BIG RED CHIP

A Design Project Report
Presented to the Engineering Division of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering (Electrical)

by
Daniel Davis, Saugata Ghose, Mitchell Kotler, Mehak Mahajan,
James Maxwell, Harrison McCreary, Kunal Parmar & John Sicilia
Project Advisers: Sunil Bhave & Rajit Manohar
May, 2008

Abstract

Master of Electrical Engineering Program
Cornell University
Design Project Report

Project Title: Big Red Chip
Author: Daniel Davis, Saugata Ghose, Mitchell Kotler, Mehak Mahajan, James Maxwell, Harrison McCreary, Kunal Parmar & John Sicilia
Abstract:

"Make no little plans; they have no magic to stir men's blood and probably will themselves not be realized. Make big plans; aim high in hope and work, remembering that a noble, logical diagram once recorded will not die."
— Daniel Burnham

The goal of the Big Red Chip (BRC) group, a new project team for Cornell students, is to design a microprocessor starting with a high-level architectural specification and ending with post-Silicon testing and debugging. In addition, the microprocessor can be tailored to provide Cornell research groups with a low-power solution for system-on-a-chip (SoC) designs. Our group has designed the first revision of the BRC microprocessor as an efficient software radio for baseband GPS computations which will be used to develop a full system around the MEMS RF GPS front-end designed by students in the Bhave research group.

The BRC group has designed a 16-bit architecture targeted for power constrained embedded applications. The architecture is implemented with a three stage pipeline capable of running at approximately 50 MHz and 35 mW of power, in under an area of $4mm^2$, and with no DRC, IR, or hold time issues. In addition, the design includes several special instructions to optimize the efficiency of GPS signal acquisition and tracking. The group has produced a full layout for the design which is currently in queue to be fabricated using the IBM 0.13um 8RF process. The design will eventually allow for the integration of a full GPS system-on-a-chip, using the custom software algorithms to enable GPS tracking. The BRC group also completed a full port of the GCC compiler, including the assembler, linker, and all necessary libraries. The team also developed several other custom scripts to test verification, including a random case generator, a software simulator for our architecture, and an automated Verilog testing and coverage tracking system.

Report Approved by
Project Adviser: _____ Date: May 9, 2008

Project Adviser: _____ Date: May 9, 2008

Contents

1 Executive Reports

"Early in life I had to choose between honest arrogance and hypocritical humility. I chose honest arrogance and have seen no occasion to change."
— Frank Lloyd Wright

Contributing authors:

D. Davis Abstract, Design Decisions for GPS (3.4), ADC (4.2.11), Gold Code (4.2.10), Software GPS on the Big Red Chip (5), GPS Results (6.1), In-line ASM for Correlations (G)

S. Ghose Register Decode (4.2.2), Clock State Machine (4.2.9), JTAG portion of DFT Range of Solutions (3.5)

M. Kotler ISA (3.1, 4.1), Pipeline (3.2, 4.2), ALU (4.2.3), Break (4.2.6), Bypass (4.2.7), Memory Controller (4.2.8), Simulator (4.5.2), Random Test Code Generator (4.5.3), Timing Fixes (6.2), and Appendix Sections for the ISA (A), GPS ISA (B), Boot Sequence (C), Debug Sequence (D), Scan Chain (E)

M. Mahajan Toolchain (3.6, 4.7), GNU Binutils (4.7.1), GCC (4.7.2), Newlib (4.7.3)

J. Maxwell Abstract, General Purpose ISA for Use at Cornell(2.1), Write Back(4.2.5), Memory System(4.3), Tools(4.6), Synthesis Results(6.3), DesignVision User Guide(8), Encounter User Guide(9)

H. McCreary DFT Range of Solutions(3.5), DFT Design Choice(4.4), Verification Methodology(4.5.1), Future Work(7), Coverage Document(F)

K. Parmar Toolchain (3.6, 4.7), GNU Binutils (4.7.1), GCC (4.7.2), Newlib (4.7.3)

J. Sicilia Memory System Range of Solutions(3.3), Instruction Fetch(4.2.1), Memory(4.2.4), Future Work(7)

1.1 Daniel Davis

I started work towards my Master of Engineering degree in Spring of 2007 working for Professor Bhave with the goal of creating an SoC solution incorporating current research in MEMS GPS radio-frequency front-end receivers. During that semester I researched a number of options but a clear challenge emerged: although there are a number of available open source solutions, verification of these cores is extremely difficult. This problem is magnified if the final design requires that any changes be made to the architecture. The Big Red Chip project became the optimal solution to these problems.

My primary goal in working on the Big Red Chip project team has been to ensure that the core will meet the minimum required specifications to eventually function as the GPS baseband processor to be integrated with the RF front-end from the Bhave research group. Because we could only expect frequencies of approximately 100MHz from the first iteration of the BRC architecture and because our primary goal was low power I also designed and implemented several special functional units designed to optimize the efficiency of GPS baseband computations.

The intended embedded GPS application ultimately defined the required specifications for the microprocessor and through my role in defining these specifications I became highly involved many of the architectural decisions. In addition, I performed system-level verification of our GPS algorithm using MatLab programs which simulated the structure of the final embedded code as well as the functionality of the GPS-specific functional units. With the results of these simulations and the architecture-level verification of the resulting hardware the result of my efforts has been that the BRC team has confidence that our system will be able to meet the requirements necessitated by the final purpose as a GPS software receiver system.

1.2 Saugata Ghose

My role on this project was as member of the architecture design team. I was involved in making a number of design decisions for the processor design, and helped to develop the initial pipeline structure.

I have written the Register-Transfer Logic (RTL) for the Register Decode (RD) block of the pipeline, and was the chief architect of the clocking system for the processor, which I also wrote the RTL for. As part of this work, I had to determine what functional units belonged in the RD stage, and then implement a hierarchical design in VHDL to tie these functional units into a single block. For the clock state machine (CSM), I was responsible for determining the various clocks that would need to be input to the system, and had to design a system that would allow for seamless transitions between these clock domains. I also contributed to work done on debugging interface for the FPGA.

As part of my initial research contributions to the project, I was responsible for trying to determine the various pipelining options available to the project, and for suggesting and providing an adaptation of the three-stage pipeline with dual-phase clocks for the project (see Section 4.2). I also extensively researched the JTAG interface standard for use with the project. As discussed in Section 3.5, it was determined that the interface was too complex for use with our project.

I was also one of the main proponents for finding and suggesting architectural changes and modifications in order to provide first-order reductions in the power dissipation of the processor. Some of my work in this area includes the successful proposal of a major change to the ISA, which allowed for a significant reduction in the decoding logic required, and the use of a dual-phase clock to allow our project to effectively double the throughput of the pipeline without requiring a corresponding scaling of power consumption.

1.3 Mitchell Kotler

I was the design lead for the project. This involved holding design meetings in order to discuss architectural changes, assigning people to write RTL, and making sure people were making progress on their blocks.

I was mainly responsible for designing the instruction set architecture. It was based on the MIPS ISA and modified for our requirements, which was mainly low power (see Section 3.1). Broad decisions were made as a team, but I wrote out the specifics of the ISA, including full documentation of the ISA, included in Appendix A. I also helped in deciding the microarchitecture of the processor, including the pipeline and memory architecture.

I wrote the RTL for the ALU stage of the processor (see Section 4.2.3). This included the RTL for the ALU itself, along with decoding logic to set the control signals for the ALU. This also includes adding custom instructions to the ALU in order to get I/O from other on chip subsystems in the overall GPS system, such as the gold code generator and an ADC. I wrote the RTL for common modules for the CPU, such as a scan register. I wrote the RTL for break logic, used for stopping and resuming execution while debugging (see Section 4.2.6). I wrote the top level RTL, which instantiates the other modules and wires them together. This includes inserting pipelining registers between modules as appropriate based on the team's decided pipeline architecture. I also wrote bypass logic to forward data in the pipeline. I also wrote the RTL for the memory controller which is used to read or write to memory from the off chip interface (see Section 4.2.8).

I wrote a software simulator for the architecture (see Section 4.5.2). It loads binary executables and simulates them in order to have a working copy of the architecture to test our RTL. It is programmed in Python. I also wrote a very simple assembler to be used in conjunction with the simulator while our software team is working on finishing the full compiler. The assembler simply replaces mnemonic assembly code with binary machine code and calculates labels. I also wrote a program which generates random test code in assembly, which can be used in conjunction with the other software I wrote to perform random testing of our RTL, along with a bash script to automatically generate, run, compare and log any errors of these random test cases (see Section 4.5.3).

I also helped with writing test benches, synthesizing RTL, fixing timing violations and debugging the system. I worked on debugging the top level RTL by comparing random test cases between our RTL and software simulations. I also thoroughly tested the scan chain and DFT features of the chip. Through my debugging I recorded the correct boot up sequence and debugging sequence (see Appendix C and D).

1.4 Mehak Mahajan

I was a part of the tools team, which was essentially the Software team. This team was responsible for the porting of the toolchain for the architecture. At the inception of the project, I was responsible for researching the various memory subsystems, particularly the instruction cache, in various low power architectures.

According to the first distribution of the software team responsibilities, I was responsible for porting the debugger. I decided on porting GDB because its the most popular and widely used open source debugger. Being new to the toolchains, I spent the first couple of months studying GDB from a porting perspective. However, owing to changes in the team structure, I was allocated the task of porting binutils and the port of GDB was postponed until later.

I studied the GNU Binutils which is a collection of programming tools for manipulation of object code in various object file formats. I started off with porting of the Binary File Descriptor (bfd). I also helped port the opcodes. This involved using CGEN, the Cpu Tools Generator. I wrote the architecture description in the CGEN description language. Subsequently, I helped in porting gas and the linker. At this point, I took complete onus of binutils. I wrote out the testcases for the purpose of testing each instruction of the ISA. I also tested the testsuite on the simulator to ensure correct functionality. I was supporting the other members in the porting the other tools by actively fixing the bugs and making changes to binutils as and when required.

1.5 James Maxwell

I served as the nominal project lead for the Big Red Chip project team. My primary technical responsibilities included writing Verilog RTL, Verilog test benches, taking the chip through the synthesis flow, and writing a user's guide for the synthesis flow. Managerial responsibilities included team recruitment, coordinating the team leads, deadline scheduling, and funding requests from AMD.

The Verilog that I personally owned included the write back stage RTL (see Section 4.2.5), and the test benches for the ADC unit and the memory subsection (see Sections 4.2.4, 4.2.11, 4.5). As the test benches uncovered bugs, unintended results, and occasionally simply broken code, I also worked closely with the owners, helping them fix the Verilog until we achieved correct functionality. In addition to my duties I worked closely with the team to develop the ISA.

I served as one of the primary workers with the synthesis tools, including both Synopsys DesignVision and Cadence Encounter (see Section 4.6). Fabricating the chip involved two distinct stages: converting the high level RTL to a gate level equivalent, and then creating a layout from this gate level Verilog. DesignVision, the tool for gate level synthesis, often required rewriting the RTL to fix critical timing paths, and I worked closely with the team so that they understood these timing paths and that we could fix them. Occasionally other RTL changes were required for synthesis outside of critical timing paths, so I worked with the team to resolve these issues as well. Place and route with this gate level Verilog was done using the Encounter toolkit, and I was responsible for many of the design choices in the backend of the flow, including the layout, power grids, IO Pad implementation and DRC fixes. This work included both automation from Encounter and my personal hand layout.

In addition to the technical segments discussing various parts of the architecture, my large contribution to the document are the DesignVision and Encounter User Guides, in Sections 8 and 9. Since the project team will hopefully be continued, with future revisions of the chip to incrementally improve upon our design, future team members, and other researchers at Cornell using these tools, can use these guides as a supplementary reference.

1.6 Harrison McCreary

I was the verification lead for the project. This involved holding verification team meetings to discuss issues that arose during the verification effort, assigning people to interface with the designers of the blocks of Verilog they were verifying, and making an effort to keep verification on time. Furthermore, Mitch the design lead and I came up with a standardized test bench for testing block level and CPU level blocks.

I was responsible for researching different Design For Testing techniques (see Section 3.5) including how to incorporate the scan chain within our design. I looked into the BIST option before deeming it too complicated, while Saugata researched JTAG, before we came to the conclusion as a team to include the scan chain with boundary control logic for externally injected tests.

At the block level of verification, I was responsible for testing the RD block, the Gold Code block, and the WB block. This involved writing test benches for each module that would test each possible input and matching that to a pre-computed correct output. At the CPU level of verification, I was responsible for testing the "simple core" to ensure that the blocks functioned correctly wired together. At this point, I was still using directed test cases and generating test cases manually.

When the full core, complete with pipelining and the GPS modules, was completed, I used the software simulator (see Section 4.5.2) and the random test case generator (see Section 4.5.3) to debug the top level RTL by comparing the output from random test cases run on the RTL with the results from the software simulator.

I was also responsible for introducing coverage metrics for our design (see Section F). This involved looking through VCS's coverage manual and compiling the relevant information for our project into a document for the rest of the team. Furthermore, to automate the process, I revised some of our existing bash scripts to run with coverage. These revised scripts allow the user to kick off a random test case sequence on the processor which automatically compiles coverage reports, as well as the ability to merge coverage from different runs.

1.7 Kunal Parmar

I was the lead for the tools team for the project. My responsibilities included holding team meetings to discuss work, assigning tasks to team members, co-ordinating with other teams, helping team members with their technical difficulties, releasing toolchains and solving problems associated with the same. Our original goal was to port a compiler, assembler and a debugger for our processor. We completed porting GCC, GNU Binutils and Newlib. We postponed the porting of the debugger (GDB) until we have some underlying support for debugging.

I started off by researching which compiler and assembler to port (see Section 3.6) for our processor. I also helped with the Instruction Set Architecture (ISA) design. Along with John Sicilia, I researched the various ISAs to help decide our ISA. I suggested the addition of the `break` instruction and a single stepping mode for debugging purposes.

I then started work on the development of our toolchain. I started by porting GNU Binutils (see Section 4.7.1) along with my team member Mehak Mahajan. After we had a basic assembler in place, I began work on GCC (see Section 4.7.2). After completing the port for GNU Binutils and GCC, I ported Newlib and Libgloss (see Section 4.7.3) for our processor.

1.8 John Sicilia

I was on the design team for this project, though towards the end of year, my work was more focused on the verification of the processor. I was involved in researching different types of instruction set architectures and with creating the original ISA that we decided on. When we began to turn our high-level designs into Verilog code, my focus was on the sections of the pipeline involved memory. I wrote the original RTL and was in charge of the design of the IF and MEM modules.

I was also in charge of the memory system in the processor. I learned how to use the memory generator and explored the different options available to us, then consulted with Dan to decide on an optimum size for each of the memories. Since the memories were only word-accessible, I researched alternatives in order to enable byte-level memory loads and stores. Finally, I revised the MEM and IF stages to reflect these changes.

At the block level, I was in charge of testing the write back, break, bypass, and memory controller modules. This involved writing Verilog test benches for each of these modules, testing the common case thoroughly, and coming up with as many corner cases as was possible. Toward the end of the semester, as new functionality was added to the GPS blocks later than expected, I took charge of verifying this new functionality in the ADC and gold code modules. I edited the original test benches and also wrote new ones. At the CPU level of verification, I wrote directed assembly tests of the GPS modules, most notably the ADC. I wrote many assembly tests and worked with Dan in order to fix bugs I found in the code.

2 Design Problem and System of Requirements

"Computers are useless. They can only give you answers."
— Pablo Picasso

2.1 General Purpose ISA for Use at Cornell

With the increasing computational power of microprocessors and a decreasing area and power cost, the desire to use these processors in embedded design is increasing due to their high flexibility and programmability. Tasks that may have been too complicated for FPGAs or even microcontrollers can be accomplished using a processor. One of the major goals of the Big Red Chip project was to be modular, yet simple, enough to enable any research group to integrate the chip into their systems effortlessly. The BRC group has designed a fully custom architecture ISA for 16-bit embedded processors, and when customizing the ISA have chosen to implement five user definable instructions. All the functionality of a MIPS architecture has been preserved with both registers and memory, but in addition also includes several unique functional blocks including an on chip ADC unit. With an open architecture, BRC seeks to allow easy implementation of additional custom units, suited to whatever a particular research team might need. Any interested research group will not only have Verilog of such an ISA, but with this document have all the references necessary to recreate the synthesis of such a design if they choose.

This architecture is currently in queue for fabrication in the MOSIS 130nm process technology. The tools necessary to take the high level RTL Verilog and produce a layout have been documented in Sections 4.6, and a tutorial for new users can be found in Sections 8 and 9 for Synopsys DesignVision and Cadence Encounter, respectively. This chip is predicted to run up to 50MHz at 35mW, which will be sufficient for most low power processing needs at Cornell. In addition, the BRC team has developed a custom compiler to convert user C code into assembly code compatible with the ISA, as described in Section 4.7. Other custom scripts for system verification, as described in Section 4.5.2.

2.2 Customized Architecture for Low Power GPS

The first iteration of the BRC architecture has embraced the BRC project goal of enabling Cornell research by targeting embedded software GPS applications. Cornell has been heavily involved in the GPS system architecture almost from its conception but has recently become very involved in researching how software algorithms can be employed to perform baseband calculations in flexible platforms. Although these software systems have already realized commercial viability, the typical implementation requires a number of discrete integrated circuits and consumes more power than an ASIC-based approach.

Recently, students in the Bhave research group have begun to explore how MEMS technology can provide a low power solution for the radio-frequency front-end of a GPS receiver. The specific goal of the first iteration of the BRC architecture is to provide a baseband processing system that will expand the MEMS receiver from the Bhave group into a fully integrated low-power GPS system. The BRC system for performing GPS baseband computations and computing pseudo-ranges is based on bit-wise software algorithms developed by Cornell's software GPS research groups. In addition, the BRC system uses its reserved operation code space to employ several special functional units designed to increase the efficiency of calculations and reduce the required power for the system.

3 Range of Solutions

"Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius—and a lot of courage—to move in the opposite direction."
— E. F. Schumacher

3.1 ISA

One of the first decisions we had to make was what instruction set architecture, or ISA, to use for our chip. As our main design constraint was power, as described in Section 2.2, we decided on a 16-bit ISA. Having a smaller instruction and data size would reduce dynamic power by having fewer bits to transmit across the chip, having less information to decode, and having smaller data on which to perform operations. The other quality we wanted in our ISA was simplicity and regularity in order to increase the power efficiency of decoding the instructions.

We researched existing 16-bit ISAs including 16-bit versions of MIPS[6], an ISA we are familiar with from class, and ARM[1], an ISA popular for use in embedded processors. These ISAs are both actually hybrids of 16 and 32 bit instructions, with ways of switching between modes. Both also were more extensive than we thought necessary for our purposes. Also, due to our GPS requirements (see Section 2.2) we needed some custom instructions. Due to these reasons we considered designing a custom ISA, by using ideas from the 16 bit ISAs we studied, in order to fit our needs. Designing our own ISA had some disadvantages however, including the increased design time and not having any tools or software available for the ISA.

We decided that the extra design time would be justified, since the existing ISAs did not adequately satisfy our needs. The tools and software we needed would mainly be a compiler and assembler, as we were not planning any operating system support, and the software team would be able to handle that in parallel to our design efforts. The design of our custom ISA is discussed more in Section 4.1.

3.2 Pipeline

One of our main microarchitecture decisions was deciding how to pipeline our processor. Pipelining allows for greater throughput, but as you increase the amount of pipelining you increase the amount of time required to flush it due to hazards. As we are designing for low power, (see Section 2.2) we decided to research the pipeline architecture of other low power chips in order to determine what would be optimal for our use.

One design we considered was the Razor[4] architecture. This is a six stage pipeline design which uses circuit level timing speculation and dynamic voltage scaling in order to be low power. It used the five pipeline stages from the traditional MIPS pipeline, with an extra stage at the end for stabilizing the data. This allows it to run very fast, with implementations running in the gigahertz range. However, as we wanted to keep our design basic, the custom circuit techniques used in this design seemed beyond the scope of our project. We also were looking to use less pipeline stages in order to avoid branch delay slots or the need for a branch predictor, as these would add more complexity than we wanted.

The next architecture we looked at was the CoolRISC[7] architecture, a popular architecture among low power designs. It uses a three stage pipeline corresponding to a fetch/decode stage, an execution/memory stage, and a write back stage. This one did not run as fast as Razor, but by using both edges of the clock could be run at approximately 100MHz, which was in the order of magnitude of what we thought we would

need for our GPS application (see Section 2.2). A problem with this approach is the inability to halt the processor, but if we can keep the processor busy with useful work then halting becomes unimportant.

The last architecture we considered was the Smart Dust[9] architecture. This had no pipelining, but used an eight phase clock in order to segment the various stages of execution. This eliminated the need to register data in between stages. This was also a very low power design, as it also was able to halt, which made it a good candidate for systems that do periodic sampling. It only runs one instruction at a time though, and ran slower than the other choices, at about 100kHz.

We decided on a microarchitecture based on CoolRISC. We made some changes to it as described in Section 4.2, but decided that a three stage pipeline was best for our needs. The Razor approach was too complicated at the circuit level for our uses. Since the Smart Dust approach was slow and had complicated clocking, the ability to halt did not seem necessary enough to justify it.

3.3 Memory System

When designing our memory system, we were focused on one goal: to minimize power while still retaining the functionality needed to run correctly. Since we were aware from the beginning that memory was going to consume a large portion of our allotted power consumption, efforts to minimize this as best as possible were made. It was the most important benchmark we had when considering our two most fundamental decisions with regard to the memory system: the type of memory architecture, and what kind of hierarchy to implement.

When looking at the memory system with power-minimization in mind, it seems clear that the best way to accomplish that goal is to be as small and as simple as possible. By moving memory off-chip, more sophisticated circuitry would be needed in the memory controller and single-cycle access times become infeasible. And if dealing with both an on-chip cache and off-chip main memory, the circuitry becomes even more complex as blocks are moved into and out of cache and write-back policies need to be considered. The question was whether or not the benefits gained by these more complex systems (vastly increased memory space for off-chip memory and low memory-access time for the combination) were worth the power drawbacks.

Since our processor was going to be used with an embedded GPS program, we should know both the size of the memory needed to contain the instructions and the average amount of data memory needed to run. In addition, since the program is embedded, we will not have to worry about multiple processes running in parallel, each of which needs to be stored in main memory. For these reasons, the choice was made to implement a fast, small, on-chip cache without a secondary off-chip memory.

The second basic choice we had to make was the type of memory architecture to follow: Harvard or Von-Neumann. While the Von-Neumann model is more flexible than the Harvard architecture, the embedded nature of the Big Red Chip means that flexibility is not high on our list of wants. In addition, the Harvard architecture has two very important features that make it far more desirable: the removal of structural hazards and flexibility of address widths.

First, the Harvard architecture allows us to access the data and instruction memory in parallel, using two different ports. Because we are able to access both at the same time, we are able to decrease the speed at which the memories must be clocked, thereby reducing their power consumption. Secondly, the Harvard architecture allows us to use different widths for the instruction and data memories. While we standardized the widths of the data buses to 16-bits, varying the size of the address bus widths let us use almost exactly

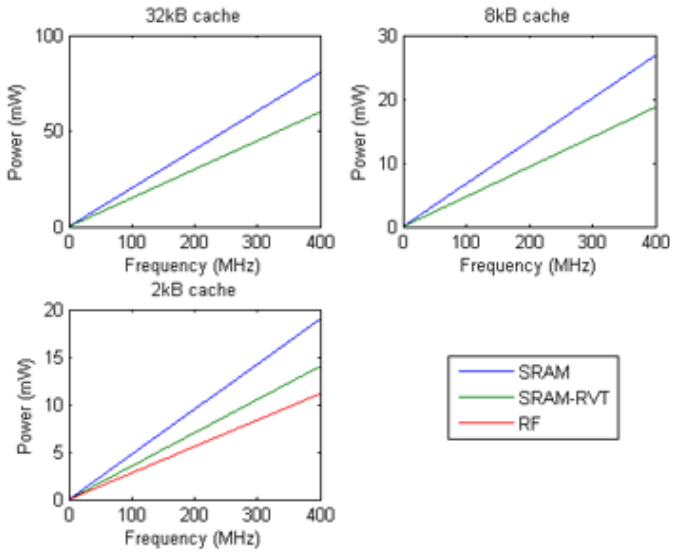

Figure 1: Power Consumption Performance of Different Memory Architectures

the size of instruction and data cache that we needed. By minimizing unused data and instruction cache space, we once again minimized power consumption.

Having chosen on-chip memory and the Harvard architecture, our next step was to create the memory. Streamlining this highly time-consuming task was the Artisan Standard Library SRAM Generator by ARM. This tool allowed us to vary different settings, such as memory size and frequency, and created output files that ranged from a documentary PDF to the Synopsys models. Since our choice of architecture allowed us to change the instruction and data cache address widths independent of each other, we tested different cache sizes at different frequencies. Our goal was to determine the maximum size and operating frequency we could use and still be within our power consumption limits. Figure 1 demonstrates these results.

The three different technologies we were able to use were SRAM, RVT SRAM, and a register file. The register file generator was only able to create small-sized caches (less than or equal to 512B with a 16-bit word, and up to 2kB by increasing the word size), so while it gave the best performance in the tests it could compete in, it was largely infeasible. Surprisingly, the RVT SRAM ended up giving us the best performance at the sizes we needed, despite the fact that these devices are typically consume more power.

