N	Pin: cachein0[2]	N	Pin: cachein1[17]	N
Pin: rowrWL[19]	N	Pin: cachein0[20]	N	Pin: cachein1[35]	N
Pin: rowrWL[20]	N	Pin: cachein0[3]	N		
Pin: rowrWL[20]	N	Pin: cachein0[21]	N	Pin: colWL[2]	N
Pin: rowrWL[21]	N	Pin: cachein0[4]	N	Pin: colrWL[2]	N
Pin: rowrWL[21]	N	Pin: cachein0[22]	N	Pin: cachein2[0]	N
Pin: rowrWL[22]	N	Pin: cachein0[5]	N	Pin: cachein2[18]	N
Pin: rowrWL[22]	N	Pin: cachein0[23]	N	Pin: cachein2[1]	N
Pin: rowrWL[23]	N	Pin: cachein0[6]	N	Pin: cachein2[19]	N
Pin: rowrWL[23]	N	Pin: cachein0[24]	N	Pin: cachein2[2]	N
Pin: rowrWL[24]	N	Pin: cachein0[7]	N	Pin: cachein2[20]	N
Pin: rowrWL[24]	N	Pin: cachein0[25]	N	Pin: cachein2[3]	N
Pin: rowrWL[25]	N	Pin: cachein0[8]	N	Pin: cachein2[21]	N
Pin: rowrWL[25]	N	Pin: cachein0[26]	N	Pin: cachein2[4]	N
Pin: rowrWL[26]	N	Pin: cachein0[9]	N	Pin: cachein2[22]	N
Pin: rowrWL[26]	N	Pin: cachein0[27]	N	Pin: cachein2[5]	N
Pin: rowrWL[27]	N	Pin: cachein0[10]	N	Pin: cachein2[23]	N
Pin: rowrWL[27]	N	Pin: cachein0[28]	N	Pin: cachein2[6]	N
Pin: rowrWL[28]	N	Pin: cachein0[11]	N	Pin: cachein2[24]	N
Pin: rowrWL[28]	N	Pin: cachein0[29]	N	Pin: cachein2[7]	N
Pin: rowrWL[29]	N	Pin: cachein0[12]	N	Pin: cachein2[25]	N
Pin: rowrWL[29]	N	Pin: cachein0[30]	N	Pin: cachein2[8]	N
Pin: rowrWL[30]	N	Pin: cachein0[13]	N	Pin: cachein2[26]	N
Pin: rowrWL[30]	N	Pin: cachein0[31]	N	Pin: cachein2[9]	N
Pin: rowrWL[31]	N	Pin: cachein0[14]	N	Pin: cachein2[27]	N
Pin: rowrWL[31]	N	Pin: cachein0[32]	N	Pin: cachein2[10]	N
		Pin: cachein0[15]	N	Pin: cachein2[28]	N
Pin: cacheout[0]	N	Pin: cachein0[33]	N	Pin: cachein2[11]	N
Pin: cacheout[18]	N	Pin: cachein0[16]	N	Pin: cachein2[29]	N
Pin: cacheout[1]	N	Pin: cachein0[34]	N	Pin: cachein2[12]	N
Pin: cacheout[19]	N	Pin: cachein0[17]	N	Pin: cachein2[30]	N
Pin: cacheout[2]	N	Pin: cachein0[35]	N		

Pin: cachein2[13] N
Pin: cachein2[31] N
Pin: cachein2[14] N
Pin: cachein2[32] N
Pin: cachein2[15] N
Pin: cachein2[33] N
Pin: cachein2[16] N
Pin: cachein2[34] N
Pin: cachein2[17] N
Pin: cachein2[35] N

Pin: colWL[3] N
Pin: colrWL[3] N
Pin: cachein3[0] N
Pin: cachein3[18] N
Pin: cachein3[1] N
Pin: cachein3[19] N
Pin: cachein3[2] N
Pin: cachein3[20] N
Pin: cachein3[3] N
Pin: cachein3[21] N
Pin: cachein3[4] N
Pin: cachein3[22] N
Pin: cachein3[5] N
Pin: cachein3[23] N
Pin: cachein3[6] N
Pin: cachein3[24] N
Pin: cachein3[7] N
Pin: cachein3[25] N
Pin: cachein3[8] N
Pin: cachein3[26] N
Pin: cachein3[9] N
Pin: cachein3[27] N
Pin: cachein3[10] N
Pin: cachein3[28] N
Pin: cachein3[11] N
Pin: cachein3[29] N
Pin: cachein3[12] N
Pin: cachein3[30] N
Pin: cachein3[13] N
Pin: cachein3[31] N
Pin: cachein3[14] N
Pin: cachein3[32] N
Pin: cachein3[15] N
Pin: cachein3[33] N
Pin: cachein3[16] N
Pin: cachein3[34] N
Pin: cachein3[17] N
Pin: cachein3[35] N

Pin: colWL[4] N
Pin: colrWL[4] N
Pin: cachein4[0] N
Pin: cachein4[18] N
Pin: cachein4[1] N
Pin: cachein4[19] N
Pin: cachein4[2] N
Pin: cachein4[20] N
Pin: cachein4[3] N
Pin: cachein4[21] N
Pin: cachein4[4] N
Pin: cachein4[22] N
Pin: cachein4[5] N
Pin: cachein4[23] N
Pin: cachein4[6] N
Pin: cachein4[24] N
Pin: cachein4[7] N
Pin: cachein4[25] N
Pin: cachein4[8] N

Pin: cachein4[26] N
Pin: cachein4[9] N
Pin: cachein4[27] N
Pin: cachein4[10] N
Pin: cachein4[28] N
Pin: cachein4[11] N
Pin: cachein4[29] N
Pin: cachein4[12] N
Pin: cachein4[30] N
Pin: cachein4[13] N
Pin: cachein4[31] N
Pin: cachein4[14] N
Pin: cachein4[32] N
Pin: cachein4[15] N
Pin: cachein4[33] N
Pin: cachein4[16] N
Pin: cachein4[34] N
Pin: cachein4[17] N
Pin: cachein4[35] N

Pin: colWL[5] N
Pin: colrWL[5] N
Pin: cachein5[0] N
Pin: cachein5[18] N
Pin: cachein5[1] N
Pin: cachein5[19] N
Pin: cachein5[2] N
Pin: cachein5[20] N
Pin: cachein5[3] N
Pin: cachein5[21] N
Pin: cachein5[4] N
Pin: cachein5[22] N
Pin: cachein5[5] N
Pin: cachein5[23] N
Pin: cachein5[6] N
Pin: cachein5[24] N
Pin: cachein5[7] N
Pin: cachein5[25] N
Pin: cachein5[8] N
Pin: cachein5[26] N
Pin: cachein5[9] N
Pin: cachein5[27] N
Pin: cachein5[10] N
Pin: cachein5[28] N
Pin: cachein5[11] N
Pin: cachein5[29] N
Pin: cachein5[12] N
Pin: cachein5[30] N
Pin: cachein5[13] N
Pin: cachein5[31] N
Pin: cachein5[14] N
Pin: cachein5[32] N
Pin: cachein5[15] N
Pin: cachein5[33] N
Pin: cachein5[16] N
Pin: cachein5[34] N
Pin: cachein5[17] N
Pin: cachein5[35] N

Pin: colWL[6] N
Pin: colrWL[6] N
Pin: cachein6[0] N
Pin: cachein6[18] N
Pin: cachein6[1] N
Pin: cachein6[19] N
Pin: cachein6[2] N
Pin: cachein6[20] N
Pin: cachein6[3] N
Pin: cachein6[21] N

Pin: cachein6[4] N
Pin: cachein6[22] N
Pin: cachein6[5] N
Pin: cachein6[23] N
Pin: cachein6[6] N
Pin: cachein6[24] N
Pin: cachein6[7] N
Pin: cachein6[25] N
Pin: cachein6[8] N
Pin: cachein6[26] N
Pin: cachein6[9] N
Pin: cachein6[27] N
Pin: cachein6[10] N
Pin: cachein6[28] N
Pin: cachein6[11] N
Pin: cachein6[29] N
Pin: cachein6[12] N
Pin: cachein6[30] N
Pin: cachein6[13] N
Pin: cachein6[31] N
Pin: cachein6[14] N
Pin: cachein6[32] N
Pin: cachein6[15] N
Pin: cachein6[33] N
Pin: cachein6[16] N
Pin: cachein6[34] N
Pin: cachein6[17] N
Pin: cachein6[35] N

Pin: colWL[7] N
Pin: colrWL[7] N
Pin: cachein7[0] N
Pin: cachein7[18] N
Pin: cachein7[1] N
Pin: cachein7[19] N
Pin: cachein7[2] N
Pin: cachein7[20] N
Pin: cachein7[3] N
Pin: cachein7[21] N
Pin: cachein7[4] N
Pin: cachein7[22] N
Pin: cachein7[5] N
Pin: cachein7[23] N
Pin: cachein7[6] N
Pin: cachein7[24] N
Pin: cachein7[7] N
Pin: cachein7[25] N
Pin: cachein7[8] N
Pin: cachein7[26] N
Pin: cachein7[9] N
Pin: cachein7[27] N
Pin: cachein7[10] N
Pin: cachein7[28] N
Pin: cachein7[11] N
Pin: cachein7[29] N
Pin: cachein7[12] N
Pin: cachein7[30] N
Pin: cachein7[13] N
Pin: cachein7[31] N
Pin: cachein7[14] N
Pin: cachein7[32] N
Pin: cachein7[15] N
Pin: cachein7[33] N
Pin: cachein7[16] N
Pin: cachein7[34] N
Pin: cachein7[17] N
Pin: cachein7[35] N

B.7 Floor Planning

```
#####
# Layout Floorplanning (density,aspect ratio,spacing)#
# By: Ryan Wu                                         #
# Modified from Erik Brunvand                       #
#####
puts "-----Floorplanning-----"
#
# Make a floorplan - this works fine for projects that are all
# standard cells and include no blocks that need hand placement...
setDrawView fplan
setFPlanRowSpacingAndType $rowgap 2
floorPlan -site IBM13SITE -r $aspect $usepct $coregap $coregap $coregap $coregap
#setRoutingStyle -top -style m
fit

#
# Save design so far
saveDesign ${BASENAME}_fplan.enc
saveFPlan ${BASENAME}.fp
puts "-----Floorplanning done-----"
```

B.8 Power Planning

```
#####
# Power Rings and Power Rails                         #
# By: Ryan Wu                                         #
# Modified from Erik Brunvand                       #
#####
puts "-----Power Planning-----"
puts "-----Making power rings-----"
#
# Make power and ground rings
# $pwidth microns wide with $pspace spacing between them
# and centered in the channel
addRing -spacing_bottom $pspace -width_left $pwidth -width_bottom $pwidth \
-width_top $pwidth -spacing_top $pspace -layer_bottom M1 -center 1 \
-stacked_via_top_layer MA -width_right $pwidth -around core \
-jog_distance $pspace -offset_bottom $pspace -layer_top M1 \
-threshold $pspace -offset_left $pspace -spacing_right $pspace \
-spacing_left $pspace -offset_right $pspace -offset_top $pspace \
-layer_right M2 -nets {VSS VDD} -stacked_via_bottom_layer M1 \
-layer_left M2

#
puts "-----making power stripes-----"
#
# Make Power Stripes. This step is optional. If you keep it in remember to
# check the stripe spacing (set-to-set-distance = $sspace)
# and stripe offset (xleft-offset = $soffset))
addStripe -block_ring_top_layer_limit M3 -max_same_layer_jog_length 8.0 \
-snap_wire_center_to_grid Grid -padcore_ring_bottom_layer_limit M1 \
-number_of_sets $sset -stacked_via_top_layer MA \
-padcore_ring_top_layer_limit M3 -spacing $sspace -xleft_offset $soffset -xright_offset $soffset \
-merge_stripes_value 0.2 -layer M2 -block_ring_bottom_layer_limit M1 \
-width $swidth -nets {VSS VDD} -stacked_via_bottom_layer M1

# Save the design so far
saveDesign ${BASENAME}_pplan.enc
puts "-----Power Planning done-----"
```

B.9 Cell Placement

```
#####
# Standard Cell Placement                                     #
# By: Ryan Wu                                                #
# Modified from Erik Brunvand                                #
#####
puts "-----Placing Cells-----"

# Place the standard cells
setPlaceMode -timingdriven 1 -reorderScan 1 -congEffort high -doCongOpt 0 -ModulePlan 1 -modulePlan 1 \
    -doCongOpt 1 -clkGateAware 1 -maxRouteLayer 3

# Only turn the optimizations in if needed. We'll do more optimization later
setDesignMode -process ${PROCESS}
#this is the one that takes a long time
placeDesign -inPlaceOpt -prePlaceOpt
#placeDesign
setDrawView place

# Now run the first optimization step - pre-CTS (Clock Tree Synthesis)
# in-place optimization.
setOptMode -yieldEffort none
setOptMode -effort high
setOptMode -leakagePowerEffort high
setOptMode -dynamicPowerEffort high
setOptMode -maxDensity 0.65
setOptMode -drcMargin 0.0
setOptMode -holdTargetSlack 0.0 -setupTargetSlack 0.0
setOptMode -simplifyNetlist false
clearClockDomains
#setClockDomains -all is not supported in common timing engine (CTE)
setClockDomains -all
setOptMode -usefulSkew false
optDesign -preCTS -drv \
    -outDir ${BASENAME}_reports/preCTSOptTimingReports

# Save the design so far
saveDesign ${BASENAME}_placed.enc
puts "-----Done Placing Cells-----"
```

B.10 Clock-Tree Synthesis

```
#####
# Clock Tree Synthesis                                       #
# By: Ryan Wu                                                #
# Modified from Erik Brunvand                                #
#####
puts "-----Clock Tree Synthesis-----"

# Create the clock tree spec from the .sdc file
#createClockTreeSpec -output $BASENAME.ctstch -invFootprint $clockBufName

specifyClockTree -file Clock.ctstch

# Use -useCTSRouteGuide to use routing guide during CTS.
setCTSMODE -routeGuide true

# Set routeClkNet to use Nanoroute during CTS.
setCTSMODE -routeClkNet true

setCTSMODE -synthLatencyEffort high
setCTSMODE -opt true
setCTSMODE -optAddBuffer true
```

```

setCTSMODE -optArea true
setCTSMODE -optLatency true
setCTSMODE -optLatencyMoveGate true
setCTSMODE -postCTSCloningForTNS true
setCTSMODE -powerAware true
setCTSMODE -traceDPinAsLeaf false
setCTSMODE -traceIoPinAsLeaf false

# Perform clocktree synthesis
clockDesign -specFile Clock.ctstch -outDir ${BASENAME}_clock_reports

# Run the second optimization - post-CTS
setOptMode -yieldEffort none
setOptMode -effort high
setOptMode -leakagePowerEffort high
setOptMode -dynamicPowerEffort high
setOptMode -maxDensity 0.8
setOptMode -drcMargin 0.0
setOptMode -holdTargetSlack 0.0 -setupTargetSlack 0.0
setOptMode -simplifyNetlist false
clearClockDomains
setClockDomains -all
setOptMode -usefulSkew false
optDesign -postCTS -drv \
    -outDir ${BASENAME}_reports/postCTS0pt

# Save the design so far
saveDesign ${BASENAME}_cts.enc
puts "-----Clock Tree Synthesis done-----"

```

B.11 Routing

B.12 Verification

```

#####
# Geometry and Connectivity Verification #
# By: Ryan Wu #
# Modified from Erik Brunvand #
#####
puts "-----Verifying and File Output-----"
#
# Verify the connectivity and geometry
verifyConnectivity -type regular -error 50 -warning 50 \
    -report ${BASENAME}_Conn_regular.rpt
verifyGeometry -report ${BASENAME}_Geom.rpt

puts "-----Metal Fill-----"

puts "-----Output ${BASENAME}.def file-----"
# Export the DEF, v, spef, sdf, lef, and lib files
global dbgLefDefOutVersion
set dbgLefDefOutVersion 5.5
defOut -floorplan -netlist -routing $BASENAME.def
saveDesign ${BASENAME}_done.enc -def

puts "-----Output ${BASENAME}_soc.v file-----"
saveNetlist [format "%s_soc.v" $BASENAME]

puts "-----Save models for hierarchical flow-----"
saveModel -dir ${BASENAME}_hier_data

```

```
extractRC -outfile $BASENAME.cap
rcOut -spef $BASENAME.spef
write_sdf -ideal_clock_network $BASENAME.sdf

do_extract_model -blackbox_2d -force ${BASENAME}_soc.lib

# Generate timing model for PrimeTime
writeTimingCon -filename ${BASENAME}_done

puts "-----Verify and file output done-----"
```

Appendix C

Verilog Codes

C.1 Core

```
//-----  
// Chip core interconnects  
//  
// By: Ryan Wu, University of Calgary  
// Last Modified: April 30, 2012  
//  
// This component specifies the interconnects between all  
// the internal components within the core, which includes  
// the MIPS process, SRAM, test circuits,  
// and some analog components.  
//-----  
  
// top level design includes both mips processor and memory  
module core (clk, reset, ex_clk, write, read, eaddress, exMEMon, exdataout, exdatain, test_clk, test_en, test_load, test_sel, test_out1, test_out2);  
  
    supply1 RAM_VDD;  
    supply0 RAM_GND;  
  
    input  clk, reset;  
    output ex_clk, write, read, exMEMon;  
    output [7:0]  exdataout;  
    input  [7:0]  exdatain;  
    output [15:0] eaddress;  
    input  test_clk, test_en, test_load;  
    input  [3:0] test_sel;  
    output test_out1, test_out2;  
  
    wire      MEMbp, ADCen;  
    wire [2:0] ADCsel;  
    wire [7:0] colWL, colrWL, mipsDAC1, mipsDAC2, mipsADC, DAC1out, DAC2out, ADCin, mipsdatain, mipsdataout;  
    wire [31:0] rowWL, rowrWL, test1, test2;  
    wire [15:0] test3;  
    wire [35:0] cacheout, cachein0, cachein1, cachein2, cachein3, cachein4, cachein5, cachein6, cachein7;  
  
    // instantiate the mips processor  
    mips mips(clk, reset, mipsdatain, cachein0, cachein1, cachein2, cachein3, cachein4, cachein5, cachein6, cachein7, write, read, ex_clk, exMEMon, cacheout, eaddress, rowrWL, rowWL, test1, test2, test3, cachein0, cachein1, cachein2, cachein3, cachein4, cachein5, cachein6, cachein7, test_out1, test_out2, test_sel, test_en, test_load, test_clk);  
  
    dft dft(test_clk, test_load, test_en, test_out1, test_out2, test_sel, test1, test2, test3, cachein0, cachein1, cachein2, cachein3, cachein4, cachein5, cachein6, cachein7, test_out1, test_out2, test_sel, test_en, test_load, test_clk);  
  
    // SRAM Cache Controller  
    memcontrol cache_controller(clk, reset, write, memen, adr, writedata, exdatain, cachein0, cachein1, cachein2, cachein3, cachein4, cachein5, cachein6, cachein7, halt, test_out1, test_out2, test_sel, test_en, test_load, test_clk);  
  
    // SRAM 256W (1024 Bytes)
```

```

RAM_256W_schematic cache(cachein0, cachein1, cachein2, cachein3, cachein4, cachein5, cachein6, cachein7, RAM_GND, RAM_VDD, cacheout, reset, colWL, colrWL, rowWL, rowrWL);

ana_mux4      MUX(vin, ADCsel[1:0], vout);
a2d_ideal     ADC(vin, clk, ADCen, ADCin);
dac_8bit_ideal DAC1(DAC1out, vout);
dac_8bit_ideal DAC2(DAC2out, vout);
endmodule

```

C.2 MIPS

```

//-----
// MIPS processor
//
// By: Ryan Wu (Modified from "simple MIPS" by Erik Brunvand)
// Last Modified: April 30, 2012
//
// This is the full modified-MIPS processor design used for
// the final synthesis and fabricated in TSMC90nm
//-----

// simplified MIPS processor
// Top level module
module mips #(parameter WIDTH = 32, REGBITS = 5)
    (input          clk, reset,
     input  [7:0]    mipsdatain,
     input  [35:0]   cachein0, cachein1, cachein2, cachein3, cachein4, cachein5, cachein6, cachein7,
     output         rw, nrw, ex_clk, exMEMon,
     output [35:0]   cacheout,
     output [15:0]   eaddress,
     output [31:0]   rowrWL, rowWL,
     output [7:0]    colrWL, colWL,
     output [7:0]    mipsdataout, mipsDAC1, mipsDAC2,
     input  [7:0]    mipsADC,
     input          ram_bypass,
     output [2:0]    ADCsel,
     output [31:0]   test1, test2,
     output [15:0]   test3,
     output         ADCen,
     input          VDD, GND);

    wire [31:0] instr, LUT_out, ALU_out, src1, src2, memdata, adr, writedata;
    wire [3:0]  alucont;
    wire [1:0]  aluop, pcsrc, alusrcb, modcount;
    wire        zero, alusrca, memtoreg, iord, pcen, regwrite, regdst, aluinit, aluen, done,
                sign, luten, multa_route, irwrite, halt, memen;

    controller cont(clk, reset, instr[31:26], instr[5:0], zero, done, halt, memen, rw,
                    alusrca, memtoreg, iord, pcen, multa_route, regwrite, regdst,
                    irwrite, pcsrc, alusrcb, aluop);

    alucontrol ac(clk, reset, sign, aluop, instr[5:0], instr[10:6], alucont[3:0], modcount, done, aluinit, aluen, luten);

```



```

alu      #(WIDTH) alunit(clk, aluinit, aluen, src2, src1, alucont, sign, ALU_out);

lutcontrol  trig_luts(luten, instr[3:0], modcount, ALU_out, LUT_out);

datapath  #(WIDTH, REGBITS)
          dp(clk, reset, memdata, ALU_out, LUT_out, alusrca, memtoreg, iord, luten,
            irwrite, pcen, multa_route, regwrite, regdst, pcsrc, alusrcb,
            zero, instr, adr, writedata, src1, src2);

assign nrw    = ~rw;
assign test1 = writedata;
assign test2 = memdata;
assign test3 = adr[15:0];

// SRAM Cache Controller
memcontrol cache_controller(clk, reset, ram_bypass, rw, memen, adr[15:0], writedata, mipsdatain, mipsADC, cachein0, cachein1, cachein2, cachein3, cachein4, cachein5, c

endmodule

// Finite state machine controller
module controller(input clk, reset,
                 input      [5:0] op, funct,
                 input      zero, done, halt,
                 output      memen,
                 output reg   memwrite, alusrca, memtoreg, iord,
                 output      pcen, multa_route,
                 output reg   regwrite, regdst, irwrite,
                 output reg [1:0] pcsource, alusrcb, aluop);

parameter  FETCH   = 4'b0000;
parameter  DECODE  = 4'b0001;

parameter  MEMADR  = 4'b0010;
parameter  MEMRD   = 4'b0011;
parameter  MEMWRB  = 4'b0100;
parameter  MEMWR   = 4'b0101;

parameter  RTYPEEX = 4'b0110;
parameter  ALUWRB  = 4'b0111;

parameter  BEQEX   = 4'b1000;
parameter  ADDIEX  = 4'b1001;
parameter  ADDIWR  = 4'b1010;
parameter  JUMP    = 4'b1011;
parameter  OVFLOW  = 4'b1101;
parameter  UNDEF   = 4'b1100;
parameter  FETCHw  = 4'b1110;

parameter  MULTAEX = 4'b1111;

parameter  LW      = 6'b100011;
parameter  SW      = 6'b101011;
parameter  RTYPE   = 6'b0;

```

```

parameter  ADDI    = 6'b001000;
parameter  BEQ     = 6'b000100;
parameter  J       = 6'b000010;

reg [3:0] state, nextstate;
reg      pcwrite, branch, memread;
assign memen = memread | memwrite;

// state register
always @(posedge clk)
    if(reset) state <= FETCH;
    else state <= nextstate;

// next state logic
always @(*)
    begin
        case(state)
            FETCH: begin
                if (~halt)
                    nextstate <= FETCHw;
                else
                    nextstate <= FETCH;
            end
            FETCHw: nextstate <= DECODE;
            DECODE: case(op)
                LW:      nextstate <= MEMADR;
                SW:      nextstate <= MEMADR;
                RTYPE:   nextstate <= RTYPEEX;
                BEQ:     nextstate <= BEQEX;
                J:       nextstate <= JUMP;
                ADDI:    nextstate <= ADDIEX;
                default: nextstate <= UNDEF; // should never happen
            endcase
            MEMADR: case(op)
                LW:      nextstate <= MEMRD;
                SW:      nextstate <= MEMWR;
                default: nextstate <= UNDEF; // should never happen
            endcase
            MEMRD: begin
                if (~halt)
                    nextstate <= MEMWRB;
                else
                    nextstate <= MEMRD;
            end
            MEMWR: begin
                if (~halt)
                    nextstate <= FETCH;
                else
                    nextstate <= MEMWR;
            end
            MEMWRB: nextstate <= FETCH;

            RTYPEEX: begin

