A CONFIGURABLE RISC V PROCESSOR CORE FOR FPGA DEVICES

A Project

Presented to the

Faculty of

California State Polytechnic University, Pomona

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

In

Electrical Engineering

By

Benjamin K. Kueffler

2019

SIGNATURE PAGE

PROJECT:	A CONFIGURABLE RISC V PROCESSOR CORE FOR FPGA DEVICES
AUTHOR:	Benjamin K. Kueffler
DATE SUBMITTED:	Fall 2019
	Department of Electrical and Computer Engineering

Dr. Mohamed El-Hadedy
Project Committee Chair
Assistant Professor

Dr. Anas Salah Eddin
Assistant Professor

Dr. Halima El Naga
ECE Department Chair

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor Dr. Mohamed El-Hadedy for his support on this project. His encouragement has allowed me to delve further in the field of computer architecture and achieve success with this project. Secondly, I'd like to thank my colleagues at Northrop Grumman for their understanding of this undertaking and encouragement of my education. Without support from these individuals, family, and friends, this project would not be possible.

ABSTRACT

Reduced instruction set computers (RISC) have benefitted embedded platforms for decades, allowing devices to gain software support with minimal cost in hardware implementation. Stemming from this success, the open source RISC V instruction set architecture (ISA) was created with flexibility and stability in mind. Due to these features, RISC V has been adopted in the industry and has taken a significant share of the market. The flexibility of RISC V allows for a variety of different hardware implementations. This project takes advantage of this flexibility to create a customizable processor core for reprogrammable devices.

This project consists of a processor targeting field-programmable gate array (FPGA) devices in an embedded environment. In order to take advantage of the reprogrammability of these devices, this processor specializes in its configurability. Different hardware configurations are able to be selected at synthesis time, allowing for increased hardware flexibility depending on the application. The performance, power, and utilization of this RISC V core are analyzed for their usefulness in an embedded environment and compared to a similar RISC processor, the Xilinx MicroBlaze.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Reduced instruction set computers (RISC) are invaluable for embedded and general purpose computers in order to provide a cost effective processor compared to their complex instruction set computer (CISC) counterpart. It’s estimated that 99% of processors used today are RISC based [1]. The presence of RISC is staggering in the industry and stems from the simplicity of its instruction set, allowing for instruction decoding within the hardware to be simpler. This results in smaller hardware with comparable performance. RISC designs have gained popularity in part due to compliers becoming more capable, allowing for complex software designs to be broken down into many simple instructions in a way that was previously only possible with a CISC instruction set architecture (ISA) [1].

The RISC V instruction set architecture (ISA) is intended to be a flexible and accessible solution for RISC designs. It allows for a small and stable base of instructions with the ability to scale up by adding extensions. Due to this flexibility, it’s desired to build a processor which can take advantage of the architecture in a configurable way. Furthermore, field-programmable gate array (FPGA) devices utilize soft processors and can benefit from flexible hardware due to the ability to quickly reconfigure. This project consists of a RISC V processor compliant with the RV32I base instruction set on an FPGA targeting embedded applications. In addition, the utilization, performance, and power will be gauged in order to verify the effectiveness of the embedded processor implementation.

1.1 The RISC V Architecture

The RISC V ISA takes the principles of RISC and adds additional flexibility while simplifying the baseline architecture. RISC V is a modular design with a base ISA and the ability to extend the number of instructions with ISA extensions [2]. RV32I, RV32E, RV64I, and RV128I represent the base instruction sets consisting of 32, 64, and 128 bit width respectively. The RV32E base is an ISA specialized for embedded applications and reduces the number of registers from 32 to 16. In addition to these baseline architectures, there are numerous extensions that are finished or in progress. An example of one of these extensions is M, which extends the base with multiplication and division operations. The purpose of these base architectures and their extensions is to reduce the complexity required for any one single design. In CISC or certain RISC ISAs, a system may want to only use certain instructions, but the CPU for these systems will still be required to have logic for the entire larger instruction set. With a modular instruction set, this problem is significantly reduced, since a processor can be designed to only support a base ISA and any number of extensions [3].

In addition to modularity, the RISC V architecture is also open source. Not only does this reduce the cost of entry to developing a RISC V implementation, but also increases security. The visibility of all instructions to the user means that there aren't any unknown instructions when running code. This reduces issues with security vulnerabilities from running unknown instructions, as is the case with x86 devices [4].

Removing any royalty fees associated with the ISA and increasing modularity has led to widespread adoption of the RISC V architecture across the industry. Companies such as Nvidia, Western Digital, and Samsung have begun creating devices using the

RISC V ISA [5]. This increased adoption of RISC V also increases the number of available extensions and software support for the ISA.

1.2 RISC V with FPGAs

Due to their reprogrammability, FPGAs can take advantage of the flexible nature of the RISC V ISA. An FPGA designer may build a processor targeting the base RV64I without complex execution units such as a floating point unit. If a software update adds many floating point instructions, the processor can be updated with a floating point unit without significantly changing the processor design. On an FPGA this is exceptionally beneficial, since the processor can be generated with minimal turnaround time. In order to reduce the time between hardware implementations further, these ISA extensions can be enabled or disabled in the RTL, providing automated configuration based on the architecture selected [6].

Like other RISC processors, RISC V processors implemented in FPGAs often utilize bus protocols for IO and memory transactions. For this project’s processor, the Advanced eXtensible Interface (AXI) was utilized for interaction with the rest of the FPGA. AXI allows for fast data transmissions via data bursts between the master and slave [7]. A processor using the protocol can access instructions or data from external memory in bursts, allowing for writing or reading from blocks of the cache. These bursts are necessary to reduce the penalty of memory access and increase overall performance of the processor.

CHAPTER 2: RELATED WORK

The usefulness of RISC V as an ISA has led to a large number of implementations, ranging from embedded implementations to high performance, multicore designs.

2.1 Berkeley Out of Order Machine (BOOM)

The BOOM implementation of the RISC V processor represented an out-of-order, superscalar approach to designing a RISC V core. The advantage that this processor brought was that instructions could be issued out of order, allowing for less stalls while waiting for instructions to complete. The superscalar nature of the processor meant that multiple instructions could be fetched and executed, allowing for higher performance than a traditional scalar processor [8]. The BOOM processor was intended as a high performance design as a prototype for research on out-of-order microarchitectures [9]. This processor achieved impressive speeds on a 45 nm ASIC device, with 1.5 GHz operation and a CoreMark/Core score of 7,050 [10]. However, since it was designed with ASICs in mind, when porting to a Xilinx UltraScale+ FPGA, it was only able to achieve 90 MHz [10]. Also, the BOOM processor lacked a lot of configurability, and it’s unclear whether it would be able to easily change from its RV64G configuration to a RV32G configuration or similar. This project’s processor addresses some of these concerns. Although this work’s processor cannot achieve the performance metrics as BOOM, it contains a more configurable design, allowing for changes and optimizations around a given application. In addition, this project targets implementation on an FPGA, allowing for significantly higher clock frequency than what was achieved by the BOOM processor.

2.2 SiFive

SiFive is a company created to generate and fabricate RISC V implementations, and provides RISC V customizable cores for both high performance and low utilization implementations. One of the main benefits of SiFive Cores is the customizability. A processor can be designed around a platform such as “high performance”, “most efficient”, or “Linux-capable” [11]. Once a platform has been selected, the processor can be customized in more detail, for example, cache size and bus protocol can be configured. This provides a way for a platform to design a core around the type of software processing it requires. The configuration, however, is not as granular as desired. For example, there doesn’t appear to be an easy way to add a cache for the most efficient designs. If a user wants to add a cache, or upgrade the associativity of the SiFive E2 series cache, then the entire processor would have to be upgraded to the next series. This discrete number of options likely reduces the amount of verification required, but can limit the configurability of the core.

2.3 VexRiscV

The VexRiscV processor is a configurable 32-bit processor core written in SpinalHDL. The use of SpinalHDL means that the underlying Verilog code can be generated based on the configuration selected. This allows for an impressive number of configuration options at a detailed level. The processor can be reduced to 494 look up tables (LUTs) and 505 flip flops (FFs) with the smallest configuration to 2,530 LUTs and 2013 FFs with the Linux capable configuration [12]. In addition to this, VexRiscV is designed with FPGAs in mind, leading to commendable utilization and performance on an FPGA platform. For Xilinx Artix 7 devices, the smallest implementation reaches 233

MHz, while the Linux capable configuration reaches 170 MHz [12]. The ability to build an efficient processor for a reprogrammable platform allows for fast configuration changes to optimize for a particular algorithm or power requirement. This implementation differs from the proposed RISC V core in how the pipeline was designed. Both processors contain a five stage pipeline, however, jumps are handled different. Jumps in the proposed core occur during the decode stage, while jumps in the VexRiscV core occur in the execute stage. Executing jumps earlier means that this work's processor can fetch the new instruction faster and reduce the number of stalls required, however, it does require extra logic in the decode stage in order compare the opcode after decoding. The proposed core is most similar to the VexRiscV core than to the other related work, however, the implementation language and details of the pipeline vary between the two implementations.

CHAPTER 3: PROCESSOR & SYSTEM

The designed processor is a 32-bit RISC V processor written in SystemVerilog, targeting the RV32I base instruction set. The processor is designed to function on an AXI4 bus and acts a master to perform reads and writes to memory mapped IO or internal memories. The processor architecture is designed with FPGAs and embedded systems in mind. An emphasis was placed on reducing power and utilization. Parameters were utilized to provide customization to the core based on the system or application.

