Spring 2019

A study of the effect of memory system configuration on the power consumption of an FPGA processor

Adam Blalock

A Study of the Effect of Memory System Configuration on the Power Consumption of an FPGA Processor

An Honors College Project Presented to

the Faculty of the Undergraduate

College of Integrated Science and Engineering

James Madison University

by Adam P. Blalock

May 2019

Accepted by the faculty of the Department of Computer Science, James Madison University, in partial fulfillment of the requirements for the Honors College.

FACULTY COMMITTEE:	HONORS COLLEGE APPROVAL:
Project Advisor: Dee A. B. Weikle, Ph.D. Associate Professor, Computer Science	Bradley R. Newcomer, Ph.D., Dean, Honors College
Reader: Michael O. Lam, Ph.D. Assistant Professor, Computer Science	
Reader: Michael S. Kirkpatrick, Ph.D. Associate Professor, Computer Science	

PUBLIC PRESENTATION

This work is accepted for presentation, in part or in full, at Computer Science Department Research Seminar on April 19, 2019.

Contents

List of Figures and Tables

Acknowledgements

I would like to thank Dr. Dee Weikle for inspiration and for help with planning and scheduling, Dr. Michael Lam and Dr. Michael Kirkpatrick for advising and for proofreading written materials, and my friends and family for their frequent emotional support and encouragement. Funding was provided by the Spring 2018 CISE Scholarship and Teaching Grant awarded to Dr. Kirkpatrick, Dr. Lam and Dr. Weikle for equipment, including PYNQ-Z1 FPGA boards.

Abstract

With electrical energy being a finite resource, feasible methods of reducing system power consumption continue to be of great importance within the field of computing, especially as computers proliferate. A victim cache is a small fully associative cache that “captures” lines evicted from L1 cache memory, thereby reducing lower memory accesses and compensating for conflict misses. Little experimentation has been done to evaluate its effect on system power behavior and consumption. This project investigates the performance and power consumption of three different processor memory designs for a sample program using a field programmable gate array (FPGA) and the Vivado Integrated Development Environment. One design has no caching whatsoever, one utilizes separate direct-mapped L1 instruction and data caches, and the last utilizes both direct-mapped L1 and smaller fully associative victim caches for both instructions and data. Each of these was given the same simple testbench program, compiled from C, disassembled, and translated into RISC-V machine code. The number of clock cycles for execution and power estimations provided by the Xilinx Vivado Integrated Development Environment were compred for a testbench program. The ratio of power over time showed a significant benefit in both power consumption and performance for the system with ony L1 caches, not not an overall benefit from including victim caches. However, other instruction streams that cause more conflict misses may still benefit.

1. Introduction

With electrical energy being a finite resource, feasible methods of reducing system power consumption continue to be of great importance within the field of computing, especially as computers proliferate. Energy-saving solutions exist for users, such as powering down when not in use, or the replacement of Cathode Ray Tube monitors with less power-hungry Liquid Crystal Displays. Timed auto-dimming, brightness sliders and sleep mode are ubiquitous in portable devices or monitors. Some machines have power vs. performance sliders built into their operating systems.

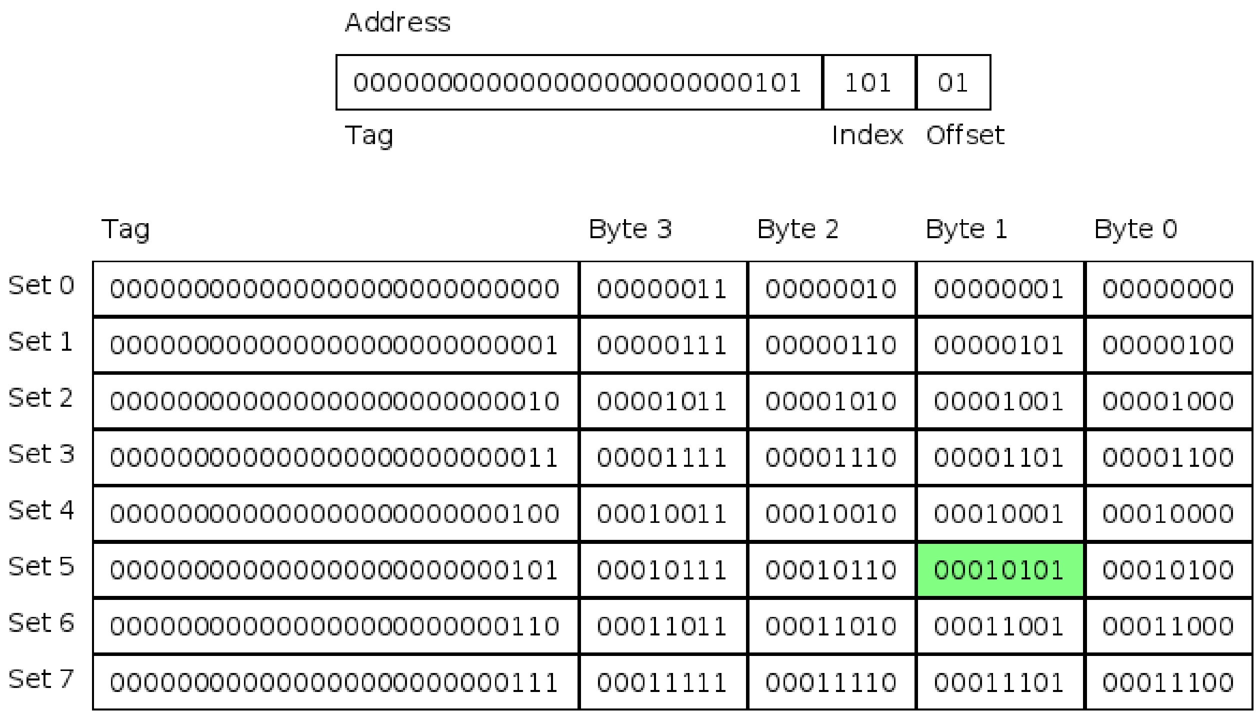
In the last decade, hardware has benefitted from experiments with transistor and materials design that have yielded both improved system performance at lower operating voltages and methods of balancing lower power and higher performance based on needs. Architecture techniques such as “drowsy-caching” and “sub-banking” with branch predictors [6] and its resources show that innovative processor designs can reduce energy consumption and potentially improve performance.

1.1 Caching

A large part of energy expenditure within computer systems is due to delays between communicating system components. The Central Processing Unit (CPU) of a computer system operates on data held in system memory, but affordable memory technologies often work at a substantially slower rate, leaving the CPU doing practically nothing while waiting for the requested data. Faster memory technologies such as Static Random Access Memory (SRAM) close the timing gap, but are substantially more expensive. To get more performance for less cost, smaller, faster memories closer to the CPU hold data from larger, slower ones which are further away, forming a memory heirarchy. Most of the time programs execute from the smaller memory, exhibiting high locality. The smaller SRAM-based memories usually reside in the CPU, are known as caches, and hold several multiple-byte copies of data from lower levels known as "lines." When the CPU requests to operate on data from a specific address in memory, a controlling circuit for the closest cache looks for a copy of the requested data. If it is not found, an event known as a cache miss, it forwards the request to the next level down and so on. Once found, an event known as a cache hit, the data is sent back up as each memory unit in the chain makes its own local copy for future use. The smaller size of caches means that existing lines must be replaced to make room for new ones, and the replaced lines may have been changed as part of execution. To preserve consistency, in most caching implementations lines are copied back down to lower levels upon being replaced. By using caches, immediately relevant data is made faster for the CPU to obtain, and reduces the time retrieving less relevant data from main memory. The hierarchy is known to benefit systems by saving time, but its effect on energy consumption is trickier. Each memory layer is another piece of hardware, so consideration must be given as to whether or not the energy expended by the additional hardware is less than the energy lost to CPU idle time.

One factor that affects the performance and usefulness of a cache is its associativity. Cache lines are held in units called sets. Caching divides binary request addresses into three segments. The "offset" segment indicates which byte in a line is requested and is comprised of the rightmost $\log_2$(bytes per line) bits of the address. The "index" segment determines which set a given line must reside in, and is comprised of the next $\log_2$(sets in cache) bits of the address. The "tag" segment is used to uniquely identify the lines in a given set, and is comprised of the remaining bits. Direct-mapped caches hold exactly one line per set, whereas an "n"-way associative cache holds "n" lines per set. Direct-mapped caches utilize simpler designs, but in the case that two frequently referenced addresses belong to the same set, they may be copied in and out over and over, a phenomenon known as "thrashing" that wastes a lot of time and subsequently power. By holding multiple lines, associative caches are less prone to thrashing, but depending on the associativity and policy for choosing which line in a given set to replace the designs can be substantially more complex and slower than their direct-mapped counterparts.

Figure 1 demonstrates a direct-mapped cache access. The cache has four-byte lines and eight sets, therefore the address is divided into a 2-bit offset, 3-bit index and 27-bit tag. The index 101 indicates set five as the search set. Since the tag in set 5 matches, it is a cache hit and the byte at offset 1 is retrieved.



Address		
000000000000000000000000101	101	01
Tag	Index	Offset

	Tag	Byte 3	Byte 2	Byte 1	Byte 0
Set 0	000000000000000000000000000	00000011	00000010	00000001	00000000
Set 1	000000000000000000000000001	00000111	00000110	00000101	00000100
Set 2	000000000000000000000000010	00001011	00001010	00001001	00001000
Set 3	000000000000000000000000011	00001111	00001110	00001101	00001100
Set 4	000000000000000000000000100	00010011	00010010	00010001	00010000
Set 5	000000000000000000000000101	00010111	00010110	00010101	00010100
Set 6	000000000000000000000000110	00011011	00011010	00011001	00011000
Set 7	000000000000000000000000111	00011111	00011110	00011101	00011100

Figure 1. Direct-mapped Cache Hit Demonstration

A victim cache, proposed in [5], is a relatively small but fully associative cache placed between the highest "L1" cache and lower memory layers. It "captures" lines evicted from L1 cache memory, thereby reducing lower memory accesses and compensating for conflict misses. Their small size in comparison to other cache layers make them worth exploring for use in both high-performance and embedded systems. Victim caching has been shown to improve performance, yet little experimentation has been done to evaluate its effect on system power behavior and consumption. This project implements two memory heirarchy designs, one with a victim cache and one without in a Field Programmable Gate Array (FPGA) to gain instight into this question.

1.2 Project Goals

The intent is to estimate how the use of caching affects power consumption when applied to FPGA-based RISC-V based Systems on a Chip (SoC). RISC-V is an open-source Instruction Set Architecture (ISA), a standardized specification for the binary machine code instructions a CPU is capable of decoding and executing. [9] Three SoCs are compared, one with no caching, one with separate L1 instruction and data caches, and one with L1 and victim instruction and data caches, all designed in Verilog. After simulating and debugging these designs, power estimation tools are used to compare performance and power behavior for a given RISC-V program.

2. Related Work

Canturk Isci and Margaret Martonisi [4] propose a power measurement framework adapted from phase-based performance analysis. The instrumentation tool Pin was used to dynamically inject the SPECCPU 2000 benchmark suite with calls to an analysis program known as a pintool. The CPU running these benchmarks, an Intel Pentium 4 with a Linux Kernel, housed a unit for measuring the frequency of events such as executions and cache accesses. Additionally, a current measurement probe was placed on the CPU and the value fed back in as an input. When called, the pintool used buffered samples of the performance counters and probe measurements to create a sample power history that accounted for the shift in control. Execution samples were compared for similarity in order to classify distinct execution phases for which power could be estimated.

In a follow-up paper [3], Isci and Martonosi also described a methodology for measuring power use for individual components. A Fluke-brand ammeter and Agilent-brand digital multimeter were used to measure power over time during the execution of similar microbenchmarks, with a sample rate of 1000 readings per second. Access rate heuristics were derived for individual components within the CPU. L1 cache access rate was defined a function of port replays, front end events and clock cycles. Power per component could then be calculated as a function of access rate, architectural scaling, estimated maximum power and estimated non-gated clock power.

In [6] Kim, Flautner, Blaauw and Mudge consider the effects of transistor leakage on overall power consumption, noting it as the primary cause of power expenditure in caches. Reducing threshold voltage is known to reduce both leakage and performance. The report details experiments with "drowsy caches," in which lines expected to be accessed less frequently are put in a lower-voltage state until further notice, reducing leakage temporarily while preserving values. The paper evaluates via simulation several drowsy prediction techniques for groups of instruction cache lines called sub-banks. Accuracy is measured as a function of correct predictions divided by the number of "wake-ups."

The design of the memory controllers in this report are partially based on examples given in [1]. The design of the RISC-V core and caching systems are partially based on examples given in [7].

3. Methodology

3.1 Hardware Description Languages and FPGAs

Hardware Description Languages (HDL) are formal programming languages used to create human-readable descriptions of computational circuitry. Verilog is a popular HDL based on C, in which designs are created by defining hardware units called "modules," their synchronous or asynchronous sequential or combinational logic, memory registers, inputs, outputs, sub-modules, the wires that connect them together, and a "top module" which binds everything together." Although mainly used for simulation, HDL descriptions can be translated or "synthesized" into a format usable by Field Programmable Gate Arrays (FPGA), a type of chip that can have its structure and function configured and is often used for experimenting with and simulating dedicated chip designs. The file used to program a specific FPGA with a synthesized design is called a bitstream. This configuration is volatile, meaning the bitstream must be re-loaded each time the chip is powered on. FPGA manufacturer Xilinx provides an Integrated Development Environment (IDE) called Vivado for writing HDL and compiling (synthesizing) it for use with their boards. One helpful feature of Vivado is the ability to abstract part of the creation process via block diagrams, where modules are represented visually as blocks with I/O ports. Ports can be connected by drawing lines between them, and Vivado will automatically create a wrapper module based on the diagram. See the appendix for examples of how block diagrams help the design process.

All HDL code for this project is developed in the Xilinx Vivado Integrated Development Environment (IDE), written in Verilog and simulated with the built-in behavioral simulator. The experiment consists of designing three variations of a memory hierarchy system and connecting them to the same simple RISC-V-based core to compare power/performance behavior among the three.

3.2 BabyRisc

The common RISC-V core is a simplified 32-bit implementation with limited instruction support and no pipelining. Because of its limited functionality, it has been given the name “BabyRisc.” As of this writing, BabyRisc only supports aligned loads and stores, doubleword size Arithmetic Logic Unit (ALU) operations, branches, and a custom halt instruction comprised of all zero bits. It consists of an instruction decoder, register file, ALU, program counter unit, and a clock-sensitive main controller. (See Figure 2) The controller is a positive edge-triggered eight-state machine with states “ready,” “fetch,” “decode,” “calculate,” “memory,” “writeback,” “done” and “error.” and transitions between states taking place at the positive edge of a periodic clock signal given certain conditions (see Figure 3 for a state transition diagram). These states represent the basic loop every CPU performs in executing a program: retrieving the next instruction from memory, determining the action to take given the instruction, performing a mathematical calculation, reading or writing to external memory, and writing results back to internal memory units called “registers.” In the “ready” state, this controller awaits a start signal before transitioning into the “fetch” state, and continues the loop until a halt instruction is encountered or an invalid instruction is read. Each state has a corresponding component, and signals are sent upon entering states to trigger their operation. For example, the decoder component reads the incoming instruction to determine how its outputs should be set, but only sets them when the controller has entered the decode state. BabyRisc utilizes a byte-aligned 32-bit address space. Memory accesses begin with a request signal to an external memory controller and end when a response signal is returned. The req_type signal is used so that the cache memory controller can distinguish between instruction and data requests. This is important as each request type causes the controller to behave differently by branching into type-exclusive states. (See Figures 6 and 8)