The smallest sizes that we could use without compromising the functionality of the program ended up being a 16kB d-cache and a 2kB i-cache, both implemented with RVT SRAM. The estimated average current drawn, assuming 100% reads, would then be 8.871mA for the i-cache and, assuming the memory is constantly on and a 50/50 mix of reads and writes, 17.168mA for the d-cache.

One last concept that needed consideration was whether or not we could include the originally envisioned instructions lb, lbu, and sb. The memory generator only includes addressing down to the word-level, not the byte-level. As a result, we were unsure of whether or not we were going to be able to include byte-addressable memory instructions. After reading the documentation more thoroughly however, we found that the memory generator provided a solution to our problems. If we could byte-mask and shift inside of the memory stage of the processor, then we were able to use a write-enable byte-mask that allowed us to only write a single

WEn[1:0]	Action
11	Read Word
10	Low-Byte Write
01	High-Byte Write
00	Write Word

Table 1: Byte-Addressable Write Enable

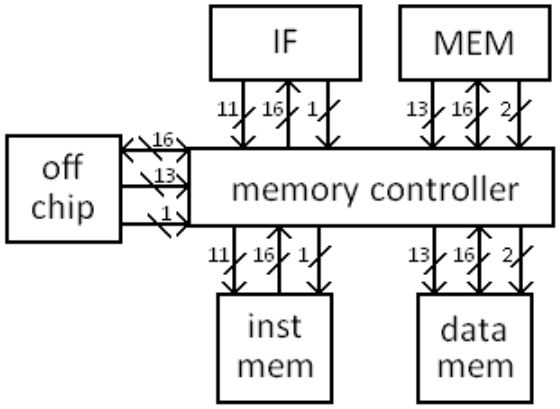

Figure 2: Memory System of BRC

byte of a word at a time. To do this, the write-enable line of our memory was extended to two bits, as shown in Table 1.

While this ability allows us to keep the functionality of byte-level memory accesses, the fact that our memory is still only word addressable introduces a few complications into the way we access our memory. First, the internal memory address lines are one bit larger than what the memory modules will actually accept, with the least-significant bit being the byte selection. The MEM stage module in the processor utilizes this bit in order to calculate the write-enable, if it is a store, and shift the data left or right appropriately. Thus, the actual memory address lines out of MEM and into the controller are one bit smaller than those going in, and the write-enable line is one-bit wider.

With the main memory system in place, the final step was to add a simple memory controller. Since we were dealing solely with an on-chip cache that was consistently accessible within a clock cycle, our memory controller did not have to deal with the problem of a memory hierarchy or stalling if the data was not yet available and a hazard was presented. Instead, it functioned largely as a multiplexer, allowing an off-chip source to directly access i-cache or d-cache instead of IF or MEM. This allows for both initial programming of the instruction cache and setting initial tables in data memory, and for reading out of memories in order to test. A simple diagram illustrating the memory system with the widths of the buses involved is shown in Figure 2.

3.4 GPS

One of the largest challenges in implementing a software receiver is ensuring that all of the input data is successfully received and processed. This problem is magnified for GPS applications because the input data rate is largely determined by the GPS system and because the incoming data must be aligned nearly perfectly with local representations of incoming signal components. Ensuring that the BRC system could meet the data processing constraints required that the team decide how the system should be triggered to receive incoming data early in the design process. We explored two options before deciding on the final data input system: first, a timer-based interrupt to initiate data retrieval; and second, a direct memory access system to allow the input data to write automatically into the data memory.

The most conventional way for a software-based system to perform any timing critical task is through interrupts. Typical interrupt systems include both timer and externally triggered interrupts. Although the most effective way to ensure that no data is lost would be to have some external system produce an interrupt signal, this signal would not be synchronized with the operations taking place in the core. For this reason the BRC team determined that an external interrupt system would be too difficult to fully verify. Timer-based systems, however, use an internal system to produce the interrupt signal and can be synchronized to the clock of the core. Because the input data rate could be computed by the processor a counter conservatively set timer could be sufficient to avoid data loss. Unfortunately, a problem with all interrupt systems is that they require the overhead of a function call whenever they execute. In addition, our final system varies the rate at which data is processed based on the Doppler frequency of the current satellite. Since the system must be able to track multiple satellites with different Doppler frequencies, it would be possible for an interrupt from one satellite to occur during an ISR triggered by another. For this reason the team decided to avoid interrupt-based systems; if an interrupt occurs during another interrupt then the timing cannot be guaranteed for either.

Many digital signal processing (DSP) processors use data input systems with direct access to the core data memory[3]. Although the goal for the BRC architecture was to be more general purpose than a typical DSP, the GPS software receiver application follows a structure similar to the multiply-accumulate architecture of a DSP. For this reason, direct memory access is compatible with the required functionality of the BRC processor. The difficulty in DMA systems arises in structural hazards as the input system and core may try to access the data memory simultaneously. However, because the BRC architecture has several reserved instructions this problem can be solved by creating an entirely new hardware memory structure for each input memory and access those memories through special instructions. The only real disadvantage to a DMA system with partitioned memory that the processor core loses control over the operation of the input memory system. A DMA system also provides the advantage of having almost no instructional overhead on the core to retrieve input data, which is critical for a power-constrained system.

For the reasons detailed above, the BRC team ruled out an interrupt-based system and opted for a modified direct memory access approach. To avoid structural hazards each trackable satellite has an independently defined two-port register memory. This memory has separately clocked read and write ports to eliminate any data hazards and is readable only through special instructions. In addition, special functions are implemented within the system to allow the processor core to gather specific information about the state of each memory to allow the processor to more closely monitor and control the input registers system.

3.5 DFT

To ensure correct behavior in processors after fabrication, most modern microprocessors include hardware that enables the designers to test for correct functionality on the physical chip. This idea of including extra hardware to the design with the specific aim of making it easier to develop tests for post-manufacturing is called Design for Testability (DFT). Our team looked at three options to incorporate DFT in our chip. These included a Built in Self Test (BIST) solution, a JTAG solution, and a scan chain with a traditional boundary-scan interface.

The BIST solution consists of an internal function that is able to verify all or part of the functionality of the chip. BIST is a good option for DFT because it reduces the need for externally generated tests and hardware associated with those tests. Furthermore, it reduces the number of I/O pins needed for external tests to be run on the chip. Finally, BIST allows for diagnostics to be run on chips in the field as the tests reside in the chip. For the scope of our project, however, BIST is too complex for what we will need to test once the chip returns from fabrication. Because our chip's design space is relatively small, the overhead for including internal storage for BIST test patterns is quite large and not justifiable in terms of our space and power requirements. Also, we have the option of using an FPGA as the external hardware needed to test the chip, which is much less expensive than the external hardware needed to test more complex systems in industry. Finally, our design has enough space for the I/O needed for external tests to make the storage of an internal test pattern unnecessary.

The JTAG interface[8] is an IEEE standard used in industry to perform design verification. The interface allows debuggers to control a device and read data stored in internal registers using a boundary scan chain. Our initial goal was to use this interface to be able to read and write register values within the processor. However, upon further research, we saw that the JTAG interface requires a significant amount of added functionality, and would require that we design an internal controller for the interface. As our intent was strictly to debug the registers, and not to affect the other operations of the processor, we decided to drop the JTAG interface. Not only would the extra required functionality have taken up a significant portion of design time for an unnecessary feature, but the extra power consumption and layout area were not justified for the scope of our project.

The final design choice we considered was a scan chain with some boundary-scan interface and custom control circuitry to externally inject test patterns. This option consists of wiring together all architectural registers within the core together into a chain and adding control logic to the registers to allow the scan chain to be serially read out of the core. This chain becomes the scan chain. Tests are generated external to the chip and must be input using some type of hardware; for our purposes an FPGA would be an acceptable solution. This option was the best for our purposes because it allowed us to include only the hardware that we needed for simple register debug, which provided the lowest complexity, size, and power consumption of the three options.

3.6 Toolchain

The adoption of any architecture is directly related to the quality of the compiler available for it. This made the choice of the compiler critical to the success of our project. Given the available time line and the feasibility concerns, writing a compiler from scratch was quickly ruled out. Hence we decided to port one of

the existing compilers to support our new architecture. Large parts of the programs were to be written in C with only a few critical functions written in assembly. Our compiler had to support such inter-leaving of high and low level languages.

Of the many compilers available, we narrowed our choices down to two retargetable compilers - LCC and GCC. After careful study of both the compilers we decided to port GCC. One of the driving factors for the same was that LCC is not under active development, the last release, LCC 4.2, being in June 2003. GCC is currently the most widely used open source compiler. It supports a variety of backends including x86, ARM, Sparc, PowerPC and MIPS - both the 16-bit and 32 bit ISAs, which forms the basis of our architecture. GCC provides higher optimization support which causes it to take longer to compile the code as compared to LCC. However, we needed a good compiler - one that produces highly optimized code, and not a fast compiler. The decision to use GCC was finally sealed owing to the software team's experience in porting GCC earlier. Given the stringent time line, this was the most feasible option as familiarizing the team with a new system would require considerable time and effort.

Along with the compiler, we needed an assembler and libraries to complete the toolchain. GNU Binutils provides the most popular and widely used open source assembler used in conjunction with GCC. GNU Binutils is a collection of binary utilities which includes the assembler, disassembler, linker etc. Newlib, another widely used open source library on embedded systems, was chosen as the library. The choice for the assembler and the library were also influenced by the team's experience with these systems.

4 Design and Implementation

"Perfection is achieved, not when there is nothing left to add, but when there is nothing left to remove."
— Antoine de Saint-Exupery

4.1 ISA

As we decided on a custom ISA for reasons explained in Section 3.1, our first design task was to define our ISA. We started by defining the instruction formats our ISA would have, based on MIPS but modified to be 16 bit for our uses. We divided our operations into four types in order to support instructions with zero through three register operands (see Table 2). The ordering of the registers is arranged so that rs and rt are always read and rd is always written. The extra bits in the three operand format are used as extra opcode bits in order to expand our opcode space, while the remaining bits in all other formats form an immediate operand. The four formats roughly translate into register ALU instructions and memory instructions for the three register format, immediate arithmetic instructions for the two register format, branches for the one register format, and jumps and other special instructions for the zero register format.

The 16-bit instruction size imposed some serious limitations on the ISA. Our opcode was only five bits, only allowing for thirty two instructions. By using left over bits from the three register format as a "function" field, the opcode space is effectively increased, as every three register opcode can encode four separate instructions. This allows us to use the three register format for instructions that did not need all three registers, but also did not need an immediate, such as memory instructions, in order to conserve opcode space. This leaves us with adequate opcode space for all the instructions we wanted, while still leaving room for reserved opcodes. These reserved opcodes were important , as we wanted to keep the BRC ISA general purpose, yet customizable to particular projects. We used some of the reserved opcodes for GPS specific functions in our version of the chip, while future implementations of the chip can customize them to suit their needs.

Another problem created by the small instruction size was limiting the ISA to only three bits per register field, which leaves us with only eight general purpose registers. This is a severe restriction, as most RISC architecture include at least thirty two registers. We considered having register banks in order to allow more registers, but determined it would add too much complexity. We opted to leave the ISA at eight registers and to leave out the zero register in order to free up one more register.

Branches also suffer, having only one register operand. In order to leave a larger immediate field for the branches, the BRC architecture does not include two register branch instructions. Although this may frequently require an extra instruction frequently to computer the branch, it leaves the branch with enough

	5	3	3	3	2
3R	op	rs	rd	rt	func
2R	op	rs	rd	imm	
1R	op	rs	imm		
0R	op	imm			

Table 2: Instruction Formats

immediate space to be useful.

The last problem that the restricted instruction size causes is the small immediate fields. Immediate ALU instructions are left with only a five bit immediate, branches with an eight bit immediate, and jumps with an eleven bit immediate. ALU instructions can always be synthesized by first computing the full immediate into a register, and then performing the corresponding three register instruction. Branches and jumps can create extra jumps and branches in the code in order to jump multiple times to get to the right location.

However, we want to minimize the number of extra instructions needed in as many cases as possible. In order to do this, we borrowed another instruction from the 16 bit MIPS ISA, the `extend` instruction. It is used to extend the immediate of almost any instruction to a full 16-bits. This is accomplished by issuing an immediate instruction (which is a zero register format instruction), with the upper eleven bits of the full immediate. These eleven bits are concatenated with the lower five bits of the *following* instruction's immediate. It is always the lower five bits of the actual instruction's immediate for consistency. This works out well as the `extend`'s immediate is adequate length to pad the smallest immediate (five bits) to the full 16 bit data size. See Appendix A for exact details of the `extend` instruction functionality.

The addition of `extend` prompted a change from the traditional MIPS instruction of `LUI`, or load upper immediate, to `LI`, or load immediate. `LUI` is normally used to fill in the upper halfword of a register, while an `ORI` is used to fill in the bottom half, using the zero register as the source register. Since we do not have a zero register, filling in the bottom half of a register could require more than one instruction. With the `extend` instruction available to us, we can switch the `LUI` to a `LI` in order to be able to fill small immediates in one cycle, and full immediates in two, similar to full MIPS.

We also included one other special instruction, `break`, which halts the processor in order to allow for it to be debugged by reading out the scan chain and to access the on chip memory (see Section 4.4). We also included some other instructions for completeness in the ISA as optional instructions, that we chose not to implement, as we felt they were unnecessary for this chip. These include MIPS style multiply and divide instructions, a system call instruction, and a return from exception instruction. For a complete ISA description, see Appendix A.

4.2 Pipeline

As mentioned in Section 3.2, we decided on a three stage pipeline for our architecture. The first stage includes instruction fetch and decode. The second stage is where our architecture is unique. Instead of having the ALU stage and MEM stage in series like in the traditional MIPS pipeline, the BRC architecture has them in parallel. Since they do not each have their own pipeline stage this helps with the cycle time, as the two in series would create many critical paths. It should also help with power, as now only one of them will ever be needed at a time and the other one can be clock gated in order to decrease power usage.

The downside of this decision is that we no longer have the ability to do base-offset addressing, and can only use register indirect addressing. This helps the ISA, as it allows us to use three register format which has extra function bits, but can cause extra instructions to be needed in certain situations. The offset is very useful for array, structure, and stack accesses and will require an extra add instruction now in order to increment the base register. This is a trade off we were willing to make.

Also in the second stage of the pipeline are two custom state machines that perform GPS specific calculations. One is a gold code generator, and the other buffers the ADC input (see Sections 4.2.10 and 4.2.11

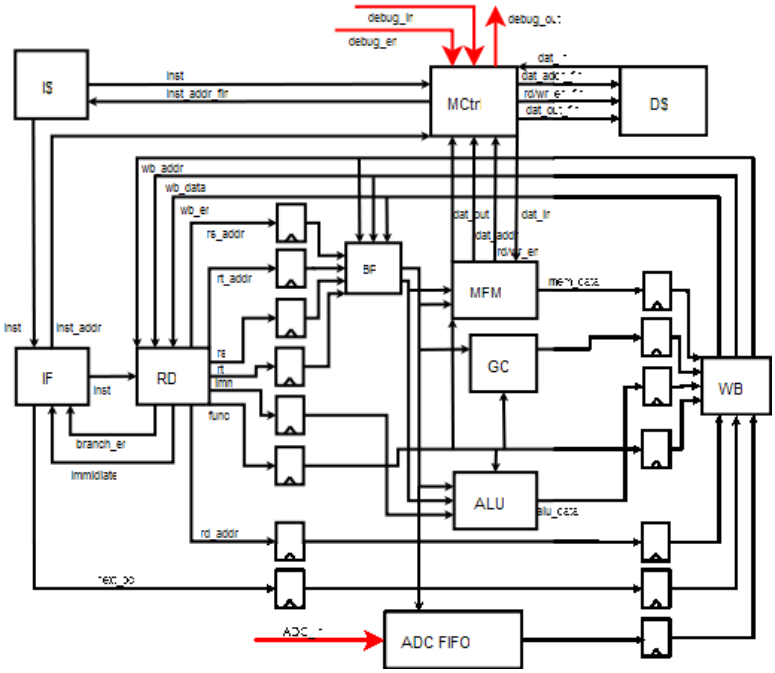

Figure 3: Block Diagram of Full Core

respectively). The second stage also requires one bypass path (Section 4.2.7) from the third stage in order to avoid data hazards. The third stage contains only the write back. We also have several sub units which lie outside of the pipeline data path. These include a break unit (Section 4.2.6), a clock state machine (Section 4.2.9), and a memory controller (Section 4.2.8), all of which are described in more detail in their respective sections.

4.2.1 Instruction Fetch

The first block of our pipeline is the Instruction Fetch (IF). As IF and RD are both inside the same stage of the pipeline, the IF stage must complete its work in the first half of the cycle: as a result, the logic in this block is minimal. The two main tasks that IF handles are updating and storing the program counter (PC) and fetching the next instruction from the instruction cache (i-cache).

On each positive edge, IF updates its internal PC register. In case of a stall, PC is updated to itself and the same instruction is retrieved. If a branch or jump is taken, then the PC is updated to the possibly extended immediate coming received from RD. Since RD and IF are in the same cycle, this eliminates the need for a branch delay slot, and all jumps and branches occur in the instruction after they are retrieved. Finally, if no control instructions have been issued, the PC updates to PC+2, since our instructions are two bytes wide.

The i-cache is word-addressable and, during normal operation, read-only. As a result, the address lines feeding out of IF are one least-significant bit shorter than the PC register.

4.2.2 Register Decode

The Register Decode (RD) block of the pipeline makes up the second part of the first pipeline stage. In order to reduce power requirements, this portion of the stage is triggered by the negative edge of the system clock (as opposed to traditional stages, which are triggered by the positive edge in most architectures). The RD block is responsible for determining the intent of each instruction and sending data based on this intent to the other blocks within the pipeline.

When the IF receives an instruction, the RD block will read the register locations specified by the instruction. The data from these registers is then latched in order to be sent to the next stage of the pipeline. To reduce logic in an effort to control power dissipation, the register operands are transmitted even if the instruction does not need them. In parallel, the value of the immediate specified by an instruction is generated within the block. As there are three immediate widths, the instruction opcode is used to decode which width is appropriate. The RD block also handles the `EXTEND` instruction operations (see Appendix A). As the `EXTEND` instruction will specify the 11 most significant bits to be used for an immediate which will arrive the *next* cycle, the RD block incorporates logic that detects an `EXTEND` instruction and stores the specified bits into a register for use the next cycle.

The RD block also handles branch computations. Traditionally, branch decisions and destinations are calculated in the ALU of a processor. However, in order to prevent the use of inefficient stalling logic or branch delay slots, and to eliminate the need to create a branch predictor, a simple ALU was placed within the RD block to compute these branch decisions. This branch computation unit (BCU) supports the ability to perform jump register and linked jump instructions. Once the address is computed, the decision of the branch, as well as the value with which the PC should be updated, are sent to the IF stage, which updates the PC register before it fetches the next instruction.

As part of its branch handling, the RD block also has the ability to stall for one or two cycles should a branch instruction be dependent on data that is still being generated inside the pipeline. Data is currently bypassed to the second stage of the pipeline from the third stage should it be needed (see Section 4.2.7). However, this cannot be done with the RD block as it is negative-edge triggered, and its data may not have been produced yet. As a result, a safer and more power-efficient solution is to stall the RD block until the data in the pipeline is written back to the register file, at which point the RD block can then serve the branch request with the needed register data.

The RD block uses a quarter-phase-shifted clock to update register values in the write back (WB) stage. Since the data is available from the WB stage very quickly, the RD block can read the value for storage at any time. However, it was found that a critical timing path propagated through this write back mechanism. In order to ease the timing constraints, this quarter-phase-shifted clock was introduced so that register data could be updated beginning at the middle of the system clock negative phase. Such a mechanism, although resulting in an additional power consumption, allows the register update to have three quarters of a cycle to complete, eliminating the critical path. For information on how the shifted clock is generated, see Section 4.2.9.

4.2.3 ALU

The ALU stage is made up of an ALU, a mux to select the second operand, and a decoder to decode the opcode into a select line for the mux and a function code for the ALU. The mux is a simple two to one mux, and chooses between the value in the RT register and the given immediate, after it has been properly extended. The ALU supports the following instructions:

- Add
- Subtract
- And
- Nor
- Or
- Xor
- Shift Left Logical
- Shift Right Logical
- Shift Right Arithmetic
- Set Less Than
- Set Less Than Unsigned
- No Operation
- Population Count

All of these are standard choices for an ALU except for population count. The population count of a number is the number of ones in that number's binary representation. This instruction is needed for our GPS calculations, and we decided to include it in hardware as a single cycle instruction. It appears in the inner loop of our code, and would take a many more cycles without dedicated hardware. This is implemented as a tree of small adders. Although this has the potential to be slow and was watched for being a critical path, it ended up not hurting our cycle time.

There is also a lack of support for multiply or divide. As our chip is suppose to be low power and these instructions are computationally intensive, we decided to leave them out. This is acceptable as our GPS code can be run without them by using exclusive-or and population count, as described in Section 5.1.1. If they are needed for other purposes they can be emulated in software.

4.2.4 Memory

The MEM block is in the second stage of the pipeline, running in parallel with the ALU and GPS blocks, allowing only for register direct memory operations. The tasks that MEM needs to handle are shifting incoming and outgoing data from the data cache (d-cache), computing the write and read-enable lines, and outputting the correct address to the memory controller.

When MEM receives output data to be stored or input data to be loaded, it checks to see if any additional processing needs to be done. Processing only needs to occur if the command received is a byte instruction (such as lb, lbu, and sb), as the data for word-length memory instructions does not need to be changed. For a store byte instruction, the data is shifted left one byte if the least-significant bit of the address is 1. For the load byte instructions, the data is shifted right one byte if the least-significant bit of the address is 1, and the topmost eight bits are zeroed out for lbu and sign-extended for lb.

The write and read enable lines are both active low, and are slightly more complex than merely checking whether or not the instruction is a store or load due to the write-word masking ability of the cache. The read enable line is low on a load, and high otherwise. The write enable line is two bits wide, with each bit being driven low if the corresponding word is to be written (see Table 1).

Like the i-cache, while the d-cache can perform byte-accesses by shifting the data word left and sign- or zero-extending it, the cache itself is word-addressable. As a result, the least-significant bit of the address line is discarded when passing it to the memory controller.

4.2.5 Write Back

As the third and final stage in the pipeline, the write back stage determines if the registers will be written with data, and which of the data lines passing through the design contain the relevant data and register address. An enable line, controlled by the write back stage, will cause the register write, and the data will be the result of multiplexing the data lines, with the select lines chosen by decoding the op code and function from the PC. The data can come from one of five different data sources: the ALU, the memory, the ADC unit, the gold code, or even the branching unit if the instruction is a linking jump. The write back data lines were originally the only lines available for bypass forwarding, but this changed in a later revision of the design.

Traditionally, the write back stage would be combinational logic following the final state holding registers. In our design, however, we deviate from this by splitting up the write back stage into two segments; write back computes logic not only after the registers, but before as well. Splitting the write back stage such that it straddled these phase three registers separated the control lines from the data, and allowing this control logic to be exposed immediately after decoding. With the decision to stall in certain situations, this decoding logic was needed before the third pipeline stage. The write back, already decoding these signals, was split instead of duplicating the logic, reducing the number of gates and decreasing power. The multiplexers after the registers also enjoy a shorter evaluation time, but since they are not on any of the critical paths this is a largely fruitless benefit.

4.2.6 Break Unit

The break unit is a simple system for determining whether or not the system should be halted for debugging. It is set by a break instruction, at which point it notifies the clock state machine (Section 4.2.9) to begin ramping down the clock. It can be reset by an external input to the chip after debugging has been completed. See Section 4.4 for more information on the DFT capabilities of the chip. This is necessary in order to examine the state of the processor and to set the state of the processor externally. This allows it to be debugged by loading up a known state, running it for one cycle, than examining the next state. It can also be used just to check the state of a program at a certain point for correctness.

4.2.7 Bypass Unit

The bypass unit allows for data to be bypassed from the third pipeline stage to the second pipeline stage. If there are two consecutive instructions with a data dependency, the second will read the old value from the register file. However, by the time it is ready to consume the value in the second stage, the value is being written back in the third stage. The bypass checks to see if the value being written back is the same address as the one currently being consumed, and forwards it if needed.

4.2.8 Memory Controller

This unit is not a memory controller in the normal sense. It simply connects the instruction and data memory to the core while it is running, and allows for data to be accessed by an off chip debug port while in debug mode. It has the ability to read and write to all instruction and data memory addresses. Along with debugging it is used upon powering up in order to program the instruction memory with the code the chip is to run. It cannot be used to support off chip memory as it does not implement a standard protocol. Extending it to do so would not be trivial, as it is designed to only allow external access while it is being run off the slow debug clock and not for real time access. It also currently only allows for either the data or instruction cache to be accessed at a one time, which would create a bottleneck. For more information on using the memory controller through the off chip interface, see Appendix D.