3.1 System Context

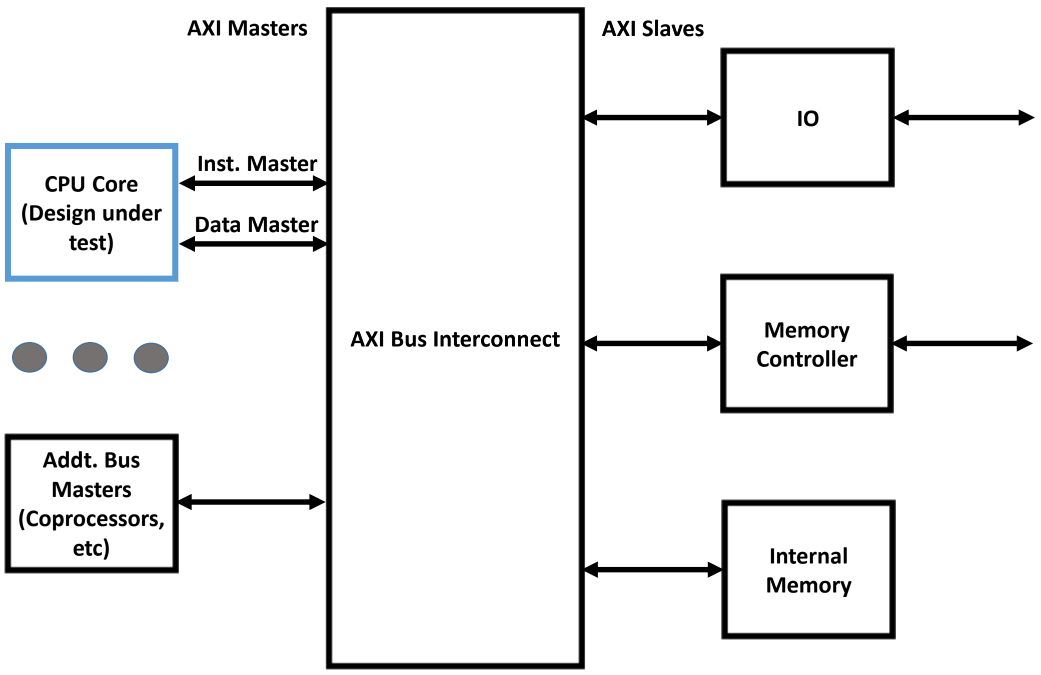


Figure 1. A typical system with the processor core implemented on the bus.

The processor core designed is intended to be used in an AXI based FPGA design (Figure 1). The processor, shown in blue, acts as a master and can initiate transfers to any one of the slaves on the bus. It was designed to have two independent AXI4 masters. One master is contained in the instruction cache of the fetch stage. The master acts as a reader, and commands the AXI R and AR channels in order to carry out read bursts. This allows for reading of memory space on the bus in bursts, enabling blocks to be loaded into the cache in order to obtain instructions. The other master is contained in the data cache of the memory access stage. This master acts as both a reader and writer, and commands the AXI AW, W, AR, R, and B channels in order to carry out writes and reads. The data master can carry out reads in the case of load instructions, and writes in the case of store instructions. It will also carry out writes to memory when block replacement occurs from the cache. The data master is more complicated than the instruction master, since it uses both the read and write channels of the AXI protocol.

Some architectures, such as the E2 core from SiFive, have an arbitrated data and instruction master, so that only one master interacts with the bus at time [11]. This can reduce the amount of logic needed in the core and interconnect, but causes more stalls and reduces performance. The approach taken with this project’s core was to create two separate instruction and data interfaces in order to not cause unnecessary stalls.

The use of AXI allows for additional masters or slaves to be added to the bus. Coprocessors, such as a direct memory access (DMA) blocks or other processing cores, can be added to the master side. This allows for accelerating processing concurrently alongside the CPU core. Typical slaves on the AXI bus include internal and external memory, allowing for an area to store the instructions and data. In addition to memory, embedded systems benefit from memory mapped IO, which allow the processor to control peripherals such as communication devices [13].

3.2 Processor Architecture

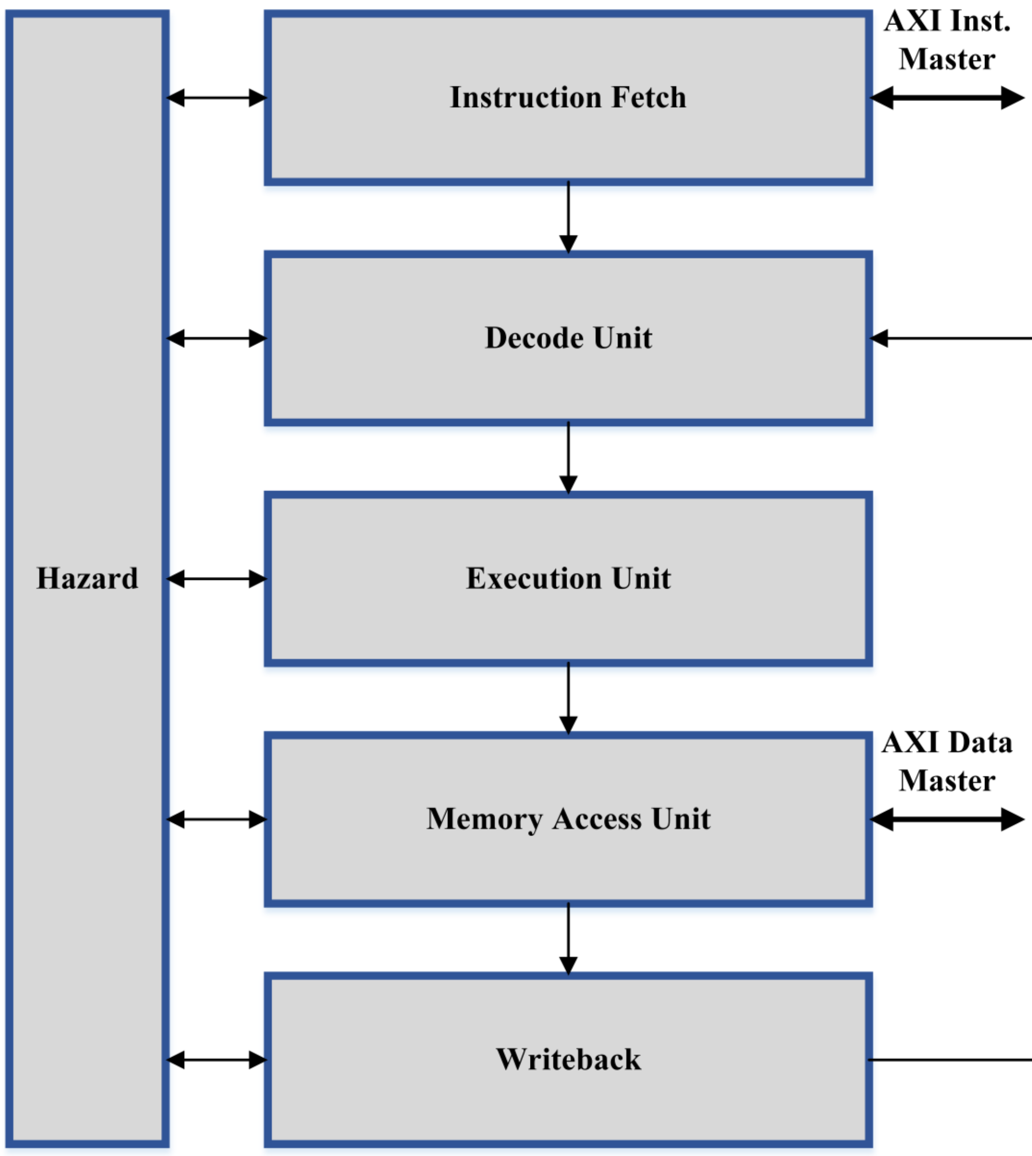


Figure 2. Top level processor architecture.

The processor architecture consists of a five stage pipeline with a hazard unit handling data forwarding and enabling the units. The instruction fetch stage accesses the AXI bus in order to load instructions into the cache. The 32-bit instructions are decoded in the decode unit in order to determine the operation. The decode stage handles the jump & link (JAL) operation (APPENDIX B: Supported Instructions (RV32I)) and sends the branch address to the fetch stage. Next, the operation is sent to the execution unit along with the selected registers and the operation is performed. Then, the data is sent to the memory access unit, and if a load or store is required, the data cache is accessed. After the data cache outputs valid data, or if the instruction was not a load or store, the data arrives in the writeback unit. Finally, the writeback unit writes to the register file found in the decode unit in the case of a register writing instruction. The hazard unit accesses all blocks to forward data, enable the units, and send branch information to the fetch stage.

3.2.1 Instruction Fetch

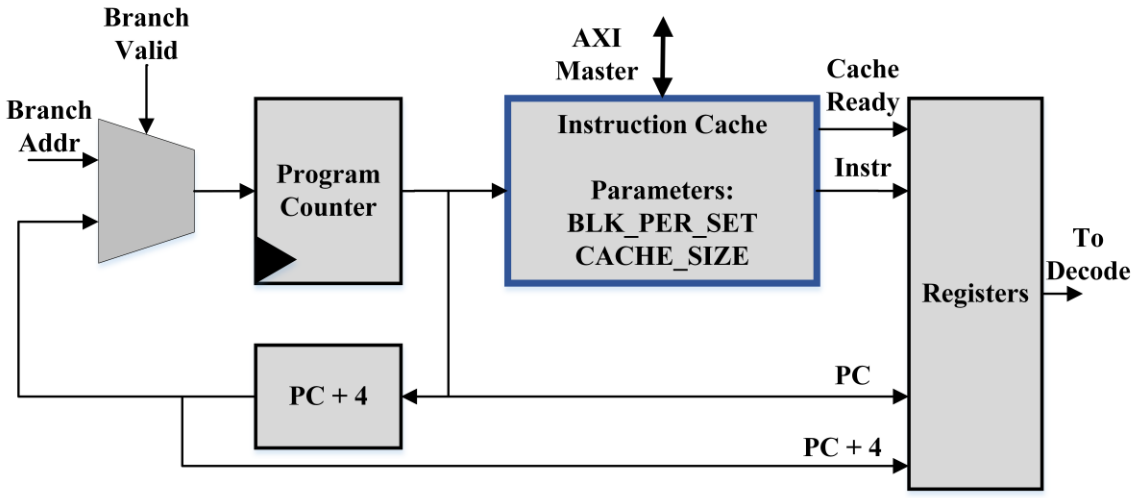


Figure 3. Instruction fetch architecture.

The instruction fetch stage is responsible for retrieving the instruction from memory and sending it to the decode stage. It is also responsible for accepting branches from the hazard unit, which indicate that the program counter must move to the new branch address. The instruction fetch uses a multiplexer to determine whether it should use the next value of the program counter (PC + 4) or the branch address. The determined value is registered and called the program counter (PC). This PC is used as an address into the instruction cache. When the instruction fetch block is enabled, it begins accessing the instruction cache at the address of the PC. If the PC given to the instruction cache is found within the cache, otherwise known as a “cache hit”, then the instruction can be output on the next cycle. However, if the instruction corresponding to the PC is not found within the cache, a “cache miss” will occur. This cache miss will cause the cache to retrieve the instruction over the AXI bus, and No-Op instructions will be propagated to the decode stage until the cache becomes ready. In addition to sending the instruction to the decode stage, the fetch unit will also propagate the PC and PC + 4. These PC values will be used in future stages for instructions involving the program counter.