```

```

        if (done) begin
            if (multa_route)
                nextstate <= MULTAEX;
            else
                nextstate <= ALUWRB;
            end
        else
            nextstate <= RTYPEEX;
        end
    end

MULTAEX: nextstate <= ALUWRB;
ALUWRB: nextstate <= FETCH;
ADDIEX: nextstate <= ADDIWR;
ADDIWR: nextstate <= FETCH;
BEQEX: nextstate <= FETCH;
JUMP: nextstate <= FETCH;
OVFLOW: nextstate <= FETCH;
UNDEF: nextstate <= FETCH;
default: nextstate <= FETCH; // should never happen
endcase
end

assign multa_route = done & ~state[3] & state[2] & state[1] & ~state[0] &
    ~funct[5] & funct[4] & funct[3] & funct[2] & ~funct[1] & ~funct[0];

always @(*)
begin
    // set all outputs to zero, then conditionally assert
    // just the appropriate ones
    irwrite <= 1'b0;
    pcwrite <= 0; branch <= 0;
    regwrite <= 0; regdst <= 0;
    memread <= 0; memwrite <= 0;
    alusrca <= 0; alusrca <= 2'b00; aluop <= 2'b00;
    pcsource <= 2'b00;
    iord <= 0; memtoreg <= 0;
    case(state)
        FETCH:
            begin
                memread <= 1; //activate memcontroller
                iord <= 0; //Select PC Register
                alusrca <= 0; //Source A = PC
                alusrca <= 2'b01; //Source B = 4
                aluop <= 2'b00; //A + B = next fetch address
                pcsource <= 2'b00; //PC Source = PC + 4
            end
        FETCHw:
            begin
                iord <= 0; //Select PC Register
                alusrca <= 0; //Source A = PC
                alusrca <= 2'b01; //Source B = 4
                aluop <= 2'b00; //A + B = next fetch address
                pcsource <= 2'b00; //PC Source = PC + 4
                irwrite <= 1; //update instruction reg
                pcwrite <= 1; //update PC Register for next fetch
            end
    end
end

```

```

DECODE:
begin
    alusrca <= 0;      //Source A = PC
    alusrcb <= 2'b11;  //Source B = Sign extended Imm
    aluop   <= 2'b00;  //A + B = branch precalculation
end

MEMADR:
begin
    alusrca <= 1;      //Source A = Register 1
    alusrcb <= 2'b10;  //Source B = Sign extended Imm
    aluop   <= 2'b00;  //A + B = Memory Address
end

MEMRD:
begin
    memread <= 1;      //activate memcontroller
    iord    <= 1;      //calculated memory address
end

MEMWRB:
begin
    regdst  <= 0;      //set reg writeback address to $rs
    regwrite <= 1;      //enable reg writeback
    memtoreg <= 1;      //select memory data for writeback
end

MEMWR:
begin
    memwrite <= 1;      //enable memory write
    iord    <= 1;      //select MEMADR calculated address
end

RTYPEEX:
begin
    alusrca <= 1;      //Source A = Register 1
    alusrcb <= 2'b00;  //Source B = Register 2
    aluop   <= 2'b10;  //op determined by funct
end

MULTAEX:
begin
    alusrca <= 1;      //Source A = Register 1
    alusrcb <= 2'b00;  //Source B = Register 2
    aluop   <= 2'b00;  //add
end

ALUWRB:
begin
    regdst  <= 1;      //set reg writeback address to $rd
    regwrite <= 1;      //enable reg writeback
    memtoreg <= 0;      //select ALU result for writeback
end

ADDIEX:
begin
    alusrca <= 1;      //Source A = Register 1
    alusrcb <= 2'b10;  //Source B = sign extended Imm
    aluop   <= 2'b00;  //A + B
end

ADDIWR:

```

```

        begin
            regdst <= 0;        //Select $rt for write back address
            regwrite <= 1;      //Enable register write back
            memtoreg <= 0;      //select ALU result for writeback
        end
    BEQEX:
        begin
            alusrca <= 1;       //Source A = Register 1
            alusrcb <= 2'b00;   //Source B = Register 2
            aluop <= 2'b01;     //A-B
            branch <= 1;        //Enable branch if 0
            pcsource <= 2'b01;
        end
    JUMP:
        begin
            pcwrite <= 1;       //Update PC Register
            pcsource <= 2'b10;   //Select Jump Address
        end
    endcase
end
assign pcen = pcwrite | (branch & zero); // program counter enable
endmodule

// Datapath, including register file, ALU, muxes, and other registers
module datapath #(parameter WIDTH = 32, REGBITS = 5)
    (input          clk, reset,
     input  [WIDTH-1:0] memdata, aluout, lutout0,
     input          alusrca, memtoreg, iord, luten, irwrite,
     input          pcen, multa_route, regwrite, regdst,
     input  [1:0]    pcsource, alusrcb,
     output         zero,
     output  [31:0]  instr,
     output  [WIDTH-1:0] adr, writedata, src1, src2);

    wire [REGBITS-1:0] readaddr1, readaddr2, wa;
    wire [WIDTH-1:0]   pc, nextpc, md, readdata1, readdata2, wd, a, result, Lresult,
                    IMM, IMMx4, data1;

    assign IMM  = {{16{instr[15]}},instr[15:0]};
    assign IMMx4 = {IMM[29:0],2'b00};

    // register file address fields
    assign readaddr1 = instr[REGBITS+20:21];
    mux2    #(REGBITS) RegWrite_MUX(instr[REGBITS+15:16],
                                     instr[REGBITS+10:11], regdst, wa);

    // independent of bit width, load instruction into four
    // 32-bit registers over four cycles
    dffn    #(32)    Instruction_reg(clk, irwrite, memdata[31:0], instr[31:0]);

    // datapath
    dffn    #(WIDTH) pc_reg(clk, reset, pcen, nextpc, pc);
    dff     #(WIDTH) mem2reg_reg(clk, memdata, md);
    dff     #(WIDTH) regout1_reg(clk, data1, a);

```

```

dff      #(WIDTH)  regout2_reg(clk, readdata2, writedata);
dff      #(WIDTH)  result_reg(clk, result, Lresult);
mux2     #(WIDTH)  addr_src(pc, Lresult, iord, adr);
mux2     #(WIDTH)  srcAmux(pc, a, alusrca, src1);
mux4     #(WIDTH)  srcBmux(writedata, {29'b0,3'b001}, IMM,
                        IMM, alusrcb, src2);
mux4     #(WIDTH)  pcmux(result, Lresult, {pc[31:26],instr[25:0]}, 32'b0,
                        pcsource, nextpc);
mux2     #(WIDTH)  write2regmux(Lresult, md, memtoreg, wd);
mux2     #(WIDTH)  outselect(aluout, lutout0, luten, result);

//multa route
mux2     #(WIDTH)  reg1_out(readdata1,aluout,multa_route,data1);
mux2     #(REGBITS) reg2_src(instr[REGBITS+15:16],instr[15:11],multa_route,readaddr2);

registers #(WIDTH,REGBITS) reg32(clk, reset, regwrite, wd, readaddr1, readaddr2, wa, readdata1, readdata2);
//alu     #(WIDTH) alunit(clk, aluinit, aluen, src1, src2, alucont, sign, aluresult);

zerodetect #(WIDTH) zd(result, zero);
endmodule

module alucontrol #(parameter WIDTH = 32)
    (input      clk, reset, sign,
     input      [1:0] aluop,
     input      [5:0] funct,
     input      [4:0] shamt,
     output reg [3:0] alucont,
     output      [1:0] modcount,
     output reg   done, init, en, luten);

    reg [2:0] state,nstate;
    reg [4:0] load;
    wire      complete;
    reg [1:0] count_in;
    wire [1:0] count_out;

    always @(posedge clk)
        if (reset) state <= 3'b000;
        else state <= nstate;
    always @(*)
        begin
            nstate <= 3'b000;
init      <= 0;
done      <= 1;
en        <= 0;
load      <= 0;
luten     <= 0;
            case(aluop)
                2'b00: alucont <= 4'b0010; // add for lb/sb
                2'b01: alucont <= 4'b0110; // sub (for beq)
                default: case (state) // R-Type instructions
                    3'b000: begin //state 1: single clk func / init for multi-clk
                        case(funct)
                            6'b100100: alucont <= 4'b0000; // A & B

```

```

        6'b100101: alucont <= 4'b0001; // A | B
        6'b100000: alucont <= 4'b0010; // A + B
        //                4'b0100; // A & ~B
        //                4'b0101; // A | ~B
        6'b100010: alucont <= 4'b0110; // A - B
        6'b101010: alucont <= 4'b0111; // slt

        6'b010000: alucont <= 4'b1010; // mfhi
        6'b010010: alucont <= 4'b1011; // mflo

        6'b011000: begin
            alucont <= 4'b1010; // mult init
nstate <= 3'b001;
load <= 5'b11111;
init <= 1;

            done <= 0;

        end

        6'b011100: begin
            alucont <= 4'b1010; // multa init
nstate <= 3'b001;
load <= 5'b11111;
init <= 1;

            done <= 0;

        end

        6'b011001: begin
            alucont <= 4'b1000; // multu init
nstate <= 3'b01;
load <= 5'b11111;
init <= 1;

            done <= 0;

        end

        6'b000000: begin
            alucont <= 4'b0000; // sll load
nstate <= 3'b001;
load <= shamt;
init <= 1;

            done <= 0;

        end

        6'b000010: begin
            alucont <= 4'b0000; // srl load
nstate <= 3'b001;
load <= shamt;
init <= 1;

            done <= 0;

        end

        6'b000011: begin
            alucont <= 4'b0000; // sra load
nstate <= 3'b001;
load <= shamt;
init <= 1;

            done <= 0;

        end

        6'b110100: begin
            alucont <= 4'b0010; // sinh
            luten <= 1'b1;
        end

```

```

        6'b110101: begin
            alucont <= 4'b0010; // cosh
            luten  <= 1'b1;
        end
        6'b110110: begin
            alucont <= 4'b0010; // csch
            luten  <= 1'b1;
        end
        6'b110111: begin
            alucont <= 4'b0010; // sech
            luten  <= 1'b1;
        end
        6'b111100: begin
            alucont <= 4'b0010; // -sinh
            luten  <= 1'b1;
        end
        6'b111101: begin
            alucont <= 4'b0010; // -cosh
            luten  <= 1'b1;
        end
        6'b111110: begin
            alucont <= 4'b0010; // -csch
            luten  <= 1'b1;
        end
        6'b111111: begin
            alucont <= 4'b0010; // -sech
            luten  <= 1'b1;
        end
        default: begin // mod, or periodic functions
            alucont <= 4'b1000; // LUT_sin (clear 64bit reg)
        end

    init <= 1;
    nstate <= 3'b100;

    done <= 0;

end

    endcase
    en <= 0;

end

    3'b001: begin // loop calculation terminated by counter
        en <= 1;
        done <= 0;
        init <= 0;
        alucont[1] <= funct[4] ^ funct[0];
        alucont[0] <= funct[4] ^^ funct[1];
        if (complete) nstate <= 3'b010;
        else nstate <= 3'b001;
    end
    3'b010: begin // set up for last multiply operation
        case (funct)
            6'b011000: begin
                alucont <= 4'b1110;
            end
        end
    nstate <= 3'b011;
    en <= 1;
end

```



```

        6'b011100: begin
            alucont <= 4'b1110;
nstate <= 3'b011;
en      <= 1;
end

        6'b011001: begin
            alucont <= 4'b1000;
nstate <= 3'b011;
en      <= 1;
end

        default: begin
            alucont <= 4'b1000; //out select for shift
nstate <= 3'b000;
end

        endcase

done <= ~funct[4];
end

3'b011: begin
    alucont <= 4'b1001; //scaled out
nstate <= 3'b000;
done <= 1'b1;
end

3'b100: begin          //load inputA to 64-bit reg
    alucont <= 4'b1001;
nstate <= 3'b101;
en      <= 1;

    done <= 0;
    count_in <= {1'b0,funct[0]} + 2'b11;

end

3'b101: begin          //A-B loop until negative
    alucont <= 4'b0101;
en      <= 1;

    done <= 0;
    count_in <= count_out + 2'b01;
    if (sign) nstate <= 3'b110;

else    nstate <= 3'b101;
end

    3'b110: begin
        alucont <= 4'b0001; //A+B loop until positive
en      <= 1;

        done <= 0;
        count_in <= count_out + 2'b11;
        if (~sign) nstate <= 3'b111;
        else      nstate <= 3'b110;

    end

    3'b111: begin
        alucont <= 4'b1010; //64-bit-High out
        done <= 1'b1;
        nstate <= 3'b000;
        luten <= 1'b1;

    end

    endcase

endcase

end

```

```

    downcounter #(5) mul_sh_count(clk, init, load, complete);
    dffcnr      #(2) mod_counter(clk, reset, en, count_in, count_out);
    assign modcount = count_out;

endmodule

module alu #(parameter WIDTH = 32)
    (input          clk, init, en,
     input  [WIDTH-1:0] a, b,
     input  [3:0]      alucont,
     output          sign,
     output reg [WIDTH-1:0] result);

    wire [WIDTH-1:0] a2,b2,b3,b4,shift_out, r_and, r_or, slt;
    wire [WIDTH:0]    sum;
    wire [WIDTH*2-1:0] loopreg_out;

    assign a2      = ((alucont[3] ^ alucont[0]) & ~(^alucont[3]&alucont[2]&alucont[1]&alucont[0])) ? loopreg_out[WIDTH*2-1:WIDTH]:a;
    assign b2[WIDTH-1] = alucont[1] ^ ((alucont[0]^shift_out[0]) & b[WIDTH-1]);
    assign b2[WIDTH-2:0] = {(WIDTH-1){(alucont[0]^shift_out[0])}} & b[WIDTH-2:0];
    assign b3          = alucont[3] ? b2:b;
    assign b4          = alucont[2] ? ^b3:b3;
    assign r_and = a & b4;
    assign r_or  = a | b4;
    assign sum = a2 + b4 + (^alucont[3] & alucont[2]);
    assign slt = {WIDTH{sum[WIDTH-1]}};
    assign sign = sum[WIDTH-1];

    shiftLR      #(WIDTH)  shifter(clk, init, ^init, alucont[1], alucont[0], a, shift_out);
    loopshiftright #(WIDTH*2) loopreg(clk, init, en, alucont[1], alucont[0], sum, loopreg_out);

    always@(*)
        case({alucont[3],alucont[1:0]})
            3'b000: result <= r_and;
            3'b001: result <= r_or;
            3'b010: result <= sum[WIDTH-1:0];
            3'b011: result <= slt;
            3'b100: result <= shift_out;
            3'b101: result <= loopreg_out[52:21];
            3'b110: result <= loopreg_out[WIDTH*2-1:WIDTH];
            3'b111: result <= loopreg_out[WIDTH-1:0];
        endcase

endmodule

module loopshiftright #(parameter WIDTH = 64)
    (input  clk, reset, en, load, shift,
     input  [WIDTH/2-1:0] data_in,
     output [WIDTH-1:0] data_out);
    wire [WIDTH-1:0] d;
    wire sum;

    dffcnrl #(2)    init_to_load(clk, reset,en,load,{d[WIDTH/2-1],d[0]},{data_out[WIDTH/2-1],data_out[0]});
    dffcnr  #(WIDTH-2) init_to_0(clk, reset,en,{d[WIDTH-1:WIDTH/2],d[WIDTH/2-2:1]},{data_out[WIDTH-1:WIDTH/2],data_out[WIDTH/2-2:1]});

```

```

    assign sum    = data_in[WIDTH/2] ^ data_out[0];

    assign d[WIDTH/2-2:0] = data_out[WIDTH/2-1:1];
    assign d[WIDTH-1:WIDTH/2-1] = shift ? {data_in[WIDTH/2-1:0], 1'b0}:{sum,data_in[WIDTH/2-1:0]};

endmodule

module counter #(parameter WIDTH = 8)
    (input          clk, reset, en,
    output reg [WIDTH-1:0] count);
    always@(posedge clk)
        if (reset) count <= 0;
        else if (en) count <= count + 1;
endmodule

module downcounter #(parameter WIDTH = 8)
    (input          clk, load,
    input    [WIDTH-1:0] val,
    output          one);
    reg [WIDTH-1:0] count;
    assign one = (~|count[WIDTH-1:1]) & count[0];
    always@(posedge clk)
        if (load)    count <= val;
        else        count <= count - 1;
endmodule

module shiftLR #(parameter WIDTH = 8)
    (input          clk, load, en, ar, lr,
    input    [WIDTH-1:0] data_in,
    output reg [WIDTH-1:0] data_out);
    always @(posedge clk)
        if (load) data_out <= data_in;
        else if (en)
            begin
                if (lr)
                    begin
                        data_out[WIDTH-1:1] <= data_out[WIDTH-2:0];
                        data_out[0] <= 1'b0;
                    end
                else if (ar)
                    begin
                        data_out[WIDTH-2:0] <= data_out[WIDTH-1:1];
                        data_out[WIDTH-1] <= data_out[WIDTH-1];
                    end
                else
                    begin
                        data_out[WIDTH-2:0] <= data_out[WIDTH-1:1];
                        data_out[WIDTH-1] <= 1'b0;
                    end
            end
    end
endmodule

```

```

module registers #(parameter D_WIDTH = 32, A_WIDTH = 5)
    (input          clk,reset,write_en,
     input  [D_WIDTH-1:0] write_data,
     input  [A_WIDTH-1:0] ra1, ra2, wa,
     output [D_WIDTH-1:0] rd1, rd2);

    wire [31:0] enable;
    wire [32*32-1:0] q;

    demux32 # (1)    writeselect(write_en, wa, enable[0], enable[1], enable[2], enable[3],
                                enable[4 ], enable[5 ], enable[6 ], enable[7 ], enable[8 ],
                                enable[9 ], enable[10], enable[11], enable[12], enable[13],
                                enable[14], enable[15], enable[16], enable[17], enable[18],
                                enable[19], enable[20], enable[21], enable[22], enable[23],
                                enable[24], enable[25], enable[26], enable[27], enable[28],
                                enable[29], enable[30], enable[31]);

    mux32 # (32)    out1select(q[32*1 -1:32*0 ],q[32*2 -1:32*1 ],q[32*3 -1:32*2 ],q[32*4 -1:32*3 ],
                                q[32*5 -1:32*4 ],q[32*6 -1:32*5 ],q[32*7 -1:32*6 ],q[32*8 -1:32*7 ],
                                q[32*9 -1:32*8 ],q[32*10-1:32*9 ],q[32*11-1:32*10],q[32*12-1:32*11],
                                q[32*13-1:32*12],q[32*14-1:32*13],q[32*15-1:32*14],q[32*16-1:32*15],
                                q[32*17-1:32*16],q[32*18-1:32*17],q[32*19-1:32*18],q[32*20-1:32*19],
                                q[32*21-1:32*20],q[32*22-1:32*21],q[32*23-1:32*22],q[32*24-1:32*23],
                                q[32*25-1:32*24],q[32*26-1:32*25],q[32*27-1:32*26],q[32*28-1:32*27],
                                q[32*29-1:32*28],q[32*30-1:32*29],q[32*31-1:32*30],q[32*32-1:32*31],
                                ra1, rd1);

    mux32 # (32)    out2select(q[32*1 -1:32*0 ],q[32*2 -1:32*1 ],q[32*3 -1:32*2 ],q[32*4 -1:32*3 ],
                                q[32*5 -1:32*4 ],q[32*6 -1:32*5 ],q[32*7 -1:32*6 ],q[32*8 -1:32*7 ],
                                q[32*9 -1:32*8 ],q[32*10-1:32*9 ],q[32*11-1:32*10],q[32*12-1:32*11],
                                q[32*13-1:32*12],q[32*14-1:32*13],q[32*15-1:32*14],q[32*16-1:32*15],
                                q[32*17-1:32*16],q[32*18-1:32*17],q[32*19-1:32*18],q[32*20-1:32*19],
                                q[32*21-1:32*20],q[32*22-1:32*21],q[32*23-1:32*22],q[32*24-1:32*23],
                                q[32*25-1:32*24],q[32*26-1:32*25],q[32*27-1:32*26],q[32*28-1:32*27],
                                q[32*29-1:32*28],q[32*30-1:32*29],q[32*31-1:32*30],q[32*32-1:32*31],
                                ra2, rd2);

    genvar i;
    generate for (i=0; i<32; i=i+1)
        begin: inst
            dffnr # (32) register(.clk(clk), .reset(reset), .en(enable[i]), .d(write_data), .q(q[32*(i+1)-1:32*i]));
        end
    endgenerate

endmodule

module zerodetect #(parameter WIDTH = 8)
    (input [WIDTH-1:0] a,
     output          y);

    assign y = (a==0);

endmodule

module dff #(parameter WIDTH = 8)
    (input          clk,
     input  [WIDTH-1:0] d,
     output reg [WIDTH-1:0] q);

```

```

    always @(posedge clk)
        q <= d;
endmodule

module dffn #(parameter WIDTH = 8)
    (input          clk, en,
     input  [WIDTH-1:0] d,
     output reg [WIDTH-1:0] q);

    always @(posedge clk)
        if (en) q <= d;
endmodule

module dffr #(parameter WIDTH = 8)
    (input          clk, reset, en,
     input  [WIDTH-1:0] d,
     output reg [WIDTH-1:0] q);

    always @(posedge clk)
        if (reset) q <= 0;
        else      q <= d;
endmodule

module dffnr #(parameter WIDTH = 8)
    (input          clk, reset, en,
     input  [WIDTH-1:0] d,
     output reg [WIDTH-1:0] q);

    always @(posedge clk)
        if (reset) q <= 0;
        else if (en) q <= d;
endmodule

module dffnrL #(parameter WIDTH = 8)
    (input          clk, reset,en,load,
     input  [WIDTH-1:0] d,
     output reg [WIDTH-1:0] q);

    always @(posedge clk)
        if (reset) q <= {WIDTH{load}};
        else if (en) q <= d;
endmodule

module mux2 #(parameter WIDTH = 8)
    (input  [WIDTH-1:0] in0, in1,
     input          sel,
     output  [WIDTH-1:0] y);

    assign y = sel ? in1 : in0;
endmodule

module mux4 #(parameter WIDTH = 8)
    (input  [WIDTH-1:0] in0, in1, in2, in3,
     input  [1:0]      sel,
     output reg [WIDTH-1:0] y);

    always @(*)
        case(sel)
            2'b00: y <= in0;
            2'b01: y <= in1;
            2'b10: y <= in2;
        endcase
endmodule

```

```

        2'b11: y <= in3;
    endcase
endmodule

module mux32 #(parameter WIDTH = 32)
    (input      [WIDTH-1:0] in0, in1, in2, in3, in4, in5, in6, in7, in8, in9, in10,
                                     in11, in12, in13, in14, in15, in16, in17, in18, in19,
                                     in20, in21, in22, in23, in24, in25, in26, in27, in28,
                                     in29, in30, in31,
    input      [4:0]      sel,
    output reg [WIDTH-1:0] y);
    always @(*)
        case(sel)
            5'b00000: y<=in0;
            5'b00001: y<=in1;
            5'b00010: y<=in2;
            5'b00011: y<=in3;
            5'b00100: y<=in4;
            5'b00101: y<=in5;
            5'b00110: y<=in6;
            5'b00111: y<=in7;
            5'b01000: y<=in8;
            5'b01001: y<=in9;
            5'b01010: y<=in10;
            5'b01011: y<=in11;
            5'b01100: y<=in12;
            5'b01101: y<=in13;
            5'b01110: y<=in14;
            5'b01111: y<=in15;
            5'b10000: y<=in16;
            5'b10001: y<=in17;
            5'b10010: y<=in18;
            5'b10011: y<=in19;
            5'b10100: y<=in20;
            5'b10101: y<=in21;
            5'b10110: y<=in22;
            5'b10111: y<=in23;
            5'b11000: y<=in24;
            5'b11001: y<=in25;
            5'b11010: y<=in26;
            5'b11011: y<=in27;
            5'b11100: y<=in28;
            5'b11101: y<=in29;
            5'b11110: y<=in30;
            5'b11111: y<=in31;
        endcase
endmodule

module demux32 #(parameter WIDTH = 1)
    (input      [WIDTH-1:0] in,
    input      [4:0]      sel,
    output reg [WIDTH-1:0] out0, out1, out2, out3, out4, out5, out6, out7,
    output reg [WIDTH-1:0] out8, out9, out10, out11, out12, out13, out14, out15,
    output reg [WIDTH-1:0] out16, out17, out18, out19, out20, out21, out22, out23,

```

```

output reg [WIDTH-1:0] out24, out25, out26, out27, out28, out29, out30, out31);
always @(*)
begin
    if (sel == 5'b00000) out0 <= in;
    else out0 <= 0;
    if (sel == 5'b00001) out1 <= in;
    else out1 <= 0;
    if (sel == 5'b00010) out2 <= in;
    else out2 <= 0;
    if (sel == 5'b00011) out3 <= in;
    else out3 <= 0;
    if (sel == 5'b00100) out4 <= in;
    else out4 <= 0;
    if (sel == 5'b00101) out5 <= in;
    else out5 <= 0;
    if (sel == 5'b00110) out6 <= in;
    else out6 <= 0;
    if (sel == 5'b00111) out7 <= in;
    else out7 <= 0;
    if (sel == 5'b01000) out8 <= in;
    else out8 <= 0;
    if (sel == 5'b01001) out9 <= in;
    else out9 <= 0;
    if (sel == 5'b01010) out10 <= in;
    else out10 <= 0;
    if (sel == 5'b01011) out11 <= in;
    else out11 <= 0;
    if (sel == 5'b01100) out12 <= in;
    else out12 <= 0;
    if (sel == 5'b01101) out13 <= in;
    else out13 <= 0;
    if (sel == 5'b01110) out14 <= in;
    else out14 <= 0;
    if (sel == 5'b01111) out15 <= in;
    else out15 <= 0;
    if (sel == 5'b10000) out16 <= in;
    else out16 <= 0;
    if (sel == 5'b10001) out17 <= in;
    else out17 <= 0;
    if (sel == 5'b10010) out18 <= in;
    else out18 <= 0;
    if (sel == 5'b10011) out19 <= in;
    else out19 <= 0;
    if (sel == 5'b10100) out20 <= in;
    else out20 <= 0;
    if (sel == 5'b10101) out21 <= in;
    else out21 <= 0;
    if (sel == 5'b10110) out22 <= in;
    else out22 <= 0;
    if (sel == 5'b10111) out23 <= in;
    else out23 <= 0;
    if (sel == 5'b11000) out24 <= in;
    else out24 <= 0;
    if (sel == 5'b11001) out25 <= in;