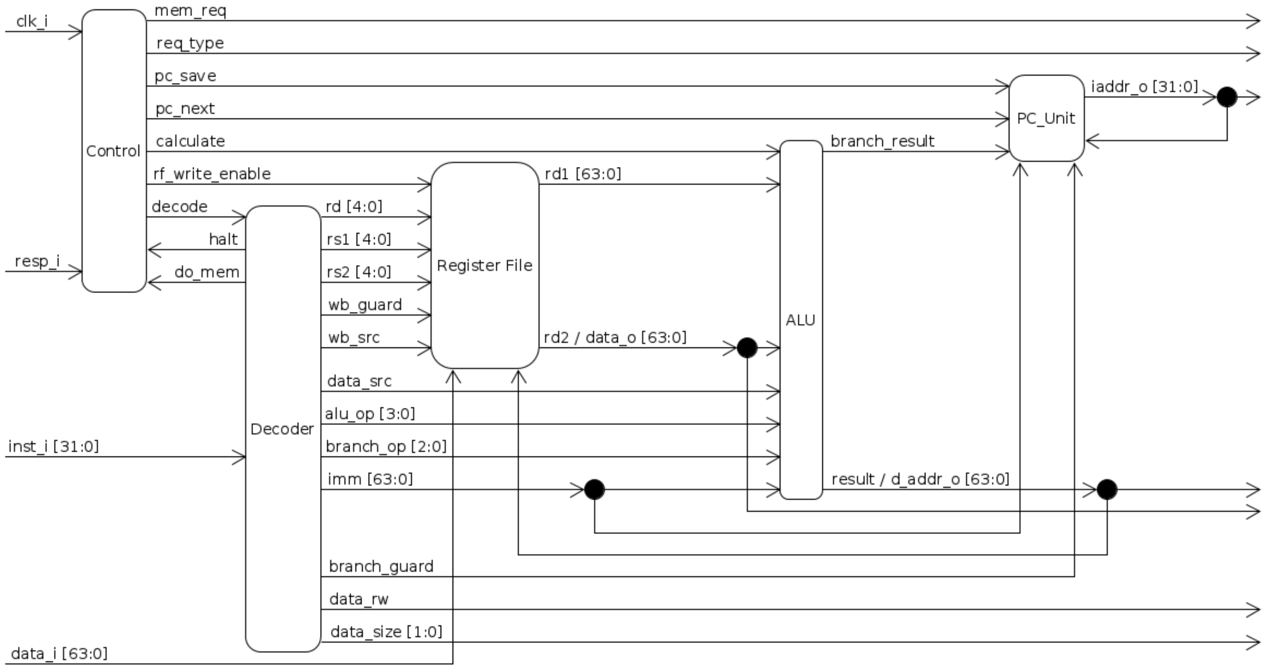


Figure 2. A Block Diagram Demonstrating the Design of BabyRisc

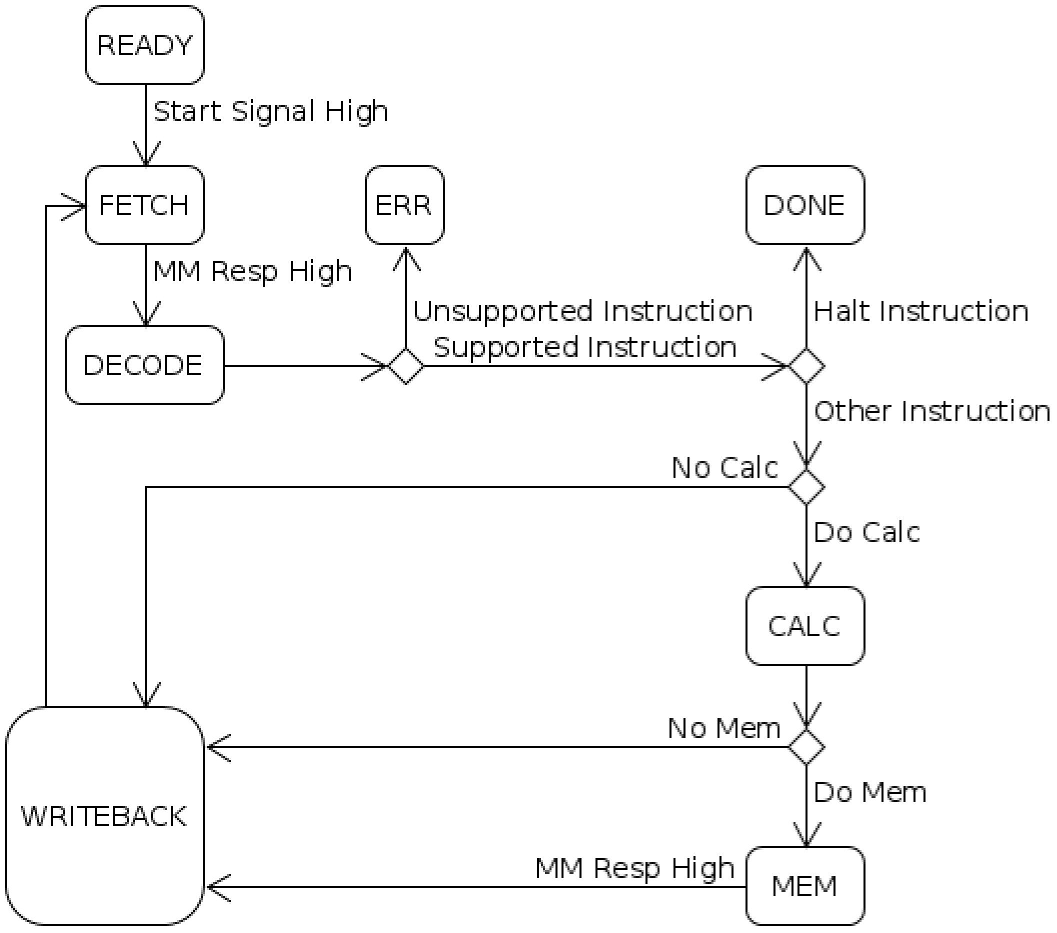


Figure 3. A State Machine Diagram for BabyRisc's Main Control Unit

3.3 Design 1: Main Memory Only

The main memory modules of the SoC behave similar to a real-world Random Access Memory (RAM). Upon receiving a request signal, the RAM writes incoming data to the specified address if necessary, sets the read data on output wires, and sets a response signal. However, as discussed earlier, in real-world systems the time to access main Dynamic RAM is usually in a magnitude four to five times greater than that required to access Static RAM-based caches due to differences in hardware speed. A raw FPGA implementation would eliminate this delay since every component uses the same hardware. To simulate this delay, the RAM modules also act like state machines so that they require eight clock periods per operation instead of the two to four required for cache operations. Because BabyRisc outputs instruction and data addresses as distinct signals, for simplicity's sake the main memory has address inputs and value outputs for both.

The following block diagram demonstrates a BabyRisc system with main memory and no caching. The request type signal 'req_o' is not used in this design, as it is intended for use by a cache controller which is not present.

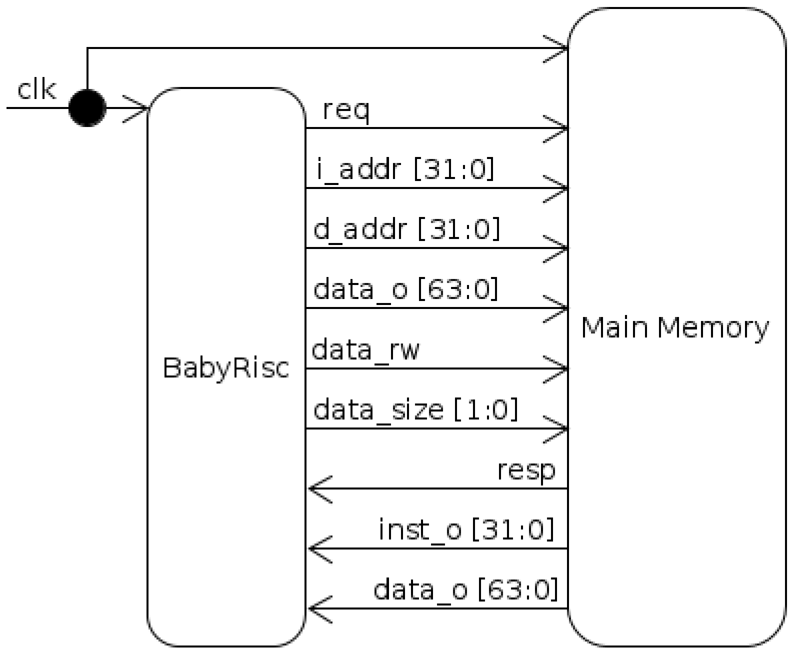


Figure 4. A Block Diagram Demonstrating a BabyRisc System with Main Memory

3.4 Design 2: L1 Caching and Main Memory

The second design uses shared main memory and distinct caches for instructions and data. The L1 cache modules are direct mapped with a 22-bit tag, 6-bit index and 4-bit offset for a total of 1024 bytes each. Both have inputs for address, write enable, and data from main memory, and outputs for combined tag and index for use in cache misses, hit/miss signal, and data. The data cache has additional inputs for write mode (regular or full line) and size (byte, half, word double), and an additional output signal for writeback if the set already contains valid data that must be written to lower memory layers upon replacement. The cache controller for this design does not handle data or address information itself, but progresses from IDLE through the appropriate sequence of states based on signals from the caches and main memory. Output signals from the controller correspond with specific states. (See Figure 5)

For the two designs in which caching is utilized, the main memory does not accept full 32-bit addresses, but instead accepts the 28-bit combined tag and index and operates on full 16-byte lines. Other than this, the function of this line-addressed memory is nearly identical to that of the byte-addressed memory.

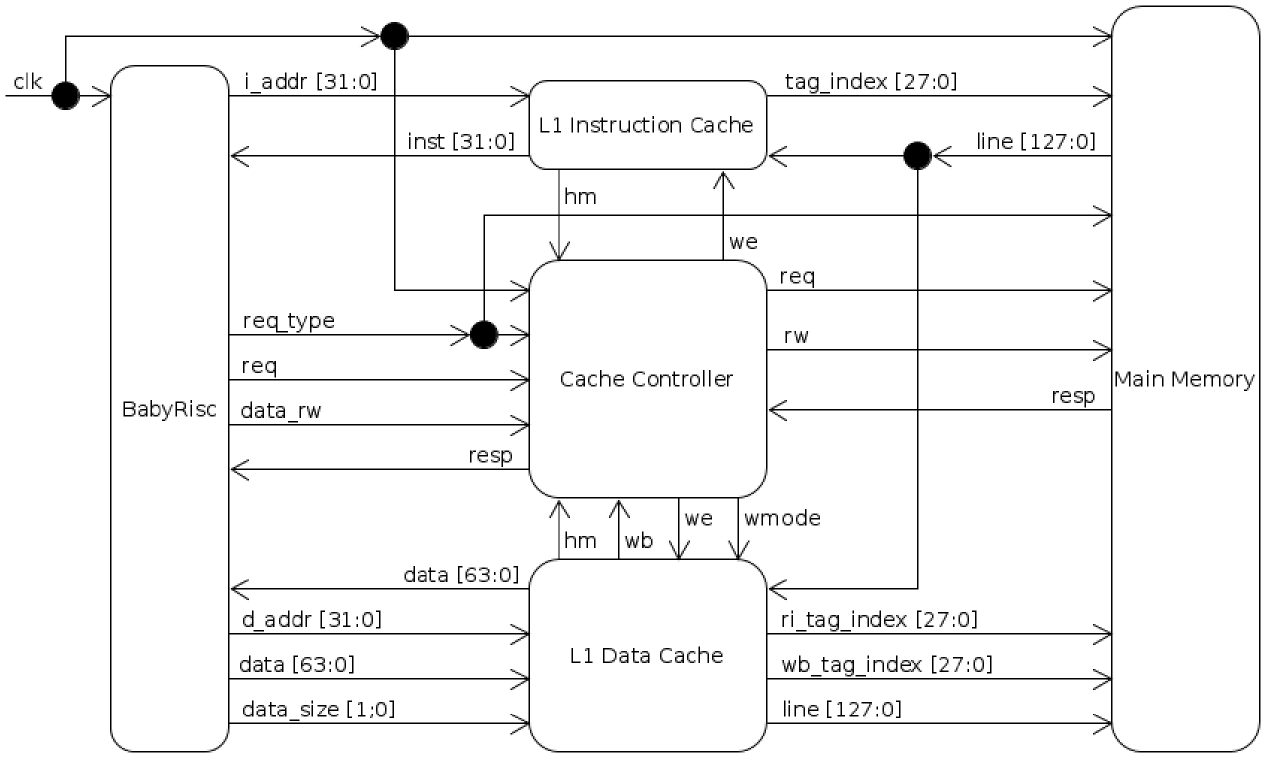


Figure 5. Block Diagram for a BabyRisc System with Main Memory and L1 Caching.

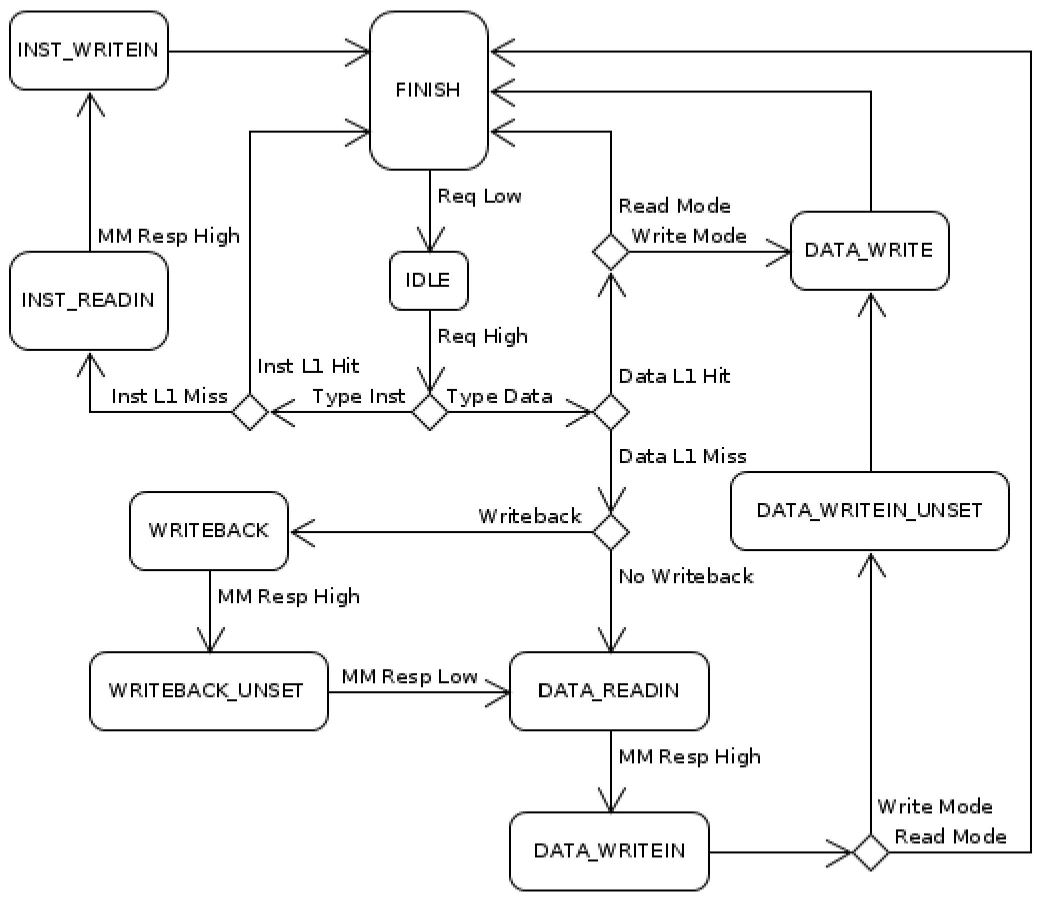


Figure 6. State Machine Diagram for the Cache Memory Controller in Figure 5.

3.5 Design 3: L1 and Victim Caching

The third design uses shared main memory, L1 caches and victim caches. The victim cache modules hold the 8 most recently evicted cache lines, which can be indexed and accessed in an arbitrary order, and are replaced on a least-recently-used basis. This is done via a queue that can be added to at the front and removed from at an arbitrary location. The L1 and victim caches are checked simultaneously, and to mimic real-world systems, upon a victim cache hit lines must be “swapped” back into the L1 cache before any further reads or writes can be performed. Any lines that are replaced in the victim cache but not swapped back into L1 are written back to main memory.

The L1 caches work slightly differently to accommodate for this. Both data and instruction L1 caches receive an address from the BabyRisc core, parsing and passing the tag and index to the victim caches and main memory. Since the L1 and victim caches must be able to exchange data between each other simultaneously, the lines being written in must not change during the operation. To achieve this, both L1 and victim caches accept a set_swap input signal, which causes the swap lines to be saved to output registers. Simultaneous write signals are asserted one cycle later to initiate the swap.

In the case that there is a miss in both caches, writeback from victim cache, eviction from L1 cache and read-in from main memory occur in respective order, skipping steps when not applicable. A data writeback is only necessary if the victim cache is completely full, and an eviction from the L1 cache is only necessary if the target set contains valid data. A read-in from main memory is always necessary (See figure 8).

