CPTR380 RISC-V RV32I FPGA Implementation

Jaron Brown & Ethan Jansen

***Abstract*—TBD**

***Index Terms*—ISA, RISC, RISC-V, RV32I FPGA, VHDL.**

I. Introduction

Since designing a custom accumulator-based CPU last quarter for the Intro to Digital Design class, both of the authors developed an interest in implementing a more commonly supported instruction set architecture (ISA). When presented with the opportunity to pursue a project related to computer architecture in their Computer Architecture class, both authors jumped at the opportunity to implement another CPU design. They initially considered implementing the MIPS (Microprocessor without Interlocked Pipelined Stages) ISA, which was being studied in class, but, after conducting more research, decided on implementing one of the RISC-V standard ISAs, RV32I. This keeps with the RISC (Reduced Instruction Set Computer) design, and takes advantage of the abundant documentation for the RISC-V ISAs, RV32I's relative simplicity, the high commercial interest in these ISA specifications, and in the interest in supporting open-source projects.

One of the dominant reasons many companies are developing RISC-V implementations is its open-source nature [1]. The dominant ISAs (x86, ARM, MIPS, etc.) all have steep licensing fees. The RISC-V ISAs, in comparison, are completely free. Having a completely open-source ISA just makes sense–it lowers the barrier to entry for designing hardware to implement standard ISAs and allows for more innovation in the realm of CPU design.

II. A Brief Introduction to RISC-V

The original design of RISC-V was begun in 2010 at the University of California, Berkeley. It was designed not only as an academic learning aid but also as a set of ISAs that could have commercial viability [1]. As its name suggests, RISC-V is a RISC architecture. This means, in comparision to CISC (Complex Instruction Set Computer) ISAs, RISC ISAs have simple instructions. This means more individual instructions will be required to complete a given task compared to CISC ISAs but also allows the hardware implementation to be much simpler than that for CISC ISAs. In fact, even implementations of x86-64, the most widely adopted CISC ISA, breaks its CISC instructions into ”microinstructions” which are easier to implement in hardware, similar to RISC ISAs [2].

RISC-V is not actually *an* ISA but rather a collection of ISAs, ranging from 32-bit to 128-bit standards and supporting various levels of complexity by way of extensions to the base ISAs [3]. Some official extensions include the M extension which adds multiplication and division support, the A extension which adds support for atomic instructions, and the F and D extensions which add support for single- and double-precision floating numbers, respectively [3]. Developers are encouraged to design their own compatible extensions to customize the base RISC-V ISA to their specific application.

The RV32I ISA defines four main instruction types, with an additional two instruction types which are subsets of the main instruction types [3]. They are as follows:

- R-type: Supports integer register-register arithmetic operations
- I-type: Supports integer register-immediate arithmetic operations
- S-type: Supports store instructions
- B-type (SB-type): Supports branch instructions
- U-type: Supports load upper immediate (`LUI`) and add upper immediate to `pc` (`AUIPC`) instructions
- J-type (UJ-type): Supports jump and link (`JAL`) instructions

In this project, revision 2.1 of the RV32I unprivileged ISA will be implemented in VHDL and tested on a Xilinx Artix 7 FPGA (Field-Programmable Gate Array). RV32I is the RISC-V 32-bit base integer ISA and implements the core functionality of the RISC-V architecture family [3]. Using this, the implemented microcontroller should easily support C code cross-compilation via clang after future I/O library support. This will help test/debug the microcontroller during implementation. Even more notable, however, is this will greatly improve the accessibility of this project as a useful processor.

A. Stretch Goals

The following goals would all improve the design, however are not necessary to meet a basic RV32I implementation. Because of time constraints these are items are not actively being pursued.

- I/O: I/O support is not part of RV32I. Because of this, an extension, or a co-processor, would be needed for proper implementation. For these reasons, proper I/O implementation is a stretch goal. However, because basic I/O support is required for testing/debugging, it will be unofficially implemented.
- Pipelining: Although RV32I supports pipelining, this project's implemetation will not implement this feature due to projected time constraints. Pipelining is not necessary to meet ISA requirements, and the lack of pipelining will only increase the clock cycles per instruction (CPI) but will greatly simplify the control logic and data path unit (DPU).

Manuscript created March, 2024.

