Rethinking CPU-FPGA Interfaces: A Reconfigurable RISC-V Co-Processor

Colin Drewes
University of California, San Diego
cdrewes@ucsd.edu

ABSTRACT

Field Programmable Gate Arrays (FPGAs) are a powerful general purpose accelerator, particularly for highly parallizable applications. Despite their versatility, their usefulness is marred by slow development cycles and complex development tools. As a result, CPU and FPGA hybrid systems to leverage mass parallelism with simplified procedural control have become commonplace—appearing on the same die as with a System on a Chip (SoC), or connected as a separate PCIe interface to a motherboard. Regardless of implementation, the FPGA is treated as a separate entity from the CPU, and interaction between the two is limited to the established protocol of the system. Though this model has taken steps towards treating an FPGA as a co-processor instead of a peripheral, it lacks the efficient register-level computation of traditional CPU instructions. The inability for the FPGA to compute directly on shared memory *as well* as CPU registers is a major limitation of such hybrid devices.

In this paper we will present the design of a CPU utilizing an FPGA as co-processor with physical access to the registers as well as memory of the processor. This extension will mirror that of co-processors in the RISC-V ecosystem. Such a device will allow a user to specify a single assembly instruction to program the device, and a second instruction to trigger the execution of the device. This architecture will have the following improvements over standard CPUs and existing CPU-FPGA hybrids: 1) ability to adjust at the hardware level to changing workloads at runtime, 2) less wasted silicon on specialized ASIC co-processors (crypto, vector operations...), 3) improved power efficiency, 4) and simplified compatibility with other ISA extensions and deprecated instructions.

1. INTRODUCTION

As consumers continue to chase faster computational speeds in a post-Moore's law era, a shift away from the historical dependence on CPUs must occur. The largest datacenter CPU manufacturers in the world, Intel and AMD, have acted on this, and begun diversifying their offerings particularly in the FPGA space—with Intel's acquisition of Altera in 2015 and AMD's acquisition of Xilinx in 2020. With the largest FPGA companies in the hands of the largest CPU companies, it is not unreasonable to expect that we will soon see a new generation of CPU-FPGA hybrid architectures aimed at addressing the limitations of traditional microprocessors.

FPGAs have taken center stage as a chosen accelerator due to their demonstrated success for many applications: including networking, neural networks, and cryptography. Despite great efforts however, the opportunity cost of leveraging these devices—as a result of challenging developer tools and slow design iterations—remains insurmountable for most. To ease the development challenges, FPGAs have been more closely integrated with CPUs, on the same die as with Systems on a Chip like the Zynq-7000[1], or more decoupled over a PCIe interface as with Amazon Web Services FPGA instances [2]. A data transaction from CPU→FPGAs must occur before computation, then a transaction from FPGAs→CPU to return the computed information.

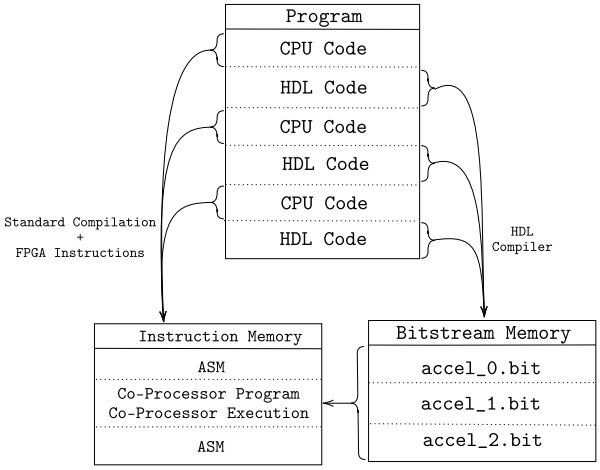


Figure 1: High level combined software-hardware flow for a CPU with a FPGA co-processor. A standard program contains code targeted for the CPU (CPU Code), and some Hardware Description Language (HDL Code) kernel accelerators targeted for the FPGA co-processor. Each HDL Code block accelerator will be compiled to a bitstream and stored in a specialized `Bitstream Memory` which can be quickly accessed by the CPU to reconfigure the co-processor. Standard CPU code is compiled to assembly and uses two specialized instructions that we propose in this paper to configure the co-processor