The instruction fetch stage contains parameters that can be used to configure the instruction cache as well as starting program counter location. The PC_BASE_ADDR location is necessary to determine the program counters value upon startup. The CACHE_SIZE parameter determines how large the cache is; this will have an effect on the amount of logic and RAM needed. The BLK_PER_SET parameter controls the number of ways within the cache. A cache with multiple ways will have less need to replace blocks than a cache with only one way, decreasing the miss rate and improving performance [14]. The number of ways in the cache has a large impact on the amount of logic for the cache controller, which makes it an ideal feature to leave as a parameter to use at the designer’s discretion.

Table 1. Configurable parameters for the instruction fetch unit.

Parameter	**Default value**	**Description**
PC_BASE_ADDR	0	The starting address for the program counter.
ADDR_SIZE	32	The size of the accessible address space in bits
CACHE_SIZE	2 ^ 14	The size of the cache desired in bytes

BLK_PER_SET	2	The number of blocks per set in the cache. This determines the number of “ways” of the cache, e.g. BLK_PER_SET = 2 is 2-way set associative

3.2.1.1 Instruction Cache

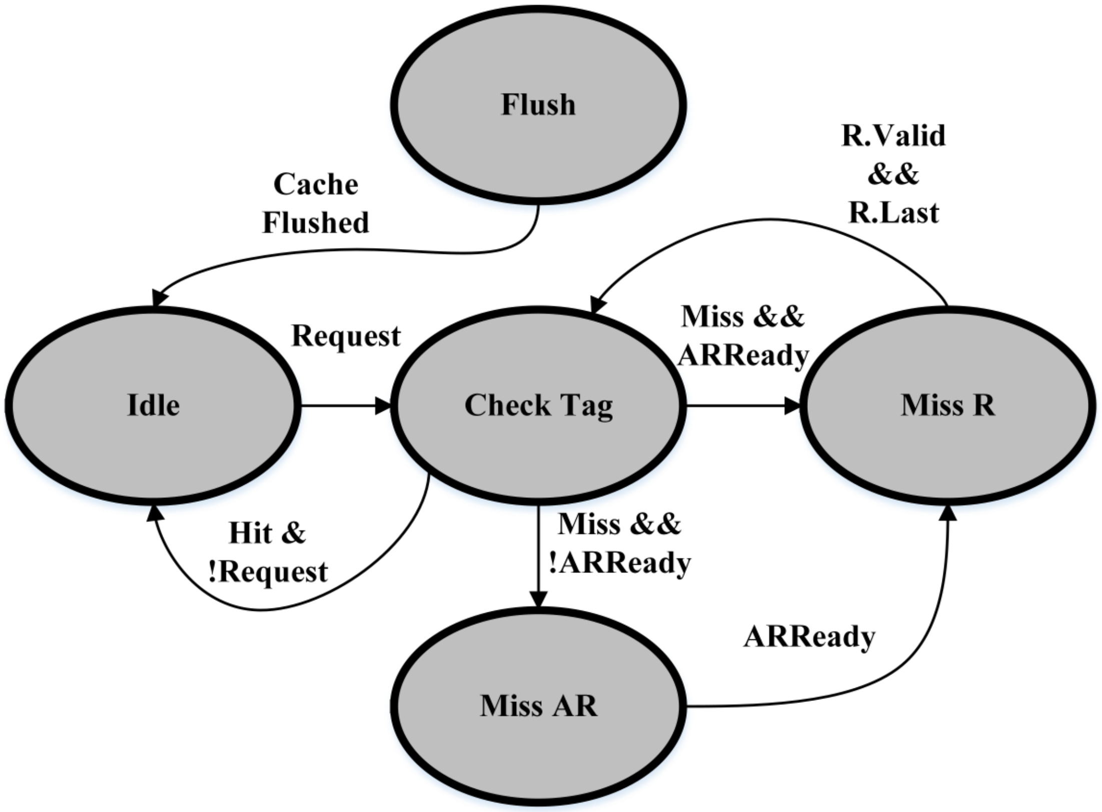


Figure 4. State machine for the instruction cache.

The instruction cache consists of a state machine and memory located within the instruction fetch unit of the processor. The memory of the cache is intended to be block RAM inside of an FPGA, though similar internal memory structures are just as suitable. The state machine begins by setting the valid bits of the memory to zero in the *flush* state. The purpose of this is to invalidate the cache, so the values in the cache upon reset are not relied on until they are valid in operation. This allows the design to be more portable, since some FPGA devices will not initialize the values in the cache to zero. The cache is ready for operation once all the valid bits are set to zero. After flushing, the instruction fetch unit can send a request alongside an address to transition the cache to the *check tag* state. This state compares the tag bits of the cache from each of the ways in order to determine if the address that was requested matches any address found within the cache. The *check tag* state produces a cache hit or miss depending on the outcome of the tag comparison.

In the case of a hit, the cache will return to idle unless another request is desired. A miss will cause the cache to issue an AXI read using the AR and R channels in order to populate the block within the cache. An AR command will be sent to the bus with a burst length equal to the size of the block. After the bus responds, the cache will begin receiving read data over the R channel until its block is updated. Once the block is updated, it will return to the *check tag* state, and a hit will occur on the newly obtained block.

A cache miss will result in stalls while the cache block is being written. The block size is 8 32-bit words, or 32 bytes. This number was chosen since it gives a good balance between miss penalty and miss rate. Due to the 8 word block size, a miss will cause 8 words to be written. An additional cycle is needed for the state machine to transition, so the miss penalty is at least 9 clock cycles. More clock cycles are needed if the AXI slave requires additional access time.

3.2.2 Decode Unit

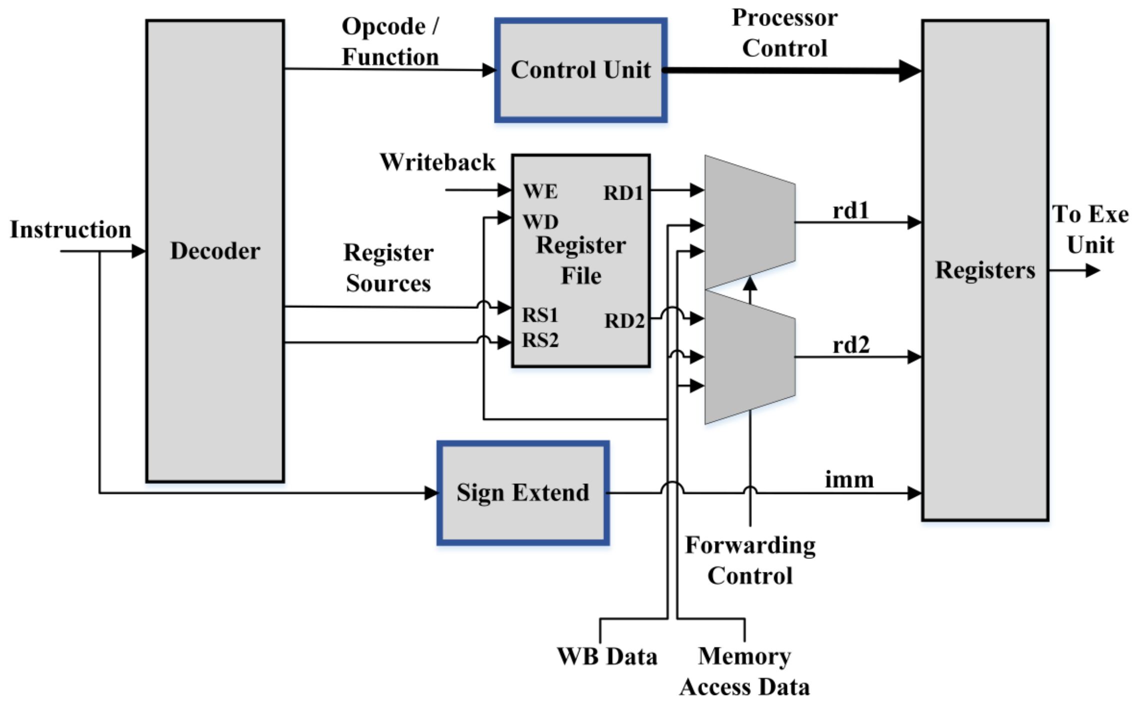


Figure 5. Decode unit architecture.

The decode unit is responsible for receiving the 32-bit instruction from the fetch stage and decoding it into an opcode, function, operands, and immediate. The register file and control unit also exist within this stage of the pipeline. The RISC V register file consists of 32 registers. Register zero is a special purpose register used to hold the constant value zero. Registers 1-31 function as regular registers from the perspective of the hardware.

The control unit with the decode stage is critical to assigning the control bits for the rest of the processor stages. The control unit receives the opcode and function (funct7, funct3) fields after decoding, and assigns each corresponding control signal based on the instruction. The control signals are routed through the pipeline to each of the stages in order to enable operations on an instruction-by-instruction basis. The list of control signals found in the processor are described in Table 2.

Table 2. Processor control signals from the control unit.

Control Field	**Description**
reg_write	When ‘1’, a write will be performed on the register file.
memtoreg	Determines if the ALU output, data memory, or PC + 4 will be written to the register file in the writeback stage.
memwrite	Determines if the memory operation will be a write (1) or read (0).
memaccess	Determines if the operation will access the dcache.
aluop	Determines the ALU operation in the execution unit.
alu_srca	Determines which operator port A of the ALU will utilize. (0 – Register, 1 – PC, 2 – Zeroes)
alu_srcb	Determines which operator port B of the ALU will utilize. (0 – Register, 1 – Immediate)
jalr	Indicates if the operation is a jump & link. Causes execution unit to always branch.
brop	The branch operation for the branch unit in the execution stage.

sysop	The system operation for the system operation unit in the execution stage.
exe_unit	Determines what execution unit will be used in the execute stage.
ldop	Determines the load instruction for the memory access stage.
sop	Determines the store instruction for the memory access stage.

The decode unit receives forwarding information from the hazard unit in order to send up to date register information to the execution stage. The hazard unit detects when a read after write hazard will occur and will forward the latest register value from the memory access or writeback stages to earlier stages in order to prevent these errors. Note that unlike some other architectures, the register file will not complete its write and output the new read data mid-cycle, so forwarding the writeback data is necessary. This is to comply with FPGA architectures which lack DDR functionality.