4.2.9 Clock State Machine

The functionality for generating the system clock and switching between the system clock and debug clock is provided by the clock state machine (CSM). The processor can either take an external clock input, or it can use an internally generated clock to create the system clock. Both the external and internal clock are referred to as the fast clock, which runs at twice the frequency of the system clock. The fast clock is needed to allow for state changes within the CSM that do not fall on any of the system clock edges (which would cause glitching that could propagate to the system). The CSM divides the fast clock into two to generate the system clock at the desired frequency. A quarter-phase-shifted clock is generated here as well, relying on registered logic that is triggered on the negative edge of the fast clock. This shifted clock is used in the RD stage (see Section 4.2.2).

The major functionality of the CSM is to provide a gradual transition from the system clock to the debug clock when the external FPGA assumes control of the processor. An instantaneous switch between the clocks could cause serious issues with glitching and potential current fluctuations from the clock distribution network, due to the multiple-order-of-magnitude difference between the clock speeds. To overcome this, once the signal is received by the CSM stating that the FPGA is ready to start debugging, the CSM will divide the system clock in two repeatedly, eventually slowing down the clock until it reaches one 256th of its initial frequency. The clock is slowed down using a mask and override bit that is generated as an output of the Mealy state machine. At this point, this slowed down clock is transitioned to the debug clock when their current phases are aligned, allowing for a clean transition. Such a transition is not necessary for when the system clock resumes control, as increasing the clock speed does not result in current aberrations. The CSM will ensure that the debug clock and system clock are on the same phase when they transition to the system clock.

The clock state machine is also responsible for generating a clock for the ADC unit. According to specifications, the ADC is currently expected to run at a twelfth of the frequency of the system clock. The CSM uses a resetting counter to generate this slower clock, which allows the CSM to continue ADC clock generation even during a switch between system and debug clocks. The clock state machine also supports an external ADC clock, if the internal clock either does not operate as expected or a new frequency that is not dependent on the internal system clock is desired.

4.2.10 Gold Code

The Gold Code system is one of the special functional units that have been included in the first revision of the BRC architecture to accelerate software GPS algorithms. A Gold Code is a pseudo-random noise sequence that can be generated using shift registers with feedback to define the next bit to be shifted in. The functional unit in the BRC that computes these codes is a state machine that contains separate sets of shift registers for each satellite currently being tracked. The system can be accessed through a special instruction and it will return the next bit, or "chip", in the sequence. For a much more detailed description of how this system works and how the Gold Code is used in GPS computations refer to the Gold Code section of the GPS top-level design description 5.3.

4.2.11 ADC

The primary function of the ADC system is to gather the input from the serial ADC input and buffer it in a reserved memory space. Because writes to this memory take place automatically when a data word is ready, the system must also control the location of writes and the next read. In addition to this basic DMA functionality, the ADC system is designed to optimize GPS calculations by removing the Doppler shift from the input automatically as it is stored in memory. Incoming data is stored in separate memories for each satellite currently in tracking and each of these memories is accessible through the reserved instruction dedicated to loading data from the ADC. For a significantly more thorough discussion of the functionality of the ADC input system and how that functionality aids in performing GPS baseband calculations refer to the ADC portion of the top-level GPS system section 5.3.

4.3 Memory System

All memory is located on chip for this design, with 2kB available for instruction memory and 8kB available for data memory. As discussed in section 3.3, instruction memory and data memory are separated into two distinct units, with additional memory reserved for the ADC unit. The memory controller, as discussed in section 4.2.8 provides the user with access to these memory units, but this is intended only for debugging purposes and general use is restricted to load and store commands as defined in the ISA. Currently there is no direct support for off chip memory, but a dedicated user could implement one through the debugging interface provided, although such a system will not be discussed in this report. The Big Red Chip memory system uses local, fast, on chip memory that is expected to evaluate within a clock cycle, comparable to modern computer architecture caches. The memory updates on the negative edge of the clock cycle, both to allow time for the next PC address to compute and for the data memory to compute in one cycle.

The memory system specifications were determined not by our requirements, desires, or even engineering ability but instead determined by the ability of the memory compiler provided to us. The ARM memory compiler will generate either SRAM cells, which we use for our instruction and data memory, or an array of registers. The icache is treated as a read only block during normal operation, and thus bit level reads masks were not included as such an ability is not supported in the ISA. Data memory, and the ADC registers, both support 8-bit reads and writes, however, enabled by masking the appropriate bits in our Verilog. Both load and store byte operations require similar masking to match the BRC Verilog to expected behavior by the memory units. All parts of the memory system have distinctive read and write ports, whose exclusivity is limited only by timing requirements generated by the tool; data can be written and then read in succession within a clock cycle as long as the cycle is sufficiently long.

4.4 DFT

Based on the available options and their applicability to our processor (see Section 3.5), we chose the scan chain with extra boundary hardware and control logic for externally injected tests as the solution to include Design for Testability in our design. Boundary hardware and control logic consist of the break logic (Section 4.2.6), the clock state machine logic (Section 4.2.9), and adding scan enable and scan chain wires to the design. The scan chain consists of all registers internal to the processor core hooked up serially. The beginning of the scan chain has an input from an off-chip source from which external tests can be injected into the core. The end of the scan chain is where the values in the scan chain can be serially read out from the processor. The values of the scan chain can be compared against expected values when externally driven test cases are run on the core.

Extra control logic had to be included within each register for the scan chain to function correctly. When a scan enable signal is set, the register will take the previous register's LSB into the MSB position while at the same time shifting out its own LSB. These shifts happen on the positive edge of the clock.

The scan chain functionality can be implemented when the processor has received a break command and halted normal operation of the core. The clock is ramped down to a debug clock (for a description of this process, see Section 4.2.9), and control is given to an external FPGA source. For the scan chain to start being read, the scan enable signal must be set high.

4.5 Verification

4.5.1 Methodology

A strong verification effort was critical to this project and can be broken down into several stages, each corresponding to a specific design cycle stage. These stages include verification on the block level, on a non-pipelined core, on the pipelined core, and post-silicon debug for hardware verification.

Verification on the block level consisted of writing test benches directed at testing the functionality of each block. For example, when testing the RD block, it was essential to test the decoding of each instruction within the ISA to ensure correct behavior. At this point in the test cycle, the goal was to catch and fix Verilog syntax errors as well as basic functionality errors. Test benches consisted of instantiating the block to be tested within a test bench, and simulating inputs to the module by assigning values in an initial block

within the test bench. Results of the test benches had to be manually checked with pre-calculated expected behavior.

When all blocks were completed and suitably verified at the block level, they were wired together under a common CPU module to test functionality on the core level. At this point, we did not want to worry about pipelining the core and having to debug pipeline issues as well as functionality issues, so we made the core a "simple core" with no pipelining to only test functionality. The simple core did not yet have instantiated instruction and data memories, which limited testing to instructions that remained within the core. Load and store instructions were not tested on instantiated memories at the core level until pipelining was implemented. As a work around to this issue, we created an array in hardware the size of memory to simulate data memory. To simulate instructions being fed into the processor, we assigned the inputs to the core in an initial block, very much like tests implemented on the block level. At first, as in the block level tests, results of each test bench were manually calculated and checked. However, the random test case generator and software simulator became available to us at the end of this testing phase.

The next stage in completing verification of the chip was on the "full core." This instance of the design was of the simple core with the three pipeline stages and included the GPS modules. The goal of this stage was to completely verify both syntactic and functional correctness in the final design of the processor. Since pipeline registers had been added, the wiring had to be verified to ensure correctness in the connections between block levels. Furthermore, because we were trying to fully verify the core, we wanted more than just manually written test cases to test a larger design space than would be possible by writing test cases manually. To this end, we created a random test case generator and a software simulator to be able to more fully test the core. The reasoning behind a random test case generator is that it would be able to create many more combinations of test patterns than would be possible with directed testing of the core and thus provide greater coverage of the entire design space. A deception of how the random test case generator and the software simulator worked to identify errors in the design is described in Sections 4.5.2 and 4.5.3.

Coverage describes how much of the design has been tested by the test cases that have been run on that design space. VCS, a Verilog compiler and simulator, provides a tool that gives coverage metrics based on different criteria including lines of code touched by the test bench, as well as conditional paths taken among others. For a full reference on how to use the coverage tool as well as for a more in depth description of what criteria are included in coverage, please see Appendix F. It should be noted that coverage metrics should only be run when the design is known to be almost completely correct as coverage numbers are irrelevant if the design is not correct. The coverage tool allows us to see what has not been covered by the test case and run tests to more fully verify the core.

One other aspect of verifying the full core was testing the scan chain. This is an extremely important part of the verification effort as the scan chain will be the primary tool used to debug our processor once it comes back from fabrication. To test the scan chain, we first had to determine the number of bits in the chain as well as their ordering; this is described in Appendix E. The test bench then consisted of allowing a certain test pattern to run on the processor, halting the processor using a break instruction, reading out the scan chain to ensure that it matches the expected scan pattern, and then re-writing the scan pattern back into the processor and restarting normal operation.

The final stage of verification will be completed after the chip comes back from fabrication. This last phase is to test that when the chip returns from fabrication, it functions as in simulation. In this stage, we

will be making extensive use of the scan chain to ensure correct behavior within the chip. Work remains to be completed for this stage however. Comprehensive test patterns as well as their respective scan chain values must be created so that we can confidently release the chip as fully functional. A FPGA will be connected to the core to inject test patterns and can be used to ensure that the scan chain has correct values at certain times within the test (see Section 7.1).

4.5.2 Simulator

As described in Section 4.5.1, we wanted to be able to run random test cases on our RTL. In order to be able to tell if they ran correctly or not, we need to have a reference design to compare the results to. The reference design we used was a software simulator of our ISA written in Python. It took a binary input file, and could be run in batch mode to log its results for comparison to the RTL implementation, or in interactive mode to be able to better debug the program. Also, since the GNU assembler port by the software team took longer than expected, a simple assembler was written in Python which only did simply mnemonic to binary conversions to use with our simulator.

The simulator works by using the given binary file as the instruction memory. The current position in the file is used as the PC. It automatically increments every time the program reads a word, so it only needs to be modified on jumps and branches by seeking to the appropriate section of the file. Decoding is done by checking the opcode against lists for each instruction format, than breaking up the instruction into sections based on which format the opcode specifies the instruction as. The instruction is then executed by performing the specified operation. This step combines the ALU, MEM and WB stage into one step.

All opcodes are defined as constants in a separate file, along with a list of which opcodes are in each format. There is also a list of instructions and their formats for the assembler to be able to parse the assembly files. This file should make it easy to expand the simulator and assembler tools for changes and enhancements to the ISA to be made in the future (see Section 7).

The simulator includes debugging features in order to help find bugs when the outputs from the software simulator and RTL implementation differ. These include the ability to have each instruction printed out in mnemonic form when it is run, including the values that are in the registers being used at the time of execution. This helps to watch the control and data flow of the program. There is also an interactive prompt that appears when a break instruction is encountered. The program has commands for reading and writing to both registers and memory, step through the program, and to restart or resume the program.

The simulator was indispensable during debugging, but could use some improvements. If it could output a trace of instructions in an identical format to Verilog, it would be much easier to compare their instruction traces with `diff` to find where errors occur. It is also important to remember that the simulator is not guaranteed to be correct, and sometimes the bug was in the simulator and not the RTL.

4.5.3 Random Test Code Generator

In order to run random test cases we needed a way to generate them. A random test case generator was written in Python in order to accomplish this. It does this by organizing the different instructions by types, and randomly chooses a type with a certain percentage, than randomly chooses an instruction from a list of that type. Their are many adjustable parameters including the percentages of different instructions,

minimum and maximum values for loop iterations and immediate values, and max control flow nesting among others.

The program begins by initializing all of the registers to random values. It then decides if the program will contain a function, and generates them with special rules to ensure the test case is sane. It then begins to loop through and randomly decide what to do next. Data instructions randomly choose a data instruction with random operands. However, data dependencies are kept track of in order to reduce values being overwritten before they are consumed to maximize the chance of incorrect operation manifesting itself in the final register values. Memory instructions are generated similarly, with the generator trying to store to addresses before it loads from them. Due to the dynamic instruction trace, this is not always guaranteed. Therefore, data memory needs to be assumed to be initialized to zeros in order for test cases to be guaranteed to match from the RTL to software simulator cases.

There are also control flow statements. In order to avoid infinite loops and other unwanted code constructs, all control flow is in the form of for loops, if statements and function calls. For loops have a random number of iterations and care is taken not to clobber the loop register to avoid infinite loops. If statements have random tests, including comparing two values and comparing values against zero. The if statement also has a random chance of having an else clause or not. The nesting of these control flow structures is kept track of by the generator and control flow structures are randomly ended, making for varying levels of nesting and varying lengths of the clauses in control flow structures.

There are also function calls if a function is generated for the test case. Loop indices are saved to memory before the call and restored afterward to avoid clobbering the loop registers. Nested function calls and recursive function calls are disallowed in order to avoid infinite recursion and to avoid dealing with stack semantics. Nothing is pushed or popped from the stack, the return address is simply kept safe in register seven and all values are treated as global.

Ending the program is another random choice. This allows for programs of varying length. If it ends the program while control flow structures are still open it first ends all of them, then ends the program. All programs end with a break in order for our simulators to detect a correct program termination and to assist in debugging.

4.6 Synthesis Tools

To fabricate the chip, the Big Red Chip project synthesized the high level Verilog into a gate level standard cell equivalent, and then used automated place and route tools to assist in generating a layout. The standard cells libraries, IO Pad libraries, wire technology, and memory generators are all licensed from ARM for the IBM 130nm 8RF process. The fabrication was scheduled for a May 12th tapeout, creating a tight schedule to achieve in one year. Three primary software tools were used to assist in meeting this deadline: `VCS` Verilog compiler, the Synopsys DesignVision synthesis tool, and the Cadence Encounter backend tool for place and route. A full, independent user's guide is written based on the design experience, and selected excerpts are available in Sections 8 and 9.

The Synopsys DesignVision tool synthesized the BRC high level Verilog, producing logically equivalent standard cell Verilog. Using the provided technology files, the design constraints that we provided, and custom scripts written by the BRC team, gate level Verilog produced not only met our initial goals of 100MHz but additionally met all hold timing. Several changes to the Verilog were necessary so that the

code could be synthesized, but more importantly the timing reports generated by DesignVision revealed the critical paths in the design. These critical paths required several major design changes, but all were either eventually solved or proved to be false. Cadence Encounter takes the gate Verilog and allows the designer to place and route the design. The tool supports both automated flow and custom routing, including support for not only standard cell placement but other design concerns such as power rails, pad IOs, the clock network, and fixing DRC violations. The results of this synthesis are documented in Section 6.3.

4.7 Toolchain

The toolchain supports generation of code in both little and big endian formats. By default, code is generated for a big endian machine.

4.7.1 GNU Binutils

GNU Binutils is a collection of binary tools. The main ones are:

- as - the GNU assembler
- ld - the GNU linker
- objdump - Displays information from object files
- objcopy - Copies and translates object files
- readelf - Displays information from any ELF format object file

The process of porting binutils involved porting BFD (Binary File Descriptor), Opcodes (the new ISA), GAS (GNU Assembler) and LD (Linker). The other binary tools make use of these tools and hence were also ported in the process.

- BFD
  BFD allows low-level manipulation of the object files created. The first task was to decide on the underlying object file representation. Of the available options, the final decision had to be made between COFF (Common Object File Format) and ELF (Executable and Linking Format). We chose ELF because it is more flexible in representing object files and is the object file representation used on all Linux machines. ELF also has better support for dynamic libraries. This feature is not required now, but may be useful in the future. Relocation support was also added to BFD. The relocations defined for our architecture are:
  - R_BRC_16_IMM5
    16-bit absolute relocation for instructions which have a 5 bit immediate field.
  - R_BRC_16_JMP
    16-bit absolute relocation for jump instructions.
  - R_BRC_16_PCREL
    16-bit PC relative relocation for branch instructions.

Name	Usage
%0	Argument word / Return value
%1	Argument word
%2	Local register variable
%3	Local register variable
%4	Local register variable
%5	Frame Pointer
%6	Stack Pointer
%7	Return address

Table 3: Register Usage

  - R_BRC_32
    Standard 32 bit absolute relocation.
  - R_BRC_16
    Standard 16-bit absolute relocation.

- Opcodes
  Opcodes contains the ISA for our processor. To ease the development effort for opcodes, CGEN (the CPU tools GENerator) was used. CGEN specifies a description language for describing the architecture and organization of the CPU without reference to any particular application.

- GAS
  The next step in porting binutils was to port GAS. To ease the programmers job of writing assembly, we provide an "autoextend" feature. With this feature enabled, the programmer no longer has to explicitly code an `extend` instruction. Whenever the assembler finds an immediate which does not fit in the instructions immediate field, the assembler produces an `extend` instruction. This feature can be enabled by writing the directive `.set autoextend` in the assembly file. Similarly, this feature can be disabled by writing the directive `.set noautoextend` in the assembly file. This feature is enabled by default.

- LD
  The next step was to port LD. We provide LD with the layout of the final executable file that is generated. The linker sets the routine named `_startup` as the entry point for the executable.

4.7.2 GCC

- Register Usage
  Table 3 shows the register usage.

  All registers are global and visible to both the calling and called function. Register %4 is a caller saved register i.e. if the register value is needed after the call then the calling function should preserve the register before making the call. All the other registers belong to the called function.

- Stack Frame
  Figure 4 shows the stack frame organization.

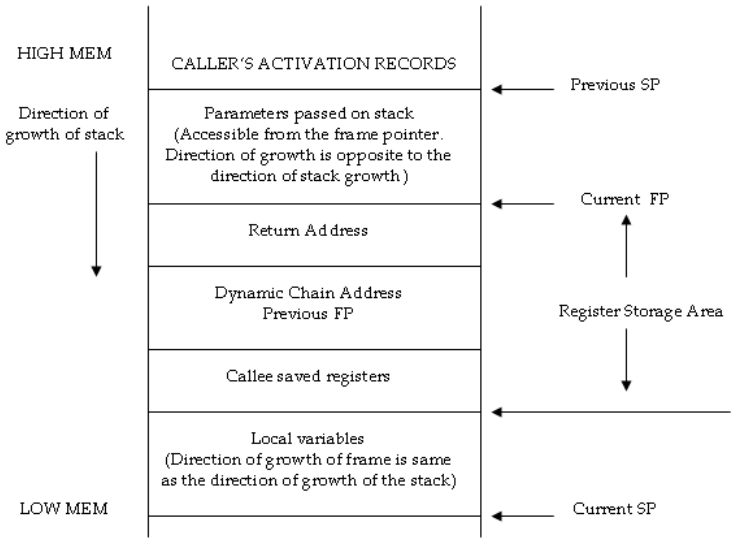

Figure 4: Stack frame layout

Some key points about the stack frame are :

- The stack is word aligned. This is a software convention even though the architecture does not require any alignment of the stack. This may lead to increasing the argument's size to make is a multiple of words by padding it.
- Argument words are pushed onto the stack in reverse order (that is, the rightmost argument in C call syntax is pushed first and hence has the highest address), preserving the stack's word alignment. Incoming arguments reside in the stack frame of the caller.

With all the aforementioned architectural restrictions made to suit our requirements, considerable effort needed to be put in to the compiler to work around them.

- No indexed addressing mode.
- Limited branch instructions - the architecture only has `bgez, bltz, bnz and bz`. All other branches need to be emulated using these branch instructions.
- No `jalr` instruction.
- Software emulation for multiplication and division.
- Software emulation for floating point arithmetic.

4.7.3 Newlib

Newlib provides the C and Math libraries for use on our embedded system. Porting newlib involved providing definitions of machine-dependent details like the endianness and implementation of setjmp and longjmp. Newlib also contains a package called libgloss. Libgloss is a library that provides the startup code and stubs

for all system calls that are needed by the libraries. As of now, these stubs are written to return success for most of the system calls. This allows us to write any code in a higher language and it will work correctly. Once we have some kind of debugging mechanism in place on our board like a serial interface or Ethernet, we can replace these stubs with code which can talk over them.

5 Software GPS on the Big Red Chip

"Simplicity is the final achievement."
— Frederic Chopin

5.1 Signal Acquisition and Tracking

5.1.1 Correlation Computations

Despite having special functional units to optimize the architecture for GPS, the bulk of the computation necessary to acquire and track satellites will be performed in software. Although control adds a significant amount of complexity, the primary task of the CPU will be to continually compute correlations between the incoming data and a locally generated copy. The processor must find separate correlations for each satellite at the same rate at which data is input to the system. For this reason, it is these computations which set the minimum requirements on performance, and the goal of all optimizations is to make this process as efficient as possible.

The incoming GPS signal will the product of a pseudo-random noise signal (the Gold code), the carrier frequency, and the data bits. To extract the data bits the signal must be multiplied by local copies of the frequency and properly aligned Gold code. In figure 5, one can see visually how the GPS signal is modulated with the code and carrier frequency.

The accumulated value of the multiplication of these three signals provides the correlation. In a typical software radio, this would be done using a fast multiply-accumulate instruction. This approach is costly from a power standpoint and not feasible at all without a fast multiplier which would likely require a full-custom design. Instead, the BRC system will incorporate the work of Cornell's Professors Mark Psiaki and Paul Kintner, who developed a bitwise method for computing GPS correlation[5].

Under this bitwise method, incoming signals that have been quantized to a single bit can be correlated to a local copy in a bit-wise manner with exclusive-or (XOR) operations instead of multiplies. For a general-purpose CPU, this is a very significant savings in speed as well as power. Following the XOR operation the sum of the number of zeros less the number of ones represents the correlation of the signal. To facilitate these sums the BRC includes one additional special instruction, a population count.

As will be discussed in detail in the ADC section 5.2, the BRC system is capable of removing the Doppler

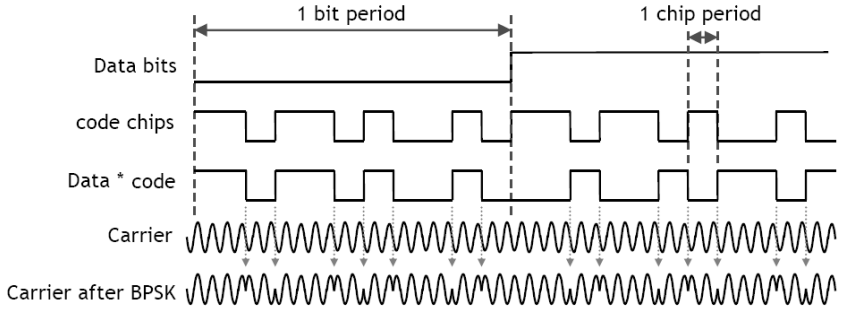

Figure 5: Modulation of a GPS data bit with the Gold Code and carrier frequency[2]

shift of a given signal in hardware. Normalizing all of the inputs to the center frequency provides additional benefits when included in the bitwise scheme for computing computations. Because all correlations will take place against the same frequency, the sample rate of the Gold code sequence will be constant. With a properly chosen input sample rate, this allows each chip (each bit of the Gold Code is referred to as a "chip" because it carries no data) of the code to become a constant length frame over which the correlations are performed. Since the Gold Code sequence chipping rate is a constant at a given carrier frequency, the sample rate becomes: $f_{sample} = b * f_{chip}$

In this equation b represents the number of bits per write in the ADC system. The BRC system data input system writes to the ADC memories in eight bit words, as this length allows for efficient byte-length loads and provides the required Doppler frequency resolution. From this equation our sample frequency becomes 8.184MHz. Note that this frequency provides sufficient over-sampling given that the maximum information carrying frequency should be approximately 3MHz.

To better illustrate exactly how the correlations are computed using bitwise methods and incorporating our special purpose instructions we have included an appendix with in-line assembly code written for the BRC architecture that performs GPS correlations. Refer to Section G after reading the portions of the document describing the functionality of the special purpose instructions.

5.1.2 Algorithmic Control Loops

Without any prior information, the system will have no initial knowledge about which satellites are in view or how quickly they are moving. This means that the carrier frequency can have a Doppler shift anywhere within +/-5kHz of the center frequency (for our purposes). In addition, the processor will not know the starting time for the coarse acquisition code of those satellites that are in view. The process of acquisition involves searching all possible frequency and gold code start times until detecting a spike in correlation, indicating a recovered signal. Figure 6 illustrates the result of such a search.

Following acquisition, the GPS system must continue to perform correlation computations to ensure that the phase of the coarse acquisition (C/A) code or the Doppler frequency do not drift far enough that the signal is lost. This can be performed using a delay locked loop that requires correlations from early, late, and prompt versions of the C/A code (C/A stands for coarse acquisition and is another term that refers to the Gold Code or PRN code). These three versions of the C/A code are deliberately shifted because the correlation falls linearly with time offset from the peak. By monitoring the difference between the correlation values for the early and late versions of the code the algorithm can compute an accurate code start time for the next iteration.

