5-2018

A Proposed Approach to Hybrid Software-Hardware Application Design for Enhanced Application Performance

Alex Shipman
University of Arkansas, Fayetteville

A Proposed Approach to Hybrid Software-Hardware Application Design for Enhanced Application Performance

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering

by

Alex Shipman
University of Arkansas
Bachelor of Science in Computer Engineering, 2008

May 2018
University of Arkansas

This thesis is approved for recommendation to the Graduate Council.

____
J. Patrick Parkerson, Ph. D.
Thesis Director

____
Gordon Beavers, Ph. D.
Committee Member

____
Jia Di, Ph. D.
Committee Member

Abstract

One important aspect of many commercial computer systems is their performance; therefore, system designers seek to improve the performance next-generation systems with respect to previous generations. This could mean improved computational performance, reduced power consumption leading to better battery life in mobile devices, smaller form factors, or improvements in many areas. In terms of increased system speed and computation performance, processor manufacturers have been able to increase the clock frequency of processors up to a point, but now it is more common to seek performance gains through increased parallelism (such as a processor having more processor cores on a single chip or an increased number of physical processors in the system) rather than increased processor frequency.

This paper proposes an additional strategy for increasing system performance by allowing application developers to design custom circuitry for either their whole application or at least some portions of it that are on the critical path or that are computationally expensive. This will improve application performance by avoiding executing application instructions on a general-purpose CPU and instead executing on a specialized circuit designed to perform the desired algorithm, offloading tasks from the CPU (thereby reducing the overall system load) and enabling true parallelism for applications that can take advantage of a more parallel architecture.

For this paper, an Altera Cyclone II FPGA is used, but in practice, a higher performance and higher density FPGA should be used.

Acknowledgements

I thank my advisor, Dr. Parkerson, for his teaching and encouragement during this process. Additionally, I thank Dr. Beavers and Dr. Di for taking the time to be on my committee. I would also like to thank all of the professors who have taught me so many things.

Dedication

I would like to dedicate this paper to my parents who have supported and encouraged me in so many ways and without whom I would not be writing this paper.

Table of Contents

Table of Figures

1 Introduction

Field programmable gate arrays (FPGAs) have been used in a wide variety of applications and have many uses, such as teaching digital logic design, prototyping designs to verify their correctness and practicality, along with a great many other uses. This paper proposes an additional use of FPGAs to improve overall system performance by allowing application developers to implement parts of their application, or potentially the whole application, on the FPGA and to call it just like one would call a normal function in a programming language.

The performance of a system or an application could mean many things, such as the rate at which it performs a task, the amount of power it uses, the amount of memory it uses, along with other possible metrics. For the purpose of this paper, however, the concepts of performance and optimization will strictly refer to the speed of an application or a system, unless otherwise stated. But this paper is focused on optimizing applications, so they will be able to take less time to perform some task than would otherwise have been required.

Application designers have a wide variety of ways to optimize an application. Some of the more typical options include, but are not limited to, choosing more efficient data structures and algorithms, improving database performance for database-driven applications, reducing network latency or reducing the number of network calls for applications that call web services or are otherwise network dependent, choosing a compiled language over an interpreted language when possible, enabling compiler optimizations, multi-threading data access and computations that are parallelizable, implementing a caching strategy rather than making the same database or service calls repeatedly, along with many other ways to improve application performance depending on the specifics and architecture of the application.

Furthermore, some languages, such as C and C++, allow the developer the ability to write directly in assembly language as a way of conceivably optimizing critical portions of applications, although modern compilers employ sophisticated optimization techniques and are oftentimes very good and they are able to take advantage of optimizations like instruction reordering and replacing instructions with machine-specific instructions, along with a large number of other optimization techniques that might be difficult to perform by hand. But the idea is that a person who knows the architecture of the system for which they are writing the application, and who is good at writing assembly language, will be able to increase application performance by hand-optimizing portions of the application to take advantage of system-specific characteristics by specifying exactly which instructions will be executed. The idea proposed in this paper is similar to this idea except rather than writing a program that runs on a general-purpose CPU, the developer will be able to bypass the CPU altogether and directly implement a circuit to perform the necessary algorithm, thereby improving the performance of the application by not having to go through all of the steps that a general-purpose CPU has to go through to lookup an instruction and execute it. The custom circuit, since it is designed to solve a single problem, is able to be optimized more than a general-purpose circuit.

When commencing a design, a software engineer has the choice to develop software for a general-purpose central processing unit (CPU) using languages such as C++ and Java, or to target a general-purpose graphics processing unit (GPGPU) using frameworks and libraries, such as OpenCL or CUDA, which are designed to target the massively parallel architecture of GPUs, or to implement the design in hardware. This paper proposes yet another choice: the hybrid hardware-software application.

1.1 General-Purpose Processors

Most applications will be written using high-level programming languages such as C, C++, Java, or a wide assortment of other languages depending on the requirements of the application, the familiarity of the developers with the language, the portability requirements, along with other factors. Some of the main reasons of doing this are because of ease of development, reduced development and support costs, the simplicity of distributing the software, portability reasons, and because it is fast enough for most applications. These applications will consist of, at the most basic level, instructions that are executed by a processor. For interpreted languages, such as Python or scripting languages, such as shell scripts or JavaScript, the interpreter will parse the statements in the program and then execute those on the processor, but the interpreter itself, or some other program executing the interpreter, will be executed natively on the processor. For the processor implemented in this paper, it is based on the MIPS (Microprocessor without Interlocked Pipeline Stages) instruction set architecture (ISA) developed by MIPS Technologies. This has a datapath with five different stages that each instruction goes through (shown in Figure 1). This leads to some inefficiency.

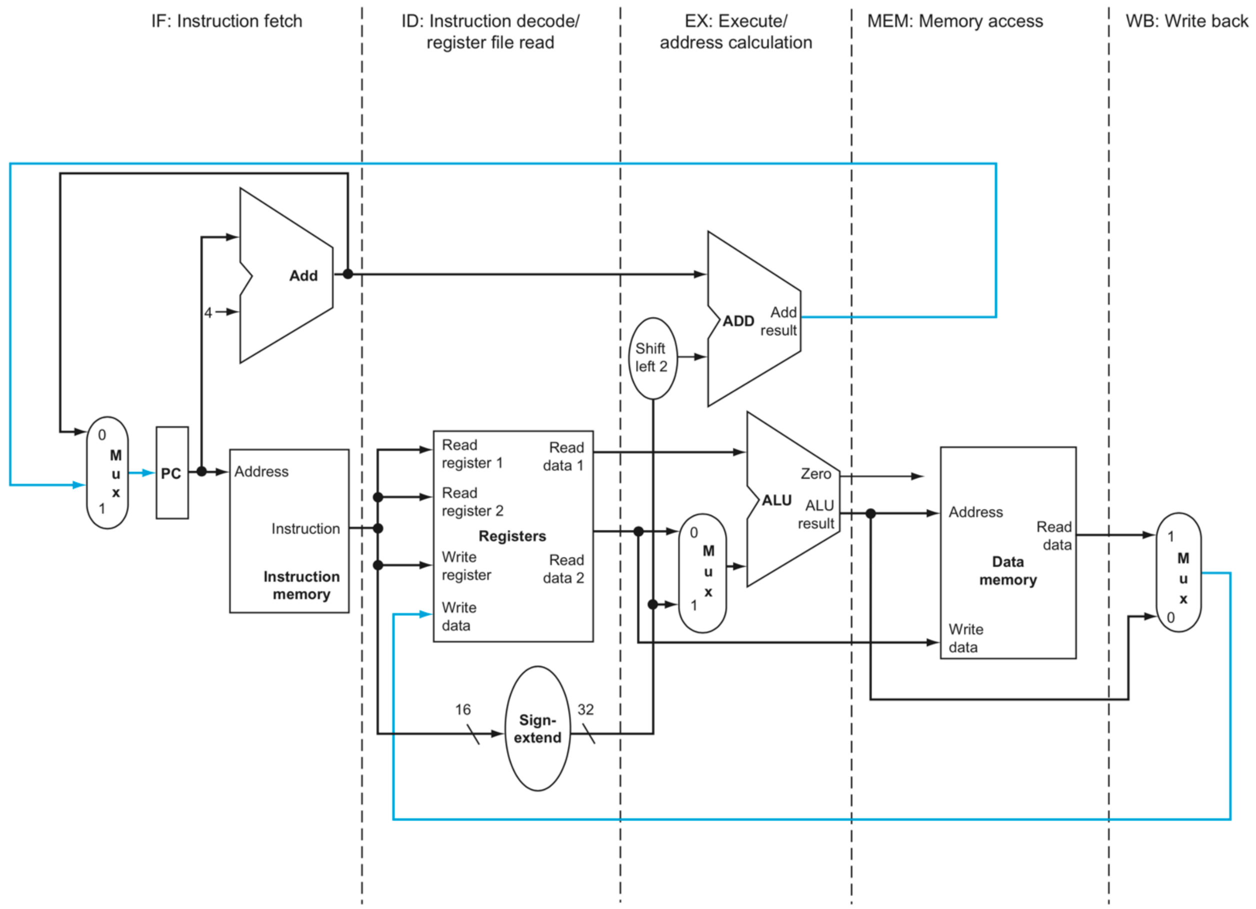


Figure 1 - A Single Cycle Datapath (Patterson and Hennessy 287)

The previous figure shows the datapath that is used for this paper. It has five stages, which are shown. The first stage is the "instruction fetch" stage. This is where the instruction is loaded from memory or disk and loaded into the instruction register. Next is the "instruction decode" step. This is where the instruction, which was previously loaded, is decoded so the processor knows what to do with it. The instruction will have different parts, such as the opcode, as defined in the instruction set architecture, and these different parts will be examined in the control unit to figure out what needs to be done with the instruction (such as whether to add two operands or to load a value from memory). The third is the actual execution phase. This is where, for some types of instructions, the instruction is actually performed (such as for ALU-based instructions like add or shift instructions). Next is the memory access step. This is where, as the name implies, memory values are loaded or written for instructions that access memory values. Finally, the “write back” step is where any necessary values are written to the register file, if necessary.

To see an example of an instruction going through this datapath, consider the add instruction. The instruction `add $1, $2, $3` gets executed and takes five clock cycles to get through the datapath. It takes one clock cycle to load the instruction from disk or memory, which may be an expensive operation as disk IO rates are usually fairly low relative to the frequency of the clock used to drive the processor. Next, the instruction is decoded according to the instruction set architecture of the processor and the register file is read. Third, the instruction is executed. In this case, it uses the adder circuit in the arithmetic logic unit (ALU) to add the contents of registers two and three. Fourth, nothing happens as the next step is the memory access step, but memory is not being accessed for this instruction. Finally, the result of the computation is written to register three in the write-back step.

So, in this architecture, the add instruction takes five clock cycles to execute. The overall throughput of the system may be improved by using a pipelined datapath, but the individual instruction latency will still take five clock cycles for this add instruction.

If the only goal of this application was to add two numbers and return the sum, then a simple full adder could be built just to take the two addends and return their sum. This full adder would be the same as the full adder that is likely implemented in the ALU in the MIPS processor, but it would not need to fetch any instructions or decode the instruction or do the memory access or write back steps. It would simply be able to execute the add instruction in one clock cycle, the equivalent to the execute step of the MIPS datapath. Therefore, the latency of the instruction would go from five clock cycles to just one clock cycle. And some modern processor architectures use pipelines that are much deeper pipelines - some go up to 17 pipeline stages, so even more of a reduction would be seen in such an architecture. Moreover, pipeline hazards, such as read after write data hazards or branch prediction errors, can lead to pipeline stalls and an even less efficient utilization of the hardware.

As another example, if the goal of the instruction is to only perform a logical left shift of the input by n-bits (such as the `sll` MIPS assembly instruction), the instruction still has to go through all the different stages in the datapath, such as trying to figure out what the instruction even is, before it will be able to perform the shift. This is true even if the processor has a barrel shifter implemented to perform the shifts in one clock cycle. It still must take the requisite number of clock cycles to get through the datapath.

But these examples are overly simple and not that useful for most applications. Most applications will likely use the add and shift instructions, but the goal of the application will be much more complicated than only that. What would really be helpful for improving the performance of an application would be to remove the overhead of the datapath of the general-purpose processor and to implement the required functionality directly as a circuit that could be called, or invoked, like a function. This could be portions of an application, which are computationally expensive, or even the whole application for some cases. To get the best performance gains, more of the application should be written as hardware. At least, more contiguous chunks of it, such as an entire function. For example, if just an add statement or a shift statement is implemented as a circuit, then there would not be any gains over the instruction already implemented in the processor, and likely the performance would suffer because of the additional overhead involved with the communication between the processor and the circuit. More about this is discussed in chapter three.

1.2 General-Purpose Graphics Processing Units

Some applications, particularly those of a scientific or a mathematically-intensive nature and, perhaps unsurprisingly, graphically-intense applications, such as games, photo editing software, 3D animation software, along with other similar types of applications, may be able to take advantage of the massively parallel architecture of modern graphics processing units. High-end modern graphics processing units tend to have thousands of cores, each of which is usually very good at carrying out floating point arithmetic. Operations, such as matrix multiplication, matrix addition, and other mathematical operations that are parallelizable and of large size, are able to take advantage of the massively parallel architecture of GPUs. But for traditional non-scientific and single threaded applications, or applications of small size, the GPU may not help much or might be counterproductive for some problems. Even though the GPU has potentially hundreds or thousands of times the number of cores that are in the CPU, each core is a much simpler core than a core in the CPU and the GPU cores are not necessarily meant to execute individual instructions very quickly, but rather they focus on speed gains by executing a large number of small chunks of the problem simultaneously. Furthermore, GPUs can add extra expense to the overall system and can use a substantial amount of power. Additionally, development and debugging of such applications is typically much more difficult and is best suited for engineers that are trained and experienced in GPU programming, leading to higher maintenance costs for the companies who own the applications.

1.3 Full Custom Circuits

For some applications that have critical performance requirements, application designers may opt to design a full-custom application-specific integrated circuit (ASIC) to realize the application logic. While this should give the absolute best performance, it is extremely cost prohibitive as the upfront manufacturing costs can be in the millions of dollars and, once the circuit is developed, it is more difficult to distribute than a software-only application and it is not possible to provide updates without replacing the IC. As a result of these negatives, even though the performance would be much better than the same thing implemented on a general-purpose processor, the majority of companies and application developers will not choose this option unless it is absolutely necessary.

1.4 Hybrid Applications

This paper aims to bridge the gap between the ability to configure higher-level programs developed for general-purpose processors and the speed and efficiency of a direct hardware implementation by offering the ability for application developers to implement different portions, or functions, of their application directly in hardware, using an FPGA, while still being able to write the remainder of their application in normal programming languages. FPGAs are not as fast as a custom ASIC, but they are reconfigurable. The ability to reconfigure the device is important as this is meant to be deployed as part of a traditional software application, which will likely receive updates, and the hardware portion of the application may receive updates too. Furthermore, there will only be a certain amount of FPGA space for the entire system, so the operating system has to be able to load and unload functions dynamically to make room for currently running applications. To support a high number of concurrent program design blocks, modern FPGAs are dense in that they provide a high number of logic elements in a small footprint due to technologies such as 3D FPGAs, where layers of FPGAs are stacked on top of each other in the same package. So, by using an FPGA with a large number of logic elements, the system may provide more program design blocks to be used by applications.

Running the application logic on an FPGA would provide the benefit of being able to optimize portions of an application so they do not need to execute instructions anymore that need to go through all of the steps of the datapath, as was described earlier in this paper, but rather are able to invoke a specially designed circuit that realizes the necessary logic for the function. Furthermore, the application developer is no longer restricted to only the instructions that are in the instruction set of the processor but are rather able to directly implement whatever logic is required for their situation and, as a result, are able to execute more than one processor instruction per clock cycle.

In addition to the aforementioned benefits, a hardware implementation is able to take advantage of true parallelism, only limited by the size of the FPGA. For example, if an application needs to add more than one number at a time, then the hardware implementation could contain more than one full adder. Or if it needs to be able to do something more complicated than that in parallel, then likewise the circuitry of the required logic could be duplicated as required. A traditional application is limited in the parallelism it is capable of by the number of actual physical cores in the CPU (or, for GPGPU applications, then the number of cores in the GPU). It is possible for an application to be using more threads than the number of cores in the system, but if this happens, then the operating system will employ time slicing, or some other technique, to divvy up the time between all of the threads. This is fine for some situations where a thread may be stalled waiting for something to happen, but it does still limit the actual true parallelism of a system based on the number of physical cores available.

One downside to implementing the application logic directly in hardware, however, is that it may be more difficult to maintain and support. This is similar to the languages that allow inline assembly language to be used as an optimization technique. To support those applications, developers need to understand assembly language as well as the primary language the application is written in (likely C or C++). In a similar fashion, implementing part of the application in hardware could make it more difficult to support the application because then the developers would need to understand digital logic design. And while it is likely true that most software engineers would not have the necessary skillset to implement arbitrary logic as a circuit, or to be able to debug those systems, those parts of the application could be developed and supported by a team of hardware engineers while the rest of the application is developed and supported by software engineers.