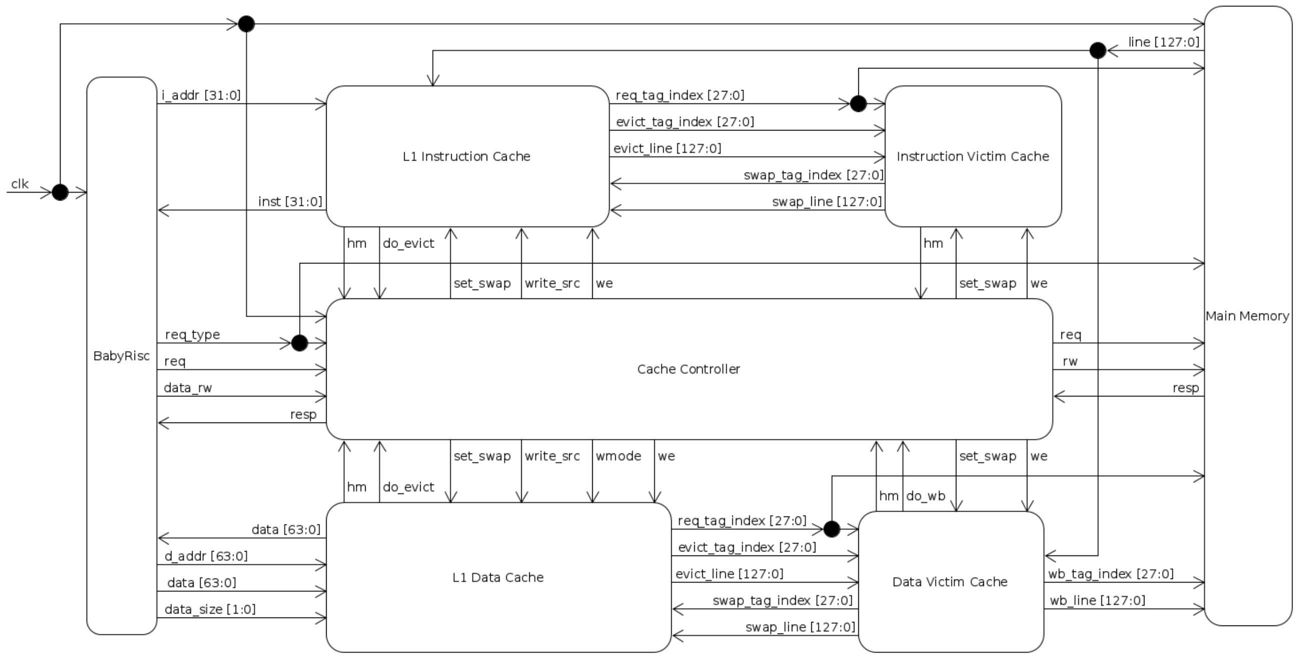


Figure 7. Block Diagram for a BabyRisc System with Main Memory,

L1 Caching and Victim Caching.

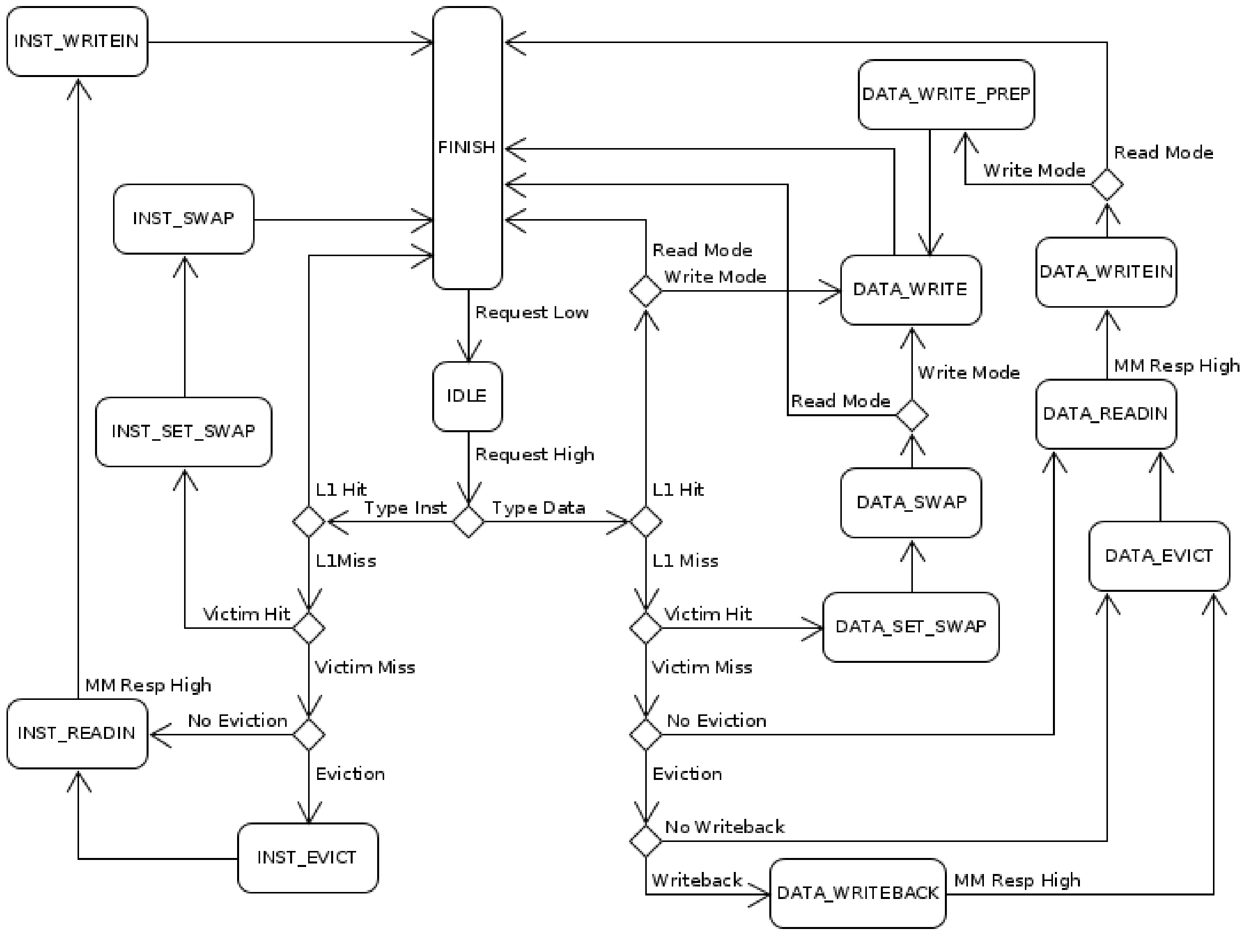


Figure 8. State Machine Diagram for the Cache Memory Controller in Figure 7.

The following is a Vivado behavioral simulation timing diagram demonstrating a successful data request that misses in the L1 cache but hits in the victim cache. When the request from BabyRisc is sent (mem_req), a hit in the victim cache has already been found. A few cycles of the clock later, the L1 and victim caches are set to swap lines (set_swap) and those lines are exchanged. Afterwards, the hit is in the L1 cache (dmc_hm) and a response is sent from the memory controller one cycle later.

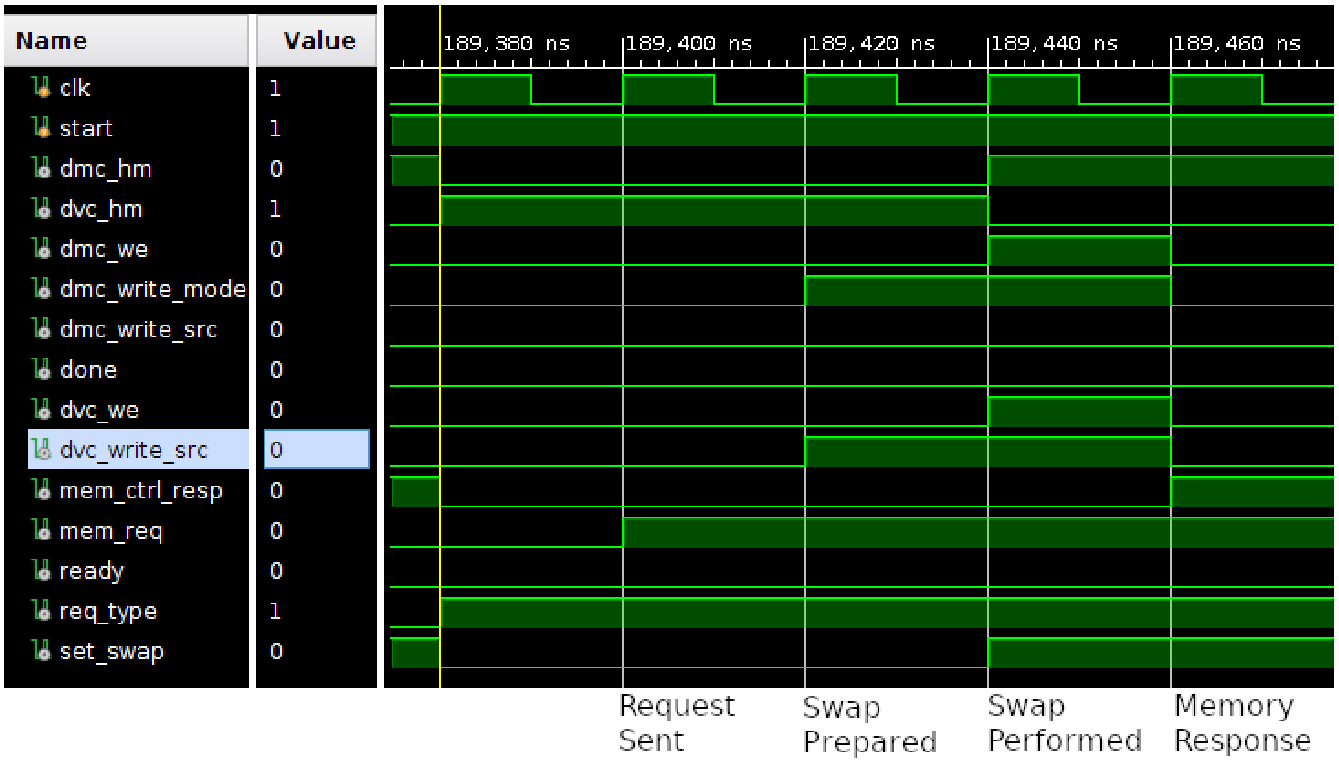


Figure 9. Timing Demonstration of a Victim Cache Swap.

3.6 Testbench and Program

Testing the given designs requires two things: a program to run and a testbench file to drive execution. BabyRisc begins program execution by fetching and executing the 32-bit instruction stored at address 0x0 of memory, then sequentially until the end of execution, the exception being when execution causes the address of the next instruction to change as part of a decision, eg. a jump. Each testbench program is hard-coded into the main memory modules. For example, the following are a few RISC-V instructions written in assembly language, followed by the binary representation read by the CPU.

```
addi  sp,   sp,   -1584    10011101000000010000000100010011
sd    s0,   1576(sp)       01100010100000010011010000100011
addi  s0,   sp,   1584     01100011000000010000010000010011
```

This binary code is hard-written into the main memory module with the first instruction starting at address 0x0. As stated before, one main memory design is byte-addressed to 32-bit addresses, while the other is line-addressed to 28-bit combined tags and indexes. Instantiating the former memory with this instruction for behavioral simulation takes the form

```
data[3] = 'b00000000;
data[2] = 'b00000000;
data[1] = 'b00000010;
data[0] = 'b10110011;
```

Whereas instantiating the same instruction into the latter takes the form

```
lines[0] = {32'b00000000000100000000111000010011,
            32'b00000000000000000000001110110011,
            32'b00010100000000000000001100010011,
            32'b00000000000000000000001010110011};
```

The program run for this experiment is a simple Fibonacci sequence calculator which calculates the first 198 numbers and places them into memory. It is written in the C language and compiled using the 32-bit gcc-based compiler provided in the RISC-V organization’s official toolchain.

```
void main() {

          #include <stdbool.h>
          #include <stdint.h>
          #define size 192

          uint64_t a = 0;
          uint64_t b = 1;
          uint64_t c = 1;
          uint64_t nums [size];

          uint64_t i = 0;
          while(true) {
              nums[i] = c;
              i += 1;
              if (i == size) break;
              a = b;
              b = c;
              c = a + b;
          }
      }
```

Running the toolchain's object dumper shows the program's corresponding assembly representation. The relevant instructions are copied, modified, translated into machine code and placed into the main memory module via Verilog statements like the examples above.

```
addi    sp, sp, -1584
sd      s0, 1576(sp)
addi    s0, sp, 1584
sd      zero, -48(s0)
addi    a5, zero, 1
sd      a5, -24(s0)
addi    a5, zero, 1
sd      a5, -32(s0)
sd      zero, -40(s0)
ld      a5, -40(s0)
slli    a5, a5, 3
addi    a4, s0, -16
add     a5, a5, a4
ld      a4, -32(s0)
sd      a4, -1568(a5)
ld      a5, -40(s0)
addi    a5, a5, 1
sd      a5, -40(s0)
ld      a4, -40(s0)
addi    a5, zero, 192
beq     a4, a5, 40
ld      a5, -24(s0)
sd      a5, -48(s0)
ld      a5, -32(s0)
sd      a5, -24(s0)
ld      a4, -48(s0)
ld      a5, -24(s0)
add     a5, a5, a4
sd      a5, -32(s0)
beq     zero, zero(-80)
ld      s0, 1576(sp)
addi    sp, sp, 1584
halt
```

Execution requires use of the Vivado IDE built-in simulator, used for testing Verilog designs and utilizing three different modes. For behavioral mode, code is interpreted as written and all assignment statements are treated as instantaneous. The two post-synthesis modes take actual hardware delay into consideration, but both are ignored in favor of behavioral since using this mode is outside of the host computer's capability and anticipated to produce little to no additional useful information. Vivado simulation requires the creation of testbench files, which wrap the top level modules of Verilog designs and allow users to manipulate and monitor input and output over time. The testbench for this experiment wraps all three SoC designs at once, providing each with a 50MHz clock and start signal while monitoring the ready, done and error outputs. A register also counts the number of cycles passed since the start signal was sent so that the clock cycles for completion can be compared between each design. Time measurement for completion is taken from the beginning of simulation to the time when the "done" signal is asserted by each SoC.

3.7 Power Estimation

Vivado also provides a built-in post-synthesis power estimation tool. By accounting for given variables, such as ambient temperature, airflow, and input voltage, an estimate of power consumption in watts when programmed onto an FPGA can be made. This estimation which can be aggregated on a module-by-module basis for both static and dynamic components. For higher accuracy, Vivado can generate usage vector files (.saif) during post-synthesis simulation which can be applied to power estimation for improved accuracy. Due to Verilog infeasibly long post-synthesis simulation times, however, vector-based estimation is left for future work. More information on Vivado power reporting can be found at [10]. See the appendix for a summary of the settings used across all power estimations. With the exception of the clock period, these are all the default settings suggested by the Vivado estimation tool.

4. Results

Figure 10 is an abridged timing diagram for running the Fibonacci program on each of the three SoCs in behavioral mode, from slightly before the start signal is asserted to slightly after the done signals of all three are asserted. “Clk,” “rst_sig,” and “start_sig,” are inputs from the testbench to the SoC designs; the “done,” “err,” and “ready” wires are outputs. “Clk_en” and “cycle_count” are used exclusively by the testbench to enable the clock and count the number of cycles since starting respectively. “Done1” corresponds to the design with no caching system in place, “done2” to the design with L1 caches, and “done3” to the design with L1 and victim caches. The same applies for the ready and error signals. The diagram’s scale gives the illusion that “clk” and “cycle_count” remain constant, but in truth they change very frequently making them appear homogenous.

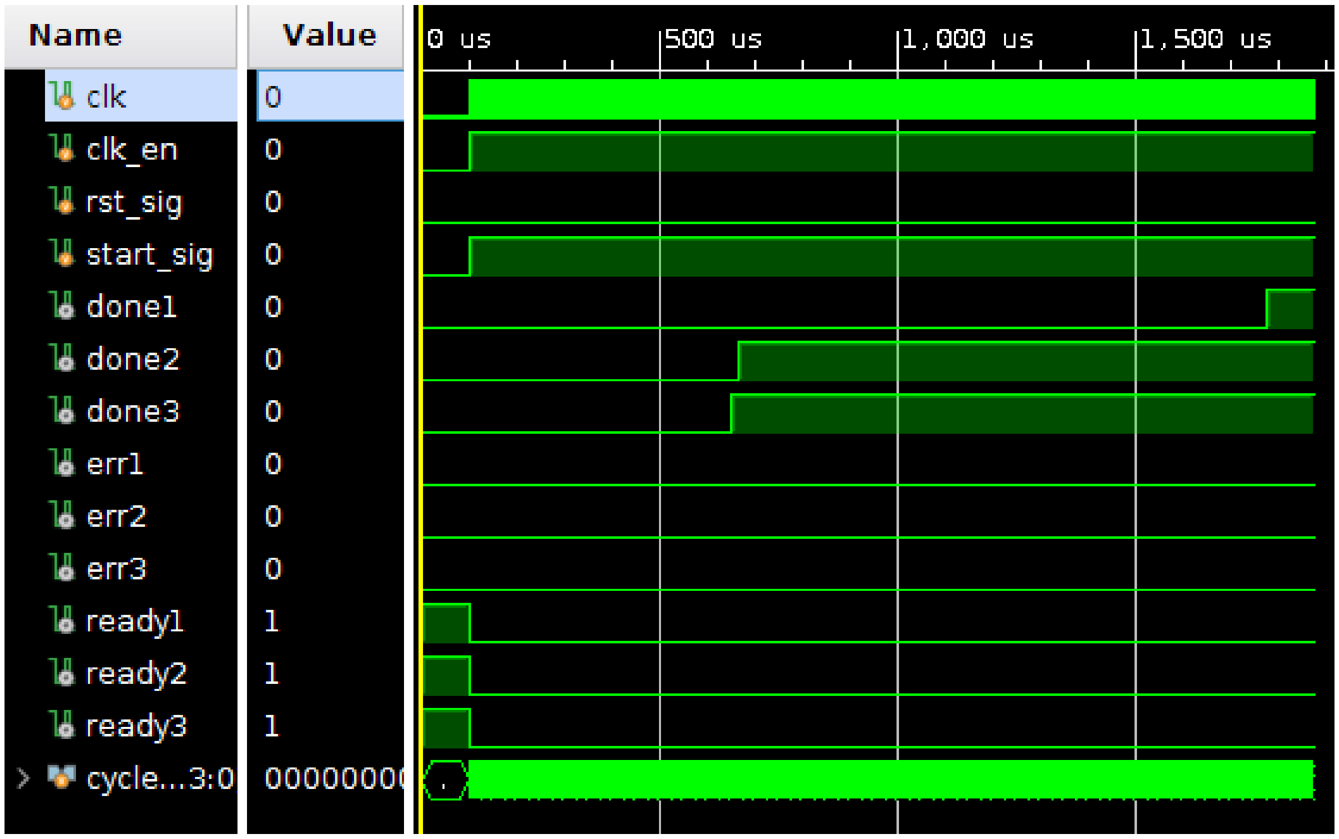


Figure 10. Vivado Behavioral Simulation Diagram.