3.2.3 Execution Unit

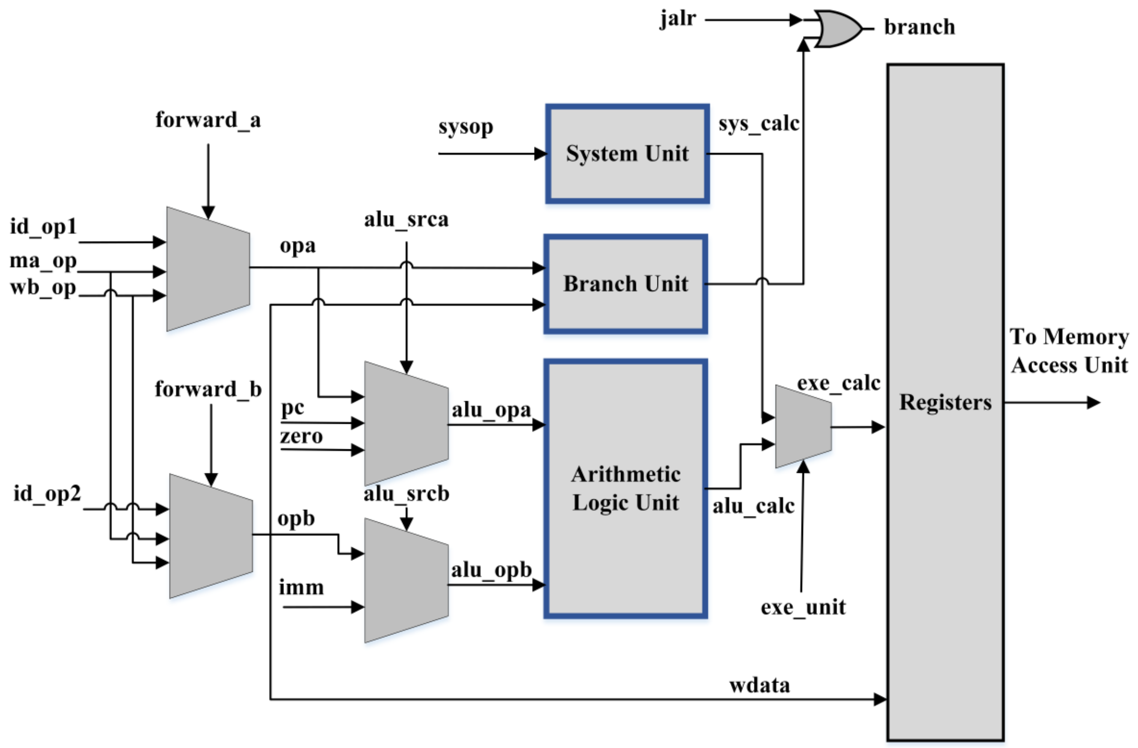


Figure 6. Execution unit architecture.

The execution unit receives the operands and control signals from the decode stage in order to perform the instruction execution. Several submodules are instantiated within the execution unit in order to process the different instruction types. The arithmetic logic unit (ALU) is utilized for arithmetic and logical operations. It receives operand A and B to process calculations with. These operands are the result of two levels of multiplexers. The first is a forwarding multiplexer, which forwards register operands from later stages. The hazard unit determines when to forward data in a similar manner as the decode stage. The second stage of multiplexers determine whether to use the registers, immediate, program counter, or the value zero for the operand. The use of multiplexers in the second stage allows for non-register based instructions to utilize the calculation units. For example, the AUIPC instruction requires addition between the PC and the immediate. In this case, the control unit selects those two operands instead of the register data. Once the calculation is complete, the result of the calculation is sent to the memory access stage.

Similar to the ALU, the branch unit within this stage accepts two operands. The branch unit then performs a comparison to detect if a branch has occurred. If the branch condition is met, and the instruction corresponds to a branch, then the branch address will be sent to the fetch stage to become the new program counter. Aside from branch instructions, the jump & link register (JALR) instruction will also trigger a jump in this stage. A control signal indicating the instruction is JALR arrives from the control unit in order to cause this unconditional jump.

The system unit within this stage handles the system instructions (APPENDIX B: Supported Instructions (RV32I), which consist of many different counters. The system unit is simpler than the ALU or branch units, because it does not need to receive any operands. Several timers are present within the system unit in order to output the status information. This status information is then sent through the pipeline to be written back to a register.

3.2.4 Memory Access Unit

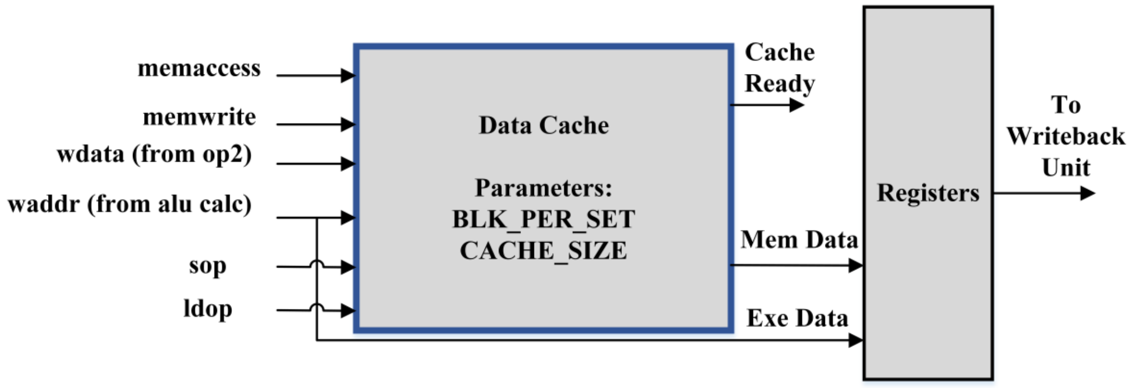


Figure 7. Memory access unit architecture.

The memory access unit processes load and store instructions by accessing a data cache. The control unit signals *memaccess* and *memwrite* determine whether the cache will be accessed and whether a store or load will be processed. The *sop* and *ldop* signals determine the type of store or load respectively. The write data into the cache is sent from operand 2 of the execution unit, while the calculated output from the execution unit is used as the address.

The cache itself is similar to the cache found in the instruction fetch stage. It allows for a configurable associativity and cache size. The cache may take additional cycles in order to carry out a store or load, this will result in stalls in the pipeline. If the instruction is a load, then the memory access unit will output the loaded data alongside the ready signal. After this step is complete, the retrieved data is sent to the writeback stage to be written to a register. For all instructions that are not load and stores, the output of the execution unit will passed through this stage to be written to a register.

3.2.4.1 Data Cache

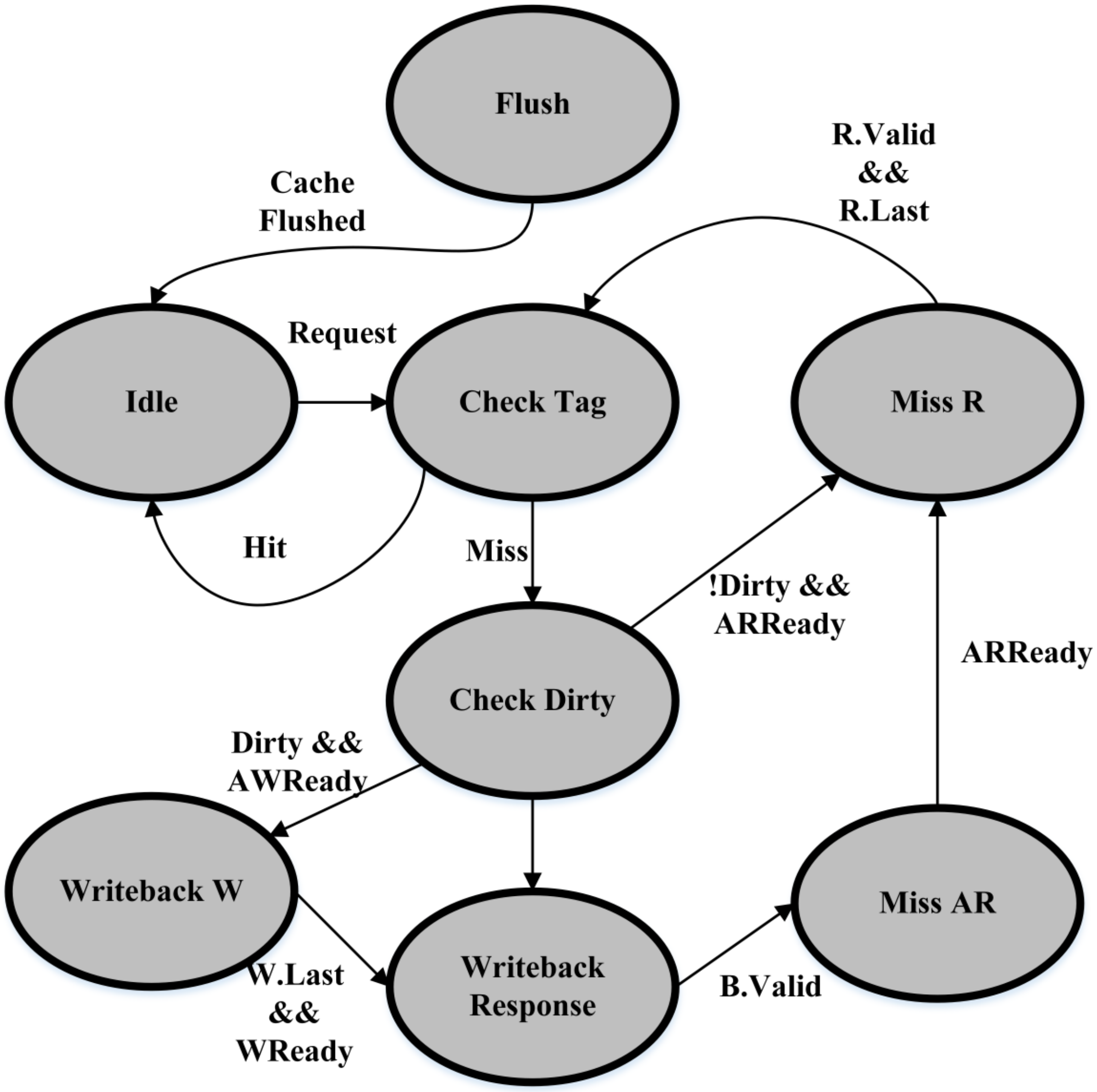


Figure 8. State machine for the data cache.