To achieve this goal of allowing arbitrary functions to be embedded in hardware directly, an FPGA will be added, alongside the CPU, and the two will be connected in a way such that special instructions in the CPU's instruction set will be able to issue different commands to the FPGA, such as being able to pass parameters to a function in the FPGA and to be able to enable and start a function in the FPGA. For the purpose of this paper, each function implemented in the FPGA will be referred to as a program design block. Refer to Figure 2 for a high-level depiction of what this looks like. This simplifies the actual implementation because it does not show any controller or how exactly the CPU is able to issue instructions to each program design block. The design and implementation details of this are provided in chapter 2 and chapter 3, but for now, it is just shown that the CPU is able to communicate with any of the program design blocks.

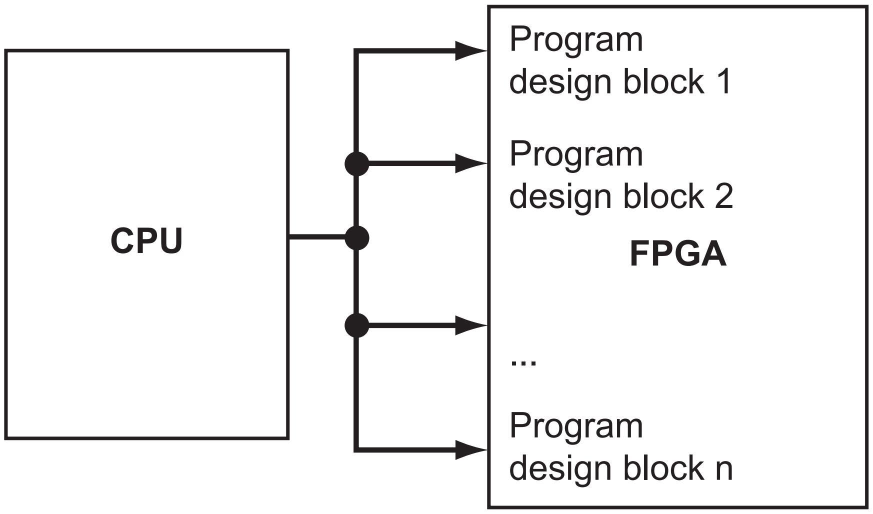


Figure 2 – High-level conceptual CPU to FPGA diagram

A given application may use one, or more program design blocks if necessary, and these program design blocks need to be able to be dynamically reprogrammed, as is deemed necessary by the operating system, to make room for newly loaded applications.

1.5 Paper Structure

This chapter provided an overview of what the goal of this thesis is. That is, to allow application developers the ability to embed certain functions of their application directly in hardware. Chapter two further explains the idea by stating specifically what it is that the finished version of this project should be able to do (the requirements) and the high-level design of how this will be achieved. After this, chapter three discusses the actual implementation, with schematics and other details that show how the project was actually realized. Chapter four provides a few sample programs that were tested, along with the results of running those on just the CPU as a software-only application and the results of writing them as hybrid applications and converting portions of the application to hardware and running that on the FPGA. Chapter five describes what else could have been done to make the implementation in this project better or what would be necessary for this to be turned into a more production-ready implementation and possibly what could be some of the difficulties or issues faced trying to implement this in a production system. Chapter six is the appendix, where a full listing of supported instructions and the HDL and schematics that make up the processor used in this project may be found.

2 Requirements and Design

To support hybrid applications as proposed in this paper, some modifications will be required to the processor to support communicating with the program design blocks on the FPGA. This includes additional instructions in the instruction set and a connection between the processor and the FPGA. This chapter presents the requirements of this project along with the corresponding design for each requirement.

2.1 New Instructions

The instruction set and the instruction format used in the processor in this paper are based on the MIPS instruction set presented in (Patterson and Hennessy), which is shown in section 7.2. Additionally, a complete list of the implemented instructions may be found in the appendix in section 7.1.

The functional requirements around the new instructions are:

1. There needs to be a way to send arbitrary input (function parameters) to the program design block.
2. There needs to be a way to start the function implemented by the program design block. This will actually begin the processing.
3. There needs to be a way to handle arbitrary output (the function return value) from the program design block.
4. There needs to be a way to stop the program design block once it has been started. This may be used by the program itself or the operating system.
5. There needs to be a way to check if the program design block is done or if it is still processing.

The following set of instructions is introduced to satisfy the previous requirements. All of these instructions use a modified version of the R (register) instruction format presented in (Patterson and Hennessy). The instructions in this paper have the following fields in their machine language representations:

op	rs	rt	000	index	shamt	funct
6 bits	5 bits	5 bits	3 bits	2 bits	5 bits	6 bits

- op – this is the op-code of the instruction. It is used to differentiate this instruction from other instructions. Note, however, that it is possible for more than one instruction to have the same op-code. So, the op-code is not necessarily a one-to-one mapping to an instruction. For example, in the processor implemented in this paper, several instructions use op-code zero. If more than one instruction uses the same op-code, the distinction is then made based on the value in the funct field. For this paper, all of the new instructions will use an op-code of five (000101_2) as this op-code is unused currently. Even though these instructions do not use the rd register, like the other r-type instructions, it would still be possible to reuse the same op-code that other r-type instructions use and just use a different value for the funct field, but a different op-code was selected because it better logically groups the new instructions together and allows for easier expansion to include new fpga_* instructions, if there is a need for that, without having to be concerned about no more possible funct values being available for the selected op-code.
- rs – this is a source register. Depending on the instruction, it can have different meanings.

- rt – this is another source register. Again, just like with rs, depending on the instruction, it can have different meanings.
- index – this is a two-bit literal representing the index of the program design block that this instruction will use. In (Patterson and Hennessy) this, along with the three preceding bits, was the destination register, or rd. This was where the result of the instruction's computation or memory operation was stored. For these new instructions, however, only the two lower order bits will be used and this will tell FPGA unit which program design block is being used. This could have been a register so it would be more like the R-type instructions, but there are only going to be four program design blocks for this paper, which are addressable in two bits instead of requiring the full 32-bits that comes with the register, so rather than trying to read from possibly three registers at once or spreading out the register reads over two clock cycles, the design decision was made to replace this register with the literal two bits required for the index. For future enhancements, there are the three unused bits that could be used as well, supporting up to 32 program design blocks if necessary.
- shamt – the shift amount. This field is not used by any of these instructions and will always be zero. It would be possible to repurpose this field if necessary so it could be used for some instruction-specific purpose or to expand the funct field or the index field if necessary, but for the purpose of this paper, this field will be ignored.
- funct – the function being implemented. This is used to make it possible to reuse the same op-code for more than one instruction. For all of the new r-type

instructions that will be added, they will all use an op-code of five. So, to allow the hardware to understand whether it is looking at an fpga_set or an fpga_call (these instructions will be explained next), the value of the funct field will allow this distinction.

fpga_set index, rs, rt – This satisfies requirement one and it sets a parameter at the index specified by the source register, rs, to the value stored in the other source register, rt, for the program design block specified by the index operand. It is up to the program design block on how to handle parameters and how it stores them. One possible way of handling the storing of parameters is to have a register, or a collection of registers, or possibly even a memory block depending on how many parameters are required for the function.

Here is the machine language representation of the instruction, where rs, rt, and rd will be the five-bit representation of the register number used for those fields.

000101	rs	rt	000	index	00000	000000

To support arbitrary function parameters, such as lists, sets, maps, or other collections along with strings and other data types, it is up to the program design block how it interprets the inputs. For example, if an array of values needs to be passed to the program design block, then there could be two parameters supplied: one for the starting address in memory of the array and the other parameter could indicate the number of elements in the array, similar to how an array in C is passed. But how the program design block interprets and uses the parameter values that are supplied to it by the application are up to the application designer. Whether one parameter is interpreted as a literal value or an address pointing to a location in memory is an application-specific design decision. But with this level of flexibility, it is possible to pass arbitrary arguments, whether simple primitive data types or complicated data structures. Just for more complicated data structures, it might require more than one parameter to be passed to indicate the location, or locations, in memory to look for the actual data.

Example: fpga_set 1, \$2, \$3, where 1 is the program design block index, \$2 is the index of the parameter (as is determined by the program design block controller), and \$3 is the value of the parameter. This will be assembled as 000101 00010 00011 000 01 00000 000000.

fpga_call index – This satisfies requirement two and it invokes the program design block with the index specified by the index operand. This should be called after all of the parameters have been set using the fpga_set instruction. Here is the machine language representation of the instruction, where index will be the two-bit representation of the program design block index. The values of the rs and rt fields are shown as zero, though it is inconsequential what is stored in those fields because they will not be used.

000101	00000	00000	000	index	00000	000001

Example: fpga_call 1, where 1 is the index of the program design block to invoke. This would get assembled as 000101 00000 00000 000 01 00000 000001.

fpga_get index, rs – This satisfies the third requirement and it gets the return value of the program design block specified by index and stores the return value in rs. For this instruction, the program design block should have the result ready to output within the same clock cycle when asked for it. If the computation is not done yet and the program design block is asked for the result, then the results are undefined. Depending on how the program design block is designed, it could be an intermediate result, or it could be some indeterminate value. Before calling this instruction, it is best to call the fpga_status instruction to make sure the program design block is done with its work.

Here is the machine language representation of the instruction, where rs will be the five-bit representation of the register number of where to store the results and index is the two-bit index of the program design block to query. The value of the rt field is shown as zero, though it does not matter what value is stored in that field because it will not be used.

000101	rs	00000	000	index	00000	000010

This instruction will only be able to return a single word. For longer return values, the program design block could write to memory and then return the starting memory address in the register rs for the application to be able to load from memory and process as required. It is up to the program design block and the application to determine how to handle the return value.

Example: fpga_get 1, $2, where $2 holds the return value from the program design block and 1 is the index of the program design block from which to get the return value. This will be assembled as 000101 00010 00000 000 01 00000 000010.

fpga_halt index – This is for the fourth requirement and it tells the program design block at the specified index to stop executing. The program design block could perform cleanup or reset itself so that it is ready for another invocation, though that is not required by this design. This is up to the application requirements and the actual responsibilities of the program design block itself. For example, if the program design block just takes in two numbers and sums them, then there probably is not much state to clear before being ready for subsequent invocations.

Here is the machine language representation of the instruction, where index will be the two-bit representation of the program design block index. The values of the rs and rt fields are shown as zero, though the contents of those fields are inconsequential because they will not be used.

000101	00000	00000	000	index	00000	000011

If this instruction is issued before processing is complete, then this will cause any return value retrieved by fpga_get to be undefined, unless the program design block implements some special logic to be able to return some return value after halting. This may be used to stop the program design block from running, either because the application decides it does not require the result anymore, because the operating system kills the application for whatever reason, or for other reasons. If this is issued after the circuit is done processing, then this has no effect, unless the program design block is implemented such that it listens to halt instructions even when it is not processing, such as for the purpose of resetting the state of the program design block.

Example: fpga_halt 1, where 1 is the index of the program design block to stop. This will be assembled as 000101 00000 00000 000 01 00000 000011.

fpga_status index, rs – This is for the fifth requirement and it tells whether or not the program design block at the index specified by the index operand is done processing and rs is where the status indicator is stored. The program design block needs to be ready to respond to this query in the same clock cycle.

Here is the machine language representation of the instruction, where rs will be the five-bit representation of the register number used for storing the result of the instruction and index is the two-bit index of the program design block. The values of the rt field is shown as zero, though it is irrelevant what is stored in it because it will not be considered for this instruction.

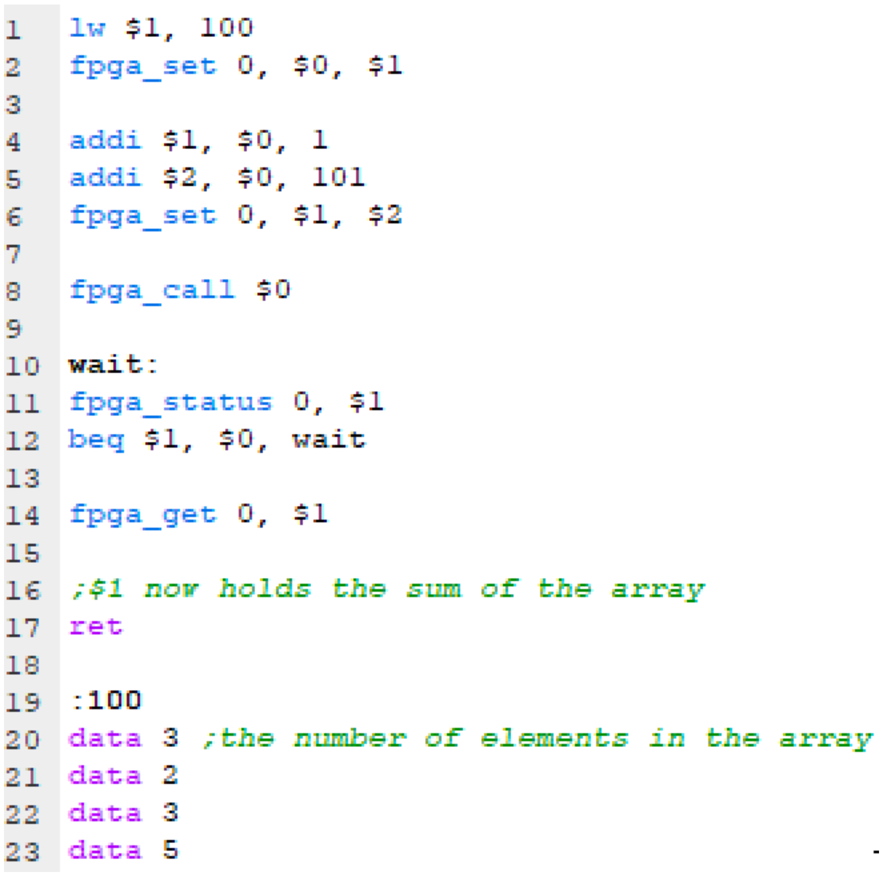
000101	rs	00000	000	index	00000	000100

This instruction may actually be used as more than just a way to determine if the program design block has completed processing or not. This instruction may be used to return any status indicator to the calling program that it needs to. This includes error codes, warning codes, success codes, or whatever else the application designer wants to return. For the purpose of this paper, however, it will always be used just to return the ready indicator, unless otherwise documented. That is, for this paper, this instruction will always return a zero when it is not ready yet (the program design block is still processing) or a one when it is done. Error handling and error codes will not be returned in this paper for simplicity, but this use case is still permitted for more production-ready designs. The values returned and what they mean are defined and interpreted on an application-specific basis, so it is up to the application designer to fully define the return values of this instruction.

Example: fpga_status 1, $2, where 1 is the index of the program design block to query for the status and $2 is the register to store the status. This will be assembled as 000101 00010 00000 000 01 00000 000100.

2.2 Example Usage of New Instructions

To get a better idea of what the instructions do and how they are used, an example program listing is shown here that uses the new instructions. This program listing will just show the code that would be used, but not actually execute it. For this sample, the goal will be to take an array of integers and to sum the values in the array. The program design block is installed at index zero for the purpose of this example, though the program design block is not shown here. Also, the intermediate registers are not stored and restored after this function is over for the purpose of brevity. In a real production method, it would be typical to store and restore the intermediate registers (the registers that are not parameters or return values).

