

Final Report

FPGA RISC-V Processor

Team Members:
Christopher Nielsen CE
Hillary Ng CE
Christopher Shamah CE

Faculty Advisor:
Peter Milder

1. Background

As the world progresses, computers have played a big role in this world with the never-ending demand for a progressively more advanced processor. The increasing demand for customizable, energy efficient, and open-source processor architectures has made the RISC-V architecture a more desirable alternative compared to the traditional proprietary instruction sets. The modularity and the extensible ISA allows RISC-V system designers to have more of a wide range of hardware applications, from embedded systems to high performance computing, while maintaining transparency, simplicity, and a broad open source toolchain support. In the academic settings, the RISC-V ISA has become an ideal opportunity for exploring and learning modern processor design principles.

Field-Programmable Gate Arrays (FPGAs) have also gained popularity as a flexible platforms to prototype digital systems since it's designed to be reconfigurable by the customers, which is valuable to use in this case of designing, developing, and testing the the RISC-V architecture.

1.1 Academic Implementations

The RISC-V architecture has been developed and implemented many times for academic ventures, as searching google for "Senior Design: RISC-V FPGA" will produce numerous implementations.

1.2 Commercial and Open-Source Cores

Companies have been using simple embedded style RISC-V processors with custom instructions on mainboards for low power and customizable implementation for a while, but now companies are starting to roll out more and more advanced consumer/developer focused RISC-V offerings such as the ATX motherboard style offering such as the Milk-V Jupiter, and even laptop implementations like the DeepComputing RISC-V Mainboard for the Framework laptop. This demonstrates the high configurability that makes RISC-V attractive for various applications.

1.3 Configurability

RISC-V by being open source is inherently customizable for companies or individuals to add the features they need, custom instructions can radically improve performance, and shows that RISC-V is more than just an open source pipedream , but a radically new way of designing hardware software development flows.

For this project a laundry list of techniques, skills, and tools are required for different stages.

As for techniques purely related to design, a strong sense of debugging is needed in order to look for common sticking points for processor pipeline functionality, VHDL design ideas and concepts, abstraction of ideas, breaking down problems, and a huge work ethic to keep pushing past blockers, bad performance, and logical fallacies.

For techniques related to the design software itself, you will need an extreme sense of patience, openness to learn antiquated GUI design, flexibility for software that seems to break itself if you look at it wrong or use it too fast, strong skill set of searching the internet for related questions and solutions especially when it comes to the XILINX family of tools used for designing hardware/software for OEM offerings.

The tools we utilized for the simulations were Adlec's built in simulation, Professor Milders Vivado Guide, Online RISC-V compilers and syntax checkers, and the RISC-V ISA manual.

The tools we used to get the hardware done for the RISC-Vivado/Vitis project were all professor milders' guides along with a huge help from the professor himself, with a mix of using the Vivado Automation Connection and example hardware/software projects which were built into the software.

The tools we used for the final expanded project (The NPU piece) were a huge part the community/open source offering, Mobilenetv2 was open source open weighted and provided for all to use and modify, the video pipeline design and hardware initialization were provided by Digilent as one of their demonstration offerings for the Zybo Z7, the supportive CNN files used by Mobilenetv2 were generated by Matlab Simulink 2022a, we also received significant help from external developers from Aimagin who specialize in developing for the Zynq 7000 and other microcontrollers, who helped us to get our model compiling for our Zynq 7000 system, as many hours were spent discussing differences in hardware, setup, and portability for adapting our system of the RISC-V core alongside their matlab simulink/vitis interposer. Utilizing the Aimagin suite of "WAIJUNG" matlab simulink libraries, functions, project setup, and vitis code generation we were able to interface the PCAM 5C with the ZYBO Z7 and running through an on board image recognition project.

2. Statement of Work and Testbed for Verification

This project aims to design and implement a pipelined RISC-V processor on an FPGA. The processor has been implemented with the RV32IM ISA and has been routed for placement on the bare metal environment. For additional features, we have also fully integrated the Vivado hardware for UART control of the internal processor system, as well as for use of multi-run instruction memory reprogramming and execution using Vitis SDK library calls.

We have ported the processor hardware to a 3X larger FPGA/ Development board, the ZYBO-Z7-7020. We have done this to support our secondary goal which was the implementation of a Convolutional Neural Network running on board for recognition of images/objects, which required the use of Vivado Video IP blocks, as well as additional arm performance to run the AI model. In order to get this working, we reached out to an experienced and talented group named Aimagin, located in Thailand to support our development with the Zynq-7000 series of processors. By interacting with this team and utilizing their Matlab Simulink Suite of tools we were able to construct and generate the necessary executables for the processor to be able to run our model of choice (MobileNetV2) on board. By referencing Digilent's drivers/documentation for the PCAM 5C 5mp camera, interfacing with the Aimagin team to develop the code blocks for the AI model execution, and utilizing example community video IP projects leveraging the Zybo Z7-20 Development board we leveraged multiple different points of software, ingenuity, and support to achieve a 10X complexity improvement over out initial "text-recognition" goal.