The data cache contains the cache memory as well as a controller to obtain and write data over the AXI bus. The data cache controller differs from the instruction cache controller because it must perform writes into the cache and bus in order to store data. As a result of this, all of the AXI channels are utilized, and the controller is more complex.

Upon reset, the data cache controller proceeds to the *flush* state. The *flush* state clears the valid bits of each element within the cache, this prevents erroneous cache hits from occurring on startup when the RAM is uninitialized. After the cache is flushed, the data cache may be accessed from the memory access stage. Upon receiving a store or load request, the cache enters the *check tag* state. This state is similar to the instruction cache, with the tag bits being checked against the requested address in order to determine if the cache has access to the area of memory requested. If a hit is found, the data cache will store or return the data and return to the idle state. If a miss occurs when checking for the tag, the data cache enters the *Check Dirty* state.

The *Check Dirty* state reads the “dirty” bit field of the cache block. The dirty bit indicates that a write has been performed on the cache and the block has not been written back to memory. In other words, the dirty bit is a status that indicates the external memory block may not match the cache block being replaced. The data cache is a writeback style cache, meaning writes to the memory space are only done when the block is replaced. Checking the dirty bit allows the cache to determine if it should proceed with a writeback before replacing the block. If the block is determined to be dirty, an AXI write request (AW) is performed and the *Writeback W* state is entered. If the block is not dirty, then the block can be replaced in the same way as the instruction cache, through the *Miss R* state.

In the case of a write to memory, the AXI bus will be accessed in order to write the entire dirty block to memory. This process is similar to performing block reads, but the AW, W, and B channels of AXI are utilized. During this process, 8 words will need to be written over AXI before receiving the write response. Once the write is complete, the block can be replaced by performing a read over AXI. The cache miss with replacement takes approximately 20 clock cycles in total, assuming the AXI slave does not add any additional delay. Since this process can be costly for the general performance of the processor, it’s important to reduce the amount of miss and replacements by increasing the cache associativity and size to meet the performance requirements of the system.

3.2.5 Writeback Unit

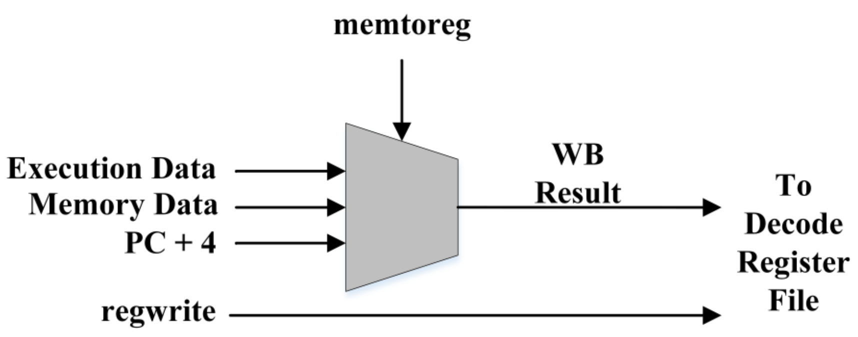


Figure 9. Writeback unit architecture.

The writeback unit receives three words from the memory access stage and multiplexes the words to write to a register in the decode stage. The control unit determines the value of *memtoreg*, which is used to select which word to write to a register. In the case of a load, the memory data will be selected. In the case of a jump & link instruction, the value of PC + 4 will be sent to the register file to perform the link of the old program counter. Otherwise, the execution data is sent to the register file, this ensures that the calculated value from the execute stage is written.

CHAPTER 4: SOFTWARE & TEST

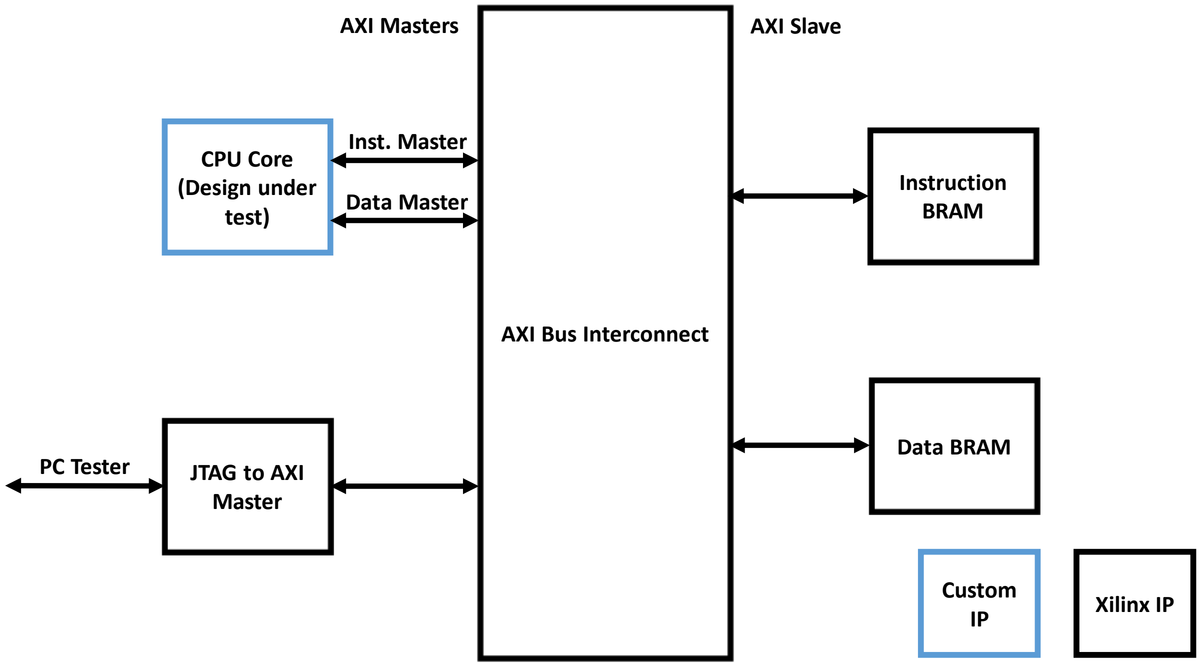


Figure 10. FPGA design for testing the processor.

Software was created in order to test performance and verify functionality of the processor. To test the processor's performance accurately, an FPGA implementation was chosen to emulate a real system and provide debugging capabilities to gauge performance and success. The design, as shown in Figure 10, consisted of the CPU core residing as a master on the bus. The slaves consisted of two AXI Block RAM (BRAM) modules in order to act as memory for the instructions and data. Internal memory was chosen as the storage unit in order to reduce access time, since the purpose of performance testing was to gauge the performance of the core, not the memory. The JTAG to AXI Master acted as an additional master on the bus. It allowed instructions and data to be stored over a JTAG connection from the host PC to the AXI memory. In addition to allowing for the program and data under test to be stored, the JTAG master allowed the memory space to be read after the program completed. Each program that was tested stored the result of the test as well as the number of clock cycles the test took to complete. The number of clock cycles was calculated internally in the system unit of the processor, and was read by using the RDCYCLE instruction.

4.1 RV32I Compliance

The RV32I compliance test was the first program tested on the processor. The test is a compliance test for the RV32I subset of RISC V instructions. Compliance tests are utilized in order to verify that a processor executes the instructions correctly. The RV32I program verifies functionality by running every instruction and checking that the instruction completed successfully. The test is a built-in self-test, so any instruction which does not return the expected result causes the processor to entire a failure state. This failure state results in a write to the data memory, which allows the JTAG master on the bus to read the state and report the results.

```
# Logical Test
li x4, 4
li x31, 6
xor x5, x2, x4
bne x31, x5, fail
xori x6, x2, 0x12
li x30, 0x10
```

```
bne x30, x6, fail

# …

fail:
    li x31, 0xfa110001
    li x30, STATUS_MEM
    li x29, (STATUS_MEM + CACHE_SIZE)
    li x28, (STATUS_MEM + 2 * CACHE_SIZE)
    sw x31, 0(x30)

    # Flush cache to write the status to memory
    sw x0, 0(x29)
    sw x0, 0(x28)

    jal x1, fail
```

Figure 11. Code snippet from RV32I compliance test assembly, showing XOR and XORI built-in self-test

4.2 Median Filter

A median filter is a commonly used benchmark for processor designs and has practical uses in signal processing applications by reducing noise. In addition to this, the instructions required to implement a median filter fall within the RV32I instruction set, so the program can be efficiently compiled for an RV32I processor. The median filter therefore allows an accurate and useful measurement of performance of the processor beyond just verification.