Table 1 summarizes the number of clock cycles and execution time in milliseconds taken from the assertion of "start" to the assertion of each respective "done" signal, assuming a clock period of 20ns corresponding to a 50MHz clock.

Memory Layout	**Cycles to Completion**	**Time to Completion**
Main Memory Only	39388	1.680 ms
L1 Caching	15066 (24322 less)	0.566 ms
L1 and Victim Caching	14874 (192 less)	0.551 ms

Table 1. Simulation Cycles and Time to Completion

Performance comparisons are evident: The design with L1 and Victim Caching required fewer cycles and therefore less time to execute, although not by a significant degree for this program. This is likely due to the nature of the program used for testing, which has only three local variables that are referenced frequently. A closer observation of simulation showed that lines were recovered from the victim cache only ten times.

For power estimation, each design was synthesized and run through individually instead of all at once. Tables 2-5 are a breakdown of power reporting results for each design using the default settings. The first entries in each table are measurements for each module arranged in order from lowest to greatest power. The last three entries on any table list the total dynamic power, the total static power, and the combined total. Dynamic power is a measurement of power spent when a transistor moves from a high state to a low state and vice-versa, whereas static power is a measure of leakage when a transistor is not changing state. A higher ratio of dynamic over static power is a ratio of useful power over wasted power, therefore based on power estimation the design with victim caches is seemingly more power efficient in the average case.

Module Name / Variable	**Power**	**Percentage of Total**
BabyRisc	0.242 W	11.86 %
Regular Main Memory	1.649 W	80.79 %
Dynamic Power	1.893 W	92.75 %
Device Static Power	0.148 W	7.25 %
Total On-Chip Power	2.041 W	100.00 %

Table 2. Power Reporting Results for the Main Memory Only Design

Module Name / Variable	**Power**	**Percentage of Total**
Memory Controller	< 0.001 W	< 0.003 %
Instruction L1 Cache	0.013 W	3.94 %
Caching Main Memory	0.030 W	9.09 %
BabyRisc	0.046 W	13.94 %
Data L1 Cache	0.133 W	40.30 %
Dynamic Power	0.222 W	67.27 %
Device Static Power	0.108 W	32.73 %
Total On-Chip Power	0.330 W	100.00 %

Table 3. Power Reporting Results for the L1 Caches Only Design

Module Name / Variable	**Power**	**Percentage of Total**
Memory Controller	< 0.001 W	< 0.283 %
Caching Main Memory	0.003 W	0.85 %
Instruction Victim Cache	0.017 W	4.81 %
Data Victim Cache	0.018 W	5.10 %
Instruction L1 Cache	0.023 W	6.51 %
BabyRisc	0.053 W	15.01 %
Data L1 Cache	0.130 W	36.83 %
Dynamic Power	0.245 W	69.40 %
Device Static Power	0.108 W	30.59 %
Total On-Chip Power	0.353 W	100.00 %

Table 4. Power Reporting Results for the L1 and Victim Caches Design

Module Name	**Main Memory Only**	**L1 Caching**	**L1 & Victim Caching**
BabyRisc	0.242 W	0.046 W	0.053 W
Main Memory	1.649 W	0.030 W	0.003 W
Memory Controller	N/A	< 0.001 W	< 0.001 W
Instruction L1 Cache	N/A	0.013 W	0.023 W
Data L1 Cache	N/A	0.133 W	0.130 W
Instruction Victim Cache	N/A	N/A	0.017 W
Data Victim Cache	N/A	N/A	0.018 W
Dynamic Power	1.892 W	0.222 W	0.245 W
Device Static Power	0.148 W	0.108 W	0.108 W
Total On-Chip Power	2.041 W	0.330 W	0.353 W

Table 5. A Comparison Between Similar Modules in each Design

First of note is the much higher power estimate for the main memory only design. The main memory module of the first design is responsible for approximately 80% of its power consumption. In the other two designs, the line-addressed main memory module contributes substantially less, and the data caches contribute substantially more. This could be due to main memory being called upon much less frequently in the latter two designs, with the burdens being placed on the data caches. It could also be due to a disjoint in complexity between the two main memories' designs - a memory comprised of 2048 single-byte registers is perhaps more complex than one comprised of 128 16-byte registers. However, observing the estimated behavior of BabyRisc brings both of these theories into question. The above theories do not explain the variance in BabyRisc's power behavior despite its uniform design across all three SoCs. The fact that the BabyRisc instance in the third design has a higher estimated power than the one in the second also implies that this difference may not be a matter of access frequency or time until completion; if that were the case the second design would perhaps have the greater power. The exact cause of this difference will require further testing, and is left for future work.

Between the two designs that utilize caching, the one with victim caches is estimated to use slightly more power. Notable, however, is that both have an identical estimated static power, yet the one with victim caching has a higher estimated dynamic power, skewing the ratio of dynamic against static in its favor.

Assuming power estimation of this accuracy produces average power at any given point in time, Table 6 shows the product of power by execution time for each design.

SoC Design	Power Estimation	Simulation Execution Time	Power * Time
Main Memory Only	2.041 W	1.680 ms	3.429
With L1 Caches	0.330 W	0.566 ms	0.187
With L1 and Victim Caches	0.353 W	0.551 ms	0.194

Table 6. Comparing the Products of Estimated Power and Execution Time

Although power efficiency may be increased on average for L1 and Victim Caching, for the Fibonacci program the product of power over execution time is in favor of the design with L1 caches only. This is likely due to the aforementioned fact that the victim cache is only accessed ten times, saving only 192 cycles. A different test program with more active variables, and therefore more opportunities to retrieve evicted lines, would likely produce results more favorable for the victim caching design.

5 Future Work

The power gaps discovered between similar modules require further investigation. Future work would first and foremost find the cause of this disparity, complexity or access frequency. Experimentation with other programs should also be considered, especially those likely to utilize the victim cache to a higher degree. One consideration was a recursive prime factorization algorithm which required instructions not yet supported by BabyRisc. The C code and assembly for this program can be found in the appendix.

Other future work could take an approach similar to that detailed in Isci and Martonosi's work [3] involving the use of power monitoring tools and usage heuristics to detail actual power behavior as opposed to simulated power behavior. Another possibility would be to improve the accuracy of Vivado power estimation with usage vectors from a post-synthesis simulation of the implemented design. Synthesizing SoC designs will require modification for size and proper post-synthesis instantiation of memories so that testing and measurement on actual hardware can be performed.

6 Conclusion

Running Vivado power estimation tools on three RISC-V SoCs with different memory hierarchy designs, one with main memory only, one with L1 data and instruction caches, and one with L1 and victim caches, showed the latter design to have a favorable dynamic to static power ratio. Although this was not reflected in the execution of a Fibonacci sequence calculator due to its low use of the victim cache, future work could improve the design of the hardware components, generate more accurate power estimation via post-synthesis usage vectors, and test programs more likely to take advantage of victim caching.

A Verilog, Block Diagrams and Simulation Settings

This appendix contains The Verilog code, Vivado block diagrams, and power estimation settings used for design and simulation at the time of writing.

Verilog header for constants op_aliases.vh

```
parameter
    WB_SRC_ALU = 'b0,
    WB_SRC_DMEM = 'b1,

    ALU_SRC_REG = 'b0,
    ALU_SRC_IMM = 'b1,

    TYPE_R = 'b000,
    TYPE_I = 'b001,
    TYPE_S = 'b010,
    TYPE_SB = 'b011,
    TYPE_H = 'b100,
    TYPE_ERR = 'b101,

    ALU_ADD = 'b00,
    ALU_SUB = 'b01,
    ALU_SLL = 'b10,

    BRANCH_EQ = 'b000,

    OP_LOAD = 'b0000011,
    OP_ALUI = 'b0010011,
    OP_STORE = 'b0100011,
    OP_ALUR = 'b0110011,
    OP_BRANCH = 'b1100011,
    OP_HALT = 'b0000000,

    F3_ALU_ADD_SUB = 'b000,
    F3_ALU_SLL = 'b001,

    F7_ADD = 'b0000000,
    F7_SUB = 'b0100000,

    PC_OP_NEXT = 'b0,
    PC_OP_BRANCH = 'b1;
```

alu.v

```
module alu(
    input wire alu_src, calc,
    input wire [1:0] alu_op,
    input wire [2:0] branch_op,
    input wire [63:0] rdata_1, imm, rdata_2,

    output reg branch_ok,
    output reg [63:0] op_result
    );

    `include "op_aliases.vh"

    initial op_result = 0;

    wire [63:0] opdata = alu_src == ALU_SRC_IMM ? imm: rdata_2;

    always @(posedge calc)
        case (alu_op)
            ALU_ADD: op_result = rdata_1 + opdata;
            ALU_SUB: op_result = rdata_1 - opdata;
            ALU_SLL: op_result = rdata_1 << opdata;
            default: op_result = 'b0;
        endcase

    always @(*) case (branch_op)
        BRANCH_EQ: branch_ok = (op_result == 0);
        default: branch_ok = 'b0;
    endcase

endmodule
```

decoder.v

```
module decoder(
    input wire decode,
    input wire [31:0] inst_i,

    output reg alu_src, data_rw, do_mem, halt, pc_op, wb_guard, wb_src,
    output reg [1:0] data_size,
    output reg [2:0] branch_op,
    output reg [1:0] alu_op,
    output reg [4:0] rd, rs1, rs2,
    output reg [63:0] imm
    );

    `include "op_aliases.vh"

    initial begin
        alu_src = 0;
        data_rw = 0;
        do_mem = 0;
        halt = 0;
        pc_op = 0;
        wb_guard = 0;
        wb_src = 0;
        data_size = 0;
        branch_op = 0;
        alu_op = 0;
        rd = 0;
        rs1 = 0;
        rs2 = 0;
        imm = 0;
    end

    wire [2:0] funct3 = inst_i[14:12];
    wire[6:0] funct7 = inst_i[31:25];
    wire [6:0] opcode = inst_i[6:0];

    reg [2:0] itype;
    always @(*) case (opcode)
        OP_LOAD: itype = TYPE_I;
        OP_ALUI: itype = TYPE_I;
        OP_STORE: itype = TYPE_S;
        OP_ALUR: itype = TYPE_R;
        OP_BRANCH: itype = TYPE_SB;
        OP_HALT: itype = TYPE_H;
        default: itype = TYPE_ERR;
    endcase

    wire [1:0] data_size_next = funct3[1:0];
    reg [1:0] alu_op_next;
    always @(*) case (opcode)
        OP_LOAD: alu_op_next = ALU_ADD;
        OP_ALUI: case (funct3)
            F3_ALU_ADD_SUB: alu_op_next = ALU_ADD;
            F3_ALU_SLL: alu_op_next = ALU_SLL;
            default: alu_op_next = 'b0;
        endcase
        OP_STORE: alu_op_next = ALU_ADD;
        OP_ALUR: case (funct3)
            F3_ALU_ADD_SUB: case (funct7)
                F7_ADD: alu_op_next = ALU_ADD;
                default: alu_op_next = 'b0;
            endcase
            F3_ALU_SLL: alu_op_next = ALU_SLL;
            default: alu_op_next = 'b0;
        endcase
        OP_BRANCH: alu_op_next = ALU_SUB;
        default: alu_op_next = 'b0;
    endcase

    reg [63:0] imm_next;
    always @(*) case (itype)
        TYPE_I: imm_next = {{52{inst_i[31]}}, inst_i[31:20]};
        TYPE_S: imm_next = {{52{inst_i[31]}}, inst_i[31:25], inst_i[11:7]};
        TYPE_SB: imm_next = {{51{inst_i[31]}}, inst_i[31], inst_i[7],
inst_i[30:25], inst_i[11:8],
                                1'b0};
        default: imm_next = 0;
    endcase

    wire pc_op_next = itype == TYPE_SB ? PC_OP_BRANCH : PC_OP_NEXT;

    wire wb_src_next = opcode == OP_LOAD ? WB_SRC_DMEM : WB_SRC_ALU;

    always @(posedge decode) begin
        // Zero for register if one of these types, otherwise 1 for immediate.
        alu_src <= (itype != TYPE_R) && (itype != TYPE_SB);
        // One if a branching type.
        // Only asserted for stores
        data_rw <= itype == TYPE_S;
        do_mem <= (opcode == OP_LOAD) || (opcode == OP_STORE);
        halt <= opcode == OP_HALT;
        // Register writes happen for all instruction types besides S and SB.
        wb_guard <= (itype != TYPE_S) && (itype != TYPE_SB);
        // Only asserted for loads
        pc_op <= pc_op_next;
        wb_src <= wb_src_next;
        data_size <= data_size_next;
        branch_op <= funct3;
        alu_op <= alu_op_next;
        rd <= inst_i[11:7];
        rs1 <= inst_i[19:15];
        rs2 <= inst_i[24:20];
        imm <= imm_next;
     end

endmodule
```

main_control.v

```
module main_control(
    input wire clk, do_mem, halt, reg_file_ready, mem_ready, mem_resp, start_sig,

    output wire calc, decode, done_o, mem_req, mem_req_type, pc_update, ready_o,
wb_sig
    );

    parameter
        STATE_SETUP = 'b000,
        STATE_READY = 'b001,
        STATE_FETCH = 'b010,
        STATE_DECODE = 'b011,
        STATE_CALC = 'b100,
        STATE_MEM = 'b101,
        STATE_WRITEBACK = 'b110,
        STATE_DONE = 'b111;

    reg [2:0] state;

    initial begin
        state = STATE_SETUP;
    end

    reg [2:0] state_next;
    always @(*) case (state)
        STATE_SETUP: state_next = (mem_ready && reg_file_ready) ? STATE_READY :
STATE_SETUP;
        STATE_READY: state_next = start_sig ? STATE_FETCH : STATE_READY;
        STATE_FETCH: state_next = mem_resp ? STATE_DECODE : STATE_FETCH;
        STATE_DECODE: state_next = halt ? STATE_DONE : STATE_CALC;
        STATE_CALC: state_next = do_mem ? STATE_MEM : STATE_WRITEBACK;
        STATE_MEM: state_next = mem_resp ? STATE_WRITEBACK : STATE_MEM;
        STATE_WRITEBACK: state_next = STATE_FETCH;
        STATE_DONE: state_next = STATE_DONE;
        default: state_next = state;
    endcase

    assign calc = state == STATE_CALC;
    assign decode = state == STATE_DECODE;
    assign done_o = state == STATE_DONE;
    assign mem_req = (state == STATE_FETCH) || (state == STATE_MEM);
    assign mem_req_type = (state_next == STATE_MEM) || (state == STATE_MEM);
    assign pc_update = (state == STATE_WRITEBACK);
    assign ready_o = state == STATE_READY;
    assign wb_sig = (state == STATE_WRITEBACK) || (state == STATE_SETUP && !
reg_file_ready && clk);
```

```
    // State change
    always @(posedge clk) state <= state_next;

endmodule
```