This project also successfully implemented a Toolchain executable designed and specified for our hardware. It takes care of compilation of standard C code with the GCC compiler with the appropriate flags and limitations in order to operate on our hardware platform, while also supporting instruction upload to the processor serial channels, and proper control signals to initialize and control the system. It is a full stack software support application for our hardware platform.

One key challenge that arose during the development of our onboard testbench program was accessing memory through the Vitis IDE required careful consideration. We referred to documentation provided by Professor Milder, which outlined two methods of memory access: one utilizing built-in board commands and another directly accessing processor memory addresses. We selected the direct memory addressing method, as it provided greater robustness and versatility, despite requiring more effort.

Because the memory addresses were dependent on our custom processor structure and IP configurations, the documentation served only as a general guideline. We identified the base memory addresses of each AXI IP module and determined internal offsets by studying each IP's structure. Initial address estimates were refined through a series of trial runs, modifying address offsets and observing processor behavior until the correct mappings were confirmed.

Our testbed for verification lies in the requirements of performance as laid out in the RISC-V ISA, as well as our numerous simulation and testbenches created specifically for checking our work across the development process.

3. Implementation Details / Innovation - uniqueness of proposed solution

For our project we sought out to develop a RISCV32IM ISA compatible processor which can be placed on an FPGA and run real RISC-V assembly binary code. The goal was to make the code run pipelined in order to allow the processor to run at a higher clock speed, and to be able to read back any and all results from program execution. This however would require dealing with many data hazards (Reading/Writing/Control Hazards, Branch Hazards / Predictions, etc). We also wanted to expand the project to include some onboard neural network operations to run on our arm chip to explore and integrate modern CNN approaches to AI.

We acquired our knowledge required for the following skills and techniques for writing VHDL within the ESE 382 and ESE 345 courses given at SBU during the computer engineering track. For the skills and tools, we simply utilized the plethora of information, examples of implementation as per the OPEN SOURCE nature of the RISC-V community, and any and all software tools were downloaded vai live updates or archived and documented programs.

3.1. Memory Access and Onboard Testbenches

Testing the processor we created required implementing what is known as a test bench, or in other words, a software based simulation cradle to observe and utilize the performance and accuracy of our processor in ALL stages of development. Creating and implementing this processor was a complex and iterative process. Throughout development, we encountered many challenges that required creative solutions. For each new hurdle, we would consult a variety of sources, including documentation, online references, and, when necessary, seek guidance from Professor Milder via email or during our weekly progress meetings.

Test Benching throughout all stages of this processor's development was a royal pain! In ALDEC HDL simulation was typical for what we have learned in our coursework for testbenching complex VHDL systems. However when moved to Vivado, simulation became a whole lot more challenging, as it required System Verilog syntax which none of us were familiar with, and also needed to use the Vivado built in functions and libraries to simulate higher level more complex calls such as AXI Read and Write operations. Then we needed to design ARM code specifically to run on the on board chip to properly initialize and utilize our FPGA chip. At each step we found more bugs, room for improvement, and overall helped us to better understand the multi dimensional and lengthy process implementing a system can be .

3.2. Custom Pipeline Design Features

Referencing the textbook for the main pipelined processor design was certainly useful, but there were many customization and engineered solutions we had to design to implement solutions for hazard situations, improvements for early branching, protecting memory writeback for invalid instructions in the pipeline after branches/jumps occur, and engineering the initialization/reset/running/finish stages of running the processor.

3.2.1. Early Branching

Early Branching under our implementation was not covered at all in the textbook, the control unit is forwarded all current states of memory. Given the instruction that reaches the control unit is a BRANCH type instruction, and it sees that the conditions are true for that branch, it will tell the IFID to switch its output reg to a NOP, to wipe that instruction from the pipeline, reducing stalls and increasing overall throughput.

```
-- Extract rs1 and rs2 addresses from instruction
rs1_addr <= instruction(19 downto 15);
rs2_addr <= instruction(24 downto 20);

-- Updated data selection with load awareness (in ControlUnit)
rs1_final <=
    exmem_regdata when (rs1_addr = exmem_rd and exmem_memread = '0') else  -- Only forward from non-loads
    memwb_regdata when (rs1_addr = memwb_rd) else
    rs1_data;

rs2_final <=
    exmem_regdata when (rs2_addr = exmem_rd and exmem_memread = '0') else
    memwb_regdata when (rs2_addr = memwb_rd) else
    rs2_data;
```

```
  case instruction(14 downto 12) is
    -- BEQ (000)
    when "000" =>
      if rs1_final = rs2_final then
        branch_taken <= '1';
        int_early_branch <= '1';
        int_RegWrite <= '0';  -- Prevent write-back when branching
      end if;
```

One Example of our method for EARLY BRANCHING, the control unit is able to look forward and preemptively clear the IFID buffer if the current state of the processor will in fact lead to a branch for that instruction or not. The control unit looks at all forwarded data and can determine ahead of schedule if the branch will take and save the processor from flushing 2 cycles stages worth of work. Meanwhile, the EXMEM buffer after a Branch condition takes knows to remove its data 1 cycle later so it does NOT affect the system (since data after a branch is invalid.