- Multiplication & Division: These are technically part of the M extension [3] and thus are not high priorities. However, because `IEEE.numeric_std` (the VHDL library) has built in support for multiplication and division this would not be difficult to add to the existing arithmetic logic unit (ALU). The ALU would simply require another bit of control.
- Standard C Code Compilation Support: Basic C support is planned for this project, however exensive support is currently not guaranteed.

The authors intend to continue supporting this project by refining the design and implemented the discussed stretch goals.

III. Implementation

A. Overview

This project's implementation of non-pipelined RV32I consists of a few core components:

- instruction & data memory (separated)
- a register file
- an ALU
- various other DPU components related to sign-extension, incrementing the program counter, branching, and routing signals
- a control module

Unlike the authors' previous CPU designs which used an accumulator-based design, this implementation uses a register file with 32 32-bit wide registers. Besides the different ISA support (the original accumulator-based CPU used a custom ISA), the switch to using a bank of registers is the most significant hardware design change between the two implementations. This change, however, is in keeping with the RV32I standard and allows for more flexibility when programming the CPU.

B. Program Counter

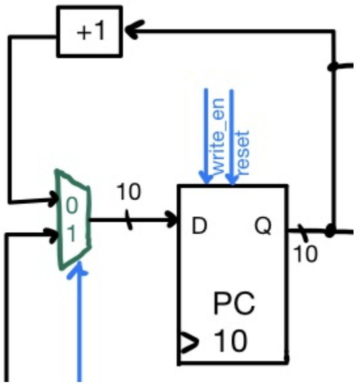

Fig. 1. Program Counter with Branch & Jump Support

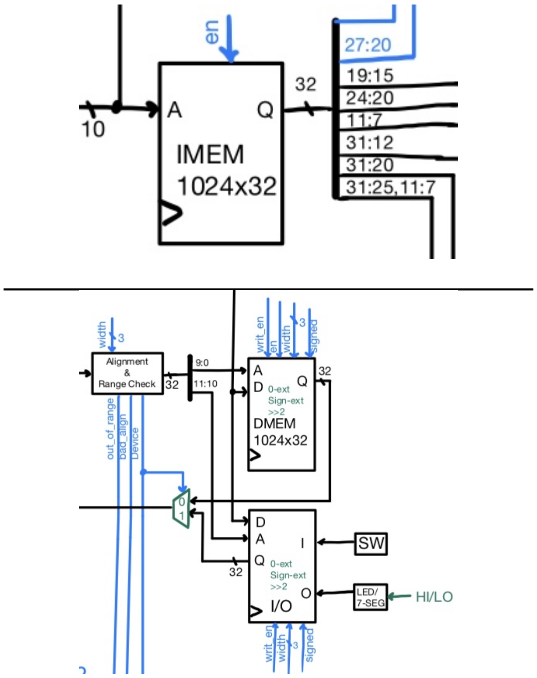

Fig. 2. Instruction Memory (top). Data Memory with I/O Support (bottom).

C. Instruction Memory, Data Memory, and I/O

Each memory (instruction and data) block was implemented in VHDL using AMD's write-first memory example using an array of `std_logic_vector`'s [4]. This allows for quick implementation; furthermore, it ensures that the memory blocks will be mapped to a physical FPGA memory block for improved performance –the clock will not have to be further delayed as it may have been otherwise.

One change from AMD's example that needed to be made was support writing bytes and half-words for data memory. This is to support the `sb` and `sh` instructions as they do not overwrite the upper bits. Without the additional support baked into the data memory, an additional register would need to be added to temporarily hold the upper bits so that they do not get overwritten–this would require additional clock cycles to implement.

The instruction memory has been implemented and holds a maximum of 1024 32-bit instructions. As of now the instruction and data memory depth is not defined, however RV32I supports 32-bit addresses. Because of this the goal to implement 2^{32} deep memory blocks, however there may be some currently unkown limitation of the FPGAs used for testing.

D. Register File

Incomplete..

Similar to the Data Memory implementation, the register block supports writing byte and half-words in addition to full words. **Note:** this might be changed as when a register is written to the full register is overwritten regardless, even when only writing bytes and half-words.