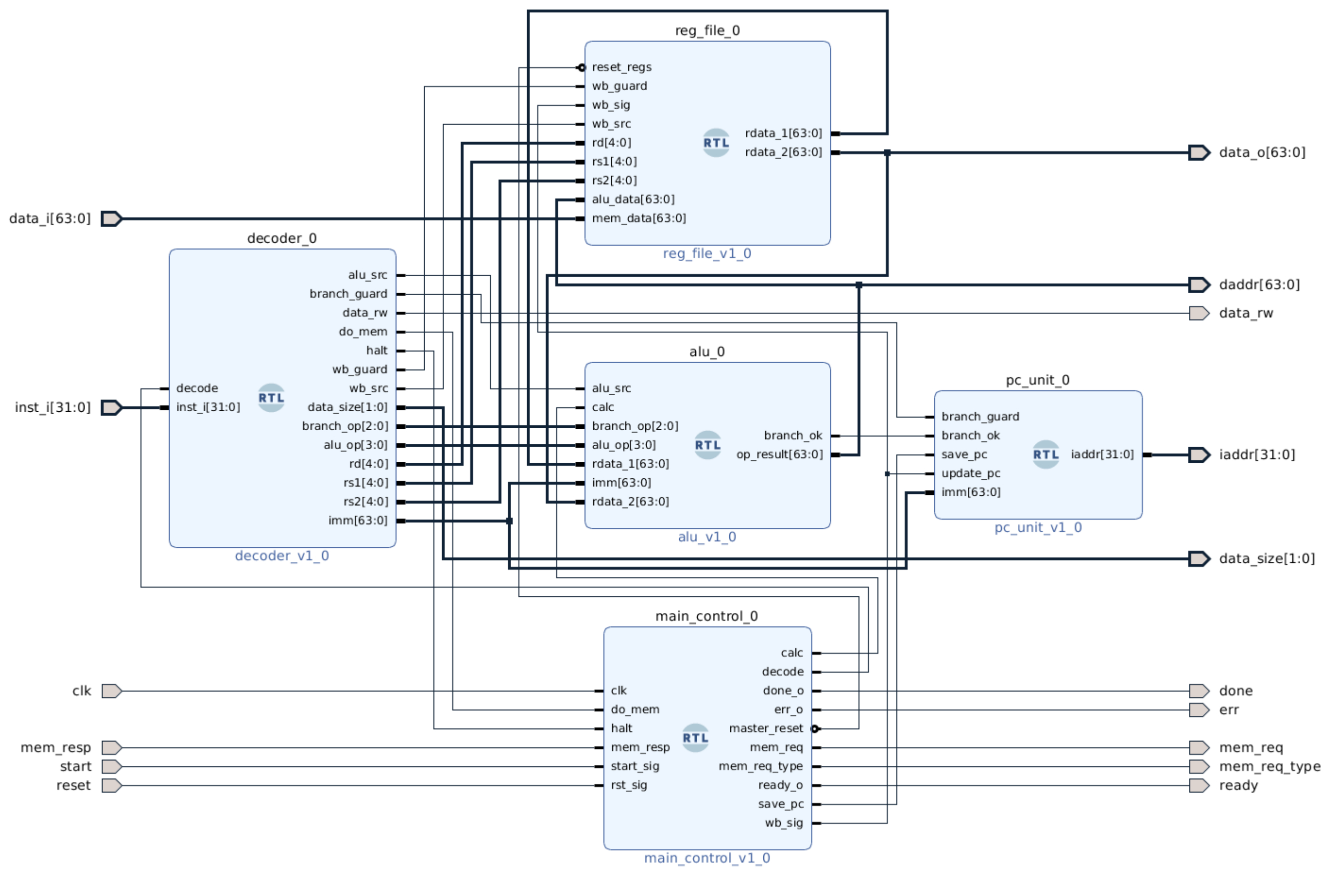
pc_unit.v

```
module pc_unit(
        input wire branch_ok, update_pc,
        input wire pc_op,
        input wire [63:0] imm,

        output reg [31:0] iaddr
    );

    `include "op_aliases.vh"

    wire [31:0] imm_trunc = imm[31:0];

    wire [31:0] branch_addr = iaddr + imm_trunc;

    initial iaddr = 0;

    always @(posedge update_pc) case(pc_op)
        PC_OP_NEXT: iaddr <= iaddr + 4;
        PC_OP_BRANCH: iaddr <= branch_ok ? iaddr + imm_trunc : iaddr + 4;
        default: iaddr <= iaddr + 4;
    endcase

endmodule
```

reg_file.v

```
module reg_file (
    input wire wb_guard, wb_sig,
    input wire [1:0] wb_src,
    input wire [4:0] rd, rs1, rs2,
    input wire [31:0] pc_data,
    input wire [63:0] alu_data, imm_data, mem_data,

    output wire [63:0] rdata_1, rdata_2
    );

    `include "op_aliases.vh"

    wire wb_final = wb_guard && wb_sig && (rd != 0);
    reg [63:0] wb_data;
    always @(*) case (wb_src)
        WB_SRC_ALU: wb_data = alu_data;
        WB_SRC_DMEM: wb_data = mem_data;
        WB_SRC_IMM: wb_data = imm_data;
        WB_SRC_PC: wb_data = pc_data;
        default: wb_data = 'bx;
    endcase

    reg [63:0] regs [31:0];

    initial begin
        regs[0] = 0; regs[1] = 0; regs[2] = 'd2040; regs[3] = 0;
        regs[4] = 0; regs[5] = 0; regs[6] = 0; regs[7] = 0;
        regs[8] = 0; regs[9] = 0; regs[10] = 0; regs[11] = 0;
        regs[12] = 0; regs[13] = 0; regs[14] = 0; regs[15] = 0;
        regs[16] = 0; regs[17] = 0; regs[18] = 0; regs[19] = 0;
        regs[20] = 0; regs[21] = 0; regs[22] = 0; regs[23] = 0;
        regs[24] = 0; regs[25] = 0; regs[26] = 0; regs[27] = 0;
        regs[28] = 0; regs[29] = 0; regs[30] = 0; regs[31] = 0;
    end

    assign rdata_1 = regs[rs1];
    assign rdata_2 = regs[rs2];

    always @(posedge wb_final)
        regs[rd] <= wb_data;

endmodule
```

Vivado Block diagram for BabyRisc



regular_main_mem.v

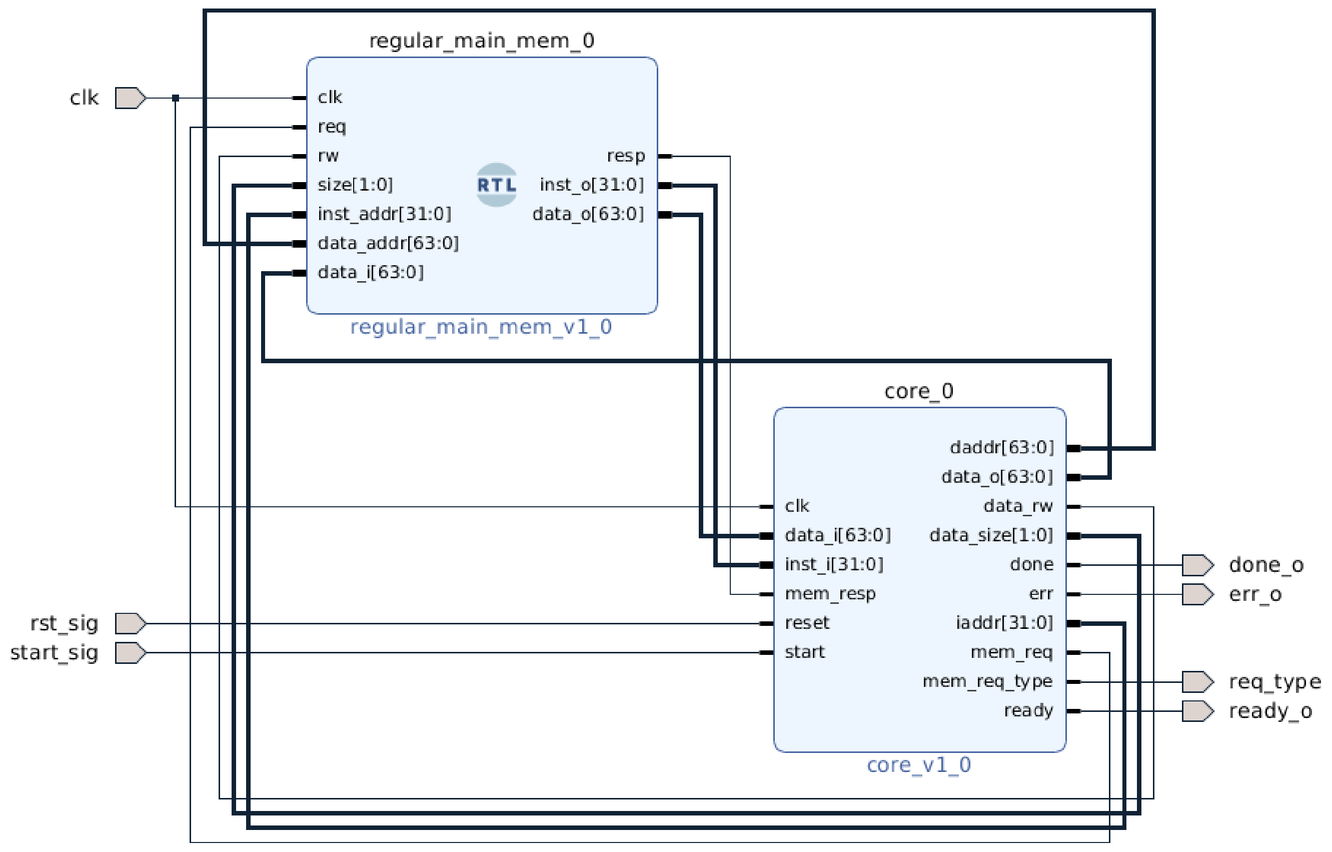
```
module regular_main_mem(
    input wire clk, req, rw,
    input wire [1:0] size,
    input wire [31:0] inst_addr,
    input wire [63:0] data_addr, data_i,

    output wire resp,
    output wire [31:0] inst_o,
    output reg [63:0] data_o
    );

    parameter
        BYTE = 'b00,
        HALF = 'b01,
        WORD = 'b10,
        MODE_INST = 'b00,
        MODE_DATA_RI = 'b01,
        MODE_DATA_WB = 'b10,
        IDLE = 'b0000,
        FETCH_1 = 'b0001,
        FETCH_2 = 'b0010,
        FETCH_3 = 'b0011,
        FETCH_4 = 'b0100,
        FETCH_5 = 'b0101,
        FETCH_6 = 'b0110,
        FETCH_7 = 'b0111,
        FETCH_8 = 'b1000,
        FETCH_9 = 'b1001,
        RETURN = 'b1010;

    reg [3:0] state;
    // reg [7:0] data [31:0];
    reg [7:0] data [2047:0];
    wire we = (state == FETCH_9) && rw;
    reg [3:0] state_next;
    always @(*) case (state)
        IDLE: state_next = (req) ? FETCH_1 : state;
        FETCH_1: state_next = FETCH_2;
        FETCH_2: state_next = FETCH_3;
        FETCH_3: state_next = FETCH_4;
        FETCH_4: state_next = FETCH_5;
        FETCH_5: state_next = FETCH_6;
        FETCH_6: state_next = FETCH_7;
        FETCH_7: state_next = FETCH_8;
        FETCH_8: state_next = FETCH_9;
        FETCH_9: state_next = RETURN;
        RETURN: state_next = (req) ? state : IDLE;
        default: state_next = state;
    endcase

    reg [11:0] init_counter;
    initial begin
        state = IDLE;

        init_counter = 0;
        while (init_counter < 2048) begin
            data[init_counter] = 0;
            init_counter = init_counter + 1;
        end

        data[3] = 'b10011101;
        data[2] = 'b00000001;
        data[1] = 'b00000001;
        data[0] = 'b00010011;

        data[7] = 'b01100010;
        data[6] = 'b10000001;
        data[5] = 'b00110100;
        data[4] = 'b00100011;

        data[11] = 'b01100011;
        data[10] = 'b00000001;
        data[9] = 'b00000100;
        data[8] = 'b00010011;

        data[15] = 'b11111100;
        data[14] = 'b00000100;
        data[13] = 'b00111000;
        data[12] = 'b00100011;

        data[19] = 'b00000000;
        data[18] = 'b00010000;
        data[17] = 'b00000111;
        data[16] = 'b10010011;

        data[23] = 'b11111110;
        data[22] = 'b11110100;
        data[21] = 'b00110100;
        data[20] = 'b00100011;

        data[27] = 'b00000000;
        data[26] = 'b00010000;
        data[25] = 'b00000111;
        data[24] = 'b10010011;

        data[31] = 'b11111110;
```

```
data[30] = 'b11110100;
data[29] = 'b00110000;
data[28] = 'b00100011;

data[35] = 'b11111100;
data[34] = 'b00000100;
data[33] = 'b00111100;
data[32] = 'b00100011;

data[39] = 'b11111101;
data[38] = 'b10000100;
data[37] = 'b00110111;
data[36] = 'b10000011;

data[43] = 'b00000000;
data[42] = 'b00110111;
data[41] = 'b10010111;
data[40] = 'b10010011;

data[47] = 'b11111111;
data[46] = 'b00000100;
data[45] = 'b00000111;
data[44] = 'b00010011;

data[51] = 'b00000000;
data[50] = 'b11100111;
data[49] = 'b10000111;
data[48] = 'b10110011;

data[55] = 'b11111110;
data[54] = 'b00000100;
data[53] = 'b00110111;
data[52] = 'b00000011;

data[59] = 'b10011110;
data[58] = 'b11100111;
data[57] = 'b10110000;
data[56] = 'b00100011;

data[63] = 'b11111101;
data[62] = 'b10000100;
data[61] = 'b00110111;
data[60] = 'b10000011;

data[67] = 'b00000000;
data[66] = 'b00010111;
data[65] = 'b10000111;
data[64] = 'b10010011;
```

```
data[71] = 'b11111100;
data[70] = 'b11110100;
data[69] = 'b00111100;
data[68] = 'b00100011;

data[75] = 'b11111101;
data[74] = 'b10000100;
data[73] = 'b00110111;
data[72] = 'b00000011;

data[79] = 'b00001100;
data[78] = 'b00000000;
data[77] = 'b00000111;
data[76] = 'b10010011;

data[83] = 'b00000010;
data[82] = 'b11110111;
data[81] = 'b00000100;
data[80] = 'b01100011;

data[87] = 'b11111110;
data[86] = 'b10000100;
data[85] = 'b00110111;
data[84] = 'b10000011;

data[91] = 'b11111100;
data[90] = 'b11110100;
data[89] = 'b00111000;
data[88] = 'b00100011;

data[95] = 'b11111110;
data[94] = 'b00000100;
data[93] = 'b00110111;
data[92] = 'b10000011;

data[99] = 'b11111110;
data[98] = 'b11110100;
data[97] = 'b00110100;
data[96] = 'b00100011;

data[103] = 'b11111101;
data[102] = 'b00000100;
data[101] = 'b00110111;
data[100] = 'b00000011;

data[107] = 'b11111110;
data[106] = 'b10000100;
data[105] = 'b00110111;
data[104] = 'b10000011;
```

```
        data[111] = 'b00000000;
        data[110] = 'b11100111;
        data[109] = 'b10000111;
        data[108] = 'b10110011;

        data[115] = 'b11111110;
        data[114] = 'b11110100;
        data[113] = 'b00110000;
        data[112] = 'b00100011;

        data[119] = 'b11111010;
        data[118] = 'b00000000;
        data[117] = 'b00001000;
        data[116] = 'b11100011;

        data[123] = 'b01100010;
        data[122] = 'b10000001;
        data[121] = 'b00110100;
        data[120] = 'b00000011;

        data[127] = 'b01100011;
        data[126] = 'b00000001;
        data[125] = 'b00000001;
        data[124] = 'b00010011;

        data[131] = 'b00000000;
        data[130] = 'b00000000;
        data[129] = 'b00000000;
        data[128] = 'b00000000;

    end

    assign resp = state == RETURN;
    assign inst_o = {data[inst_addr + 3], data[inst_addr + 2], data[inst_addr +
1],
                     data[inst_addr]};
    always @(*) case (size)
        BYTE: data_o = {56'b0, data[data_addr]};
        HALF: data_o = {48'b0, data[data_addr + 1], data[data_addr]};
        WORD: data_o = {32'b0, data[data_addr + 3], data[data_addr + 2],
data[data_addr + 1],
                        data[data_addr]};
        default: data_o = {data[data_addr + 7], data[data_addr + 6],
data[data_addr + 5],
                           data[data_addr + 4], data[data_addr + 3],
data[data_addr + 2],
                           data[data_addr + 1], data[data_addr]};
    endcase
    always @(posedge clk) state <= state_next;
    always @(posedge we) begin
        data[data_addr] <= data_i[7:0];
        if (size > BYTE) begin
            data[data_addr + 1] <= data_i[15:8];
            if (size > HALF) begin
                data[data_addr + 2] <= data_i[23:16];
                data[data_addr + 3] <= data_i[31:24];
                if (size > WORD) begin
                    data[data_addr + 4] <= data_i[39:32];
                    data[data_addr + 5] <= data_i[47:40];
                    data[data_addr + 6] <= data_i[55:48];
                    data[data_addr + 7] <= data_i[63:56];
                end
            end
        end
    end

endmodule
```