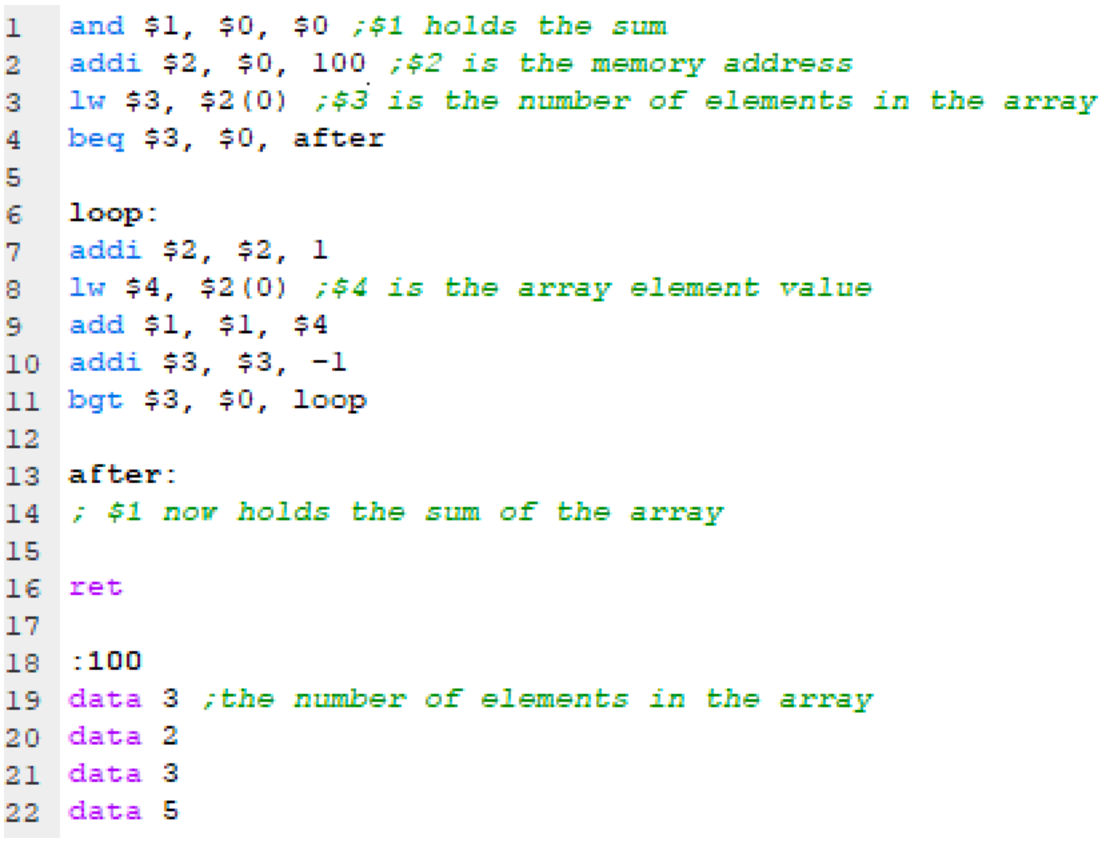


```
lw $1, 100
fpga_set 0, $0, $1

addi $1, $0, 1
addi $2, $0, 101
fpga_set 0, $1, $2

fpga_call $0

wait:
fpga_status 0, $1
beq $1, $0, wait

fpga_get 0, $1

;$1 now holds the sum of the array
ret

:100
data 3 ;the number of elements in the array
data 2
data 3
data 5
```

Figure 3 - Hybrid Program to Sum an Array

The previous code sample illustrates using the new instructions to call the necessary program design block. First, the parameters are set. The program design block takes two parameters. The first parameter is the number of elements in the array and the second parameter is the starting position in memory of the array. So first, parameter zero is set to the length of the array and parameter one is set to the memory address of the start of the array. Then it calls the program design block and then it waits for it to be done. The waiting is just a simple polling mechanism for this program, but depending on the nature of the problem, it may be more efficient to continue processing while the program design block is working and only check back on a sufficient schedule. But for the simplicity of this example and the fact that the whole purpose of this example is to compute the sum of the array and nothing else, a simple polling strategy was selected. Finally, after it is done computing the sum of the array, the sum is then placed in $1.

As a contrast, here is the same program, but not written as a hybrid application, rather it is just a normal application. For this example, there probably would not be much of a performance boost at all to using the hybrid application because the size of the array is so small, but the purpose of this example is just to illustrate how to use the new instructions.



```
and $1, $0, $0 ;$1 holds the sum
addi $2, $0, 100 ;$2 is the memory address
lw $3, $2(0) ;$3 is the number of elements in the array
beq $3, $0, after

loop:
addi $2, $2, 1
lw $4, $2(0) ;$4 is the array element value
add $1, $1, $4
addi $3, $3, -1
bgt $3, $0, loop

after:
; $1 now holds the sum of the array

ret

:100
data 3 ;the number of elements in the array
data 2
data 3
data 5
```

Figure 4 - Traditional Program to Sum an Array

2.3 General Interface

For all of the new instructions described in section 2.1 to be possible, each program design block will need to conform to a general interface so that the processor will know how to issue each instruction across any program design block. For this, it is likely that each program design block will implement a controller circuit that will accept the input from the processor and then determine how to handle that request. Several of these new instructions are very application-specific and do not have much in terms of defined requirements that they must meet, such as the fpga_status instruction, which may return anything as long as the calling program understands it. But even though there is not much structure defined around the return value of those instructions or how they are implemented, each program design block will still need to recognize when it is being asked for the fpga_status as opposed to the fpga_get instruction. The processor will need to know how to communicate with the program design blocks, and as a result, each program design block will have to understand the same set of instructions.

To fulfill this requirement, the processor will pass along the six-bit instruction field to the program design block, which will then need to take the appropriate actions to fulfill the request. A high-level block diagram is shown in Figure 5.

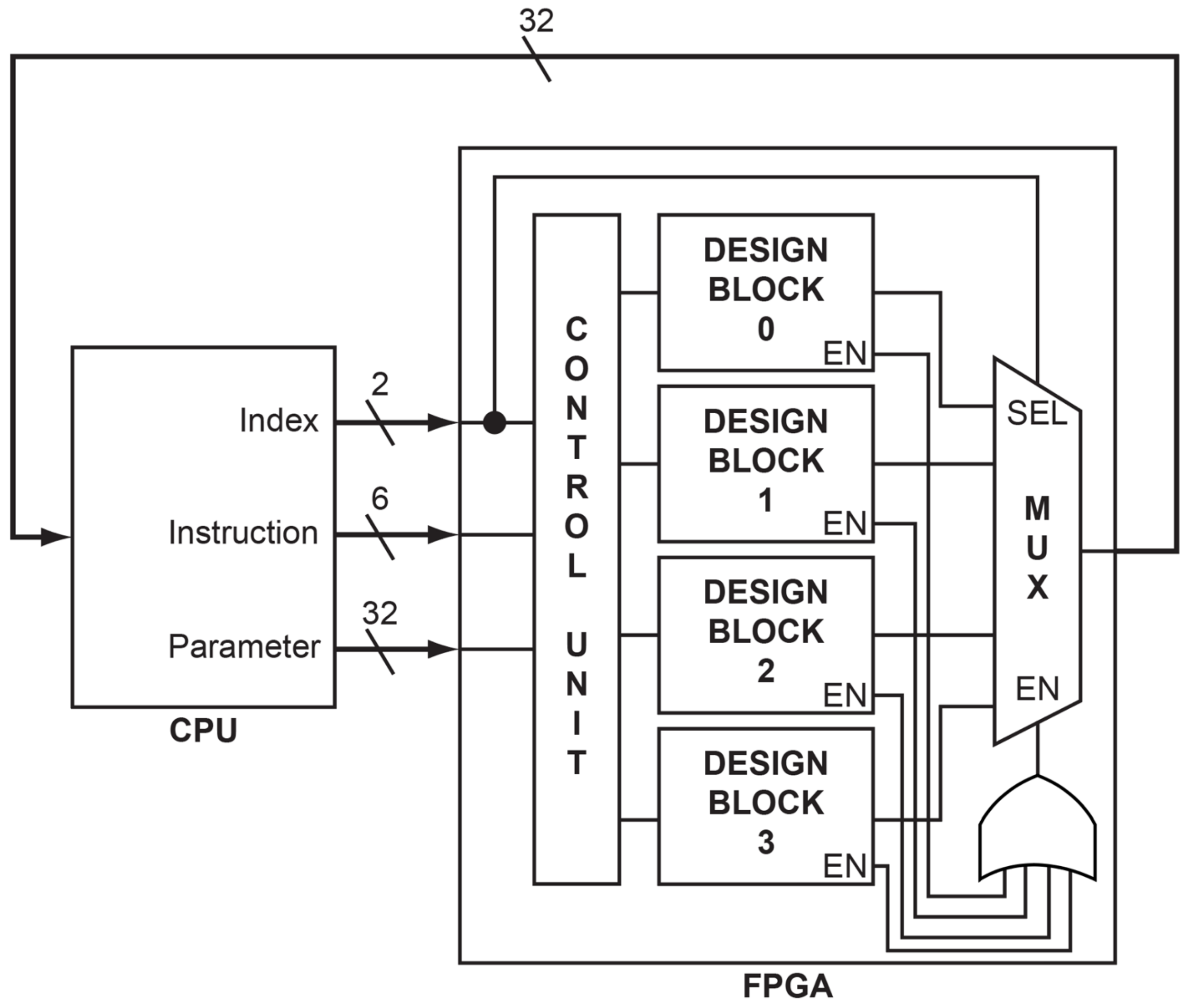


Figure 5 - High-Level FPGA Design

The previous figure shows a high-level diagram of the CPU-FPGA interconnects and the overall FPGA layout. It shows there only being four design blocks. While that is all that is being done for this paper, in an actual production-ready system, it should support more than four. In a production-ready system, the index line should likely be at least four or five bits, depending on the size of the FPGA used. But for simplicity and as a proof-of-concept, only four design blocks will be supported in this paper.

The control unit in the FPGA is used to select which design block to send the signals to. Other than that, it will just pass through the instruction and the parameter inputs. Each program design block outputs a value (for the fpga_get and fpga_status instructions) and it outputs an enable line. The enable line is used to turn on or off the multiplexer which is used to return a value to the CPU. Based on the index input, the multiplexer will output the output from the correct program design block if it is enabled.

Each program design block will need to have a controller that will take in the instruction and the parameter inputs and then take the necessary action based on the inputs it receives. This controller is how each program design block will conform to the interface mentioned in this requirement. Figure 6 shows a very conceptual diagram of what a program design block should consist of.

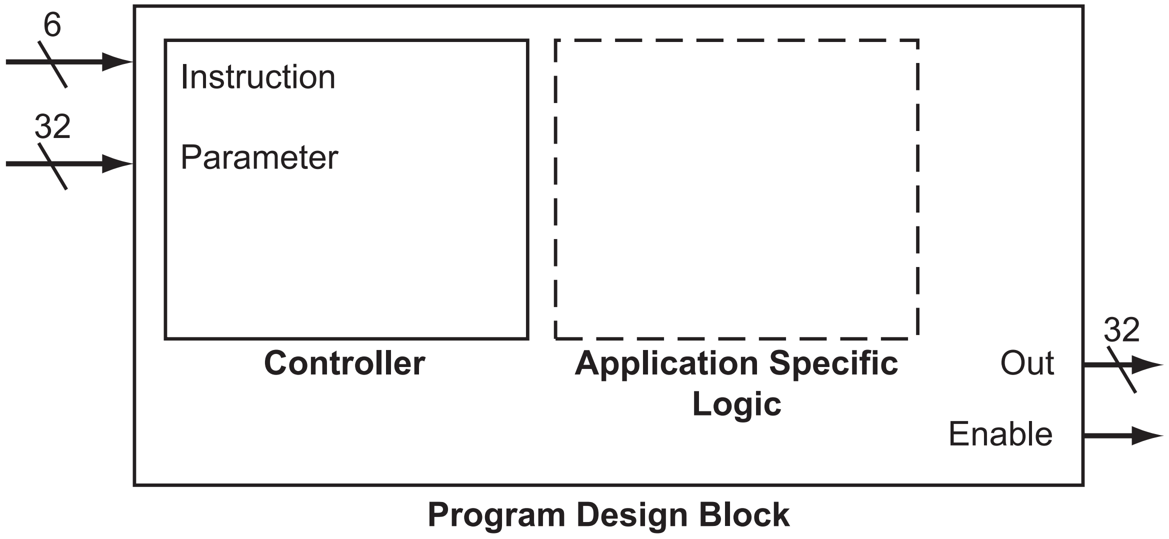


Figure 6 - A Conceptual View of a Program Design Block

The program design block has to take in the instruction input, so it knows what it is being asked to do, and it has to take in the parameter input, even though not all instructions will make use of the parameter input. Actually, only the fpga_set instruction will supply this input. Based on these inputs, the controller circuit will need to cause the application specific logic to do whatever is being asked. Figure 6 does not show any outputs from the controller and there are no inputs or outputs shown from the application specific logic. This is because it is completely up to the application designer how everything works inside the program design block. The only requirement is that it has the specified inputs and outputs and that it takes the requested action when asked to do so. Other than that, how it is actually implemented is not defined in this design.

3 Implementation

This project was implemented and simulated using Altera Quartus II Version 7.2 Build 207 03/18/2008 SJ Web Edition. The circuit is designed to run on the Terasic DE2 board, which has the Altera Cyclone II EP2C35F672C6 FPGA.

To implement the project requirements as described in the previous chapter, a few modifications to the processor described in (Patterson and Hennessy) were necessary, all of which are described in this chapter. This includes the modifications to the control unit to handle the new instructions and the processor make use of the new control lines for the FPGA instructions. Additionally, this chapter shows how the FPGA unit was integrated with the rest of the system, such as the processor. Finally, the FPGA unit is shown along with a sample program design block.

3.1 Control Unit Modifications

The control unit in the processor is a state machine that has been implemented in VHDL. To support the new FPGA instructions, three new states have been added: FPGA, FPGA_READ, and FPGA_DELAY. The FPGA state enables the FPGA unit and then, depending on whether the FPGA instruction is a read instruction or not, it goes to either the FPGA_READ or FPGA_DELAY state, respectively. The FPGA_READ state sets the MemtoReg and RegDst outputs so that the output from the program design block is written to the register specified by bits 25..21 of the instruction (the rs register). The FPGA_DELAY is added so that necessary FPGA register writes have the time to write the value to the necessary register.

```
entity control_unit is
port(
    …
    fpga_enable : out std_logic
);
end control_unit;
```

```
architecture behavior of control_unit is
type state_type is(…, FPGA, FPGA_READ, FPGA_DELAY);

case y is
    …
when B =>
    …
    elsif Op="000101" then --FPGA-related opcode
        y<=FPGA;
    …
    end if;
…
when FPGA =>
    CurrentState<="1101";
    fpga_enable<='1';

    --fpga_get or fpga_status
    if funct="000010" or funct="000100" then
        y<=FPGA_READ;
    else
        y<=FPGA_DELAY;
    end if;
when FPGA_READ => --fpga_get or fpga_status
    CurrentState<="1110";

    --these instructions need to read the
    --value produced by the FPGA
    --and store this value in the register A
    MemtoReg<="100"; --use the output from the FPGA
    RegWrite<='1';

    --this chooses register A (instruction[25..21])
    --as the destination register
    RegDst<="10";
    fpga_enable<='1';
    y<=A;
when FPGA_DELAY =>
    --this just delays the instruction to give
    --the FPGA time to perform the necessary action
    CurrentState<="1101";
    y<=A;
```

Figure 7 Control Unit Modifications

The FPGA_DELAY state could have been removed by increasing the clock frequency of the clock signal for the FPGA unit so that it is approximately at least twice the frequency of the processor's clock signal. This would allow there to be a rising edge of the clock, necessary for the FPGA unit to write to its register, during one clock cycle of the processor's clock. This was not done because, for performance comparison purposes, it was desirable to have the same clock frequency driving the FPGA unit, though in a productionized system, it could supply a higher frequency clock signal to the FPGA unit if there is one available.

Another way to remove the extra delay state is to phase shift the clock signal about half a period or so. This could be done using something like a phase locked loop (PLL) or simply negating the clock signal, which would essentially phase shift the clock signal by half a period, or, equivalently, the falling edge of the clock could have been used to trigger the register writes. These options were not selected because the risk that the half clock cycle time for the write instruction to propagate to the FPGA unit was not worth just losing one extra clock cycle and being guaranteed the write would have time to get to the FPGA unit. Again, in a production system, something like this would probably be more closely evaluated, but for the purposes of this paper, a one extra clock cycle delay was not significant enough to warrant the extra risk.

3.2 Processor Modifications

To support the requirement to have the FPGA read instructions (fpga_status and fpga_get) to be able to read from the FPGA program design blocks and write the result to a register, it is necessary to modify the multiplexor which controls the data that is written to the register file. The extra input, called fpga_in, is added to the memory-to-register multiplexor as input four (100_2). In the previous section, the control unit was modified to set the MemtoReg output to a value of four to select the value that is coming from the FPGA.

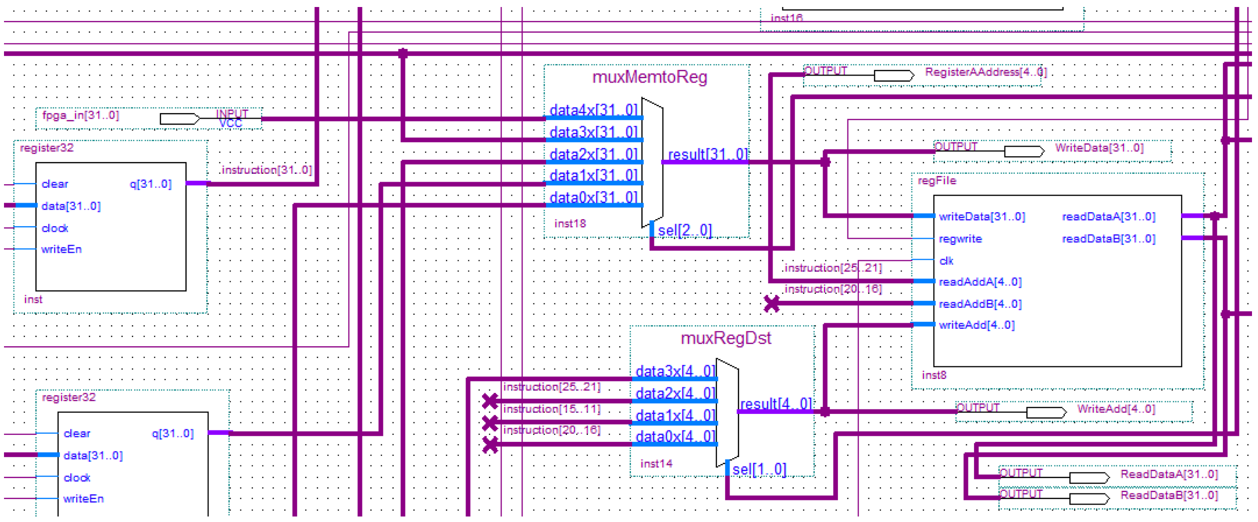


Figure 8 Processor Modifications

The FPGA read instructions, fpga_get and fpga_status, which use this new input, write to the rs register, as was specified in the requirements in the previous chapter. The rs register is specified by instruction[25..21], select line two (10_2) of the register destination multiplexor.