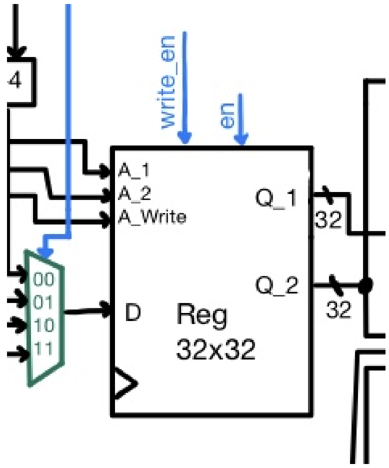

Fig. 3. 32-bit Register File

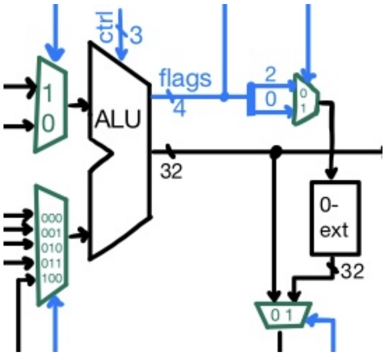

Fig. 4. ALU with Conrol Flag Output

E. ALU

The arithmetic logic unit is a fully combinational block which will perform the desired arithmetic operation based on the input control lines. Shown in Table III-E are the main operations the ALU is required to support. Because there are various instructions associated with the same arithmetic operation (such as needing to determine the address used for load and store using ADD) the control line into the ALU is only 3-bits wide in order to support the eight different arithmetic operations, but the actual control signal is determined from a separate ”ALU Control” block.

This ALU Control block takes in `opcode`, `funct3`, and `funct7` from the encoded instruction and translates this into the necessary 3-bit ALU control signal. Separating this block also allows more easily decoding the necessary information from the six different [3] instruction types.

The ALU also outputs flags indicating if the result is less than or greater than zero. With these two flag outputs any of the branch conditions can be supported:

- BEQ: equals
- BNE: not equals
- BLT & BLTU: less-than
- BGE & BGEU: greater-than

Furthermore, the $r < 0$ flag is also zero-extended and output (via a mux) in order to support the SLT & SLTI (set less-than) instructions–which set a register to 1 or 0 if the first imput is less-than/greater-than the second input.

Operation	Instructions
ADD	ADD(I), Load & Store (various), JALR, AUIPC
SUB	SUB, SLT (various), Branch (various)
XOR	XOR(I)
OR	OR(I)
AND	AND(I)
Left Shift	SLL(I)
Right Shift	SRL(I)
Right Shift (Arith.)	SRA(I)

TABLE I
RV32I BASE ARITHMETIC OPERATIONS MATCHED TO THE INSTRUCTIONS

F. DPU Miscellaneous

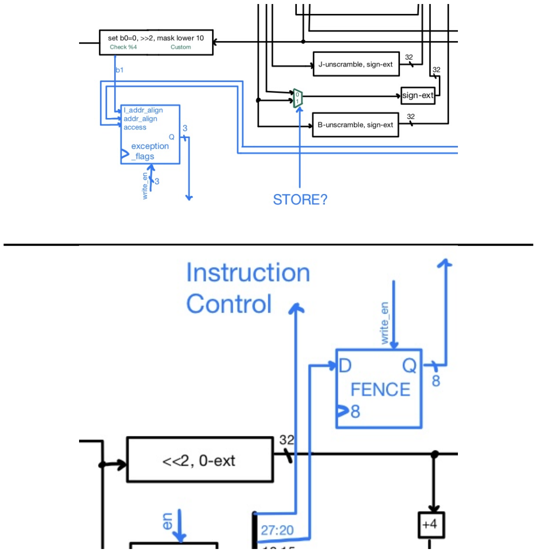

Fig. 5. Stuff

TBD

G. Control

Incomplete...

The control is primarily separate from the DPU (apart from a few components such as the ALU control). This nicely separates the control from the rest of the CPU processing and makes a nice division in the design. We were able to do this by not considering pipelining, thus removing any forwarding registers and branch prediction modules from being interweaved into the DPU.

However, because there is no pipelining, the controller needs to be aware of how long each instruction takes to execute. This is mostly separated into the six different instruction types–where each instruciton within an instruction type takes the same number of clock cycles to execute. For the current implementations, the instruction length is hardcoded into the controller. For instance, the controller will wait four clock cycles for an ADD to execute before continuing to the next instruction.

The controller also has support of pausing and single-stepping. This greatly improves the debugging of the device. This is useful not only during hardware design of the CPU but also when writing software to run on the CPU. Currently, however, there is no proper I/O support, so there are dedicated buttons on the FPGA board to control the single-stepping. Thus, single-stepping can not be controlled via a software debugger. This is planned for future work.