Block diagram for design 1



caching_main_mem.v

```
module caching_main_mem(
    input wire clk, req, req_type, rw,
    input wire [27:0] data_ri_tag_index, data_wb_tag_index, inst_tag_index,
    input wire [127:0] line_i,

    output wire resp,
    output wire [127:0] line_o
    );

    parameter
        MODE_INST = 'b00,
        MODE_DATA_RI = 'b01,
        MODE_DATA_WB = 'b10,
        IDLE = 'b0000,
        FETCH_1 = 'b0001,
        FETCH_2 = 'b0010,
        FETCH_3 = 'b0011,
        FETCH_4 = 'b0100,
        FETCH_5 = 'b0101,
        FETCH_6 = 'b0110,
        FETCH_7 = 'b0111,
        FETCH_8 = 'b1000,
        FETCH_9 = 'b1001,
        RETURN = 'b1010;

    reg [3:0] state;
    reg [127:0] lines [127:0];
    wire we = (state == FETCH_9) && rw;
    reg [3:0] state_next;
    always @(*) case (state)
        IDLE: state_next = (req) ? FETCH_1 : state;
        FETCH_1: state_next = FETCH_2;
        FETCH_2: state_next = FETCH_3;
        FETCH_3: state_next = FETCH_4;
        FETCH_4: state_next = FETCH_5;
        FETCH_5: state_next = FETCH_6;
        FETCH_6: state_next = FETCH_7;
        FETCH_7: state_next = FETCH_8;
        FETCH_8: state_next = FETCH_9;
        FETCH_9: state_next = RETURN;
        RETURN: state_next = (req) ? state : IDLE;
        default: state_next = state;
    endcase

    reg [7:0] init_counter;
    initial begin
        state = IDLE;
```

```
init_counter = 0;
while (init_counter < 128) begin
    lines[init_counter] = 0;
    init_counter = init_counter + 1;
end

lines[0] = {32'b11111100000001000011100000100011,
            32'b01100011000000010000010000010011,
            32'b01100010100000001001101000100011,
            32'b10011101000000001000000000100010011};

lines[1] = {32'b11111110111101000011000000100011,
            32'b00000000000010000000001110010011,
            32'b11111110111101000011010000100011,
            32'b00000000000010000000001110010011};

lines[2] = {32'b11111111000001000000011100010011,
            32'b00000000001101111001011110010011,
            32'b11111101100001000011011110000011,
            32'b11111100000001000011110000100011};

lines[3] = {32'b11111101100001000011011110000011,
            32'b10011110111001111011000000100011,
            32'b11111110000001000011011100000011,
            32'b00000000111001111000011110110011};

lines[4] = {32'b00001100000000000000001110010011,
            32'b11111101100001000011011100000011,
            32'b11111100111101000011110000100011,
            32'b00000000000101111000011110010011};

lines[5] = {32'b11111110000001000011011110000011,
            32'b11111100111101000011100000100011,
            32'b11111110100001000011011110000011,
            32'b00000010111101110000010001100011};

lines[6] = {32'b00000000111001111000011110110011,
            32'b11111110100001000011011110000011,
            32'b11111101000001000011011100000011,
            32'b11111110111101000011010000100011};

lines[7] = {32'b01100011000000010000000100010011,
            32'b01100010100000001001101000000011,
            32'b11111010000000000000100011100011,
            32'b11111110111101000011000000100011};

lines[8] = {32'b0,
            32'b0,
                    32'b0,
                    32'b00000000000000000000000000000000};
    end

    assign resp = state == RETURN;
    assign line_o = req_type ? lines[data_ri_tag_index] : lines[inst_tag_index];

    always @(posedge clk) state <= state_next;
    always @(posedge we) lines[data_wb_tag_index] <= line_i;

endmodule
```

data_main_cache.v

```
module data_main_cache(
    input wire we, wmode,
    input wire [1:0] size,
    input wire [63:0] addr, data_i,
    input wire [127:0] line_i,

    output wire hm, wb,
    output wire [27:0] ri_tag_index, wb_tag_index,
    output reg [63:0] data_o,
    output wire [127:0] line_o
    );

    parameter
        BYTE = 'b00,
        HALF = 'b01,
        WORD = 'b10,
        DOUBLE = 'b11;

    wire [21:0] tag = addr[31:10];
    wire [5:0] index = addr[9:4];
    wire [3:0] offset = addr[3:0];

    reg [63:0] valid;
    reg [23:0] tags [63:0];
    reg [7:0] data [63:0] [15:0]; // 64 lines, 16 bytes per line

    reg [6:0] init_counter_1;
    reg [4:0] init_counter_2;
    initial begin
        valid = 0;
        init_counter_1 = 0;
        while (init_counter_1 < 64) begin
            tags[init_counter_1] = 0;
            init_counter_2 = 0;
            while (init_counter_2 < 16) begin
                data[init_counter_1][init_counter_2] = 0;
                init_counter_2 = init_counter_2 + 1;
            end
            init_counter_1 = init_counter_1 + 1;
        end
    end

    assign hm = (tag == tags[index]) && valid[index];
    assign wb = valid[index];
    assign ri_tag_index = {tag, index};
    assign wb_tag_index = {tags[index], index};
    always @(*) case (size)
```

```
        BYTE: data_o = {56'b0, data[index][offset]};
        HALF: data_o = {48'b0, data[index][offset + 1], data[index][offset]};
        WORD: data_o = {32'b0, data[index][offset + 3], data[index][offset + 2],
                        data[index][offset + 1], data[index][offset]};
        default: data_o = {data[index][offset + 7], data[index][offset + 6],
                           data[index][offset + 5], data[index][offset + 4],
                           data[index][offset + 3], data[index][offset + 2],
                           data[index][offset + 1], data[index][offset]};
    endcase

    assign line_o = {data[index][15], data[index][14], data[index][13],
data[index][12],
                        data[index][11], data[index][10], data[index][9],
data[index][8],
                        data[index][7], data[index][6], data[index][5],
data[index][4],
                        data[index][3], data[index][2], data[index][1],
data[index][0]};

    always @(posedge we) if (~wmode) begin // Normal write
        data[index][offset] <= data_i[7:0];
        if (size > BYTE) begin
            data[index][offset + 1] <= data_i[15:8];
            if (size > HALF) begin
                data[index][offset + 2] <= data_i[23:16];
                data[index][offset + 3] <= data_i[31:24];
                if (size > WORD) begin
                    data[index][offset + 4] <= data_i[39:32];
                    data[index][offset + 5] <= data_i[47:40];
                    data[index][offset + 6] <= data_i[55:48];
                    data[index][offset + 7] <= data_i[63:56];
                end
            end
        end
    end else begin // Line Write
        valid[index] <= 1;
        tags[index] <= tag;
        data[index][0] <= line_i[7:0];
        data[index][1] <= line_i[15:8];
        data[index][2] <= line_i[23:16];
        data[index][3] <= line_i[31:24];
        data[index][4] <= line_i[39:32];
        data[index][5] <= line_i[47:40];
        data[index][6] <= line_i[55:48];
        data[index][7] <= line_i[63:56];
        data[index][8] <= line_i[71:64];
        data[index][9] <= line_i[79:72];
        data[index][10] <= line_i[87:80];
        data[index][11] <= line_i[95:88];
        data[index][12] <= line_i[103:96];
        data[index][13] <= line_i[111:104];
        data[index][14] <= line_i[119:112];
        data[index][15] <= line_i[127:120];
    end

endmodule
```

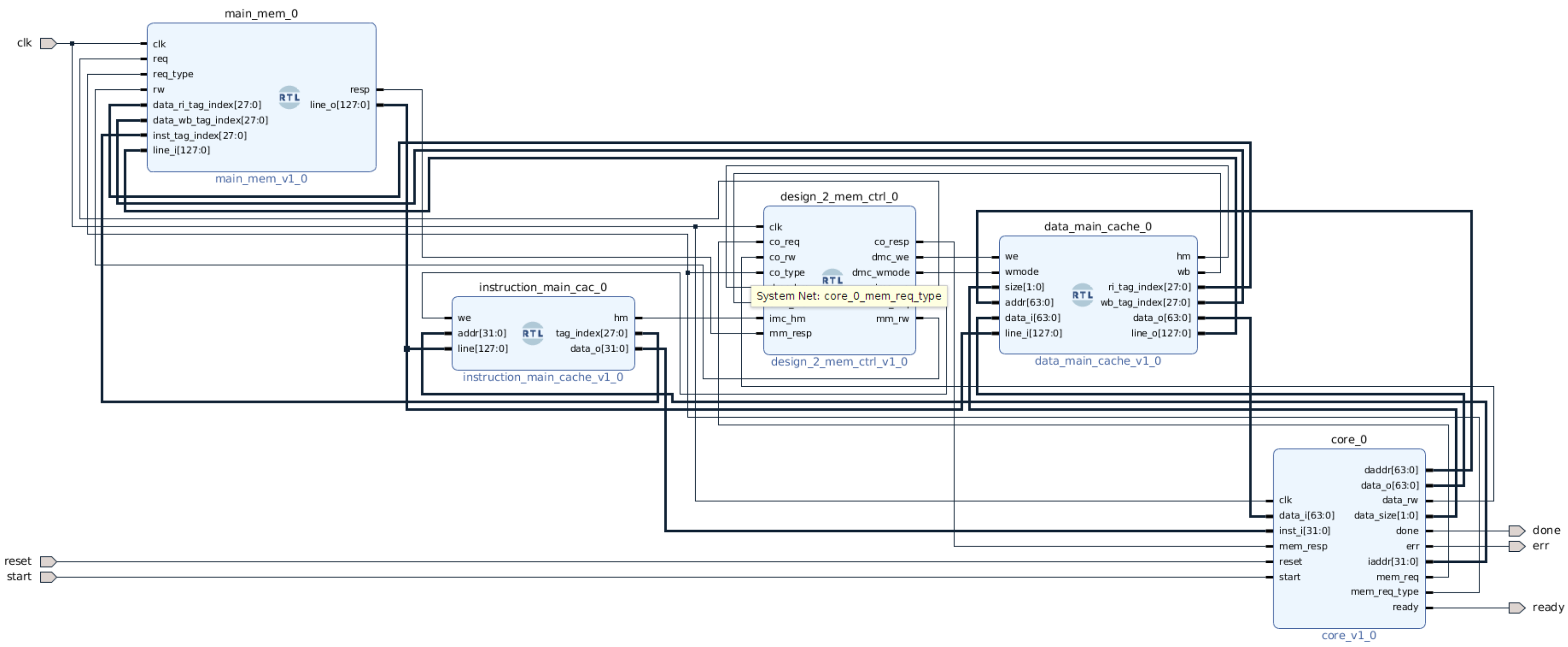
design_2_mem_ctrl.v

```
module design_2_mem_ctrl(
    input wire clk, co_req, co_rw, co_type, dmc_hm, dmc_wb, imc_hm, mm_resp,

    output wire co_resp, dmc_we, dmc_wmode, imc_we, mm_req, mm_rw
    );

    parameter
        // MODE_INST = 'b00,
        // MODE_DATA_RI = 'b01,
        // MODE_DATA_WB = 'b10,

        IDLE = 'b0000,
        WRITEBACK = 'b0001,
        WRITEBACK_UNSET = 'b0010,
        INST_READIN = 'b0011,
        INST_WRITEIN = 'b0100,
        DATA_READIN = 'b0101,
        DATA_WRITEIN = 'b0110,
        DATA_WRITEIN_UNSET = 'b0111,
        DATA_WRITE = 'b1000,
        FINISH = 'b1001;

    reg [3:0] state;
    reg [3:0] state_next;
    always @(*) case (state)
        IDLE: if (co_req)
                  if (co_type)
                      if (dmc_hm)
                          state_next = (co_rw) ? DATA_WRITE : FINISH;
                      else state_next = (dmc_wb) ? WRITEBACK : DATA_READIN;
                  else state_next = (imc_hm) ? FINISH : INST_READIN;
              else state_next = state;
        WRITEBACK: state_next = mm_resp ? WRITEBACK_UNSET : state;
        WRITEBACK_UNSET: state_next = (mm_resp) ? state : DATA_READIN;
        INST_READIN: state_next = (mm_resp) ? INST_WRITEIN : state;
        INST_WRITEIN: state_next = FINISH;
        DATA_READIN: state_next = (mm_resp) ? DATA_WRITEIN : state;
        DATA_WRITEIN: state_next = (co_rw) ? DATA_WRITEIN_UNSET : FINISH;
        DATA_WRITEIN_UNSET: state_next = DATA_WRITE;
        DATA_WRITE: state_next = FINISH;
        FINISH: state_next = (co_req) ? state : IDLE;
        default: state_next = state;
    endcase

    initial state = IDLE;

    // Requset is finished
    assign co_resp = state == FINISH;
    // Request is data type and operation is either a writein or normal write
    assign dmc_we = (state == DATA_WRITEIN) || ((state == DATA_WRITE) && co_rw);
    // Line write for state before and during write-in, otherwise normal
    assign dmc_wmode = (state == DATA_READIN) || (state == DATA_WRITEIN);
    // Request is instrution type and operation is a writein
    assign imc_we = state == INST_WRITEIN;
    // Writeback or read-in request to main memory
    assign mm_req = (state == WRITEBACK) || (state == INST_READIN) || (state ==
DATA_READIN);
    assign mm_rw = ((state == IDLE) && ~dmc_hm && dmc_wb) || (state == WRITEBACK);

    always @(posedge clk) state <= state_next;

endmodule
```

instruction_main_cache.v

```
module instruction_main_cache(
    input wire we,
    input wire [31:0] addr,
    input wire [127:0] line,

    output wire hm,
    output wire [27:0] tag_index,
    output wire [31:0] data_o
    );

    wire [3:0] offset = addr[3:0];
    wire [5:0] index = addr[9:4];
    wire [21:0] tag = addr[31:10];

    reg [63:0] valid;
    reg [23:0] tags [63:0];
    reg [7:0] data [63:0] [15:0]; // 64 lines, 16 bytes per line

    reg [6:0] init_counter_1;
    reg [4:0] init_counter_2;
    initial begin
        valid = 0;
        init_counter_1 = 0;
        while (init_counter_1 < 64) begin
            tags[init_counter_1] = 0;
            init_counter_2 = 0;
            while (init_counter_2 < 16) begin
                data[init_counter_1][init_counter_2] = 0;
                init_counter_2 = init_counter_2 + 1;
            end
            init_counter_1 = init_counter_1 + 1;
        end
    end

    assign hm = (tag == tags[index]) && valid[index];
    assign tag_index = addr[31:4];
    assign data_o = {data[index][offset + 3], data[index][offset + 2], data[index]
[offset + 1],
                        data[index][offset]};

    always @(posedge we) begin
        valid[index] <= 1;
        tags[index] <= tag;
        data[index][0] <= line[7:0];
        data[index][1] <= line[15:8];
        data[index][2] <= line[23:16];
        data[index][3] <= line[31:24];
        data[index][4] <= line[39:32];
        data[index][5] <= line[47:40];
        data[index][6] <= line[55:48];
        data[index][7] <= line[63:56];
        data[index][8] <= line[71:64];
        data[index][9] <= line[79:72];
        data[index][10] <= line[87:80];
        data[index][11] <= line[95:88];
        data[index][12] <= line[103:96];
        data[index][13] <= line[111:104];
        data[index][14] <= line[119:112];
        data[index][15] <= line[127:120];
    end

endmodule
```