In this paper we will present the design of an alternative CPU-FPGA architecture. Instead of treating the FPGA as a separate entity and limiting interaction to a predefined protocol (like PCIe), the device will be physically connected to the CPU's register file and memory as a co-processor. The device can be interfaced on the instruction set level, allowing

for the rapid execution and reconfiguration of the FPGA in a single instruction. This will allow accelerator designs to be embedded directly into standard procedural code, compiled in advance, then activated at runtime as shown in Figure 1. We argue that such a CPU architecture will have the following advantages:

1. This architecture has the ability to rapidly reconfigure the FPGA co-processor (effectively adding an instruction to the ISA) during the execution of the program to adjust to changing workloads on the CPU
2. Less dedicated silicon for intermittently used co-processors is needed as all of this functionality can be reproduced with the FPGA co-processor—potentially reducing the dark silicon of the system
3. In reducing the amount of excess silicon for dedicated co-processors, the overall power consumption of the system is reduced, leading to a more efficient performance per watt device
4. Backwards compatibility to older iterations of an ISA without explicit silicon support—instructions simply become configurations of the FPGA co-processor

The rest of this paper will be organized as follows. Section 2 discusses the current interest in embedding an FPGA as a co-processor accessible via assembly instructions as well as the proposed architecture used in this paper. Section 3 discusses the current state of ASIC co-processors, particularly for the RISC-V ecosystem, and discusses how the FPGA co-processor fits into this. The architecture of the proposed RISC-V instructions are presented in Section 4, with compiler support for these in Section 5, and a hardware interface as defined in Section 6. Then, in Section 7, we present an implementation of this architecture purely on an FPGA, leveraging a soft-processor and Xilinx partial reconfiguration. Finally, we present the limitations in Section 9 and conclude in Section 10.

2. RELATED WORK

A recently published patent application [3] by AMD is the most similar to the work being discussed in this paper. The primary goal of this patent—embed a reconfigurable co-processor directly into the CPUs pipeline—is the very same as the goals we have set out in this work. Instead of the catch-all language intended to cast the largest net over intellectual property however, our paper will provide a hypothetical physical implementation of such a device and and example use case.

3. RISC-V CO-PROCESSORS

The RISC-V instruction set is built on a base ISA and a composition of useful extensions. An extension is in effect a co-processor—a tightly integrated piece of silicon accessible to the CPU itself. These extensions range in complexity, from simple arithmetic additions (multiplication/division, floating point, etc) to more complicated tasks like vector operations and cryptography[4].

This architecture allows chip designs to implementat an ISA catered to their specific needs while remaining within the umbrella of RISC-V. These extensions have grown in complexity, the most recent if which, cryptography primitives for AES, SM4 and others, was recently ratified. However, as these continue growing in complexity the total amount of silicon in the chip will increase with the utilization of that silicon at any given point decreasing.

The FPGA co-processor will take the place of the majority of these extensions through reconfiguration. Instead of requiring custom silicon to implement new instructions in a RISC-V core, only a bitstream to program the FPGA is required under the system we propose. This offers similar functionality to that of the RISC-V extension ecosystem with a moderate reduction in speed compared to ASICs, but increases the re-usability of the co-processor silicon.

4. INSTRUCTION SET SUPPORT

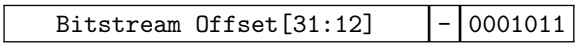
We propose the addition of two instructions to the RISC-V ISA: programming and execution. The program instruction, discussed in Section 4.1, allows the programmer/compiler to reconfigure the state of the FPGA with a new bitstream. The execute instruction, discussed in Section 4.2, provides the FPGA with register values and triggers its execution. After the execute instruction, the FPGA has modified the memory state, or computed a value which is sent to the destination register of the original instruction. We examine a 32 bit version of the RISC-V ISA, but the same principles are applicable to any length. The details of this implementation follow:

4.1 Program Instruction

The program instruction merely needs to include the memory address of the bitstrem in memory which needs to be loaded on the FPGA. This will be implemented as a U-type instruction, which is takes the following form:

The immediate value, bits `[31:12]`, will encode the address of a bitstream stored in memory which the CPU can rapidly access. It is essential that the bitstream be stored in local fast memory, as a primary goal is to reduce reprogramming overhead to the point that the FPGA can rapidly switch tasks at runtime. This is an architectural decision that is discussed in more detail later, but for now we assume some region of memory at a particular offset—which contains all of the FPGA re-configurations used in the execution of the program—is addressed by the `imm[31:12]` value. The `rd[11:7]` value serves no purpose in our implementation, but could hypothetically used to distinguish between multiple FPGA co-processors if such a design appeared beneficial.

RISC-V has built in a set of custom op-codes to the ISA which serve no immediate purpose and are intended for experimental extension of the standard ISA. According to the RISC-V Foundation specifications [4], these instructions, *custom-0* and *custom-1*, have a predefined opcodes of `0001011` and `0101011`. We will use *custom-0* for this instruction. Finally, to program the FPGA co-processor the instruction takes the following form (for brevity unused instruction space is rendered as "-"):



Once this instruction is issued the CPU may choose to stall, if a simplistic in-order pipeline, or evaluate other instructions while the FPGA is reconfigured. These all depend on the underlying implementation of the CPU as well as the reconfiguration speed of the FPGA which are addressed later. Our implementation, which will be a simple single cycle in-order core will stall till the part is complete with its reconfiguration.

4.2 Execute Instruction

Once the re-configurable co-processor has been programmed in 4.1, it can be executed at any point with a second specialized instruction. This instruction will contain two source registers and a single destination register. The source registers may contain either raw values, or an address within memory where the FPGA should perform some reduction. Similarly, after execution the destination register will contain the result of the computation or nothing if the goal of the FPGA was purely to manipulate data. The execute instruction will be in RISC-V R-type format. This is the structure of a standard r-type instruction, which takes the following form:

funct7	rs2	rs1	funct3	rd	op

The register values of rs2 and rs1 will be forwarded to the FPGA co-processor. If applicable, any result will then be stored in the register rd. Similar to the previous instruction, we will use the *custom-1* opcode, which has a value of `0101011`. As a result the final instruction will appear as follows:

-	rs2[24:20]	rs1[19:15]	-	rd[11:5]	0101011

Similar to the program instruction the CPU may choose to stall as this execution is completed, or perform other computation if the FPGA's computation is taking many CPU cycles. This is implementation dependent, but we will examine later in this paper the case of CPU stalling to await the result of the computation.

While the *custom-0* and *custom-1* instructions are protected in the RISC-V ISA, their use does not have native support (meaning you may no longer expect "custom0" to be accepted by the assembler). As a result we examine the common technique for encoding custom instructions for the **RISC-V gcc** compiler in the following section.

5. COMPILER SUPPORT

The `.insn` assembly instruction allows for a programmer to leverage the numeric representation of instructions for the assembler to interpret. This allows you to specify a chosen RISC-V instruction format (R, I, U...etc), and provide the arguments for the assembler to insert into that chosen format. We can implement our two instructions with this method. The `.insn` format for U-type instructions is as follows:

```
.insn u opcode, rd, simm20
```

For example, if given that a bitstream is located at some offset into the bitstream address space of `0x00100`, we may construct the following Program Instruction to program the device with this chosen design:

```
.insn u 0x0B, x0, 0x00100
```

This declares a U-type instruction with the opcode 0x0B (00001011). This is the opcode encoding for the program instruction, but with an extra 0 on the left hand side so it may be encoded as two hexadecimal values which will be truncated off in the 7 bit opcode. The U-type instruction requires a register which serves no purpose in the program step so we just use the hard-0 register `x0`.

We use a similar technique when we wish to execute the co-processor. The `.insn` format of a R-type instruction is given by:

```
.insn r opcode, func3, func7, rd, rs1, rs2
```

For example, if we wished to execute the FPGA with a two input registers, `a1` and `a2`, and store the any output in the destination register `a0`, we could write the following instruction.

```
.insn r 0x2B, 0, 0, a0, a1, a2
```

In this case the opcode is given by `0101011` which we encode with `0x2B`. Again, this is one bit longer than the 7 bit opcode and as a result the upper bit is truncated. The values of `func3` and `func6` are unimportant for the co-processor, and so they are zeroed.