3.3 System-Level Integration

Outside the processor, the FPGA unit needs to be connected and integrated with the rest of the system. To do this, an output signal from the processor, fpga_enable, was added that will enable the FPGA unit to receive new instructions. This is turned on from the control unit of the processor and is only enabled for two clock cycles during an FPGA instruction. Without the enable input being high, the FPGA program design blocks will still be able to perform their work, it is just that no new instructions will be able to be received as long as the fpga_enable line is not high.

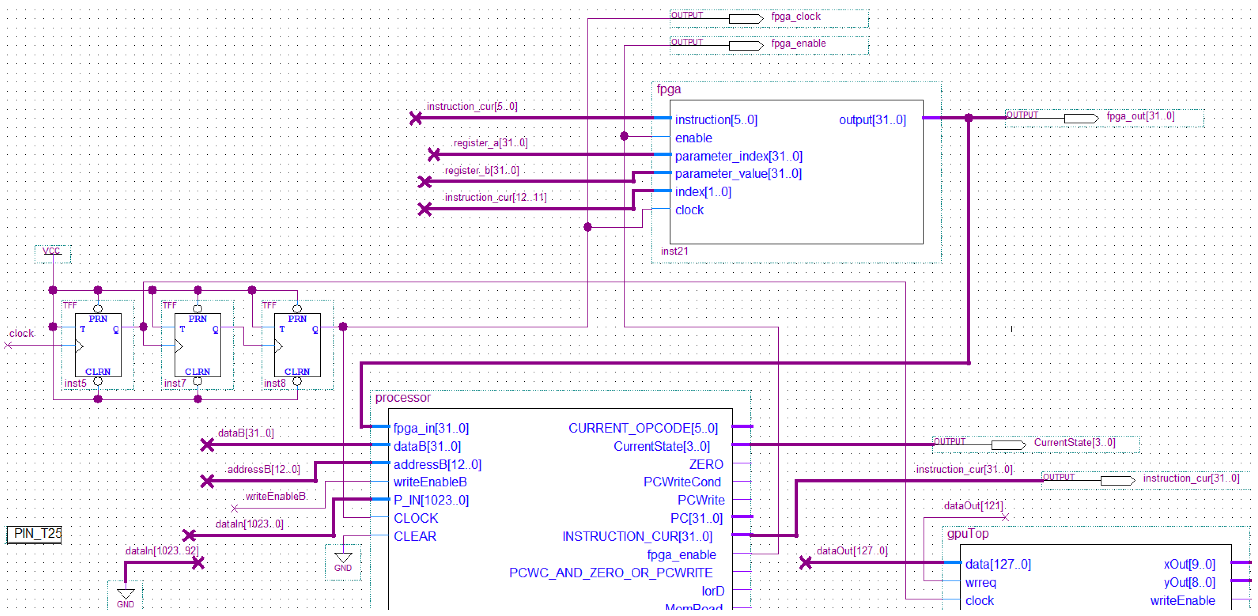


Figure 9 System-level integration

In addition to the output from the processor, there is an input to the processor too that comes from the FPGA unit. This is called the fpga_in bus and it is 32 bits wide to match the size of the registers, since these values will be written to the register specified by the instruction. This is used for instructions that read from the FPGA, such as the fpga_get and fpga_status instructions. This is just the output from those instructions. For example, for the fpga_get instruction, this would be the computed return value. For the fpga_status instruction, this could be the current status (either not done or done) or it could be a percentage complete, or it could contain an error code, or whatever else the application designer would deem useful.

3.4 FPGA Implementation

Inside the FPGA unit, only four program design blocks are shown for simplicity. However, in a real production implementation of this, more program design blocks should be allowed as this is to be possibly shared by more than one currently active application, as defined or permitted by the operating system. The operating system will also be responsible for loading and unloading the individual program design blocks and assigning them to an index in the FPGA unit, but the modification of the operating system to support dynamically loading and unloading these program design blocks is outside the scope of this paper.

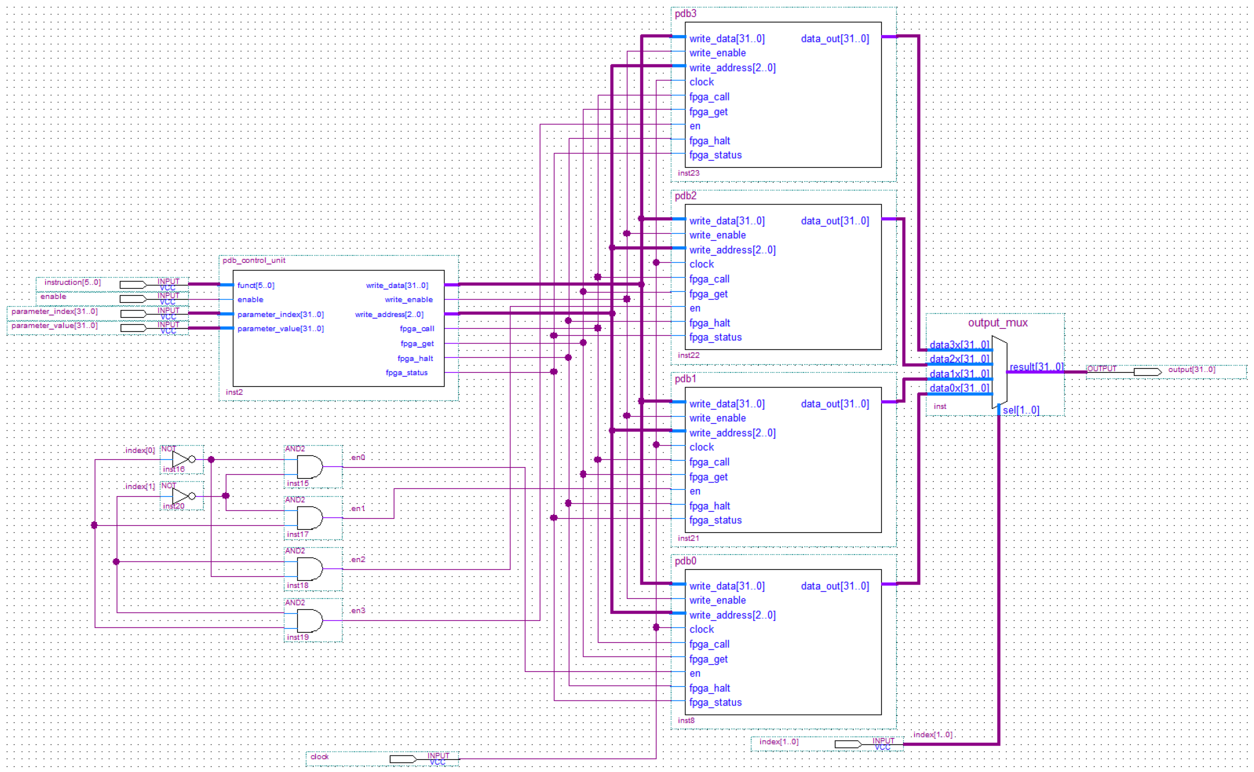


Figure 10 FPGA Implementation

A simple line decoder is implemented to gate access to the correct program design block, based on the index input. Only one program design block may be accessed (read from or written to) in a single clock cycle. The enable input that goes to the pdb_control_unit will turn on or off writes along with the functionality in the program design block and the enable that comes from the index decoder will turn on only the program design block at the specified index, while turning off the other units. This means that no matter what the inputs are from the control unit, if the index is not the same as the index of the program design block, then the program design block will ignore the inputs. If it was already processing, however, it will still be able to continue processing. This allows more than one PDB to be processing concurrently. Only writes to or reads from the program design block are limited to one per clock cycle.

The pdb_control_unit is simple and just determines which FPGA instruction is being issued and whether to write to the register file of the specified program design block, as is shown in Figure 11.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity pdb_control_unit is
port(
  funct : in std_logic_vector(5 downto 0);
  enable : in std_logic; --active high enable signal
  parameter_index : in std_logic_vector(31 downto 0);
  parameter_value : in std_logic_vector(31 downto 0);

  write_data : out std_logic_vector(31 downto 0);
  write_enable : out std_logic;
  write_address : out std_logic_vector(2 downto 0);

  fpga_call : out std_logic;
  fpga_get : out std_logic;
  fpga_halt : out std_logic;
  fpga_status : out std_logic
);
end pdb_control_unit;

architecture behavior of pdb_control_unit is
begin
  process(funct, enable, parameter_index, parameter_value) begin
    if enable='1' then
      if funct="000000" then --fpga_set
        write_enable<='1';
        write_address<=parameter_index(2 downto 0);
        write_data<=parameter_value;
        fpga_call<='0';
        fpga_get<='0';
        fpga_halt<='0';
        fpga_status<='0';
      elsif funct="000001" then --fpga_call
        write_enable<='0';
        fpga_call<='1';
        fpga_get<='0';
        fpga_halt<='0';
        fpga_status<='0';
      elsif funct="000010" then --fpga_get
        write_enable<='0';
        fpga_get<='1';
        fpga_call<='0';
        fpga_halt<='0';
        fpga_status<='0';
      elsif funct="000011" then --fpga_halt
        write_enable<='0';
        fpga_halt<='1';
        fpga_call<='0';
        fpga_get<='0';
        fpga_status<='0';
      elsif funct="000100" then --fpga_status
        write_enable<='0';
        fpga_status<='1';
        fpga_call<='0';
        fpga_get<='0';
        fpga_halt<='0';
      end if;
    else
      write_enable<='0';
    end if;
  end process;
end behavior;
```

Figure 11 PDB Control Unit

3.5 Sample Program Design Block Implementation

How the individual program design blocks are implemented is an application-specific detail, but Figure 12 shows a simple reference implementation. Most program design blocks should probably look somewhat similar to this as there is no application logic here, though the choice of how to store parameters may vary. For example, here a register file with eight registers is used even though only one register is actually used. The reason for this is to allow all of the examples in this paper to reuse this register file design without having to design a separate register file for each example. But in a production system, this could likely be synthesized by the design tools so that the application designer does not need to specify the program design block implementation. Instead, they would be able to focus on the implementation of the logic unit, which is where the application-specific logic goes. The register file is shown in Figure 14.

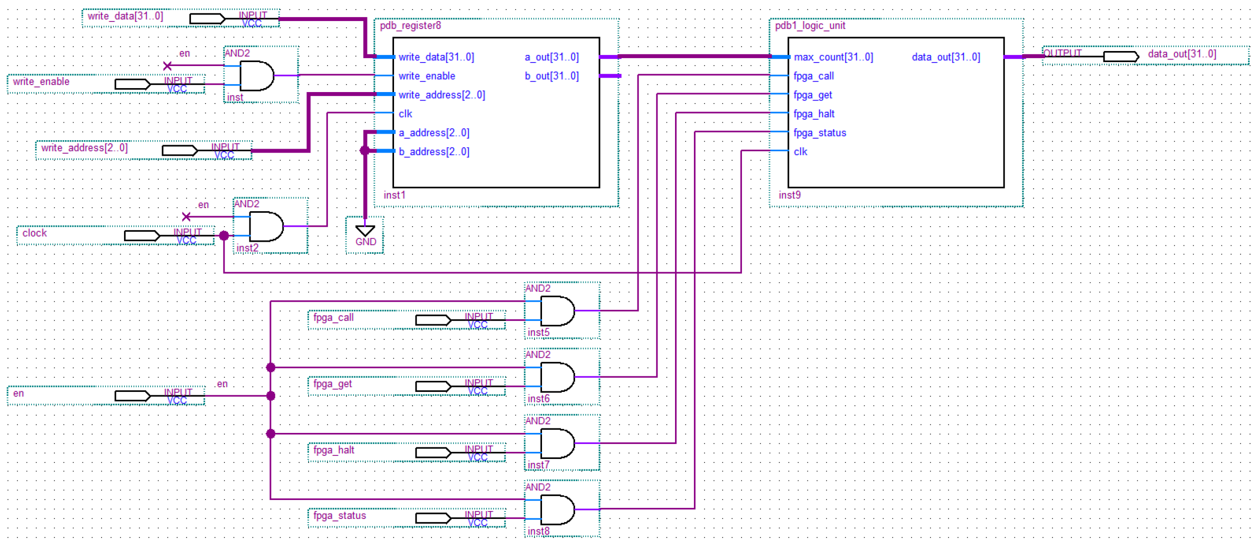


Figure 12 Sample Program Design Block

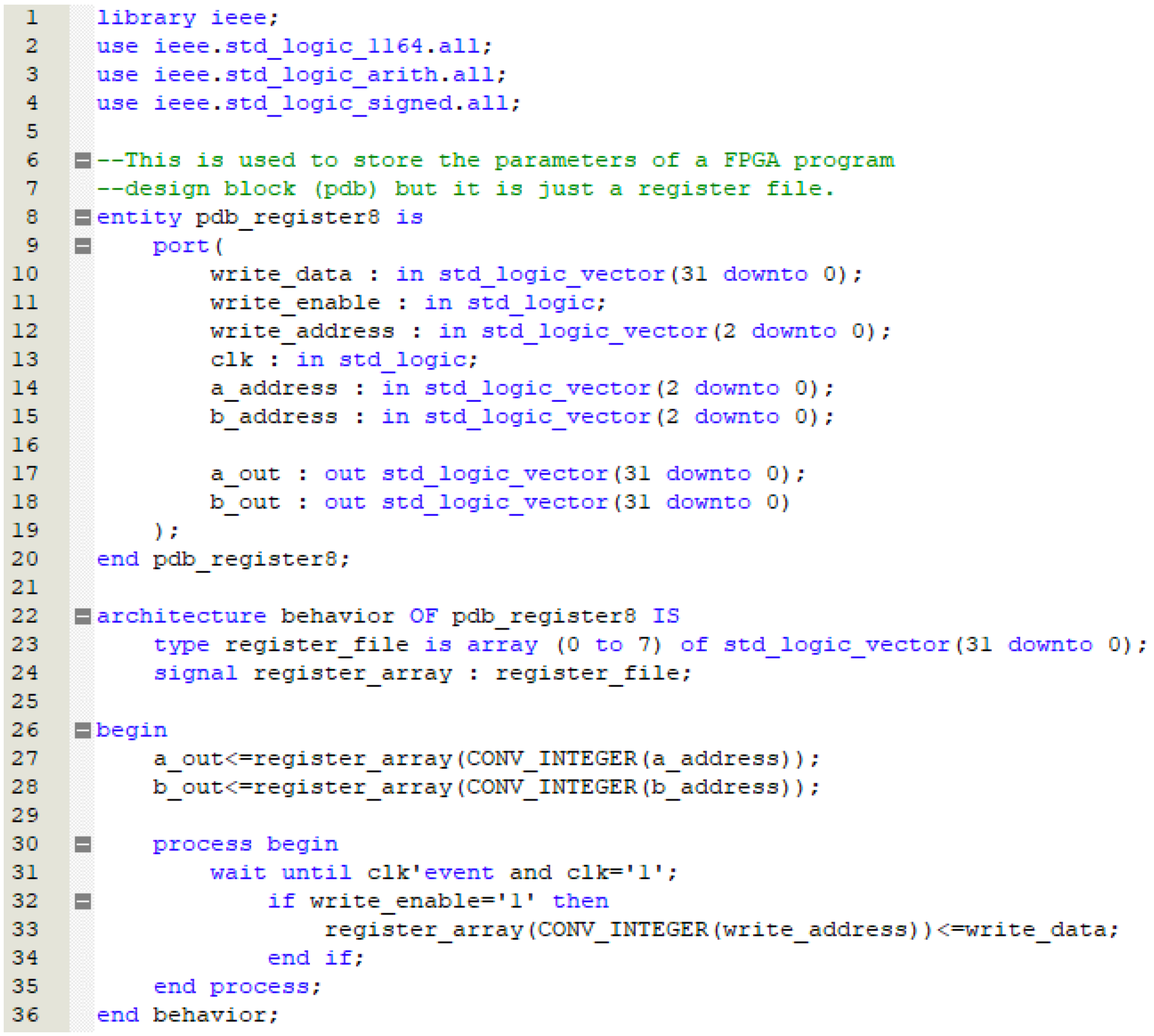
The pdb_logic_unit is where the application-specific logic goes. It takes the value(s) in from the register along with the signals from the pdb_control_unit, which indicate the current instruction, and it processes the data as is required by application requirements. Figure 13 shows a template of a simple implementation. This does not show any actual logic, but rather just a template of how to handle the inputs. A complete implementation will be shown in chapter 4.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity pdb_logic_unit is
port(
  max_count : in std_logic_vector(31 downto 0);
  fpga_call : in std_logic;
  fpga_get : in std_logic;
  fpga_halt : in std_logic;
  fpga_status : in std_logic;
  clk : in std_logic;

  data_out : out std_logic_vector(31 downto 0)
);
end pdb_logic_unit;

architecture behavior of pdb_logic_unit is
  signal enable : std_logic;
  --Application-specific signals or variables
begin
  process(clk) begin
    if clk'event and clk='1' then
      if enable='1' then
        --This is where the application-specific processing
        --could go. Perform necessary computations as long
        --as the processing is not complete.
      end if;

      if fpga_halt = '1' then
        enable <= '0';
        --Perform any application-specific cleanup here, or
        --that could be performed in the fpga_call statement.
      elsif fpga_status = '1' then
        --Populate the data_out output with the application-
        --specific status codes. This could be as simple as
        --zero means not done, and one means done, or it could
        --have error codes or different status codes, such as
        --a percent complete, if available. This is completely
        --up to the application designer.
      elsif fpga_get = '1' then
        --Populate the data_out output with the current
        --computation results. If the computation is not
        --complete, then the return value here is undefined
        --and it is up to the application designer how to handle
        --this.
      elsif fpga_call = '1' then
        enable <= '1';
        --Start the computation process. This could also
        --perform any cleanup or resetting that needs
        --to be done between invocations.
      end if;
    end if;
  end process;
end behavior;
```

Figure 13 Sample PDB Logic Unit

The register file used here contains eight registers that are each 32-bits wide, and it allows reading from two registers at once. The dual port register file is used in one of the examples in the next chapter that takes in two parameters. In general, for each parameter the circuit requires, a new read port would be added to the register file to accommodate the additional parameter.