Block diagram for design with L1 Caches



data_main_cache_v2.v

```
module data_main_cache_v2 (
    input wire set_swap, we, wmode, write_src,
    input wire [1:0] size,
    input wire [63:0] addr, data_i,
    input wire [127:0] mm_line, swap_line_i,

    output wire do_evict, hm,
    output wire [27:0] evict_tag_index, read_tag_index,
    output reg [63:0] data_o,
    output wire [127:0] evict_line_o,
    output reg [27:0] swap_tag_index,
    output reg [127:0] swap_line_o
    );

    parameter
        BYTE = 'b00,
        HALF = 'b01,
        WORD = 'b10,
        DOUBLE = 'b11;

    wire [3:0] offset = addr[3:0];
    wire [5:0] index = addr[9:4];
    wire [21:0] tag = addr[31:10];
    wire [127:0] write_line = write_src ? mm_line : swap_line_i;

    reg [63:0] valid;
    reg [23:0] tags [63:0];
    reg [7:0] data [63:0] [15:0]; // 64 lines, 16 bytes per line

    reg [6:0] init_counter_1;
    reg [4:0] init_counter_2;
    initial begin
        valid = 0;
        init_counter_1 = 0;
        while (init_counter_1 < 64) begin
            tags[init_counter_1] = 0;
            init_counter_2 = 0;
            while (init_counter_2 < 16) begin
                data[init_counter_1][init_counter_2] = 0;
                init_counter_2 = init_counter_2 + 1;
            end
            init_counter_1 = init_counter_1 + 1;
        end
    end

    assign do_evict = valid[index];
    assign hm = valid[index] && (tags[index] == tag);
    assign evict_tag_index = {tags[index], index};
    assign read_tag_index = {tag, index};
    always @(*) case (size)
        BYTE: data_o = {56'b0, data[index][offset]};
        HALF: data_o = {48'b0, data[index][offset + 1], data[index][offset]};
        WORD: data_o = {32'b0, data[index][offset + 3], data[index][offset + 2],
                        data[index][offset + 1], data[index][offset]};
        default: data_o = {data[index][offset + 7], data[index][offset + 6],
                          data[index][offset + 5], data[index][offset + 4],
                          data[index][offset + 3], data[index][offset + 2],
                          data[index][offset + 1], data[index][offset]};
    endcase
    assign evict_line_o = {data[index][15], data[index][14], data[index][13],
data[index][12],
                               data[index][11], data[index][10], data[index][9],
data[index][8],
                               data[index][7], data[index][6], data[index][5],
data[index][4],
                               data[index][3], data[index][2], data[index][1],
data[index][0]};

    always @(posedge set_swap) begin
        swap_line_o <= evict_line_o;
        swap_tag_index <= evict_tag_index;
    end

    always @(posedge we) if (wmode) begin // Full line write
        valid[index] <= 1;
        tags[index] <= tag;
        data[index][0] <= write_line[7:0];
        data[index][1] <= write_line[15:8];
        data[index][2] <= write_line[23:16];
        data[index][3] <= write_line[31:24];
        data[index][4] <= write_line[39:32];
        data[index][5] <= write_line[47:40];
        data[index][6] <= write_line[55:48];
        data[index][7] <= write_line[63:56];
        data[index][8] <= write_line[71:64];
        data[index][9] <= write_line[79:72];
        data[index][10] <= write_line[87:80];
        data[index][11] <= write_line[95:88];
        data[index][12] <= write_line[103:96];
        data[index][13] <= write_line[111:104];
        data[index][14] <= write_line[119:112];
        data[index][15] <= write_line[127:120];
    end else begin // Regular write
        data[index][offset] <= data_i[7:0];
        if (size > 0) begin
            data[index][offset + 1] <= data_i[15:8];
            if (size > 1) begin
                data[index][offset + 2] <= data_i[23:16];
                data[index][offset + 3] <= data_i[31:24];
                if (size > 2) begin
                    data[index][offset + 4] <= data_i[39:32];
                    data[index][offset + 5] <= data_i[47:40];
                    data[index][offset + 6] <= data_i[55:48];
                    data[index][offset + 7] <= data_i[63:56];
                end
            end
        end
    end

endmodule
```

data_victim_cache.v

```
module dvc(
        input wire set_swap, we, wsrc,
        input wire [27:0] evict_tag_index, read_tag_index, swap_tag_index_i,
        input wire [127:0] evict_line_i, swap_line_i,

        output wire hm, wb,
        output wire [27:0] wb_tag_index,
        output wire [127:0] wb_line,

        output reg [127:0] swap_line_o
    );

    parameter
        TARGET_0 = 'b000,
        TARGET_1 = 'b001,
        TARGET_2 = 'b010,
        TARGET_3 = 'b011,
        TARGET_4 = 'b100,
        TARGET_5 = 'b101,
        TARGET_6 = 'b110,
        TARGET_7 = 'b111,
        MATCH_0 = 'b1000,
        MATCH_1 = 'b1001,
        MATCH_2 = 'b1010,
        MATCH_3 = 'b1011,
        MATCH_4 = 'b1100,
        MATCH_5 = 'b1101,
        MATCH_6 = 'b1110,
        MATCH_7 = 'b1111,
        MATCH_NONE = 'b0000,
        VALID_0 = 'b00000000,
        VALID_1 = 'b00000001,
        VALID_2 = 'b00000011,
        VALID_3 = 'b00000111,
        VALID_4 = 'b00001111,
        VALID_5 = 'b00011111,
        VALID_6 = 'b00111111,
        VALID_7 = 'b01111111,
        VALID_8 = 'b11111111;

    reg [2:0] target_swap;
    reg [7:0] valid;
    reg [27:0] tag_indexes [7:0];
    reg [127:0] lines [7:0];
    // Target for the line coming in.
    reg [2:0] target_push;
    always @(*) case (valid)
```

```
        VALID_0: target_push = TARGET_0;
        VALID_1: target_push = TARGET_1;
        VALID_2: target_push = TARGET_2;
        VALID_3: target_push = TARGET_3;
        VALID_4: target_push = TARGET_4;
        VALID_5: target_push = TARGET_5;
        VALID_6: target_push = TARGET_6;
        default: target_push = TARGET_7;
    endcase
    reg [3:0] match;
    always @(*) if (valid[0] && (read_tag_index == tag_indexes[0])) match =
MATCH_0;
                else if (valid[1] && (read_tag_index == tag_indexes[1])) match =
MATCH_1;
                else if (valid[2] && (read_tag_index == tag_indexes[2])) match =
MATCH_2;
                else if (valid[3] && (read_tag_index == tag_indexes[3])) match =
MATCH_3;
                else if (valid[4] && (read_tag_index == tag_indexes[4])) match =
MATCH_4;
                else if (valid[5] && (read_tag_index == tag_indexes[5])) match =
MATCH_5;
                else if (valid[6] && (read_tag_index == tag_indexes[6])) match =
MATCH_6;
                else if (valid[7] && (read_tag_index == tag_indexes[7])) match =
MATCH_7;
                else match = MATCH_NONE;

    reg [2:0] init_loop;
    initial begin
        swap_line_o = 0;

        target_swap = 0;
        valid = 0;

        for (init_loop = 0; init_loop < 7; init_loop = init_loop + 1) begin
            tag_indexes[init_loop] = 0;
            lines[init_loop] = 0;
        end
        tag_indexes[7] = 0;
        lines[7] = 0;
    end

    assign hm = match[3];
    assign wb = valid[7];
    assign wb_tag_index = tag_indexes[0];
    assign wb_line = lines[0];

    always @(posedge set_swap) begin
```

```
        case (match)
            MATCH_0: begin
                target_swap <= 0;
                swap_line_o <= lines[0];
            end
            MATCH_1: begin
                target_swap <= 1;
                swap_line_o <= lines[1];
            end
            MATCH_2: begin
                target_swap <= 2;
                swap_line_o <= lines[2];
            end
            MATCH_3: begin
                target_swap <= 3;
                swap_line_o <= lines[3];
            end
            MATCH_4: begin
                target_swap <= 4;
                swap_line_o <= lines[4];
            end
            MATCH_5: begin
                target_swap <= 5;
                swap_line_o <= lines[5];
            end
            MATCH_6: begin
                target_swap <= 6;
                swap_line_o <= lines[6];
            end
            MATCH_7: begin
                target_swap <= 7;
                swap_line_o <= lines[7];
            end
            MATCH_NONE: begin
                target_swap <= 0;
                swap_line_o <= 0;
            end
        endcase
    end

    always @(posedge we) begin
        if (~wsrc) begin // Eviction mode
            if (valid[7]) begin // Cache is full. Shift everything down and push
to top.
                tag_indexes[0] <= tag_indexes[1];
                tag_indexes[1] <= tag_indexes[2];
                tag_indexes[2] <= tag_indexes[3];
                tag_indexes[3] <= tag_indexes[4];
                tag_indexes[4] <= tag_indexes[5];
                tag_indexes[5] <= tag_indexes[6];
                tag_indexes[6] <= tag_indexes[7];
                tag_indexes[7] <= evict_tag_index;

                lines[0] <= lines[1];
                lines[1] <= lines[2];
                lines[2] <= lines[3];
                lines[3] <= lines[4];
                lines[4] <= lines[5];
                lines[5] <= lines[6];
                lines[6] <= lines[7];
                lines[7] <= evict_line_i;
            end else begin // Cache is not full. Push new line to top and make
valid.
                tag_indexes[target_push] <= evict_tag_index;
                lines[target_push] <= evict_line_i;
                valid[target_push] <= 'b1;
            end
        end else begin // Swap mode
            // Everything above the swap target gets pushed down.
            if (target_swap == 0) begin
                tag_indexes[0] <= tag_indexes[1];
                lines[0] <= lines[1];
            end
            if (target_swap <= 1) begin
                tag_indexes[1] <= tag_indexes[2];
                lines[1] <= lines[2];
            end
            if (target_swap <= 2) begin
                tag_indexes[2] <= tag_indexes[3];
                lines[2] <= lines[3];
            end
            if (target_swap <= 3) begin
                tag_indexes[3] <= tag_indexes[4];
                lines[3] <= lines[4];
            end
            if (target_swap <= 4) begin
                tag_indexes[4] <= tag_indexes[5];
                lines[4] <= lines[5];
            end
            if (target_swap <= 5) begin
                tag_indexes[5] <= tag_indexes[6];
                lines[5] <= lines[6];
            end
            if (target_swap <= 6) begin
                tag_indexes[6] <= tag_indexes[7];
                lines[6] <= lines[7];
            end
            if (valid[7]) begin // Cache is full. Push swapped-in line to top.
                tag_indexes[target_push] <= swap_tag_index_i;
                lines[target_push] <= swap_line_i;
            end else begin // Cache is not full. Push swapped-in line to top - 1.
                tag_indexes[target_push - 1] <= swap_tag_index_i;
                lines[target_push - 1] <= swap_line_i;
            end
        end
    end
endmodule
```

instruction_main_cache_v2.v

```
module instruction_main_cache_v2 (
    input wire set_swap, we, write_src,
    input wire [31:0] addr,
    input wire [127:0] mm_line, swap_line_i,

    output wire do_evict, hm,
    output wire [27:0] evict_tag_index, read_tag_index,
    output wire [31:0] data_o,
    output wire [127:0] evict_line_o,
    output reg [27:0] swap_tag_index,
    output reg [127:0] swap_line_o
    );

    wire [3:0] offset = addr[3:0];
    wire [5:0] index = addr[9:4];
    wire [21:0] tag = addr[31:10];
    wire [127:0] write_line = write_src ? mm_line : swap_line_i;

    reg [63:0] valid;
    reg [23:0] tags [63:0];
    reg [7:0] data [63:0] [15:0]; // 64 lines, 16 bytes per line

    reg [6:0] init_counter_1;
    reg [4:0] init_counter_2;
    initial begin
        valid = 0;
        init_counter_1 = 0;
        while (init_counter_1 < 64) begin
            tags[init_counter_1] = 0;
            init_counter_2 = 0;
            while (init_counter_2 < 16) begin
                data[init_counter_1][init_counter_2] = 0;
                init_counter_2 = init_counter_2 + 1;
            end
            init_counter_1 = init_counter_1 + 1;
        end
    end

    assign do_evict = valid[index];
    assign hm = valid[index] && (tags[index] == tag);
    assign evict_tag_index = {tags[index], index};
    assign read_tag_index = {tag, index};
    assign data_o =  {data[index][offset + 3], data[index][offset + 2],
data[index][offset + 1],
                          data[index][offset]};
    assign evict_line_o = {data[index][15], data[index][14], data[index][13],
data[index][12],
                               data[index][11], data[index][10], data[index][9],
data[index][8],
                               data[index][7], data[index][6], data[index][5],
data[index][4],
                               data[index][3], data[index][2], data[index][1],
data[index][0]};

    always @(posedge set_swap) begin
        swap_line_o <= evict_line_o;
        swap_tag_index <= evict_tag_index;
    end

    always @(posedge we) begin // Full line write
        valid[index] <= 1;
        tags[index] <= tag;
        data[index][0] <= write_line[7:0];
        data[index][1] <= write_line[15:8];
        data[index][2] <= write_line[23:16];
        data[index][3] <= write_line[31:24];
        data[index][4] <= write_line[39:32];
        data[index][5] <= write_line[47:40];
        data[index][6] <= write_line[55:48];
        data[index][7] <= write_line[63:56];
        data[index][8] <= write_line[71:64];
        data[index][9] <= write_line[79:72];
        data[index][10] <= write_line[87:80];
        data[index][11] <= write_line[95:88];
        data[index][12] <= write_line[103:96];
        data[index][13] <= write_line[111:104];
        data[index][14] <= write_line[119:112];
        data[index][15] <= write_line[127:120];
    end

endmodule
```

instruction_victim_cache.v

```
module instruction_victim_cache(
        input wire set_swap, we, wsrc,
        input wire [27:0] evict_tag_index, read_tag_index, swap_tag_index_i,
        input wire [127:0] evict_line_i, swap_line_i,

        output wire hm,
        output reg [127:0] swap_line_o
    );

    parameter
        TARGET_0 = 'b000,
        TARGET_1 = 'b001,
        TARGET_2 = 'b010,
        TARGET_3 = 'b011,
        TARGET_4 = 'b100,
        TARGET_5 = 'b101,
        TARGET_6 = 'b110,
        TARGET_7 = 'b111,
        MATCH_0 = 'b1000,
        MATCH_1 = 'b1001,
        MATCH_2 = 'b1010,
        MATCH_3 = 'b1011,
        MATCH_4 = 'b1100,
        MATCH_5 = 'b1101,
        MATCH_6 = 'b1110,
        MATCH_7 = 'b1111,
        MATCH_NONE = 'b0000,
        VALID_0 = 'b00000000,
        VALID_1 = 'b00000001,
        VALID_2 = 'b00000011,
        VALID_3 = 'b00000111,
        VALID_4 = 'b00001111,
        VALID_5 = 'b00011111,
        VALID_6 = 'b00111111,
        VALID_7 = 'b01111111,
        VALID_8 = 'b11111111;

    reg [2:0] target_swap;
    reg [7:0] valid;
    reg [27:0] tag_indexes [7:0];
    reg [127:0] lines [7:0];
    // Target for the line coming in.
    reg [2:0] target_push;
    always @(*) case (valid)
        VALID_0: target_push = TARGET_0;
        VALID_1: target_push = TARGET_1;
        VALID_2: target_push = TARGET_2;
        VALID_3: target_push = TARGET_3;
        VALID_4: target_push = TARGET_4;
        VALID_5: target_push = TARGET_5;
        VALID_6: target_push = TARGET_6;
        default: target_push = TARGET_7;
    endcase
    reg [3:0] match;
    always @(*) if (valid[0] && (read_tag_index == tag_indexes[0])) match =
MATCH_0;
                else if (valid[1] && (read_tag_index == tag_indexes[1])) match =
MATCH_1;
                else if (valid[2] && (read_tag_index == tag_indexes[2])) match =
MATCH_2;
                else if (valid[3] && (read_tag_index == tag_indexes[3])) match =
MATCH_3;
                else if (valid[4] && (read_tag_index == tag_indexes[4])) match =
MATCH_4;
                else if (valid[5] && (read_tag_index == tag_indexes[5])) match =
MATCH_5;
                else if (valid[6] && (read_tag_index == tag_indexes[6])) match =
MATCH_6;
                else if (valid[7] && (read_tag_index == tag_indexes[7])) match =
MATCH_7;
                else match = MATCH_NONE;

    reg [2:0] init_loop;
    initial begin
        swap_line_o = 0;

        target_swap = 0;
        valid = 0;

        for (init_loop = 0; init_loop < 7; init_loop = init_loop + 1) begin
            tag_indexes[init_loop] = 0;
            lines[init_loop] = 0;
        end
        tag_indexes[7] = 0;
        lines[7] = 0;
    end

    assign hm = match[3];

    always @(posedge set_swap) begin
        case (match)
            MATCH_0: begin
                target_swap <= 0;
                swap_line_o <= lines[0];
            end
            MATCH_1: begin
```