This implementation allows us to write assembly code that will directly utilize the extensions of our CPU architecture. It will also allow us to embed FPGA control directly in our C code with simplistic access (through the use of `__asm__()`).

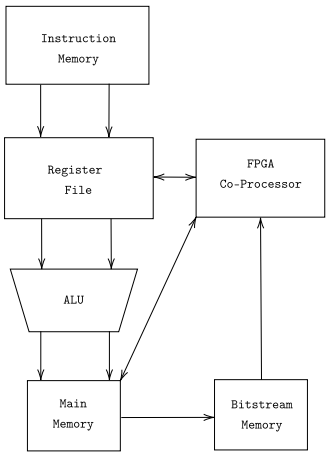


Figure 2: The FPGA co-processor will sit adjacent to the register decoding step of a simple RISC pipeline. When a specialized instruction enters the pipeline, the decoded register values as well as the original instruction will be forwarded to the co-processor. Depending on the instruction, a design will be loaded onto the FPGA from the `Bitstream Memory`, or the current design loaded on the FPGA will be triggered with the input operands of the instruction.

6. HARDWARE SUPPORT

We will examine amending a simple pipelined in-order RISC-V processor with the FPGA co-processor. The high-level architecture is presented in Figure 2. There are two additions to the simple RISC pipeline: the FPGA Co-Processor, and the bitstream memory. The FPGA Co-Processor is the reconfigurable fabric which can be programmed from bitstream designs located in the Bitstream Memory. The `Bitstream Memory` in Figure 2 stores all of the configurations of the co-processor. This memory must be fast, and local to the FPGA itself to accomplish efficient swapping of co-processor designs. This specialized memory has access to the main CPU memory in order for the bitstreams for a given application to be loaded into the high speed `Bitstream Memory`. Once a set of design bitstreams have been loaded into the reconfiguration memory, the FPGA Co-Processor can be quickly programmed with these designs. The two instructions we present (Program and Execute), will be handled differently when forwarded to the co-processor. Both cases will be handled through the following interface on the CPU:

```
output [31:0] cproc_insn,
output [31:0] cproc_rs1,
output [31:0] cproc_rs2,
output        cproc_valid,
input         cproc_ready,
input         cproc_wait,
input  [31:0] cproc_rd,
input         cproc_wr,
```

6.1 Program Instruction

In this instance, the `cproc_ready` signal will be held at zero if the FPGA is left in an un-configured state. This will prevent the CPU from erroneously triggering the co-processor and expecting a result when it will never complete. The `cproc_wait` is held at zero for the extent of the FPGA reconfiguration. This signal will allow the CPU to stall as the FPGA is reconfigured, and know when the `Program Instruction` has completed. Alongside this, the instruction itself is sent to the co-processor through the `cproc_insn` bus. This allows the co-processor to access immediate values, particularly the `simm2` value of the U-type instruction which we use to encode the address s of a bitstream to be loaded. The co-processor will be initialized, and the instruction forwarded, only when the `cproc_valid` signal is asserted.

6.2 Execute Instruction

Similar to the previous instruction, the `cproc_ready` signal will be held at zero if the FPGA is left in an un-configured state. Once configured, the core is ready to activate so `cproc_ready` is asserted. When the `cproc_valid` signal is asserted, and the FPGA activated, the `cproc_rs1`, `cproc_rs2`, and `cproc_insn` values are forwarded to the co-processor. The `cproc_rs1`, `cproc_rs2` are the two input operands of the standard R-type instruction, which have been decoded from the `Register File`. As the FPGA computes, the `cproc_wait` signal is asserted to communicate back to the CPU that the instruction is active and awaiting completion. After the completion of the FPGA, if the input instruction was a standard R-type which has a return argument to a register, that value can be set through the `cproc_rd` bus and written back to the `Register File` with the assertion of `cproc_wr`.