3.2.2. Memory Writeback After Branches

For Branch/Jump Type instructions there is a condition that any instructions after that instruction that were pre-fetched are invalid and shouldnt be run if a branch is taken. Therefore our hardware implements fixes in the EX/MEM stage. The Alu will tell the EXMEM buffer whether or not it just executed a successful branch/jump calculation. IF this condition is true the EXMEM buffer will forget the FOLLOWING cycle's write back/read data. This ensures that no invalid commits to memory slip by AFTER a change in program flow.

```
elsif rising_edge(clk) then
    -- Compute branch condition (AND Branch with JALorBRANCH)
    current_branch_condition <= Branchin and JALorBRANCH and aluzeroin;

    -- If branch condition is true, insert a bubble (NOP)
    if current_branch_condition = '1' then
        -- NOP: Clear control signals and results
        aluzero_reg        <= '0';
        aluresult_reg      <= (others => '0');
        readdata2_reg      <= (others => '0');

        MemtoReg_reg <= '0';
        RegWrite_reg <= '0';
        MemRead_reg <= '0';
        MemWrite_reg <= '0';
        Branch_reg <= '0';
        JALorBRANCH_reg <= '0';  -- Clear new signal

        rd_reg <= (others => '0');
    else
        -- Normal operation
        aluzero_reg        <= aluzeroin;
        aluresult_reg      <= aluresultin;
        readdata2_reg      <= readdata2in;

        MemtoReg_reg <= MemtoRegin;
        RegWrite_reg <= RegWritein;
        MemRead_reg <=  MemReadin;
        MemWrite_reg <= MemWritein;
        Branch_reg <= Branchin;
        JALorBRANCH_reg <= JALorBRANCH;  -- Store new input

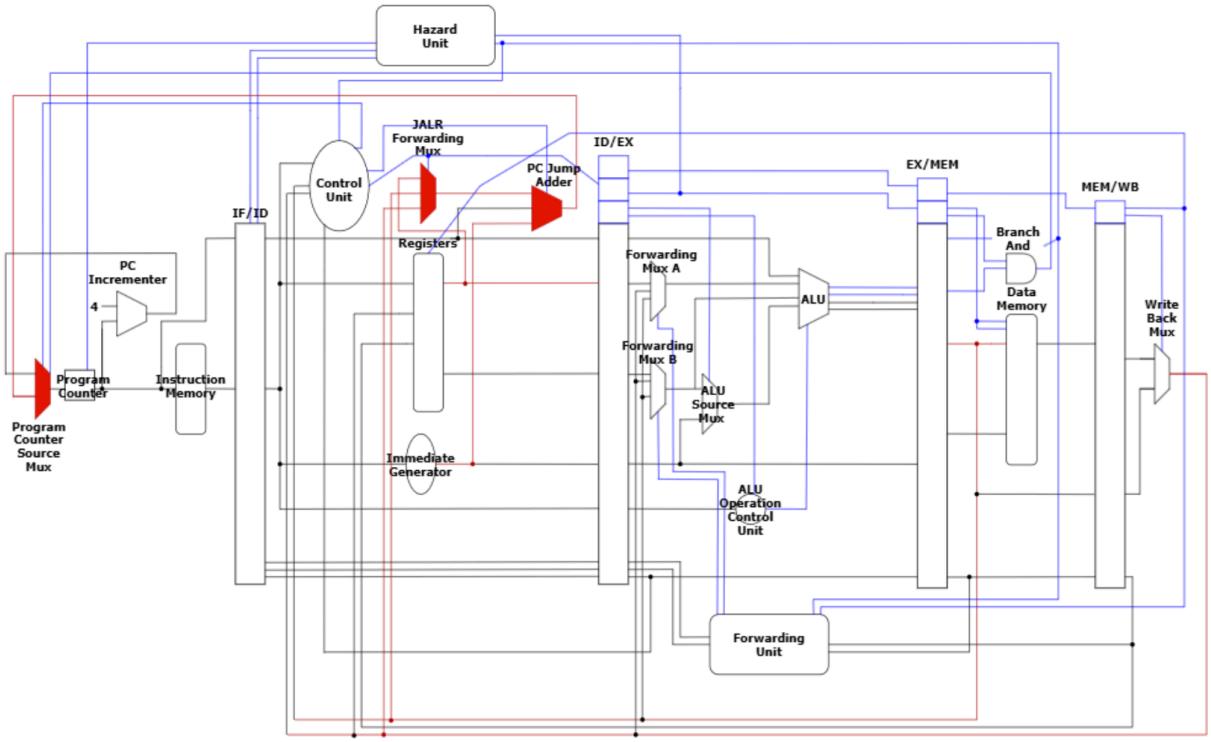
        rd_reg <= rdin;
        addrReg <= dataMemRdIn;
```

