Single Cycle RISC-V Micro Architecture Processor and its FPGA Prototype

Don Kurian Dennis, Ayushi Priyam, Sukhpreet Singh Virk, Sajal Agrawal, Tanuj Sharma,
Arijit Mondal and Kailash Chandra Ray
Indian Institute of Technology Patna
{donkdennis, ayushipriyam07, sukhpreetsvirk, sajal7295, tanujsharma1807}@gmail.com, {arijit, kcr} @iitp.ac.in

***Abstract*—In this paper, development of a fully synthesizable 32-bit processor based on the open-source RISC-V (RV32I) ISA is presented. This processor is designed for targeting low-cost embedded devices. A RISC-V development and validation framework with assembling tools and automated test suits is also presented in this paper. The resulting processor is a single core, in-order, non-bus based, RISC-V processor with low hardware complexity. The proposed processor is implemented in Verilog HDL and further prototyped on FPGA "Spartan 3E XC3S500E" board. This is found that the maximum operating frequency is 32MHz. The power is estimated to be 7.9mW using Xilinx Power Analyzer.**

***Keywords*—RISC-V, RV32I ISA, Micro-architecture, FPGA prototype.**

I. Introduction

Originally designed to support research and education in the area of Computer Architecture, RISC-V instruction set architecture (ISA) is now set to become a standard free and open architecture for academic and industrial applications [1]. For the success and adoption of RISC-V, it has been designed to support for 32-bit, 64-bit and 128-bit address spaces. The ISA is separated into a small base integer ISA, which is a minimal set of instructions adequate to provide a reasonable target for assemblers, linkers, compilers and operating systems. The RISC-V foundation provides its own set of compatible tool chains [2] which includes the above suits.

RISC based architectures have been used in both low level applications and mobile systems by the beginning of the 21st century [3]. The low power and low cost embedded market is dominated by the RISC based ARM [4] architectures. Most of the android based devices, Apple iPhone an iPad and most hand-held devices uses the ARM architecture. The MIPS [5] line can currently be found in games like PlayStation Portable game consoles, Nintendo 64 and personal residential gateways like Linksys WRT54G series. SuperH (SH) is another 32-bit RISC ISA developed by Hitachi. As many of the patents for SuperH are expiring, SuperH2 is being reimplemented as an open source hardware under the name J2 [6].

OpenRISC [7] is aimed at developing an open source ISA based on RISC principles. OpenRISC implements architecture with 16 or 32 general purpose registers (32/64-bit) and a fixed 32-bit instruction length. Two mainline processor core implementations for OpenRSIC are *OR1200* and *mor1kx*. Although not actively developed, *OR1200* is the first original widely used implementation of the processor in Verilog HDL. *Mor1kx* is a novel implementation which is more refined and has diversities with respect to tightly coupled memory, presence of a delay slot or the number of pipeline stages. A number of system-on-chip (SoC) are available for OpenRISC that are used to perform RTL simulations, SystemC simulations or an FPGA synthesis of OpenRISC-powered entire system.

This work presents a hardware design architecture to realize RV32I base integer instruction set for 32-bit address space. The implementation is single core, in-order, non-bus based, single cycle architecture with full support for RV32I base integer instruction set. The application domains include acoustic signal processing, real-time embedded systems, sensor technology and myriad other domains. As required by target applications related to Internet of Things (IoT) and other embedded low-cost devices, the focus has been to optimize for price, power and design complexity at the cost of stringent timing constraints. This designed processor is prototyped on an FPGA board. To facilitate our required works in RISC-V based tiny processor, a suit of tools and test framework around RISC-V were created targeted at 32-bit architectures. The work includes an outline of the developed framework and its use on the preliminary design.

This paper is presented as follows. Section II introduces an overview of the ISA by comparing with other existing free and proprietary ISAs. Section III describes the processor design. Section IV discusses the tool sets created for HDL simulation and experimental setup for FPGA synthesis. Conclusion is provided in section V with remarks on future work.

II. RISC-V (RV32I) Instruction Set

Before selecting the RISC-V instruction set for implementing this processor, an analysis on base integer instructions in the MIPS, OpenRISC, ARM and RISC-V was done [8]. Apart from RISC-V, other architectures lack some features like ARM and MIPS are proprietary standard and the open ISA OpenRISC includes condition codes and branch slots, which embroils higher performance implementations. Using other ISAs would substantially increase implementation complexity.