```

```

        else                out25 <= 0;
        if (sel == 5'b11010) out26 <= in;
        else                out26 <= 0;
        if (sel == 5'b11011) out27 <= in;
        else                out27 <= 0;
        if (sel == 5'b11100) out28 <= in;
        else                out28 <= 0;
        if (sel == 5'b11101) out29 <= in;
        else                out29 <= 0;
        if (sel == 5'b11110) out30 <= in;
        else                out30 <= 0;
        if (sel == 5'b11111) out31 <= in;
        else                out31 <= 0;
    end
endmodule

```

C.3 Memory Controller

```

//-----
// Cache Controller
//
// By: Ryan Wu, University of Calgary
// Last Modified: April 30, 2012
//
// This component manages all the data and controls
// between the MIPS processor, SRAM, and external
// memory and I/O components. It has additional
// support for ADC and DACs for 90nm fabrication.
//-----

module memcontrol #(parameter UWIDTH=16, MWIDTH=8, EWIDTH=16)
    (input                clk, reset, ram_bypass, write, en,
     input  [UWIDTH-1:0] uaddress, //uController address
     input  [31:0]       udatain,
     input  [7:0]        exdatain, ADC,
     input  [35:0]       cachein0,cachein1,cachein2,cachein3,cachein4,cachein5,cachein6,cachein7,
     output             halt, ex_clk, exMEMon,
     output [EWIDTH-1:0] eaddress, //External memory address
     output reg [31:0]   rowrWL, rowWL, //Cache memory row address
     output reg [7:0]    colrWL, colWL, //Cache memory colume address
     output [31:0]       udataout,
     output [7:0]        exdataout,
     output reg [7:0]    DAC1, DAC2,
     output reg [2:0]    ADCsel,
     output reg [35:0]   cacheout,
     output             ADCen);

    wire validbit, databit, Lclk, Len1, Len2, Len3, Len4, hit, valid, WL, rWL;
    wire [1:0] tag;
    wire [2:0] count;
    wire [7:0] d1, d2, d3, d4, q1, q2, q3, q4, exdataout1, exdataout2;
    wire [MWIDTH-1:0] maddress;

```



```

reg [35:0] cachein;
reg [15:0] luaddress;
reg data;
reg [7:0] ADCreg;

reg STANDBY, CHECKHIT, CHECKHITw, write2cache, exMEM, READY, DACnADC;

always @(posedge clk) begin
    STANDBY    <= reset | READY | (STANDBY & ~en);
    READY      <= ((CHECKHITw & hit & valid) | (exMEM & (&count) & (data | write | (!luaddress[13:10]))) | write2cache | DACnADC) & ~reset;
    CHECKHIT    <= STANDBY & en & ~write & ~luaddress[13:10] & ~luaddress[15:14] & ~reset;
    CHECKHITw   <= CHECKHIT & ~reset;
    exMEM       <= ((CHECKHITw & (~hit | ~valid)) | (STANDBY & en & luaddress[13:10] & ~luaddress[15:14]) | (exMEM & ~&count)) & ~reset;
    write2cache <= ((&count & exMEM & ~write & ~data) | (STANDBY & en & write)) & (~luaddress[13:10] & ~luaddress[15:14] & ~reset;
    DACnADC     <= STANDBY & en & luaddress[15:14] & ~reset;
end

always @(*)
    if (DACnADC) begin
        case (luaddress[15:14])
            2'b01: DAC1 <= udatain[7:0];
            2'b10: DAC2 <= udatain[7:0];
            2'b11: ADCreg <= ADC;
        endcase
    end

always @(*)
    if (&luaddress[15:14])
        ADCsel = luaddress[2:0];

always @(negedge STANDBY)
//    if (en & STANDBY)
        luaddress <= uaddress;

always @(*) begin
    case (maddress[7:5])
        3'b000: cachein <= cachein0;
        3'b001: cachein <= cachein1;
        3'b010: cachein <= cachein2;
        3'b011: cachein <= cachein3;
        3'b100: cachein <= cachein4;
        3'b101: cachein <= cachein5;
        3'b110: cachein <= cachein6;
        3'b111: cachein <= cachein7;
    endcase
end

always @(posedge Lclk)
    if (rWL)
        data <= cachein[34];

always @(*) begin
    if (WL)
        case (maddress[7:5])

```

```

3'b000: colWL <= 8'b00000001;
3'b001: colWL <= 8'b00000010;
3'b010: colWL <= 8'b00000100;
3'b011: colWL <= 8'b00001000;
3'b100: colWL <= 8'b00010000;
3'b101: colWL <= 8'b00100000;
3'b110: colWL <= 8'b01000000;
3'b111: colWL <= 8'b10000000;
endcase
else
colWL <= 8'b00000000;
end
always @(*) begin
if (rWL)
case (maddress[7:5])
3'b000: colrWL <= 8'b00000001;
3'b001: colrWL <= 8'b00000010;
3'b010: colrWL <= 8'b00000100;
3'b011: colrWL <= 8'b00001000;
3'b100: colrWL <= 8'b00010000;
3'b101: colrWL <= 8'b00100000;
3'b110: colrWL <= 8'b01000000;
3'b111: colrWL <= 8'b10000000;
endcase
else
colrWL <= 8'b11111111;
end
always @(*) begin
if (WL)
case (maddress[4:0])
5'b00000: rowWL <= 32'b00000000000000000000000000000001;
5'b00001: rowWL <= 32'b00000000000000000000000000000010;
5'b00010: rowWL <= 32'b000000000000000000000000000000100;
5'b00011: rowWL <= 32'b0000000000000000000000000000001000;
5'b00100: rowWL <= 32'b00000000000000000000000000000010000;
5'b00101: rowWL <= 32'b000000000000000000000000000000100000;
5'b00110: rowWL <= 32'b0000000000000000000000000000001000000;
5'b00111: rowWL <= 32'b00000000000000000000000000000010000000;
5'b01000: rowWL <= 32'b000000000000000000000000000000100000000;
5'b01001: rowWL <= 32'b0000000000000000000000000000001000000000;
5'b01010: rowWL <= 32'b00000000000000000000000000000010000000000;
5'b01011: rowWL <= 32'b000000000000000000000000000000100000000000;
5'b01100: rowWL <= 32'b0000000000000000000000000000001000000000000;
5'b01101: rowWL <= 32'b00000000000000000000000000000010000000000000;
5'b01110: rowWL <= 32'b000000000000000000000000000000100000000000000;
5'b01111: rowWL <= 32'b0000000000000000000000000000001000000000000000;
5'b10000: rowWL <= 32'b00000000000000000000000000000010000000000000000;
5'b10001: rowWL <= 32'b000000000000000000000000000000100000000000000000;
5'b10010: rowWL <= 32'b0000000000000000000000000000001000000000000000000;
5'b10011: rowWL <= 32'b00000000000000000000000000000010000000000000000000;
5'b10100: rowWL <= 32'b000000000000000000000000000000100000000000000000000;
5'b10101: rowWL <= 32'b0000000000000000000000000000001000000000000000000000;
5'b10110: rowWL <= 32'b00000000000000000000000000000010000000000000000000000;
5'b10111: rowWL <= 32'b000000000000000000000000000000100000000000000000000000;

```

```

5'b11000: rowWL <= 32'b00000001000000000000000000000000;
5'b11001: rowWL <= 32'b00000010000000000000000000000000;
5'b11010: rowWL <= 32'b00000100000000000000000000000000;
5'b11011: rowWL <= 32'b00001000000000000000000000000000;
5'b11100: rowWL <= 32'b00010000000000000000000000000000;
5'b11101: rowWL <= 32'b00100000000000000000000000000000;
5'b11110: rowWL <= 32'b01000000000000000000000000000000;
5'b11111: rowWL <= 32'b10000000000000000000000000000000;
endcase
else
    rowWL <= 32'b00000000000000000000000000000000;
end
always @(*) begin
    if (rWL)
        case (maddress[4:0])
            5'b00000: rowrWL <= 32'b00000000000000000000000000000001;
            5'b00001: rowrWL <= 32'b00000000000000000000000000000010;
            5'b00010: rowrWL <= 32'b000000000000000000000000000000100;
            5'b00011: rowrWL <= 32'b0000000000000000000000000000001000;
            5'b00100: rowrWL <= 32'b00000000000000000000000000000010000;
            5'b00101: rowrWL <= 32'b000000000000000000000000000000100000;
            5'b00110: rowrWL <= 32'b0000000000000000000000000000001000000;
            5'b00111: rowrWL <= 32'b00000000000000000000000000000010000000;
            5'b01000: rowrWL <= 32'b000000000000000000000000000000100000000;
            5'b01001: rowrWL <= 32'b0000000000000000000000000000001000000000;
            5'b01010: rowrWL <= 32'b00000000000000000000000000000010000000000;
            5'b01011: rowrWL <= 32'b000000000000000000000000000000100000000000;
            5'b01100: rowrWL <= 32'b0000000000000000000000000000001000000000000;
            5'b01101: rowrWL <= 32'b00000000000000000000000000000010000000000000;
            5'b01110: rowrWL <= 32'b000000000000000000000000000000100000000000000;
            5'b01111: rowrWL <= 32'b0000000000000000000000000000001000000000000000;
            5'b10000: rowrWL <= 32'b00000000000000000000000000000010000000000000000;
            5'b10001: rowrWL <= 32'b000000000000000000000000000000100000000000000000;
            5'b10010: rowrWL <= 32'b0000000000000000000000000000001000000000000000000;
            5'b10011: rowrWL <= 32'b00000000000000000000000000000010000000000000000000;
            5'b10100: rowrWL <= 32'b000000000000000000000000000000100000000000000000000;
            5'b10101: rowrWL <= 32'b0000000000000000000000000000001000000000000000000000;
            5'b10110: rowrWL <= 32'b00000000000000000000000000000010000000000000000000000;
            5'b10111: rowrWL <= 32'b000000000000000000000000000000100000000000000000000000;
            5'b11000: rowrWL <= 32'b000000010000000000000000000000000000000000000;
            5'b11001: rowrWL <= 32'b000000100000000000000000000000000000000000000;
            5'b11010: rowrWL <= 32'b000001000000000000000000000000000000000000000;
            5'b11011: rowrWL <= 32'b000010000000000000000000000000000000000000000;
            5'b11100: rowrWL <= 32'b000100000000000000000000000000000000000000000;
            5'b11101: rowrWL <= 32'b001000000000000000000000000000000000000000000;
            5'b11110: rowrWL <= 32'b010000000000000000000000000000000000000000000;
            5'b11111: rowrWL <= 32'b100000000000000000000000000000000000000000000;
        endcase
    else
        rowrWL <= 32'b00000000000000000000000000000000;
    end
end

always @(*) begin
    if (reset)

```



```

assign d[2:0] = sum[2:0];
assign a[2:0] = q[2:0];
assign b[2:0] = {carry[1:0],1'b1};

halfadder b0(a[0], b[0], sum[0], carry[0]);
halfadder b1(a[1], b[1], sum[1], carry[1]);
halfadder b2(a[2], b[2], sum[2], carry[2]);

dffenr #(3) outlatch(clk, reset, en, d, q);
assign count = q;

endmodule

module halfadder (input  a, b,
                  output sum, carry);
    wire nor1, nand1;

    assign nor1  = ~(a | b);
    assign nand1 = ~(a & b);
    assign carry = ~nand1;
    assign sum   = ~(carry | nor1);
endmodule

```

C.4 Test Signal Routing

```

//-----
// Test Circuit
//
// By: Ryan Wu, University of Calgary
// Last Modified: April 30, 2012
//
// Test module that samples different signals and
// isolates components for testing.  Additional
// features include SRAM bypass, ADC bypass, DAC
// bypass, ADC/DAC overwrite and isolation.
//
// Requires 8 digital inputs, 2 digital outputs
//-----
/*      out1      out2
sel = 0000:
sel = 0001: rdata      rdata
sel = 0010: adr        ADC
sel = 0011: DAC1       DAC2
sel = 0100: cachein0   cachein1
sel = 0101: cachein2   cachein3
sel = 0110: cachein4   cachein5
sel = 0111: cachein6   cachein7
sel = 1000: rowWL      rowrWL

```

```

sel = 1001: colWL      colrWL
sel = 1010: cacheout   cacheout

sel = 1011: cache memory bypass
sel = 1100: ADC bypass
sel = 1101: DAC bypass
sel = 1110: ADC through
sel = 1111: DAC through
*/

module dft (input      clk, load, en,
            output reg  out1, out2,

            input [3:0] sel,
            input [31:0] wdata, rdata,
            input [15:0] adr,
            input [35:0] cachein0, cachein1, cachein2, cachein3,
                        cachein4, cachein5, cachein6, cachein7, cacheout,
            input [31:0] rowWL, rowrWL,
            input [7:0] colWL, colrWL,

            output      MEMbp,
            input  [7:0] mipsdataout, exdatain, mipsDAC1, mipsDAC2, ADCin,

```

```

        output reg [7:0] mipsdatain, exdataout, DAC1out, DAC2out, mipsADC,
        input GND, VDD
    );

reg [6:0] count;
reg [7:0] testarray;
reg [63:0] outarray1, outarray2;

assign MEMbp = sel[3] & ~sel[2] & sel[1] & sel[0];

always @(*) begin
    case (sel)
        4'b1100: begin
            mipsADC <= testarray;
            exdataout <= mipsdataout;
            mipsdatain <= exdatain;
            DAC1out <= mipsDAC1;
            DAC2out <= mipsDAC2;
        end
        4'b1101: begin
            DAC1out <= exdatain;
            exdataout <= mipsdataout;
            mipsdatain <= exdatain;
            mipsADC <= ADCin;
            DAC2out <= mipsDAC2;
        end
        4'b1110: begin
            DAC2out <= exdatain;
            exdataout <= mipsdataout;
            mipsdatain <= exdatain;
            mipsADC <= ADCin;
            DAC1out <= mipsDAC1;
        end
        4'b1111: begin
            exdataout <= ADCin;
            mipsdatain <= exdatain;
            mipsADC <= ADCin;
            DAC1out <= mipsDAC1;
            DAC2out <= mipsDAC2;
        end
        default: begin
            exdataout <= mipsdataout;
            mipsdatain <= exdatain;
            mipsADC <= ADCin;
            DAC1out <= mipsDAC1;
            DAC2out <= mipsDAC2;
        end
    endcase
end

always @(posedge load)
    testarray <= exdatain;

always @(posedge clk) begin
        if (load)
            count <= 0;
        else
            count <= count + 1;
    end

always @(*)
    case (sel)
        //4'b0000:
        4'b0001: begin
            outarray1 <= {32'b0,wdata};
            outarray2 <= {32'b0,rdata};
        end
        4'b0010: begin
            outarray1 <= {48'b0,adr};
            outarray2 <= {56'b0,ADCin};
        end
        4'b0011: begin
            outarray1 <= {56'b0,DAC1out};
            outarray2 <= {56'b0,DAC2out};
        end
        5'b0100: begin
            outarray1 <= {28'b0,cachein0};
            outarray2 <= {28'b0,cachein1};
        end
        5'b0101: begin
            outarray1 <= {28'b0,cachein2};
            outarray2 <= {28'b0,cachein3};
        end
        5'b0110: begin
            outarray1 <= {28'b0,cachein4};
            outarray2 <= {28'b0,cachein5};
        end
        5'b0111: begin
            outarray1 <= {28'b0,cachein6};
            outarray2 <= {28'b0,cachein7};
        end
        5'b1000: begin
            outarray1 <= {32'b0,rowWL};
            outarray2 <= {32'b0,rowrWL};
        end
        5'b1001: begin
            outarray1 <= {56'b0,colWL};
            outarray2 <= {56'b0,colrWL};
        end
        5'b1010: begin
            outarray1 <= {28'b0,cacheout};
            outarray2 <= {28'b0,cacheout};
        end
        default: begin
            outarray1 <= 0;
            outarray2 <= 0;
        end
    end
end

```

```

endcase

always @(*)
case (count)
0: begin
    out1 <= outarray1[0];
    out2 <= outarray2[0];
    end
1: begin
    out1 <= outarray1[1];
    out2 <= outarray2[1];
    end
2: begin
    out1 <= outarray1[2];
    out2 <= outarray2[2];
    end
3: begin
    out1 <= outarray1[3];
    out2 <= outarray2[3];
    end
4: begin
    out1 <= outarray1[4];
    out2 <= outarray2[4];
    end
5: begin
    out1 <= outarray1[5];
    out2 <= outarray2[5];
    end
6: begin
    out1 <= outarray1[6];
    out2 <= outarray2[6];
    end
7: begin
    out1 <= outarray1[7];
    out2 <= outarray2[7];
    end
8: begin
    out1 <= outarray1[8];
    out2 <= outarray2[8];
    end
9: begin
    out1 <= outarray1[9];
    out2 <= outarray2[9];
    end
10: begin
    out1 <= outarray1[10];
    out2 <= outarray2[10];
    end
11: begin
    out1 <= outarray1[11];
    out2 <= outarray2[11];
    end
12: begin
    out1 <= outarray1[12];

    out2 <= outarray2[12];
    end
13: begin
    out1 <= outarray1[13];
    out2 <= outarray2[13];
    end
14: begin
    out1 <= outarray1[14];
    out2 <= outarray2[14];
    end
15: begin
    out1 <= outarray1[15];
    out2 <= outarray2[15];
    end
16: begin
    out1 <= outarray1[16];
    out2 <= outarray2[16];
    end
17: begin
    out1 <= outarray1[17];
    out2 <= outarray2[17];
    end
18: begin
    out1 <= outarray1[18];
    out2 <= outarray2[18];
    end
19: begin
    out1 <= outarray1[19];
    out2 <= outarray2[19];
    end
20: begin
    out1 <= outarray1[20];
    out2 <= outarray2[20];
    end
21: begin
    out1 <= outarray1[21];
    out2 <= outarray2[21];
    end
22: begin
    out1 <= outarray1[22];
    out2 <= outarray2[22];
    end
23: begin
    out1 <= outarray1[23];
    out2 <= outarray2[23];
    end
24: begin
    out1 <= outarray1[24];
    out2 <= outarray2[24];
    end
25: begin
    out1 <= outarray1[25];
    out2 <= outarray2[25];
    end

```

```

26: begin
    out1 <= outarray1[26];
    out2 <= outarray2[26];
end
27: begin
    out1 <= outarray1[27];
    out2 <= outarray2[27];
end
28: begin
    out1 <= outarray1[28];
    out2 <= outarray2[28];
end
29: begin
    out1 <= outarray1[29];
    out2 <= outarray2[29];
end
30: begin
    out1 <= outarray1[30];
    out2 <= outarray2[30];
end
31: begin
    out1 <= outarray1[31];
    out2 <= outarray2[31];
end
32: begin
    out1 <= outarray1[32];
    out2 <= outarray2[32];
end
33: begin
    out1 <= outarray1[33];
    out2 <= outarray2[33];
end
34: begin
    out1 <= outarray1[34];
    out2 <= outarray2[34];
end
35: begin
    out1 <= outarray1[35];
    out2 <= outarray2[35];
end
36: begin
    out1 <= outarray1[36];
    out2 <= outarray2[36];
end
37: begin
    out1 <= outarray1[37];
    out2 <= outarray2[37];
end
38: begin
    out1 <= outarray1[38];
    out2 <= outarray2[38];
end
39: begin
    out1 <= outarray1[39];

    out2 <= outarray2[39];
end
40: begin
    out1 <= outarray1[40];
    out2 <= outarray2[40];
end
41: begin
    out1 <= outarray1[41];
    out2 <= outarray2[41];
end
42: begin
    out1 <= outarray1[42];
    out2 <= outarray2[42];
end
43: begin
    out1 <= outarray1[43];
    out2 <= outarray2[43];
end
44: begin
    out1 <= outarray1[44];
    out2 <= outarray2[44];
end
45: begin
    out1 <= outarray1[45];
    out2 <= outarray2[45];
end
46: begin
    out1 <= outarray1[46];
    out2 <= outarray2[46];
end
47: begin
    out1 <= outarray1[47];
    out2 <= outarray2[47];
end
48: begin
    out1 <= outarray1[48];
    out2 <= outarray2[48];
end
49: begin
    out1 <= outarray1[49];
    out2 <= outarray2[49];
end
50: begin
    out1 <= outarray1[50];
    out2 <= outarray2[50];
end
51: begin
    out1 <= outarray1[51];
    out2 <= outarray2[51];
end
52: begin
    out1 <= outarray1[52];
    out2 <= outarray2[52];
end

```



```

53: begin
    out1 <= outarray1[53];
    out2 <= outarray2[53];
end
54: begin
    out1 <= outarray1[54];
    out2 <= outarray2[54];
end
55: begin
    out1 <= outarray1[55];
    out2 <= outarray2[55];
end
56: begin
    out1 <= outarray1[56];
    out2 <= outarray2[56];
end
57: begin
    out1 <= outarray1[57];
    out2 <= outarray2[57];
end
58: begin
    out1 <= outarray1[58];
    out2 <= outarray2[58];
end

59: begin
    out1 <= outarray1[59];
    out2 <= outarray2[59];
end
60: begin
    out1 <= outarray1[60];
    out2 <= outarray2[60];
end
61: begin
    out1 <= outarray1[61];
    out2 <= outarray2[61];
end
62: begin
    out1 <= outarray1[62];
    out2 <= outarray2[62];
end
63: begin
    out1 <= outarray1[63];
    out2 <= outarray2[63];
end
endcase

endmodule

```

C.5 Look-up Tables

```

//-----
// Look-Up-Table for Non-Linear Functions
//
// By: Ryan Wu, University of Calgary
// Last Modified: April 30, 2012
//
// This LUT supports non-linear functions including
// cos, sin, sec, csc, cosh, sinh, sech, csch, and
// the 2's complement for each result. The precision
// of the output is tuned to the lowest requirement
// (still with optimal performance) for the path-
// planning-algorithm.
//-----

module lutcontrol(input      en,
                  input  [3 :0] sel,      // neg/hyp/inv/cos
                  input  [1 :0] count,
                  input  [31:0] data_in,
                  output [31:0] data_out);

    wire sin_en, csc_en, sinh_en, csch_en, cosh_en, sech_en, rev, neg, zero;
    wire [14:0] in1;
    reg  [1:0] in3;
    reg  [4:0] in2;
    reg  [3:0] LUTin;

```

```

wire [15:0] SINout, CSCout, Hout, CSCHout, SECHout, out2;
reg [15:0] LUTout;

assign sin_en = (~sel[2:1]) & en;
assign csc_en = (~sel[2] & sel[1]) & en;
assign sinh_en = ( sel[2] & ~sel[1] & ~sel[0]) & en;
assign cosh_en = ( sel[2] & ~sel[1] & sel[0]) & en;
assign csch_en = ( sel[2] & sel[1] & ~sel[0]) & en;
assign sech_en = ( sel[2] & sel[1] & sel[0]) & en;

assign in1 = data_in[31] ? ~data_in[31:17]:data_in[31:17];
assign rev = (sin_en | csc_en) ? count[0]:1'b0;
assign neg = (sin_en | csc_en) ? (count[1] ^ sel[3]):(((sinh_en | csch_en) & data_in[31]) ^ sel[3]);

always @(*)
    if (en) begin
        if (sin_en | csc_en) begin //periodic functions
            in2 <= in1[4:0];
            in3 <= 0;
            end
        else begin //hyperbolic functions
            in2 <= in1[14:7] ? 5'b11111:in1[4:0];
            in3 <= in1[14:7] ? 2'b11 :in1[6:5];//max cap2
            end
        end
    else in2 <= 4'b0000;

always @(*)
    if (rev)
        case (in2[4:1])
            4'b0000: LUTin <= 4'b1100;
            4'b0001: LUTin <= 4'b1011;
            4'b0010: LUTin <= 4'b1010;
            4'b0011: LUTin <= 4'b1001;
            4'b0100: LUTin <= 4'b1000;
            4'b0101: LUTin <= 4'b0111;
            4'b0110: LUTin <= 4'b0110;
            4'b0111: LUTin <= 4'b0101;
            4'b1000: LUTin <= 4'b0100;
            4'b1001: LUTin <= 4'b0011;
            4'b1010: LUTin <= 4'b0010;
            4'b1011: LUTin <= 4'b0001;
            4'b1100: LUTin <= 4'b0000;
            default: LUTin <= 0;
        endcase
    else LUTin <= in2[4:1];

assign SINout[15:6] = 0;
assign CSCout[15:9] = 0;
assign Hout[15 ] = 0;
assign Hout[ 2:0] = 0;
assign SECHout[15:6] = 0;
assign CSCHout[15:11]= 0;