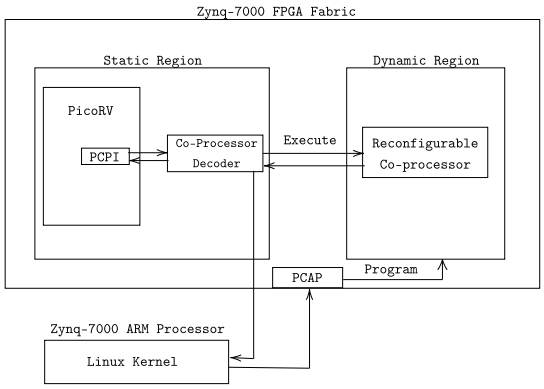


Figure 3: A purely soft-implementation of a RISC-V CPU (PicoRV) with a reconfigurable co-processor interface. The PicoRV sits inside a statically defined region of the FPGA. The core implements the Pico Co-Processor Interface (PCPI), which enables communication from the CPU pipeline to an accelerator. The accelerator in this design is the co-processor sitting inside the partially reconfigurable region of the FPGA. This region can be modified at any point from the ARM processor, through the Processor Configuration Access Port (PCAP).

7. A PURE FPGA IMPLEMENTATION

While it remains the end goal to have an hard CPU with an embedded FPGA co-processor, the realities of an individual creating such an integrated design are prohibitive. As a result, it is the goal of this paper to provide a simplified physical implementation of the proposed architecture to begin prototyping workloads that can benefit from this reconfigurable co-processor.

We will offer the design of a pure-FPGA implementation. This means that both the CPU will be implemented on the reconfigurable fabric—as a soft-processor—and the FPGA co-processor will be integrated on the same fabric with partial reconfiguration. Partial reconfiguration is a tool offered on particular Xilinx parts that allows applications to reserve a region of the FPGA fabric to be dynamic, meaning it can be updated at runtime [5]. This dynamic region will become the co-processor, while the static region will hold the CPU and logic for reprogramming the dynamic region.

Figure 3 presents the high level architecture of this design. We will utilize a Zynq-7000 FPGA for this implementation, but any part with dynamic reconfiguration capabilities will suffice. Both the static region and the dynamic region are present on the same FPGA fabric. The static region holds the CPU which is the `PicoRV`, a 32 bit RISC-V microprocessor targeted for FPGA platforms as discussed in Section 7.2. Internal to the `PicoRV` is the `Pico Co-Processor Interface (PCPI)`, which allows us to interpret custom instructions and dispatch those to the reconfigurable co-processor logic—discussed in Section 7.3. Once the PCPI forwards the instruction and relevant register data to the `Co-processor Decoder`, as discussed in Section 4, there are two possible

results: 1) the instruction calls for the activation of the `Reconfigurable Co-Processor`, at which point the logic in the dynamic region is activated or 2) the instruction specifies the address of the new partial bitstream (FPGA configuration file) to reprogram the co-processor which is handled entirely on the FPGA with the `Peripheral Configuration Access Port (PCAP)` as discussed in Section 7.4. We now discuss these structures in greater depth, as well as the method in which we interact with this system.

7.1 RISC-V-On-PYNQ

We leverage a tool, RISC-V-On-PYNQ [6], designed for interacting with soft-processors on an FPGA system, particularly for the Pynq-Z2 platform. Pynq-Z2 is a development board which implements the Zynq-7000 part and provides the infrastructure to run Linux on the associated ARM core. This means the FPGA can be programmed from the on-board ARM chip, streamlining the testing of designs. The Pynq-Z2 also offers a Python interface for programming and interacting with data to and from the FPGA. The RISC-V-On-PYNQ package takes this a step further to allow us to program and execute RISC-V processors running on the FPGA fabric through a Python interface. This will be used to prime the the PicoRV with a program, and write the partial reconfiguration bitstreams to memory so that they may be access from within the FPGA fabric for reconfiguring the co-processor.