In addition to correlations required on shifted versions of the PRN code, computations must also be performed on both in-phase and quadrature representations of the local frequency. The GPS system implemented on the BRC is a non-coherent delay locked loop, meaning that the local intermediate frequency representation does not need to be aligned with the signal carrier. Instead, correlations are performed on

Instruction	Function
`POPC rd, rs`	`GPR[rd]` $\leftarrow$ `pop_count(GPR[rs])`

Table 4: Format for Population Count Instruction

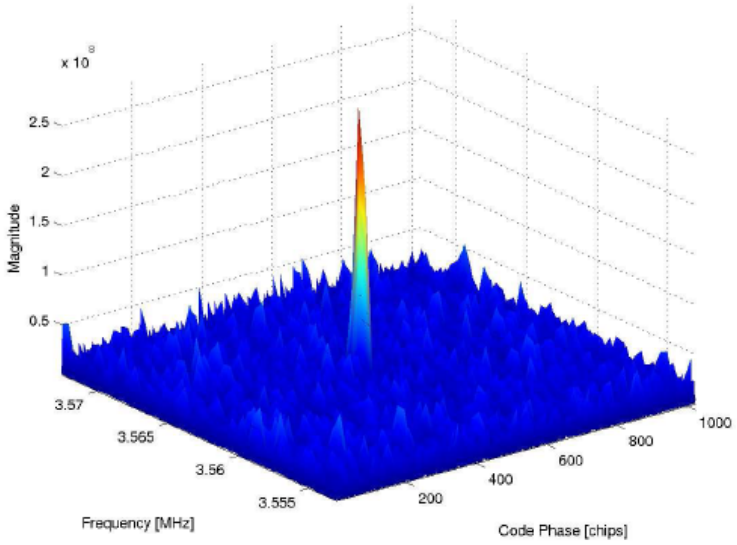

Figure 6: Result of an acquisition search with spike indicating satellite in view[2]

two local frequency copies that are 90 degrees out of phase. Using these two correlations, the real value for correlation can be found using a Pythagorean identity.

$$(Correlation)^2 = (Correlation_{InPhase})^2 + (Correlation_{Quadrature})^2 \quad (1)$$

For these reasons all data pulled from the ADC must go through six sets of correlations for each satellite currently being tracked: in-phase and quadrature computations on early, prompt, and late code shifts. We have included a block diagram of the computation and control that must operate continuously in real time in figure 7. Although we have not yet tested this optimization, one should note that it should be possible to track the satellites using only the early and late code-shifted correlations. This optimization would remove a third of the instructions required in the critical inner correlation loop of the algorithm but would reduce the sensitivity of the system because early and late correlations will be significantly lower than the prompt version.

Although figure 7 shows that the GPS algorithm is a complex system, the control will only execute after correlating with a minimum of one full Gold Code sequence. Each Gold Code sequence is 1023 chips in length, meaning that many correlations will take place between each subsequent control iteration. With the common adage "Make the common case fast" in mind, the BRC system concentrates on making each correlation as efficient as possible. This is the basis for our system including functional units to produce the Gold Code sequences and remove Doppler shift in hardware while lacking a hardware multiply unit. The following sections describe in detail the operation of these special units and how they enable increased efficiency for the common case: computing correlations.

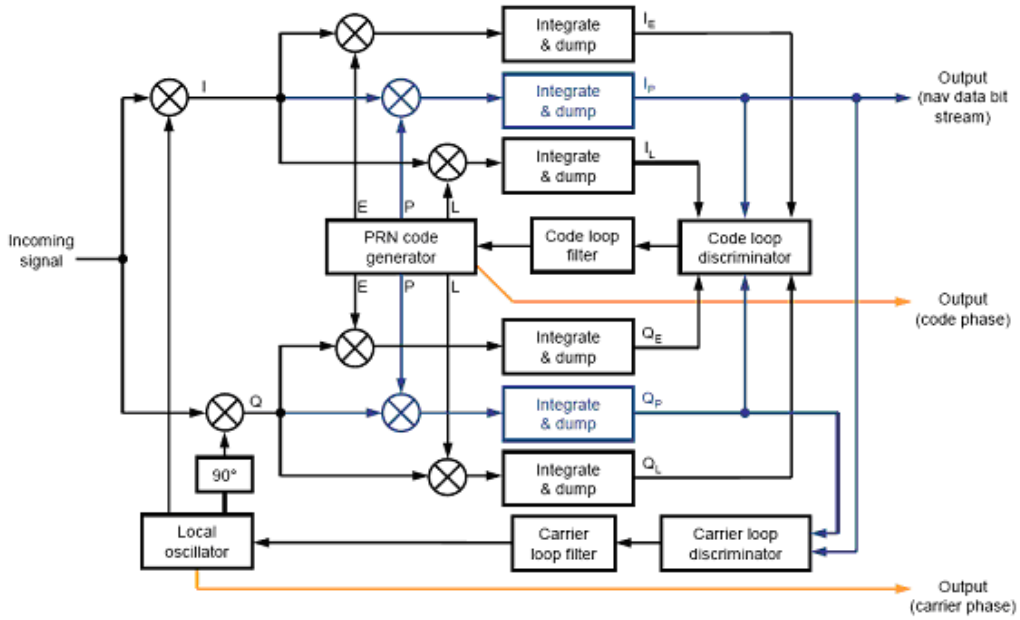

Figure 7: Satellite tracking loop[2]

5.2 ADC Input System

5.2.1 Serial Input Buffering

The input to the BRC processor begins at the output of the RF front-end, which is an analog to digital converter (ADC). The functionality of the fully custom designed front-end can be largely abstracted for the purposes of the digital core. The important points are that the front-end will define an intermediate frequency (IF) on which the GPS signal will be modulated. In addition, the ADC will quantize the input signal to a single bit (two quantization levels) and sample at a frequency defined by the required functionality of the BRC Doppler removal system. The IF is approximately 1.4MHz and the sampling frequency is 8.192MHz.

After crossing into the microprocessor, the output bit stream from the ADC will enter a chain of shift registers. This shift register will be eight bits long, corresponding to the size of a byte (the smallest addressable data length) in the BRC architecture. Each of the eight registers will also have a corresponding valid bit to indicate whether a given bit of data has been consumed by the processor. The structure of these registers and the corresponding logic is illustrated in Figure 8.

For reasons relating to matching bit rates at various possible Doppler shift frequencies, the valid bits can be reset corresponding to one of three options: first, all bits are set to zero indicating that all eight bits have been consumed by the processor and one bit will be skipped; second, the first valid bit (corresponding to data connected directly to the ADC output) remains one while all others are set to zero indicating that all bits have been used and the register should continue normally; third, the second two valid bits remain high indicating that only seven of the bits are actually being used and the last bit should be repeated. For simplicity, all three options will return the full eight-bit word from the shift register.

The goal of using short or long reads is to be able to modulate the carrier frequency back to the interme-

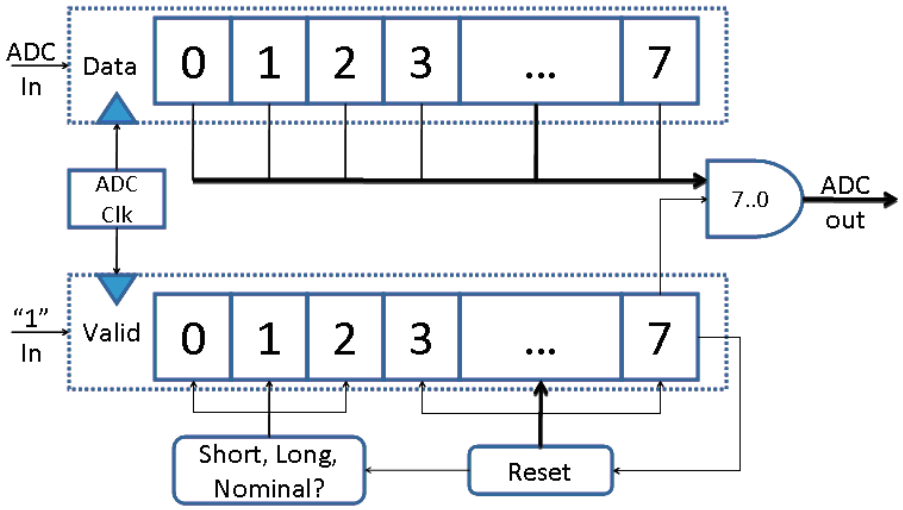

Figure 8: ADC input shift register structure

diate frequency in hardware. These reads serve as small corrections and their frequency will be a function of the Doppler shift. In figure 9, frequency signals with positive and negative Doppler shifts are matched to the local frequency copy using this technique.

Notice from the graphs in figure 9 that the short read has the effect of repeating an input bit while the long read skips one. Although this results in a discontinuity, the aberration is small. Also, the Doppler shift in these figures is large relative to the local frequency, resulting in a correction every second word. In the real GPS system the Doppler shift is a much smaller percentage of the nominal carrier frequency and the corrections will be much less frequent.

To use such a bit correction technique there must be a closed form equation relating the Doppler frequency to the number of words per correction. The equation below describes the error time per bit as a function of the sample period, intermediate frequency, and Doppler shifted carrier frequency.

$$\delta\tau = T_{sample}\left(1 - \frac{f_{IF}}{f_{IF} + abs(f_{dop})}\right) \tag{2}$$

Using this equation, the number of words per correction W is defined in terms of the time at which the accumulated error will be equal to one sample period. In the equation for the number of ADC input words per correction, W, the quantity b is defined as the number of bits per word that that ADC input buffer will write to the input data memories at a given time.

$$W = \frac{T_{sample}}{b * \delta\tau} \tag{3}$$

With these equations for corrections as a function of Doppler frequency, software algorithms can compute the number of unmodified words between each correction required to track a satellite at a given frequency. Having computed this value, the BRC architecture uses a simple state machine to continually count down towards the next correction. Figure 10 illustrates the implementation of this system.

In the Doppler control state machine, the highest-order valid bit from the input shift registers serves

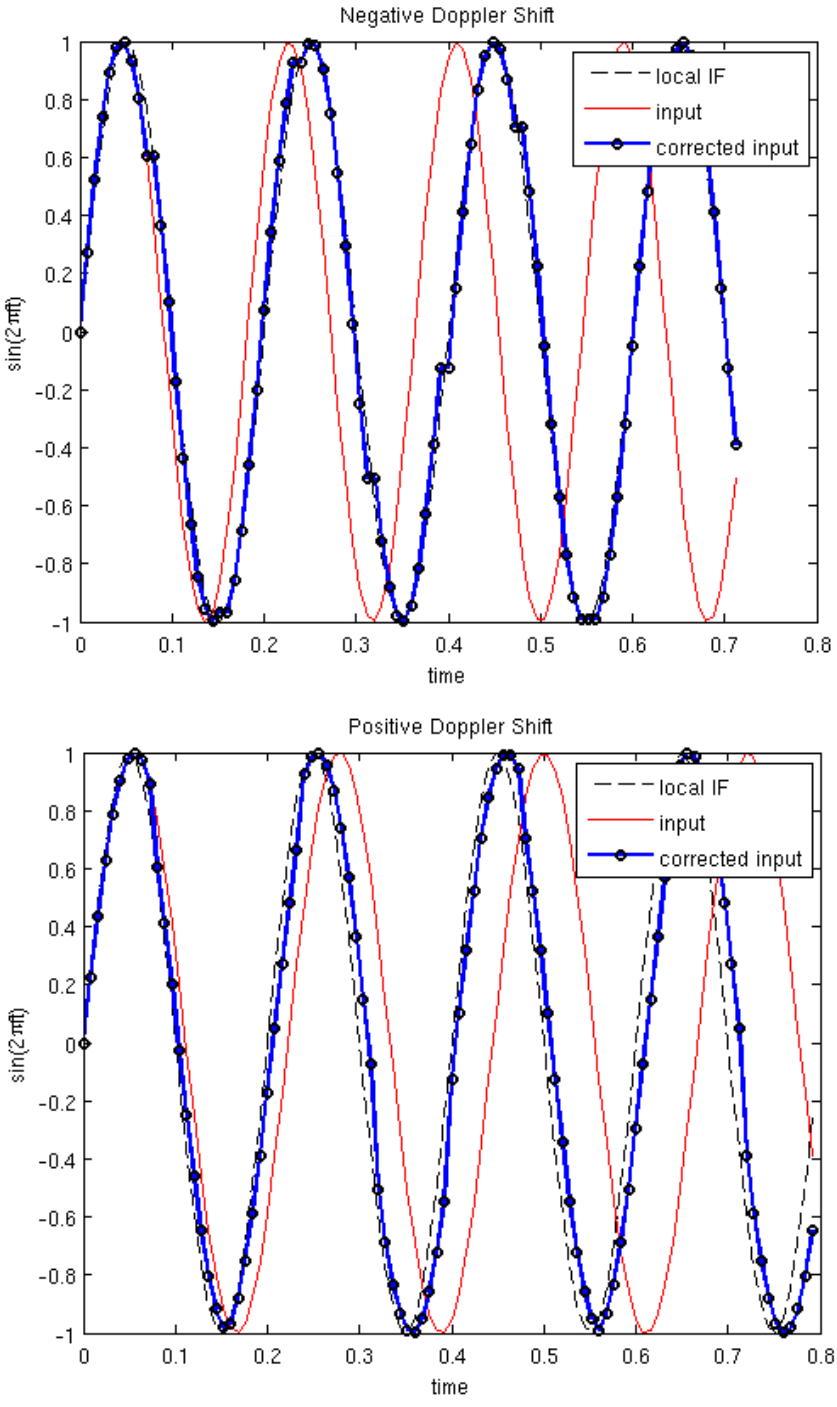

Figure 9: Inputs with negative (above) and positive (below) Doppler shifts are modulated to match the local IF copy

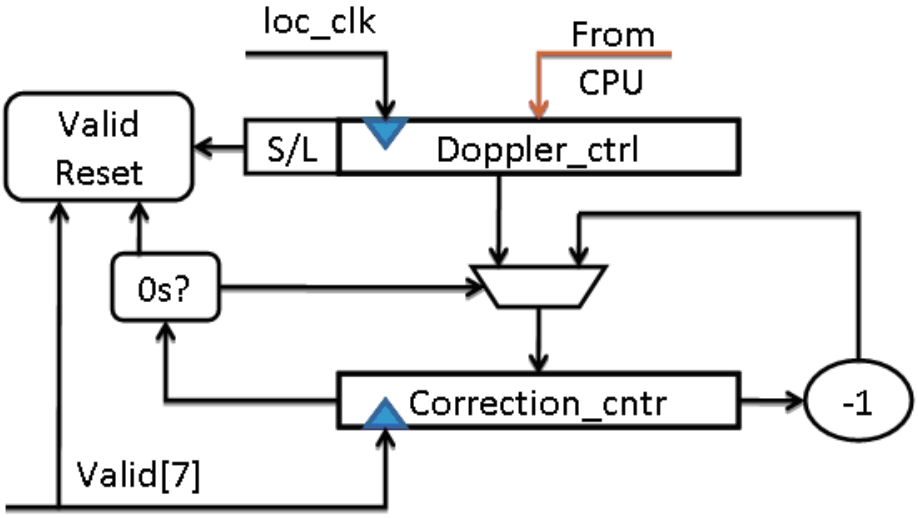

Figure 10: ADC frequency correction state machine

Instruction	Function
`ADCCNT rs, rt`	$\texttt{Rcnt}[\texttt{rt}] \leftarrow \texttt{rs}[11:0]$

Table 5: Format for instruction to set Doppler counter registers

as the enable for the counter update. The CPU uses a special instruction to set the initial counter value; the sign of this counter value will indicate whether the coming correction is a short or long read. Also, to enable tracking of multiple satellites with varying Doppler shifts, the above state machine is replicated for each satellite that the system is tracking. The format for the instruction used to set the correction counter initialization registers (Doppler_ctrl in the Figure 10) follows in Table 5.

Together, the input shift registers and the Doppler control state machine form the system which controls writes from the ADC input to a satellite's respective input memory. The system is designed such that the frequency of writes can be controlled through the software algorithm with a minimal instructional overhead. To complete the DMA system this write enable logic must be input to a system which can automatically control the location of reads and writes. The top-level of the ADC system controls this functionality, as described in the following section.

5.2.2 Direct Memory Access

With the Doppler normalization technique, the frequency at which the most significant valid bit is set high will automatically change proportionally to the Doppler shift described in its frequency correction register. Because this bit serves as the output enable, it is difficult to predict the exact times at which the output is ready for each satellite tracked by the system. Instead, the proposed BRC system implements direct memory access (DMA) into a reserved stack of memory for each satellite so that this process can be abstracted at the CPU level.

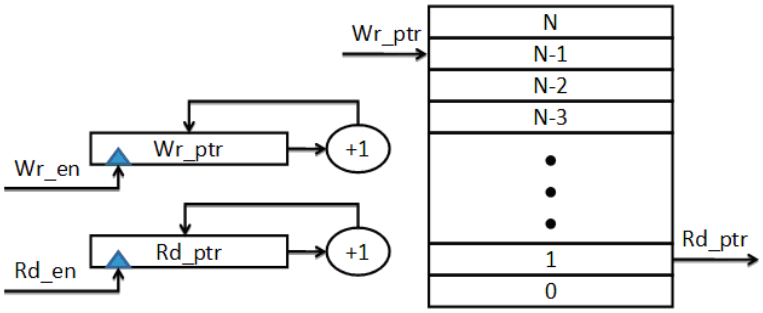

Figure 11: Structure of the ADC FIFOs

Instruction	Function
`ADCLD rd, rs`	`GPR[rd]` ← `FIFO[rs[7..0]]`

Table 6: Format for ADC FIFO load instruction

The reserved ADC memories are structured as pointer-based first-in first-out (FIFO) memory stacks. There are separate pointers indicating the location of the next write and read which point to the memory locations above the newest data or of the oldest data, respectively. Both of these pointers update automatically following a read or write. Figure 11 illustrates the structure of the DMA memory.

Because the input from the ADC is continuous, the rate of reads and writes for each FIFO memory must be equal on average. The size of the memory needs only to be large enough to provide buffering while the CPU executes control loops and are therefore relatively small. The first revision of the BRC architecture has 256 16-bit words per FIFO.

Each satellite that the BRC tracks has its own ADC memory. Since each satellite has a different Doppler shift, the data written into each FIFO will be similar, but shifted such that its carrier is modulated back to the intermediate frequency. Although this results in extra hardware, the software-level complexity of the correlation computations will be greatly reduced. Because these correlations are the primary driver for the minimum frequencies of operation, the area overhead for these memory structures should be more than paid back in frequency reduction.

In many DMA systems, the largest challenge is ensuring that there will be no conflicts between separate attempts to access the memory from both the CPU and ADC. The BRC system mitigates this risk by creating each ADC memory as a separate and discrete structure rather than as software-defined segments of main memory. These memories will be read-only with access through another special-purpose instruction. The format for this instruction is provided in Table 6.

The format of this instruction is very similar to the register-indirect system that the BRC architecture uses for standard loads. However, the important difference is that because the read pointer in each FIFO determines the position of the read within the stack, the source register in this instruction need only determine which ADC stack from which to retrieve data. From the point of view of the CPU, there is only a single memory location for data from each satellite.

Function Modifier	FIFO selector
15..8	7..0

Table 7: Interpretation of RT and RS in ADC instructions

5.2.3 Corner Cases

During typical operation, the write and read pointers should never be equal. The exceptions to this case occurs when the FIFO is empty or in the event of data loss. At initialization, the pointers are both set to point to the lowest entry in the array. In this case, there will be no available data until the first write has completed and the pointers are no longer the same. A similar situation will occur if the CPU has consumed data faster than the ADC writes and emptied the FIFO buffer, but in this case the pointer values will not necessarily be equal to zero. In this situation the ADC system will stall the CPU until the FIFO data becomes valid and the instruction can complete. This method has the advantage of creating an automatic mechanism by which the system frequency can be scaled down for power savings if the CPU runs ahead of schedule. At the other end of the spectrum, if our system cannot keep up with the rate of incoming data, the write pointer will wrap around until it passes the read pointer. At this point there has been data loss. Our system deals with this problem through prevention. In the event that data loss does occur, the algorithm should be robust enough to detect the failure and restart the acquisition process.

5.2.4 Special ADC Functions

When taking the GPS algorithm into account, it becomes clear that certain extra functions within the ADC system can significantly increase the reliability of the system and reduce total power. For reliability, we have included mechanisms to reset and read the values of the pointers. In addition, special instruction formats can enable or disable individual FIFO systems. These special instructions are variants on the two ADC-specific instructions `adccnt` and `adcld` which utilize the fact that the maximum values for the RS and RT fields is significantly smaller than one byte, meaning that the top byte could be used to encode additional information. The architecture uses the `adccnt` instruction for any special instruction which alters the state of the ADC and the `adcld` instruction to read out any state. Table 7 indicates the byte-wise interpretation of RS and RT for ADC instructions.

During the acquisition phase, if no satellite can be found at a given frequency the processor updates the Doppler and continues to search. However, because the Doppler shift is effectively encoded in the input signal as it is stored in the FIFO, significant changes to the Doppler frequency during acquisition will render any data currently in that FIFO useless. To avoid loss in signal strength due to old data at an incorrect frequency and to minimize the possibility of filling the FIFO during acquisition, the BRC architecture includes a variant on the `adccnt` instruction that effectively clears a given FIFO. To avoid any timing issues with the write pointer (which must be clocked based on the ADC sample clock) the read and write pointers are not set to zero; instead, the read pointer is simply updated with whatever value is currently in the write pointer.

When the system tracks multiple satellites with different Doppler frequencies, the input buffer system stores data corresponding to FIFO registers at different rates. Because a read stalls if there is no available data, a naïve implementation could become rate-limited by the signal with the most negative Doppler frequency. To avoid this problem we have included functionality to read the state of the pointers so that the

Instruction	Modifier	Function
ADCCNT	8'h00	Rcnt[rs(7..0)] ← rs(11..0)
	8'h01	RdPtr[rt(7..0)] ← WrPtr[rt(7..0)]
	8'h02	adc$_{EN}$[rt(7..0)] ← ENABLE
	8'h03	adc$_{EN}$[rt(7..0)] ← DISABLE
ADCLD	8'h00	GPR[rd] ← FIFO[rs(7..0)]
	8'h01	GPR[rd] ← WrPtr[rs(7..0)]
	8'h01	GPR[rd] ← RdPtr[rs(7..0)]

Table 8: Summary of ADC instructions and modifiers

Unit Selection	Phase
15..8	7..0

Table 9: Interpretation of RS in GC instructions

system can monitor the fill of each FIFO. Separate modifiers allow for the system to load the read pointer and write pointer values into a CPU-level register.

The last set of functional modifiers allows the system to enable or disable individual ADC sub-systems to manage power dissipation. While the current design only includes hardware to track the minimum of four satellites, future designs may provide support for additional tracking to produce a more accurate navigation solution. In this case, providing a software mechanism to disable and enable individual ADC functional units will allow the GPS algorithm to trade power for accuracy.

In table 8 we have summarized the assembly-level ADC instructions along with the functional modifiers.

5.3 Gold Code Generation

The GPS system uses a code-division multiple-access (CDMA) protocol to enable signal transmissions from multiple satellites to occupy the same frequency space. The codes used for consumer GPS are Gold codes have a length of 1023 chips (each bit is referred to as a "chip" because it does not carry any information on its own). Because our design is power and area constrained, storing the codes for each satellite in memory is not an option. Instead, the BRC design includes four Gold code generation units that can each produce the code for an individual satellite in real time. Although these generators can track only one satellite at a time, each is capable of producing any Gold code. The diagram in Figure 12 describes the structure of an individual gold code generator.

Each gold code generator stores its current phase selector bits. If the requested phase selection is different from that stored, then a new Gold code has been requested and the shift registers will be reset to all ones (the initialization for all GPS PRN codes). If the phase selection matches the current state the shift registers will advance by one cycle and the returned value will be the next value in its sequence.

The instruction used to access the Gold code generators must provide a way to indicate the phase selection and which state machine to access. To provide maximum flexibility and efficiency the state machine and phase selection is encoded in a single value and stored in a source register. The encoding of that register is described in Table 9.

Using one byte each for the two fields allows for easy loading of values into the fields and provides the system with room to expand to a much larger number of possible codes. In the first revision of the BRC

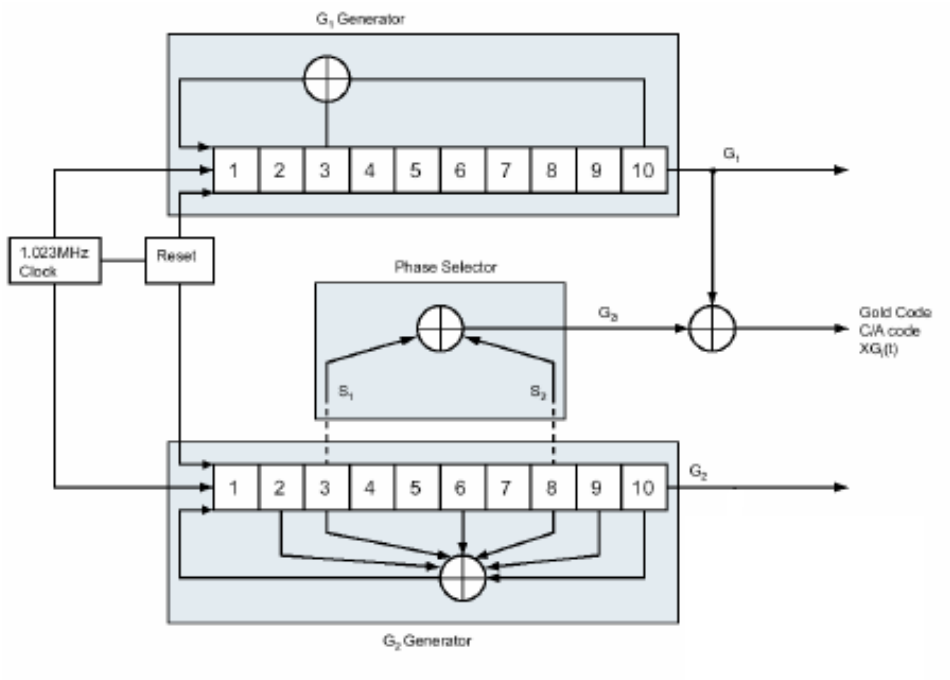

Figure 12: Structure of the Gold Code generation shift registers

Instruction	Function
`GC rd, rs`	`GPR[rd]` $\leftarrow$ `16gc[rs]`

Table 10: Format for ADC FIFO load instruction

we only attempt to track the minimum number of satellites, so there are only four code generation units to address. A code generation instruction returns only a single chip of the code. Each bit of the destination register is then filled with that value. The format of the instruction and the result appears below in Table 10.