This register file is used for all of the examples in this paper, even though not all of the examples actually use all of the registers. This is so that it will not be necessary to design separate register files for each example. In a real, production-ready, system however, a more appropriate register system would most likely be selected. For example, if only one parameter is used, then a single register could be used, instead of eight. As has been previously mentioned, this is also something that would likely be automated and abstracted away from the application developer and synthesized by the design tools. All the application developer should have to focus on is the implementation of the logic unit.



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

--This is used to store the parameters of a FPGA program
--design block (pdb) but it is just a register file.
entity pdb_register8 is
    port(
        write_data : in std_logic_vector(31 downto 0);
        write_enable : in std_logic;
        write_address : in std_logic_vector(2 downto 0);
        clk : in std_logic;
        a_address : in std_logic_vector(2 downto 0);
        b_address : in std_logic_vector(2 downto 0);

        a_out : out std_logic_vector(31 downto 0);
        b_out : out std_logic_vector(31 downto 0)
    );
end pdb_register8;

architecture behavior OF pdb_register8 IS
    type register_file is array (0 to 7) of std_logic_vector(31 downto 0);
    signal register_array : register_file;

begin
    a_out<=register_array(CONV_INTEGER(a_address));
    b_out<=register_array(CONV_INTEGER(b_address));

    process begin
        wait until clk'event and clk='1';
            if write_enable='1' then
                register_array(CONV_INTEGER(write_address))<=write_data;
            end if;
    end process;
end behavior;
```

Figure 14 FPGA Register File

4 Results

This section of the paper shows the performance differences between the software-only application and the hybrid software-hardware application. The processor used in this paper is not pipelined, so each instruction takes several clock cycles to complete and the clock frequency used to drive the processor is 6.25MHz. Both of these factors affect the speed of the program that is simulated and in a modern commercial system, the processor would be pipelined, and the clock frequency would be much higher than the frequency used for this paper. However, neither of these factors affect the ability to compare the software-only implementation and the hybrid implementation. Both implementations are run on the same processor.

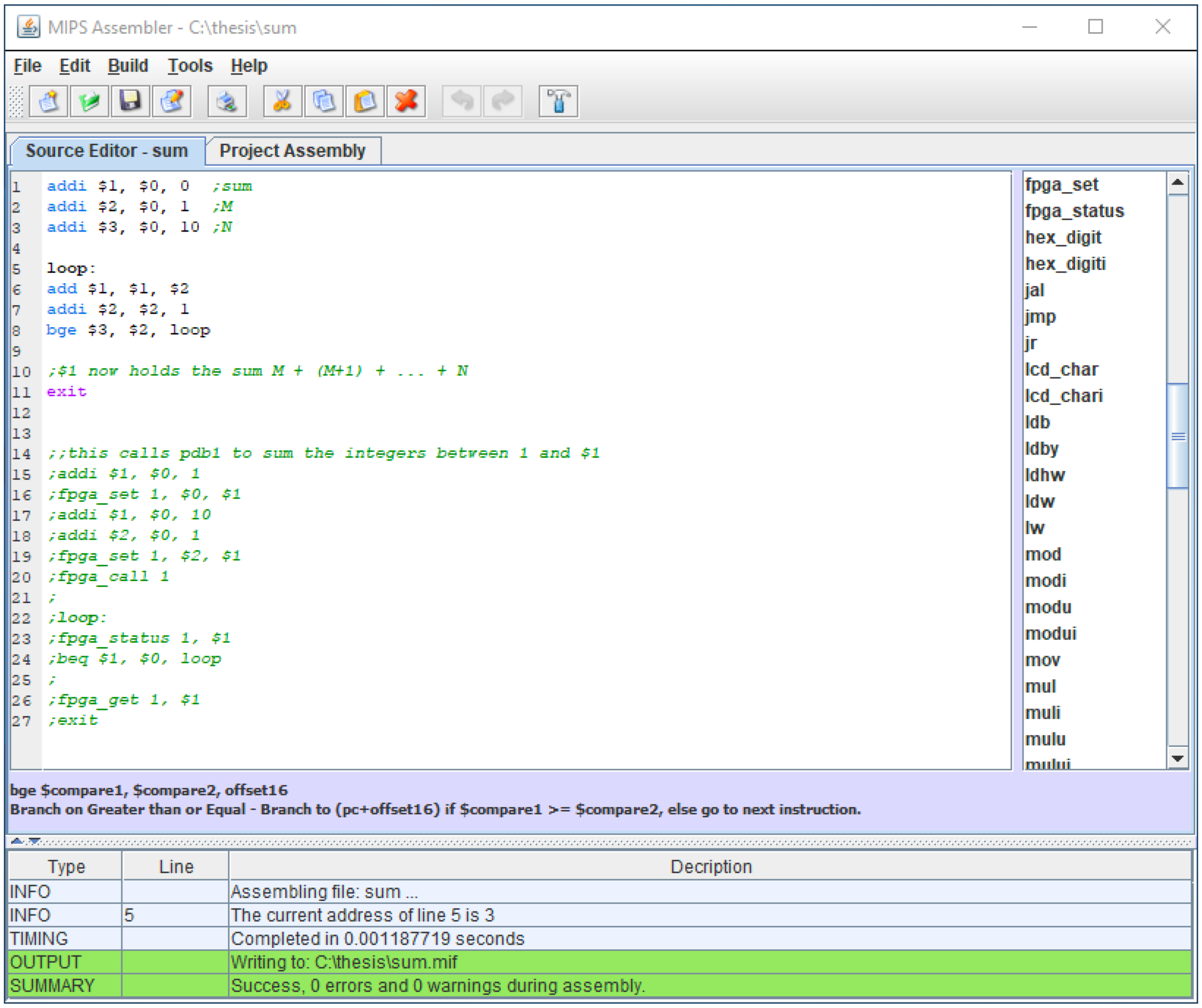
If the processor was pipelined, then it would be possible to have an average rate of one clock cycle per instruction, which is known as a scalar processor. As is, the average rate of clock cycles per instruction is much more than one, or sub-scalar. But this is true for the new FPGA instructions as well as for the regular instructions, since they are all running on the same processor. The comparisons in this section count the number of instructions that are used by the software-only implementation and the hybrid implementation. This way the results are not tied to how many clock cycles each example takes to complete, as this would vary between a sub-scalar and scalar processor, but rather the number of instructions executed, which is a more fair comparison as it is not dependent on the architecture of the processor on which the program is run.

This section shows the hybrid implementation outperforms the software-only implementation because the hardware logic is not limited to only the instructions in the instruction set of the processor, but rather can implement any arbitrary logic. This allows something that would normally take several processor instructions to complete to execute in a single instruction. For example, the parity computation performed in section 4.3 illustrates this concept by performing a bitwise exclusive-or on all 32 bits of the input in one instruction, whereas the software-only implementation takes several instructions per bit. This is just one example. But one of the benefits of the hybrid approach is that application-specific logic that normally takes many instructions on a general-purpose processor to execute can be executed much more efficiently since it is not limited by the instruction set of the processor.

4.1 Simulation Process

The programs in this paper were all assembled using a custom assembler made for this processor, shown in Figure 15. This produces a memory initialization file that is loaded into the memory unit from which the processor reads. The program is then simulated using the simulator provided in Altera Quartus II 7.2.

This same simulation process is followed for both the software-only program and the hybrid software-hardware program. The results of the simulation are then compared by examining the number of instructions required by each version of the program.



```
1  addi $1, $0, 0  ;sum
2  addi $2, $0, 1  ;M
3  addi $3, $0, 10 ;N
4
5  loop:
6  add $1, $1, $2
7  addi $2, $2, 1
8  bge $3, $2, loop
9
10 ;$1 now holds the sum M + (M+1) + ... + N
11 exit
12
13
14 ;;this calls pdb1 to sum the integers between 1 and $1
15 ;addi $1, $0, 1
16 ;fpga_set 1, $0, $1
17 ;addi $1, $0, 10
18 ;addi $2, $0, 1
19 ;fpga_set 1, $2, $1
20 ;fpga_call 1
21 ;
22 ;loop:
23 ;fpga_status 1, $1
24 ;beq $1, $0, loop
25 ;
26 ;fpga_get 1, $1
27 ;exit
```

bge $compare1, $compare2, offset16
Branch on Greater than or Equal - Branch to (pc+offset16) if $compare1 >= $compare2, else go to next instruction.

Type	Line	Decription
INFO		Assembling file: sum ...
INFO	5	The current address of line 5 is 3
TIMING		Completed in 0.001187719 seconds
OUTPUT		Writing to: C:\thesis\sum.mif
SUMMARY		Success, 0 errors and 0 warnings during assembly.

Figure 15 Assembler

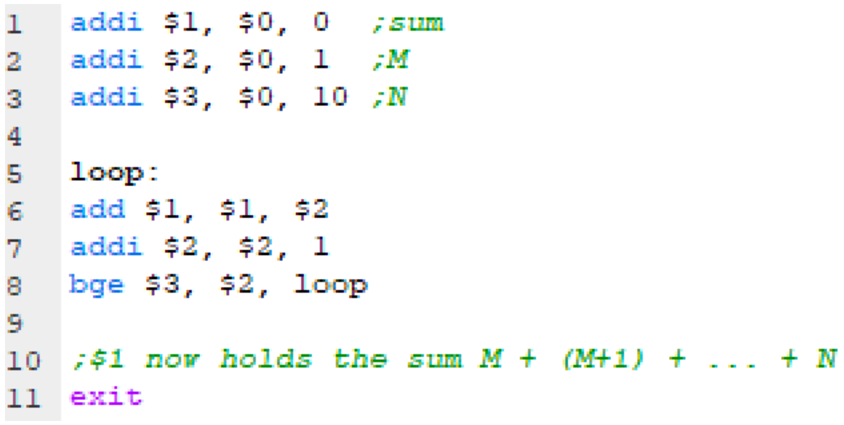
4.2 Example: Sum M to N, Inclusive

This example is a simple example just showing looping and polling with the FPGA instructions. The program takes in two parameters, M and N, and calculates the sum of the integers between M and N, inclusive. This could be implemented using the following summation formula:

$$\sum_{i=M}^{N} i = \sum_{i=1}^{N} i - \sum_{i=1}^{M-1} i = \frac{N(N+1) - M(M-1)}{2}$$

But it has been implemented by just iterating through the integers between M and N, inclusive, and summing the integers. This has been done to show the polling of the FPGA and to show that the FPGA unit is producing one sum per clock cycle, as opposed to the software-only approach, which, even on a pipelined processor that executes one instruction per clock cycle, still takes three clock cycles per loop.

4.2.1 Software-Only Implementation

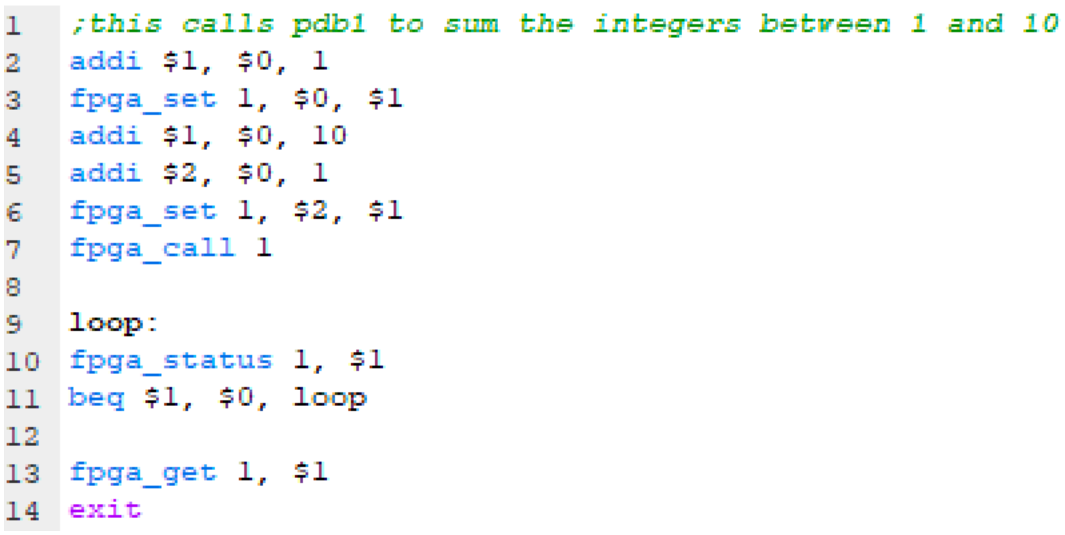


```
addi $1, $0, 0  ;sum
addi $2, $0, 1  ;M
addi $3, $0, 10 ;N

loop:
add $1, $1, $2
addi $2, $2, 1
bge $3, $2, loop

;$1 now holds the sum M + (M+1) + ... + N
exit
```

Figure 16 Software-only summation program

This program just initializes the parameters, M and N, and it sets the current sum to zero. Then, in each iteration of the loop from M to N, inclusive, it increments the sum by the current value of M and it increments the value of M. Once M is larger than N, the loop stops, and the program is done. The result of the summation is in register $1.

4.2.2 Hybrid Implementation