7.2 PicoRV Processor

The PicoRV CPU is a 32 bit microprocessor designed to be used on FPGA systems [7]. It is configured to implement the RISC-V RV32IM instruction set. This implementation of the PicoRV in particular is configured with a hybrid 128KB of BRAM memory space and 256MB of DDR memory space. The advantage of the BRAM memory is its speed, due to it being embedded directly into the FPGA fabric—though this locality comes at the cost of size. This memory space is primarily used as instruction memory. The DDR memory space, as it is connected to the processing system, is ideal for exchanging data to and from the ARM CPU of the Zynq chip. We will be able to map our co-processors directly into the DDR memory space, allowing for in-memory reductions. Our core is operation at 50MHz, a fairly conservative speed for this core (rated for $F_{max} = 250$MHz).

7.3 Pico Co-Processor Interface (PCPI)

Besides the simplicity of the PicoRV, it also has the benefit of being equipped with an existing co-processor interface—the Pico Co-Processor Interface (PCPI). When an unsupported instruction is encountered in the PicoRV pipeline, the following information is forwarded out of the PCPI:

```
output        pcpi_valid
output [31:0] pcpi_insn
output [31:0] pcpi_rs1
output [31:0] pcpi_rs2
```

The `pcpi_valid` signal indicates the co-processor should be activated with a given instruction `pcpi_insn`, and decoded register values `pcpi_rs1` and `pcpi_rs2`. When the co-processor has completed the following interface is used for returning results to the PicoRV:

```
input         pcpi_wr
input  [31:0] pcpi_rd
input         pcpi_wait
input         pcpi_ready
```

The assertion of `pcpi_wr` will enable register write back, which will store `pcpi_rd` into the destination register of the original instruction. Activating `pcpi_wait` will signal to the PicoRV that the instruction was correctly received and is being processed as weall as causing the pipeline to stall, and importantly that the instruction is legal and should not cause a hardware trap. Once execution has completed, the `pcpi_ready` signal communicates to the PicoRV pipeline the completion of the co-processor execution and the pipeline is unstalled.

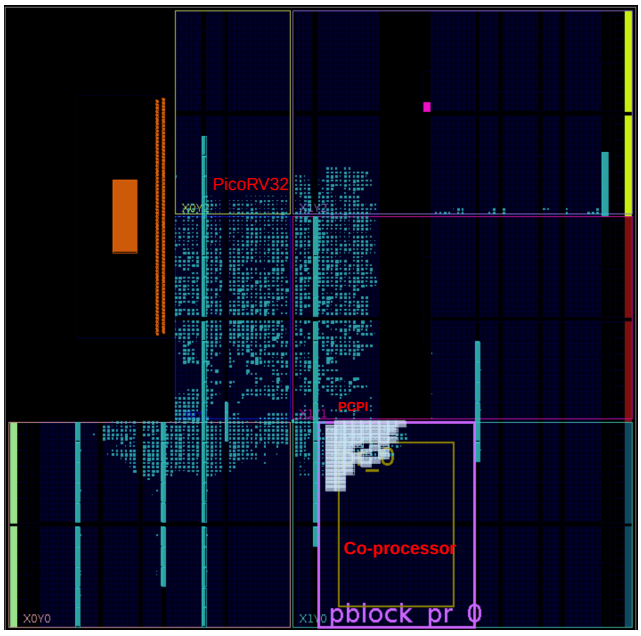


Figure 4: The implemented PicoRV + FPGA Co-Processor design on the Zynq-7000. The `pblock_pr_0` region defines the reconfigurable region of the fabric which is used for the co-processors. The PCPI, and a set of muxes (in grey) used for assuring the state during reconfiguration, bridge the gap between processor pipeline and the accelerator.

7.4 Partial Reconfiguration

Partial reconfiguration is an “expert” (read: poorly documented) flow within the Vivado design tools [5] that allows a full bitstream to have "partial" regions of its design reconfigured with "partial bitstreams." In our design, the full bitstream contains the PicoRV, and the partial bitstream contains the co-processor. In Figure 4 the partial region is contained in a "p-block" labeled `pblock_pr_0`. The static region is everything outside of that "p-block."

There are two primary methods for partial reconfiguration on the Zynq-7000 part: Internal Configuration Access Port (ICAP), and Processor Configuration Access Port (PCAP) [5]. We will examine the timing overhead of each of these techniques, as quick reconfiguration is essential, and base on

implementation on that.