```

```

assign CSCHout[2:0] = 0;
assign SECHout[10:6]= 5'b00000;

lut_sin  sin_table(sin_en, LUTin, SINout[5:0]);
lut_csc  csc_table(csc_en, LUTin, CSCout[8:0]);
lut_sinhcosh hype_table(sinh_en, cosh_en, {in3,in2[4:2]}, Hout[14:3]);
lut_csch  csch_table(csch_en, {in3,in2}, CSCHout[10:3]);
lut_sech  sech_table(sech_en, {in3,in2[4:1]}, SECHout[5:0]);

always @(*)
    case (sel[2:0])
        3'b000: LUTout <= SINout;
        3'b001: LUTout <= SINout;
        3'b010: LUTout <= CSCout;
        3'b011: LUTout <= CSCout;
        3'b100: LUTout <= Hout;
        3'b101: LUTout <= Hout;
        3'b110: LUTout <= CSCHout;
        3'b111: LUTout <= SECHout;
    endcase

assign zero = ~|LUTout;
assign data_out[31:16] = (neg & ~zero) ? ~LUTout:LUTout;
assign data_out[15:0 ] = {16{LUTout[15]}};

endmodule

module lut_sin (input          en,
                input  [3:0] in,
                output reg [5:0] out);

always @(*)
    if (en)
        case (in[3:0])//1,5
            4'b0000: out <= 6'b0000000;
            4'b0001: out <= 6'b000110;
            4'b0010: out <= 6'b001010;
            4'b0011: out <= 6'b001101;
            4'b0100: out <= 6'b010001;
            4'b0101: out <= 6'b010100;
            4'b0110: out <= 6'b010111;
            4'b0111: out <= 6'b011010;
            4'b1000: out <= 6'b011100;
            4'b1001: out <= 6'b011110;
            4'b1010: out <= 6'b011111;
            4'b1011: out <= 6'b100000;
            4'b1100: out <= 6'b100000;
            default: out <= 6'b000000; //should never happen
        endcase
    else out <= 0;
endmodule

module lut_csc (input          en,
                input  [3:0] in,

```

```

        output reg [8:0] out);

always @(*)
    if (en)
        case (in[3:0])//4,5
            4'b0000: out <= 9'b11111111;
            4'b0001: out <= 9'b01110000;
            4'b0010: out <= 9'b01000000;
            4'b0011: out <= 9'b00101010;
            4'b0100: out <= 9'b00100000;
            4'b0101: out <= 9'b00011011;
            4'b0110: out <= 9'b00010110;
            4'b0111: out <= 9'b00010101;
            4'b1000: out <= 9'b00010010;
            4'b1001: out <= 9'b00010001;
            4'b1010: out <= 9'b00010000;
            4'b1011: out <= 9'b00010000;
            4'b1100: out <= 9'b00010000;
            default: out <= 9'b00000000;
        endcase
    else out <= 0;
endmodule

module lut_sinhcosh(input sinh, cosh,
    input [4:0] in,
    output reg [11:0] out);

always @(*)
    if (sinh|cosh) begin
        if (~in[4]) begin
            if (sinh)
                case (in[3:0])//5,2
                    4'b0000: out[6:0] <= 7'b0000000;
                    4'b0001: out[6:0] <= 7'b0000001;
                    4'b0010: out[6:0] <= 7'b0000010;
                    4'b0011: out[6:0] <= 7'b0000100;
                    4'b0100: out[6:0] <= 7'b0000101;
                    4'b0101: out[6:0] <= 7'b0000111;
                    4'b0110: out[6:0] <= 7'b0001001;
                    4'b0111: out[6:0] <= 7'b0001100;
                    4'b1000: out[6:0] <= 7'b0010000;
                    4'b1001: out[6:0] <= 7'b0010101;
                    4'b1010: out[6:0] <= 7'b0011011;
                    4'b1011: out[6:0] <= 7'b0100011;
                    4'b1100: out[6:0] <= 7'b0101100;
                    4'b1101: out[6:0] <= 7'b0111010;
                    4'b1110: out[6:0] <= 7'b1001010;
                    4'b1111: out[6:0] <= 7'b1011111;
                endcase
            else if (cosh)
                case (in[3:0])
                    4'b0000: out[6:0] <= 7'b0000100;
                    4'b0001: out[6:0] <= 7'b0000100;
                    4'b0010: out[6:0] <= 7'b0000101;
                    4'b0011: out[6:0] <= 7'b0000101;
                    4'b0100: out[6:0] <= 7'b0000110;
                endcase
        end
    end
endmodule

```

```

        4'b0101: out[6:0] <= 7'b0001000;
        4'b0110: out[6:0] <= 7'b0001010;
        4'b0111: out[6:0] <= 7'b0001101;
        4'b1000: out[6:0] <= 7'b0010001;
        4'b1001: out[6:0] <= 7'b0010110;
        4'b1010: out[6:0] <= 7'b0011100;
        4'b1011: out[6:0] <= 7'b0100011;
        4'b1100: out[6:0] <= 7'b0101101;
        4'b1101: out[6:0] <= 7'b0111001;
        4'b1110: out[6:0] <= 7'b1001011;
        4'b1111: out[6:0] <= 7'b1100000;
    endcase
    out[11:7] <= 0;
end
else begin
    case (in[3:0])
        4'b0000: out[11:5] <= 7'b0000011;
        4'b0001: out[11:5] <= 7'b0000100;
        4'b0010: out[11:5] <= 7'b0000101;
        4'b0011: out[11:5] <= 7'b0000111;
        4'b0100: out[11:5] <= 7'b0001001;
        4'b0101: out[11:5] <= 7'b0001100;
        4'b0110: out[11:5] <= 7'b0010000;
        4'b0111: out[11:5] <= 7'b0010101;
        4'b1000: out[11:5] <= 7'b0011011;
        4'b1001: out[11:5] <= 7'b0100011;
        4'b1010: out[11:5] <= 7'b0101101;
        4'b1011: out[11:5] <= 7'b0111101;
        4'b1100: out[11:5] <= 7'b1001011;
        4'b1101: out[11:5] <= 7'b1100000;
        4'b1110: out[11:5] <= 7'b1111001;
        4'b1111: out[11:5] <= 7'b1111111;
    endcase
    out[4:0] <= 5'b11111;
end
end
else
    out <= 0;

endmodule

module lut_sech(input      en,
                input  [5:0] in,
                output reg [5:0] out);

always @(*)
    if (en & ~in[5])
        case (in[4:1])/1,5
            4'b0000: out <= 6'b100000;
            4'b0001: out <= 6'b011110;
            4'b0010: out <= 6'b011011;
            4'b0011: out <= 6'b010111;
            4'b0100: out <= 6'b010011;
            4'b0101: out <= 6'b010000;

```

```

4'b0110: out <= 6'b001100;
4'b0111: out <= 6'b001010;
4'b1000: out <= 6'b001000;
4'b1001: out <= 6'b000110;
4'b1010: out <= 6'b000101;
4'b1011: out <= 6'b000100;
4'b1100: out <= 6'b000011;
4'b1101: out <= 6'b000010;
    4'b1110: out <= 6'b000010;
    4'b1111: out <= 6'b000000;
endcase
else out <= 0;
endmodule

module lut_csch(input      en,
                input  [6:0] in,
                output reg [7:0] out);
always @(*)
    if (en & ~|in[6:4])
        case (in[3:0])//7,1
            4'b0000: out <= 8'b11111111;
            4'b0001: out <= 8'b00101000;
            4'b0010: out <= 8'b00011001;
            4'b0011: out <= 8'b00010010;
            4'b0100: out <= 8'b00001110;
            4'b0101: out <= 8'b00001011;
            4'b0110: out <= 8'b00001010;
            4'b0111: out <= 8'b00001001;
            4'b1000: out <= 8'b00001000;
            4'b1001: out <= 8'b00000111;
            4'b1010: out <= 8'b00000110;
            4'b1011: out <= 8'b00000101;
            4'b1100: out <= 8'b00000101;
            4'b1101: out <= 8'b00000101;
            4'b1110: out <= 8'b00000100;
            4'b1111: out <= 8'b00000100;
        endcase
    else if (en & ~|in[6:5])
        case (in[3:2])
            2'b00: out <= 8'b00000011;
            2'b01: out <= 8'b00000010;
            2'b10: out <= 8'b00000001;
            2'b11: out <= 8'b00000000;
        endcase
    else out <= 0;
endmodule

```

C.6 Memory

```

//-----
// External Memory + Cache
//
// By: Ryan Wu, University of Calgary
// Last Modified: April 30, 2012
//
// Behavioural external memory is built for simulation
// purpose, where it is a word-addressable memory connected
// external to the chip.
// Cache (SRAM) structural code here is the extracted net
// list from Virtuoso, to verify it's operation and
// compatibility in simulation.
//-----
module exmem #(parameter WIDTH = 8, RAM_ADDR_BITS = 10)
    (input clk, en, memwrite,
     input [RAM_ADDR_BITS-1:0] adr,
     input [WIDTH-1:0] writedata,
     output reg [WIDTH-1:0] memdata
    );

    reg [WIDTH-1:0] external_memory [(2**RAM_ADDR_BITS)-1:0];

    // The following $readmemh statement initializes the RAM contents
    // via an external file (use $readmemb for binary data). The fib.dat
    // file is a list of bytes, one per line, starting at address 0.
    initial $readmemh("bench.dat", external_memory);

    // The behavioral description of the RAM - note clocked behavior
    always @(*)//negedge clk
        if (en) begin
            if (memwrite)
                external_memory[adr] <= writedata;
            memdata <= external_memory[adr];
        end
endmodule

module xINVD1 ( ZN, VDD, VSS, I );
output ZN;

inout VDD, VSS;

input I;

specify
    specparam CDS_LIBNAME = "tcbn90ghp";
    specparam CDS_CELLNAME = "INVD1";
    specparam CDS_VIEWNAME = "schematic";
endspecify

nch Inst_0 ( .D(ZN), .B(VSS), .G(I), .S(VSS));
pch Inst_1 ( .D(ZN), .B(VDD), .G(I), .S(VDD));
endmodule

module RAM_256W_schematic ( rBL0, rBL1, rBL2, rBL3, rBL4, rBL5, rBL6,
    rBL7, GND, RAM_VDD, BL, RST, colWL, colrWL, rowWL, rowrWL );

inout GND, RAM_VDD;

input RST;

output [35:0] rBL6;
output [35:0] rBL1;
output [35:0] rBL2;
output [35:0] rBL3;
output [35:0] rBL5;
output [35:0] rBL0;
output [35:0] rBL7;
output [35:0] rBL4;

input [7:0] colWL;
input [7:0] colrWL;
input [31:0] rowWL;
input [35:0] BL;
input [31:0] rowrWL;

// Buses in the design

wire [0:35] net239;

wire [0:31] net134;

wire [0:35] net266;

wire [0:35] net203;

wire [0:35] net137;

wire [0:35] net248;

wire [0:35] net221;

wire [0:31] net200;

wire [0:35] net212;

wire [0:35] net138;

wire [0:31] net194;

wire [7:0] rWL;

wire [0:35] net230;

wire [0:35] net257;

```

```
wire [7:0] WL;

wire [0:31] net190;
```

```
specify
```

```
    specparam CDS_LIBNAME = "RAM";
    specparam CDS_CELLNAME = "256W";
    specparam CDS_VIEWNAME = "schematic";
```

```
endspecify
```

```
xINVD2 I35_31_ ( net200[0], RAM_VDD, GND, net190[0]);
xINVD2 I35_30_ ( net200[1], RAM_VDD, GND, net190[1]);
xINVD2 I35_29_ ( net200[2], RAM_VDD, GND, net190[2]);
xINVD2 I35_28_ ( net200[3], RAM_VDD, GND, net190[3]);
xINVD2 I35_27_ ( net200[4], RAM_VDD, GND, net190[4]);
xINVD2 I35_26_ ( net200[5], RAM_VDD, GND, net190[5]);
xINVD2 I35_25_ ( net200[6], RAM_VDD, GND, net190[6]);
xINVD2 I35_24_ ( net200[7], RAM_VDD, GND, net190[7]);
xINVD2 I35_23_ ( net200[8], RAM_VDD, GND, net190[8]);
xINVD2 I35_22_ ( net200[9], RAM_VDD, GND, net190[9]);
xINVD2 I35_21_ ( net200[10], RAM_VDD, GND, net190[10]);
xINVD2 I35_20_ ( net200[11], RAM_VDD, GND, net190[11]);
xINVD2 I35_19_ ( net200[12], RAM_VDD, GND, net190[12]);
xINVD2 I35_18_ ( net200[13], RAM_VDD, GND, net190[13]);
xINVD2 I35_17_ ( net200[14], RAM_VDD, GND, net190[14]);
xINVD2 I35_16_ ( net200[15], RAM_VDD, GND, net190[15]);
xINVD2 I35_15_ ( net200[16], RAM_VDD, GND, net190[16]);
xINVD2 I35_14_ ( net200[17], RAM_VDD, GND, net190[17]);
xINVD2 I35_13_ ( net200[18], RAM_VDD, GND, net190[18]);
xINVD2 I35_12_ ( net200[19], RAM_VDD, GND, net190[19]);
xINVD2 I35_11_ ( net200[20], RAM_VDD, GND, net190[20]);
xINVD2 I35_10_ ( net200[21], RAM_VDD, GND, net190[21]);
xINVD2 I35_9_ ( net200[22], RAM_VDD, GND, net190[22]);
xINVD2 I35_8_ ( net200[23], RAM_VDD, GND, net190[23]);
xINVD2 I35_7_ ( net200[24], RAM_VDD, GND, net190[24]);
xINVD2 I35_6_ ( net200[25], RAM_VDD, GND, net190[25]);
xINVD2 I35_5_ ( net200[26], RAM_VDD, GND, net190[26]);
xINVD2 I35_4_ ( net200[27], RAM_VDD, GND, net190[27]);
xINVD2 I35_3_ ( net200[28], RAM_VDD, GND, net190[28]);
xINVD2 I35_2_ ( net200[29], RAM_VDD, GND, net190[29]);
xINVD2 I35_1_ ( net200[30], RAM_VDD, GND, net190[30]);
xINVD2 I35_0_ ( net200[31], RAM_VDD, GND, net190[31]);
xINVD2 I36_31_ ( net134[0], RAM_VDD, GND, net194[0]);
xINVD2 I36_30_ ( net134[1], RAM_VDD, GND, net194[1]);
xINVD2 I36_29_ ( net134[2], RAM_VDD, GND, net194[2]);
xINVD2 I36_28_ ( net134[3], RAM_VDD, GND, net194[3]);
xINVD2 I36_27_ ( net134[4], RAM_VDD, GND, net194[4]);
xINVD2 I36_26_ ( net134[5], RAM_VDD, GND, net194[5]);
xINVD2 I36_25_ ( net134[6], RAM_VDD, GND, net194[6]);
xINVD2 I36_24_ ( net134[7], RAM_VDD, GND, net194[7]);
xINVD2 I36_23_ ( net134[8], RAM_VDD, GND, net194[8]);
xINVD2 I36_22_ ( net134[9], RAM_VDD, GND, net194[9]);
```

```
xINVD2 I36_21_ ( net134[10], RAM_VDD, GND, net194[10]);
xINVD2 I36_20_ ( net134[11], RAM_VDD, GND, net194[11]);
xINVD2 I36_19_ ( net134[12], RAM_VDD, GND, net194[12]);
xINVD2 I36_18_ ( net134[13], RAM_VDD, GND, net194[13]);
xINVD2 I36_17_ ( net134[14], RAM_VDD, GND, net194[14]);
xINVD2 I36_16_ ( net134[15], RAM_VDD, GND, net194[15]);
xINVD2 I36_15_ ( net134[16], RAM_VDD, GND, net194[16]);
xINVD2 I36_14_ ( net134[17], RAM_VDD, GND, net194[17]);
xINVD2 I36_13_ ( net134[18], RAM_VDD, GND, net194[18]);
xINVD2 I36_12_ ( net134[19], RAM_VDD, GND, net194[19]);
xINVD2 I36_11_ ( net134[20], RAM_VDD, GND, net194[20]);
xINVD2 I36_10_ ( net134[21], RAM_VDD, GND, net194[21]);
xINVD2 I36_9_ ( net134[22], RAM_VDD, GND, net194[22]);
xINVD2 I36_8_ ( net134[23], RAM_VDD, GND, net194[23]);
xINVD2 I36_7_ ( net134[24], RAM_VDD, GND, net194[24]);
xINVD2 I36_6_ ( net134[25], RAM_VDD, GND, net194[25]);
xINVD2 I36_5_ ( net134[26], RAM_VDD, GND, net194[26]);
xINVD2 I36_4_ ( net134[27], RAM_VDD, GND, net194[27]);
xINVD2 I36_3_ ( net134[28], RAM_VDD, GND, net194[28]);
xINVD2 I36_2_ ( net134[29], RAM_VDD, GND, net194[29]);
xINVD2 I36_1_ ( net134[30], RAM_VDD, GND, net194[30]);
xINVD2 I36_0_ ( net134[31], RAM_VDD, GND, net194[31]);
xINVD2 I34_35_ ( net138[0], RAM_VDD, GND, net137[0]);
xINVD2 I34_34_ ( net138[1], RAM_VDD, GND, net137[1]);
xINVD2 I34_33_ ( net138[2], RAM_VDD, GND, net137[2]);
xINVD2 I34_32_ ( net138[3], RAM_VDD, GND, net137[3]);
xINVD2 I34_31_ ( net138[4], RAM_VDD, GND, net137[4]);
xINVD2 I34_30_ ( net138[5], RAM_VDD, GND, net137[5]);
xINVD2 I34_29_ ( net138[6], RAM_VDD, GND, net137[6]);
xINVD2 I34_28_ ( net138[7], RAM_VDD, GND, net137[7]);
xINVD2 I34_27_ ( net138[8], RAM_VDD, GND, net137[8]);
xINVD2 I34_26_ ( net138[9], RAM_VDD, GND, net137[9]);
xINVD2 I34_25_ ( net138[10], RAM_VDD, GND, net137[10]);
xINVD2 I34_24_ ( net138[11], RAM_VDD, GND, net137[11]);
xINVD2 I34_23_ ( net138[12], RAM_VDD, GND, net137[12]);
xINVD2 I34_22_ ( net138[13], RAM_VDD, GND, net137[13]);
xINVD2 I34_21_ ( net138[14], RAM_VDD, GND, net137[14]);
xINVD2 I34_20_ ( net138[15], RAM_VDD, GND, net137[15]);
xINVD2 I34_19_ ( net138[16], RAM_VDD, GND, net137[16]);
xINVD2 I34_18_ ( net138[17], RAM_VDD, GND, net137[17]);
xINVD2 I34_17_ ( net138[18], RAM_VDD, GND, net137[18]);
xINVD2 I34_16_ ( net138[19], RAM_VDD, GND, net137[19]);
xINVD2 I34_15_ ( net138[20], RAM_VDD, GND, net137[20]);
xINVD2 I34_14_ ( net138[21], RAM_VDD, GND, net137[21]);
xINVD2 I34_13_ ( net138[22], RAM_VDD, GND, net137[22]);
xINVD2 I34_12_ ( net138[23], RAM_VDD, GND, net137[23]);
xINVD2 I34_11_ ( net138[24], RAM_VDD, GND, net137[24]);
xINVD2 I34_10_ ( net138[25], RAM_VDD, GND, net137[25]);
xINVD2 I34_9_ ( net138[26], RAM_VDD, GND, net137[26]);
xINVD2 I34_8_ ( net138[27], RAM_VDD, GND, net137[27]);
xINVD2 I34_7_ ( net138[28], RAM_VDD, GND, net137[28]);
xINVD2 I34_6_ ( net138[29], RAM_VDD, GND, net137[29]);
xINVD2 I34_5_ ( net138[30], RAM_VDD, GND, net137[30]);
xINVD2 I34_4_ ( net138[31], RAM_VDD, GND, net137[31]);
```



```

xINVD1 I37_3_ ( rBL6[3], RAM_VDD, GND, net257[32]);
xINVD1 I37_2_ ( rBL6[2], RAM_VDD, GND, net257[33]);
xINVD1 I37_1_ ( rBL6[1], RAM_VDD, GND, net257[34]);
xINVD1 I37_0_ ( rBL6[0], RAM_VDD, GND, net257[35]);
xINVD1 I12_7_ ( rWL[7], RAM_VDD, GND, colrWL[7]);
xINVD1 I12_6_ ( rWL[6], RAM_VDD, GND, colrWL[6]);
xINVD1 I12_5_ ( rWL[5], RAM_VDD, GND, colrWL[5]);
xINVD1 I12_4_ ( rWL[4], RAM_VDD, GND, colrWL[4]);
xINVD1 I12_3_ ( rWL[3], RAM_VDD, GND, colrWL[3]);
xINVD1 I12_2_ ( rWL[2], RAM_VDD, GND, colrWL[2]);
xINVD1 I12_1_ ( rWL[1], RAM_VDD, GND, colrWL[1]);
xINVD1 I12_0_ ( rWL[0], RAM_VDD, GND, colrWL[0]);
xINVD1 I10_7_ ( WL[7], RAM_VDD, GND, colWL[7]);
xINVD1 I10_6_ ( WL[6], RAM_VDD, GND, colWL[6]);
xINVD1 I10_5_ ( WL[5], RAM_VDD, GND, colWL[5]);
xINVD1 I10_4_ ( WL[4], RAM_VDD, GND, colWL[4]);
xINVD1 I10_3_ ( WL[3], RAM_VDD, GND, colWL[3]);
xINVD1 I10_2_ ( WL[2], RAM_VDD, GND, colWL[2]);
xINVD1 I10_1_ ( WL[1], RAM_VDD, GND, colWL[1]);
xINVD1 I10_0_ ( WL[0], RAM_VDD, GND, colWL[0]);
xINVD1 I23_35_ ( rBL7[35], RAM_VDD, GND, net266[0]);
xINVD1 I23_34_ ( rBL7[34], RAM_VDD, GND, net266[1]);
xINVD1 I23_33_ ( rBL7[33], RAM_VDD, GND, net266[2]);
xINVD1 I23_32_ ( rBL7[32], RAM_VDD, GND, net266[3]);
xINVD1 I23_31_ ( rBL7[31], RAM_VDD, GND, net266[4]);
xINVD1 I23_30_ ( rBL7[30], RAM_VDD, GND, net266[5]);
xINVD1 I23_29_ ( rBL7[29], RAM_VDD, GND, net266[6]);
xINVD1 I23_28_ ( rBL7[28], RAM_VDD, GND, net266[7]);
xINVD1 I23_27_ ( rBL7[27], RAM_VDD, GND, net266[8]);
xINVD1 I23_26_ ( rBL7[26], RAM_VDD, GND, net266[9]);
xINVD1 I23_25_ ( rBL7[25], RAM_VDD, GND, net266[10]);
xINVD1 I23_24_ ( rBL7[24], RAM_VDD, GND, net266[11]);
xINVD1 I23_23_ ( rBL7[23], RAM_VDD, GND, net266[12]);
xINVD1 I23_22_ ( rBL7[22], RAM_VDD, GND, net266[13]);
xINVD1 I23_21_ ( rBL7[21], RAM_VDD, GND, net266[14]);
xINVD1 I23_20_ ( rBL7[20], RAM_VDD, GND, net266[15]);
xINVD1 I23_19_ ( rBL7[19], RAM_VDD, GND, net266[16]);
xINVD1 I23_18_ ( rBL7[18], RAM_VDD, GND, net266[17]);
xINVD1 I23_17_ ( rBL7[17], RAM_VDD, GND, net266[18]);
xINVD1 I23_16_ ( rBL7[16], RAM_VDD, GND, net266[19]);
xINVD1 I23_15_ ( rBL7[15], RAM_VDD, GND, net266[20]);
xINVD1 I23_14_ ( rBL7[14], RAM_VDD, GND, net266[21]);
xINVD1 I23_13_ ( rBL7[13], RAM_VDD, GND, net266[22]);
xINVD1 I23_12_ ( rBL7[12], RAM_VDD, GND, net266[23]);
xINVD1 I23_11_ ( rBL7[11], RAM_VDD, GND, net266[24]);
xINVD1 I23_10_ ( rBL7[10], RAM_VDD, GND, net266[25]);
xINVD1 I23_9_ ( rBL7[9], RAM_VDD, GND, net266[26]);
xINVD1 I23_8_ ( rBL7[8], RAM_VDD, GND, net266[27]);
xINVD1 I23_7_ ( rBL7[7], RAM_VDD, GND, net266[28]);
xINVD1 I23_6_ ( rBL7[6], RAM_VDD, GND, net266[29]);
xINVD1 I23_5_ ( rBL7[5], RAM_VDD, GND, net266[30]);
xINVD1 I23_4_ ( rBL7[4], RAM_VDD, GND, net266[31]);
xINVD1 I23_3_ ( rBL7[3], RAM_VDD, GND, net266[32]);
xINVD1 I23_2_ ( rBL7[2], RAM_VDD, GND, net266[33]);