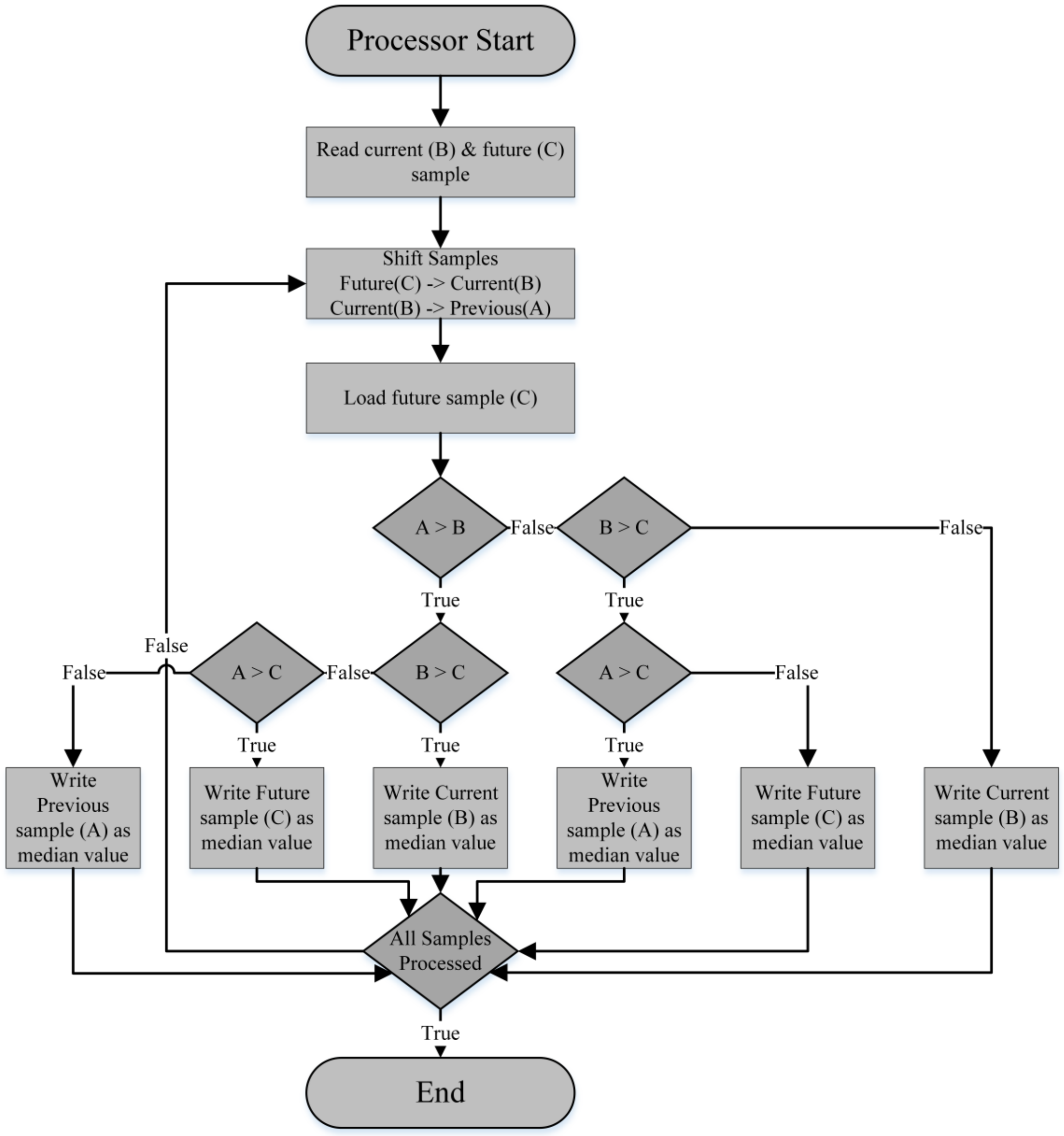

Processor Start
Read current (B) & future (C) sample
Shift Samples Future(C) -> Current(B) Current(B) -> Previous(A)
Load future sample (C)
A > B
False
B > C
False
True
True
False
False
A > C
False
B > C
A > C
False
True
True
True
Write Previous sample (A) as median value
Write Future sample (C) as median value
Write Current sample (B) as median value
Write Previous sample (A) as median value
Write Future sample (C) as median value
Write Current sample (B) as median value
All Samples Processed
True
End

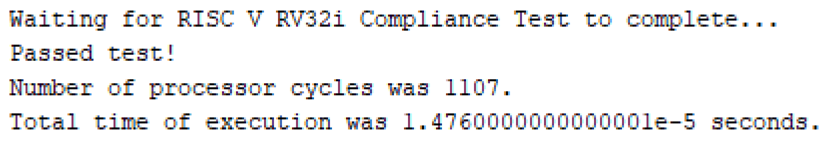
Figure 12. Median filter process flow diagram.

In order to test the median filter for performance, a noisy signal is stored in the data memory before the processor is started. This ensures that no additional IO latency is added to the system for performance testing. After storing, the processor is started. The median filter program performs load instructions to load the first three samples into the processor’s register file. It then begins a sorting algorithm, shown in Figure 12, which determines the median sample. Once the median sample is determined, this value is stored into memory. When all median samples are stored into memory, the program ends and the number of cycles is recorded in memory. The JTAG master can then read the filtered samples out of memory in order to obtain the filtered waveform.

CHAPTER 5: RESULTS

The following details the output from each of the programs that were executed on the RISC V processor. These programs were all run on a Xilinx Spartan 7 series FPGA (XC7S50-CSGA324) at 75MHz. The instruction cache was set to be 8kB direct mapped. The data cache was configured to be 16kB 2-way set associative.

5.1 RV32I Compliance



```
Waiting for RISC V RV32i Compliance Test to complete...
Passed test!
Number of processor cycles was 1107.
Total time of execution was 1.4760000000000001e-5 seconds.
```

Figure 13. Log from RV32I compliance test indicating success.

The RV32I compliance test and verification script were run in order to verify that the memory contents of the system matched the expected results. The output of the verification script is shown in Figure 13. Although the RV32I compliance test did not gauge the performance of the processor, it verified the processors functionality and reliability when running software programs.

5.2 Median Filter

To test the median filter, noise was added to a sine waveform in Octave to generate a noisy dataset. In order to verify and compare the results of the median filter, the Octave function medfilt1 was utilized on the noisy data. The processor that ran the medfilt1 function was an Intel i7-4710HQ CPU at 2.50 GHz.

```
% Number of bytes
b = 4;

% Number of samples
n = 60000 / b;
k = 0:(n - 1);
```

```
% Frequency
w = 2;

% Variance of noise
var = .1;

% Clean Signal
d = 2 * sin(2 * pi * w * (k / n));

% Noisy Signal
xa = int32(1000 * (d + randn([1,n]) * sqrt(var)))';

% Start a wall-clock timer
tic();

% Perform median filter for verification
xf = medfilt1(xa, 3);

% Record the time it took to perform the median filter
elapsed_time = toc();
```

Figure 14. The median filter implemented using medfilt1.

```
>> gen_signal
Medfilt1 process time: 1.078129 ms.
```

Figure 15. The process time for the median filter using medfilt1.

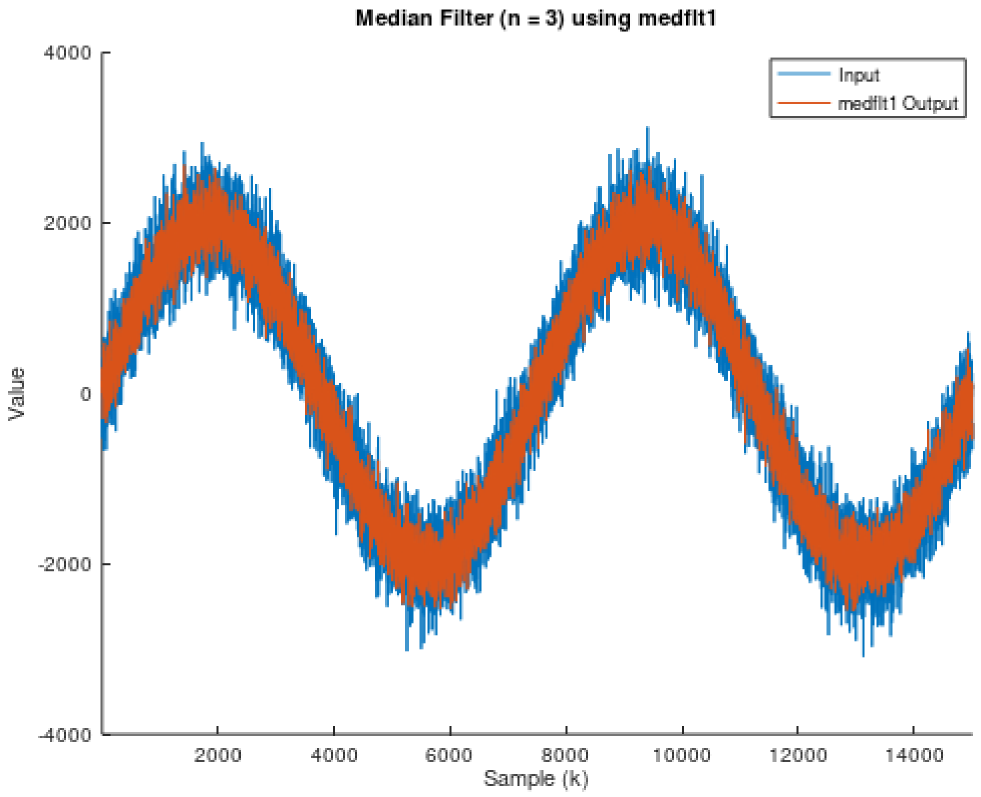
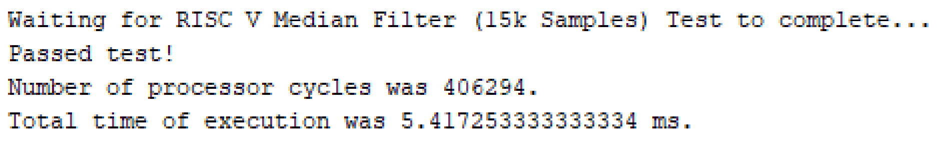


Figure 16. Noisy input data plotted with medfilt1 filtered output using Octave.

The same generated noisy dataset was then stored within the data memory of the FPGA and the processor was started. A script was utilized to verify that the processor had finished filtering the data, and then the data was read to a file on the test PC.



```
Waiting for RISC V Median Filter (15k Samples) Test to complete...
Passed test!
Number of processor cycles was 406294.
Total time of execution was 5.417253333333334 ms.
```

Figure 17. Log from median filter test indicating performance.

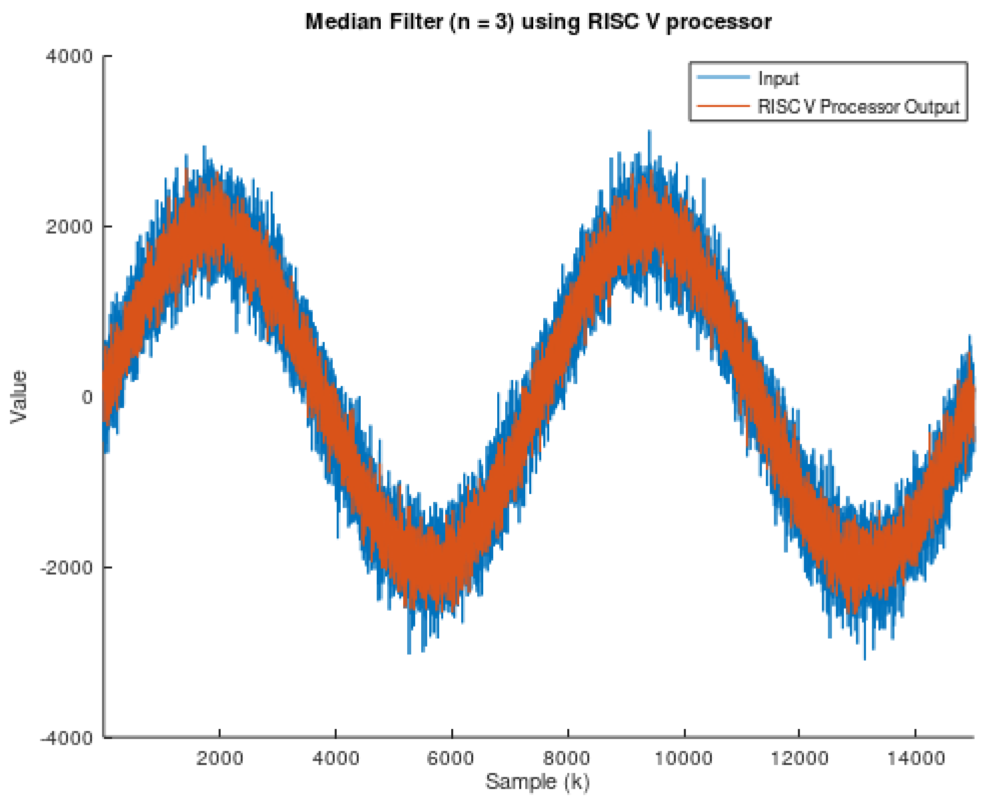


Figure 18. Noisy input signal with filtered output from processor.