5.4 System Specifications

Because the microprocessor must minimally be able to perform the necessary correlation calculations described in the flow chart in Figure 7 for each satellite that it is tracking, the following equation describes an reasonable approximation of the required system clock frequency.

$$f_{clk} \geq N * S * \left(\frac{f_{chip}}{floor(16/b)} \right) \tag{4}$$

In the above equation N represents the number of instructions per correlation computation for early and late C/A codes both in-phase and quadrature, S represents the number of satellites currently being tracked, b is the number of bits per word, and f_{clk} is the minimum possible operating frequency for the microprocessor. Assuming that S will be four (the minimum number of satellites necessary to produce a navigation solution) and the number of bits per word will be eight, the clock frequency must be greater than 2MHz multiplied by the number of instructions necessary to perform a correlation.

The number of instructions per correlation therefore becomes the most significant criterion in defining the clock frequency and power output of the microprocessor. Preliminary assembly code written to perform the correlations in the BRC system show that each iteration requires 35 operations. Using equation (4), 70MHz is the resulting estimate for the required clock frequency. However, since this does not take the control overhead into consideration, the BRC team targeted a clock frequency of 100MHz. When testing the final implementation the ultimate goal is to see a correlation plot similar to that displayed in Figure 6. Even in the event that the final system is not capable of tracking four satellites simultaneously, seeing such a correlation spike for any signal will be an indication of success.

6 Results

"I love deadlines. I like the whooshing sound they make when they fly by."
— Douglas Adams

6.1 GPS Algorithm

Achieving the goal of a low-power GPS system based on a software baseband receiver required several unique hardware systems. Of these, the system by which our ADC input system automatically removes the Doppler frequency shift as inputs are buffered in memory is particularly novel. Because these designs are based on new ideas they required testing to ensure that the theoretical system would work in reality. Unfortunately, testing for these systems at the hardware level is difficult because the design depends on the specifications of the custom RF front-end which has not yet been implemented. Instead, the BRC GPS system was verified at the algorithmic level using software simulation in MatLab. The MatLab platform allowed the team to create simulated inputs at the proper intermediate frequency and sampling rate. Using these inputs we developed a software algorithm that could be easily ported to a BRC compilable C format. In addition, the software system simulated the functionality of our special-purpose GPS functional units to ensure that the Gold Code system could be used in conjunction with the ADC Doppler removal state machine to allow highly efficient correlation computations.

Although we received initial assistance from the GPS research group in developing our software algorithms, ultimately we were forced to develop our own MatLab system because our algorithm needed to incorporate a structure capable of real-time tracking to accurately model the BRC system and allow for easy code conversion. The real-time algorithm that we developed using MatLab automatically time-shares the processor resources by continuously cycling through each of the satellites that system is currently tracking or attempting to acquire. When a given signal has been correlated against a full iteration of its pseudo-random noise (PRN) code the algorithm runs an iteration of the control loops. The control will be an iteration of the tracking loop if the signal has been previously acquired, otherwise the system check for acquisition. The pseudo-code displayed in figure 13 represents the basic structure of the BRC real-time GPS system.

The MatLab system mimics our proposed hardware system as closely as possible. The code uses a counter value similar to the Doppler correction register to determine the frequency of single-bit corrections and the Gold Code is never resampled. All correlation computations are simply a vector sum of a bit-wise multiply operation, which is a direct analog of the XOR-POPC instructions that compute correlations in hardware. After some learning - most notably discovering that the value of the Doppler counter must be set by rounding the computed value the the next higher integer - the algorithm was successfully able to acquire and track signals generated to represent those found an incoming GPS bit-stream.

Acquisition of signals typically occurs at approximately 150kHz Doppler shift away from the precise value. Following acquisition, the tracking algorithm is capable of pulling in towards the exact Doppler shift in successive iterations. Since our technique for Doppler shift correction requires more corrections at larger magnitude frequency shifts, the possibility of decreased signal correlation at the ends of the spectrum was a significant concern prior to algorithmic simulations. However, simulations showed very little loss at all Doppler frequencies so this is no longer expected to be an issue. In figure 14 we have shown the correlation output of our GPS algorithm at different code shifts (the graph is generated by setting the acquisition

```
INITIALIZE;
always
  foreach( satellite )
    //Repeat for in-phase and quadrature on
    // early, prompt, and late goldCodes
    correlation = goldCode ^ signalIn ^ localIF
    if( chipNum[satellite] == codeLength )
      if ( acquired[satellite] )
        tracking_iteration( satellite )
      else
        acquisition_iteration( satellite )
      end
    end
  end
end
```

Figure 13: Pseudo code detailing the real-time GPS algorithm for use on the BRC system

threshold impossibly high - else the algorithm would not have continued to change the code shift). Note that the ascent towards the correlation peak is linear and that magnitude is near the maximum possible value of 8192.

Having obtained results indicating that our system architecture was valid we began the process of converting the MatLab algorithm to C, which could be compiled into BRC assembly language by our custom port of the GCC compiler. For this code, the control loops are written using standard C constructs. The correlations, however, must be implemented using the special instructions implemented for this revision of the BRC architecture. Because the compiler does not support these instruction the correlation portions of the final GPS code are written in in-line assembly. Because these computations dominate the performance of the algorithm this portion of code would likely have been written in assembly regardless.

Although the C code necessary for running GPS on the BRC processor has been written, it remains untested. Finalizing this code and testing it on the fabricated core will be one of the largest tasks left to future BRC teams.

6.2 Timing Fixes

After writing and fixing our RTL for functional correctness, we began to synthesize it for timing. The design failed to meet the timing goals we gave it, which was our target frequency goal of 100MHz. DesignVision reports the timing paths and how much they missed or made timing by. By reviewing failed paths, they can either be marked as false paths if they are guaranteed not to occur or design changes can be made in order to fix the critical paths. This section describes the design changes that needed to be made in order for the critical paths to pass timing.

The first time fix involved the path from write back through the register file to the branch compare unit. The register file was updated on the negative edge of the clock; if a branch was in RD and needed a value currently being written back, it would only have a half cycle to compare and compute the new PC. This was not enough time. However, the write back logic is actually faster than half a clock cycle, so it arrives at the RD early. By allowing the RD to use this information early, we could eliminate this path. By having the

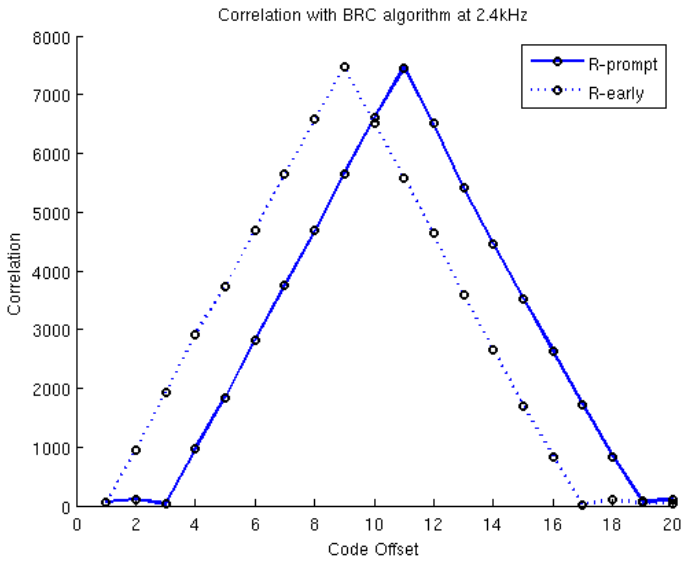

Figure 14: Correlation at various code shift offsets generated using the BRC real-time algorithm

RD check if the current input is being written to the register the branch wants to check, it can calculate the correct value before it is actually written back to the register file.

The next fix relied heavily on our the bypass path we created around the registers that we created. This problem was in the first half of the cycle. The stall signal from the ADC could not reach the register file in half a cycle. After our previous fix, the only timing path after the half cycle for the register file was the decoded value directly to the pipeline register. As there was no logic to be performed, we decided it would be possible to shift the register file write to later in the cycle, in order to give the stall signal more time. In order to do this we generated a quarter phase clock in the clock state machine. The stall signal now had 75% of a cycle instead of only a half cycle, which was enough for it to make timing.

The last major timing fix involved the gold code and ADC units. When they consumed a value produced by the immediately preceding instruction, the value would be bypassed to them. However, since these units use the negative edge of the clock internally, this meant that there was now a path through write back, through the bypass logic, and into the gold code or ADC unit in only half a cycle. Since this path failed timing, we decided to not allow data forwarding to these two units. Instead, we stalled any gold code or ADC instruction that had a data dependency on the previous instruction. These instructions will normally be used in assembly code where it should be possible to write code that will avoid having to stall, making this trade off worth while.

6.3 Synthesis

Our goal to produce full GDSII in the IBM 130nm process was successful. Synthesis of the Big Red Chip processor successfully compiled with no DRC or hold violations, and is expected to be able to run at 50MHz. Our original goals were to achieve a 100MHz frequency to be able to track multiple satellites, but the Encounter backend reported unfixable timing paths that will force the chip to run at a slower frequency.

Results from the gate level compilation in DesignVision suggested that our timing goals would be met, but these were made with only rough wire model estimates.

Real results from the back end include:

Standard Cells, Macros, IO Pads Placed As shown in Figure 15, all modules were successfully placed in a logical fashion. The memory systems are close to one another, but still have some spacing to allow routing near them.

Power Grid Synthesized The power grid is shown in Figure 17, displaying not only the power ring around the core and stripes to distribute this evenly over the core, but the standard cell power lines as well.

IR Drop acceptable Figure 18 confirms that our IR drop is acceptable: all cells indicate that there is less then a .1V drop to any standard cell in the design.

Clock Synthesized The clock distribution network, as seen in Figure 16 is split fairly evenly over the chip, and although automatically distributed, a natural H-tree configuration is readily apparent.

Design Routed The final picture of the chip, routed and with all DRC fixes is shown in Figure 19. There are no overly congested wire paths, but wire distance along with aggressive cycle times with the quarter clock do not allow for the desired timing to be met.

DRC Clean All geometry violations, process antenna violations, and metal density violations were corrected either automatically using the Encounter tool or by custom routing.

Hold Timing Clean Hold timing violations, which affect the functionality of the processor regardless of the clock speed, were corrected by inserting buffer or delay cells while minimizing their effect on setup timing. There were no hold timing violations in the final GSDII.

Setup Timing Setup timing was not fully fixed. Several paths were declared as false, but until the final chip returns the fastest that the processor will predictably run at is 50Mhz.

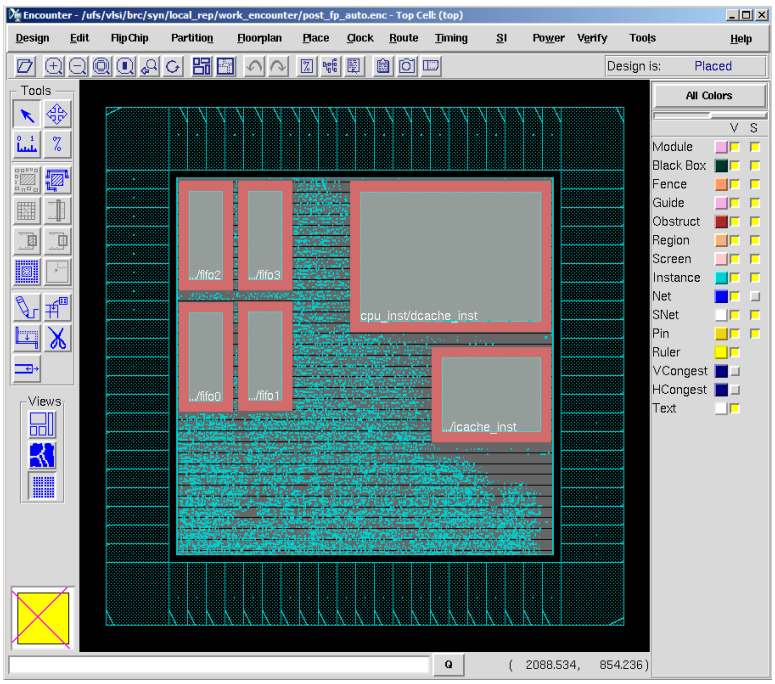

Figure 15: Standard Cells, Macros, IO Pads placed

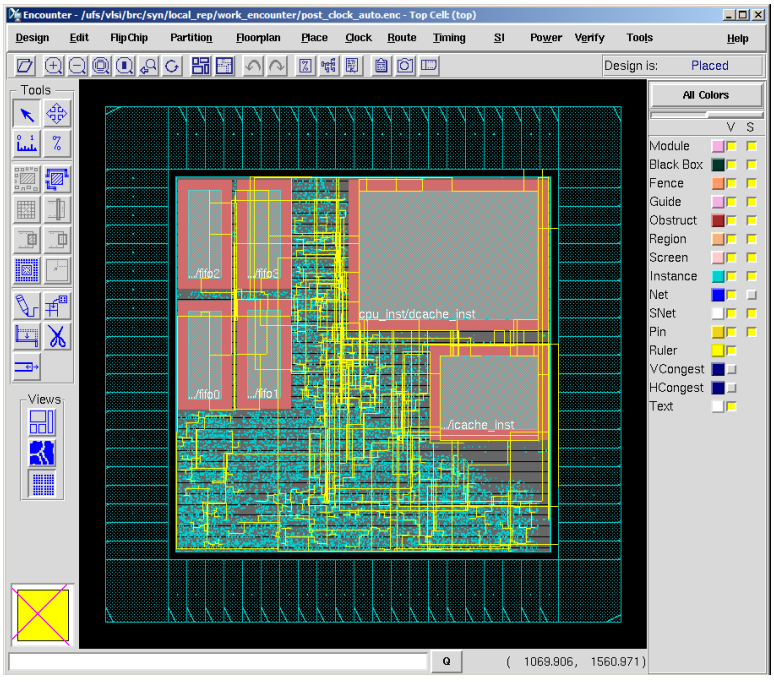

Figure 16: Clock Tree Synthesized

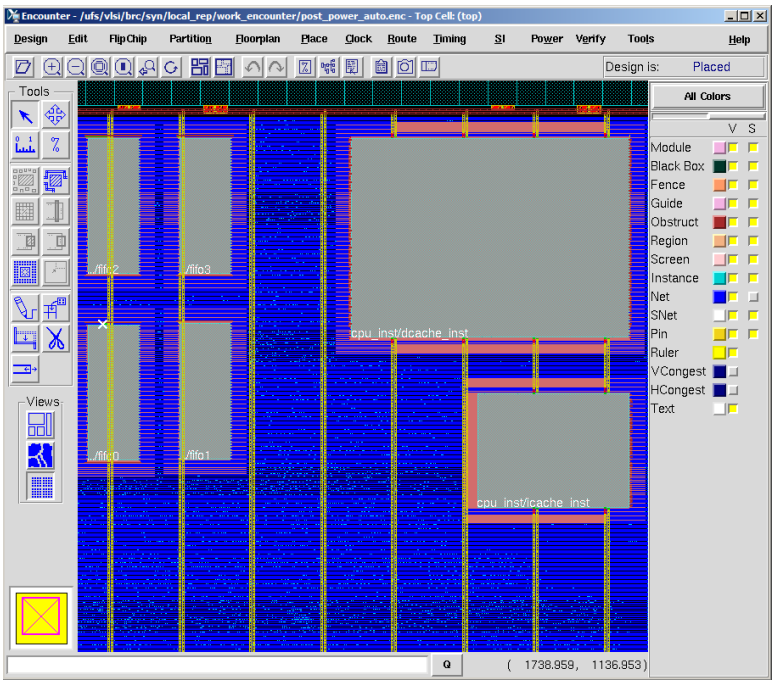

Figure 17: Chip with Power Grid

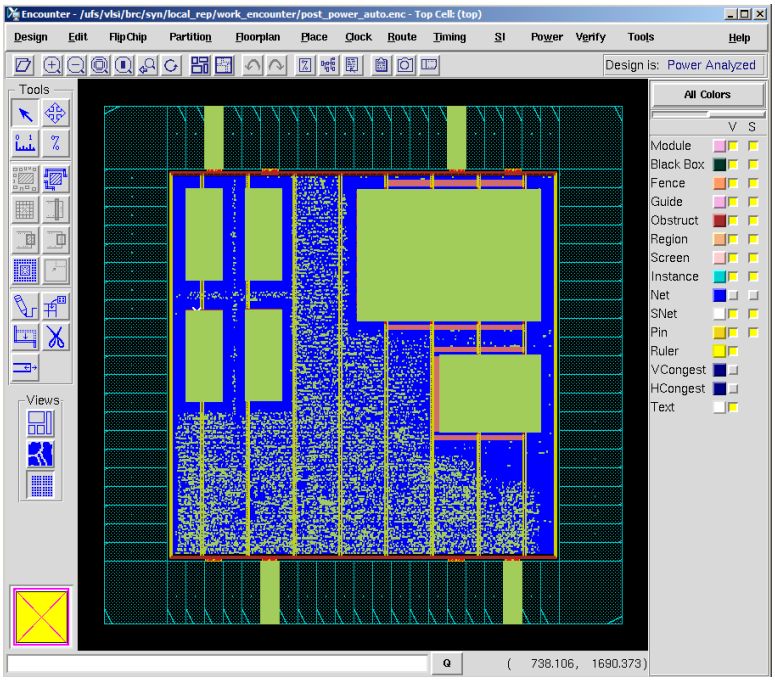

Figure 18: IR Drop across the chip

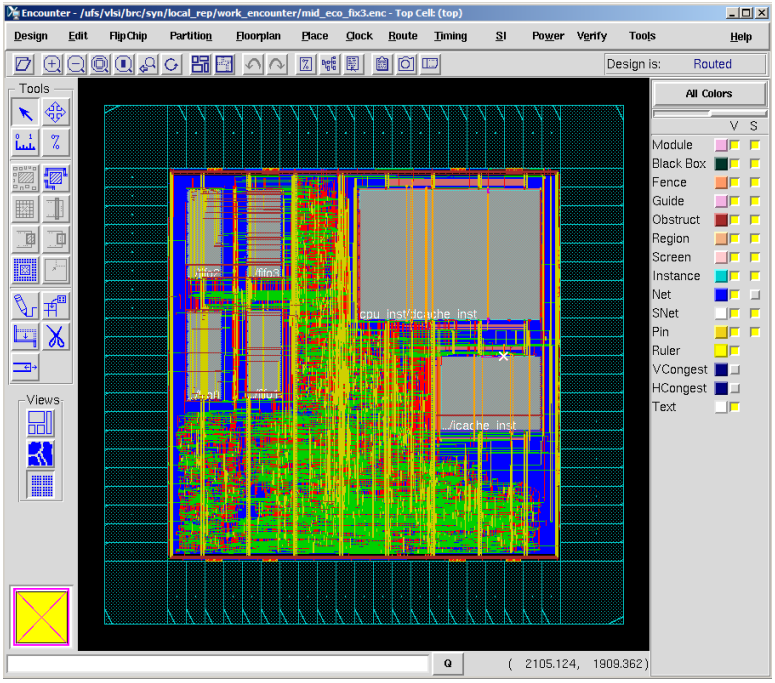

Figure 19: Chip, Fully Routed, Final Synthesis

7 Future Work

"As for the Future, your task is not to foresee, but to enable it."
— Antoine de Saint-Exupery

7.1 Post-Silicon Verification

As stated in the verification methodology section, there remains a significant amount of work to be done on the chip once it comes back from fabrication. This will be the biggest priority of next year's team, as we hope to deliver a working chip to another project group as soon as possible. The verification effort will require implementing an FPGA-based system to load/read the memories and read/write the scan chains. Ideally, it could also enable dynamic debugging, allowing the user to see the state of the system at all times as the chip steps through the program. This project would probably require some knowledge of scan chain fault analysis as well (stuck-at faults, etc). In addition to the FPGA-based system, we will also require a test suite that will be able to test for register to register faults within the processor. This will require the team to write a test case, halt the processor at a fault, know the expected values in the scan and compare them to actual values. A script may need to be written to generate the expected values in the scan chain as instructions are run through the core.

Another area that requires attention is working toward complete coverage of the design. On our current design, we achieved approximately 85% line coverage, with much smaller coverage percentages on all other metrics. This was due, in part, to different top level test benches whose coverage metrics could not be combined, as well as not having the time to write directed test cases to increase coverage at the end of the year. While we are confident that our processor will be shipping without functional errors due to our random test cases, without coverage metrics to confirm this, we cannot be completely sure. Therefore, on the next design iteration, we feel that a strong emphasis should be placed on verifying the chip such that we can have close to 100% coverage in all metrics.

7.2 Architecture

In the future, an expansion of the existing ALU operations to include instructions which were not fully implemented in this iteration of the processor would be excellent functionality to add into BRC. These instructions are currently reserved and unused and include the more complicated `mult` and `div`, along with corresponding `multu`, `divu`, `mfhi` and `mflo`. Currently, `mfhi` and `mflo` along with `HI` and `LO` registers are not implemented, and `mult` and `div` are unrolled to shifts and adds in software. A module that includes a hardware multiply and divide would provide significant time-savings in arithmetic-intensive programs.

Additionally, one of the largest drawbacks we suffer from in a 16-bit architecture is the limitation to only eight registers. It would be useful to increase this number: even sixteen registers would afford a great deal more flexibility. One solution to this would be to have multiple sets of eight registers and be able to switch between them at the compiler level using interrupts or context switching. A second solution, though requiring more fundamental changes, may be to amend the ISA to include instructions with four or five-bit register fields, allowing an instruction to access 16 or 32 registers instead of the current eight.

Building off of this last idea, interrupts are not implemented in the current processor, but having that functionality would allow for much more flexibility. Having interrupts implemented could let BRC evolve

from a simple embedded processor to one that is both low-power and general purpose. Whether it is a simple timer-based interrupt or a fully asynchronous and driven from outside the core, this added functionality would make our core far more robust.

Finally, from an architectural standpoint, more aggressive efforts can be made to reduce power. While the overall, high-level design choices were made to conserve as much power as possible, there is still a good deal of power optimization at the implementation level, especially when considering enable logic and clock gating.

7.3 Circuit Design

Another task that remains for future generations of the Big Red Chip is to fully integrate the entire GPS system onto the chip. Right now, the ADC module only processes the data coming in from an off-chip ADC, it does not actually include one of its own. Designing a custom ADC in whichever process we are working in, to be included on-chip is task that remains unfulfilled.

Additionally, the possibility of migrating to a smaller technology size, such as 90nm from our current 130nm, is intriguing. This would require designing and characterizing a new, complete standard cell library as well as the memories for the caches. If this is a task that future teams choose to take on, there would need to be a group of experienced people working on it.

7.4 Software

Finally, in addition to the work that needs to be done in hardware, the software could also be optimized. The GCC compiler has been completed, but the focus has been purely on making it functional as quickly as possible in order to begin testing, instead of on optimizations that could have been done. Similarly, all of the time-critical and ADC-related portions of the embedded code are written in assembly, but could also be further optimized. Both of these tasks are important, because as the instruction count of the program goes down, more time can be spent tracking additional satellites.

8 DesignVision User Guide

"We shall not fail or falter; we shall not weaken or tire. Neither the sudden shock of battle nor the long-drawn trials of vigilance and exertion will wear us down. Give us the tools and we will finish the job. "
— Sir Winston Churchill

8.1 Before You Begin

Synopsys DesignVision is a software tool to convert high level Verilog RTL to gate level equivalent. The tool can handle many design constraints, from timing restrictions to power constraints to wire load models. Before starting, consider reading any relevant sections in the Encounter section, including if the design will have power or IO pads, as these will require changes to Verilog. The following files are necessary, and will either be provided by the designer's Verilog, technology vendor or from a memory generators:

- High Level Verilog RTL
- Technology Libraries (.db)
  - Timing Libraries for both standard cells, macros
  - Wiring models (Data may be contained in timing libraries)
- Symbol Libraries (.sdb)

8.2 Finding Help

DesignVision provides two primary sources of help for the user. The GUI window has a link to "Online Help," providing a complete GUI option. For more experienced users, or for anyone unsatisfied by the GUI support, help can be obtained from the command prompt by issuing `help` to find a listing of commands, and `man COMMAND` to find out more about a particular command. Any command will be recorded in command.log, including GUI commands, so this file can be referenced to find the command prompt equivalent to any GUI operation.