```

```

xINVD1 I23_1_ ( rBL7[1], RAM_VDD, GND, net266[34]);
xINVD1 I23_0_ ( rBL7[0], RAM_VDD, GND, net266[35]);
xINVD1 I43_35_ ( rBL0[35], RAM_VDD, GND, net203[0]);
xINVD1 I43_34_ ( rBL0[34], RAM_VDD, GND, net203[1]);
xINVD1 I43_33_ ( rBL0[33], RAM_VDD, GND, net203[2]);
xINVD1 I43_32_ ( rBL0[32], RAM_VDD, GND, net203[3]);
xINVD1 I43_31_ ( rBL0[31], RAM_VDD, GND, net203[4]);
xINVD1 I43_30_ ( rBL0[30], RAM_VDD, GND, net203[5]);
xINVD1 I43_29_ ( rBL0[29], RAM_VDD, GND, net203[6]);
xINVD1 I43_28_ ( rBL0[28], RAM_VDD, GND, net203[7]);
xINVD1 I43_27_ ( rBL0[27], RAM_VDD, GND, net203[8]);
xINVD1 I43_26_ ( rBL0[26], RAM_VDD, GND, net203[9]);
xINVD1 I43_25_ ( rBL0[25], RAM_VDD, GND, net203[10]);
xINVD1 I43_24_ ( rBL0[24], RAM_VDD, GND, net203[11]);
xINVD1 I43_23_ ( rBL0[23], RAM_VDD, GND, net203[12]);
xINVD1 I43_22_ ( rBL0[22], RAM_VDD, GND, net203[13]);
xINVD1 I43_21_ ( rBL0[21], RAM_VDD, GND, net203[14]);
xINVD1 I43_20_ ( rBL0[20], RAM_VDD, GND, net203[15]);
xINVD1 I43_19_ ( rBL0[19], RAM_VDD, GND, net203[16]);
xINVD1 I43_18_ ( rBL0[18], RAM_VDD, GND, net203[17]);
xINVD1 I43_17_ ( rBL0[17], RAM_VDD, GND, net203[18]);
xINVD1 I43_16_ ( rBL0[16], RAM_VDD, GND, net203[19]);
xINVD1 I43_15_ ( rBL0[15], RAM_VDD, GND, net203[20]);
xINVD1 I43_14_ ( rBL0[14], RAM_VDD, GND, net203[21]);
xINVD1 I43_13_ ( rBL0[13], RAM_VDD, GND, net203[22]);
xINVD1 I43_12_ ( rBL0[12], RAM_VDD, GND, net203[23]);
xINVD1 I43_11_ ( rBL0[11], RAM_VDD, GND, net203[24]);
xINVD1 I43_10_ ( rBL0[10], RAM_VDD, GND, net203[25]);
xINVD1 I43_9_ ( rBL0[9], RAM_VDD, GND, net203[26]);
xINVD1 I43_8_ ( rBL0[8], RAM_VDD, GND, net203[27]);
xINVD1 I43_7_ ( rBL0[7], RAM_VDD, GND, net203[28]);
xINVD1 I43_6_ ( rBL0[6], RAM_VDD, GND, net203[29]);
xINVD1 I43_5_ ( rBL0[5], RAM_VDD, GND, net203[30]);
xINVD1 I43_4_ ( rBL0[4], RAM_VDD, GND, net203[31]);
xINVD1 I43_3_ ( rBL0[3], RAM_VDD, GND, net203[32]);
xINVD1 I43_2_ ( rBL0[2], RAM_VDD, GND, net203[33]);
xINVD1 I43_1_ ( rBL0[1], RAM_VDD, GND, net203[34]);
xINVD1 I43_0_ ( rBL0[0], RAM_VDD, GND, net203[35]);
xINVD1 I41_35_ ( rBL2[35], RAM_VDD, GND, net221[0]);
xINVD1 I41_34_ ( rBL2[34], RAM_VDD, GND, net221[1]);
xINVD1 I41_33_ ( rBL2[33], RAM_VDD, GND, net221[2]);
xINVD1 I41_32_ ( rBL2[32], RAM_VDD, GND, net221[3]);
xINVD1 I41_31_ ( rBL2[31], RAM_VDD, GND, net221[4]);
xINVD1 I41_30_ ( rBL2[30], RAM_VDD, GND, net221[5]);
xINVD1 I41_29_ ( rBL2[29], RAM_VDD, GND, net221[6]);
xINVD1 I41_28_ ( rBL2[28], RAM_VDD, GND, net221[7]);
xINVD1 I41_27_ ( rBL2[27], RAM_VDD, GND, net221[8]);
xINVD1 I41_26_ ( rBL2[26], RAM_VDD, GND, net221[9]);
xINVD1 I41_25_ ( rBL2[25], RAM_VDD, GND, net221[10]);
xINVD1 I41_24_ ( rBL2[24], RAM_VDD, GND, net221[11]);
xINVD1 I41_23_ ( rBL2[23], RAM_VDD, GND, net221[12]);
xINVD1 I41_22_ ( rBL2[22], RAM_VDD, GND, net221[13]);
xINVD1 I41_21_ ( rBL2[21], RAM_VDD, GND, net221[14]);
xINVD1 I41_20_ ( rBL2[20], RAM_VDD, GND, net221[15]);

```

```

xINVD1 I41_19_ ( rBL2[19], RAM_VDD, GND, net221[16]);
xINVD1 I41_18_ ( rBL2[18], RAM_VDD, GND, net221[17]);
xINVD1 I41_17_ ( rBL2[17], RAM_VDD, GND, net221[18]);
xINVD1 I41_16_ ( rBL2[16], RAM_VDD, GND, net221[19]);
xINVD1 I41_15_ ( rBL2[15], RAM_VDD, GND, net221[20]);
xINVD1 I41_14_ ( rBL2[14], RAM_VDD, GND, net221[21]);
xINVD1 I41_13_ ( rBL2[13], RAM_VDD, GND, net221[22]);
xINVD1 I41_12_ ( rBL2[12], RAM_VDD, GND, net221[23]);
xINVD1 I41_11_ ( rBL2[11], RAM_VDD, GND, net221[24]);
xINVD1 I41_10_ ( rBL2[10], RAM_VDD, GND, net221[25]);
xINVD1 I41_9_ ( rBL2[9], RAM_VDD, GND, net221[26]);
xINVD1 I41_8_ ( rBL2[8], RAM_VDD, GND, net221[27]);
xINVD1 I41_7_ ( rBL2[7], RAM_VDD, GND, net221[28]);
xINVD1 I41_6_ ( rBL2[6], RAM_VDD, GND, net221[29]);
xINVD1 I41_5_ ( rBL2[5], RAM_VDD, GND, net221[30]);
xINVD1 I41_4_ ( rBL2[4], RAM_VDD, GND, net221[31]);
xINVD1 I41_3_ ( rBL2[3], RAM_VDD, GND, net221[32]);
xINVD1 I41_2_ ( rBL2[2], RAM_VDD, GND, net221[33]);
xINVD1 I41_1_ ( rBL2[1], RAM_VDD, GND, net221[34]);
xINVD1 I41_0_ ( rBL2[0], RAM_VDD, GND, net221[35]);
xINVD1 I38_35_ ( rBL5[35], RAM_VDD, GND, net248[0]);
xINVD1 I38_34_ ( rBL5[34], RAM_VDD, GND, net248[1]);
xINVD1 I38_33_ ( rBL5[33], RAM_VDD, GND, net248[2]);
xINVD1 I38_32_ ( rBL5[32], RAM_VDD, GND, net248[3]);
xINVD1 I38_31_ ( rBL5[31], RAM_VDD, GND, net248[4]);
xINVD1 I38_30_ ( rBL5[30], RAM_VDD, GND, net248[5]);
xINVD1 I38_29_ ( rBL5[29], RAM_VDD, GND, net248[6]);
xINVD1 I38_28_ ( rBL5[28], RAM_VDD, GND, net248[7]);
xINVD1 I38_27_ ( rBL5[27], RAM_VDD, GND, net248[8]);
xINVD1 I38_26_ ( rBL5[26], RAM_VDD, GND, net248[9]);
xINVD1 I38_25_ ( rBL5[25], RAM_VDD, GND, net248[10]);
xINVD1 I38_24_ ( rBL5[24], RAM_VDD, GND, net248[11]);
xINVD1 I38_23_ ( rBL5[23], RAM_VDD, GND, net248[12]);
xINVD1 I38_22_ ( rBL5[22], RAM_VDD, GND, net248[13]);
xINVD1 I38_21_ ( rBL5[21], RAM_VDD, GND, net248[14]);
xINVD1 I38_20_ ( rBL5[20], RAM_VDD, GND, net248[15]);
xINVD1 I38_19_ ( rBL5[19], RAM_VDD, GND, net248[16]);
xINVD1 I38_18_ ( rBL5[18], RAM_VDD, GND, net248[17]);
xINVD1 I38_17_ ( rBL5[17], RAM_VDD, GND, net248[18]);
xINVD1 I38_16_ ( rBL5[16], RAM_VDD, GND, net248[19]);
xINVD1 I38_15_ ( rBL5[15], RAM_VDD, GND, net248[20]);
xINVD1 I38_14_ ( rBL5[14], RAM_VDD, GND, net248[21]);
xINVD1 I38_13_ ( rBL5[13], RAM_VDD, GND, net248[22]);
xINVD1 I38_12_ ( rBL5[12], RAM_VDD, GND, net248[23]);
xINVD1 I38_11_ ( rBL5[11], RAM_VDD, GND, net248[24]);
xINVD1 I38_10_ ( rBL5[10], RAM_VDD, GND, net248[25]);
xINVD1 I38_9_ ( rBL5[9], RAM_VDD, GND, net248[26]);
xINVD1 I38_8_ ( rBL5[8], RAM_VDD, GND, net248[27]);
xINVD1 I38_7_ ( rBL5[7], RAM_VDD, GND, net248[28]);
xINVD1 I38_6_ ( rBL5[6], RAM_VDD, GND, net248[29]);
xINVD1 I38_5_ ( rBL5[5], RAM_VDD, GND, net248[30]);
xINVD1 I38_4_ ( rBL5[4], RAM_VDD, GND, net248[31]);
xINVD1 I38_3_ ( rBL5[3], RAM_VDD, GND, net248[32]);
xINVD1 I38_2_ ( rBL5[2], RAM_VDD, GND, net248[33]);

```

```

xINVD1 I38_1_ ( rBL5[1], RAM_VDD, GND, net248[34]);
xINVD1 I38_0_ ( rBL5[0], RAM_VDD, GND, net248[35]);
xINVD1 I42_35_ ( rBL1[35], RAM_VDD, GND, net212[0]);
xINVD1 I42_34_ ( rBL1[34], RAM_VDD, GND, net212[1]);
xINVD1 I42_33_ ( rBL1[33], RAM_VDD, GND, net212[2]);
xINVD1 I42_32_ ( rBL1[32], RAM_VDD, GND, net212[3]);
xINVD1 I42_31_ ( rBL1[31], RAM_VDD, GND, net212[4]);
xINVD1 I42_30_ ( rBL1[30], RAM_VDD, GND, net212[5]);
xINVD1 I42_29_ ( rBL1[29], RAM_VDD, GND, net212[6]);
xINVD1 I42_28_ ( rBL1[28], RAM_VDD, GND, net212[7]);
xINVD1 I42_27_ ( rBL1[27], RAM_VDD, GND, net212[8]);
xINVD1 I42_26_ ( rBL1[26], RAM_VDD, GND, net212[9]);
xINVD1 I42_25_ ( rBL1[25], RAM_VDD, GND, net212[10]);
xINVD1 I42_24_ ( rBL1[24], RAM_VDD, GND, net212[11]);
xINVD1 I42_23_ ( rBL1[23], RAM_VDD, GND, net212[12]);
xINVD1 I42_22_ ( rBL1[22], RAM_VDD, GND, net212[13]);
xINVD1 I42_21_ ( rBL1[21], RAM_VDD, GND, net212[14]);
xINVD1 I42_20_ ( rBL1[20], RAM_VDD, GND, net212[15]);
xINVD1 I42_19_ ( rBL1[19], RAM_VDD, GND, net212[16]);
xINVD1 I42_18_ ( rBL1[18], RAM_VDD, GND, net212[17]);
xINVD1 I42_17_ ( rBL1[17], RAM_VDD, GND, net212[18]);
xINVD1 I42_16_ ( rBL1[16], RAM_VDD, GND, net212[19]);
xINVD1 I42_15_ ( rBL1[15], RAM_VDD, GND, net212[20]);
xINVD1 I42_14_ ( rBL1[14], RAM_VDD, GND, net212[21]);
xINVD1 I42_13_ ( rBL1[13], RAM_VDD, GND, net212[22]);
xINVD1 I42_12_ ( rBL1[12], RAM_VDD, GND, net212[23]);
xINVD1 I42_11_ ( rBL1[11], RAM_VDD, GND, net212[24]);
xINVD1 I42_10_ ( rBL1[10], RAM_VDD, GND, net212[25]);
xINVD1 I42_9_ ( rBL1[9], RAM_VDD, GND, net212[26]);
xINVD1 I42_8_ ( rBL1[8], RAM_VDD, GND, net212[27]);
xINVD1 I42_7_ ( rBL1[7], RAM_VDD, GND, net212[28]);
xINVD1 I42_6_ ( rBL1[6], RAM_VDD, GND, net212[29]);
xINVD1 I42_5_ ( rBL1[5], RAM_VDD, GND, net212[30]);
xINVD1 I42_4_ ( rBL1[4], RAM_VDD, GND, net212[31]);
xINVD1 I42_3_ ( rBL1[3], RAM_VDD, GND, net212[32]);
xINVD1 I42_2_ ( rBL1[2], RAM_VDD, GND, net212[33]);
xINVD1 I42_1_ ( rBL1[1], RAM_VDD, GND, net212[34]);
xINVD1 I42_0_ ( rBL1[0], RAM_VDD, GND, net212[35]);
xBUFFD1 I14 ( RSTbuf, RAM_VDD, GND, RST);
xBUFFD1 I31_35_ ( net137[0], RAM_VDD, GND, BL[35]);
xBUFFD1 I31_34_ ( net137[1], RAM_VDD, GND, BL[34]);
xBUFFD1 I31_33_ ( net137[2], RAM_VDD, GND, BL[33]);
xBUFFD1 I31_32_ ( net137[3], RAM_VDD, GND, BL[32]);
xBUFFD1 I31_31_ ( net137[4], RAM_VDD, GND, BL[31]);
xBUFFD1 I31_30_ ( net137[5], RAM_VDD, GND, BL[30]);
xBUFFD1 I31_29_ ( net137[6], RAM_VDD, GND, BL[29]);
xBUFFD1 I31_28_ ( net137[7], RAM_VDD, GND, BL[28]);
xBUFFD1 I31_27_ ( net137[8], RAM_VDD, GND, BL[27]);
xBUFFD1 I31_26_ ( net137[9], RAM_VDD, GND, BL[26]);
xBUFFD1 I31_25_ ( net137[10], RAM_VDD, GND, BL[25]);
xBUFFD1 I31_24_ ( net137[11], RAM_VDD, GND, BL[24]);
xBUFFD1 I31_23_ ( net137[12], RAM_VDD, GND, BL[23]);
xBUFFD1 I31_22_ ( net137[13], RAM_VDD, GND, BL[22]);
xBUFFD1 I31_21_ ( net137[14], RAM_VDD, GND, BL[21]);

```

```

xBUFFD1 I31_20_ ( net137[15], RAM_VDD, GND, BL[20]);
xBUFFD1 I31_19_ ( net137[16], RAM_VDD, GND, BL[19]);
xBUFFD1 I31_18_ ( net137[17], RAM_VDD, GND, BL[18]);
xBUFFD1 I31_17_ ( net137[18], RAM_VDD, GND, BL[17]);
xBUFFD1 I31_16_ ( net137[19], RAM_VDD, GND, BL[16]);
xBUFFD1 I31_15_ ( net137[20], RAM_VDD, GND, BL[15]);
xBUFFD1 I31_14_ ( net137[21], RAM_VDD, GND, BL[14]);
xBUFFD1 I31_13_ ( net137[22], RAM_VDD, GND, BL[13]);
xBUFFD1 I31_12_ ( net137[23], RAM_VDD, GND, BL[12]);
xBUFFD1 I31_11_ ( net137[24], RAM_VDD, GND, BL[11]);
xBUFFD1 I31_10_ ( net137[25], RAM_VDD, GND, BL[10]);
xBUFFD1 I31_9_ ( net137[26], RAM_VDD, GND, BL[9]);
xBUFFD1 I31_8_ ( net137[27], RAM_VDD, GND, BL[8]);
xBUFFD1 I31_7_ ( net137[28], RAM_VDD, GND, BL[7]);
xBUFFD1 I31_6_ ( net137[29], RAM_VDD, GND, BL[6]);
xBUFFD1 I31_5_ ( net137[30], RAM_VDD, GND, BL[5]);
xBUFFD1 I31_4_ ( net137[31], RAM_VDD, GND, BL[4]);
xBUFFD1 I31_3_ ( net137[32], RAM_VDD, GND, BL[3]);
xBUFFD1 I31_2_ ( net137[33], RAM_VDD, GND, BL[2]);
xBUFFD1 I31_1_ ( net137[34], RAM_VDD, GND, BL[1]);
xBUFFD1 I31_0_ ( net137[35], RAM_VDD, GND, BL[0]);
xBUFFD1 I32_31_ ( net190[0], RAM_VDD, GND, rowWL[31]);
xBUFFD1 I32_30_ ( net190[1], RAM_VDD, GND, rowWL[30]);
xBUFFD1 I32_29_ ( net190[2], RAM_VDD, GND, rowWL[29]);
xBUFFD1 I32_28_ ( net190[3], RAM_VDD, GND, rowWL[28]);
xBUFFD1 I32_27_ ( net190[4], RAM_VDD, GND, rowWL[27]);
xBUFFD1 I32_26_ ( net190[5], RAM_VDD, GND, rowWL[26]);
xBUFFD1 I32_25_ ( net190[6], RAM_VDD, GND, rowWL[25]);
xBUFFD1 I32_24_ ( net190[7], RAM_VDD, GND, rowWL[24]);
xBUFFD1 I32_23_ ( net190[8], RAM_VDD, GND, rowWL[23]);
xBUFFD1 I32_22_ ( net190[9], RAM_VDD, GND, rowWL[22]);
xBUFFD1 I32_21_ ( net190[10], RAM_VDD, GND, rowWL[21]);
xBUFFD1 I32_20_ ( net190[11], RAM_VDD, GND, rowWL[20]);
xBUFFD1 I32_19_ ( net190[12], RAM_VDD, GND, rowWL[19]);
xBUFFD1 I32_18_ ( net190[13], RAM_VDD, GND, rowWL[18]);
xBUFFD1 I32_17_ ( net190[14], RAM_VDD, GND, rowWL[17]);
xBUFFD1 I32_16_ ( net190[15], RAM_VDD, GND, rowWL[16]);
xBUFFD1 I32_15_ ( net190[16], RAM_VDD, GND, rowWL[15]);
xBUFFD1 I32_14_ ( net190[17], RAM_VDD, GND, rowWL[14]);
xBUFFD1 I32_13_ ( net190[18], RAM_VDD, GND, rowWL[13]);
xBUFFD1 I32_12_ ( net190[19], RAM_VDD, GND, rowWL[12]);
xBUFFD1 I32_11_ ( net190[20], RAM_VDD, GND, rowWL[11]);
xBUFFD1 I32_10_ ( net190[21], RAM_VDD, GND, rowWL[10]);
xBUFFD1 I32_9_ ( net190[22], RAM_VDD, GND, rowWL[9]);
xBUFFD1 I32_8_ ( net190[23], RAM_VDD, GND, rowWL[8]);
xBUFFD1 I32_7_ ( net190[24], RAM_VDD, GND, rowWL[7]);
xBUFFD1 I32_6_ ( net190[25], RAM_VDD, GND, rowWL[6]);
xBUFFD1 I32_5_ ( net190[26], RAM_VDD, GND, rowWL[5]);
xBUFFD1 I32_4_ ( net190[27], RAM_VDD, GND, rowWL[4]);
xBUFFD1 I32_3_ ( net190[28], RAM_VDD, GND, rowWL[3]);
xBUFFD1 I32_2_ ( net190[29], RAM_VDD, GND, rowWL[2]);
xBUFFD1 I32_1_ ( net190[30], RAM_VDD, GND, rowWL[1]);
xBUFFD1 I32_0_ ( net190[31], RAM_VDD, GND, rowWL[0]);
xBUFFD1 I33_31_ ( net194[0], RAM_VDD, GND, rowrWL[31]);

```

```

xBUFFD1 I33_30_ ( net194[1], RAM_VDD, GND, rowrWL[30]);
xBUFFD1 I33_29_ ( net194[2], RAM_VDD, GND, rowrWL[29]);
xBUFFD1 I33_28_ ( net194[3], RAM_VDD, GND, rowrWL[28]);
xBUFFD1 I33_27_ ( net194[4], RAM_VDD, GND, rowrWL[27]);
xBUFFD1 I33_26_ ( net194[5], RAM_VDD, GND, rowrWL[26]);
xBUFFD1 I33_25_ ( net194[6], RAM_VDD, GND, rowrWL[25]);
xBUFFD1 I33_24_ ( net194[7], RAM_VDD, GND, rowrWL[24]);
xBUFFD1 I33_23_ ( net194[8], RAM_VDD, GND, rowrWL[23]);
xBUFFD1 I33_22_ ( net194[9], RAM_VDD, GND, rowrWL[22]);
xBUFFD1 I33_21_ ( net194[10], RAM_VDD, GND, rowrWL[21]);
xBUFFD1 I33_20_ ( net194[11], RAM_VDD, GND, rowrWL[20]);
xBUFFD1 I33_19_ ( net194[12], RAM_VDD, GND, rowrWL[19]);
xBUFFD1 I33_18_ ( net194[13], RAM_VDD, GND, rowrWL[18]);
xBUFFD1 I33_17_ ( net194[14], RAM_VDD, GND, rowrWL[17]);
xBUFFD1 I33_16_ ( net194[15], RAM_VDD, GND, rowrWL[16]);
xBUFFD1 I33_15_ ( net194[16], RAM_VDD, GND, rowrWL[15]);
xBUFFD1 I33_14_ ( net194[17], RAM_VDD, GND, rowrWL[14]);
xBUFFD1 I33_13_ ( net194[18], RAM_VDD, GND, rowrWL[13]);
xBUFFD1 I33_12_ ( net194[19], RAM_VDD, GND, rowrWL[12]);
xBUFFD1 I33_11_ ( net194[20], RAM_VDD, GND, rowrWL[11]);
xBUFFD1 I33_10_ ( net194[21], RAM_VDD, GND, rowrWL[10]);
xBUFFD1 I33_9_ ( net194[22], RAM_VDD, GND, rowrWL[9]);
xBUFFD1 I33_8_ ( net194[23], RAM_VDD, GND, rowrWL[8]);
xBUFFD1 I33_7_ ( net194[24], RAM_VDD, GND, rowrWL[7]);
xBUFFD1 I33_6_ ( net194[25], RAM_VDD, GND, rowrWL[6]);
xBUFFD1 I33_5_ ( net194[26], RAM_VDD, GND, rowrWL[5]);
xBUFFD1 I33_4_ ( net194[27], RAM_VDD, GND, rowrWL[4]);
xBUFFD1 I33_3_ ( net194[28], RAM_VDD, GND, rowrWL[3]);
xBUFFD1 I33_2_ ( net194[29], RAM_VDD, GND, rowrWL[2]);
xBUFFD1 I33_1_ ( net194[30], RAM_VDD, GND, rowrWL[1]);
xBUFFD1 I33_0_ ( net194[31], RAM_VDD, GND, rowrWL[0]);

RAM_32W_schematic I8 ( net203[0:35], GND, RAM_VDD, net138[0:35],
    RSTbuf, WL[0], rWL[0], net200[0:31], net134[0:31]);
RAM_32W_schematic I7 ( net212[0:35], GND, RAM_VDD, net138[0:35],
    RSTbuf, WL[1], rWL[1], net200[0:31], net134[0:31]);
RAM_32W_schematic I6 ( net221[0:35], GND, RAM_VDD, net138[0:35],
    RSTbuf, WL[2], rWL[2], net200[0:31], net134[0:31]);
RAM_32W_schematic I5 ( net230[0:35], GND, RAM_VDD, net138[0:35],
    RSTbuf, WL[3], rWL[3], net200[0:31], net134[0:31]);
RAM_32W_schematic I4 ( net239[0:35], GND, RAM_VDD, net138[0:35],
    RSTbuf, WL[4], rWL[4], net200[0:31], net134[0:31]);
RAM_32W_schematic I3 ( net248[0:35], GND, RAM_VDD, net138[0:35],
    RSTbuf, WL[5], rWL[5], net200[0:31], net134[0:31]);
RAM_32W_schematic I2 ( net257[0:35], GND, RAM_VDD, net138[0:35],
    RSTbuf, WL[6], rWL[6], net200[0:31], net134[0:31]);
RAM_32W_schematic IO ( net266[0:35], GND, RAM_VDD, net138[0:35],
    RSTbuf, WL[7], rWL[7], net200[0:31], net134[0:31]);

endmodule

module RAM_32W_schematic ( rBLout, GND, RAM_VDD, BLin, RST, WL, rWL,
    rowWL, rowrWL );

    inout GND, RAM_VDD;