7.4.1 *Internal Configuration Access Port*

The ICAP interface allows an internal design of the FPGA to reconfigure a region of the FPGA [8]. The advantage of this method is everything is internal to the FPGA, and there is no reliance on the potentially slower ARM processor for reconfiguraion. According to Xilinx documentation [9], the ICAP is capable of reconfiguring a 134,392 Byte partial bitstream in 1,060μs. Our partial bitstreams are 598K in size, which is ≈ 4.5 times as large. According to [9] this reconfiguaration time scales roughly linearly, and so we can expect that our regionfigurable region will take $1,060\mu s * 4.5 = 4770\mu s$. As our processor is running at 50MHz, a reconfiguration will take $\frac{4770\mu s}{\frac{1}{50MHz}} = 238500$ cycles. In order words, under these estimates our processor will have to stall for 238500 cycles to reconfigure the co-processor. This is an unrealistic amount of time to stall the CPU pipeline, as the vast majority of computations would have been completed using standard instructions in the amount of time it takes to reconfigure the co-processor.

7.4.2 *Partial Configuration Access Port*

The PCAP interface allows the ARM processor, rather than the FPGA fabric itself, to reconfigure a region of the FPGA [5]. According to [9], reconfiguration from the PCAP through the Linux interface will take 2,000μs. Using the method as Section 7.4.1: $\frac{9000\mu s}{\frac{1}{50MHz}} = 450000$ cycles. This is worse than the ICAP interface, but both are infeasible to use at runtime. For simplicity, we will use the PCAP interface, as it is integrated with the PYNQ interface which adds great ease in reconfiguring the co-processor. However, this means we will not be capable of reconfiguration within a single program, which is unfortunate, but the partial reconfiguration speeds on this device are prohibitive. Reconfiguration of the co-processor remains useful though as co-processor demands vary between a user's application and the co-processor could be configured in advance depending on the demands of a single application. We will move forward with this simplification, but it remains the original goal to have rapid on-the-fly reconfiguration as presented in Section 1.

8. SOFTWARE + HARDWARE DEMO

We now leverage our design in 7 to accelerate a simplistic and common CPU task: computing the popcount, or Hamming distance, of an unsigned integer. The goal here is to compute the number of 1s in a binary representation of a number. This has particular value in cryptography and information theory. We will implement this both with and without co-processor acceleration, and study the difference. First, we discuss the method for timing code execution on the PicoRV core.

8.1 RISC-V `rdcycle` Instruction

The RV32IM instruction set the PicoRV implements has built in support for getting the internal cycle cout of the processor. This value can be accessed with the `rdcycle x0` instruction, where `x0` is the destination register for the current cycle count. The instruction can be used as above in raw assembly, but also embedded in C code and read into a C variable as follows:

```
__asm__("rdcycle a0");
register unsigned int start __asm__("a0");
```

This allows us to quickly time applications in C.

8.2 Standard Implementation

```
%%riscvc pop_c overlay.processor

unsigned int pc(unsigned x, unsigned y) {
    int c = 0;
    for(; x!=0;x>>=1)
        if(x&1)
            c++;

    int q = 0;
    for(; y!=0;y>>=1)
        if(y&1)
        q++;
    return c+q;
}

int main(int argc, char ** argv) {
    unsigned int x = 0xFFFFFFFF;
    unsigned int y = 0xFFFFFFFF;
    __asm__("rdcycle a0");
    register unsigned int start __asm__("a0");
    unsigned int pop = pc(x,y);
    __asm__("rdcycle a1");
    register unsigned int end __asm__("a1");
    return end-start;
}
```

This application times the execution of computing the popcount of two unsigned integers. After running this on the PicoRV, we get a return value of 6570 meaning this program takes 6570 cycles to complete. For comparison, we hardware accelerate this using the FPGA co-processor.