IV. Testing & Design Review

To verify this project's RISC-V implementation is valid, a few methods can be used. First of all, the individual modules must be tested in isolation. This can be done one of two ways: implementing testbenches in either VHDL or Python using cocotb or by creating testing wrappers which provide inputs to the modules and display the outputs on either the LEDs, seven-segment display, or digital pins on the FPGA board. Implementing testbenches is the ideal method of testing modules. Both the authors have experience using testing wrappers and reading the test results using the FPGA board. They hope to use the built-in testbench implementation build into Xilinx ISE, which is the program the authors use for synthesis and configuration of the the FPGA. They did consider using cocotb, which is a HDL testbench written in Python. However, cocotb was found to be difficult to configure and may be considered later on.

V. Conclusion

TBD

Complete Data Path Unit Block Diagram

VHDL Implementations

Sample: TBD due to current overfull issues...

A. Program Counter

```
library IEEE;
use IEEE.std_logic_1164.all;

entity program_counter is
  port
  (
    clk   : in std_logic; --! Clock
    reset : in std_logic; --! Reset to 0
    en    : in std_logic; --! Enable
    D     : in std_logic_vector(9 downto
     ↪  0); --! New pc
    Q     : out std_logic_vector(9
     ↪  downto 0) --! Current pc
  );
end program_counter;

architecture pc of program_counter is
begin
```

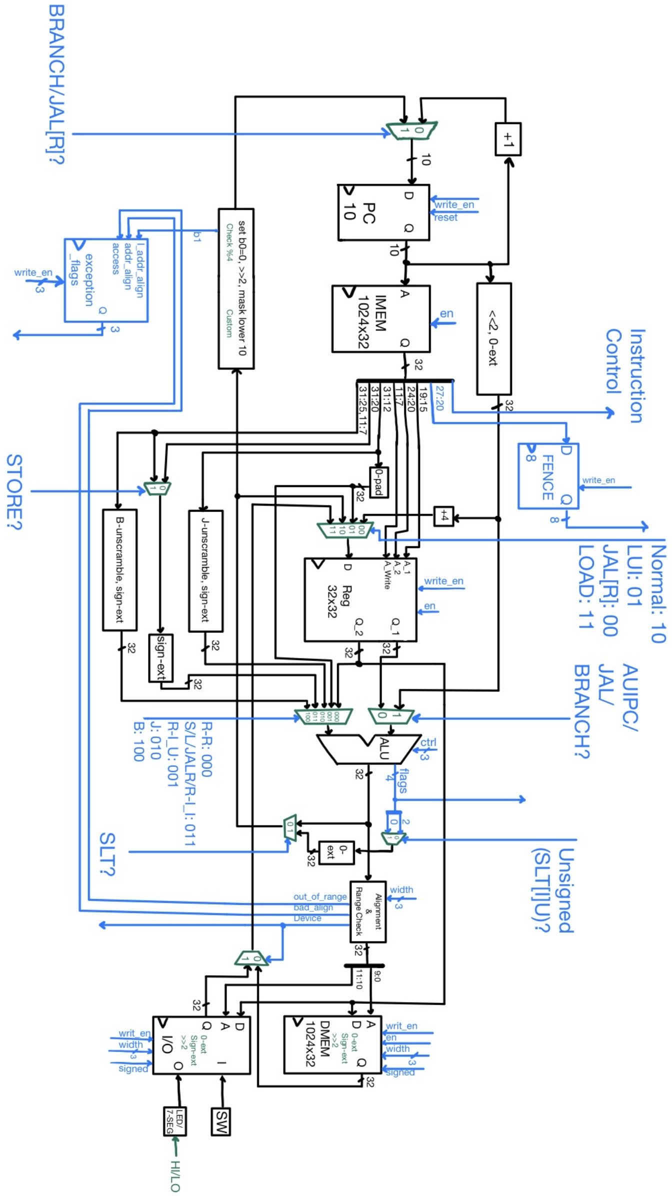

Fig. 6. Data Path Unity with Control Signals (blue)

```
  process (clk)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        Q <= "0000000000";
      elsif en = '1' then
        Q <= D; --! When reset != 0 and
         ↪  en = 1
      else
        Q <= Q; --! When reset != 0 and
         ↪  en = 0
      end if;
    end if;
  end process;

end pc;
```