```

```

input  RST, WL, rWL;

output [35:0]  rBLout;

input [31:0]  rowrWL;
input [35:0]  BLin;
input [31:0]  rowrWL;

// Buses in the design

wire [35:0]  BLbuf;

wire [35:0]  nBLbuf;

wire [35:0]  rBLi;

wire [31:0]  rowrWLout;

wire [31:0]  rowWLout;

specify
    specparam CDS_LIBNAME = "RAM";
    specparam CDS_CELLNAME = "32W";
    specparam CDS_VIEWNAME = "schematic";
endspecify

word I21 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[22],
    nBLbuf[35:0], rowrWLout[22]);
word I22 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[18],
    nBLbuf[35:0], rowrWLout[18]);
word I23 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[26],
    nBLbuf[35:0], rowrWLout[26]);
word I24 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[30],
    nBLbuf[35:0], rowrWLout[30]);
word I25 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[21],
    nBLbuf[35:0], rowrWLout[21]);
word I26 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[17],
    nBLbuf[35:0], rowrWLout[17]);
word I27 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[25],
    nBLbuf[35:0], rowrWLout[25]);
word I28 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[29],
    nBLbuf[35:0], rowrWLout[29]);
word I29 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[20],
    nBLbuf[35:0], rowrWLout[20]);
word I30 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[16],
    nBLbuf[35:0], rowrWLout[16]);
word I31 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[24],
    nBLbuf[35:0], rowrWLout[24]);
word I32 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[28],
    nBLbuf[35:0], rowrWLout[28]);
word I19 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[23],
    nBLbuf[35:0], rowrWLout[23]);
word I20 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[19],
    nBLbuf[35:0], rowrWLout[19]);

word I18 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[27],
    nBLbuf[35:0], rowrWLout[27]);
word I33 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[12],
    nBLbuf[35:0], rowrWLout[12]);
word I34 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[8],
    nBLbuf[35:0], rowrWLout[8]);
word I35 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[0],
    nBLbuf[35:0], rowrWLout[0]);
word I36 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[4],
    nBLbuf[35:0], rowrWLout[4]);
word I37 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[13],
    nBLbuf[35:0], rowrWLout[13]);
word I38 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[9],
    nBLbuf[35:0], rowrWLout[9]);
word I39 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[1],
    nBLbuf[35:0], rowrWLout[1]);
word I40 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[5],
    nBLbuf[35:0], rowrWLout[5]);
word I41 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[14],
    nBLbuf[35:0], rowrWLout[14]);
word I42 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[10],
    nBLbuf[35:0], rowrWLout[10]);
word I43 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[2],
    nBLbuf[35:0], rowrWLout[2]);
word I44 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[6],
    nBLbuf[35:0], rowrWLout[6]);
word I45 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[7],
    nBLbuf[35:0], rowrWLout[7]);
word I46 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[3],
    nBLbuf[35:0], rowrWLout[3]);
word I47 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[11],
    nBLbuf[35:0], rowrWLout[11]);
word I48 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[15],
    nBLbuf[35:0], rowrWLout[15]);
word I2 ( GND, RAM_VDD, rBLi[35:0], BLbuf[35:0], RSTout, rowWLout[31],
    nBLbuf[35:0], rowrWLout[31]);

RAMIO I1 ( BLbuf[35:0], RSTout, nBLbuf[35:0], rBLout[35:0],
    rowWLout[31:0], rowrWLout[31:0], GND, RAM_VDD, rBLi[35:0],
    BLin[35:0], RST, WL, rWL, rowWL[31:0], rowrWL[31:0]);

endmodule

module RAMIO ( BLbuf, RSTout, nBLbuf, rBLout, rowWLout, rowrWLout, GND,
    RAM_VDD, rBLi, BLin, RST, WL, rWL, rowWL, rowrWL );

output  RSTout;

inout  GND, RAM_VDD;

input  RST, WL, rWL;

output [35:0]  rBLout;
output [35:0]  nBLbuf;
output [31:0]  rowWLout;
output [31:0]  rowrWLout;

```

```

output [35:0] BLbuf;

inout [35:0] rBLi;

input [31:0] rowrWL;
input [35:0] BLin;
input [31:0] rowrWL;

// Buses in the design

wire [0:31] net084;

wire [0:35] net73;

specify
    specparam CDS_LIBNAME = "RAM";
    specparam CDS_CELLNAME = "RAMIO";
    specparam CDS_VIEWNAME = "schematic";
endspecify

xBUFFD3 I13 ( net48, RAM_VDD, GND, WL);
xBUFFD3 I11 ( RSTout, RAM_VDD, GND, RST);
xBUFFD3 I4 ( net59, RAM_VDD, GND, rWL);
xBUFFD2 rowrWLbuf_31_ ( rowrWLout[31], RAM_VDD, GND, net084[0]);
xBUFFD2 rowrWLbuf_30_ ( rowrWLout[30], RAM_VDD, GND, net084[1]);
xBUFFD2 rowrWLbuf_29_ ( rowrWLout[29], RAM_VDD, GND, net084[2]);
xBUFFD2 rowrWLbuf_28_ ( rowrWLout[28], RAM_VDD, GND, net084[3]);
xBUFFD2 rowrWLbuf_27_ ( rowrWLout[27], RAM_VDD, GND, net084[4]);
xBUFFD2 rowrWLbuf_26_ ( rowrWLout[26], RAM_VDD, GND, net084[5]);
xBUFFD2 rowrWLbuf_25_ ( rowrWLout[25], RAM_VDD, GND, net084[6]);
xBUFFD2 rowrWLbuf_24_ ( rowrWLout[24], RAM_VDD, GND, net084[7]);
xBUFFD2 rowrWLbuf_23_ ( rowrWLout[23], RAM_VDD, GND, net084[8]);
xBUFFD2 rowrWLbuf_22_ ( rowrWLout[22], RAM_VDD, GND, net084[9]);
xBUFFD2 rowrWLbuf_21_ ( rowrWLout[21], RAM_VDD, GND, net084[10]);
xBUFFD2 rowrWLbuf_20_ ( rowrWLout[20], RAM_VDD, GND, net084[11]);
xBUFFD2 rowrWLbuf_19_ ( rowrWLout[19], RAM_VDD, GND, net084[12]);
xBUFFD2 rowrWLbuf_18_ ( rowrWLout[18], RAM_VDD, GND, net084[13]);
xBUFFD2 rowrWLbuf_17_ ( rowrWLout[17], RAM_VDD, GND, net084[14]);
xBUFFD2 rowrWLbuf_16_ ( rowrWLout[16], RAM_VDD, GND, net084[15]);
xBUFFD2 rowrWLbuf_15_ ( rowrWLout[15], RAM_VDD, GND, net084[16]);
xBUFFD2 rowrWLbuf_14_ ( rowrWLout[14], RAM_VDD, GND, net084[17]);
xBUFFD2 rowrWLbuf_13_ ( rowrWLout[13], RAM_VDD, GND, net084[18]);
xBUFFD2 rowrWLbuf_12_ ( rowrWLout[12], RAM_VDD, GND, net084[19]);
xBUFFD2 rowrWLbuf_11_ ( rowrWLout[11], RAM_VDD, GND, net084[20]);
xBUFFD2 rowrWLbuf_10_ ( rowrWLout[10], RAM_VDD, GND, net084[21]);
xBUFFD2 rowrWLbuf_9_ ( rowrWLout[9], RAM_VDD, GND, net084[22]);
xBUFFD2 rowrWLbuf_8_ ( rowrWLout[8], RAM_VDD, GND, net084[23]);
xBUFFD2 rowrWLbuf_7_ ( rowrWLout[7], RAM_VDD, GND, net084[24]);
xBUFFD2 rowrWLbuf_6_ ( rowrWLout[6], RAM_VDD, GND, net084[25]);
xBUFFD2 rowrWLbuf_5_ ( rowrWLout[5], RAM_VDD, GND, net084[26]);
xBUFFD2 rowrWLbuf_4_ ( rowrWLout[4], RAM_VDD, GND, net084[27]);
xBUFFD2 rowrWLbuf_3_ ( rowrWLout[3], RAM_VDD, GND, net084[28]);
xBUFFD2 rowrWLbuf_2_ ( rowrWLout[2], RAM_VDD, GND, net084[29]);

xBUFFD2 rowrWLbuf_1_ ( rowrWLout[1], RAM_VDD, GND, net084[30]);
xBUFFD2 rowrWLbuf_0_ ( rowrWLout[0], RAM_VDD, GND, net084[31]);
xBUFFD2 BLbuffer_35_ ( BLbuf[35], RAM_VDD, GND, net73[0]);
xBUFFD2 BLbuffer_34_ ( BLbuf[34], RAM_VDD, GND, net73[1]);
xBUFFD2 BLbuffer_33_ ( BLbuf[33], RAM_VDD, GND, net73[2]);
xBUFFD2 BLbuffer_32_ ( BLbuf[32], RAM_VDD, GND, net73[3]);
xBUFFD2 BLbuffer_31_ ( BLbuf[31], RAM_VDD, GND, net73[4]);
xBUFFD2 BLbuffer_30_ ( BLbuf[30], RAM_VDD, GND, net73[5]);
xBUFFD2 BLbuffer_29_ ( BLbuf[29], RAM_VDD, GND, net73[6]);
xBUFFD2 BLbuffer_28_ ( BLbuf[28], RAM_VDD, GND, net73[7]);
xBUFFD2 BLbuffer_27_ ( BLbuf[27], RAM_VDD, GND, net73[8]);
xBUFFD2 BLbuffer_26_ ( BLbuf[26], RAM_VDD, GND, net73[9]);
xBUFFD2 BLbuffer_25_ ( BLbuf[25], RAM_VDD, GND, net73[10]);
xBUFFD2 BLbuffer_24_ ( BLbuf[24], RAM_VDD, GND, net73[11]);
xBUFFD2 BLbuffer_23_ ( BLbuf[23], RAM_VDD, GND, net73[12]);
xBUFFD2 BLbuffer_22_ ( BLbuf[22], RAM_VDD, GND, net73[13]);
xBUFFD2 BLbuffer_21_ ( BLbuf[21], RAM_VDD, GND, net73[14]);
xBUFFD2 BLbuffer_20_ ( BLbuf[20], RAM_VDD, GND, net73[15]);
xBUFFD2 BLbuffer_19_ ( BLbuf[19], RAM_VDD, GND, net73[16]);
xBUFFD2 BLbuffer_18_ ( BLbuf[18], RAM_VDD, GND, net73[17]);
xBUFFD2 BLbuffer_17_ ( BLbuf[17], RAM_VDD, GND, net73[18]);
xBUFFD2 BLbuffer_16_ ( BLbuf[16], RAM_VDD, GND, net73[19]);
xBUFFD2 BLbuffer_15_ ( BLbuf[15], RAM_VDD, GND, net73[20]);
xBUFFD2 BLbuffer_14_ ( BLbuf[14], RAM_VDD, GND, net73[21]);
xBUFFD2 BLbuffer_13_ ( BLbuf[13], RAM_VDD, GND, net73[22]);
xBUFFD2 BLbuffer_12_ ( BLbuf[12], RAM_VDD, GND, net73[23]);
xBUFFD2 BLbuffer_11_ ( BLbuf[11], RAM_VDD, GND, net73[24]);
xBUFFD2 BLbuffer_10_ ( BLbuf[10], RAM_VDD, GND, net73[25]);
xBUFFD2 BLbuffer_9_ ( BLbuf[9], RAM_VDD, GND, net73[26]);
xBUFFD2 BLbuffer_8_ ( BLbuf[8], RAM_VDD, GND, net73[27]);
xBUFFD2 BLbuffer_7_ ( BLbuf[7], RAM_VDD, GND, net73[28]);
xBUFFD2 BLbuffer_6_ ( BLbuf[6], RAM_VDD, GND, net73[29]);
xBUFFD2 BLbuffer_5_ ( BLbuf[5], RAM_VDD, GND, net73[30]);
xBUFFD2 BLbuffer_4_ ( BLbuf[4], RAM_VDD, GND, net73[31]);
xBUFFD2 BLbuffer_3_ ( BLbuf[3], RAM_VDD, GND, net73[32]);
xBUFFD2 BLbuffer_2_ ( BLbuf[2], RAM_VDD, GND, net73[33]);
xBUFFD2 BLbuffer_1_ ( BLbuf[1], RAM_VDD, GND, net73[34]);
xBUFFD2 BLbuffer_0_ ( BLbuf[0], RAM_VDD, GND, net73[35]);

xINVD2 I9 ( net63, RAM_VDD, GND, net59);
xINVD2 BLinv_35_ ( nBLbuf[35], RAM_VDD, GND, net73[0]);
xINVD2 BLinv_34_ ( nBLbuf[34], RAM_VDD, GND, net73[1]);
xINVD2 BLinv_33_ ( nBLbuf[33], RAM_VDD, GND, net73[2]);
xINVD2 BLinv_32_ ( nBLbuf[32], RAM_VDD, GND, net73[3]);
xINVD2 BLinv_31_ ( nBLbuf[31], RAM_VDD, GND, net73[4]);
xINVD2 BLinv_30_ ( nBLbuf[30], RAM_VDD, GND, net73[5]);
xINVD2 BLinv_29_ ( nBLbuf[29], RAM_VDD, GND, net73[6]);
xINVD2 BLinv_28_ ( nBLbuf[28], RAM_VDD, GND, net73[7]);
xINVD2 BLinv_27_ ( nBLbuf[27], RAM_VDD, GND, net73[8]);
xINVD2 BLinv_26_ ( nBLbuf[26], RAM_VDD, GND, net73[9]);
xINVD2 BLinv_25_ ( nBLbuf[25], RAM_VDD, GND, net73[10]);
xINVD2 BLinv_24_ ( nBLbuf[24], RAM_VDD, GND, net73[11]);
xINVD2 BLinv_23_ ( nBLbuf[23], RAM_VDD, GND, net73[12]);
xINVD2 BLinv_22_ ( nBLbuf[22], RAM_VDD, GND, net73[13]);
xINVD2 BLinv_21_ ( nBLbuf[21], RAM_VDD, GND, net73[14]);

```



```

xNR2D0 WLnor_12_ ( rowWLout[12], RAM_VDD, GND, rowWL[12], net48);
xNR2D0 WLnor_11_ ( rowWLout[11], RAM_VDD, GND, rowWL[11], net48);
xNR2D0 WLnor_10_ ( rowWLout[10], RAM_VDD, GND, rowWL[10], net48);
xNR2D0 WLnor_9_ ( rowWLout[9], RAM_VDD, GND, rowWL[9], net48);
xNR2D0 WLnor_8_ ( rowWLout[8], RAM_VDD, GND, rowWL[8], net48);
xNR2D0 WLnor_7_ ( rowWLout[7], RAM_VDD, GND, rowWL[7], net48);
xNR2D0 WLnor_6_ ( rowWLout[6], RAM_VDD, GND, rowWL[6], net48);
xNR2D0 WLnor_5_ ( rowWLout[5], RAM_VDD, GND, rowWL[5], net48);
xNR2D0 WLnor_4_ ( rowWLout[4], RAM_VDD, GND, rowWL[4], net48);
xNR2D0 WLnor_3_ ( rowWLout[3], RAM_VDD, GND, rowWL[3], net48);
xNR2D0 WLnor_2_ ( rowWLout[2], RAM_VDD, GND, rowWL[2], net48);
xNR2D0 WLnor_1_ ( rowWLout[1], RAM_VDD, GND, rowWL[1], net48);
xNR2D0 WLnor_0_ ( rowWLout[0], RAM_VDD, GND, rowWL[0], net48);
charge_inv charge_inv_35_ ( rBLout[35], GND, RAM_VDD, rBLi[35], net63);
charge_inv charge_inv_34_ ( rBLout[34], GND, RAM_VDD, rBLi[34], net63);
charge_inv charge_inv_33_ ( rBLout[33], GND, RAM_VDD, rBLi[33], net63);
charge_inv charge_inv_32_ ( rBLout[32], GND, RAM_VDD, rBLi[32], net63);
charge_inv charge_inv_31_ ( rBLout[31], GND, RAM_VDD, rBLi[31], net63);
charge_inv charge_inv_30_ ( rBLout[30], GND, RAM_VDD, rBLi[30], net63);
charge_inv charge_inv_29_ ( rBLout[29], GND, RAM_VDD, rBLi[29], net63);
charge_inv charge_inv_28_ ( rBLout[28], GND, RAM_VDD, rBLi[28], net63);
charge_inv charge_inv_27_ ( rBLout[27], GND, RAM_VDD, rBLi[27], net63);
charge_inv charge_inv_26_ ( rBLout[26], GND, RAM_VDD, rBLi[26], net63);
charge_inv charge_inv_25_ ( rBLout[25], GND, RAM_VDD, rBLi[25], net63);
charge_inv charge_inv_24_ ( rBLout[24], GND, RAM_VDD, rBLi[24], net63);
charge_inv charge_inv_23_ ( rBLout[23], GND, RAM_VDD, rBLi[23], net63);
charge_inv charge_inv_22_ ( rBLout[22], GND, RAM_VDD, rBLi[22], net63);
charge_inv charge_inv_21_ ( rBLout[21], GND, RAM_VDD, rBLi[21], net63);
charge_inv charge_inv_20_ ( rBLout[20], GND, RAM_VDD, rBLi[20], net63);
charge_inv charge_inv_19_ ( rBLout[19], GND, RAM_VDD, rBLi[19], net63);
charge_inv charge_inv_18_ ( rBLout[18], GND, RAM_VDD, rBLi[18], net63);
charge_inv charge_inv_17_ ( rBLout[17], GND, RAM_VDD, rBLi[17], net63);
charge_inv charge_inv_16_ ( rBLout[16], GND, RAM_VDD, rBLi[16], net63);
charge_inv charge_inv_15_ ( rBLout[15], GND, RAM_VDD, rBLi[15], net63);
charge_inv charge_inv_14_ ( rBLout[14], GND, RAM_VDD, rBLi[14], net63);
charge_inv charge_inv_13_ ( rBLout[13], GND, RAM_VDD, rBLi[13], net63);
charge_inv charge_inv_12_ ( rBLout[12], GND, RAM_VDD, rBLi[12], net63);
charge_inv charge_inv_11_ ( rBLout[11], GND, RAM_VDD, rBLi[11], net63);
charge_inv charge_inv_10_ ( rBLout[10], GND, RAM_VDD, rBLi[10], net63);
charge_inv charge_inv_9_ ( rBLout[9], GND, RAM_VDD, rBLi[9], net63);
charge_inv charge_inv_8_ ( rBLout[8], GND, RAM_VDD, rBLi[8], net63);
charge_inv charge_inv_7_ ( rBLout[7], GND, RAM_VDD, rBLi[7], net63);
charge_inv charge_inv_6_ ( rBLout[6], GND, RAM_VDD, rBLi[6], net63);
charge_inv charge_inv_5_ ( rBLout[5], GND, RAM_VDD, rBLi[5], net63);
charge_inv charge_inv_4_ ( rBLout[4], GND, RAM_VDD, rBLi[4], net63);
charge_inv charge_inv_3_ ( rBLout[3], GND, RAM_VDD, rBLi[3], net63);
charge_inv charge_inv_2_ ( rBLout[2], GND, RAM_VDD, rBLi[2], net63);
charge_inv charge_inv_1_ ( rBLout[1], GND, RAM_VDD, rBLi[1], net63);
charge_inv charge_inv_0_ ( rBLout[0], GND, RAM_VDD, rBLi[0], net63);

endmodule

module charge_inv ( rBLout, GND, RAM_VDD, rBL, rWL );
output rBLout;

inout GND, RAM_VDD, rBL;

input rWL;

specify
    specparam CDS_LIBNAME = "RAM";
    specparam CDS_CELLNAME = "charge_inv";
    specparam CDS_VIEWNAME = "schematic";
endspecify

pch M4 ( .D(rBL), .B(RAM_VDD), .G(rWL), .S(RAM_VDD));
pch M2 ( .D(rBLout), .B(RAM_VDD), .G(rBL), .S(RAM_VDD));
nch M0 ( .D(rBLout), .B(GND), .G(rBL), .S(GND));

endmodule

module xNR2D0 ( ZN, VDD, VSS, A1, A2 );
output ZN;

inout VDD, VSS;

input A1, A2;

specify
    specparam CDS_LIBNAME = "tcn90ghp";
    specparam CDS_CELLNAME = "NR2D0";
    specparam CDS_VIEWNAME = "schematic";
endspecify

nch Inst_0 ( .D(ZN), .B(VSS), .G(A1), .S(VSS));
nch Inst_1 ( .D(ZN), .B(VSS), .G(A2), .S(VSS));
pch Inst_2 ( .D(ZN), .B(VDD), .G(A1), .S(net13));
pch Inst_3 ( .D(net13), .B(VDD), .G(A2), .S(VDD));

endmodule

module xNR3D0 ( ZN, VDD, VSS, A1, A2, A3 );
output ZN;

inout VDD, VSS;

input A1, A2, A3;

specify
    specparam CDS_LIBNAME = "tcn90ghp";
    specparam CDS_CELLNAME = "NR3D0";
    specparam CDS_VIEWNAME = "schematic";
endspecify

nch M_u4 ( .D(ZN), .B(VSS), .G(A3), .S(VSS));
nch MI2 ( .D(ZN), .B(VSS), .G(A2), .S(VSS));
nch MI3 ( .D(ZN), .B(VSS), .G(A1), .S(VSS));
pch MI0 ( .D(net13), .B(VDD), .G(A2), .S(net17));

```



```

pch M_u1 ( .D(net17), .B(VDD), .G(A3), .S(VDD));
pch MI1 ( .D(ZN), .B(VDD), .G(A1), .S(net13));

endmodule

module xBUFFD2 ( Z, VDD, VSS, I );
output Z;

inout VDD, VSS;

input I;

specify
    specparam CDS_LIBNAME = "tcbn90ghp";
    specparam CDS_CELLNAME = "BUFFD2";
    specparam CDS_VIEWNAME = "schematic";
endspecify

pch Inst_0 ( .D(net11), .B(VDD), .G(I), .S(VDD));
pch Inst_1 ( .D(Z), .B(VDD), .G(net11), .S(VDD));
pch Inst_2 ( .D(Z), .B(VDD), .G(net11), .S(VDD));
nch Inst_3 ( .D(Z), .B(VSS), .G(net11), .S(VSS));
nch Inst_4 ( .D(net11), .B(VSS), .G(I), .S(VSS));
nch Inst_5 ( .D(Z), .B(VSS), .G(net11), .S(VSS));

endmodule

module xBUFFD3 ( Z, VDD, VSS, I );
output Z;

inout VDD, VSS;

input I;

specify
    specparam CDS_LIBNAME = "tcbn90ghp";
    specparam CDS_CELLNAME = "BUFFD3";
    specparam CDS_VIEWNAME = "schematic";
endspecify

pch Inst_0 ( .D(net11), .B(VDD), .G(I), .S(VDD));
pch Inst_1 ( .D(Z), .B(VDD), .G(net11), .S(VDD));
pch Inst_2 ( .D(Z), .B(VDD), .G(net11), .S(VDD));
pch Inst_3 ( .D(Z), .B(VDD), .G(net11), .S(VDD));
nch Inst_4 ( .D(Z), .B(VSS), .G(net11), .S(VSS));
nch Inst_5 ( .D(net11), .B(VSS), .G(I), .S(VSS));
nch Inst_6 ( .D(Z), .B(VSS), .G(net11), .S(VSS));
nch Inst_7 ( .D(Z), .B(VSS), .G(net11), .S(VSS));

endmodule

module word ( GND, RAM_VDD, rBL, BL, RST, WL, nBL, rWL );
inout GND, RAM_VDD;

input RST, WL, rWL;

```

```

inout [35:0] rBL;

input [35:0] BL;
input [35:0] nBL;

specify
    specparam CDS_LIBNAME = "RAM";
    specparam CDS_CELLNAME = "word";
    specparam CDS_VIEWNAME = "schematic";
endspecify

reset I1 ( net20, GND, RAM_VDD, net22, RST, WL);
bitcell bit_35_ ( GND, net22, rBL[35], BL[35], net20, nBL[35], rWL);
bitcell bit_34_ ( GND, net22, rBL[34], BL[34], net20, nBL[34], rWL);
bitcell bit_33_ ( GND, net22, rBL[33], BL[33], net20, nBL[33], rWL);
bitcell bit_32_ ( GND, net22, rBL[32], BL[32], net20, nBL[32], rWL);
bitcell bit_31_ ( GND, net22, rBL[31], BL[31], net20, nBL[31], rWL);
bitcell bit_30_ ( GND, net22, rBL[30], BL[30], net20, nBL[30], rWL);
bitcell bit_29_ ( GND, net22, rBL[29], BL[29], net20, nBL[29], rWL);
bitcell bit_28_ ( GND, net22, rBL[28], BL[28], net20, nBL[28], rWL);
bitcell bit_27_ ( GND, net22, rBL[27], BL[27], net20, nBL[27], rWL);
bitcell bit_26_ ( GND, net22, rBL[26], BL[26], net20, nBL[26], rWL);
bitcell bit_25_ ( GND, net22, rBL[25], BL[25], net20, nBL[25], rWL);
bitcell bit_24_ ( GND, net22, rBL[24], BL[24], net20, nBL[24], rWL);
bitcell bit_23_ ( GND, net22, rBL[23], BL[23], net20, nBL[23], rWL);
bitcell bit_22_ ( GND, net22, rBL[22], BL[22], net20, nBL[22], rWL);
bitcell bit_21_ ( GND, net22, rBL[21], BL[21], net20, nBL[21], rWL);
bitcell bit_20_ ( GND, net22, rBL[20], BL[20], net20, nBL[20], rWL);
bitcell bit_19_ ( GND, net22, rBL[19], BL[19], net20, nBL[19], rWL);
bitcell bit_18_ ( GND, net22, rBL[18], BL[18], net20, nBL[18], rWL);
bitcell bit_17_ ( GND, net22, rBL[17], BL[17], net20, nBL[17], rWL);
bitcell bit_16_ ( GND, net22, rBL[16], BL[16], net20, nBL[16], rWL);
bitcell bit_15_ ( GND, net22, rBL[15], BL[15], net20, nBL[15], rWL);
bitcell bit_14_ ( GND, net22, rBL[14], BL[14], net20, nBL[14], rWL);
bitcell bit_13_ ( GND, net22, rBL[13], BL[13], net20, nBL[13], rWL);
bitcell bit_12_ ( GND, net22, rBL[12], BL[12], net20, nBL[12], rWL);
bitcell bit_11_ ( GND, net22, rBL[11], BL[11], net20, nBL[11], rWL);
bitcell bit_10_ ( GND, net22, rBL[10], BL[10], net20, nBL[10], rWL);
bitcell bit_9_ ( GND, net22, rBL[9], BL[9], net20, nBL[9], rWL);
bitcell bit_8_ ( GND, net22, rBL[8], BL[8], net20, nBL[8], rWL);
bitcell bit_7_ ( GND, net22, rBL[7], BL[7], net20, nBL[7], rWL);
bitcell bit_6_ ( GND, net22, rBL[6], BL[6], net20, nBL[6], rWL);
bitcell bit_5_ ( GND, net22, rBL[5], BL[5], net20, nBL[5], rWL);
bitcell bit_4_ ( GND, net22, rBL[4], BL[4], net20, nBL[4], rWL);
bitcell bit_3_ ( GND, net22, rBL[3], BL[3], net20, nBL[3], rWL);
bitcell bit_2_ ( GND, net22, rBL[2], BL[2], net20, nBL[2], rWL);
bitcell bit_1_ ( GND, net22, rBL[1], BL[1], net20, nBL[1], rWL);
bitcell bit_0_ ( GND, net22, rBL[0], BL[0], net20, nBL[0], rWL);

endmodule

module bitcell ( GND, VVDD, rBL, BL, WL, nBL, rWL );
input GND, VVDD;