978-1-5386-3032-7/17/$31.00

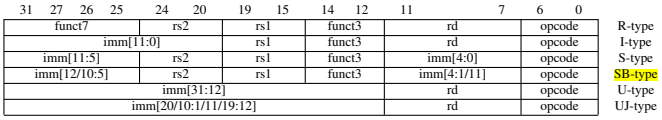
TABLE I
INSTRUCTION TYPE AND OPCODE

Instruction Category (Type)	RV32I Instruction	Example
Integer Register-Register Instructions (R-Type)	add, sub, sll, slt, sltu, xor, srl, sra, or, and	*add* rd,rs1,rs2 The content of register rs1 is added with that of rs2 and the result is stored in register rd.
Integer Register-Immediate Instructions (I-Type)	addi, slti, sltiu, ori, xori, andi, slli, srli, srai	*xori* rd,rs1,imm XOR operation is performed on the content of register rs1 and immediate value and the result is stored in register rd.
Integer Computational Instructions (U-Type)	lui, auipc	*lui* rd,imm The immediate value is placed in the top 20 bits of the register rd, filling in the lowest 12 bits with zeros.
Control Transfer Instructions -Unconditional Jumps (UJ, I-Type)	jal, jalr	*jal* rd,imm The immediate offset is sign -extended and added to PC to form the jump target address. The address of the instruction following the jump (PC+4) is stored into register rd.
Control Transfer Instructions -Conditional Branches (SB-Type)	beq, bne, blt, bltu, bge, bgeu	*beq* rs1,rs2,imm The 12-bit immediate value is added to the current PC to give the target address. Branch is taken if the contents of register rs1 and rs2 are equal.
Load Instructions (I-Type)	lw, lh, lb, lhu, lbu	*lw* rd,rs1,imm A 32-bit value is copied from memory to register rd. The effective byte address is obtained by adding register rs1 to the sign-extended 12-bit offset.
Store Instructions (S-Type)	sw, sh, sb	*sw* rs1,rs2,imm The 32-bit value in register rs2 is copied to memory. The effective byte address is obtained by adding register rs1 to the sign-extended 12-bit offset.

RV32I was designed to be ample to form a compiler target and support modern operating system environments. It was also designed to reduce the hardware required for a minimal implementation. RV32I contains 47 unique instructions. A more integrated and complex implementation might cover the eight SCALL/SBREAK/CSRR instructions with a single SYSTEM hardware instruction that always traps and might be able to implement the FENCE and FENCE.I instructions as NOPs which will reduce hardware instruction count to a total of 38. This implementation includes these reduced 38 hardware instructions.

RISC-V has 32 registers in register file indexed from 0 to 31, out of which 31 are general purpose registers (indexed 1 to 31) and register indexed 0 is hardwired to constant 0. RISC-V does not specify any software calling convention, so MIPS calling convention [9] was followed for this work. The program counter is another user visible register in RISC-V. There are four core instruction formats in the RV32I base integer instruction set: R, I, S and U. Table I summarizes the instructions with examples and Table II represents the encoding of these instructions in RV32I [1]. Every instruction is of fixed 32-bits in length and must be memory-aligned to four-byte boundary. If this is not followed, an instruction address misaligned exception is generated. RISC-V base ISA does not have instructions for checking integer arithmetic overflows and hence no hardware level checks for overflow in integer arithmetic operations is implemented in RV32I as they can be implemented using RISC-V branches.

TABLE II
INSTRUCTION ENCODING



III. ARCHITECTURE DEVELOPMENT AND IMPLEMENTATION

32-bit processor is used for most embedded applications where a tiny processor is required. A single cycle design was chosen for initial baseline performance analysis. The subsequent subsections describe about top level and microarchitecture of this design in detail.

A. Top Level Overview

At the top level, the processor core can be divided into four logical modules as shown in Figure 1. The *fetch, decode and control logic* block is responsible for fetching the instruction from the instruction memory, decoding the instruction and generating the control signals. It is also responsible for resolving jump and branch target addresses. It hosts the program counter, the target address selection logic, the instruction memory controller, the instruction decoder and a control unit. A dedicated adder is also included for incrementing the PC in each cycle. The control unit is purely combinatorial with all the control signals generated in the same cycle. Dedicated signals have been provided for both internal and external exceptions.