```
1  ;this calls pdb1 to sum the integers between 1 and 10
2  addi $1, $0, 1
3  fpga_set 1, $0, $1
4  addi $1, $0, 10
5  addi $2, $0, 1
6  fpga_set 1, $2, $1
7  fpga_call 1
8
9  loop:
10 fpga_status 1, $1
11 beq $1, $0, loop
12
13 fpga_get 1, $1
14 exit
```

Figure 17 Hybrid summation program

Program design block one is used for the FPGA instructions in this example as that is where the summation program is loaded. It takes two parameters. Parameter index zero is the starting count value (M) and parameter index one is the maximum count value (N). The first few lines set the parameters and then the program design block is invoked by calling fpga_call on line seven. The loop just polls the status of the circuit. If it is done, then it will produce a value of one. Otherwise, if it is still processing, the status will be zero.

The parameter indices that are used in the assembly language program listing in Figure 17 are specified in Figure 18 by the a_address and b_address inputs to the register file, since that is what supplies the value to the logic unit max_count and min_count inputs. The input to a_address is a value of one, which means register one will be read from for the value of max_count. The input to b_address is zero, which will cause register zero to be used for the input to min_count.

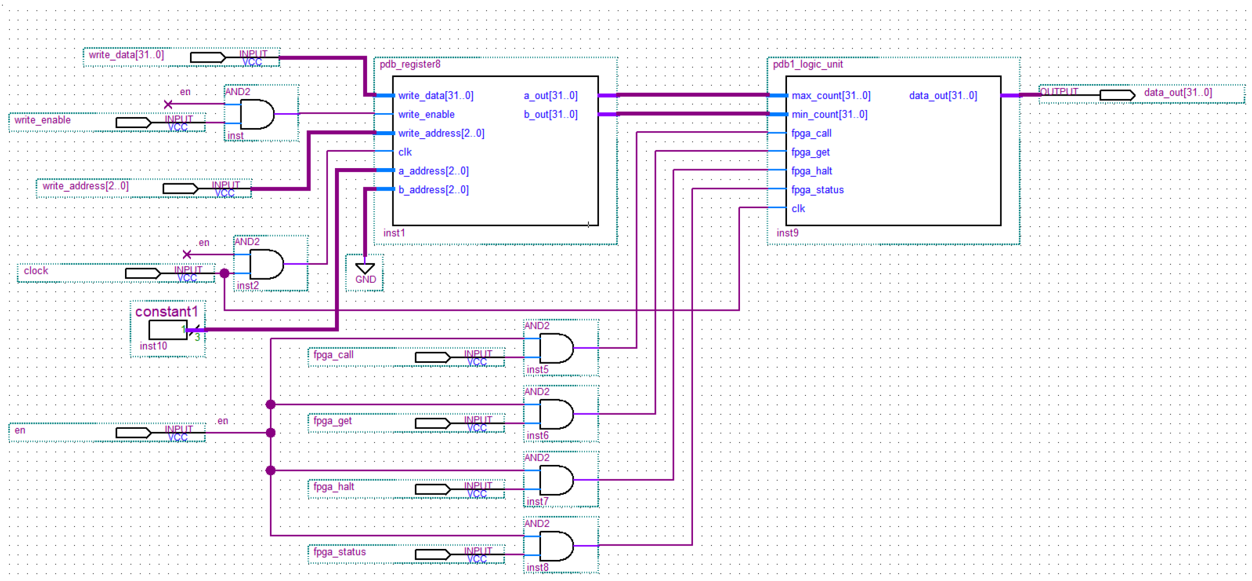
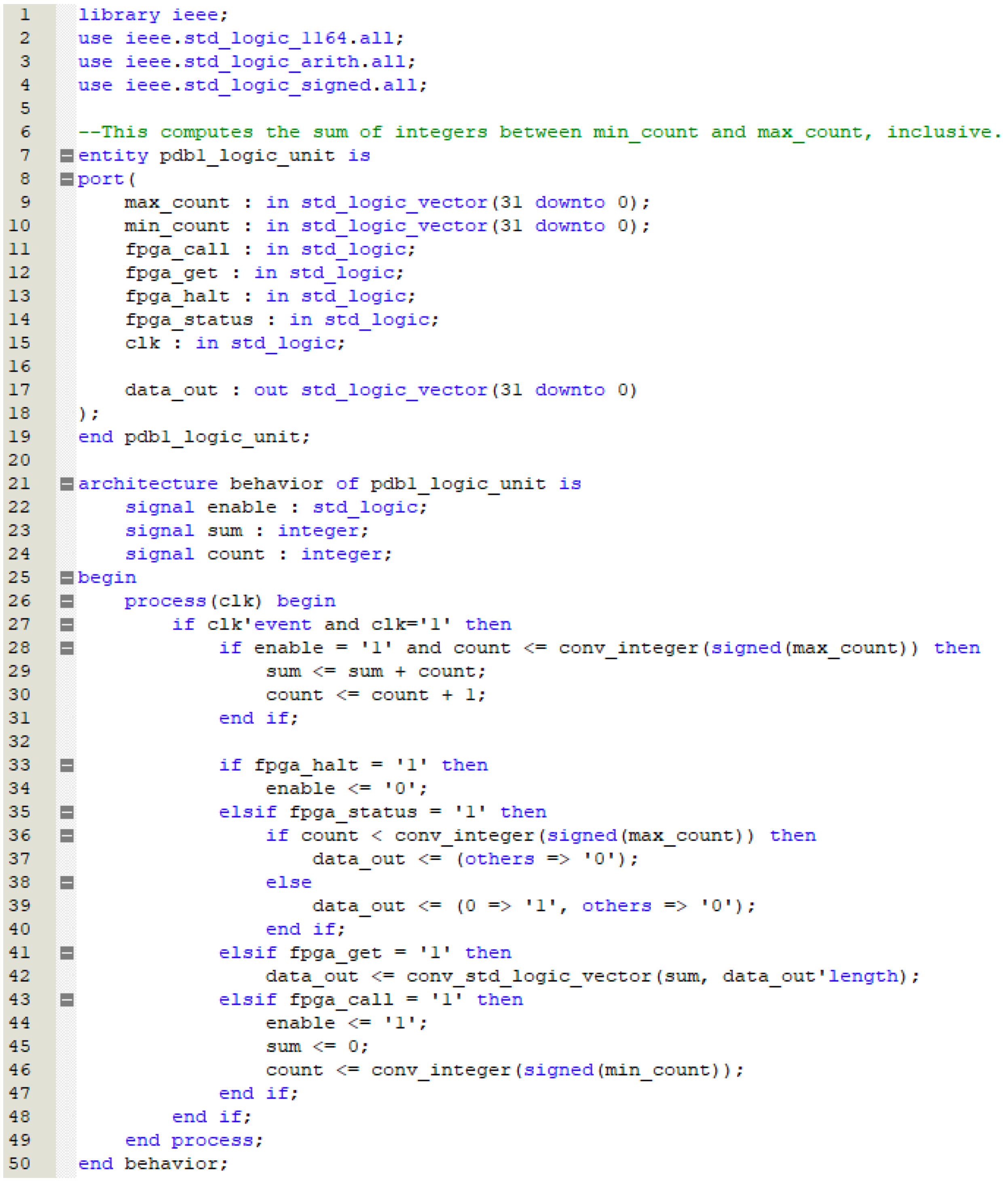


Figure 18 Summation program PDB

The logic unit, shown in Figure 19, performs the actual summation.



```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

--This computes the sum of integers between min_count and max_count, inclusive.
entity pdbl_logic_unit is
port(
    max_count : in std_logic_vector(31 downto 0);
    min_count : in std_logic_vector(31 downto 0);
    fpga_call : in std_logic;
    fpga_get : in std_logic;
    fpga_halt : in std_logic;
    fpga_status : in std_logic;
    clk : in std_logic;

    data_out : out std_logic_vector(31 downto 0)
);
end pdbl_logic_unit;

architecture behavior of pdbl_logic_unit is
    signal enable : std_logic;
    signal sum : integer;
    signal count : integer;
begin
    process(clk) begin
        if clk'event and clk='1' then
            if enable = '1' and count <= conv_integer(signed(max_count)) then
                sum <= sum + count;
                count <= count + 1;
            end if;

            if fpga_halt = '1' then
                enable <= '0';
            elsif fpga_status = '1' then
                if count < conv_integer(signed(max_count)) then
                    data_out <= (others => '0');
                else
                    data_out <= (0 => '1', others => '0');
                end if;
            elsif fpga_get = '1' then
                data_out <= conv_std_logic_vector(sum, data_out'length);
            elsif fpga_call = '1' then
                enable <= '1';
                sum <= 0;
                count <= conv_integer(signed(min_count));
            end if;
        end if;
    end process;
end behavior;
```

Figure 19 Summation program logic unit

It does the addition, count update, and comparison in one clock cycle, as opposed to the software-only implementation that does each of those operations in a separate instruction.

4.2.3 Performance Comparison

4.2.3.1 Waveform of the Software-Only Implementation

The first waveform, shown in Figure 20, shows the software-only implementation that was shown in Figure 16.

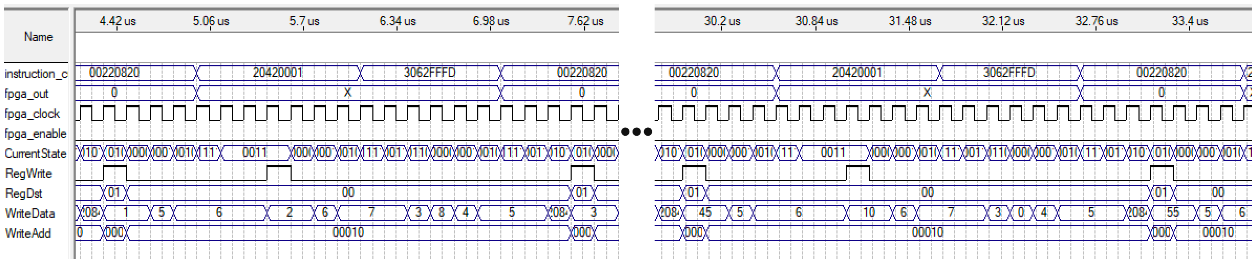


Figure 20 Software-only summation program waveform

The waveform starts at the `add $1, $1, $2` instruction (0x00220820), which is the first line in the loop. This is around 3.8us where it starts. The summation is complete around the 33us mark, but there is still the increment and comparison instructions left in the loop to do after this.

4.2.3.2 Waveform of the Hybrid Implementation

The next figure shows the waveform of the hybrid program.

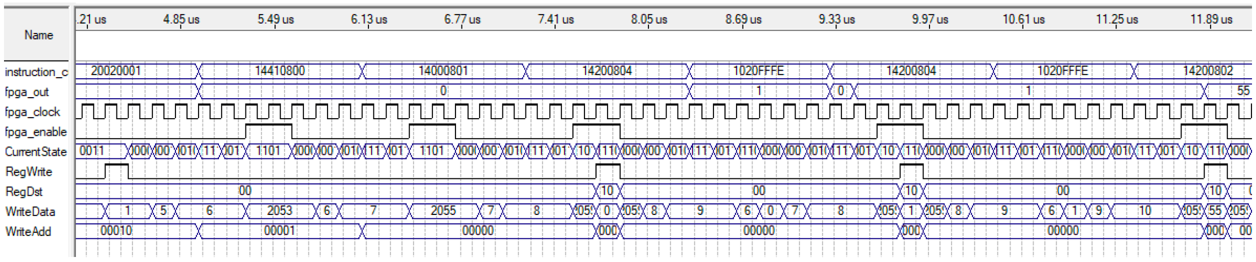


Figure 21 Hybrid summation program waveform

The fpga_call instruction (0x14000801) starts around 6.1us. At the rising clock edge after the fpga_enable line goes high in this instruction, the program design block will start processing, which happens around time 6.55us. The first fpga_status call happens at time 7.5us, and the status is zero at that point, which means the computation is not done yet. This is six clock cycles into the computation, so in four more clock cycles, the circuit will be done, but thanks to the branch instruction, there is an extra instruction delay. The next time the circuit is checked, which happens at time 9.6us, the status returned is one, which means the circuit is done. Actually, it was done at around time 8.15us, but due to the instructions taking several clock cycles each to complete and the branch instruction being there, the result was delayed. The sum is read, using the fpga_get instruction, and written to the register at time 11.83us.

4.2.3.3 Comparison Analysis

The software-only program uses 30 instructions in the loop because it uses three instructions per iteration for 10 iterations. The hybrid program uses one instruction for the fpga_call, two fpga_get instructions, two branch instructions, and then an fpga_get instruction, for a total of six instructions. The two branch instructions could have been removed by just putting a no-op instruction, an instruction that does nothing, in place of the first loop and then calling the fpga_get after that. That would reduce the instruction count to three instructions (fpga_call, no-op, fpga_get).

The hybrid program took a few extra instructions to get setup because of the fpga_set instructions. This could have been reduced to where it would have taken the same number of instructions as the software-only program by having a version of the fpga_set instruction that took an immediate value as an operand rather than having to load the index into a register and then set the parameter index using the value in the register. This would be more closely aligned with what the software-only version is able to do with the add immediate instructions. But even with it taking a few more instructions to initialize the program design block, the computation is done in much fewer instructions than the software-only approach. This is because during each clock cycle, the program design block is able to perform an entire iteration of the sum, increment, and compare loop that takes the software-only implementation three instructions to do. Even if the processor was pipelined, and it was capable of executing one instruction per clock cycle, it is still a two-thirds reduction in clock cycles per loop because the hybrid version is able to execute more than one instruction per clock cycle.

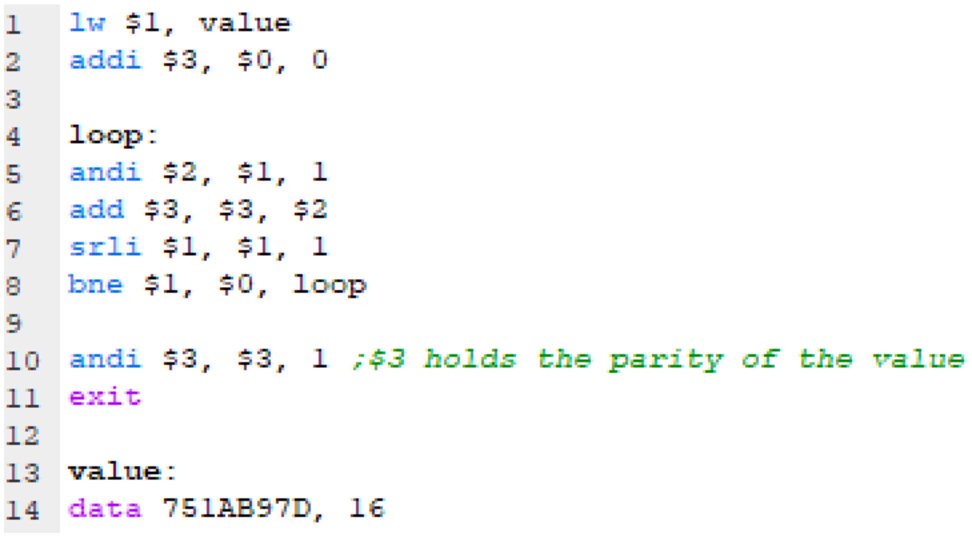
4.3 Example: Parity Checker

This example implements a parity computation. For this example, the parity of a number refers to the evenness or oddness of the number of bits in a number that are set to one. If the input has an even number of bits set to one, then the parity will be zero. If the input has an odd number of bits set to one, then the parity will be one. For example, the number five has a binary representation of 101_2. This has two bits that are set to one, which is an even number of bits set to one, so the parity returned by this function will be zero for an input of five.

Parity computation is commonly done as a simple way of validating transmitted data. For example, if the data transmitted is 0101, then this will have an even number of bits set to one, so the parity will be zero. If using an even parity bit, the value zero will be appended to the input, like so: 01010. If using an odd parity bit, the value zero will be inverted and appended to the input, like so: 01011. This parity bit is then used on the receiving system to verify that the number of ones transmitted is still the same and did not get messed up during transmission. Of course, this just ensures the number of ones transmitted is the same. The message could still have been changed if an even number of ones all got changed, but that is another problem and is outside the scope of this example. This example just implements a simple parity computation.

The program takes in one parameter, N, which is the number of which to compute the parity.

4.3.1 Software-Only Implementation



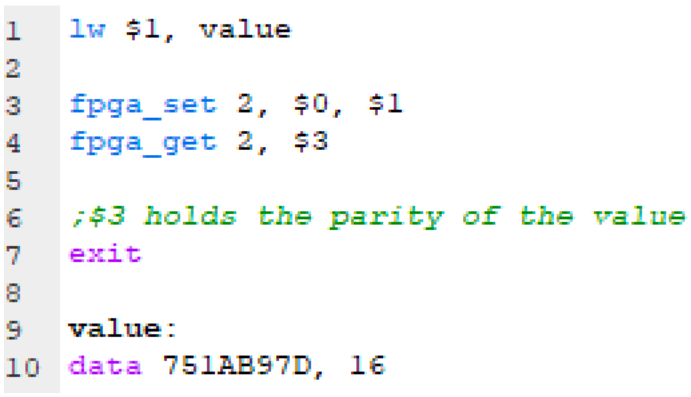
```
lw $1, value
addi $3, $0, 0

loop:
andi $2, $1, 1
add $3, $3, $2
srli $1, $1, 1
bne $1, $0, loop

andi $3, $3, 1 ;$3 holds the parity of the value
exit

value:
data 751AB97D, 16
```

Figure 22 Software-only parity checker

The value to use when computing the parity is loaded from memory into register one. Register three is used to contain the count of bits that are set to one, so it is initialized to zero next. Then, in the loop, the least significant bit is examined to determine if it is a one or a zero. This is put into a temporary register, register two, and then the result, which is either a one or a zero if the bit is set or not, respectively, is added to register three. Next, the input value is halved so that the next least significant bit may be examined the next time through the loop. This continues as long as the value is non-zero. If the value is zero, the loop is terminated. This is helpful for small values such as 101_2, which will only run through the loop three times. So, this algorithm runs in time that is logarithmic with respect to the input value.

For the example shown, a value of 0x751AB97D is selected so that it will run through the loop a sufficient number of times.

4.3.2 Hybrid Implementation