B. Instruction Memory

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_textio.all;
use std.textio.all;

entity rom_with_init is
  port
  (
    clk  : in std_logic; --! Clock
    en   : in std_logic; --! Enable
    addr : in std_logic_vector(9 downto
     ↪  0); --! Address
    d    : out std_logic_vector(31
     ↪  downto 0) --! Data Out
  );
end rom_with_init;

architecture rom of rom_with_init is
  type rom_type is array(0 to 1023) of
   ↪  std_logic_vector(31 downto 0);

  impure function
   ↪  initRomFromFile(romFileName : in
   ↪  string) return rom_type is
    file romFile
     ↪  : text is in romFileName;
    variable romFileLine
     ↪  : line;
    variable rom
     ↪  : rom_type;
  begin
    for i in rom_type'range loop
      readline(romFile, romFileLine);
      read(romFileLine, rom(i));
    end loop;
    return rom;
  end function;

  signal rom : rom_type :=
   ↪  initRomFromFile("rom.data");
begin
  process (clk)
  begin
    if rising_edge(clk) then
      if en = '1' then
        d <= rom(to_integer(
          unsigned(addr)));
      end if;
    end if;
  end process;
end rom;
```

C. ALU

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity arithmetic_logic_unit is
  port
  (
    ctrl               : in
     ↪  std_logic_vector(2 downto 0);
     ↪  --! From control module
    data1_in, data2_in : in
     ↪  std_logic_vector(31 downto 0);
     ↪  --! Assuming immediate sign
     ↪  extensions happen outside of alu
    data_out           : out
     ↪  std_logic_vector(31 downto 0);
    flags              : out
     ↪  std_logic_vector(3 downto 0)
  );
end arithmetic_logic_unit;

architecture alu of
 ↪  arithmetic_logic_unit is
  constant sel_add
   ↪  : std_logic_vector(2 downto 0) :=
   ↪  "000";
  constant sel_sll
   ↪  : std_logic_vector(2 downto 0) :=
   ↪  "001";
  constant sel_sub
   ↪  : std_logic_vector(2 downto 0) :=
   ↪  "010";
  constant sel_sra
   ↪  : std_logic_vector(2 downto 0) :=
   ↪  "011";
  constant sel_xor
   ↪  : std_logic_vector(2 downto 0) :=
   ↪  "100";
  constant sel_srl
   ↪  : std_logic_vector(2 downto 0) :=
   ↪  "101";
  constant sel_or
   ↪  : std_logic_vector(2 downto 0) :=
   ↪  "110";
  constant sel_and
   ↪  : std_logic_vector(2 downto 0) :=
   ↪  "111";
  signal nonzero_s, nonzero_u
   ↪  : std_logic;
  signal shift
   ↪  : std_logic_vector(4 downto 0);
  signal sig_sra
   ↪  : std_logic_vector(31 downto 0);
  signal sig_xor, sig_srl, sig_or,
   ↪  sig_and : std_logic_vector(31
   ↪  downto 0);
  signal sig_sll
   ↪  : unsigned(31 downto 0);
  signal sig_add, sig_sub
   ↪  : signed(31 downto 0);
  signal unsigned_ext_1, unsigned_ext_2
   ↪  : std_logic_vector(32 downto 0);
  signal sig_unsigned_comp
   ↪  : signed(32 downto 0);

begin
  -- signals all calculated to be muxed
   ↪  via ctrl

  -- logic
  shift   <= data2_in(4 downto 0);
  sig_add <= signed(data1_in) +
   ↪  signed(data2_in);
  sig_sll <=
   ↪  shift_left(unsigned(data1_in),
   ↪  to_integer(unsigned(shift)));
  sig_sub <= signed(data1_in) -
   ↪  signed(data2_in);
  sig_sra <=
   ↪  std_logic_vector(shift_right(
    signed(data1_in),
     ↪  to_integer(unsigned(shift))));
  sig_xor <= data1_in xor data2_in;
  sig_srl <=
   ↪  std_logic_vector(shift_right(
    unsigned(data1_in),
     ↪  to_integer(unsigned(shift))));
  sig_or  <= data1_in or data2_in;
  sig_and <= data1_in and data2_in;

  nonzero_s <= '0' when sig_sub =
   ↪  X"00000000" else
    '1'; --! Check difference is
     ↪  non-zero
  flags(3)           <= (not sig_sub(31))
   ↪  and nonzero_s; --! Positive and
   ↪  non-zero
  flags(2)           <= sig_sub(31); --!
   ↪  Sign bit of signed subtraction
   ↪  operation
  unsigned_ext_1     <= '0' & data1_in;
   ↪  --! Effectively create positive
   ↪  signed value from unsigned
  unsigned_ext_2     <= '0' & data2_in;
   ↪  --! Effectively create positive
   ↪  signed value from unsigned
  sig_unsigned_comp <=
   ↪  signed(unsigned_ext_1) -
   ↪  signed(unsigned_ext_2);
  nonzero_u          <= '0' when
   ↪  sig_unsigned_comp =
   ↪  "000000000000000000000000000000000"
   ↪  else
    '1'; --! Check difference is
     ↪  non-zero
  flags(1) <= (not
   ↪  sig_unsigned_comp(32)) and
   ↪  nonzero_u;
  flags(0) <= sig_unsigned_comp(32);
  -- signal select mux
  with ctrl select
    data_out <=
    std_logic_vector(sig_add) when
     ↪  sel_add,
    std_logic_vector(sig_sll) when
     ↪  sel_sll,
    std_logic_vector(sig_sub) when
     ↪  sel_sub,
    sig_sra when sel_sra,
    sig_xor when sel_xor,
    sig_srl when sel_srl,
    sig_or when sel_or,
    sig_and when others;
end alu;
```