As you can see, if the EMEM detects the ALU calculated a proper branch it can remove the following cycle's results for logical consistency. We save a lot of cycles and keep our CPI relatively low. This is closely tied to our early branching logic to reap all benefits to avoid any branch effects on performance.

3.2.3. JALR

JALR works by setting the value of the PC to the value stored in a requested register plus an immediate given in the instruction. This is a relatively complex instruction, requiring several operations to occur in abnormal locations.To improve instruction throughput we perform this jump as early as possible, in the ID stage of the pipeline.

Due to this being one of the final instructions that we implemented we tried to hook into any pre-existing architecture whenever possible. Below is a diagram of the processor architecture but with the data signals related to the JALR turned red.



In order to handle the addition of the register and the immediate, we decided to connect to the adder we had already made for our JAL instruction, and simply give a new control signal to decide whether to add the register or pc with the immediate. A problem with this arises very quickly however, as the JALR instruction, when used by a compiler, is almost always immediately preceded by an AUIPC instruction to modify the source register for the jump. This causes an issue with forwarding, as the AUIPC instruction still has not been executed by the time the JALR reaches the ID stage.

We decided there were 2 ways to fix this hazard, the first of which was to connect the JALR forwarding unit to the output of the ALU, which would in theory be where the AUIPC output would be sitting until the end of the cycle. This would prevent the need for a stalled instruction, but would be a very long signal path, possibly a new critical path. We decided that with all of the timing issues we were facing at the time that adding a new critical path was not

worth the lack of a stall. So we decided to go with our other option, stalling. We decided to add a new “Pause” output signal to the hazard unit. This would look at the instruction in the ID phase, and if it was a JALR signal it would look at the register sources for both this instruction and the instruction in the EX phase. If the 2 addresses matched we would assert the Pause. The purpose of this would be to stall the back half of the processor for a cycle to give the instruction in the EX phase a cycle to run without worry of unforwarded data.

Initially we connected the Pause signal to the PC and the IF/ID registers, and forced them not to update their outputs, the idea being that if the pause was assert for just a little to long and the inputs were also frozen, it might have caused issues with the PC not updating when it should have. This ended up not being the case, and was changed so that the INPUT would not be updated if the Pause was asserted. There was however still an issue. At the moment whenever a pause was asserted, the processor would just lock up, as we had no way of telling the processor to only pause for one cycle.

The initial solution we used for this was to just set the pause signal to “Not Pause” every time the clock cycled. This worked somewhat but ended up causing several issues, especially on board. It also would lead to an issue where the processor could lock up fully sometimes and alternate between paused and unpaused forever without any actual progress being made. We realized the issue was that the IDEX register would still take the output from the IFID register as it was not pausing, then the register in the IDEX stage would match the register in the IFID stage, and it would loop forever. To fix this we initially tried shutting the output of the IFID stage, but this is where the control unit gets its input to send out its signals, so this just bricked the processor. After several other attempted solutions we ended up settling on sending the Pause signal to the IDEX register as well. The pause signal on the IDEX register would just flush the input on the next clock cycle, essentially making it see all 0’s on its input. This would allow the whole pipeline to continue as normal without making any changes to everything before it. Below is this snippet of code.

```
else    --if paused
    pcin_reg         <= (others => '0');
    readdata1_reg    <= (others => '0');
    readdata2_reg    <= (others => '0');
    immediate_reg    <= (others => '0');
    instruction_reg  <= (others => '0');

    -- Normal control signal passing
    MemtoReg_reg     <= '0';
    RegWrite_reg     <= '0';
    MemRead_reg      <= '0';
    MemWrite_reg     <= '0';
    Branch_reg       <= '0';
    ALUSrc_reg       <= '0';
    ALUOp_reg        <= (others => '0');

    -- Normal register address passing
    rs1_reg          <= (others => '0');
    rs2_reg          <= (others => '0');
    rd_reg           <= (others => '0');
end if;
```

One final issue occurred with this, which would be if the source register for the JALR instruction was R0 it would stay forever locked. To remedy this, we just added an exception to ignore R0, as it is always kept at 0 anyway, so there is no need to forward it. We then changed

the Pause to be asserted when the 2 non-zero source registers matched during a JALR and deasserted otherwise. Below is this snippet of code.

```
if is_jal_jalr = '1' then

    if idex_rd = jalrReg and not(jalrReg = "00000") then
        pause_sig <= '1';
        --ctrl_disable <= '0';
    else
        pause_sig <= '0';
    end if;

end if;
```

3.3. Processor Control Signal: START, HOLD, RESET,PAUSE

The textbook does not provide any strategy for initializing the processor. The strategy we engineered is simple and effective in the simulation/hardware sensitive aspect.