```
lw $1, value

fpga_set 2, $0, $1
fpga_get 2, $3

;$3 holds the parity of the value
exit

value:
data 751AB97D, 16
```

Figure 23 Hybrid parity checker

The assembly language for the hybrid implementation is simpler, only involving the fpga_set instruction to set the parameter to the value loaded from memory, and the fpga_get instruction to retrieve the parity value from the program design block. This example is using program design block index two.

No fpga_call instruction is necessary because it is only setting the enable of the circuit and there is no state for this circuit. It is just combinational logic. So, when it gets an input, it produces an output right then. This is unlike the previous example that calculated the sum between two integers. For that example, it could take more than one clock cycle to complete, so it did use the enable signal. For this example, the enable signal is not necessary because there is really nothing to enable or disable as this is just a combinational logic circuit.

Likewise, no fpga_status instruction is necessary because once the program design block receives the input, the output is then ready.

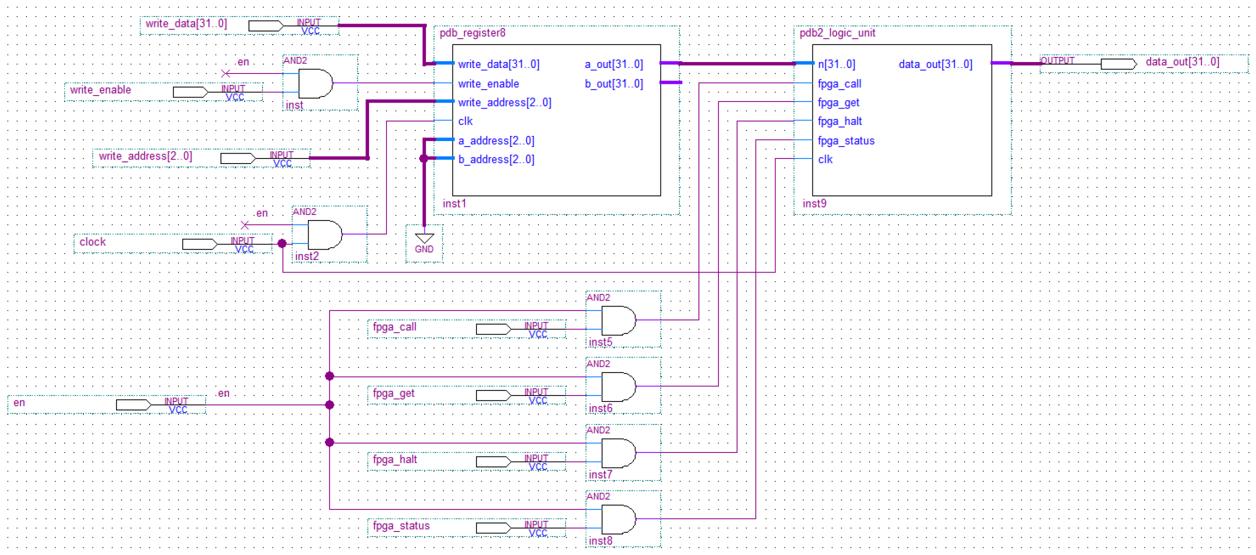


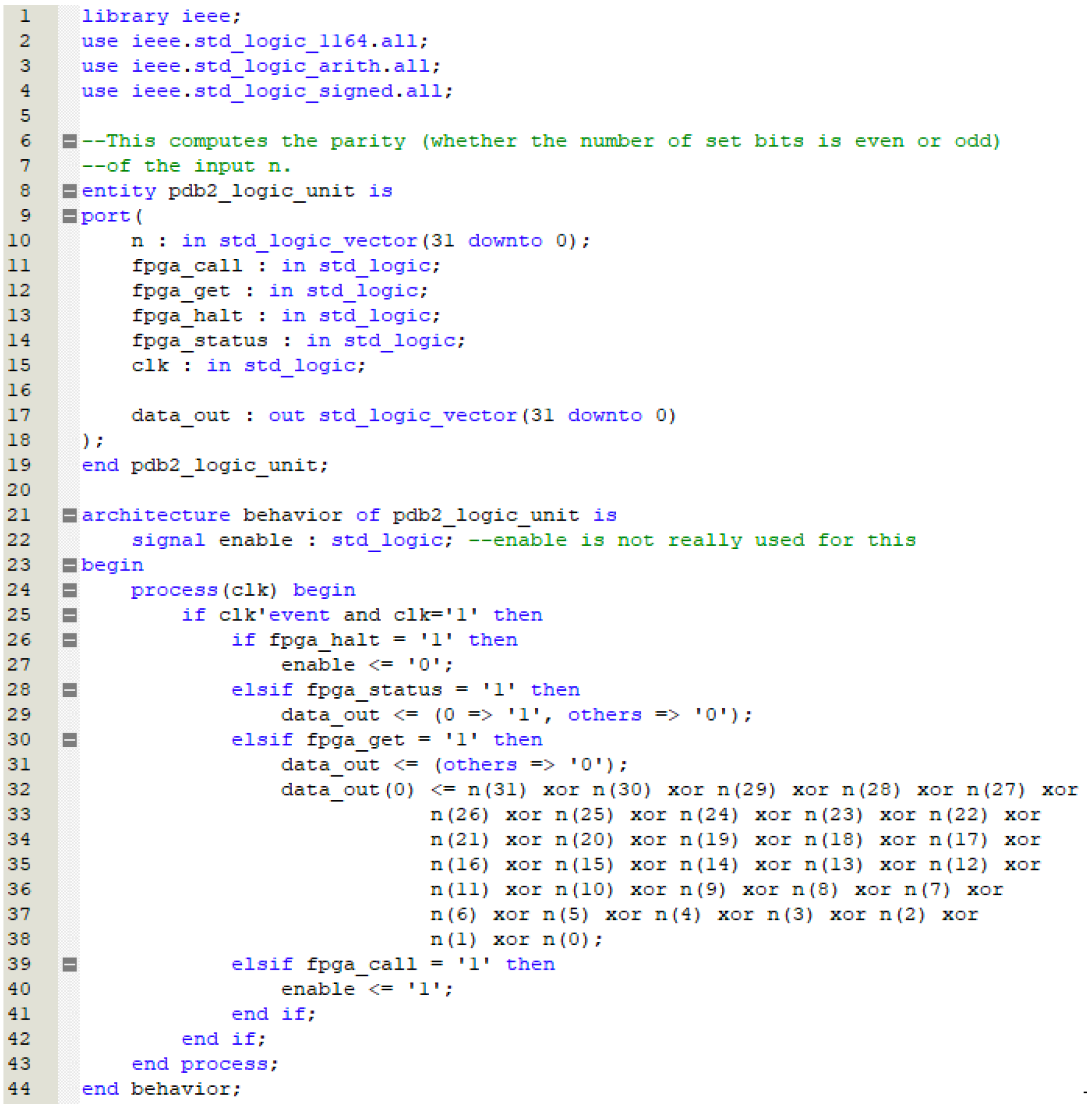
Figure 24 Parity checker program PDB

The program design block is shown, and it is very similar to the program design block shown in Figure 18 of section 4.2. The only difference is the number of parameters and the parameter indices. Other than that, they are the same.

The logic unit, shown in Figure 25, performs the actual parity computation. It performs a bitwise exclusive-or of all of the bits in the input.

$$parity = N_{31} \oplus N_{30} \oplus N_{29} \oplus \cdots N_1 \oplus N_0$$

This is slightly different from how the software-only implementation is performed because even if the XOR instruction in the processor is used, it is still necessary to get just the least significant bit and to XOR this with the previous XOR result. So, there will not be any instructions saved by doing this in the software-only implementation. However, with the hardware implementation, the entire logic is achievable in one instruction, rather than having to perform all of the instructions that the software-only implementation performs.



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

--This computes the parity (whether the number of set bits is even or odd)
--of the input n.
entity pdb2_logic_unit is
port(
    n : in std_logic_vector(31 downto 0);
    fpga_call : in std_logic;
    fpga_get : in std_logic;
    fpga_halt : in std_logic;
    fpga_status : in std_logic;
    clk : in std_logic;

    data_out : out std_logic_vector(31 downto 0)
);
end pdb2_logic_unit;

architecture behavior of pdb2_logic_unit is
    signal enable : std_logic; --enable is not really used for this
begin
    process(clk) begin
        if clk'event and clk='1' then
            if fpga_halt = '1' then
                enable <= '0';
            elsif fpga_status = '1' then
                data_out <= (0 => '1', others => '0');
            elsif fpga_get = '1' then
                data_out <= (others => '0');
                data_out(0) <= n(31) xor n(30) xor n(29) xor n(28) xor n(27) xor
                            n(26) xor n(25) xor n(24) xor n(23) xor n(22) xor
                            n(21) xor n(20) xor n(19) xor n(18) xor n(17) xor
                            n(16) xor n(15) xor n(14) xor n(13) xor n(12) xor
                            n(11) xor n(10) xor n(9) xor n(8) xor n(7) xor
                            n(6) xor n(5) xor n(4) xor n(3) xor n(2) xor
                            n(1) xor n(0);
            elsif fpga_call = '1' then
                enable <= '1';
            end if;
        end if;
    end process;
end behavior;
```

Figure 25 Parity checker program logic unit

4.3.3 Performance Comparison

4.3.3.1 Waveform of the Software-Only Implementation

The first waveform, shown in Figure 26, shows the software-only implementation that was shown in section 4.3.1.

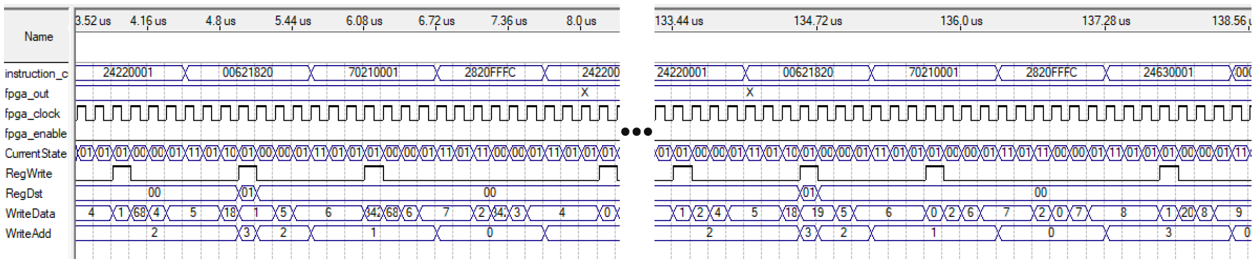


Figure 26 Software-only parity checker program waveform

The waveform starts at the `andi $2, $1, 1` instruction (0x24220001), which is the first line in the loop. It shows all the way up to the `andi $3, $3, 1` instruction (0x24630001), which is after the loop exits. This shows a value of one being written to register three, meaning there is an odd number of bits that are set to one in the input value. Previously, a few instructions prior to that, it can be seen that the value of 19 is being written to register three. This means there are 19 bits with a value of one in the input value, which is correct. So, the parity of the number is a one, meaning there are an odd number of ones in the number.

4.3.3.2 Waveform of the Hybrid Implementation

The next figure shows the waveform of the hybrid program.

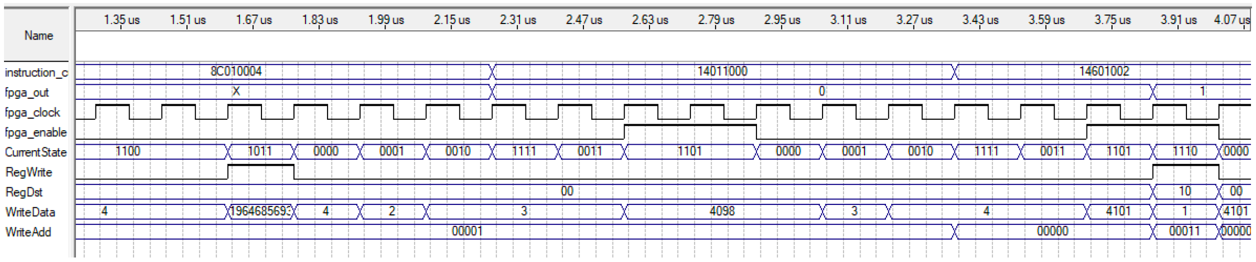


Figure 27 Hybrid parity checker program waveform

The entire program is shown. The load word instruction that loads the value from memory is shown (0x8C010004), followed by the fpga_set instruction where the value is passed to the program design block (0x14011000), and finally the fpga_get instruction to retrieve the value from the program design block and to store it in register three (0x14601002). This shows a value of one being written to register three, which is consistent with the software-only implementation.

4.3.3.3 Comparison Analysis

The software-only program uses 124 instructions in the loop because there are four instructions per loop for 31 iterations of the loop for the particular value selected. As a contrast, the hybrid program does not have a loop. It uses one instruction to set the parameter value and one instruction to get the result. So, it only uses two instructions as compared to 124. And it will always be able to perform this work in two instructions no matter how big the number is. The software-only implementation depends on the number to determine the number of instructions required, whereas the hybrid program does not.

So, even if the processor used a pipelined architecture, and was able to execute one instruction per clock cycle, the hybrid program is still much faster because it required far fewer instructions due to not being restricted to a limited instruction set.

5 Future Work

The proposal presented in this paper is mostly a proof-of-concept to show the performance gains of implementing some functionality directly in hardware. There is still a lot of remaining work that needs to be done before this could be productionized in a commercial system. This section describes the remaining work and some possible complications that could arise when implementing the remaining work.

5.1 Use a Higher Capacity FPGA

The FPGA used in this paper is the Altera Cyclone II EP2C35F672C6 FPGA. This is a relatively low capacity and low performance FPGA compared to higher-end FPGAs. For instance, the Altera Stratix 10 GX 5500 FPGA has 5.51 million logic elements and is built on a 14nm process (Altera). Similarly, Xilinx has the Xilinx Virtex UltraScale+ VU13P FPGA, which has 3.78 million logic elements and is built on a 16nm process (Xilinx). Both of these FPGAs allow much higher clock frequencies to be used and have many times more logic elements than the Cyclone II FPGA used in this paper. But both of these FPGAs are also considerably more expensive than the Cyclone II FPGA, as well as most FPGAs, and as a result, could cause the computer system in which they would be used to become too expensive to be widely sold. But there are plenty of other FPGAs that are cheaper and still high enough capacity and high enough performance for most uses.

The choice of the FPGA is important because it will limit how many program design blocks may be installed at a time and how fast of a clock signal may be used in those designs. It is also important because the price of the FPGA will affect the overall system price. So, a balance needs to be reached between capacity, performance, and price. It may be that low-end systems may not have an FPGA due to cost considerations and would only run the software-only applications, whereas mid-end to high-end systems may have varying levels of FPGAs used in them. The application designers would have to develop their application to be able to adjust to the features of the system on which the application is running. For example, on a low-end system with no FPGA, the application would need to be able to query the system resources available through the operating system and determine that no FPGA is present and run in a software-only mode. To achieve the ability to run a method in either hardware or software mode, the application developer would have to have both a hardware and software implementation of the methods that are hardware-enabled. Alternatively, the developer could choose to not support systems that do not meet the minimum requirements of the application if they want to only run on systems with FPGAs and to not provide a software implementation.

Similarly, operating systems should provide the ability for applications to query about the available system resources, such as the FPGA. The operating system should be able to tell whether there is or is not an FPGA present, what its capacity is, along with other characteristics of the device. This way the application can, if programmed to handle it, scale its features to be able to run on the system. This could pose more work to application developers, but it would ensure their application is able to run on a wider range of systems.

5.2 Add FPGA Instructions to the Instruction Set

Processor manufacturers, such as Intel and AMD along with others, would need to add support for the FPGA instructions to their instruction set. This is just like was done in this paper. The instruction set was modified to include support for the new FPGA instructions. But that was just for this one custom processor. For this proposal to be used commercially, commercial processor manufacturers would need to apply the same, or similar, modifications to their instruction set. They could also add support for additional instructions, such as immediate addressed FPGA instructions, as were mentioned briefly in chapter four.

5.3 Add the FPGA to the System

Similar to the previous section, system manufacturers, such as HP and Apple along with others, would need to actually add the FPGA to the system and connect it to the processor. This is similar to what was done in section 3.3. Additionally, manufacturers could also connect the FPGA to the system memory, so it would have access to values in memory. But this could also pose additional security concerns if the memory access is not regulated by the operating system and would likely need to be addressed.

One requirement the FPGA needs to support though is the ability to do partial reconfiguration. This is required because the operating system will need to be able to load and unload program design blocks dynamically, while the other program design blocks remain unaffected. This will happen, for example, when a new hybrid program is loaded. The program design blocks it uses will be loaded into the FPGA by the operating system. Any existing program design blocks for currently running programs will need to remain unchanged. Or if a hybrid program is unloaded from memory and a new hybrid program is launched and there is not enough free room to just load the new program design blocks, then the operating system will need to unload the inactive program design blocks and replace them with the blocks from the newly launched program. All of this while the remaining blocks are unaffected.

5.4 Operating System Modifications

There are several modifications that would be required for operating systems. None of them were implemented in this paper because the programs written for this paper were written to run directly on the processor, but real-world programs would be written to run on an operating system.

5.4.1 Dynamic FPGA Management

Operating systems would need to manage the FPGA allocations. For example, if an application is requesting to install a program design block, the operating system would need to determine to which index to install the program design block. The application would then need to ask the operating system to run the program design block as there is no way for the program to know ahead of time where the operating system will install the program design block. This is because more than one program at a time could be using the FPGA, so the operating system would be responsible for assigning each FPGA request to a particular index.

Likewise, if a program requests to install a program design block, but there is not enough room for it, the operating system would have to determine if there are any program design blocks that could be unloaded because they are not in use anymore or it would have to tell the requesting program that there is no room on the FPGA and then the application would have to either run the software-only version of this method or it would have to wait for a slot to become available or display an error. The choice of how the application responds is up to the application developer.

The operating system would also have to have some way of being able to allow a program to request a program design block and then the operating system would map that to the index where it installed the specified program design block. So, when the application issues a fpga_call instruction, the program design block identifier it uses would be used by the operating system to map to the actual program design block index. For this paper, the index used is the actual index where the program design block is installed. But this will not be possible in a commercial system since the operating system will dynamically load and unload program design blocks based on the applications that are loaded and their FPGA utilization and there is no way of knowing at build time what the correct index will be. This is something the operating system will have to determine dynamically and map the requests of the applications to the correct program design block index at runtime.

The operating system would possibly need to limit the size of the program design block allowed. The program would need a way to query the operating system to see what the maximum allowed size is for a program design block and then it could choose to use either the hardware version of the method or fallback to a software implementation if provided, or it could limit its functionality or display an error. The way of handling insufficient system resources is up to the application developer.