D. Register File

```
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity register_block is
    port
    (
      clk               : in std_logic;
       ↪  --! Master clock
      we                : in std_logic;
       ↪  --! Write enable
      en                : in std_logic;
       ↪  --! Keeps Q_1 and Q_2 from
       ↪  changing
      A_1, A_2, A_Write : in
       ↪  std_logic_vector(4 downto 0);
       ↪  --! Address inputs
      D                 : in
       ↪  std_logic_vector(31 downto 0);
       ↪  --! Data in
      Q_1, Q_2          : out
       ↪  std_logic_vector(31 downto 0)
       ↪  --! Data out
    );
end register_block;

architecture reg_blk of register_block
 ↪  is
  type reg_type is array (31 downto 0)
   ↪  of std_logic_vector(31 downto 0);
  signal REG : reg_type :=
   ↪  (others=>(others=>'0'));


begin
  process(clk)
  begin
    if rising_edge(clk) then
      if en = '1' then
```

```
        if we = '1' and A_Write /=
         ↪  "00000" then

           ↪  REG(to_integer(unsigned(A_Write)))
           ↪  <= D;
          if (A_Write = A_1) and
           ↪  (A_Write = A_2) then
            Q_1 <= D;
            Q_2 <= D;
          elsif (A_Write = A_1) then
            Q_1 <= D;
            Q_2 <=
             ↪  REG(to_integer(unsigned(A_2)));
          elsif (A_Write = A_2) then
            Q_2 <= D;
            Q_1 <=
             ↪  REG(to_integer(unsigned(A_1)));
          else
            Q_1 <=
             ↪  REG(to_integer(unsigned(A_1)));
            Q_2 <=
             ↪  REG(to_integer(unsigned(A_2)));
          end if;
        else
          Q_1 <=
           ↪  REG(to_integer(unsigned(A_1)));
               Q_2 <=
                ↪  REG(to_integer(unsigned(A_2)));
        end if;

      end if;
    end if;
  end process;
end reg_blk;
```

References

[1] RISC-V International, “History of risc-v,” 2023. [Online]. Available: https://riscv.org/about/history/

[2] M. E. Thomadakis, “The architecture of the nehalem processor and nehalem-ep smp platforms,” *Texas A&M University*, pp. 24–25, Mar 2011. [Online]. Available: https://web.archive.org/web/20140811023120/http://sc.tamu.edu/systems/eos/nehalem.pdf

[3] RISC-V Foundation, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*, pp. i–30, Dec 2019. [Online]. Available: https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf

[4] AMD, *Vivado Design Suit User Guide, UG901 (v2023.2)*, pp. 120–160, Nov 2023. [Online]. Available: https://docs.amd.com/viewer/book-attachment/VXDyposQ4vTSzkZQR83coQ/b6zTvHNSY07j1WdniHx~dw