```

```

output rBL;
input  BL, WL, nBL, rWL;
reg    store;

specify
    specparam CDS_LIBNAME  = "RAM";
    specparam CDS_CELLNAME = "bitcell";
    specparam CDS_VIEWNAME = "schematic";
endspecify
/*
pch  M12 ( .D(net24), .B(VVDD), .G(net44), .S(VVDD));
pch  M15 ( .D(net055), .B(VVDD), .G(net44), .S(VVDD));
pch  M11 ( .D(net44), .B(VVDD), .G(net24), .S(VVDD));
nch  M14 ( .D(nBL), .B(GND), .G(WL), .S(net44));
nch  M13 ( .D(net24), .B(GND), .G(WL), .S(BL));
nch  M16 ( .D(net059), .B(GND), .G(net44), .S(GND));
nch  M18 ( .D(rBL), .B(GND), .G(rWL), .S(net055));
nch  M19 ( .D(net055), .B(GND), .G(rWL), .S(net059));
nch  M3  ( .D(net24), .B(GND), .G(net44), .S(GND));
nch  M0  ( .D(net44), .B(GND), .G(net24), .S(GND));
*/
always @(*)
    if (WL)
        store <= BL;

assign rBL = rWL ? store:1'bz;

endmodule

module reset ( WLout, GND, RAM_VDD, VVDD, RST, WLin );
output WLout;

inout  GND, RAM_VDD, VVDD;

input  RST, WLin;

specify
    specparam CDS_LIBNAME  = "RAM";
    specparam CDS_CELLNAME = "reset";
    specparam CDS_VIEWNAME = "schematic";
endspecify

pch  M8 ( .D(net31), .B(RAM_VDD), .G(WLin), .S(net22));
pch  M9 ( .D(WLout), .B(RAM_VDD), .G(net31), .S(RAM_VDD));
pch  M7 ( .D(net22), .B(RAM_VDD), .G(RST), .S(RAM_VDD));

```

```

pch  M10 ( .D(VVDD), .B(RAM_VDD), .G(WLout), .S(RAM_VDD));
nch  M3  ( .D(WLout), .B(GND), .G(net31), .S(GND));
nch  M1  ( .D(net31), .B(GND), .G(RST), .S(GND));
nch  M0  ( .D(net31), .B(GND), .G(WLin), .S(GND));

endmodule

module xINVD2 ( ZN, VDD, VSS, I );
output ZN;

inout  VDD, VSS;

input  I;

specify
    specparam CDS_LIBNAME  = "tcbn90ghp";
    specparam CDS_CELLNAME = "INVD2";
    specparam CDS_VIEWNAME = "schematic";
endspecify

nch  Inst_0 ( .D(ZN), .B(VSS), .G(I), .S(VSS));
pch  Inst_1 ( .D(ZN), .B(VDD), .G(I), .S(VDD));

endmodule

module xBUFFD1 ( Z, VDD, VSS, I );
output Z;

inout  VDD, VSS;

input  I;

specify
    specparam CDS_LIBNAME  = "tcbn90ghp";
    specparam CDS_CELLNAME = "BUFFD1";
    specparam CDS_VIEWNAME = "schematic";
endspecify

nch  Inst_0 ( .D(Z), .B(VSS), .G(net5), .S(VSS));
nch  Inst_1 ( .D(net5), .B(VSS), .G(I), .S(VSS));
pch  Inst_2 ( .D(Z), .B(VDD), .G(net5), .S(VDD));
pch  Inst_3 ( .D(net5), .B(VDD), .G(I), .S(VDD));

endmodule
/**/

```

C.7 ADC Controller

```

//-----
// ADC Control Unit
//

```

```

// By: Ryan Wu, University of Calgary
// Last Modified: April 30, 2012
//
// This component is added to chip late into the layout
// phase, hence has not been integrated into the Memory
// Controller. It is a decoder that controls the ADC's
// inputs and outputs, and latches the digital signals.
//-----
`timescale 1ns / 1ns
// top level design includes both mips processor and memory
module ADC_Control(load, select_in, select_out, digital_in, digital_out);

    input          load;
    input    [2:0] select_in;
    output    [7:0] select_out;
    input    [7:0] digital_in;
    output reg [7:0] digital_out;

    assign select_out[0] = ~|select_in[2:0];
    assign select_out[1] = ~|select_in[2:1] & select_in[0];
    assign select_out[2] = ~select_in[2] & select_in[1] & ~select_in[0];
    assign select_out[3] = ~select_in[2] & &select_in[1:0];
    assign select_out[4] = select_in[2] & ~|select_in[1:0];
    assign select_out[5] = select_in[2] & ~select_in[1] & select_in[0];
    assign select_out[6] = &select_in[2:1] & ~select_in[0];
    assign select_out[7] = &select_in[2:0];

    always @(posedge load)
        digital_out <= digital_in;

endmodule

module ADCState(input    en, done,
                output reg convert);

    always @(posedge en, posedge done)
        if (done)
            convert <= 0;
        else
            convert <= 1;

endmodule