- **RESET**: A asynchronous active low signal that wipes all memory in the processor, and tells the register file to reset its STACK pointer to a custom value appropriate for our DATA MEMORY size.
- **HOLD**: A necessary 3rd signal that freezes all memory entities in the processor from updating or outputting while this signal is high. Necessary to load the instruction memory without having it output anything.
- **START**: The signal that, when HIGH, allows the Program Counter to update. Unless this signal is high the processor can execute new instructions.
- **Pause:** A signal specifically for JALR instructions to prevent possible data hazards caused by jumping in the ID stage. It freezes the Program Counter and the IFID register, while also setting the input of the IDEX register to a 0, so the next clock cycle it is passed a NOP instruction. This allows for register data that has not yet been calculated to have a cycle to be calculated for forwarding into the PC Immediate Adder.

3.4. Compiler Wrapper

The compiler is made to streamline and simplify the development process, by creating a Python wrapper around the GNU GCC RISC-V toolchain, we aim at improving the process of compiling C code into RV32IM-compatible binaries for our custom processor. The wrapper features a simple GUI interface that allows users to input C code and specify output path, abstracting away the command-line complexity and helping users quickly generate RISC-V programs. This script automates compilation by calling the appropriate riscv32-unknown-elf-gcc commands and processes both the binary and disassembled output. It extracts clean and raw

assembly without pseudo-instructions, making it customized for our specific RISC-V architecture instruction memory.

The following flags are used to tailor the GCC output for our pipeline's specific constraints:

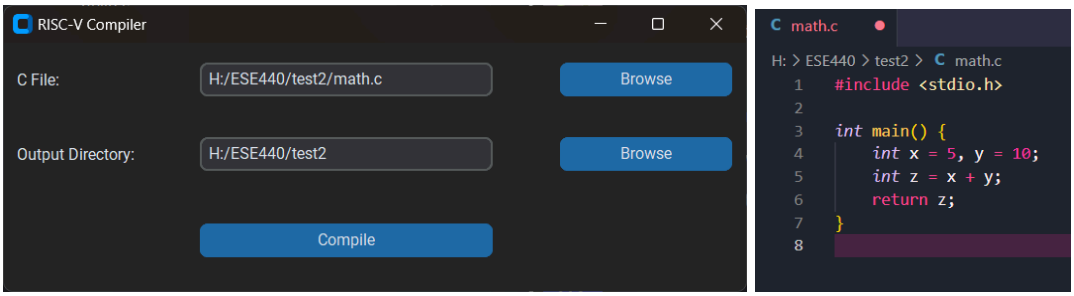
- -S: Stops after compilation to produce assembly only
- -march=rv32im: Targets the RV32IM ISA (no division support)
- -mabi=ilp32: Uses 32-bit integers, longs, and pointers
- -fomit-frame-pointer: Saves registers by omitting frame pointers
- -ffreestanding: Compiles for bare-metal (no standard runtime)
- -nostdlib: Omits linking with standard libraries like libc
- -mno-relax: Disables linker instruction relaxation
- -mno-div: Explicitly removes division instructions to avoid unsupported behavior

Additionally, when generating the final machine code and disassembly for loading into our custom instruction memory, the following were also used:

- -O0 (no optimization): Ensures predictable assembly output directly mapping to C code, avoiding unexpected optimizations that could interfere with hardware verification.
- objcopy and objdump tools: Used to extract binary memory images and clean disassembly listings without pseudo-instructions (objdump -d and objdump -D).

Once compiled and formatted, the binary payload is wrapped in the expected S[payload]P format and sent via UART using TeraTerm. This automated flow significantly reduces iteration and the chance for user error during code development and testing.