The *register bank and ALU logic* module comprises of the 31 general purpose registers, the register addressing, read and write logic as well as arithmetic logic unit (ALU). ALU in this module deals only with arithmetic register-register and register-immediate operations. For register-immediate operations, the *sign extension, shift and shuffle logic* block provides the properly sign extended and reordered 32-bit immediate operand for ALU and the second operand is drawn from the register bank. ALU is not involved in branch and jump target address calculation and dedicated adders have been included for the same. For branch instructions, ALU calculates the condition and provides the results to the control unit which then decides whether a branch is to be taken. ALU is kept close to the register bank to decrease critical path delay. There are five different types of immediate formats, named as I, S, B, U and J-type immediate. The encoding of immediate in instruction often has shuffled bits as can be seen in Table II. The decoding involves the proper reordering, left shift or sign extension of the immediate values. Such an encoding is chosen to minimize implementation complexity across ISA extensions . This type of immediate decoding is handled by the *sign extension, shift and shuffle logic* module. The *memory control logic* module is included to accommodate the various store/load operations which are allowed to have word aligned as well as misaligned addresses of the data memory.

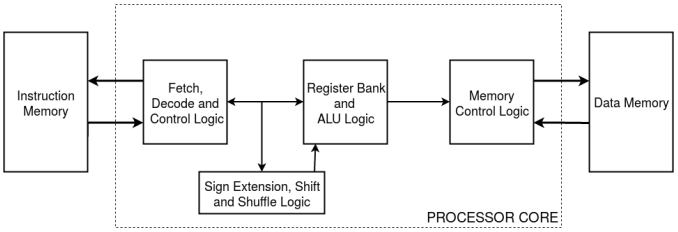


Fig. 1. Top Level Logical Overview

B. Micro-Architecture

Figure 2 describes the implementation of data and control path of the architecture. The *control unit* (CU) takes three inputs namely, opcode(0 to 6 bits), funct3(12 to 14 bits) and the 30th bit of the instruction. On the basis of these inputs CU provides different control signals to different modules for selection and proper execution of instructions. There are total of 12 control lines provided by the CU. For instance, ALU_Ctrl (4-bit) tells ALU which operation has to be performed on the operands; when Reg_WR (1-bit) is "high", it puts the register file in write mode and when it is "low", it puts the register in read only mode. Similarly, other modules are controlled.

The *sign extension, shift and shuffle logic* module encompasses two sign extending modules, a left shift module and two sign extend and shuffle module. The SIGN EXTN.1 and SIGN EXTN.2 modules deal with the 12-bit, I-type and S-type immediate respectively. S-type immediate is encoded into two disjoint segments in the instruction and this distinguishes it from the I-type immediate. The SHIFT LEFT 12 block is used for the U-type instructions. These encode a 20-bit immediate which is used in the most significant 20 bits, hence it is shifted left by 12. Both SIGN EXTN. & SHUFFLE blocks are active for J-type and B-type control transfer instructions. The immediates encoded in the instructions, are reordered in the decoding process and sign extended as required by the ISA in the *sign extension, shift and shuffle logic* module. The maximum allowable size of instruction memory and data memory can be 4GB, though for simulation process the size of instruction memory was taken as 64KB and data memory as 16KB. The register file contains 32 registers of 32-bit width. The program counter (specifically, program counter + 4) can be accessed using *jal* instruction by setting offset as 0.

The *store block* assumes that data is written one word at a time at word aligned addresses. Hence, it calculates the appropriate word aligned address and the offset in the word at this address, when a misaligned store word/half/byte is invoked. For instance, with a misaligned store byte instruction *(sb)*, the store logic identifies the word aligned address containing the specified byte and the offset of the byte within the word. The store logic reads the word containing the required byte from the data memory using the calculated word aligned address. Using the offset calculated, it creates a new word to be written by masking the bits that are not to be altered. This word is then written back to the memory. Similarly, the *load block* calculates the correct word aligned address and offset for a load instruction, reads the word and shifts the halfword or byte within the word to the lower bits if required and finally sign extends or zero extends them as per the ISA specifications.

Because of the single cycle nature of this proposed design, a level of redundancy is needed so that a balance can be achieved in component reuse between instructions and critical path length. Hence, the branch target calculation is accomplished using an additional adder and each of the five I-type instruction has dedicated sign-extension and reordering modules. The program counter also has a dedicated adder for incrementing its value.