The output from the processor is shown in Figure 18. The processor successfully filters the noisy input using the median filter program and matches with the Octave medfilt1 function.

5.3 Utilization

Utilization was recorded hierarchically in order to determine the impact of each module within the processor. In addition, the utilization is compared with a similar embedded soft processor, the Xilinx MicroBlaze.

Table 3. Hierarchical utilization for the RISC V processor.

Module	**Look up Tables**	**Flip flops**	**RAMB18**
Instruction Fetch	468	176	8

Decode Unit	749	1068	0
Execution Unit	1046	340	0
Memory Access Unit	1164	241	16
Writeback Unit	66	104	0
Hazard Unit	32	0	0
Total	3525	1929	24

The RISC V processor's utilization is dominated by the memory access and execution unit. The large amount of combinational logic within the execution unit to handle forwarding is responsible for that unit's heavy usage of LUTs. The data cache is also responsible for a significant share of LUTs due to the doubling of tag comparison logic needed in a 2-way cache.

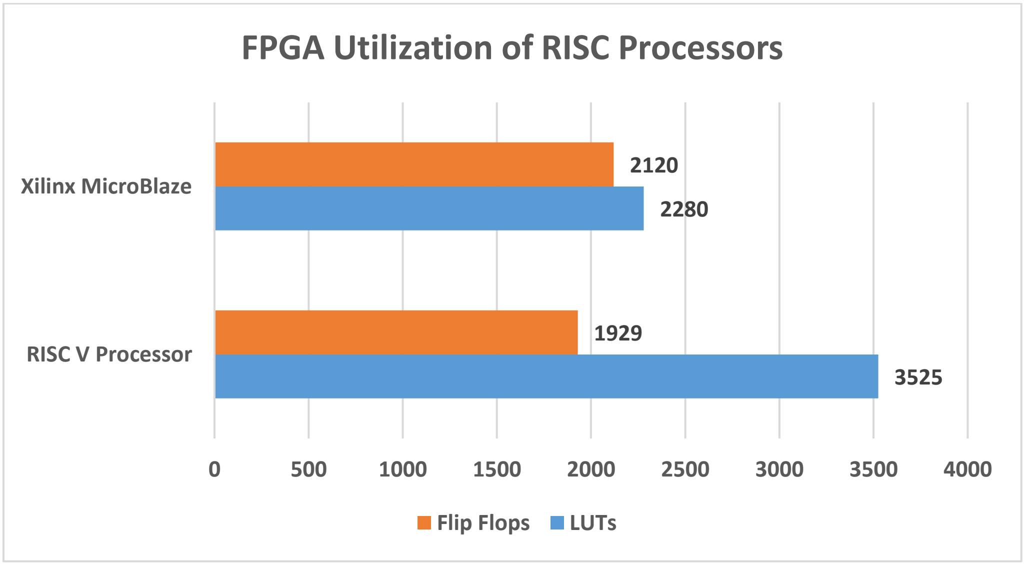


Figure 19. Comparison of FPGA utilization between the Xilinx MicroBlaze and the processor under test.

Table 4. Comparison of high utilization modules in the MicroBlaze and processor under test.

Module	RISC V Processor	Xilinx MicroBlaze
Execution Unit	1046	257
Memory Access	1164	564

The RISC V processor consumes more LUTs, but less flip flops than the MicroBlaze. Table 4 indicates that the execution and memory access units are responsible for this difference. The Xilinx MicroBlaze does not contain data forwarding, so it's able to save a significant amount of logic within its execution unit at the cost of having to insert more stalls within the pipeline [15]. The MicroBlaze also contains a direct-mapped cache, unlike the RISC V processor built which has a 2-way set associative cache [15]. The MicroBlaze is able to save resources by not having additional logic for each cache way, but will exhibit more cache misses over time than the RISC V processor.

5.4 Power

For an embedded system, power is a critical measure of the success of a processor, since it determines the thermal requirements and battery consumption of the core. Power for the design under test is shown hierarchically for the modules and compared to a similar Xilinx MicroBlaze processor.

Table 5. Hierarchical power usage for the RISC V processor.

Module	Power (mW)
Instruction Fetch	21
Decode Unit	6

Execution Unit	3
Memory Access Unit	57
Writeback Unit	1
Hazard Unit	< 1
Total	88

The power usage of the RISC V processor is dominated by the instruction and data cache blocks. This power comes almost entirely from the block RAM within the FPGA. Since the size of the cache within the processor is entirely configurable, the amount of power can be scaled up or down depending on the requirements of the design.

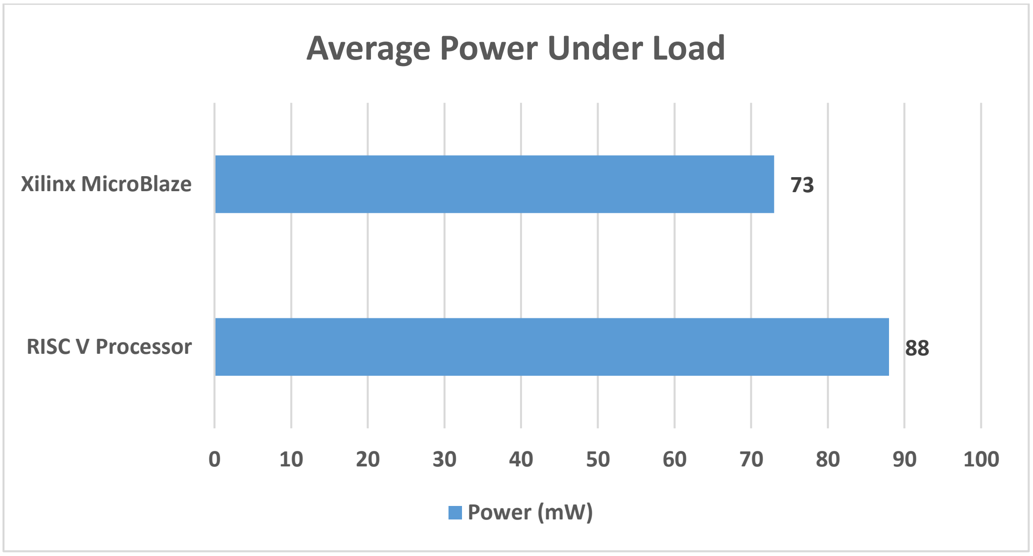


Figure 20. Comparison of average power under load for RISC processors.

Table 6. Comparison of high power modules in the MicroBlaze and processor under test.

Module	**RISC V Power (mW)**	**MicroBlaze Power (mW)**
Instruction Fetch	19	14

Memory Access	57	32

The Xilinx MicroBlaze processor consumes less power than the RISC V processor. The MicroBlaze and RISC V processor diverge in power statistics when it comes to the data cache. Similar to the utilization, this is because the MicroBlaze only has a 1-way cache, while the RISC V has multiple ways to concurrently perform calculations on. The 2-way associativity of the RISC V processor reduces the number of misses, but also increases the power when compared to the direct mapped cache.

5.5 Performance

Although the RISC V processor implemented is intended to be used in an embedded platform optimized for power efficiency and low utilization, performance was also measured for the software tested in order to find an estimate of processing capability.

Table 7. Median filter (15k Samples) benchmark performance

Core	**Frequency**	**Execution Time (ms)**
Intel i7-4710 HQ (running medfilt1)	2.5 GHz	1.078129
RISC V Processor	75 MHz	5.417253
Xilinx MicroBlaze	100 MHz	14.16378

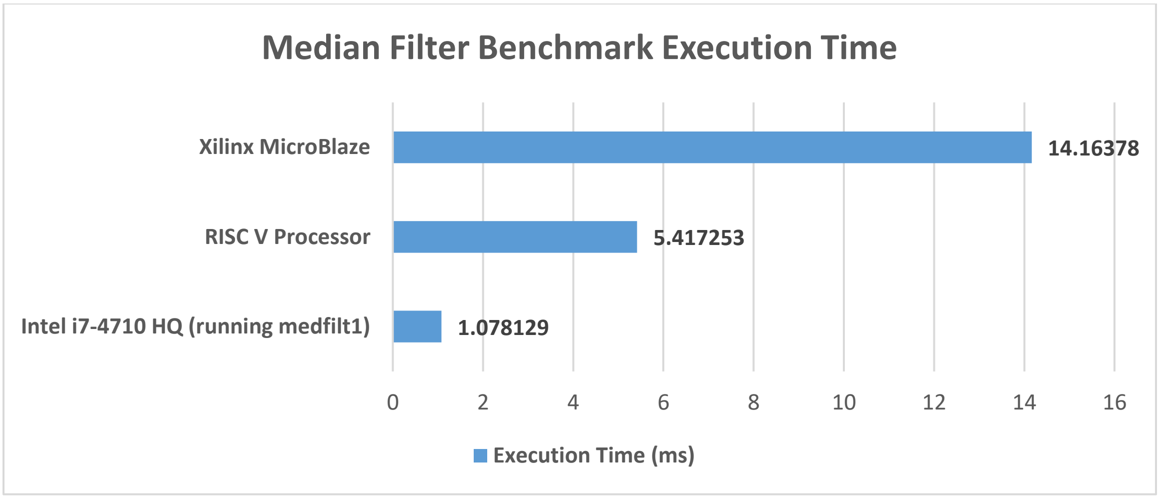


Figure 21. Median filter (15k Samples) benchmark execution time.