Below are screenshots of the compiler in action:



GUI for the RISC-V compiler wrapper & C code to be compiled by the compiler

math.c	3/20/2025 11:51 PM	C Source File
math.elf	3/20/2025 9:55 PM	ELF File
math.s	3/20/2025 9:55 PM	S File
math_disasm.s	3/20/2025 9:55 PM	S File

Output generated by the compiler (math.elf (binary file), math.s (assembly file), & math_disasm.s (disassembled binary file - assembly file))

```
        .file   "math.c"
        .option nopic
        .option norelax
        .attribute arch, "rv32i2p0_m2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .align  2
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-16
        li      a5,5
        sw      a5,12(sp)
        li      a5,10
        sw      a5,8(sp)
        lw      a4,12(sp)
        lw      a5,8(sp)
        add     a5,a4,a5
        sw      a5,4(sp)
        lw      a5,4(sp)
        mv      a0,a5
        addi    sp,sp,16
        jr      ra
        .size   main, .-main
        .ident  "GCC: (GNU) 10.1.0"
```

```
H:/ESE440/test2\math.elf:     file format elf32-littleriscv


Disassembly of section .text:

00010054 <main>:
   10054:       ff010113                addi    sp,sp,-16
   10058:       00500793                addi    a5,zero,5
   1005c:       00f12623                sw      a5,12(sp)
   10060:       00a00793                addi    a5,zero,10
   10064:       00f12423                sw      a5,8(sp)
   10068:       00c12703                lw      a4,12(sp)
   1006c:       00812783                lw      a5,8(sp)
   10070:       00f707b3                add     a5,a4,a5
   10074:       00f12223                sw      a5,4(sp)
   10078:       00412783                lw      a5,4(sp)
   1007c:       00078513                addi    a0,a5,0
   10080:       01010113                addi    sp,sp,16
   10084:       00008067                jalr    zero,0(ra)
```

math.s output on the left (highlighted is a pseudo instruction)
math_disasm.s output on the right (highlighted is the disassembled pseudo instruction)

4. Constraints and Applicable Standards

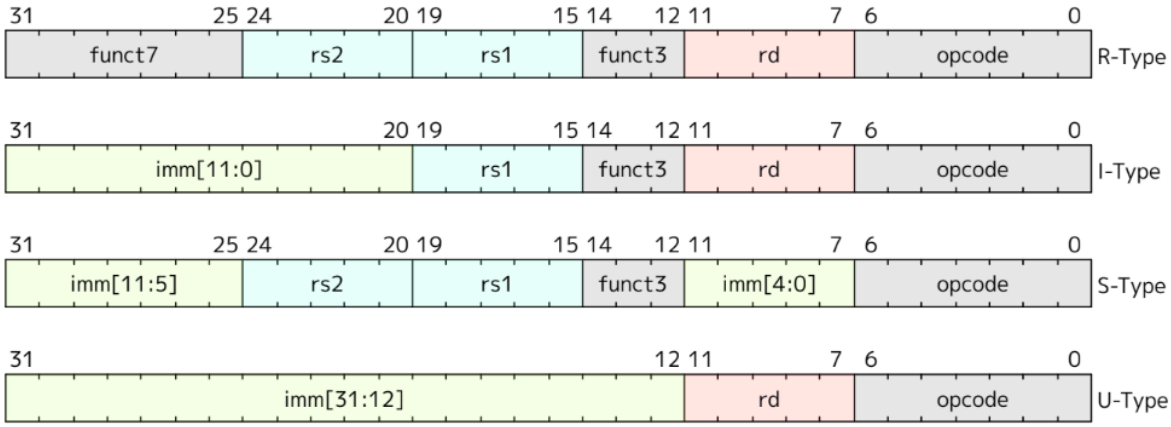
4.1. Compliance with the RV32IM ISA

The main constraints we had when designing our processor were following the RV32IM ISA, making sure our processor would run on the FPGA we used, and making sure it would fit on the FPGA.

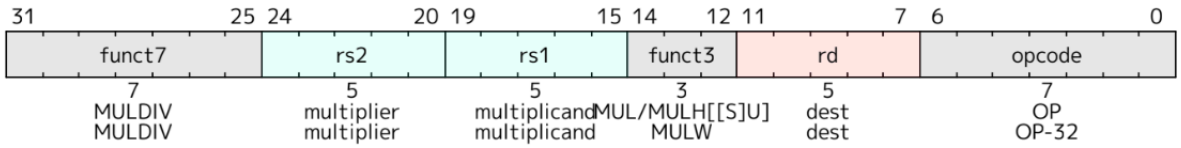
While we will be providing a condensed overview of the ISA, the precise constraints for the RV32IM ISA, and all other RISC-V ISA's can be found on the official website (RISC-V International, 2023)

What RV32IM really means is that we follow the RV32I (or Risc-V Integer base) instruction set along with having multiplication instructions included. According to their official website "RV32I was designed to be sufficient to form a compiler target and to support modern operating system environments. The ISA was also designed to reduce the hardware required in a minimal implementation." In other words, the entire purpose of this ISA is to have a simpler but fully effective processor standard that modern operating systems could be run on. In fact, RISC stands for "reduced instruction set computing." These reduced instructions lead to a simpler, and more modular overall processor design. Many of the hardware constraints within the standard, deal with the required structure for these instructions, for example, within the ISA there are 4 different types of instructions, they are R/I/S/U type instructions. Below is a screenshot of how these instruction types must be structured, along with another screenshot of how MUL operations must be structured.