```

Appendix D

Additional Supporting Material

D.1 Look-Up-Table Test Results

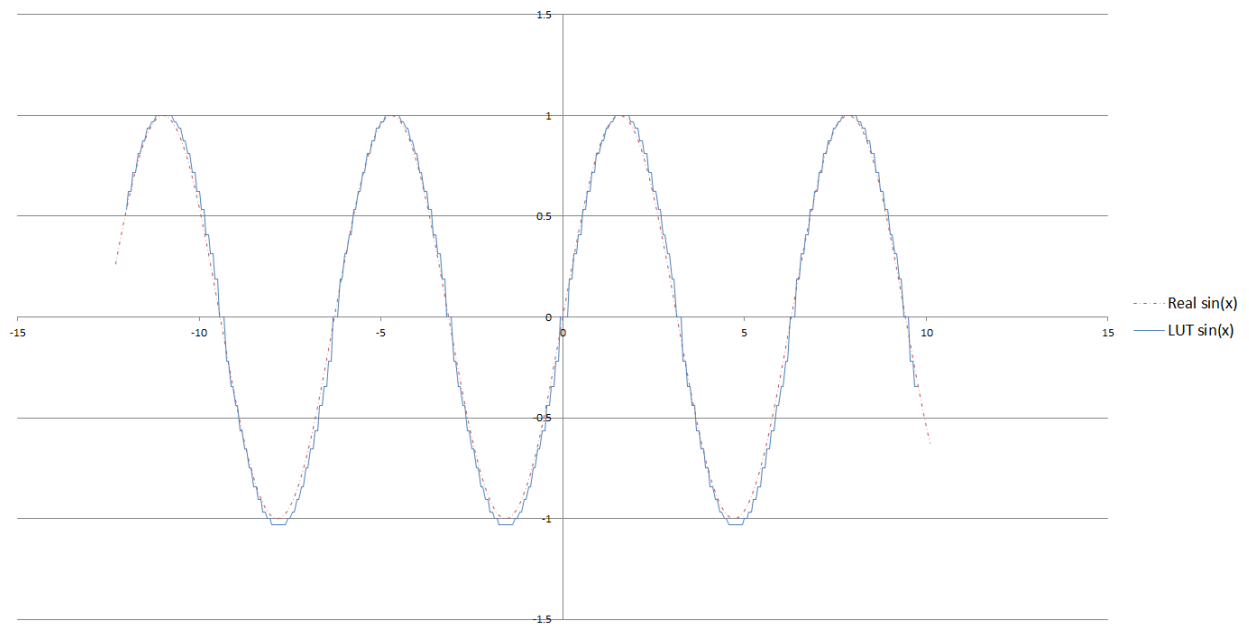


Figure D.1: Look-Up-Table outputs of a sine function

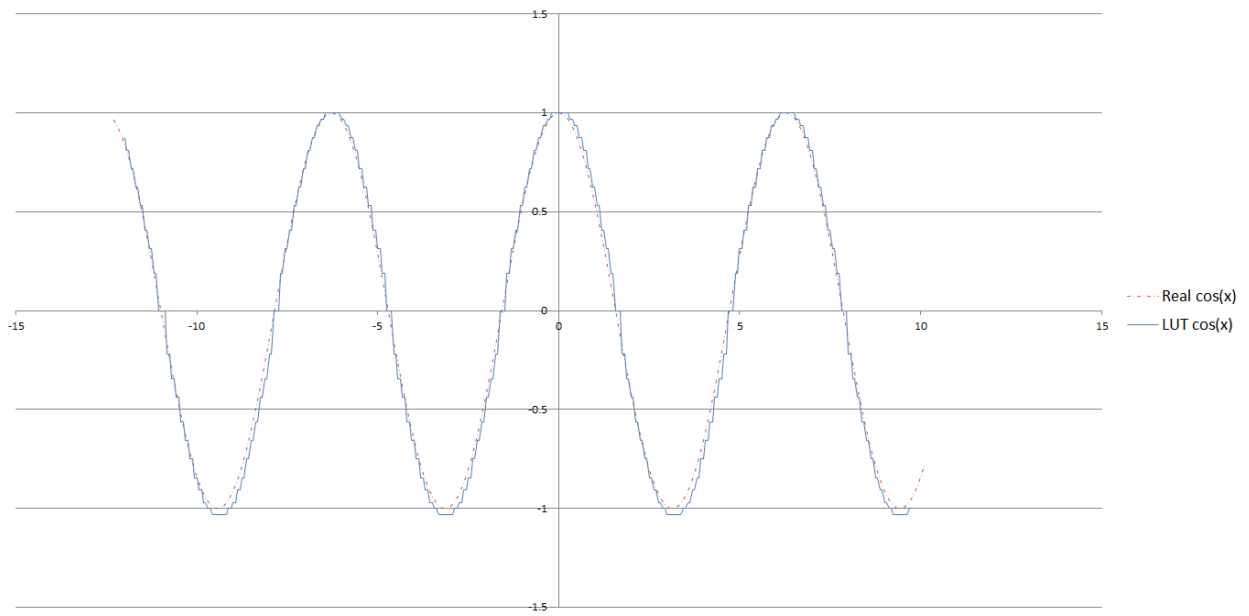


Figure D.2: Look-Up-Table outputs of a cosine function

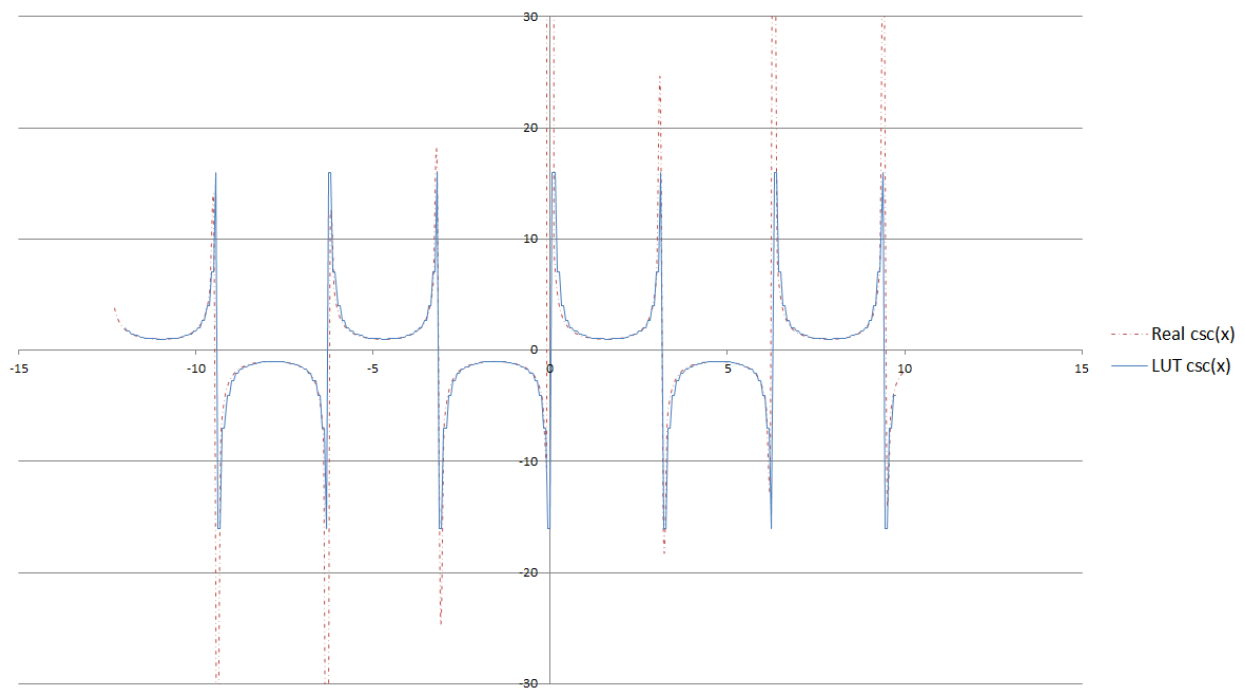


Figure D.3: Look-Up-Table outputs of a cosecant function

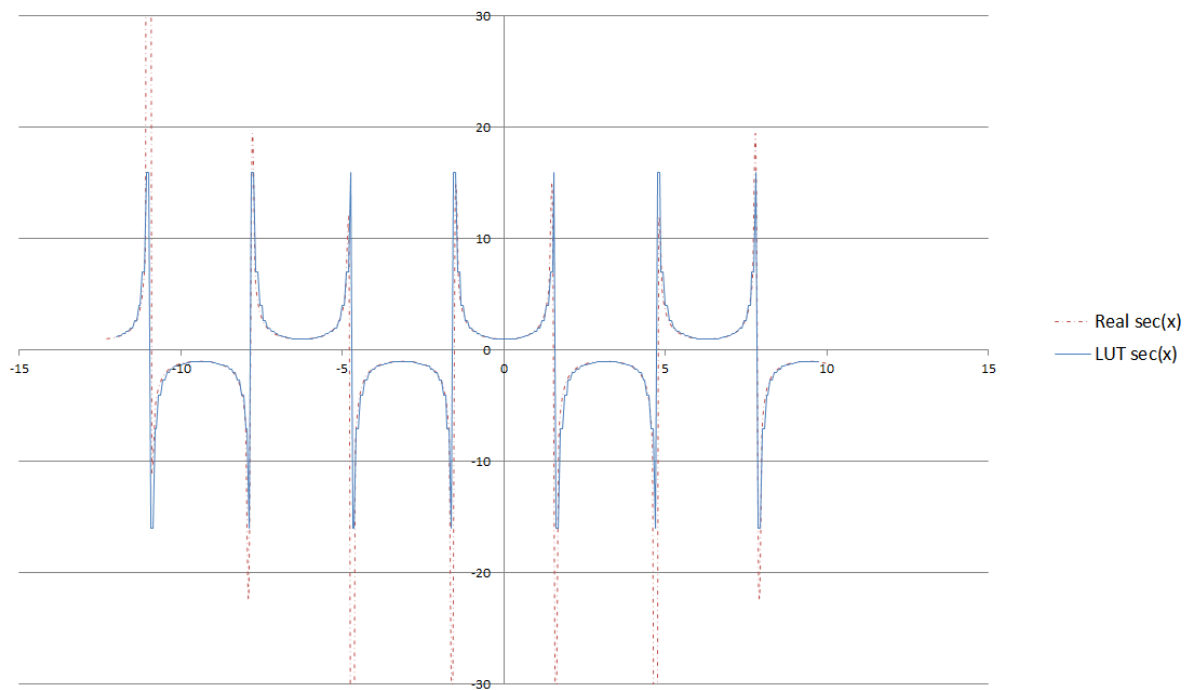


Figure D.4: Look-Up-Table outputs of a secant function

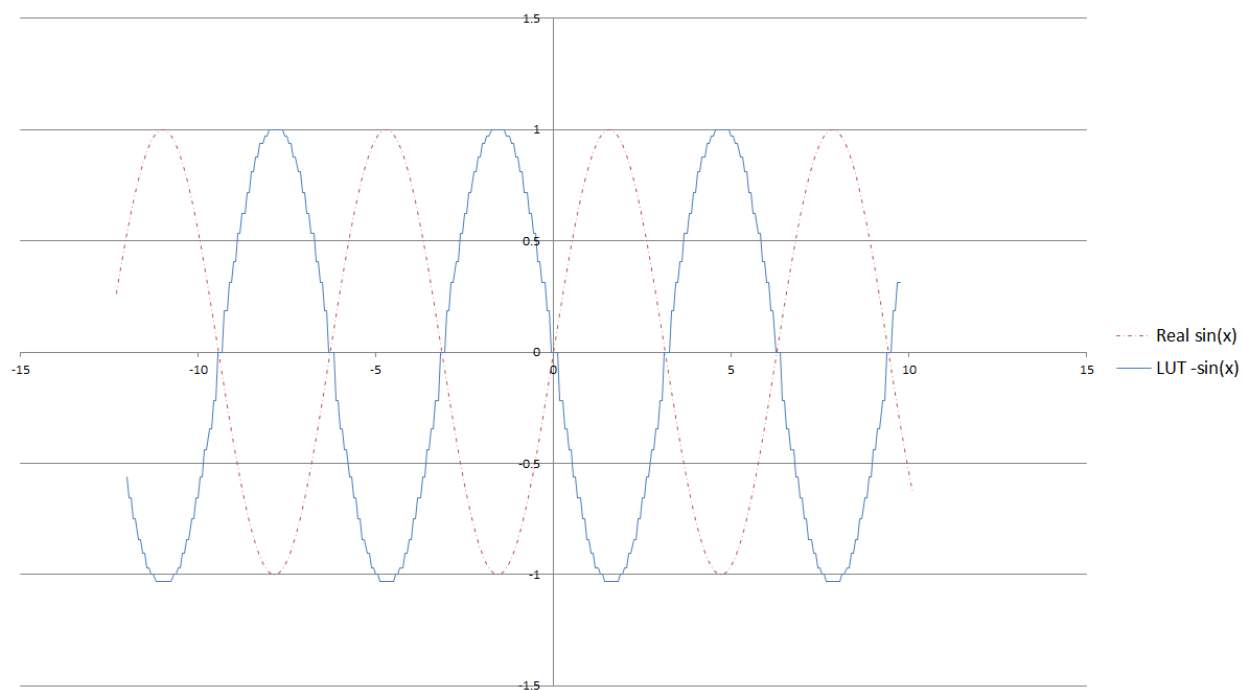


Figure D.5: Look-Up-Table outputs of a negative sine function

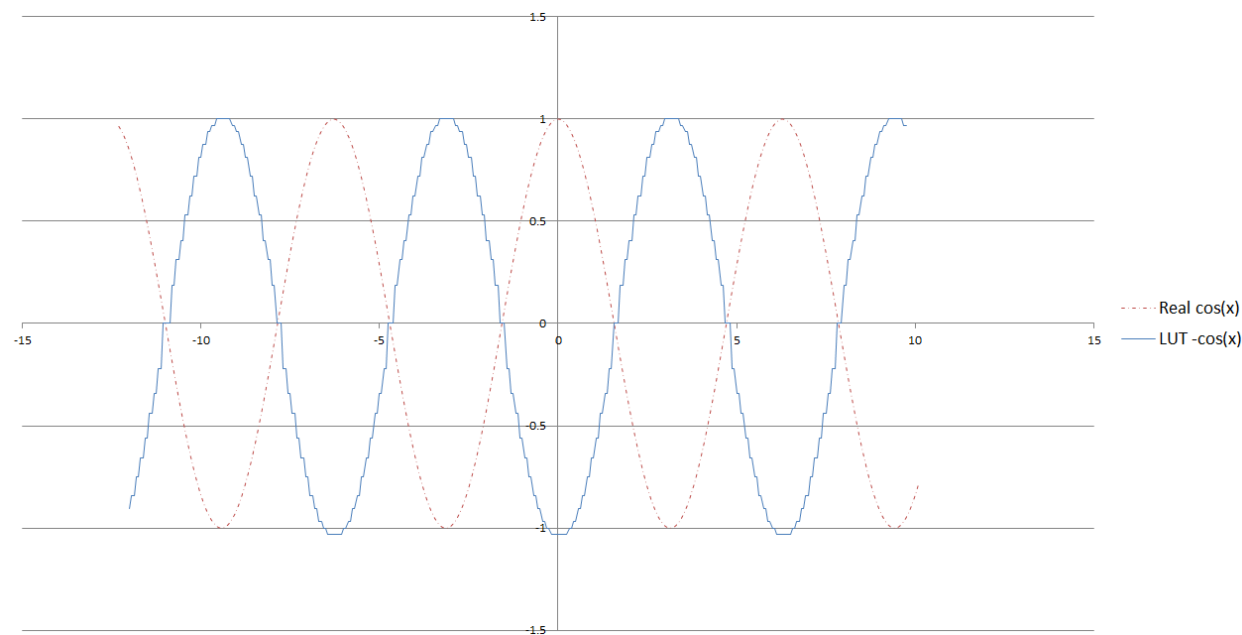


Figure D.6: Look-Up-Table outputs of a negative cosine function

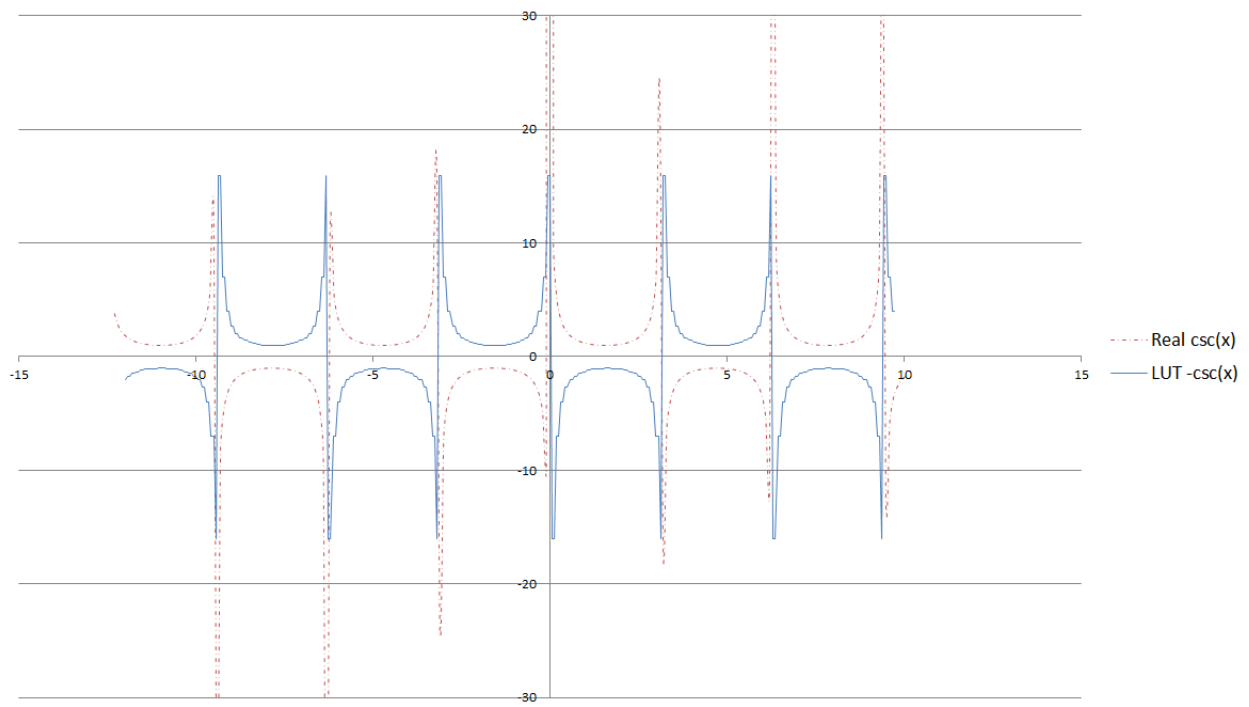


Figure D.7: Look-Up-Table outputs of a negative cosecant function

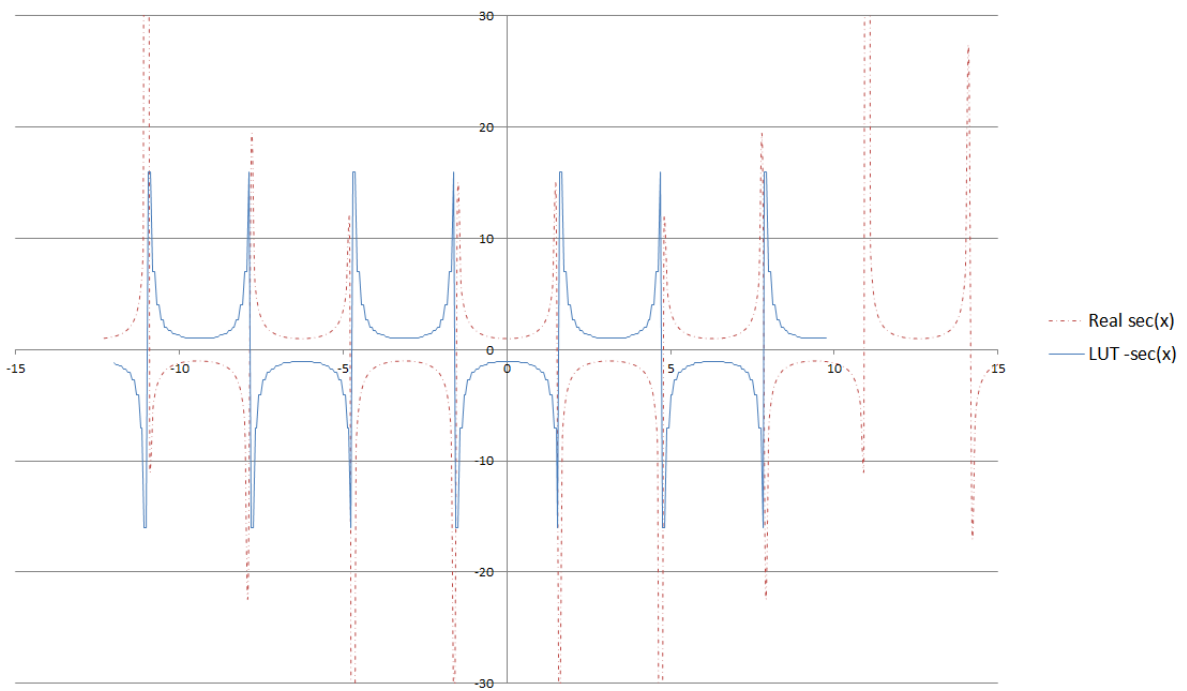


Figure D.8: Look-Up-Table outputs of a negative secant function

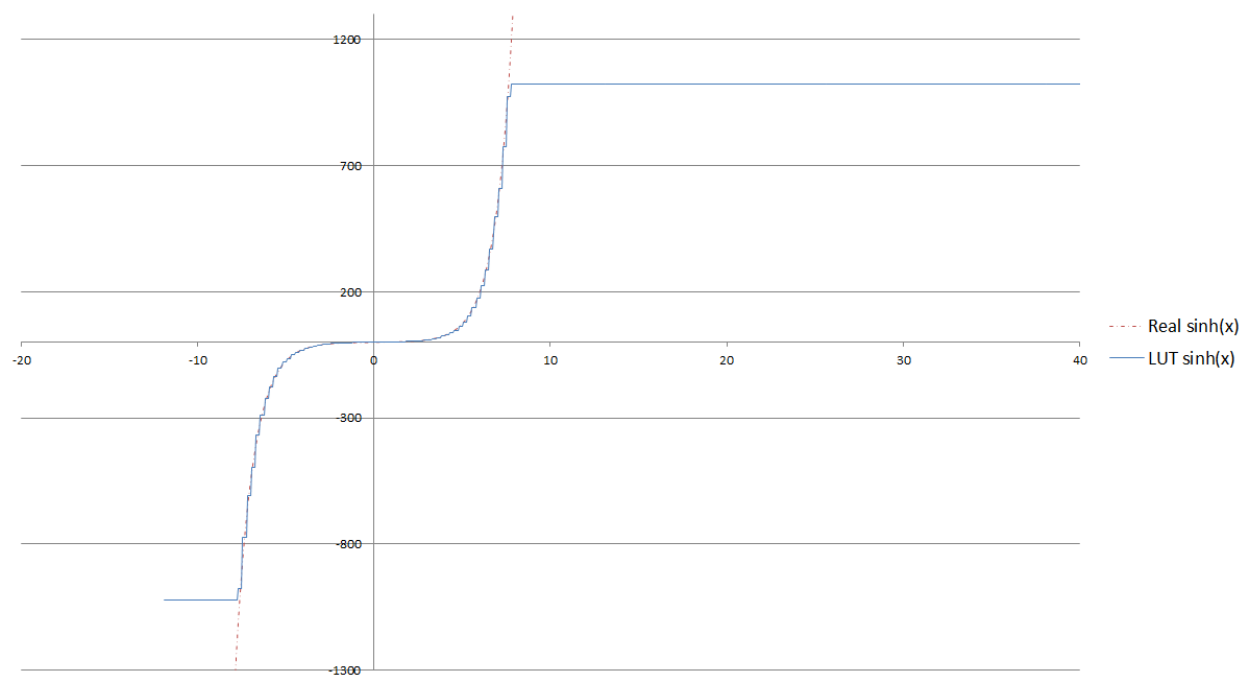


Figure D.9: Look-Up-Table outputs of a hyperbolic-sine function

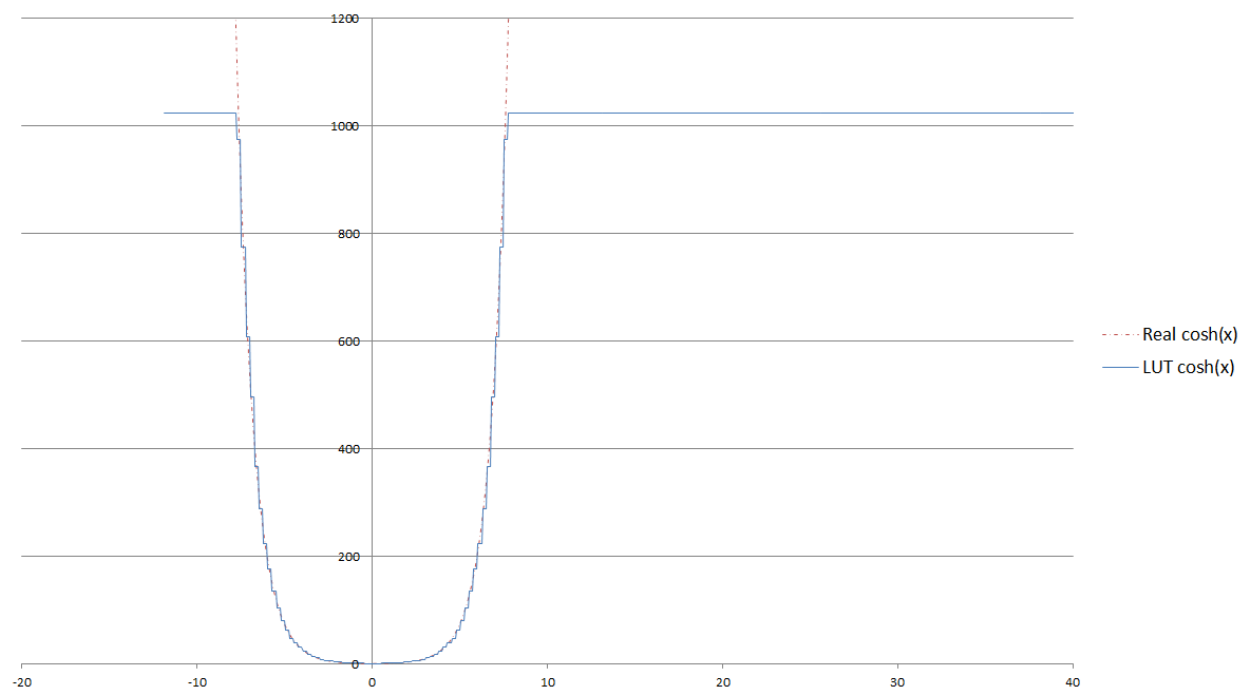


Figure D.10: Look-Up-Table outputs of a hyperbolic-cosine function

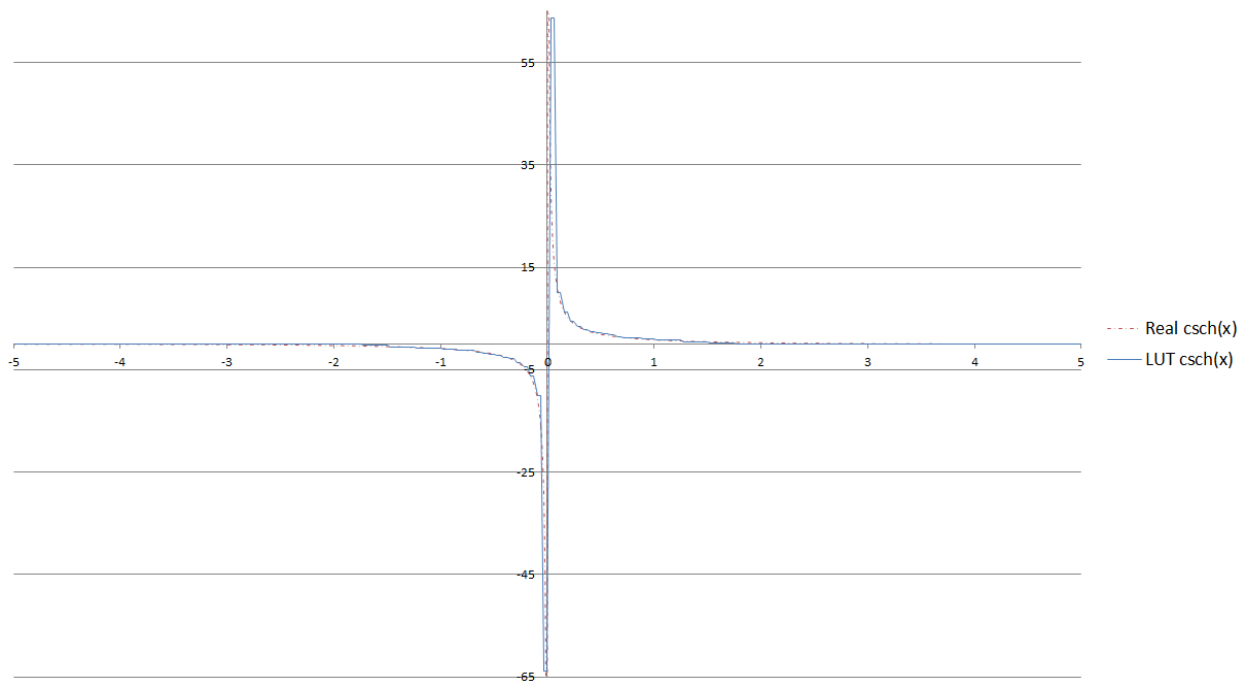


Figure D.11: Look-Up-Table outputs of a hyperbolic-cosecant function

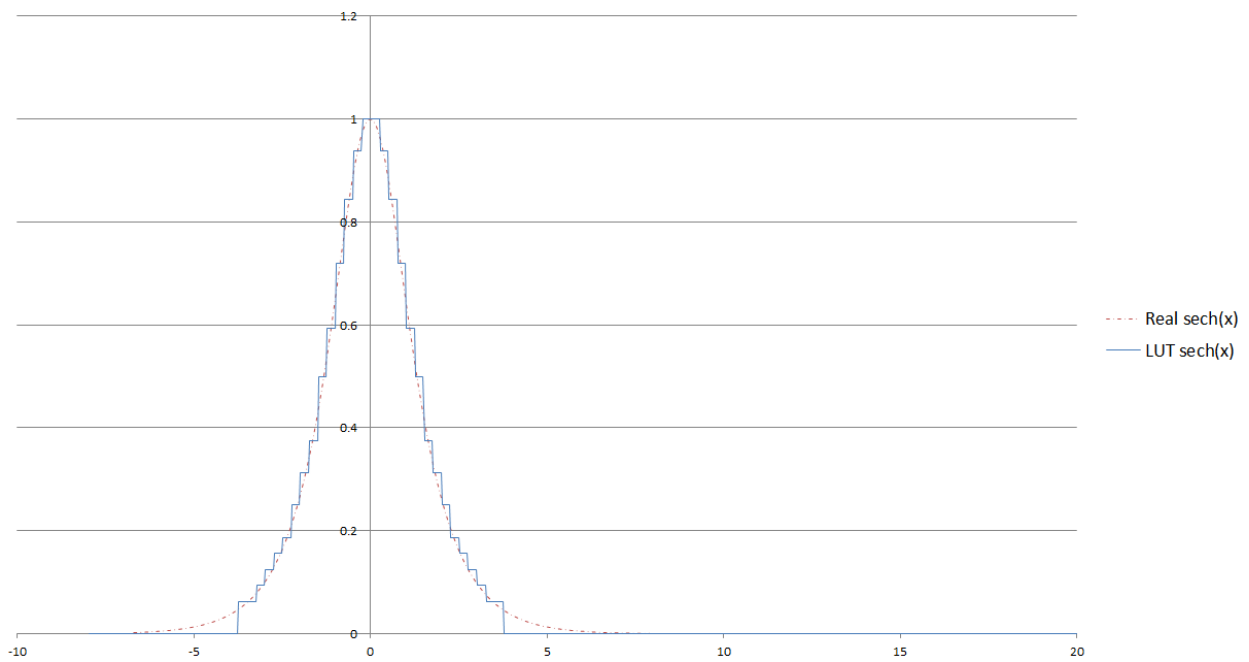


Figure D.12: Look-Up-Table outputs of a hyperbolic-secant function

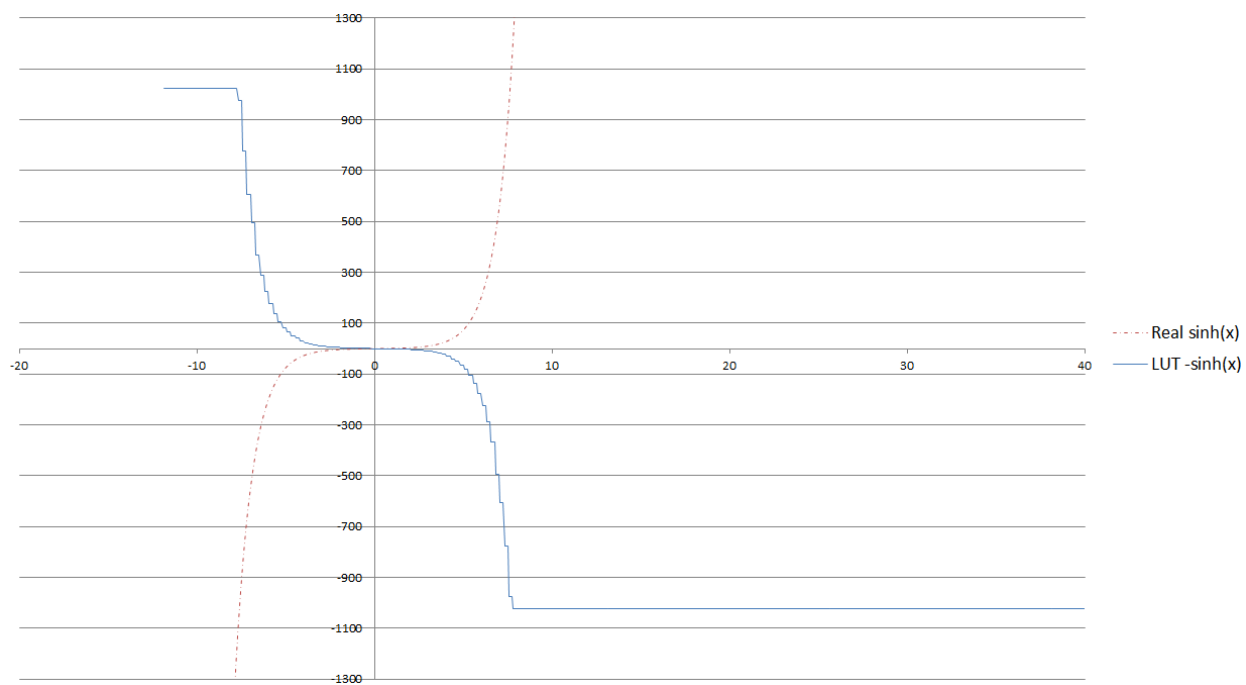


Figure D.13: Look-Up-Table outputs of a negative hyperbolic-sine function

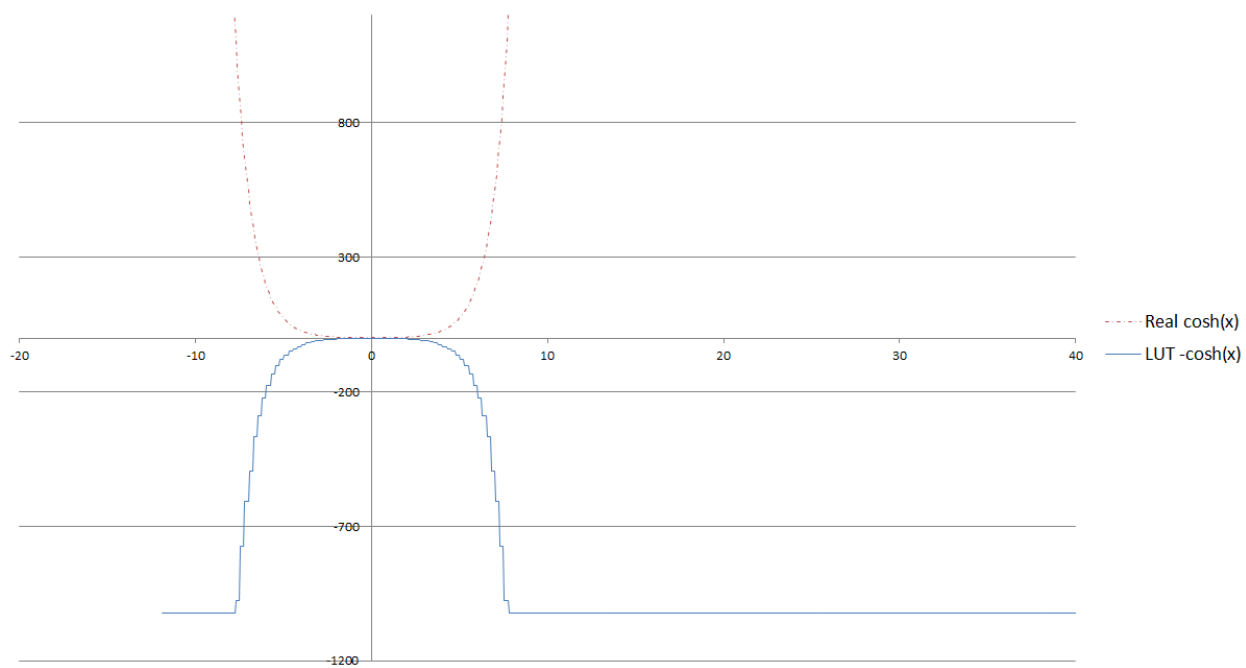


Figure D.14: Look-Up-Table outputs of a negative hyperbolic-cosine function

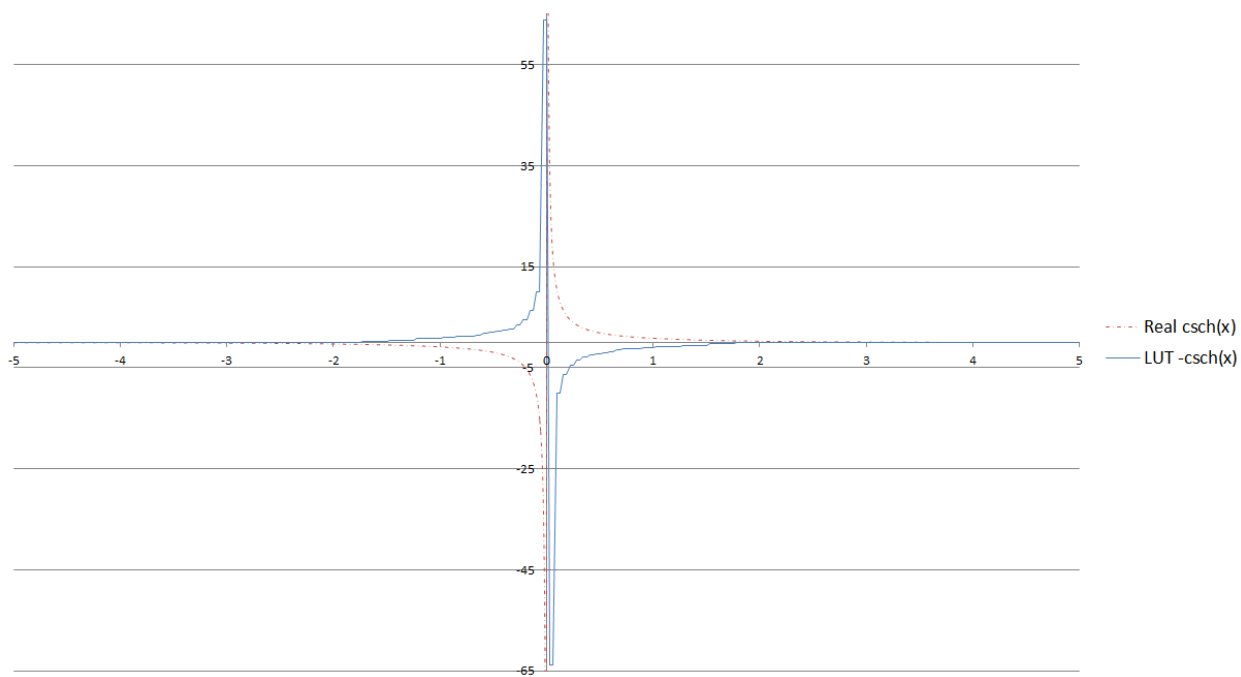


Figure D.15: Look-Up-Table outputs of a negative hyperbolic-cosecant function

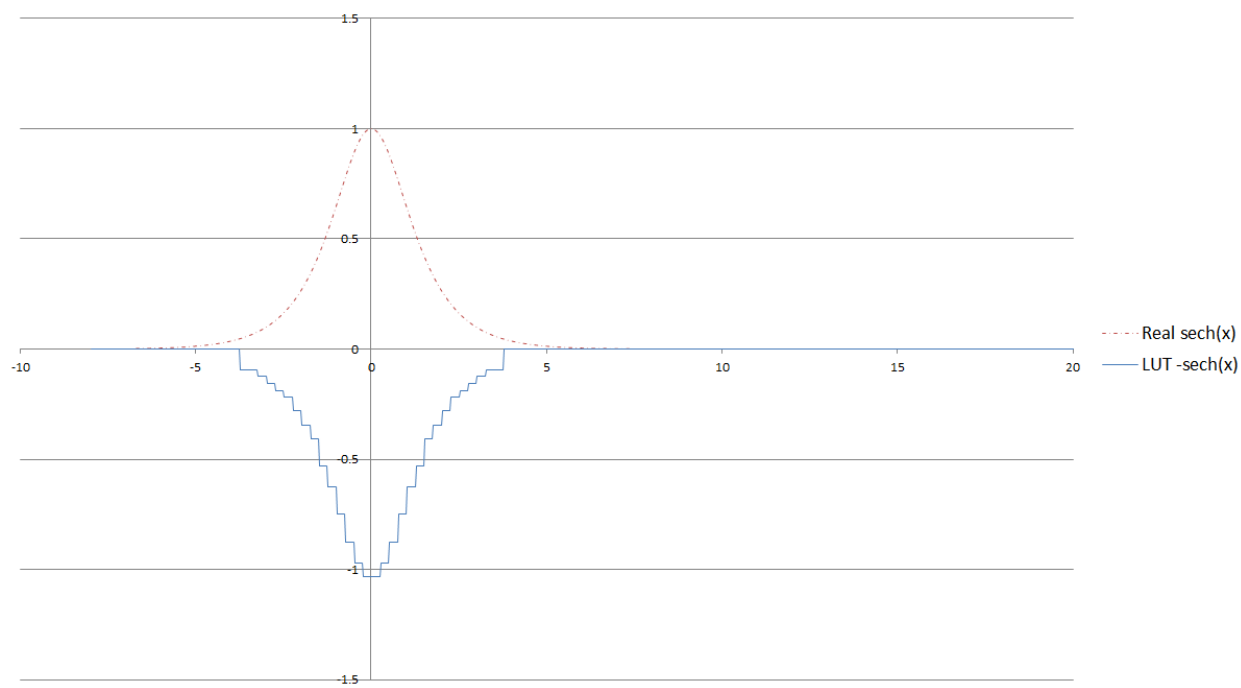


Figure D.16: Look-Up-Table outputs of a negative hyperbolic-secant function