8.3 Setup

8.3.1 Setup Libraries

After starting DesignVision, the first task the user must accomplish is to define the technology libraries, as described in section 8.1. The target libraries and link libraries are the same for the purposes of this document, and the designer should use the same .db files for both. If only the .lib is provided, DesignVision can convert this file to .db using the following steps:

- From the DesignVision terminal:
- `read_lib LIBRARY.lib`

- `write_lib -format db USERLIB`
  NOTE: After using the read_lib command, DesignVision will output some text saying it was successful. In this text it will say what the internal name of the library that it read it is (usually matches the name of the .lib, but not alway). Use this name as USERLIB.

Once all .db files are prepared, they can be read in using the following method:

- GUI:
  - Menu:File-Setup
  - Click "..." next to Link Libraries
  - Remove previously defined libraries (.db)
  - Add user's libraries for timing, wire models, etc. All models must be included here. Exit.
  - Repeat the steps for Link Libraries for the Target Libraries.
  - Add user's Symbol libraries (.sdb) in the same fashion.
- Command Prompt:
  - `set link_library { LIBRARIES.DB }`
  - `set target_library { LIBRARIES.DB }`
  - `set symbol_library { SYMBOLS.SDB }`

For more advanced users, a .Synopsys_dc.setup file can be included in the run directory with the terminal commands listed above. By default, dv reads one from the install directory if none is defined. We chose to modify the install directory file, but if multiple users share the same install with differing technologies, personal setup files are recommended.

8.3.2 Importing Design

When the libraries are setup the user is then ready to import the design. If the Verilog contains hierarchy (likely), then they are recommended to use the Analyze, Elaborate flow; if there is no hierarchy in the design then the user may use the GUI Menu-Read command directly. User's with hierarchy in their design must remove any `include *.v` in their Verilog that can instantiate a module, as this will cause multiple instantiations at this step and are likely to cause show stopping errors.

In order to run timing analysis, the wire load must be defined as well, combined in the steps below. There is likely to be a choice of wire load models, ranging from most optimistic to most pessimistic. Refer to the .lib technology file that the .db was generated from to see the exact RC estimates the tool will use. Choosing pessimistic wire load models will cause the timing to skew towards fixing setup timing, while choosing optimistic timing will cause the timing to do a better job fixing hold timing. These wire models are only estimations of what the back-end Encounter tool will actually calculate as the RC delays, but wisely choosing early on can save headaches later. Wire load models may be later extracted from Encounter to give a more realistic estimate.

- GUI:

- Menu:File-Analyze
- Add all Verilog files, including all levels of hierarchy.
- Note the Work Library, this will be necessary for Elaborate
- Menu:File-Elaborate
- Choose the Work Library defined in the Analyze step
- Choose the top level module in the Design drop down selection, click OK
- Menu:Attributes-Operating Environment-Wire Load
- Choose the appropriate wire model (they are listed from most optimistic to most pessimistic)

- Command Prompt:
  - `analyze -library WORK -format Verilog { VERILOG.v }`
    NOTE: WORK can be any name here, but keep it consistent with the elaborate step
  - `elaborate TOP_MODULE -architecture Verilog -library WORK`
    NOTE: TOP_MODULE is the name of the user's top level module
  - `set_wire_load_model -name NAME -library LIBRARY`

8.3.3 Clocks & False Paths

To run timing based compilation, the clocks must be defined so that the tool knows which cycle time to optimize to. Aggressive designs may choose to define a lower period so that even if DesignVision cannot compile to a clock cycle, it may be stretched so that it increases iteration and effort. There are two types of clocks: port level clocks and derived clocks. Many designs will not have derived clocks, clocks that are the result of clock gating and must go through combinational logic, but the steps to address them are included as well.

- GUI
  - In the GUI, a logical hierarchy window will have appeared. This window is a design browser, and for now find the top most module and click on it.
  - Next to the tree structure of hierarchy is another frame with a drop down menu. Select pins/ports.
  - Select the clock net with the mouse.
  - Menu:Attributes-Specify Clock
  - In the window that pops up, name the clock, define a period (in nS) and two edges.
  - Repeat for all clocks in the design.

- Command Prompt:
  - `create_clock -name "CLOCK_NAME" -period PERIOD -waveform { EDGES } { CLOCK_PIN }`
    NOTE: This command will match a CLOCK_PIN to a clock with a PERIOD and two EDGES (rising and falling), this is considered a "root" clock
    EX: create_clock -name myclock -period 10 -waveform {0 5} { clkPIN }

- `create_generated_clock -name "CLOCK_NAME" { GEN_CLOCK_PIN }`
  `-source CLOCK_PIN -divide_by NUMBER`
  NOTE: This command will match a GEN_CLOCK_PIN to a generated clock, whose source is CLOCK_PIN. The generated clock is slower by NUMBER in this case. Experiment with this command for the desired value - we suggest using the most pessimistic period the clock will see for timing compilation. Additionally, we don't believe that this is possible with the GUI.

While the clock provides a restrictions on the design to meet timing, not all paths analyzed are truly critical paths. If the designer has defined the ISA such that a particular scenario will never occur, but could still possibly happen in the Verilog, this is considered a false path. Even just the designers understanding of the Verilog may justify such false paths - impossible race conditions. With each false path, the designer must think very critically on whether they might mask a true timing path and take a similar level of precaution as a surgeon might. If the designer knows there are false paths in the design, however, they can remove these from consideration by taking the following steps: (NOTE that this may require at least one compilation of the design for the tool to recognize the paths)

- GUI
  - See the Clock section above on how to select a pin or port using the GUI.
  - Select up to two pins that are on the critical path: the D-Q clock on the path and the destination register D pin. Use the CTRL or SHIFT buttons to select multiple pins. If the pins are in different blocks, a new hierarchy window can be opened with Menu:Hierarchy-New Logical Hierarchy View.
  - Menu:Attributes-Optimization Constraints-Timing Constraints
  - In the pop-up box the nets selected will appear divided into two sections. Arrange the nets into the two sections appropriately using the GUI buttons. The Reset Paths check box can be checked here if overwriting a previous false paths.
  - For multiple false paths in the design, the previous steps can be repeated.
- Command Prompt:
  - `set_false_path -setup -reset_path -to { REG_INPUT_PIN }`
    `-through { NET } -from { CLOCK_PIN }`
    NOTE: -to, -through, and -from restrict the possible paths, but a user can define anywhere from one restriction to all three restriction. We commonly used two restrictions.
    NOTE: A false path can be a false setup and / or hold path. Replace -setup with -hold for hold paths.
    NOTE: This command will accept wild cards * if there is regularity with the false paths (such as an array of flops)

8.4 Compilation

At this point, the designer is almost ready to compile the design. Before this can be done however, the designer is encouraged to run the Check Design script under Menu:Design-Check Design, analyze all warnings

and fix all errors or violations. These errors and violations will appear in the terminal, with active links to help debug. Checking the design may dictate design changes in the Verilog, as the synthesis tool is more restrictive of what it'll accept then the normal Verilog compiler. Once all synthesis errors are fixed, warnings understood, and setup complete, compilation may be accomplished with the following steps:

- GUI
  - Menu:Design-Compile Design
  - Ensure Map Design, Exact Map are checked, along with the Fix design rules and optimize mapping option.
  - Choices for compilation are not all obvious, but high mapping effort is suggested for the best timing results.
  - Incremental mapping will allow the compiler to reiterate, improving over several passes instead of abandoning it's work. This option is also suggested, and may be required for some designs.
  - Auto ungroup may be attractive, as it completely flattens the design for full optimization based on either Area or Delay. Floorplanning in the Encounter flow and debugging the design will become nearly impossible though, so use with caution.
- Command Prompt:
  - `compile -exact_map -area_effort none -power_effort none -map_effort high -incremental_mapping`
    NOTE: If your design is area or power constrained adjust the area or power efforts accordingly.

8.5 Results

Immediately following compilation, again check all errors and warnings and ensure there is no unexpected behavior occurring. If the design is timing constrained, a slack histogram can be analyzed to find critical timing paths, supported in the GUI under MENU:Timing-Path Slack. Double clicking a histogram will pull up more detailed timing analysis, from which the designer can evaluate what needs to be done. While some changes may require RTL to be rewritten if some paths are simply too long, other options include:

- Define a more optimistic or pessimistic wire model, depending if hold time or setup timing is failing.
- Rerun compilation with incremental mapping, DesignVision will reiterate to meet the timing goals.
- A combination of both: run a compile run with optimistic wire modeling to fix hold timing, then a compile run with pessimistic modeling to fix setup timing.

When satisfied, there still remains one step before moving onto the backend. Encounter takes in a .sdc file to determine which nets are clocks, what the desired clock period is, and other timing constraints. To write a design with the name NAME to a .sdc file, the following terminal command must be used: `write_sdc -nosplit NAME.sdc`.

8.6 Scripts

For users that wish to automate these steps, DesignVision supports executing .scr scripts. The format for these scripts, as mentioned in section 8.2, comes from the command.log file that DesignVision automatically generates as it runs both GUI and terminal commands. Custom user created scripts will typically be a string of such commands, and they can be run from inside DesignVision either by `source SCRIPT.scr` or MENU:File-Execute Script. The scripts that the authors used in their flow of the tools are attached in the appendix, with comments detailing every step of the way.

8.7 FAQs

- *Analyze and Elaborate won't work!*
  Make sure that library setup files are all properly defined, and that there are no include statements in the Verilog. Include statements can cause multiple instantiation, and hierarchy is better handled by DesignVision by explicitly defining all files in the Analyze step. Also ensure that the work library between Analyze and Elaborate is consistent.

- *Hold time fixes takes forever! Why won't this converge?*
  If the wire load is very optimistic for setup timing, hold fixes may struggle to be implemented without breaking setup timing. Consider if there are any false hold paths, and if the design still will not converge there is a chance that the back-end Encounter may be able to fix it with more realistic timing. Try starting with a pessimistic wire model, which are the best case for hold. Let it compile once and then compile using the "fix hold only" command. You can then incrementally work towards a more realistic wire model.

- *Paths previously defined for false paths, clock nets, etc. seem to change after compilation, why would the tool do this?*
  Upon compilation DesignVision will read in the standard cell libraries, and this may change the low level hierarchy of the design (most notably anything defined as reg in the Verilog). Sufficiently generalized paths will not break, but any steps that seem to be "lost" can also probably be safely rerun with the new names.

9 Encounter User Guide

"Victory belongs to the most persevering. "
— Napoleon Bonaparte

9.1 Before You Begin

Cadence Encounter is a backend tool that will place and route a gate level design either by automation or by user specifications. This document will only go over the automated flow, with minimal custom routing and placement at the end. Before beginning, the user should have the following files:

- Gate level Verilog file (.v) from DesignVision, or another source.
- Timing constraints file (.sdc) from DesignVision that matches the gate level Verilog.
- Physical libraries (.lef or .vclef) that describe the physical shapes and locations of pins, blockages, and power rails or rings. These may be provided by the technology vendor, or might need to be created with the memory generator.
- Timing libraries (.lib or .tlf) for standard cells and macros that were used to generate the .db files used in DesignVision. These may be provided by the technology vendor.
- IO Assignment File (Optional). If the design will have IO Pads, this file will specify the order of those IOs around the design. If not specified, Encounter will automatically generate one.
- Mapping File (Optional). If creating a GDSII file, the appropriate .map file is necessary to match the metal and via layers to the technology being used. This may need to be modified by hand.

9.2 Finding Help

Encounter primarily supports terminal command help through two commands: `help COMMAND` and `man COMMAND`. When the user does not know what command might be relevant, `help COMMAND` will search the man pages for relevant topics and return possible commands containing the search string COMMAND; `help -k KEY` will return any relevant topics that have KEY in their definition. Both methods can be used to guess at potentially helpful commands. `man COMMAND` will bring up the man page for any specific command, and is useful for exploring various commands and their arguments. Encounter also stores .log files for every session, so crashes may be traceable in these files.

9.3 Design Import

Before even importing the design, considerations for how Encounter will read the Verilog must be accounted for. If the Verilog contains any level of hierarchy (likely), it will have very likely created assign statements in the gate level equivalent. Encounter will not accept assign statements, the designer will need to set a flag by running a command, `setDoAssign`, that allows Encounter to replace assign statements with buffers. This command must be run first, and Encounter will not accept it after importing a design. To verify that the command set correctly, run `setDoAssign -stat`.

After assessing whether setDoAssign is necessary or not, the design may be imported by going into the GUI and running MENU:Design-Design Import. In the pop-up window that appears, the following fields must be filled out:

Design

Verilog Files gate level Verilog generated by DesignVision

Top Cell By User: (TOP CELL NAME)

LEF Files all .lef or .vclef files relevant to the design

Common Timing Libraries all .lib files relevant to the design

Buffer Name/Footprint list of buffer cell names, or footprint name from the standard cell library

Delay Name/Footprint list of delay cell names, or footprint name from the standard cell library

Inverter Name/Footprint list of inverter cell names, or footprint name from the standard cell library

IO Assignment File Pad assignment file, if desired

Timing

Timing Constraint File .sdc generated in DesignVision correlating to the gate level Verilog

Power

Power Nets A list of the power nets included in the Verilog, such as VDD and 1'b1.

Ground Nets A list of the ground nets included in the Verilog, such as VSS and 1'b0.

9.4 Floorplanning & Place

Once the design is imported, Cadence will display an estimated chip area with modules on the left of the chip and blocks on the right, and IO Pads surrounding it if included. Modules include any hierarchy that will be built using standard cells that can be flexible in shape, while blocks are inflexible macros consisting of memory or any custom layout. These blocks must first be placed in a rough sketch of where they will eventually reside, known as the floorplan. In addition, before place can be run any blocks must have a halo defined around them, creating placement blockage near them so standard cells will not fail DRC. Again these commands can be done either in the GUI or the terminal:

- GUI
  - MENU:Floorplan-Specify Floorplan...
  - Core size can be defined here. The only recommended change is to create some distance between the core and the IO's:
    Adjust Core to Boundary to be nonzero. With the units in microns, we used 30 for each side boundary.
  - MENU:Floorplan-Place Blocks-Place, pick a nonzero block halo size.

- This will create a floorplan of the various modules in the design. In addition, each block must have its halo specified:
  Click on any block, then MENU:Floorplan-Edit Block Halo. Again a nonzero number is recommended; default units are microns.
- End cap filler cells must be added as well, MENU:Place-Filler-Add End Cap. Select reasonable size filler cells.
  Filler cells are blank cells which have no pins, but prevent minimum density DRC errors. End caps are necessary for the power and ground rail terminations.
- `cutCoreRow` terminal command must be run here as well to create standard cell rows.

- Command Prompt
  - `floorplan -r RATIO UTILIZATION LEFTBOUNDARY RIGHTBOUNDARY TOPBOUNDARY BOTTOMBOUNDARY`
    RATIO: Height to width ratio, UTILIZATION: Core utilization (between 25-75%), BOUNDARY: Core to IO boundary size
    EX: floorlpan -r .99 60 30 30 30 30
  - `addHaloToBlock LEFTBOUNDARY RIGHTBOUNDARY TOPBOUNDARY BOTTOMBOUNDARY BLOCK`
    BLOCK: Instance name of the block
  - `modulePlace -effort h -variableCore -placeBlocksModule -solutionCount 1 -halo HALOSIZE -minAR MINAR -maxAR MAXAR -solFilePrefix mp`
    HALOSIZE: biggest halo size, MINAR & MAXAR: possible area ratios for modules.
  - `addEndCap -preCap FILLERCELL -endCap FILLERCELL -prefix PREFIX`
  - `cutCoreRow` terminal command must be run here as well to create standard cell rows.

Once the floorplan is defined, the user is free to move modules and blocks if they feel there is a better logical placement. While modules can overlap each other, and overlap blocks, blocks and their halos cannot overlap each other or the boundaries. Following floorplanning, the standard cells can then populate the modules and the macros fixed in place:

- GUI
  - MENU:Place-Place..., Choose high effort, timing driven.
  - Double click on any blocks. Change their status to “FIXED”.

- Command Prompt
  - `amoebaPlace -timingDriven -highEffort -ipo`
    NOTE: The -ipo flag will allow Encounter to resize standard cells based on the timing information that it has along with a rough RC calculation based on distance. This option is not available using the GUI.
  - `setBlockPlacementStatus -allHardMacros -status preplaced`
    NOTE: There is no support for ”fixed” in this command, but ”preplaced” achieves the same purpose - the clock placement will not touch the macros.

9.5 Clock

After the standard cells are placed, the clock buffers and inverters may be introduced into the design. If the .sdc file was not correctly specified earlier, and Encounter cannot determine the timing constraints, this step is likely to fail. The clock is created first by specifying the clock tree and then synthesizing; designs with multiple interleaving clock networks must be careful, as they may need to account for reconvergence.

- GUI
  - MENU:Clock-Create Clock Tree Spec, define the clock buffers and inverters.
  - MENU:Clock-Specify Clock Tree, the default will work here as long as it matches the previous step
  - MENU:Clock-Synthesize Clock Tree, and enable "Handle Crossover and Reconvergence" if there is any clock gating or propagated clocks in the design.
- Command Prompt
  - `createClockTreeSpec -output NAME.ctsch -bufFootprint CLK_BUF_FOOTPRINT -invFootprint INV_FOOTPRINT`
  - `specifyClockTree -clkfile NAME.ctsch`
  - `clkSynthesis -forceReconvergent -report NAME/NAME.ctsch`

To view if the clock tree seems reasonable, the GUI command MENU:Clock-Display-Display Clock Tree, has several options for viewing both phase and predicted routing of the tree.

9.6 Power

Integrating power into the design is a multiple step process. Since the gate level Verilog is not likely to include pins for power and ground, the user must specify that the global nets in the Verilog connect to the pins in the technology libraries, and then connect these nets. A power and ground ring around the core, supplemented by power and ground stripes, will distribute power from the pads in a regular fashion over the chip. Special routes then connect the macros and standard cells to these rings and stripes, providing the final connection to the pads. To run this series of commands must be done in the Command Prompt, as the GUI implementation does not correctly connect the global nets.

- Command Prompt
  - `globalNetConnect VERILOG_POWER_NET -type pgpin -pin TECHNOLOGY_POWER_PIN -inst * module {}`
    NOTE: Both VDD and VSS must be defined in this manner
  - `globalNetConnect VERILOG_TIEHI_NET -type tiehi -inst * module {}`
    NOTE: TIEHI is the name of the verilog equivalent of 1'b1, or whatever the design uses for tying pins high.

- `globalNetConnect VERILOG_TIELO_NET -type tielo -inst * module {}`
  NOTE: TIELO is the name of the verilog equivalent of 1'b0, or whatever the design uses for tying pins low.
- `applyGlobalNets`
- `connectToGlobalNet -tie_high_lows 1 -to_global_net VERILOG_POWER_NET -verbose 0 -override_prior_global_connection 0 -in_instances * -nets -pins TECHNOLOGY_POWER_PIN -connect Pins -under_module {}`
- `addRing -spacing_bottom S -spacing_top S -spacing_left S -spacing_right S -width_left W -width_bottom W -width_top W -width_right W -layer_bottom TECH_MLAYER -center 1 -stacked_via_top_layer TECH_MLAYER -via_using_exact_crossover_size 1 -around core -jog_distance 0.2 -offset_bottom 0.2 -layer_top TECH_MLAYER -layer_right TECH_MLAYER -layer_left TECH_MLAYER -threshold 0.2 -offset_left 0.2 -offset_right 0.2 -offset_top 0.2 -offset_bottom 0.2 -stacked_via_bottom_layer TEC_VIA -nets { POWER_NETS }`
  NOTE: This seemingly complicated instruction can also be accomplished in the GUI, and this is merely the command output found in the .cmd file.
  NOTE: Please see `man addRing` for more about these paramters.
- `addStripe -block_ring_top_layer_limit TECH_MLAYER -max_same_layer_jog_length 8 -padcore_ring_bottom_layer_limit TECH_MLAYER -number_of_sets 8 -stacked_via_top_layer TECH_MLAYER -via_using_exact_crossover_size 1 -padcore_ring_top_layer_limit TECH_MLAYER -spacing 5 -xleft_offset 100 -xright_offset 100 -merge_stripes_value 0.2 -layer TECH_MLAYER -block_ring_bottom_layer_limit TECH_MLAYER -width 4 -stacked_via_bottom_layer TECH_MLAYER -nets { POWER_NETS }`
  NOTE: This seemingly complicated instruction can also be accomplished in the GUI, and this is merely the command output found in the .cmd file.
  NOTE: Please see `man addRing` for more about these paramters.
- `sroute -jogControl {preferWithChanges differentLayer } -corePinJogViaMultiplier 2 -blockPinJogViaMultiplier 2`
  NOTE: The via multipliers correct DRCs that Encounter creates while routing the special nets. The vias violate min space violations, and the multipliers ensure that the vias will be sufficiently large enough to pass these DRCs. Again, experimentation and further reading is encouraged.

To check to see if the power distribution network is sufficiently robust, the designer should run an IR check. Since $V = I * R$, the voltage drop across the chip will vary as the resistance on the wires varies. Standard cells need sufficiently high voltage to maintain true digital operation, and the drop from true V_{dd} is too much, the chip will not function as intended. To run the IR check:

- GUI

- MENU:Route-Trial Route, the defaults should be fine
  NOTE: Creating a fake trial route is necessary for first order RC approximations.
- MENU:Timing, Extract RC
  NOTE: This step is to estimate the resistance in $V = IR$.
- MENU:Power-Edit Pad Location, enter the power line (ex: VDD), auto-fetch and save
- MENU:Power-Power Analysis-Statistical, with the Net Names: power line (ex: VDD), and Pad Location Files: the file created in the previous step
- MENU:Power-Display-Display Rail Analysis Results, set the Net Name: power line, Type: IRD or IVD, and IRD Threshold to worst possible drop (ex .2V), Apply

9.7 Route

Once the standard cells are placed, the clock tree is synthesized, and the power grid is created, the design may be routed and real timing results extracted. There are two parts to the routing: a global route handling most high level and distant connections, and then detail routing which connects the instances to these global lines and can iteratively fix broken timing paths, DRC violations, and noise failures. The command `globalDetailRoute` will do both for the user, but if they so choose they can can also split up the effort into distinctive stages with the `globalRoute` and `detailRoute` commands.

By using `detailRoute` separate from global routing, the designer can choose different focuses for the iterations, choosing between timing fixes at first and then moving to specifically DRC fixes. All nanoroute attributes (the Encounter Routing tool) are set using the `setNanoRouteMode` command. Nanoroute can fix antenna DRC violations on its own, either by routing or replacing filler cells with antenna diodes if provided by the technology library. Additionally, Nanoroute can give different nets varying weights, and in the example below the clocks are given a very high weight so that they are routed first.

The flow below separates global and detailed routing, first doing a global route, with one iteration of detail routing to asses hold time violations. Once hold time violations are fixed using the `fixHoldViolation`, filler cells are added to the design. The order is important, as the buffers necessary to fix hold timing cannot replace the filler cells, only be inserted. Global routing and one iteration of detail routing is done again before one pass to fix setup timing violations. Several iterations are then devoted to fixing DRC violations, as this design project valued correctness of design over timing. The detail iterations will incrementally improve a design, and the various options for `setNanoRouteMode` should be explored for each individual designers needs. A sample routing script is included below:

- Command Prompt
  - `setAttribute -net @CLOCK -weight 100`
    NOTE: Setting the weight on the clocks high so they are routed first
  - `setNanoRouteMode -routeAllowPowerGroundPin true`
  - `setNanoRouteMode -routeWithEco false`
  - `setNanoRouteMode -routeWithSiDriven false`
  - `setNanoRouteMode -routeSelectedNetOnly false`

- `setNanoRouteMode -envNumberProcessor 1`
- `trialRoute -handlePreroute`
- `globalRoute`
- `setNanoRouteMode -drouteStartIteration 0`
- `setNanoRouteMode -drouteEndIteration 0`
- `detailRoute`
  STATUS: Global Route and Detail Route done, fix Hold Violations and add filler cells
- `fixHoldViolation`
- `findCoreFillerCells`
- `addFiller -cell FILL8TS FILL64TS FILL4TS FILL32TS FILL2TS FILL1TS FILL16TS -prefix FILLER`
  STATUS: Hold time buffers, filler cells added, do a quick global and detail route
- `setNanoRouteMode -routeWithTimingDriven false`
- `setNanoRouteMode -drouteFixAntenna false`
- `setNanoRouteMode -routeInsertAntennaDiode false`
- `setNanoRouteMode -routeInsertAntennaDiodeForClockNet false`
- `setNanoRouteMode -drouteStartIteration 0`
- `setNanoRouteMode -drouteEndIteration 0`
- `globalRoute`
- `detailRoute`
  STATUS: Do a route fix for timing
- `setNanoRouteMode -timingEngine CTE`
- `setNanoRouteMode -routeWithTimingDriven true`
- `setNanoRouteMode -routeTdrEffort 6`
- `setNanoRouteMode -drouteStartIteration 1`
- `setNanoRouteMode -drouteEndIteration 1`
- `detailRoute`
  STATUS: Do a route fixes for DRC violations
- `setNanoRouteMode -drouteFixAntenna true`
- `setNanoRouteMode -routeInsertAntennaDiode true`
- `setNanoRouteMode -routeInsertAntennaDiodeForClockNet true`
- `setNanoRouteMode -routeAntennaCellName ‘‘ANTENNATS’’`
- `setNanoRouteMode -routeReInsertFillerCellList fillerCell.list`
- `setNanoRouteMode -drouteStartIteration 2`

- `setNanoRouteMode -drouteEndIteration 19`
- `detailRoute`
  STATUS: Finish routing optimizations
- `setNanoRouteMode -drouteStartIteration 20`
- `setNanoRouteMode -drouteEndIteration default`
- `detailRoute`

Nanoroute can also be issued from the GUI using MENU:Route-Nanoroute.