8.3 Co-Processor Implementation

We first need to build the hardware accelerator, which we can do simply in verilog. This core accepts two 32 bit inputs, which are the unsigned integers we wish to compute the popcount of. After computing the popcount, we forward the result back, and assert the done signal. This core is compiled into a partial bitstream, which can be used to reconfigure the co-processor.

```
module cp_popount(
    input clk,
    input[31:0] pcpi_rs1,
    input[31:0] pcpi_rs2,
    output reg[31:0] pcpi_rd = 0,
    input trigger,
    output reg done = 0
);
integer i;
always @(posedge clk) begin
    if (trigger) begin
        for(i=0;i<32;i=i+1)
            if(pcpi_rs1[i] == 1'b1)
                pcpi_rd = pcpi_rd + 1;
        for(i=0;i<32;i=i+1)
            if(pcpi_rs2[i] == 1'b1)
                pcpi_rd = pcpi_rd + 1;
```

```
        done <= 1;
    end else begin
        done <= 0;
    end
end
endmodule
```

We then can leverage this accelerator from within our C program. To do this, we write a simple function, pc, which executes the co-processor instruction and then returns the value. From within our main method, we call this function and time its execution.

```
%%riscvc pop_c overlay.processor

extern unsigned int pc(unsigned x, unsigned y);
__asm__("pc:\n \
        \t .insn r 0x2B, 0, 0, a0, a0, a1\n \
        \t ret \
        ");
int main(int argc, char ** argv) {
    unsigned int x = 0xFFFFFFFF;
    unsigned int y = 0xFFFFFFFF;
    __asm__("rdcycle a0");
    register unsigned int start __asm__("a0");
    unsigned int pop = pc(x,y);
    __asm__("rdcycle a1");
    register unsigned int end __asm__("a1");
    return end-start;
}
```

This program takes 547 cycles to complete in comparison to the 6570 cycles of the unaccelerated version. Our accelerator results in a 12X speedup over a standard implementation.

9. LIMITATIONS

9.1 Harware Limitations

This architecture is primarily motivated by the ideal of more efficient silicon use: both in space and power. We make only an informal argument in this paper that by compressing potentially many ASIC co-processors into a single FPGA will reduce silicon usage and power consumption—this needs much deeper analysis. Partial reconfiguration also proofed to be far to slow at least within the Zynq-7000 to reach the desired runtime reconfiguration of the co-processor.

9.2 Software Limitations

The goal of this project is to enable software developers to create micro-kernel hardware accelerators for their applications. This has no future with the current state of Xilinx toolchains, which are unintuitive with poorly documented features. High Level Synthesis tools begin to solve this problem, but they still rely on a significant amount of knowledge about underlying hardware to effectively utilized.

10. CONCLUSION

In this paper we have presented the architecture of a reconfigurable RISC-V co-processor. Utilizing an FPGA embedded directly into a CPU's pipeline, we can dynamically adjust to the needs of a given program, and maybe even eventually the changing needs of a single program. Through the tight coupling of CPU pipeline and FPGA, we can design specialized instructions for interfacing the co-processor at the ISA level, which adds speed and simplicity. We present a hypothetical "soft" implementation entirely on an FPGA leveraging the PicoRV processors and partial reconfiguration. A minimal example is presented showing a simple task, the popcount of an integer, can be accelerated by 12X through minimal hardware development effort.

11. REFERENCES

[1] P. L. PL, "Zynq-7000 all programmable soc overview," 2012.

[2] J. Shan, M. R. Casu, J. Cortadella, L. Lavagno, and M. T. Lazarescu, "Exact and heuristic allocation of multi-kernel applications to multi-fpga platforms," in *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, 2019.

[3] A. G. Kegel, "Method and apparatus for efficient programmable instructions in computer systems," Dec. 31 2020. US Patent App. 16/451,804.

[4] R.-V. Foundation, "The risc-v instruction set manual, volume i: User-level isa," 2017. Document Version 2.2.

[5] Xilinx, "Partial reconfiguration," 2018.

[6] D. Richmond, M. Barrow, and R. Kastner, "Everyone's a critic: A tool for exploring risc-v projects," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 260–2604, IEEE, 2018.

[7] C. Wolf, "Picorv32 - a size-optimized risc-v cpu." `https://github.com/cliffordwolf/picorv32`, n.d.

[8] Xilinx, "Axi hwicap v3.0," 2016.

[9] C. Kohn, "Partial reconfiguration of a hardware accelerator on zynq-7000 all programmable soc devices," 2013.