The RISC-V ISA keeps the source (*rs1* and *rs2*) and destination (*rd*) registers at the same position in all formats to simplify decoding. Except for the 5-bit immediates used in CSR instructions (Chapter 7), immediates are always sign-extended, and are generally packed towards the leftmost available bits in the instruction and have been allocated to reduce hardware complexity. In particular, the sign bit for all immediates is always in bit 31 of the instruction to speed sign-extension circuitry.



13.1. Multiplication Operations



We can then use these constraints to design the instruction decoding stage of the processor, which will send all of the relevant information from these instructions to the relevant registers they need to go. But the instructions are not the only constraints, for example, a more fundamental constraint to the ISA is that there must be 32 general purpose registers, x0-x31, and of those registers, x0 must be hardwired to ground, forcing it to always be a 0.

There are tons of simple constraints like these that must be followed in order for the processor to work properly when provided code written using this ISA. Another constraint like this involves the “JAL” and “JALR” instructions. These instructions allow the user to add to the program counter of the processor. The program counter (or PC) is a less accessible register, which must be incremented by 4 every cycle, unless told otherwise. This register is responsible for the advancement through coded instructions.

Part of the ISA states that all instructions must be saved in instruction memory, 4 bytes apart. This combined with the incrementing of the PC by 4 means the PC should always be pointing to the start of an instruction. However, JAL and JALR allow any number to be added to the PC, if the number to be added was 2, this would put the PC in the middle of 2 instructions,

causing completely unpredictable behavior. Due to this, one constraint for these instructions is to mask away the last 2 bits of the sum of the PC and what we wish to increment it by, before Placing it back into the PC.

4.2. Hardware Constraints

In addition to this we had to ensure that the processor would properly run and fit on the board. Ensuring everything fit properly was easy enough, as within the Vivado IDE you can see how much space the design is taking up on the board. During initial designs the processor was taking up around 150% of the available LUT ram memory, so some changes had to be made. Otherwise everything was fine there. The next thing was ensuring that everything actually ran properly. To do this we designed several test programs which tested as many different possible errors as we could. We then loaded them onto the board and ensured they were outputting the expected results.

4.2.1 Hardware Considerations/Upgrade Choice

For our initial development platform, the MiniZed from AVNET, we were quite tight on usage (in terms of LUTRAM, LUTS, and such) placing the size of our design and since we had built the processor in terms of combinational reads/writes, we could not utilize the BLOCK RAM which Vivado offers for large memory mappings. Looking forward to the later stages of our project we also wanted to implement machine vision and other more complex projects and solutions on board, and there was zero documentation on this board, as well as a complete departure of support from the company who built it. We began research on a new board to use with larger FPGA to support more LUTRAM in the design for our larger data memory and instruction buffer, as well as the support from the community, better documentation, example projects, as well as video content hardware capabilities to aid us in the intake of visual data when we began to tackle the CNN visual addition of the project.

4.3 Fitness Criteria

The tradeoffs involved in not implementing more of the RISC-V 32 ISA or even a 64 bit implementation of the processor, we judged that the tradeoff of time development complexity and space on an FPGA for a 100% increase in bits, was just not worth it.

5. Best/Worst Case Design, Theoretical Analysis, Simulations

5.1 Best Case Design

The best-case scenario as we originally saw for the processor was to have made a pipelined processor with full RISC-V-32-I-M (32-bit base integer plus multiplication) compatibility. We would have liked to implement all of Division into our processor as well but given the time constraints, and development time to get the proper hardware/software config for the Neural Processing for MobileNetV2 running with Vivados Video IP took significantly longer than expected.

Despite this, we did in fact get all of RV32IM working in a 5 stage pipelined design, so we would say we achieved a majority of the goals of the project as well as the extra ML-Vision based goals we originally set out. We originally intended to just get text/digit recognition working but we set our sights on a more complex and interesting goal to get object recognition working.

5.2 System formulation process and identification of key components.

The system boundaries we set were to limit the ISA range we supported as well as the number of DATA MEMORY and INSTRUCTION MEMORY indexes and structure we supported as expanding the range of either of these would increase the LUTRAM used in our assumed Combinational and not sequential read system. Because of the way our pipeline assumes data flows and is accessed, the structures must be created with LUTRAM.

The critical components identified for our system were all pipeline registers, arithmetic units, memory units, recommended control units, and all other recommended units from the 5 stage pipeline design in the textbook (*Computer Organization and Design RISC-V Edition*).

Our methodology for analyzing best and worst-case scenarios was just a matter of what combination of complexity, performance, and what compiled and synthesized within the tools, as this was quite a complicated matter of getting something working rather than nothing.