9.8 Post-Route

Following routing, timing analysis and DRC violations can be completed. Before timing analysis can be completed, however, the RC values must be extracted by going into MENU:Timing-Extract RC. Timing analysis in Encounter is very similar to DesignVision, a path slack histogram is available in the GUI MENU:Timing-Timing Debug-End Point Slack Histogram. Timing violations may require more robust routing, verilog changes, or more extreme measures. The routing script above should address all hold time violations using a custom Encounter command.

DRC violations can be found under the variety of commands in MENU:Verify. Some common fixes for DRC errors may include the following:

- `fixMinCutVia` will automatically try to fix all min cut via DRC violations.
- Geometry violations can be addressed by changing Nanoroute settings to either increase or decrease multipliers, depending on whether the majority of violations are spacing or min area violations. Also fixable in ECO mode by wire editing.
- Antenna violations may be fixed automatically by Nanoroute as well, using the settings described above to address antenna violations. If these still do not prove sufficient, ECO changes may be necessary. `setNanoRouteMode -routeWithEco true` will enable ECO routing for Nanoroute, but `globalDetailRoute` is the only routing tool available in this mode. Possible ECO changes include:
  - Wire Editing: Use “e” to select a net for editing, and the GUI options on the left side of the screen provide multiple wire manipulation tools.
  - MENU:Timing-Interactive ECO, allows the designer to change individual cells. If nets from the pads fail the antenna check, try removing a filler cell and adding a buffer along the line before rerouting. Ensure that the doRefinePlacement checkbox is enabled.
  - If filler cells are removed and buffers added, adding the filler cells again will be necessary to ensure no other new DRC failures.
- Metal Density violations can be fixed by MENU:Route-Metal Fill-Add Metal Fill.

9.9 Scripts

Encounter terminal runs .tcl scripts, and provides a limited shell for the user. All Encounter commands are recorded in a *.cmd file which can then be merged with custom user tcl scripts. To run these scripts, issue `source SCRIPT.tcl`, where SCRIPT is the name of the tcl file you choose to run. Again, sample tcl scripts are included at the end of this document.

9.10 Verilog Considerations: Power

For the Encounter backend to correctly recognize the global power and ground nets necessary for section 9.6, global nets must be connected in the Verilog file. These nets are inout, and may be included only on the topmost level of hierarchy, and do not need to descend in the hierarchy, but must be present as a global net. The tie high and low verilog equivalents (1'b1 and 1'b0) do not need to be propagated up the hierarchy either, they are automatically recognized by Encounter.

9.11 IO Pads

In order to include IO Pads in the design, additional changes to the Verilog are again necessary. The technology vendor should provide all the necessary files, including the .lib, .lef, .db, and .v files to incorporate the pads into the design. At every point in the flow these files must be included much like the other macros, but Encounter will be able to identify the pads as unique and treat them separate from the rest of the design. Since the pads will vary with each vendor, the only way to correctly identify what each pad will do is to refer to the vendor documentation. Any changes to the Verilog must include these gates, and be written by hand. In general however, there will be several types of cells:

- Power and Ground for the Core - instantiate in the Verilog, connect pins to Power and Ground nets.
- Power and Ground for the Rings - instantiate in the Verilog, but do not connect nets.
  NOTE: The pads have a power ring internal to them, and these pads will provide power to the rest of the pads without any explicit routing on the designers part. The designer must ensure that there are no gaps in the pad ring, however, as this disconnects the pad power ring.
- Other Power and Grounds - For multiple power domains, including analog power
- Input Driver Pads - instantiate and connect the appropriate nets, change top level port connectivity to reflect that inputs must go through these drivers.
- Output Driver Pads - instantiate and connect the appropriate nets, change top level port connectivity to reflect that inputs must go through these drivers.
- Empty / Dummy Pads - instantiate as necessary (ex: four corner pads) to fill in gaps in the pads.

In addition, a designer may choose a specific order for their pads, either to group related signals, place power and ground pads specifically, or to ensure correct orientation on cells such as the corner pads. This order can be written to any text file and will be inported as described in section 9.1. The pads should be described in the following format, line separated `Pad: IO_INST ORIENTATION`, e.g. `Pad: io_inst/inst_PCORNER_SE SE`.

10 Bibliography

References

[1] ARM Limited. *Thumb 16-bit Instruction Set Quick Reference Card*, March, 2007.

[2] Kai Borre, Dennis Akos, Nicolaj Bertelsen, Peter Rinder, and Søren Jensen. *A Software-Defined GPS and Galileo Receiver: A Single-Frequency Approach*. Birkhäuser, 2007.

[3] Dave Comisky, Sanjive Agarwala, and Charles Fuoco. A scalable high-performance dma architecture for dsp applications. Technical report, Texas Instruments, 2000.

[4] Dan Ernst, Nam Sung Kim, Shidhartha Das, and et al. Razor: A low-power pipeline based on circuit-level timing speculation. *36th Annual International Symposium on Microarchitecture*, 2003.

[5] Brent M. Ledvina, Mark L. Psiaki, Steven P. Powell, and Paul M. Kitner. Bit-wise parallel algorithms for efficient software correlation applied to gps software receiver. *IEEE Transactions on Wireless Communications, Vol. 3, No. 5, September 2004*, 2004.

[6] MIPS Technologies, MIPS Technologies, Inc., 1225 Charleston Road, Mountain View, CA 94043-1353. *The MIPS16e. Application-Specific Extension to the MIPS32 Architecture*, July 1, 2005.

[7] Christian Piguet, Jean-Marc Masgonty, Claude Arm, and et al. Low-power design of 8-bit embedded coolrisc microcontroller cores. *IEEE Journal Of Solid-State Circuits, Vol. 32, No. 7, July 1997*, 1997.

[8] Texas Instruments Semiconductor Group. *IEEE Std 1149.1 (JTAG) Testability*, 2007.

[9] Brett Alan Warneke. *Ultra-Low Energy Architectures and Circuits for Cubic Millimeter Distrubuted Wireless Sensor Networks*. PhD thesis, Univerty of California in Berkeley, 2003.

11 Appendix

"The truth is always a compound of two half-truths, and you never reach it, because there is always something more to say."
— Tom Stoppard

A BRC ISA

A.1 Description

This is the reference manual for the Big Red Chip instruction set architecture. It has been designed for use in low power, embedded systems. As such, it is a 16-bit ISA, with each instruction being 16-bit. This causes it to have access to fewer registers (eight) and less range on branch and jump instructions, but this should be acceptable as the data cache access time should be one cycle, so loads and stores will be fast. Since the instruction cache will not be very large a small branch and jump range is acceptable.

The ISA is a RISC, load-store architecture, based off of MIPS. The instructions that were not included were mainly unsigned arithmetic instructions and branches. We decided to leave out unsigned arithmetic instructions and never detect overflow, as it is not commonly used in C programming. Many branches were left out as we could not support branches which required two registers, as this would not leave us with a big enough immediate field.

A.2 Formats

The instructions come in four formats: a three register format, a two register format, a one register format, and a zero register format. The three register format contains an extra 2 bits for the function, an addition to the opcode. The other formats all include an immediate field of varying size. The formats are all very regular as to allow for extremely easy decoding.

	5	3	3	3	2
3R	op	rs	rd	rt	func
2R	op	rs	rd	imm	
1R	op	rs	imm		
0R	op	imm			

Table 11: Instruction Formats

A.3 Notation

A.4 Instructions

add Add

5	3	3	3	2
op	rs	rd	rt	func
0x02				0x0
arith				

Symbol	Meaning
GPR[x]	The value of the xth General Purpose Register
DMEM[$addr$]	The value of the data memory at $addr$
PC	The Program Counter register
HI	The high register used by multiplication and division instructions
LO	The low register used by multiplication and division instructions
$\leftarrow$	Assign the value on the right hand side to the left hand side
$+$	Addition
$-$	Subtraction
$\times$	Multiplication
$\div$	Integer division
mod	Integer modulo
$\wedge$	Bitwise and
$\vee$	Bitwise or
$\oplus$	Bitwise xor
$\neg$	Bitwise negation
$<<$	Shift left
$>>$	Shift right
$\|\|$	Bit concatenation
$X_{a..b}$	Bit selection - take bits a through b of X
X_a	Bit selection - take bit a of X
$=, \neq$	Check for equality, inequality (true = 1, false = 0)
$<, \leq$	Check for less than, less than or equal to (true = 1, false = 0)
$>, \geq$	Check for greater than, greater than or equal to (true = 1, false = 0)
sign_ext(x)	Sign extend x
zero_ext(x)	Zero extend x
$\emptyset$	Do nothing

Table 12: Notations Used in This Document

Format: `ADD rd, rs, rt`
Operation: GPR[rd] $\leftarrow$ GPR[rs] + GPR[rt]

addi Add Immediate

5	3	3	5
op 0x0E	rs	rd	imm

Format: `ADDI rd, rs, imm`
Operation: GPR[rd] $\leftarrow$ GPR[rs] + sign_ex(imm)

and And

5	3	3	3	2
op 0x01 log	rs	rd	rt	func 0x0

Format: `AND rd, rs, rt`
Operation: `GPR[rd]` $\leftarrow$ `GPR[rs]` $\wedge$ `GPR[rt]`

andi And Immediate

5	3	3	5
op 0x0A	rs	rd	imm

Format: `ANDI rd, rs, imm`
Operation: `GPR[rd]` $\leftarrow$ `GPR[rs]` $\wedge$ `sign_ex(imm)`

bgez Branch on Greater than or Equal to Zero

5	3	8
op 0x14	rs	imm

Format: `BGEZ rs, imm`
Operation: `if(GPR[rs]` $\geq$ `0) then PC` $\leftarrow$ `PC` $+$ `sign_ex(imm` $<<$ `1)`

bltz Branch on Less Than Zero

5	3	8
op 0x15	rs	imm

Format: `BLTZ rs, imm`
Operation: `if(GPR[rs]` $<$ `0) then PC` $\leftarrow$ `PC` $+$ `sign_ex(imm` $<<$ `1)`

bnz Branch on Not Zero

5	3	8
op 0x13	rs	imm

Format: `BNZ rs, imm`
Operation: `if(GPR[rs]` $\neq$ `0) then PC` $\leftarrow$ `PC` $+$ `sign_ex(imm` $<<$ `1)`

break Break

5	11
op 0x18	imm

Format: `BREAK`
Note:
The break instruction puts the processor into debug mode, causing it to halt execution. In order for execution to resume the processor must be restarted through an external chip interface.
This instruction currently does not need an immediate field. Future generations may use it to define different types of breaks.

bz Branch on Zero

5	3	8
op 0x12	rs	imm

Format: `BZ rs, imm`

Operation: `if(GPR[rs] = 0) then PC ← PC + sign_ex(imm << 1)`

div Divide

5	3	3	3	2
op 0x03 md	rs	rd 0x0	rt	func 0x2

Format: `DIV rs, rt`

Operation:

`LO` ← `GPR[rs]` ÷ `GPR[rt]`

`HI` ← `GPR[rs]` mod `GPR[rt]`

Note:

DIV treats both numbers as signed values.

This is an optional instruction which should be emulated in software if not implemented in hardware. We will not be implementing this instruction in the first chip.

divu Unsigned Divide

5	3	3	3	2
op 0x03 md	rs	rd 0x0	rt	func 0x3

Format: `DIVU rs, rt`

Operation:

`LO` ← `GPR[rs]` ÷ `GPR[rt]`

`HI` ← `GPR[rs]` mod `GPR[rt]`

Note:

DIVU treats both numbers as unsigned values.

This is an optional instruction which should be emulated in software if not implemented in hardware. We will not be implementing this instruction in the first chip.

extend Extend

5	11
op 0x19	imm

Format: `EXTEND imm`

Note:

In order to compensate for BRC ISAs small immediate fields and lack of a zero register, an instruction that is able to set a 16-bit immediate on any instruction has been defined. When executed, the 11-bit immediate is stored. The next instruction is then extended with this value instead of sign or zero extended, as follows: $(\texttt{ext\_imm} << 5)||\texttt{imm}_{4..0}$

This instruction should only proceed instructions with meaningful immediate fields or it has undefined behavior. There should not be two extend instructions in a row. If implemented on a system supporting exceptions and interrupts, this instruction and the following should be treated as atomic and no exceptions or interrupts may occur between them.

j Jump

5	11
op 0x16	imm

Format: `J imm`

Operation: $\texttt{PC} \leftarrow \texttt{PC}_{15..12}||(\texttt{imm} << 1)$

jal Jump And Link

5	11
op 0x17	imm

Format: `JAL imm`

Operation:

$\texttt{GPR}[7] \leftarrow \texttt{PC} + 2$

$\texttt{PC} \leftarrow \texttt{PC}_{15..12}||(\texttt{imm} << 1)$

jr Jump to Register

5	3	3	3	2
op 0x06 spec	rs	rd 0x0	rt 0x0	func 0x0

Format: `JR rs`

Operation: $\texttt{PC} \leftarrow \texttt{GPR}[\texttt{rs}]_{15..1}||0$

Note:

As the PC must always be word aligned, the LSB of the register value is ignored when using this instruction.

lb Load Byte

5	3	3	3	2
op	rs	rd	rt	func
0x04			0x0	0x0
meml				

Format: `LB rd, rs`
Operation: GPR[rd] ← sign_ex(DMEM[GPR[rs]]$_{7..0}$)

lbu Load Byte Unsigned

5	3	3	3	2
op	rs	rd	rt	func
0x04			0x0	0x1
meml				

Format: `LBU rd, rs`
Operation: GPR[rd] ← zero_ex(DMEM[GPR[rs]]$_{7..0}$)

li Load Immediate

5	3	3	5
op	rs	rd	imm
0x11	0x0		

Format: `LI rd, imm`
Operation: GPR[rd] ← sign_ext(imm)

lw Load Word

5	3	3	3	2
op	rs	rd	rt	func
0x04			0x0	0x2
meml				

Format: `LW rd, rs`
Operation: GPR[rd] ← DMEM[GPR[rs]$_{15..1}$||0]$_{15..0}$
Note:
Load word is required to have aligned memory accesses, which is enforced by ignoring the LSB of the address value.

mfhi Move From Hi

5	3	3	3	2
op	rs	rd	rt	func
0x06	0x0		0x0	0x2
spec				

Format: `MFHI rd`

Operation: `GPR[rd]` $\leftarrow$ `HI`

Note:

This is an optional instruction which is not needed if all multiply and divide instructions are not implemented. We will not be implementing this instruction in the first chip.

mflo Move From Lo

5	3	3	3	2
op	rs	rd	rt	func
0x06	0x0		0x0	0x3
spec				

Format: `MFLO rd`

Operation: `GPR[rd]` $\leftarrow$ `LO`

Note:

This is an optional instruction which is not needed if all multiply and divide instructions are not implemented. We will not be implementing this instruction in the first chip.

mult Multiply

5	3	3	3	2
op	rs	rd	rt	func
0x03		0x0		0x0
md				

Format: `MULT rs, rt`

Operation: $(\texttt{HI}, \texttt{LO}) \leftarrow \texttt{GPR[rs]} \times \texttt{GPR[rt]}$

Note:

MULT treats both numbers as signed values.

This is an optional instruction which should be emulated in software if not implemented in hardware. We will not be implementing this instruction in the first chip.

multu Multiply Unsigned

5	3	3	3	2
op	rs	rd	rt	func
0x03		0x0		0x1
md				

Format: `MULTU rs, rt`

Operation: $(\texttt{HI}, \texttt{LO}) \leftarrow \texttt{GPR[rs]} \times \texttt{GPR[rt]}$

Note:

MULTU treats both numbers as unsigned values.

This is an optional instruction which should be emulated in software if not implemented in hardware. We will not be implementing this instruction in the first chip.

nop No Operation

5	3	3	3	2
op 0x00 shift	rs 0x0	rd 0x0	rt 0x0	func 0x0

Format: `NOP`
Operation: $\emptyset$

nor Nor

5	3	3	3	2
op 0x01 log	rs	rd	rt	func 0x2

Format: `NOR rd, rs, rt`
Operation: `GPR[rd]` $\leftarrow \neg$ (`GPR[rs]` $\vee$ `GPR[rt]`)

nori Nor Immediate

5	3	3	5
op 0x0C	rs	rd	imm

Format: `NORI rd, rs, imm`
Operation: `GPR[rd]` $\leftarrow \neg$ (`GPR[rs]` $\vee$ `sign_ex(imm)`)

or Or

5	3	3	3	2
op 0x01 log	rs	rd	rt	func 0x1

Format: `OR rd, rs, rt`
Operation: `GPR[rd]` $\leftarrow$ `GPR[rs]` $\vee$ `GPR[rt]`

ori Or Immediate

5	3	3	5
op 0x0B	rs	rd	imm

Format: `ORI rd, rs, imm`
Operation: `GPR[rd]` $\leftarrow$ `GPR[rs]` $\vee$ `sign_ex(imm)`

rfe Return From Exception

5	3	3	3	2
op	rs	rd	rt	func
0x06	0x0	0x0	0x0	0x1
spec				

Note:
This is an optional instruction. We do not have exceptions in our implementation and therefore do not need this instruction. It is included for completeness.

sb Store Byte

5	3	3	3	2
op	rs	rd	rt	func
0x05		0x0		0x0
mems				

Format: `SB rs, rt`
Operation: $\texttt{DMEM}[\texttt{GPR}[\texttt{rs}]]_{7..0} \leftarrow \texttt{GPR}[\texttt{rt}]_{7..0}$

sll Shift Left Logical

5	3	3	5
op	rs	rd	imm
0x07			

Format: `SLL rd, rs, imm`
Operation: $\texttt{GPR}[\texttt{rd}] \leftarrow \texttt{GPR}[\texttt{rs}] << \texttt{imm}_{3..0}$

sllv Shift Left Logical Variable

5	3	3	3	2
op	rs	rd	rt	func
0x00				0x1
shift				

Format: `SLLV rd, rs, rt`
Operation: $\texttt{GPR}[\texttt{rd}] \leftarrow \texttt{GPR}[\texttt{rs}] << \texttt{GPR}[\texttt{rt}]_{3..0}$

slt Set Less Than

5	3	3	3	2
op	rs	rd	rt	func
0x02				0x2
arith				

Format: `SLT rd, rs, rt`
Operation: `GPR[rd]` ← (`GPR[rs]` < `GPR[rt]`)
Note:
SLT compares the two numbers as signed two's complement numbers.

slti Set Less Than Immediate

5	3	3	5
op 0x0F	rs	rd	imm

Format: `SLTI rd, rs, imm`
Operation: `GPR[rd]` ← (`GPR[rs]` < `sign_ex(imm)`)
Note:
SLTI compares the two numbers as signed two's complement numbers.

sltiu Set Less Than Immediate Unsigned

5	3	3	5
op 0x10	rs	rd	imm

Format: `SLTIU rd, rs, imm`
Operation: `GPR[rd]` ← (`GPR[rs]` < `sign_ex(imm)`)
Note:
SLTIU compares the two numbers as unsigned numbers.

sltu Set Less Than Unsigned

5	3	3	3	2
op 0x02 arith	rs	rd	rt	func 0x3

Format: `SLTU rd, rs, rt`
Operation: `GPR[rd]` ← (`GPR[rs]` < `GPR[rt]`)
Note:
SLTU compares the two numbers as unsigned numbers.

sra Shift Right Arithmetic

5	3	3	5
op 0x09	rs	rd	imm

Format: `SRA rd, rs, imm`

Operation: $\texttt{GPR[rd]} \leftarrow \texttt{GPR[rs]} >> \texttt{imm}_{3..0}$

Note:

An arithmetic right shift shifts in the most significant bit, as to not change the sign of the number.

srav Shift Right Arithmetic Variable

5	3	3	3	2
op	rs	rd	rt	func
0x00				0x3
shift				

Format: `SRAV rd, rs, rt`

Operation: $\texttt{GPR[rd]} \leftarrow \texttt{GPR[rs]} >> \texttt{GPR[rt]}_{3..0}$

Note:

An arithmetic right shift shifts in the most significant bit, as to not change the sign of the number.

srl Shift Right Logical

5	3	3	5
op	rs	rd	imm
0x08			

Format: `SRL rd, rs, imm`

Operation: $\texttt{GPR[rd]} \leftarrow \texttt{GPR[rs]} >> \texttt{imm}_{3..0}$

Note:

A logical right shift will always shift in zeros.

srlv Shift Right Logical Variable

5	3	3	3	2
op	rs	rd	rt	func
0x00				0x2
shift				

Format: `SRLV rd, rs, rt`

Operation: $\texttt{GPR[rd]} \leftarrow \texttt{GPR[rs]} >> \texttt{GPR[rt]}_{3..0}$

Note:

A logical right shift will always shift in zeros.

sub Subtract

5	3	3	3	2
op	rs	rd	rt	func
0x02				0x1
arith				

Format: `SUB rd, rs, rt`
Operation: $\texttt{GPR}[\texttt{rd}] \leftarrow \texttt{GPR}[\texttt{rs}] - \texttt{GPR}[\texttt{rt}]$

sw Store Word

5	3	3	3	2
op 0x05 mems	rs	rd 0x0	rt	func 0x1

Format: `SW rs, rt`
Operation: $\texttt{DMEM}[\texttt{GPR}[\texttt{rs}]_{15..1}||0]_{15..0} \leftarrow \texttt{GPR}[\texttt{rt}]$
Note:
Store word is required to have aligned memory accesses, which is enforced by ignoring the LSB of the address value.

syscall System Call

5	11
op 0x1A	imm

Note:
This is an optional instruction. We do not plan on running a full featured operating system on our chip or having a kernel and user mode. It is included for completeness.

xor Xor

5	3	3	3	2
op 0x01 log	rs	rd	rt	func 0x3

Format: `XOR rd, rs, rt`
Operation: $\texttt{GPR}[\texttt{rd}] \leftarrow \texttt{GPR}[\texttt{rs}] \oplus \texttt{GPR}[\texttt{rt}]$

xori Xor Immediate

5	3	3	5
op 0x0D	rs	rd	imm

Format: `XORI rd, rs, imm`
Operation: $\texttt{GPR}[\texttt{rd}] \leftarrow \texttt{GPR}[\texttt{rs}] \oplus \texttt{sign\_ex}(\texttt{imm})$

A.5 Reserved Op Codes

Op codes which are unused and should be treated as reserved for customizations:

- 0x1B
- 0x1C
- 0x1D
- 0x1E
- 0x1F

B BRC GPS ISA Additions

B.1 Description

This is a description of the custom instructions included in the BRC ISA for the first generation chip which will be integrated into a GPS tracking system.

B.2 Instructions

adccnt ADC Count

5	3	3	3	2
op	rs	rd	rt	func
0x1B		0x0		0x0

Format: `ADCCNT rs, rt`
Operation: `Rcnt[GPR[rt]]` ← `GPR[rs]`
Note:
This instruction sets the count for the particular ADC in order to fix for the Doppler effect.

adcld ADC Load

5	3	3	3	2
op	rs	rd	rt	func
0x1C			0x0	0x0

Format: `ADCLD rd, rs`
Operation: `GPR[rd]` ← `FIFO[GPR[rs]]`
Note: This instruction loads the next value from the ADC queue.

gc Load Gold Code

5	3	3	5
op	rs	rd	imm
0x1D			0x0

Format: `GC rd, rs`

Operation: $\texttt{GPR}[\texttt{rd}] \leftarrow \texttt{gc}(\texttt{GRP}[\texttt{rs}]_{9..8}, \texttt{GPR}[\texttt{rs}]_{4..0})$

Note:

This instruction gets the next gold code chip from the state machine specified in bits 9 through 8 with a phase selection specified by bits 4 through 0.

popc Population Count

5	3	3	5
op	rs	rd	imm
0x1E			0x0

Format: `POPC rd, rs`

Operation: $\texttt{GPR}[\texttt{rd}] \leftarrow \texttt{pop\_count}(\texttt{GPR}[\texttt{rs}])$

Note:

A population count is the number of 1's appearing in the binary representation of a number.