The program counter calculates the address of the instruction to be executed. The required inputs from instruction are provided to the CU, register file and sign extension, shift and shuffle logic modules. CU controls each module by providing required signal to each module. The instruction gives the address of the registers (rs1, rs2 and rd) to the register file. ALU performs the operation specific to the instruction on its inputs, specified by the multiplexers, controlled by CU. According to the control bits ALU output is then used to write in register file or address specification of data memory (in load or store instructions) or address specification of program counter (in jump instructions). Address specification of the program counter is controlled by the multiplexer.

IV. Simulation Environment and FPGA Prototype

Figure 3 shows the full flow of the experiment. The assembler translates assembly code of the instructions into binary and stores them in a separate file. This data is loaded to the instruction memory for the execution. Then the given program is tested using the simulation environment. After validation the processor was prototyped on FPGA with the results displayed on the LCD. The description of our core was built using Verilog HDL. For simulation vvp [10] with the validation framework was used. Xilinx ISE 14.7 was used for synthesis and FPGA prototyping.

A. Assembler

For the early development processes, an assembler [11] was developed for one-to-one translation between instructions and its corresponding binary equivalent as can be seen in Table III. The binary equivalent obtained is written on the instruction memory for further execution. The assembler developed is useful for base integer ISA. With this configuration, the assembler is able to validate instructions with respect to the ISA, tokenize them, allow label based jumps and create one-to-one translations. There are provisions to allow invalid instructions at the translation stage so that the behavior of the hardware on receiving faulty instructions can be observed.

B. HDL Simulation

A generic, architecture independent 32-bit RISC-V validation suit was developed. It currently supports the RV32I base integer instruction set fully. The validation suit can work with any RV32I architecture written in Verilog HDL for validation purposes. The framework includes a general set

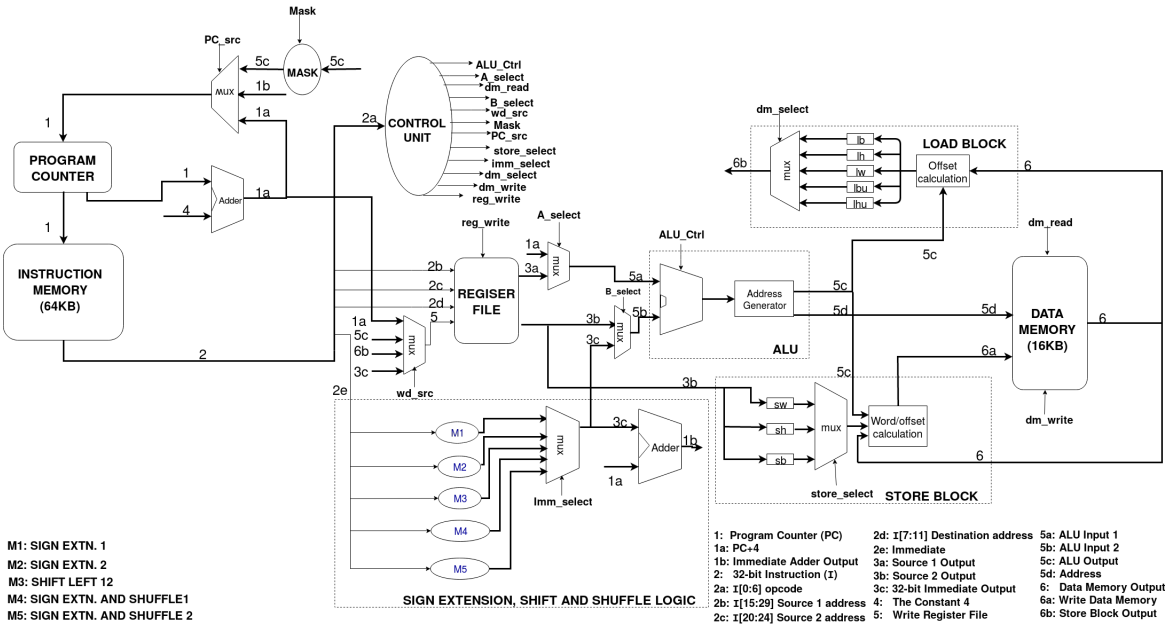


Fig. 2. Micro Architecture

of integrated tests, an automated testing and analysis module and related output verification scripts. This framework can be used to validate any RV32I compatible implementation or simulation. For testing at different levels, various programs as test cases are provided to assembler for translation and then to instruction memory for testing and validation. The test cases extensively test the correctness on an instruction level basis as well as on a program level basis. A set of unit tests cases for each individual module which unlike the integrated tests are not generalizable, were developed. The automated test cases proposed can be configured to work with any other simulation suit.