The RISC V processor outperformed the Xilinx MicroBlaze significantly in the median filter benchmarking test. Both the processor architecture and ISA of the MicroBlaze were responsible for limiting its performance during the benchmark. In order to perform a conditional branch on the MicroBlaze ISA, both a comparison and branch instruction are needed. In RISC V, a comparison and branch can be performed within the same instruction (Figure 22). Since the median filter test consisted of a series of nested branches to determine the median for each sample, this increased the number of instructions and number of cycles for the MicroBlaze processor.

RISC V Comparison & Branch	MicroBlaze Comparison & Branch
# if (r20 >= r21) blt r21, r20, addr	# if (r20 >= r21) cmp r25, r21, r20 bgei r25, addr

Figure 22. Comparison of conditional branch instructions in RISC V & MicroBlaze ISAs.

In addition, the processor architecture of the MicroBlaze does not allow for forwarding of the register values within the pipeline. Since the same three registers (median filter, n = 3) were utilized for comparison and assignment in this test, the MicroBlaze was forced to insert stalls whereas the RISC V processor could forward the latest register status.

To determine the success of the RISC V processor for embedded systems, the performance and total power utilization were combined to determine the performance per Watt (Figure 23). Although the RISC V processor consumed more power than the Xilinx MicroBlaze, its performance during the median filter test places it well ahead of the MicroBlaze in terms of efficiency. The emphasis that embedded systems have on efficiency makes the RISC V processor implemented a strong contender as the processor of choice for embedded platforms when compared to the similar Xilinx MicroBlaze.

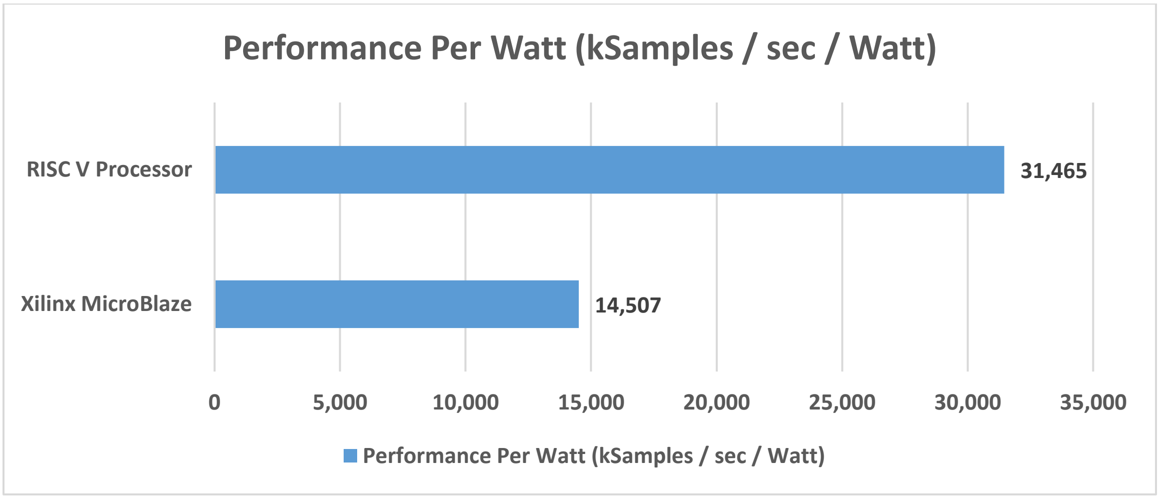


Figure 23. Performance per Watt of the RISC processors (median filter benchmark).

CHAPTER 6: FUTURE WORK

The processor currently contains configurable features, such as the ability to configure the cache size or associativity, however, the processor can be improved by expanding on this configurability. It may be desired in the future to be able to do multithreaded workloads within an embedded environment. Multicore functionality can be added to alleviate these workloads by modifying the processor cache implementation to support the AXI Coherency Extension (ACE). This would allow cache coherency and the ability to place multiple cores within a single FPGA. This multicore enabled design would allow configurability for any number of cores depending on the platform requirements.

In addition to allowing for a configurable number of processing cores, it can be useful to support instruction set extensions. Since RISC V allows for flexibility by adding extensions to the ISA, the hardware architecture can be made configurable to turn on certain extensions. In particular, the floating point extension RV32F and the multiply/divide extension RV32M may be useful for certain algorithms. To implement these extensions, the execute unit would need to be expanded to have additional computational units for floating point and multiplies/divides respectively. Like the multicore feature, it would be beneficial for these ISA extensions to be left as a configurable parameter so the processor could be tailored to a particular environment or algorithm.

CHAPTER 7: CONCLUSION

The RISC V processor implemented demonstrated efficient processing while consuming minimal resources on the device. Utilizing the RISC V RV32I instruction set allows for a small amount of instructions and complexity while still outperforming competitors within the embedded space. The processor architecture features performance enhancing components such as n-way cache associativity and register forwarding while keeping within a constrained size. Power consumed by the processor remained under 100mW while tasked, allowing the processor to be used in power constrained platforms and environments. In addition, the RISC V processor exhibited considerable processing capability by running the median filter signal processing benchmark in 40% of the time as the Xilinx MicroBlaze embedded processor. The efficiency and performance of the core implemented makes it a compelling solution for embedded processing platforms.

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, “A new golden age for computer architecture,” *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, Feb. 2019.

[2] A. Waterman and K. Asanovic, *The RISC-V Instruction Set Manual*. (2019). Accessed: 26-Nov-2019. [Online]. Available: https://riscv.org/specifications/

[3] S. Leibson, K. Stevens, and TotallyLost, “RISC-V Business,” *EEJournal*, 06-Mar-2019. [Online]. Available: https://www.eejournal.com/article/risc-v-business/. [Accessed: 27-Nov-2019].

[4] C. Domas, *Breaking the x86 ISA*. (2017). Accessed: 26-Nov-2019. [Online]. Available: https://www.blackhat.com/docs/us-17/thursday/us-17-Domas-Breaking-The-x86-Instruction-Set-wp.pdf

[5] *RISC V*, 17-Oct-2019. [Online]. Available: http://www.riscv.org/. [Accessed: 27-Nov-2019].

[6] “RISC-V Core IP - SiFive,” *https://www.sifive.com/share.png*. [Online]. Available: https://www.sifive.com/risc-v-core-ip. [Accessed: 27-Nov-2019].

[7] ARM. *AMBA® AXI™ and ACE™ Protocol Specification*. (2019). Accessed: 26-Nov-2019. [Online]. Available: https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf

[8] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson, and K. Asanovic, 26-Sep-2017. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.pdf. [Accessed: 26-Nov-2019].

[9] “Berkeley Out-of-Order Machine,” *Berkeley Architecture Research*. [Online]. Available: https://bar.eecs.berkeley.edu/projects/boom.html [Accessed: 27-Nov-2019].

[10] C. Celio, D. A. Patterson, and K. Asanovic, 26-Sep-2017. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.pdf [Accessed: 26-Nov-2019].

[11] “Core Designer,” *SiFive*. [Online]. Available: https://scs.sifive.com/core-designer/. [Accessed: 27-Nov-2019].

[12] “VexRiscv,” *GitHub*, 09-Nov-2019. [Online]. Available: https://github.com/SpinalHDL/VexRiscv#vexriscv-architecture. [Accessed: 27-Nov-2019].

[13] “Resources: Accessing memory-mapped peripherals,” *ARM Developer*. [Online]. Available: https://developer.arm.com/tools-and-software/embedded/legacy-tools/ds-5-development-studio/resources/tutorials/accessing-memory-mapped-peripherals. [Accessed: 27-Nov-2019].

[14] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface,* 5th ed. Brantford, Ontario: W. Ross MacDonald School Resource Services Library, 2017.

[15] Xilinx. *MicroBlaze Processor Reference Guide*, 2017.3. (2017). Accessed: 29-Nov-2019. [Online] Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug984-vivado-microblaze-ref.pdf

APPENDIX A: Code Repository

The code repository, containing both the RTL and tests, can be found at https://github.com/Reconfigurable-Computing-CalPoly-Pomona/Reconfigurable-RISC-V.

APPENDIX B: Supported Instructions (RV32I)

Name	Format
Loads	
Load Byte	LB rd, rs1, imm
Load Halfword	LH rd, rs1, imm
Load Word	LW rd, rs1, imm
Load Byte Unsigned	LBU rd, rs1, imm
Load Half Unsigned	LHU rd, rs1, imm
Stores	
Store Byte	SB rs1, rs2, imm
Store Halfword	SH rs1, rs2, imm
Store Word	SW rs1, rs2, imm
Shifts	
Shift Left	SLL rd, rs1, rs2
Shift Left Immediate	SLLI rd, rs1, shamt
Shift Right	SRL rd, rs1, rs2
Shift Right Immediate	SRLI rd, rs1, shamt
Shift Right Arithmetic	SRA rd, rs1, rs2
Shift Right Arithmetic Immediate	SRAI rd, rs1, shamt
Arithmetic	
Add	ADD rd, rs1, rs2
Add Immediate	ADDI rd, rs1, imm
Subtract	SUB rd, rs1, rs2
Load Upper Immediate	LUI rd, imm
Add Upper Immediate to PC	AUIPC rd, imm
Logical	
XOR	XOR rd, rs1, rs2
XOR Immediate	XORI rd, rs1, imm
OR	OR rd, rs1, rs2
OR Immediate	ORI rd, rs1, imm
AND	AND rd, rs1, rs2
AND Immediate	ANDI rd, rs1, imm
Compare	
Set Less than	SLT rd, rs1, rs2
Set Less than Immediate	SLTI rd, rs1, imm
Set Less than Unsigned	SLTU rd, rs1, rs2
Set Less than Immediate Unsigned	SLTIU rd, rs1, imm
Branches	
Branch Equal	BEQ rs1, rs2, imm
Branch Not Equal	BNE rs1, rs2, imm
Branch Less than	BLT rs1, rs2, imm
Branch Greater than or Equal to	BGE rs1, rs2, imm
Branch Less than Unsigned	BLTU rs1, rs2, imm
Branch Greater than or Equal to Unsigned	BGEU rs1, rs2, imm

Jump & Link	
Jump and Link	JAL rd, imm
Jump and Link Register	JALR rd, rs1, imm
Synch	
Synch thread	FENCE
Synch Instruction and Data	FENCE.I
System	
System Call	SCALL
System Break	SBREAK
Counters	
Read Cycle	RDCYCLE rd
Read Cycle Upper Half	RDCYCLEH rd
Read Time	RDTIME rd
Read Time Upper Half	RDTIMEH rd
Read Retired Instructions	RDINSTRET rd
Read Retired Instructions Upper Half	RDINSTRETH rd