5.5 Memory Access

The FPGA should have access to the system memory since applications running on the system processor have access to the memory and the FPGA contains functions that are called from programs running on the processor. This would allow a pointer to be passed that points to an array or some other object and the functions implemented on the FPGA would be able to reference it. However, allowing memory access does pose some challenges.

For one thing, it would complicate the caching protocols implemented by the operating system. They would need to be able to handle updates from the FPGA and invalidate the necessary entries.

It would also complicate things if the process and the FPGA were both trying to write to the same memory location. But this is sort of like multiple processor cores trying to write to the same memory location, with the exception that the FPGA is not controlled by the operating system, so it would not know if a write is happening to a memory location from the FPGA.

5.6 Programming Language Integration

The programming samples in this paper are all written in assembly language, and the FPGA instructions would need to be added to the assemblers used for each system, but hybrid support would also require adoption in higher level languages, such as Java and C++, too for it to become more popular. The details of what this language-level integration would look like is beyond the scope of this paper and is up to the language designers, but at a minimum it would need to allow developers to write a function in a hardware description language or to draw a schematic, and then be able to call that function from the rest of the program. This function would be the logic unit in the program design block, as is shown in Figure 13 of section 3.5.

The languages could allow the entity name to be used as a function. Some of the inputs, such as the clock signal and the control signals, such as the fpga_call, fpga_get, fpga_halt, and fpga_status would not need to be passed into the function. These are details that are handled at the hardware level. The control signals would be generated by the control unit and passed to the logic unit. This is not something that should have to be dealt with on the software side. The only inputs the function would need on the software side would be the inputs that are actually needed to perform the logic of the function. For example, Figure 19 in section 4.2 shows the logic unit for the M to N summation program. For this function, only the min_count and max_count parameters would be required. The other input signals should be automatically synthesized and handled at the hardware level as there is no application-specific logic required for determining the value of the control signals.

Upon invoking the compiler, if there are any schematics or hardware description language files in the list of files to compile, then the compiler would also need to call any necessary synthesizers to generate the necessary netlists.

5.7 IDE Support

Although not strictly required, support from popular IDEs, such as IntelliJ IDEA, Eclipse, NetBeans, along with others, would be beneficial for improving the developer experience. They could provide features such as allowing the developer to add a new program design block to the project. When adding the program design block, the developer could have the choice to add a schematic or some hardware description language, such as VHDL, Verilog, or some other options.

6 Works Cited

Altera. "Intel Stratix 10 Product Table." n.d. *Altera.* 3 April 2018. <https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/pt/stratix-10-product-table.pdf>.

Patterson, David A. and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. 5th Edition. Elsevier Science, 2014.

Xilinx. "UltraScale Architecture and Product Data Sheet: Overview." Vers. 3.3. 12 March 2018. *Xilinx*. 3 April 2018. <https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf>.

7 Appendix

7.1 Instruction Format Reference

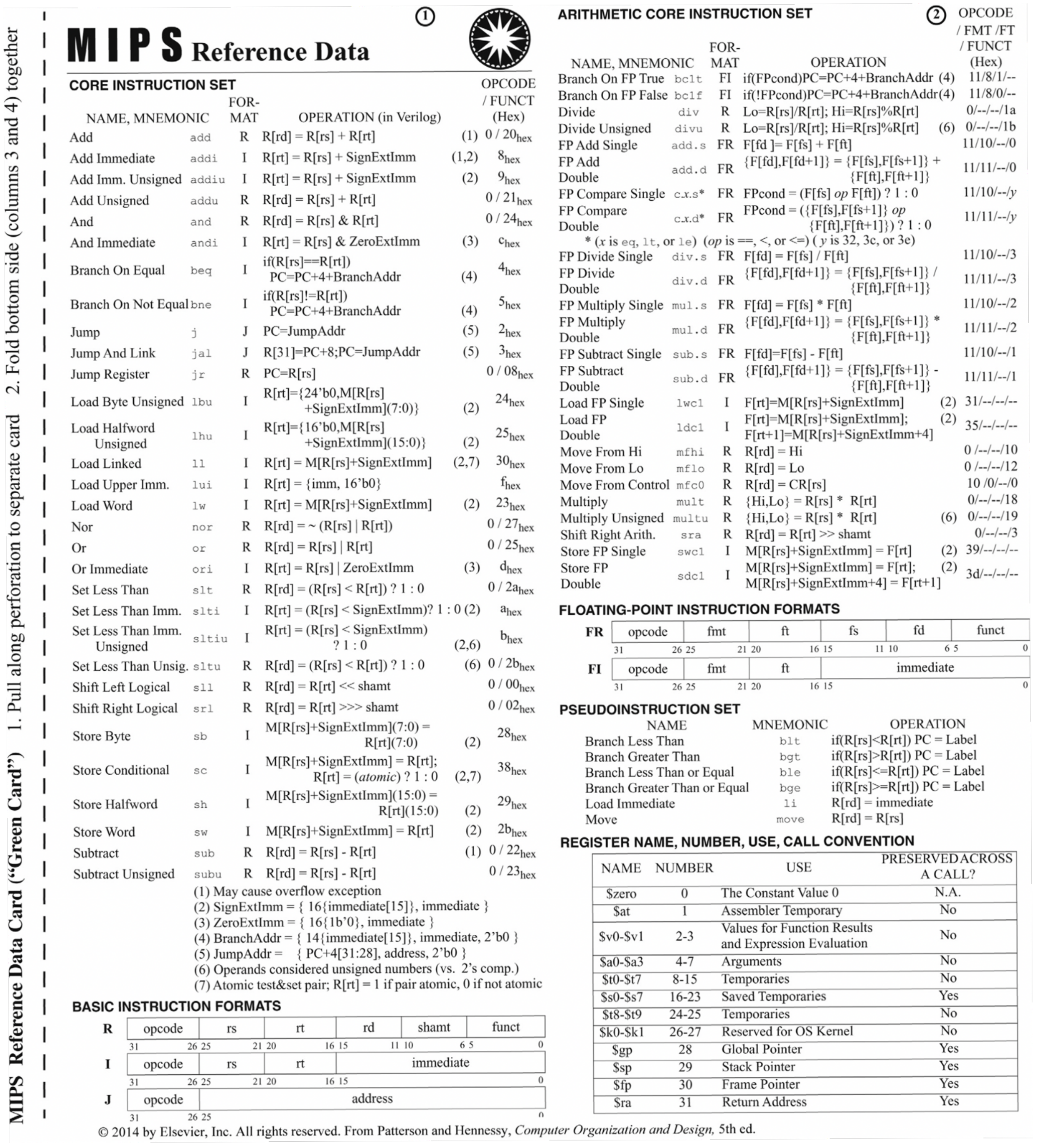
Sample Instruction	Machine Language Representation
`abs $1, $2`	`000000 00010 00000 00001 00000 000100`
`add $1, $2, $3`	`000000 00010 00011 00001 00000 100000`
`addi $1, $2, 5`	`001000 00010 00001 0000000000000101`
`and $1, $2, $3`	`000000 00010 00011 00001 00000 100100`
`andi $1, $2, 7`	`001001 00010 00001 0000000000000111`
`beq $1, $2, 5`	`000100 00001 00010 1111111111111111`
`beqal $1, $2, 7`	`110000 00001 00010 0000000000000000`
`bequ $1, $2, 11`	`001111 00001 00010 0000000000000011`
`bequal $1, $2, 11`	`110110 00001 00010 0000000000000010`
`bge $1, $2, 14`	`001100 00001 00010 0000000000000100`
`bgeal $1, $2, 5`	`110100 00001 00010 1111111111111010`
`bgeu $1, $2, 7`	`010010 00001 00010 1111111111111011`
`bgeual $1, $2, 11`	`111010 00001 00010 1111111111111110`
`bgt $1, $2, 14`	`001110 00001 00010 0000000000000000`
`bgtal $1, $2, 5`	`110010 00001 00010 1111111111110110`
`bgtu $1, $2, 7`	`010100 00001 00010 1111111111110111`
`bgtual $1, $2, 11`	`111000 00001 00010 1111111111111010`
`ble $1, $2, 14`	`001011 00001 00010 1111111111111100`
`bleal $1, $2, 3`	`110011 00001 00010 1111111111110000`
`bleu $1, $2, 5`	`010001 00001 00010 1111111111110001`
`bleual $1, $2, 7`	`111001 00001 00010 1111111111110010`
`blt $1, $2, 11`	`001101 00001 00010 1111111111110101`
`bltal $1, $2, 14`	`110001 00001 00010 1111111111110111`
`bltu $1, $2, 3`	`010011 00001 00010 1111111111101011`
`bltual $1, $2, 5`	`110111 00001 00010 1111111111101100`
`bne $1, $2, 7`	`001010 00001 00010 1111111111101101`
`bneal $1, $2, 11`	`110101 00001 00010 1111111111110000`
`bneu $1, $2, 14`	`010000 00001 00010 1111111111110010`
`bneual $1, $2, 3`	`111011 00001 00010 1111111111100110`
`dequeue $1`	`111110 00001 00000 00000 00000 000001`
`div $1, $2, $3`	`000000 00010 00011 00001 00000 011010`
`divi $1, $2, 5`	`010111 00010 00001 0000000000000101`

Sample Instruction	**Machine Language Representation**
`divu $1, $2, $3`	`000000 00010 00011 00001 00000 011011`
`divui $1, $2, 5`	`011011 00010 00001 0000000000000101`
`enqueue $1`	`111110 00001 00000 00000 00000 000000`
`exit`	`000010 00000000000000000000100100`
`jal 7`	`000011 00000000000000000000000111`
`jmp 5`	`000010 00000000000000000000000101`
`jr $1`	`000000 00001 00000 00000 00000 001000`
`ldb $1, 3`	`111111 00001 0000000011 00000 000000`
`ldby $1, 3`	`111111 00001 0000011 00000000 000100`
`ldhw $1, 3`	`111111 00001 000011 000000000 000110`
`ldw $1, 3`	`111111 00001 00011 0000000000 000001`
`lw $1, 5`	`100011 00000 00001 0000000000000101`
`mod $1, $2, $3`	`000000 00010 00011 00001 00000 011100`
`modi $1, $2, 7`	`011110 00010 00001 0000000000000111`
`modu $1, $2, $3`	`000000 00010 00011 00001 00000 011101`
`modui $1, $2, 7`	`011111 00010 00001 0000000000000111`
`mov $1, $2`	`000000 00010 00000 00001 00000 100000`
`mul $1, $2, $3`	`000000 00010 00011 00001 00000 011000`
`muli $1, $2, 11`	`010110 00010 00001 0000000000001011`
`mulu $1, $2, $3`	`000000 00010 00011 00001 00000 011001`
`mului $1, $2, 11`	`011010 00010 00001 0000000000001011`
`neg $1, $2`	`000000 00010 00000 00001 00000 000111`
`nop`	`00000000000000000000000000000000`
`nor $1, $2, $3`	`000000 00010 00011 00001 00000 100111`
`nori $1, $2, 14`	`011000 00010 00001 0000000000001110`
`not $1, $2`	`000000 00010 00000 00001 00000 000011`
`or $1, $2, $3`	`000000 00010 00011 00001 00000 100101`
`ori $1, $2, 3`	`010101 00010 00001 0000000000000011`
`pop $1`	`111101 00001 00000 00000 00000 000001`
`push $1`	`111101 00001 00000 00000 00000 000000`
`queue_clear`	`111110 00000 00000 00000 00000 000010`
`queue_size $1`	`111110 00001 00000 00000 00000 000011`
`ret`	`000000 11111 00000 00000 00000 001000`
`rol $1, $2, $3`	`000000 00010 00011 00001 00000 011111`
`roli $1, $2, 5`	`000001 00010 00001 0000000000000101`

Sample Instruction	Machine Language Representation
ror $1, $2, $3	000000 00010 00011 00001 00000 011110
rori $1, $2, 5	100010 00010 00001 0000000000000101
seq $1, $2, $3	000000 00010 00011 00001 00000 110100
sequ $1, $2, $3	000000 00010 00011 00001 00000 110101
sge $1, $2, $3	000000 00010 00011 00001 00000 101110
sgeu $1, $2, $3	000000 00010 00011 00001 00000 101111
sgt $1, $2, $3	000000 00010 00011 00001 00000 110000
sgtu $1, $2, $3	000000 00010 00011 00001 00000 110001
sla $1, $2, $3	000000 00010 00011 00001 00000 000110
slai $1, $2, 7	100001 00010 00001 0000000000000111
sle $1, $2, $3	000000 00010 00011 00001 00000 101100
sleu $1, $2, $3	000000 00010 00011 00001 00000 101101
sll $1, $2, $3	000000 00010 00011 00001 00000 000000
slli $1, $2, 7	011101 00010 00001 0000000000000111
slt $1, $2, $3	000000 00010 00011 00001 00000 101010
sltu $1, $2, $3	000000 00010 00011 00001 00000 101011
sne $1, $2, $3	000000 00010 00011 00001 00000 110010
sneu $1, $2, $3	000000 00010 00011 00001 00000 110011
sra $1, $2, $3	000000 00010 00011 00001 00000 000001
srai $1, $2, 11	100000 00010 00001 0000000000001011
srl $1, $2, $3	000000 00010 00011 00001 00000 000010
srli $1, $2, 14	011100 00010 00001 0000000000001110
stack_clear	111101 00000 00000 00000 00000 000010
stack_size $1	111101 00001 00000 00000 00000 000011
stb $1, 3	111111 00001 0000000011 00000 000010
stby $1, 3	111111 00001 0000011 00000000 000101
sthw $1, 3	111111 00001 000011 000000000 000111
stw $1, 3	111111 00001 00011 0000000000 000011
sub $1, $2, $3	000000 00010 00011 00001 00000 100010
sw $1, 5	101011 00000 00001 0000000000000101
xor $1, $2, $3	000000 00010 00011 00001 00000 000101
xori $1, $2, 7	011001 00010 00001 0000000000000111

Table 1 - Instruction Format

7.2 MIPS Instruction Set (Patterson and Hennessy)



MIPS Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

MIPS Reference Data ①

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	`add`	R	R[rd] = R[rs] + R[rt]	(1)	$0 / 20_{hex}$
Add Immediate	`addi`	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	`addiu`	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	`addu`	R	R[rd] = R[rs] + R[rt]		$0 / 21_{hex}$
And	`and`	R	R[rd] = R[rs] & R[rt]		$0 / 24_{hex}$
And Immediate	`andi`	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	`beq`	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	`bne`	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	`j`	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	`jal`	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	`jr`	R	PC=R[rs]		$0 / 08_{hex}$
Load Byte Unsigned	`lbu`	I	R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	`lhu`	I	R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	`ll`	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	`lui`	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	`lw`	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	`nor`	R	R[rd] = ~ (R[rs] | R[rt])		$0 / 27_{hex}$
Or	`or`	R	R[rd] = R[rs] | R[rt]		$0 / 25_{hex}$
Or Immediate	`ori`	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	`slt`	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		$0 / 2a_{hex}$
Set Less Than Imm.	`slti`	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	`sltiu`	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	`sltu`	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	$0 / 2b_{hex}$
Shift Left Logical	`sll`	R	R[rd] = R[rt] << shamt		$0 / 00_{hex}$
Shift Right Logical	`srl`	R	R[rd] = R[rt] >>> shamt		$0 / 02_{hex}$
Store Byte	`sb`	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	`sc`	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	`sh`	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	`sw`	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	`sub`	R	R[rd] = R[rs] - R[rt]	(1)	$0 / 22_{hex}$
Subtract Unsigned	`subu`	R	R[rd] = R[rs] - R[rt]		$0 / 23_{hex}$

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

BASIC INSTRUCTION FORMATS

R	opcode	rs	rt	rd	shamt	funct
	31 26	25 21	20 16	15 11	10 6	5 0
I	opcode	rs	rt	immediate		
	31 26	25 21	20 16	15 0		
J	opcode	address				
	31 26	25 0				

ARITHMETIC CORE INSTRUCTION SET ②

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	`bc1t`	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	`bc1f`	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	`div`	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	`divu`	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	`add.s`	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	`add.d`	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	`c.`*x*`.s`*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	`c.`*x*`.d`*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
* (*x* is `eq`, `lt`, or `le`) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)					
FP Divide Single	`div.s`	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	`div.d`	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	`mul.s`	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	`mul.d`	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	`sub.s`	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	`sub.d`	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	`lwc1`	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	`ldc1`	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	`mfhi`	R	R[rd] = Hi		0 /--/--/10
Move From Lo	`mflo`	R	R[rd] = Lo		0 /--/--/12
Move From Control	`mfc0`	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	`mult`	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	`multu`	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	`sra`	R	R[rd] = R[rt] >> shamt		0/--/--/3
Store FP Single	`swc1`	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	`sdc1`	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS

FR	opcode	fmt	ft	fs	fd	funct
	31 26	25 21	20 16	15 11	10 6	5 0
FI	opcode	fmt	ft	immediate		
	31 26	25 21	20 16	15 0		

PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	`blt`	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	`bgt`	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	`ble`	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	`bge`	if(R[rs]>=R[rt]) PC = Label
Load Immediate	`li`	R[rd] = immediate
Move	`move`	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

Figure 28 - MIPS Instruction Set (Patterson and Hennessy)