For overall testing various programs were written including Fibonacci series generation, sum of first N natural numbers and multiplication by repeated addition. A code snippet shown in Table III is a part of the program that computes the sum of the first N natural numbers recursively and returns the

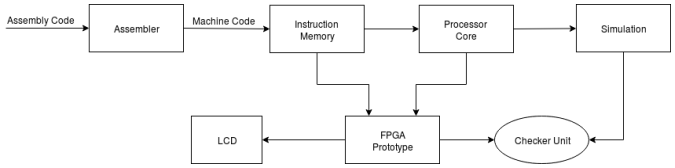


Fig. 3. Experiment Flow

resulting sum. The code is written using 38 instructions of RV32I in assembly language described by the assembler [11]. The program is assembled by the assembler and a tiny boot-loader which initializes the hardware/simulation, loads the program onto the memory file and transfers the control to the entry point. The *PUSH* and *POP* labels used as *jal* targets are routines emulating the push and pop functionality of a stack

TABLE III
CODE SNIPPET AND ITS MACHINE CODE

Assembly Code			Machine Code
# Recursively finds the sum			
# of first N natural numbers.			
FAB:			
	addi	r3 , r31 , 0	00000000000001111100000011001 0011
	jal	r31, PUSH	00000011110000000000111111101111
	addi	r4, r0, 2	00000000001000000000001000010011
	blt	r1, r4, RETONE	00000010010000001100001001100011
	add	r3, r0, r1	00000000000010000000000110110011
	jal	r31, PUSH	00000010110000000000111111101111
	addi	r1, r1, -1	11111111111110000100000010010011
	jal	r31, FAB	11111110010111111111111111101111
	jal	r31, POP	00000010110000000000111111101111
	add	r29, r29, r3	00000000001111101000111010110011
	addi	r3, r0, 0	00000000000000000000000110010011
	beq	r3, r0, RET	00000000000000011000010001100011
RETONE:			
	addi	r29 , r0 , 1	00000000000100000000111010010011
RET:			
	jal	r31,POP	00000001100000000000111111101111
	add	r31,r0,r3	00000000001100000000111110110011
	jalr	r0,r31,0	00000000000011111000000001100111

using the load and store instructions. *PUSH* saves the value in register *r3* to the top of the stack indicated by register *r30*, which is considered as the stack pointer for this particular implementation. Similarly, *POP* pops the value at the top of the stack and loads it into register *r3*. The result of a sample run of the simulation on Verilog HDL with $N = 20$ is shown in Figure 4. Register *r29* is used to hold the return value. Figure 4 depicts the instruction in which the sum is outputted by ALU (*walu_out*) and is being written back to register *r29*.

C. FPGA Prototype

The experimental setup for the FPGA prototyping is presented in Figure 5. Synthesized design code was simulated at RTL level using Xilinx ISE 14.7. The design was mapped