C Boot Sequence

Instructions on booting the chip

Power up the chip. The initial values of the following signals should be set to the following values (all values are single bits unless otherwise specified in Verilog format):

Pin name	Value	Description
debug_clk	clock	Slow debug clock - ~100kHz
ext_clk	clock	Fast external clock - 2x the running frequency
int_clk	clock	Fast internal clock - generated by an oscillator
ext_adc_clk	clock	Slow ADC clock - 8.192MHz
reset	1	Main chip reset
clk_reset	1	Clock state machine reset
cont_en_f	1	FPGA controller enable signal
scan_en_f	0	Main scan chain enable
scan_in_f	0	Main scan chain scan in
clk_scan_en_f	0	Clock state machine scan enable
clk_scan_in_f	0	Clock state machine scan in
sc_over_f	0	Stop clock override
reset_br_f	0	Turns break off to run the program
cont_addr_f	13'h0	Address to access memory at
cont_wdata_f	16'h0	Data to write to memory
cont_rw_sel_f	1	Read/write control - 0 is write, 1 is read
cont_di_sel_f	1	Data/inst control - 0 is data, 1 is instruction
cont_c_en_f	0	Core/port control - 0 is the port, 1 is the core
chip_en	1	Overall chip enable - should be tied to power
clk_sel	1	Internal/External clock select - 0 is int, 1 is ext
adc_clk_sel	0	ADC clock select - 0 is generating, 1 is external
adc_in	input	Input from the ADC

Start the clock state machine. Be sure the debug clock and external clock is running. Deassert the clock reset after at least one full cycle of the external clock to ensure the clock state machine has been correctly initialized.

Signal	New Value
clk_reset	0

Load the instructions into the instruction memory. Be sure that cont_di_sel_f is 1 for instruction and that cont_c_en_f is 0 for port control. Now switch cont_rw_sel_f to 0 to begin writing. This will write one word per cycle, on the negative edge of the debug clock. It is recommended to update the cont_addr_f and cont_wdata_f on the positive edge of the debug clock in order to avoid metastability problems. Switch con_rw_sel_f back to 1 after finishing the writing.

Signal	New Value
cont_rw_sel_f	0
On each positive edge of debug clock:	
cont_addr_f	cont_addr_f + 1
cont_wdata_f	next instruction word
When no more instructions are left:	
cont_rw_sel_f	1

Initialize data memory if necessary. If the program requires preloaded data, such as global arrays, load it now. It is very similar to loading instructions. First switch cont_di_sel_f to 0 for data. Set cont_addr_f to the initial address of the data to load. Then follow the instruction load directions.

Signal	New Value
cont_di_sel_f	0
cont_addr_f	initial data address
Wait one cycle	
cont_rw_sel_f	0
On each positive edge of debug clock:	
cont_addr_f	next data address location
cont_wdata_f	next data word
When no more data is left:	
cont_rw_sel_f	1

Start the program. In order to begin the program assert cont_c_en_f, deassert reset, and assert reset_br_f. This will give the core control of the memory, stop reseting the registers, and take the processor out of debug mode. Set them in this order. After reset break has been asserted ready will go low, meaning that break has been reset. After this reset_br should be deasserted to allow the next break instruction to correctly enter debug mode. The debug clock should also be turned off at this point to allow for a smooth transition back to debug mode.

Signal	New Value
cont_c_en_f	1
Wait one cycle	
reset	0
Wait one cycle	
reset_br_f	1
Wait for ready to deassert	
reset_br_f	0
debug_clk	turn the clock off

Wait for break command. The ready signal should now be monitored. After the next break command, the clock will begin ramping down. When it is finished and has entered debug mode, ready will go high. The chip will now be waiting for external input before it will continue to do anything. To debug the chip, please see Appendix D for directions on the debug sequence. If the program is not being

debugged and has no break statements, disconnect the FPGA controller and allow the chip to run.

D Debug Sequence

When ready is asserted, the chip has entered debug mode. The debug clock should have been turned off, so the chip will stay paused upon entering this mode. Deassert cont_en_f after ready is asserted, so that the debug clock can safely turn on without changing the state of the chip. Once the debug clock is running, use any of the DFT features described in the subsections of this appendix.

Signal	New Value
Ready is asserted	
cont_en_f	0
Wait a couple cycles to ensure a smooth transition	
debug_clk	start the debug clock

To restart the chip after examining the chip with the following DFT features, restart it in a similar manner to the first boot, except also assert the cont_en_f signal during the last step.

Signal	New Value
cont_c_en_f	1
Wait one cycle	
reset	0
Wait one cycle	
reset_br_f	1
Wait for ready to deassert	
reset_br_f	0
cont_en_f	1
debug_clk	turn the clock off

D.1 Clock Scan Chain

The clock scan chain may either be debugged before starting a program or after a program has entered debug mode through a break command. The clock scan chain updates on the positive edge of the external clock. You should slow down the external clock and update the inputs on the negative edge of the external clock. You should assert the clock scan chain enable and read out the first value from clock scan out on the same cycle. Keep the signal high while reading out the entire scan chain over the next nine cycles. The clock scan chain is ten bits in total, see Appendix E for the registers they correspond to. You should then deassert the clock scan enable. You may now store and analyze this data. In order to restart the processor without reseting it, now repeat this process, while changing clock scan in to the same values read out, in order. These values may be changed in order to change the state of the clock state machine for test vectors and other debugging purposes.

To read out:

Signal	New Value
clk_scan_en_f	1
Wait nine cycles	
clk_scan_en_f	0

To write in:

Signal	New Value
clk_scan_en_f	1
clk_scan_in_f	first value read out
Wait one cycle	
clk_scan_in_f	next value read out
	⋮
After all data has been written back in	
clk_scan_en_f	0

D.2 Scan Chain

The scan chain may either be debugged before starting a program or after a program has entered debug mode through a break command, although it probably will be most useful after a break command. The scan chain updates on the positive edge of the debug clock. You should update the inputs on the negative edge of the debug clock. You should assert the scan chain enable and read out the first value from scan out on the same cycle. Keep the signal high while reading out the entire scan chain over the next 654 cycles. The clock scan chain is 655 bits in total, see Appendix E for the registers they correspond to. You should then deassert the scan enable. You may now store and analyze this data. In order to restart the processor without reseting it, now repeat this process, while changing scan in to the same values read out, in order. These values may also be changed in order to change the state of the processor for test vectors and other debugging purposes.

To read out:

Signal	New Value
scan_en_f	1
Wait 654 cycles	
scan_en_f	0

To write in:

Signal	New Value
clk_scan_en_f	1
clk_scan_in_f	first value read out
Wait one cycle	
clk_scan_in_f	next value read out
	⋮
After all data has been written back in	
clk_scan_en_f	0

D.3 Memory Access

The memory controller allows read and write access to all data and instruction memory while in debug mode. The memory updates on the negative edge of the debug clock so all inputs should be changed on the positive edge of the debug clock. To read from memory, make sure cont_rw_sel_f is set to 1, cont_di_sel_f is set to 0 for data or 1 for instruction memory, cont_c_en_f is set to 0 for port control of the memory, and cont_addr_f is set to the first address to read from. After the next negative edge, the read data will appear on cont_rdata. You may change the address once per cycle to read out any amount of data. When finished, set cont_c_en_f back to 1 to give the core control of the memory. To write data to memory, set cont_rw_sel_f to 0, cont_c_en_f to 0 for port control, and cont_di_sel_f to the appropriate value and set cont_addr_f and cont_wdata_f to the first address and data value to write. When finished, set cont_rw_sel_f back to 1 and cont_c_en_f back to 1.
Note: The cont_addr_f port is only word addressable. You should therefore convert byte addresses to word addresses by dividing the byte address by two (right shift by one).

To read:

Signal	New Value
cont_rw_sel_f	1
cont_di_sel_f	0 for data, 1 for instruction
cont_c_en_f	0
cont_addr_f	address to read from
Wait one cycle	
cont_addr_f	next address to read from
	⋮
After reading the data	
cont_c_en_f	1

To write:

Signal	New Value
cont_rw_sel_f	0
cont_di_sel_f	0 for data, 1 for instruction
cont_c_en_f	0
cont_addr_f	address to write to
cont_wdata_f	data to write
Wait one cycle	
cont_addr_f	next address to read from
cont_wdata_f	next data to write
	⋮
After reading the data	
cont_c_en_f	1
cont_rw_sel_f	1

E Scan Chain Sequence

Note: These are all in big endian.

The complete order of the clock scan chain:

Index	Register	Size
0	CPU.clock_inst.system_clock_divider.counter	1
1	CPU.clock_inst.system_clock_mask.state_machine	8
9	CPU.clock_inst.system_clock_mask.reg_clock_generator	1

The complete order of the scan chain:

Index	Register	Size
0	CPU.break_inst.break_reg	1
1	CPU.clock_inst.adc_clock_divider.counter	4
5	CPU.iff_inst.PC_reg	12
17	CPU.rd_inst.registers.reg00	16
33	CPU.rd_inst.registers.reg01	16
49	CPU.rd_inst.registers.reg02	16
65	CPU.rd_inst.registers.reg03	16
81	CPU.rd_inst.registers.reg04	16
97	CPU.rd_inst.registers.reg05	16
113	CPU.rd_inst.registers.reg06	16
129	CPU.rd_inst.registers.reg07	16
145	CPU.rd_inst.immediate_gen.ext_reg	11
156	CPU.rd_inst.immediate_gen.ext_valid	1
157	CPU.rd_bp_rs_addr_01	3
160	CPU.rd_bp_rt_addr_01	3
163	CPU.rd_bp_rs_01	16
179	CPU.rd_bp_rt_01	16
195	CPU.rd_out_imm_01	16
211	CPU.rd_out_func_01	7
218	CPU.rd_wb_rd_addr_01	3
221	CPU.if_wb_next_pc_01	12
233	CPU.gc_inst.phase_gen0_B	5
238	CPU.gc_inst.phase_gen1_B	5
243	CPU.gc_inst.phase_gen2_B	5
248	CPU.gc_inst.phase_gen3_B	5
253	CPU.gc_inst.gen0.g1_B	10
263	CPU.gc_inst.gen0.g2_B	10
273	CPU.gc_inst.gen1.g1_B	10
283	CPU.gc_inst.gen1.g2_B	10
293	CPU.gc_inst.gen2.g1_B	10

303	CPU.gc_inst.gen2.g2_B	10
313	CPU.gc_inst.gen3.g1_B	10
323	CPU.gc_inst.gen3.g2_B	10
333	CPU.adc_inst.adc_stopStart0_regB	1
334	CPU.adc_inst.adc_stopStart1_regB	1
335	CPU.adc_inst.adc_stopStart2_regB	1
336	CPU.adc_inst.adc_stopStart3_regB	1
337	CPU.adc_inst.stall_hold_regB	2
339	CPU.adc_inst.wr_ptr_clk_reg	4
343	CPU.adc_inst.rd_ptr_clk_reg	4
347	CPU.adc_inst.bufIn.adc_out_reg0	8
355	CPU.adc_inst.bufIn.adc_out_reg1	8
363	CPU.adc_inst.bufIn.adc_out_reg2	8
371	CPU.adc_inst.bufIn.adc_out_reg3	8
379	CPU.adc_inst.bufIn.dat	8
387	CPU.adc_inst.bufIn.valid0	8
395	CPU.adc_inst.bufIn.valid1	8
403	CPU.adc_inst.bufIn.valid2	8
411	CPU.adc_inst.bufIn.valid3	8
419	CPU.adc_inst.dopCtrl0.cntr_init	12
431	CPU.adc_inst.dopCtrl0.dopNew_dly_B	1
432	CPU.adc_inst.dopCtrl0.cntr	12
444	CPU.adc_inst.dopCtrl1.cntr_init	12
456	CPU.adc_inst.dopCtrl1.dopNew_dly_B	1
457	CPU.adc_inst.dopCtrl1.cntr	12
469	CPU.adc_inst.dopCtrl2.cntr_init	12
481	CPU.adc_inst.dopCtrl2.dopNew_dly_B	1
482	CPU.adc_inst.dopCtrl2.cntr	12
494	CPU.adc_inst.dopCtrl3.cntr_init	12
506	CPU.adc_inst.dopCtrl3.dopNew_dly_B	1
507	CPU.adc_inst.dopCtrl3.cntr	12
519	CPU.adc_inst.wr_ptr0_B	9
528	CPU.adc_inst.wr_ptr1_B	9
537	CPU.adc_inst.wr_ptr2_B	9
546	CPU.adc_inst.wr_ptr3_B	9
555	CPU.adc_inst.rd_ptr0_B	8
563	CPU.adc_inst.rd_ptr1_B	8
571	CPU.adc_inst.rd_ptr2_B	8
579	CPU.adc_inst.rd_ptr3_B	8
587	CPU.alu_wb_data_12	16
603	CPU.mem_wb_data_12	16

619	CPU.gc_wb_data_12	1
620	CPU.adc_wb_data_12	16
636	CPU.if_wb_next_pc_12	12
648	CPU.wb1_wb2_addr_12	3
651	CPU.wb1_wb2_en_12	1
652	CPU.wb1_wb2_cont_12	3

F Coverage Document

This section provides a description of coverage metrics and how to implement coverage in our design.

Types of Coverage :

Line / Statement Coverage : Tells which lines of procedural code in the file are not exercised. Will keep metrics on procedural statements, procedural statement blocks, missing conditional statements, and branches for conditional statements.

Condition Coverage : Tests if all conditionals in the following cases are exercised: conditional expression used with "?". Sub-expressions that are operands for AND and OR logical operators in conditional expressions with operators "?", "if" and procedural statements.

Toggle Coverage : Monitors nets and registers for value transitions from 0 to 1 and 1 to 0. Tells you how much activity is being performed in each net, which allows you to see which nets are not being used at all / with low activity.

FSM Coverage : Identifies groups of statements in code belonging to an FSM and monitors current state and state transition metrics.

Path Coverage : Determines whether conditional expressions in IF and CASE statements are true which allows the tester to see which paths have not been tested.

Branch Coverage : Analyzes how IF and CASE statements establish branches in the design.

Limitations : Does not extract statements for FSM's where state is stored in memory.
Does not cover multi-dimensional arrays.
Does not cover the power operator in sub expressions.

How to enable Coverage Metrics :

Compile the design for coverage metrics :
Tell VCS command line which coverage metric you want to compile the design for.
i.e.
`VCS -cm line source_file.v` // runs line coverage on the file
other arguments include: cond, fsm, line, tgl, path, branch
you can include more than 1 argument with (+) between arguments

Run the simulation :

Here, you can tell VCS to collect coverage information for fewer types than the design was compiled for.

i.e.

```
simv -cm line (or other arguments you compiled for)
```

To see the metrics do the following :

GUI: bring up cmView

i.e.

```
vcs -cm_pp gui -cm line
```

file→open coverage→statement menu command
open design hierarchy to view coverage reports
double click on summary window to see which lines not covered in module (in red)

Text output: run cmView in batch mode. Reports will be compiled in text files and formatted to display different types of coverage and result.

i.e.

```
vcs -cm_pp
```

cmView will write the output text files to reports directory. "simv.cm/reports/cmView"
cmView.short_l would report percent line coverage for entire design. Other files provide more in depth analysis

How to Merge Coverage results :

Build simv executable for different test cases, and run the same test bench with different inputs (for the random test cases)

Steps:

1. Compile for coverage, i.e. vcs -cm coverage_arguments source_file
2. Start simulation and specify directory and name for intermediate data log file, i.e. vcs -cm_dir dir_name -cm_name test1 -cm coverage_arguments
3. Do the last step repeatedly with different data log file names and different input vectors
4. run cmView in batch and write reports, i.e. `vcs -cm_pp -cm_dir dir_name -cm_dir dir_name1 -cm_name merged`

Note: When running the above command, cmView will take the intermediate coverage logs from each of the directories and compile them under the first directory listed under the report name "merged".

To put all data log files into a directory other than the standard "./simv.cm/coverage/verilog" directory, use command line option -cm_dir "desired_directory_path/dir_name.cm" in the compilation stage.

Then, to merge all together, run command:

```
vcs -cm_pp -cm_dir dir_path (several directories) -cm_name merged
```

For BRC:

To run coverage, go to the "~/coverage" directory on BRC. In the "rand_testing" subdirectory, you can find a "rand_testing_coverage" script. This script will run until an error is found and die, or run indefinitely. To kill the process, just type "CTRL-C." Even though you kill the process, the script will have enabled logging for all types of coverage for the test cases that were run. If you cd to the cpu symbolic link, you will see a simv.cm directory which contains the intermediate log files for coverage. These are in the "coverage/verilog" directory. To get the final coverage results, from the cpu directory, run "vcs -cm_pp" for the command line option. The GUI option is also available. If you would like to merge coverage results from two different sources you must make some small changes in some command line option at compile time. First, make the new test compile coverage reports in a different directory by using the -cm_dir command line option. For example:

```
vcs +v2k -cm_dir  /coverage/rand_testing/cpu/newsimv.cm -cm path+branch+line+cond+fsm+tgl
+incdir+../../../../design/core_full/trunk/cmn +incdir+../../../../design/core_full/trunk/alu
+incdir+../../../../design/core_full/trunk/rd -f filelist
```

In the above command, the coverage directory is in the newsimv.cm directory. It compiles for all types of coverage with the -cm options being all included.

Finally to merge the results from this test bench and, say, the previous randomly generated test bench coverage numbers, you will run (from the cpu directory again)

```
vcs -cm_pp -cm_dir simv.cm -cm_dir newsimv.cm -cm_name merged
```

This will compile the merged results into a file named "merged." These files will be found in the first directory specified in the command, in this case, "simv.cm/reports/." In that directory, you will see the "merged.*" reports.

Note, currently, we have not found a way to incorporate coverage results that come from different test benches, i.e. the core_full_run_tb.v and core_full_scan_tb.v. VCS does not like the different top level test benches and even trying to exclude them from coverage reports fails. The solution at this time is to put both top level test benches together and maybe adding a control to select between the two.

G GPS Correlations in BRC Assembly

Although it would beyond the scope of this document to include all code involved in the BRC GPS system, we have elected to include a segment showing the required assembly sequence to perform correlation computations on in-phase and quadrature components of the early, late, and promptly shifted C/A code. Hopefully this example code will illustrate exactly how the special instructions implemented in the BRC system can be used to aid in baseband computations.

Also, the reader should note that the code has been arranged such that no special purpose instruction is dependent on data produced in the previous instruction. The code has been arranged this way by the author knowing that the system will stall in this case. This segment of code also illustrates how the software team's full port of the GCC compiler has helped enable the use of our special purpose instructions. As one

can see below, GCC includes functionality to take variables as input arguments to in-line assembly blocks, allowing for smooth introduction of assembly to C programs.

Listing 1: In-line ASM to compute correlations using special-purpose instructions

```
// INLINE ASM does the following:
//    Use adcld function to get new data word
//    Get new PRN chips and align eary and late
//    Correlate early
//    Correlate prompt (this is optional)
//    Correlate late

CA_km1[sat] = CA_k[sat];

// This code creates the new CA_k[sat] and CA_kp1[sat].
//  Note that no dependent instructions immediately follow
//  the gc instruction.
asm volatile ( "gc    %%0 %3;"        // %%0 = gc(CA_id[sat])
    "srl    %2 %2 $8;"    // CA_kp1[sat](input) = CA_kp1[sat](input)>>8
    "gc     %1 %3;"       // CA_kp1[sat](output) = gc(CA_id[sat])
    "andi %%0 $0x00ff;" // %%0 = %%0 & 0x00ff
    "or     %0 %2 %%0;"  // CA_k[sat] = CA_kp1(input) | %%0
    : "=g" (CA_k[sat]), "=g" (CA_kp1[sat]) //Output operands
    : "g" (CA_kp1[sat]), "g" (CA_id[sat]) //Input operands
    : "%0" //Clobbered registers
);

// This code gets the new input data from the ADC fifo and
//  produces the early and late CA codes. I do this all at
//  once to ensure that adcld is not followed by a dependent
//  instruction.
asm volatile ( "adcld  %0  %6;"           // sig_word[sat] = adcld(sat)
    "srl    %%0  %4  $2;"       // %%0 = CA_k[sat] >> 2
    "andi   %%1  %5  $0xc000;" // %%1 = CA_kp1[sat] & 0xc000
    "or      %1 %%0 %%1;"       // CA_early = %%0 | %%1
    "sll    %%0  %4  $2;"       // %%0 = CA_k[sat] << 2
    "andi   %%1  %3  $0x0003;" // %%1 = CA_km1[sat] & 0x0003
    "or      %2 %%0 %%1;"       // CA_late = %%0 | %%1
    : "=g" (sig_word[sat]), "=g" (CA_early),
                  "=g" (CA_late) //Output operands
    : "g" (CA_km1[sat]), "g" (CA_k[sat]),
                  "g" (CA_kp1[sat]), "g" (sat) //Input operands
    : "%0", "%1" //Clobbered registers
```

```
);

sig_word_BB_I = sig_word[sat] ^ IF_word_I;
sig_word_BB_Q = sig_word[sat] ^ IF_word_Q;

corr_word_IE = sig_word_BB_I ^ CA_early;
corr_word_IP = sig_word_BB_I ^ CA_k[sat];
corr_word_IL = sig_word_BB_I ^ CA_late;

corr_word_QE = sig_word_BB_Q ^ CA_early;
corr_word_QP = sig_word_BB_Q ^ CA_k[sat];
corr_word_QL = sig_word_BB_Q ^ CA_late;

// Update correlation accumulator for IE
asm ( "popc %%0 %2;"          // %%0 = popc(corr_word_IE)
"add   %1 %1 %%0;"      // I_early_k[sat] = I_early_k[sat] + %%0
"xori  %2 %2 0xffff;"   // corr_word_IE = ~corr_word_IE
"popc %%0 %2;"          // %%0 = popc(~corr_word_IE)
"sub   %0 %1 %%0;"      // I_early_k[sat] = I_early_k[sat] - %%0
: "=g" (I_early_k[sat]) //Output operands
: "g" (I_early_k[sat]), "g" (corr_word_IE) //Input operands
: "%%0" //Clobbered registers
);
// Update correlation accumulator for QE
asm ( "popc %%0 %2;"          // %%0 = popc(corr_word_IE)
"add   %1 %1 %%0;"      // I_early_k[sat] = I_early_k[sat] + %%0
"xori  %2 %2 0xffff;"   // corr_word_IE = ~corr_word_IE
"popc %%0 %2;"          // %%0 = popc(~corr_word_IE)
"sub   %0 %1 %%0;"      // I_early_k[sat] = I_early_k[sat] - %%0
: "=g" (Q_early_k[sat]) //Output operands
: "g" (Q_early_k[sat]), "g" (corr_word_QE) //Input operands
: "%%0" //Clobbered registers
);
// Update correlation accumulator for IP
//  NOTE: This may not be necessary in the final implementation
asm ( "popc %%0 %2;"          // %%0 = popc(corr_word_IE)
"add   %1 %1 %%0;"      // I_early_k[sat] = I_early_k[sat] + %%0
"xori  %2 %2 0xffff;"   // corr_word_IE = ~corr_word_IE
"popc %%0 %2;"          // %%0 = popc(~corr_word_IE)
"sub   %0 %1 %%0;"      // I_early_k[sat] = I_early_k[sat] - %%0
: "=g" (I_prompt_k[sat]) //Output operands
```

```
: "g" (I_prompt_k[sat]), "g" (corr_word_IP) //Input operands
: "%%0" //Clobbered registers
);
// Update correlation accumulator for QP
//  NOTE: This may not be necessary in the final implemention
asm ( "popc %%0 %2;"          // %%0 = popc(corr_word_IE)
"add   %1 %1 %%0;"      // I_early_k[sat] = I_early_k[sat] + %%0
"xori  %2 %2 0xffff;"   // corr_word_IE = ~corr_word_IE
"popc %%0 %2;"          // %%0 = popc(~corr_word_IE)
"sub   %0 %1 %%0;"      // I_early_k[sat] = I_early_k[sat] - %%0
: "=g" (Q_prompt_k[sat]) //Output operands
: "g" (Q_prompt_k[sat]), "g" (corr_word_QP) //Input operands
: "%%0" //Clobbered registers
);
// Update correlation accumulator for IL
asm ( "popc %%0 %2;"          // %%0 = popc(corr_word_IE)
"add   %1 %1 %%0;"      // I_early_k[sat] = I_early_k[sat] + %%0
"xori  %2 %2 0xffff;"   // corr_word_IE = ~corr_word_IE
"popc %%0 %2;"          // %%0 = popc(~corr_word_IE)
"sub   %0 %1 %%0;"      // I_early_k[sat] = I_early_k[sat] - %%0
: "=g" (I_late_k[sat]) //Output operands
: "g" (I_late_k[sat]), "g" (corr_word_IL) //Input operands
: "%%0" //Clobbered registers
);
// Update correlation accumulator for QL
asm ( "popc %%0 %2;"          // %%0 = popc(corr_word_IE)
"add   %1 %1 %%0;"      // I_early_k[sat] = I_early_k[sat] + %%0
"xori  %2 %2 0xffff;"   // corr_word_IE = ~corr_word_IE
"popc %%0 %2;"          // %%0 = popc(~corr_word_IE)
"sub   %0 %1 %%0;"      // I_early_k[sat] = I_early_k[sat] - %%0
: "=g" (Q_late_k[sat]) //Output operands
: "g" (Q_late_k[sat]), "g" (corr_word_QL) //Input operands
: "%%0" //Clobbered registers
);
```

H List of Tables

List of Tables

I List of Figures

List of Figures