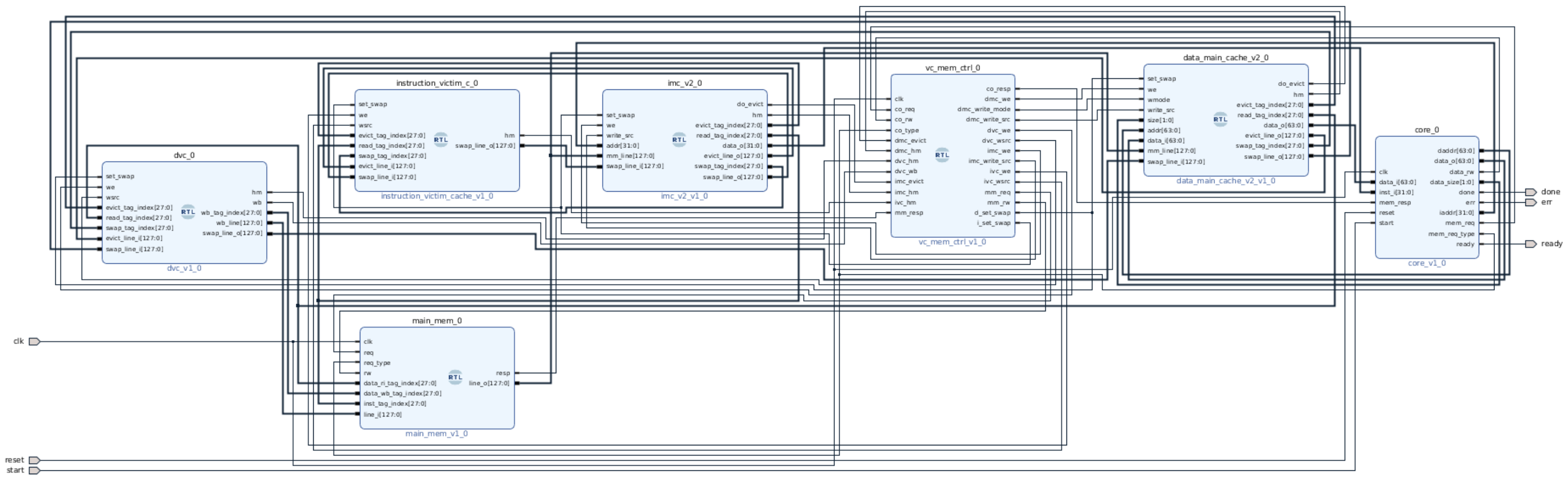
```
                target_swap <= 1;
                swap_line_o <= lines[1];
            end
            MATCH_2: begin
                target_swap <= 2;
                swap_line_o <= lines[2];
            end
            MATCH_3: begin
                target_swap <= 3;
                swap_line_o <= lines[3];
            end
            MATCH_4: begin
                target_swap <= 4;
                swap_line_o <= lines[4];
            end
            MATCH_5: begin
                target_swap <= 5;
                swap_line_o <= lines[5];
            end
            MATCH_6: begin
                target_swap <= 6;
                swap_line_o <= lines[6];
            end
            MATCH_7: begin
                target_swap <= 7;
                swap_line_o <= lines[7];
            end
            MATCH_NONE: begin
                target_swap <= 0;
                swap_line_o <= 0;
            end
        endcase
    end

    always @(posedge we) begin
        if (~wsrc) begin // Eviction mode
            if (valid[7]) begin // Cache is full. Shift everything down and push
to top.
                tag_indexes[0] <= tag_indexes[1];
                tag_indexes[1] <= tag_indexes[2];
                tag_indexes[2] <= tag_indexes[3];
                tag_indexes[3] <= tag_indexes[4];
                tag_indexes[4] <= tag_indexes[5];
                tag_indexes[5] <= tag_indexes[6];
                tag_indexes[6] <= tag_indexes[7];
                tag_indexes[7] <= evict_tag_index;

                lines[0] <= lines[1];
                lines[1] <= lines[2];
                lines[2] <= lines[3];
                lines[3] <= lines[4];
                lines[4] <= lines[5];
                lines[5] <= lines[6];
                lines[6] <= lines[7];
                lines[7] <= evict_line_i;
            end else begin // Cache is not full. Push new line to top and make
valid.
                tag_indexes[target_push] <= evict_tag_index;
                lines[target_push] <= evict_line_i;
                valid[target_push] <= 'b1;
            end
        end else begin // Swap mode
            // Everything above the swap target gets pushed down.
            if (target_swap == 0) begin
                tag_indexes[0] <= tag_indexes[1];
                lines[0] <= lines[1];
            end
            if (target_swap <= 1) begin
                tag_indexes[1] <= tag_indexes[2];
                lines[1] <= lines[2];
            end
            if (target_swap <= 2) begin
                tag_indexes[2] <= tag_indexes[3];
                lines[2] <= lines[3];
            end
            if (target_swap <= 3) begin
                tag_indexes[3] <= tag_indexes[4];
                lines[3] <= lines[4];
            end
            if (target_swap <= 4) begin
                tag_indexes[4] <= tag_indexes[5];
                lines[4] <= lines[5];
            end
            if (target_swap <= 5) begin
                tag_indexes[5] <= tag_indexes[6];
                lines[5] <= lines[6];
            end
            if (target_swap <= 6) begin
                tag_indexes[6] <= tag_indexes[7];
                lines[6] <= lines[7];
            end

            if (valid[7]) begin // Cache is full. Push swapped-in line to top.
                tag_indexes[target_push] <= swap_tag_index_i;
                lines[target_push] <= swap_line_i;
            end else begin // Cache is not full. Push swapped-in line to top - 1.
                tag_indexes[target_push - 1] <= swap_tag_index_i;
                lines[target_push - 1] <= swap_line_i;
            end
        end
    end
endmodule
```

vc_mem_ctrl.v

```
module vc_mem_ctrl(
    input wire clk, co_req, co_rw, co_type, dmc_evict, dmc_hm, dvc_hm, dvc_wb,
imc_evict, imc_hm,
        ivc_hm, mm_resp,

    output wire co_resp, dmc_we, dmc_write_mode, dmc_write_src, dvc_we, dvc_wsrc,
imc_we,
        imc_write_src, ivc_we, ivc_wsrc, mm_req, mm_rw, d_set_swap, i_set_swap
    );

    parameter
        MODE_INST = 'b00,
        MODE_DATA_RI = 'b01,
        MODE_DATA_WB = 'b10,

        IDLE = 'b0000,
        INST_EVICT = 'b0001,
        INST_SET_SWAP = 'b0010,
        INST_SWAP = 'b0011,
        INST_READIN = 'b0100,
        INST_WRITEIN = 'b0101,

        DATA_EVICT = 'b0110,
        DATA_SET_SWAP = 'b0111,
        DATA_SWAP = 'b1000,
        DATA_WRITEBACK = 'b1001,
        DATA_READIN = 'b1010,
        DATA_WRITEIN = 'b1011,
        DATA_WRITE_PREP = 'b1100,
        DATA_WRITE = 'b1101,
        FINISH = 'b1110;

    reg [3:0] state;
    reg [3:0] state_next;
    always @(*) case (state)
        IDLE: if (co_req)
                  if (co_type) begin
                      if (dmc_hm) state_next = co_rw ? DATA_WRITE : FINISH;
                      else if (dvc_hm) state_next = DATA_SET_SWAP;
                      else if (dmc_evict) state_next = DATA_EVICT;
                      else state_next = DATA_READIN;
                  end else begin
                      if (imc_hm) state_next = FINISH;
                      else if (ivc_hm) state_next = INST_SET_SWAP;
                      else if (imc_evict) state_next = INST_EVICT;
                      else state_next = INST_READIN;
                  end
```

```
              else state_next = IDLE;
        INST_EVICT: state_next = INST_READIN;
        INST_SET_SWAP: state_next = INST_SWAP;
        INST_SWAP: state_next = FINISH;
        INST_READIN: state_next = INST_WRITEIN;
        INST_WRITEIN: state_next = FINISH;
        DATA_EVICT: state_next = DATA_READIN;
        DATA_SET_SWAP: state_next = DATA_SWAP;
        DATA_SWAP: state_next = co_rw ? DATA_WRITE_PREP : FINISH;
        DATA_WRITEBACK: state_next = mm_resp ? DATA_EVICT : state;
        DATA_READIN: state_next = mm_resp ? DATA_WRITEIN : state;
        DATA_WRITEIN: state_next = co_rw ? DATA_WRITE_PREP : FINISH;
        DATA_WRITE_PREP: state_next = DATA_WRITE;
        DATA_WRITE: state_next = FINISH;
        FINISH: state_next = co_req ? state : IDLE;
        default: state_next = state;
    endcase

    initial state = IDLE;

    assign co_resp = state == FINISH;
    assign dmc_we = (state == DATA_SWAP) || (state == DATA_WRITEIN) || (state ==
DATA_WRITE);
    // Applicable states and the previous. 1 means full line.
    assign dmc_write_mode = (state == DATA_SET_SWAP) || (state == DATA_SWAP) ||
                                (state == DATA_READIN) || (state == DATA_WRITEIN);
    // 1 means main memory source. Applicable and previous states
    assign dmc_write_src = (state == DATA_READIN) || (state == DATA_WRITEIN);
    assign dvc_we = (state == DATA_SWAP) || (state == DATA_EVICT);
    // 1 means swap data. Applicable and previous states
    assign dvc_wsrc = (state == DATA_SET_SWAP) || (state == DATA_SWAP);
    assign imc_we = (state == INST_SWAP) || (state == INST_WRITEIN);
    // 1 means main memory source. Applicable and previous states
    assign imc_write_src = (state == INST_READIN) || (state == INST_WRITEIN);
    assign ivc_we = (state == INST_SWAP) || (state == INST_EVICT);
    // 1 means swap data. Applicable and previous states
    assign ivc_wsrc = (state == INST_SET_SWAP) || (state == INST_SWAP);
    assign mm_req = (state == INST_READIN) || (state == DATA_WRITEBACK) || (state
== DATA_READIN);
    assign d_set_swap = (state == DATA_SET_SWAP);
    assign i_set_swap = (state == INST_SET_SWAP);
    // Applicable and previous state (given proper signals)
    assign mm_rw = ((state == IDLE) && co_type && ~dmc_hm && ~dvc_hm && dvc_wb) ||
                       (state == DATA_WRITEBACK);

    always @(posedge clk) state <= state_next;

endmodule
```

Block diagram for design with L1 and victim caches



Default Settings used for Vivado power estimation

Junction Temperature	26.183 °C
Ambient Temperature	25 °C
Effective Thermal Resistance	11.533 °C/W
Airflow	250 LFM
Heat Sink	None
Board Selection	Medium 10”x10”
Number of Board Layers	8 to 11
Vccint (Voltage)	1.000 V
Vccaux	1.800 V
Vcco33	3.300 V
Vcco25	2.500 V
Vcco18	1.800 V
Vcco15	1.500 V
Vcco135	1.350 V
Vcco12	1.200 V
Vccaux_io	1.800 V
Vccbram	1.000 V
MGTAVcc	1.000 V
MGTAVtt	1.200 V
MGTVccaux	1.800 V
Vccpint	1.000 V
Vccpaux	1.800 V
Vccpll	1.800 V
Vcco_ddr	1.500 V
Vcco_mio0	1.800 V
Vcco_mio1	1.800 V
Vccadc	1.800 V
Default Toggle Rate	12.5
Default Static Probability	0.5
Clock Period	20 ns

B Future Work Example Code

primeFactors.c

```
#include <stdint.h>

void factor();

void main() {
    int array [32];
    int index = 0;
    factor(15876000, array, &index);
}

void factor(int value, int *array, int *index) {
    int wFactor = 2;
    int lowFactor = 0;
    int highFactor = value;

    while (wFactor < highFactor) {
        if (value % wFactor == 0) {
            lowFactor = wFactor;
            highFactor = value / wFactor;
        }
        wFactor += 1;
    }

    if (lowFactor != 0) {
        factor(lowFactor, array, index);
        factor(highFactor, array, index);
    } else {
       array[*index] = value;
        *index += 1;
    }
    return;
}
```

Excerpt from the primefactors object dump created with the RISC-V GCC compiler

```
00000000000103dc <main>:
   103dc: 7135               addi   sp,sp,-160
   103de: ed06               sd  ra,152(sp)
   103e0: e922               sd  s0,144(sp)
   103e2: 1100               addi   s0,sp,160
   103e4: f6042623              sw  zero,-148(s0)
   103e8: f6c40713              addi   a4,s0,-148
   103ec: f7040793              addi   a5,s0,-144
   103f0: 863a               mv  a2,a4
   103f2: 85be               mv  a1,a5
   103f4: 00f247b7              lui a5,0xf24
   103f8: fa078513              addi   a0,a5,-96
   103fc: 00e000ef              jal ra,1040a <factor>
   10400: 0001               nop
   10402: 60ea               ld  ra,152(sp)
   10404: 644a               ld  s0,144(sp)
   10406: 610d               addi   sp,sp,160
   10408: 00000000             halt

000000000001040a <factor>:
   1040a: 7139               addi   sp,sp,-64
   1040c: fc06               sd  ra,56(sp)
   1040e: f822               sd  s0,48(sp)
   10410: 0080               addi   s0,sp,64
   10412: 87aa               mv  a5,a0
   10414: fcb43823              sd  a1,-48(s0)
   10418: fcc43423              sd  a2,-56(s0)
   1041c: fcf42e23              sw  a5,-36(s0)
   10420: 4789               li  a5,2
   10422: fef42623              sw  a5,-20(s0)
   10426: fe042423              sw  zero,-24(s0)
   1042a: fdc42783              lw  a5,-36(s0)
   1042e: fef42223              sw  a5,-28(s0)
   10432: a815               j   10466 <factor+0x5c>
   10434: fdc42703              lw  a4,-36(s0)
   10438: fec42783              lw  a5,-20(s0)
   1043c: 02f767bb              remw   a5,a4,a5
   10440: 2781               sext.w a5,a5
   10442: ef89               bnez   a5,1045c <factor+0x52>
   10444: fec42783              lw  a5,-20(s0)
   10448: fef42423              sw  a5,-24(s0)
   1044c: fdc42703              lw  a4,-36(s0)
   10450: fec42783              lw  a5,-20(s0)
   10454: 02f747bb              divw   a5,a4,a5
   10458: fef42223              sw  a5,-28(s0)
   1045c: fec42783              lw  a5,-20(s0)
   10460: 2785               addiw  a5,a5,1
```

```
10462: fef42623                sw  a5,-20(s0)
10466: fec42703                lw  a4,-20(s0)
1046a: fe442783                lw  a5,-28(s0)
1046e: 2701                 sext.w a4,a4
10470: 2781                 sext.w a5,a5
10472: fcf741e3                blt a4,a5,10434 <factor+0x2a>
10476: fe842783                lw  a5,-24(s0)
1047a: 2781                 sext.w a5,a5
1047c: c785                 beqz   a5,104a4 <factor+0x9a>
1047e: fe842783                lw  a5,-24(s0)
10482: fc843603                ld  a2,-56(s0)
10486: fd043583                ld  a1,-48(s0)
1048a: 853e                 mv  a0,a5
1048c: f7fff0ef                jal ra,1040a <factor>
10490: fe442783                lw  a5,-28(s0)
10494: fc843603                ld  a2,-56(s0)
10498: fd043583                ld  a1,-48(s0)
1049c: 853e                 mv  a0,a5
1049e: f6dff0ef                jal ra,1040a <factor>
104a2: a02d                 j   104cc <factor+0xc2>
104a4: fc843783                ld  a5,-56(s0)
104a8: 439c                 lw  a5,0(a5)
104aa: 078a                 slli   a5,a5,0x2
104ac: fd043703                ld  a4,-48(s0)
104b0: 97ba                 add a5,a5,a4
104b2: fdc42703                lw  a4,-36(s0)
104b6: c398                 sw  a4,0(a5)
104b8: fc843783                ld  a5,-56(s0)
104bc: 439c                 lw  a5,0(a5)
104be: 2785                 addiw  a5,a5,1
104c0: 0007871b                sext.w a4,a5
104c4: fc843783                ld  a5,-56(s0)
104c8: c398                 sw  a4,0(a5)
104ca: 0001                 nop
104cc: 70e2                 ld  ra,56(sp)
104ce: 7442                 ld  s0,48(sp)
104d0: 6121                 addi   sp,sp,64
104d2: 8082                 ret
```

Bibliography

[1] Chu, Pong P. *FPGA Prototyping Using Verilog Examples: Xilinx Spartan -3 Version*. Wiley, 2008.

[2] EEVblog (2013, Jul. 20) *EEVblog #496 - What Is An FPGA?*. Retrieved from EEVblog https://www.eevblog.com/2013/07/20/eevblog-496-what-is-an-fpga/

[3] Isci, C., & Martonosi, M. (2003) Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. *Princeton University Department of Electrical Engineering.*

[4] Isci, C., & Martonosi, M. (2006) Phase Characterization for Power: Evaluating Control-Flow-Based and Event-Counter-Based Techniques. *Princeton University Department of Electrical Engineering*

[5] Jouppi, N. (1990) Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. *Digital Equipment Corporation Western Research Lab*

[6] Kim, N., Flaunter, K., Blaauw, D., & Mudge, T. (2002) Drowsy Instruction Caches: Leakage Power Reduction using Dynamic Voltage Scaling and Cache Sub-bank Prediction. *Advanced Computer Architecture Lab, The University of Michigan*

[7] *Patterson, D., & Hennessey, J. (2018) Computer Organization and Design: The Hardware/Software Interface, RISC-V Edition. Walthman, MA: Elsevier Inc*

[8] Sanguinetti, J. (2002) *Introduction to Verilog. Retrieved from http://vol.verilog.com/*

[9] "The RISC-V Instruction Set Manual." Edited by Andrew Waterman and Krste Asanovic, *RISC-V Foundation*, 7 May 2017, riscv.org/specifications/.

[10] Xilinx *Power Estimation and Analysis using Vivado*. Retrieved from Xilinx https://www.xilinx.com/video/hardware/power-estimation-analysis-using-vivado.html