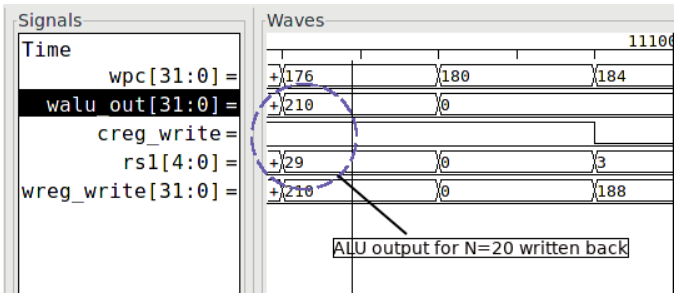


Fig. 4. Sample Simulation Output

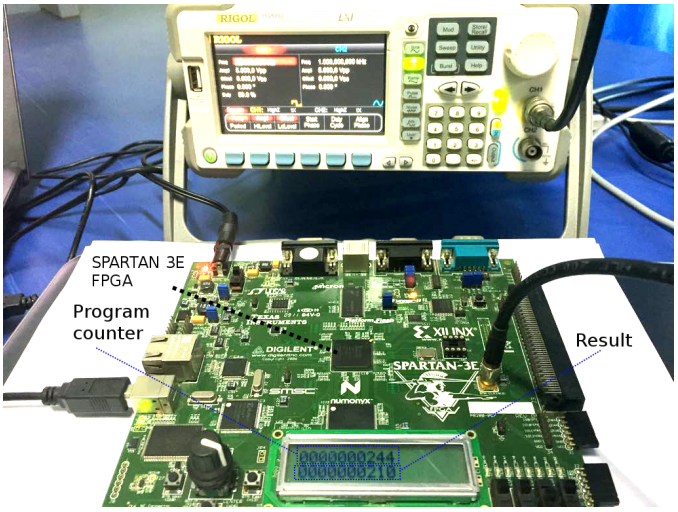


Fig. 5. Experimental Setup

onto Xilinx's Spartan-3E XC3S500E FPGA. The utilization of the hardware for the processor is given in Table IV. The on-chip block RAM of 64KB for instruction memory and 16KB for data memory was used. It is reported that the design can obtain a maximum operating frequency of 32MHz on Spartan-3E FPGA. It consumes a total 7.9mW of power. For displaying the result, on-board 16x2 LCD display of Spartan-3E was interfaced with the processor using memory mapped I/O (MMIO). Many programs such as multiplication by repeated addition, finding Nth Fibonacci number and recursive sum of N numbers were tested on Spartan 3-E FPGA. Figure 5 displays the output of the routine that recursively finds the sum of the first N numbers as described in the previous subsection. For this experimental run, N was taken as 20 and the resulting sum is shown. The first row of LCD display shows the program counter value and second row shows the output value.

V. Conclusion

The enormous complexity of modern microprocessor designs poses significant challenges in the usage of these systems. In the current paper, authors have presented their effort towards implementing a novel single-cycle tiny embedded processor design, which is modular and highly extensible. On-chip block RAMs of 16KB for data memory and 64KB for instruction memory are used. A peak clock speed of 32MHz is

TABLE IV
Design Utilization Summary

Logic Utilization	Used
Total LUTs	5578
Number of FFs	1073
Number of Slices	3393
RAM	1 (512 x 32-bit)
ROM	1 (64 x 32-bit)
Adder/Subtractor	313
Counters	1
Registers	1071
Comaparator	312
Multiplexers	131
Logic Shifters	3
XOR Gates	4

attained by the processor by consuming total power of 7.9mW targeting a commercially available FPGA device "Spartan 3EXC3S500E". Considering the advantage of the growing RISC-V community together with the existing tool-chain and software around this new instruction set and based upon this design paradigm, the current processor design paves the way of future implementations for specific and general applications in the world of IoT and real-life embedded applications. Future goals include integrating I/O bus, adding multi-level cache mechanism, incorporating interrupts and amalgamating with a floating point co-processor [12]. These results might be reported in a future paper for use in education, research and industry, while at the same time creating opportunities for faster designs.

References

[1] A. Waterman, Y. Lee, D. A. Patterson, K. Asanovic, V. I. U. level Isa, A. Waterman, Y. Lee, and D. Patterson, "The risc-v instruction set manual," 2014.
[2] (2016) Software tools, risc-v foundation. [Online]. Available: https://riscv.org/software-tools/
[3] S. P. Dandamudi, *Guide to RISC Processors: For Programmers and Engineers*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.
[4] Wikipedia, "Arm architecture — wikipedia, the free encyclopedia," 2017, [Online; accessed 26-February-2017]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=ARM_architecture&oldid=766746198
[5] Primefac, "Reduced instruction set computing wikipedia, the free encyclopedia," 2017, [Online; accessed 26-February-2017]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Reduced_instruction_set_computing&oldid=765887154
[6] J. core Organisation, "J2 open processor," 2014, [Online; accessed 28-February-2017]. [Online]. Available: http://j-core.org/
[7] J. Tandon, "The openrisc processor: Open hardware and linux," *Linux J.*, vol. 2011, no. 212, Dec. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2123870.2123876
[8] K. Asanovi and D. A. Patterson, "Instruction sets should be free: The case for risc-v," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146, Aug 2014. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html
[9] E. Farquhar and P. Bunce, *The Mips Programmer's Handbook*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.
[10] S. Williams, "vvp(1) - linux man page," 2001, [Online; accessed 8-January-2017]. [Online]. Available: https://linux.die.net/man/1/vvp
[11] Removed for anonymity.
[12] V. Patil, A. Raveendran, P. M. Sobha, A. D. Selvakumar, and D. Vivian, "Out of order floating point coprocessor for risc v isa," in *2015 19th International Symposium on VLSI Design and Test*, June 2015, pp. 1–7.