The theoretical analysis and simulations were incorporated into examining the CPI and simulation rate of the pipeline design. After we successfully built the processor and were examining the throughput of the processor given hazards and pipeline considerations.

5.3 Potential solutions for various key components

Some different conceptual approaches to our system could have been a more simplified AXI interface, as we could've combined DATA memory, Registers, and Instruction buffer to simplify access readback and processor operations. We could have also, for the sake of the CNN, taken a simpler approach and designed a custom CNN specified arithmetic unit rather than utilizing c code on the arm.

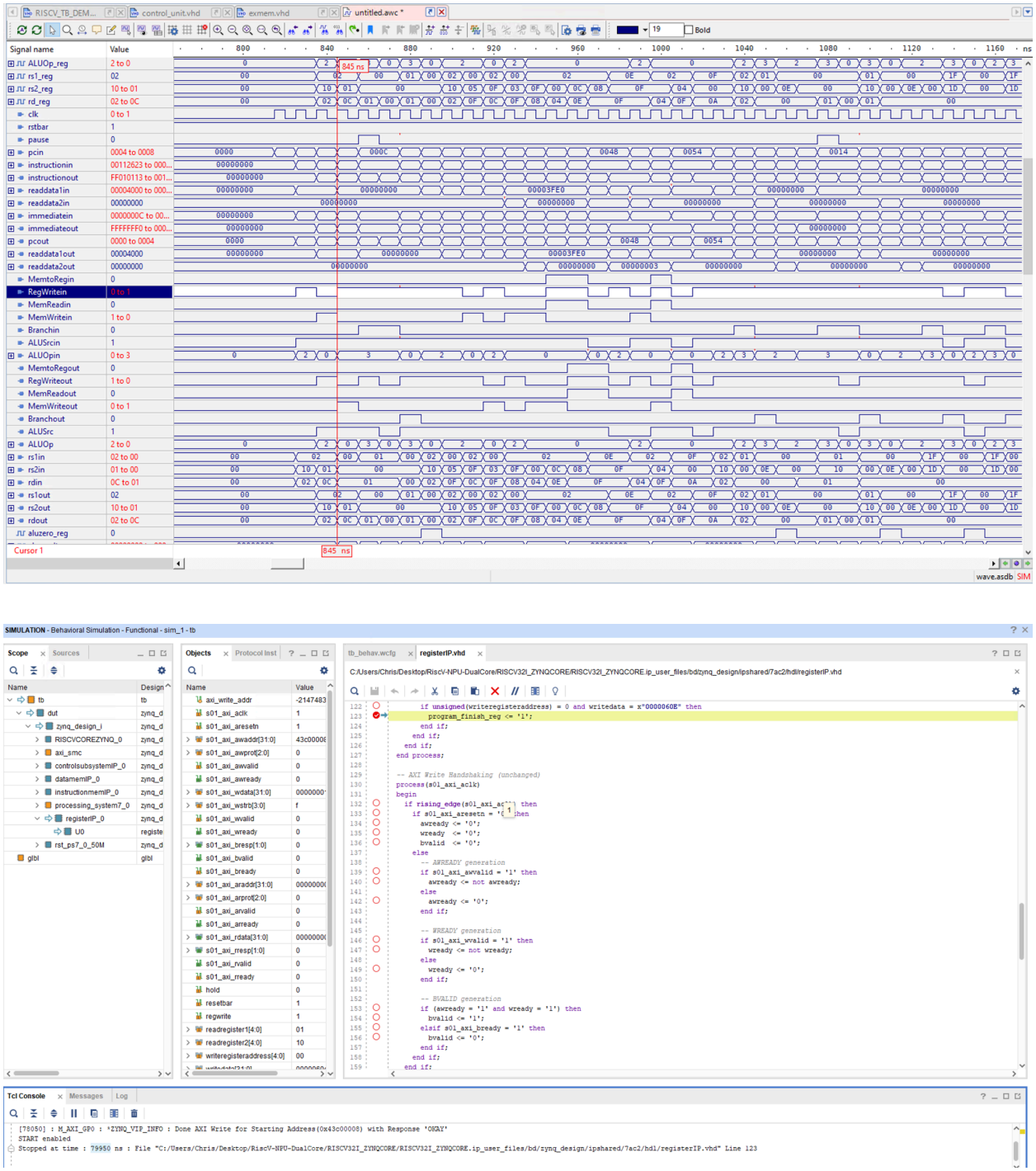
These different solutions would have performed much worse however as we would not have been able to design something complicated enough to do image processing PURELY in VHDL hardware both with our knowledge base and the time allotted for the development of the processor.

Considering the trade-offs between the different design options of using BRAM in our design would have required extensive reworking of the pipeline design philosophy, delay work, and pipeline behavior within VIVADO, we decided for the sake of the time allotted and getting a product out the door to ship the processor with less efficient and slower memory in the form of LUTRAM.

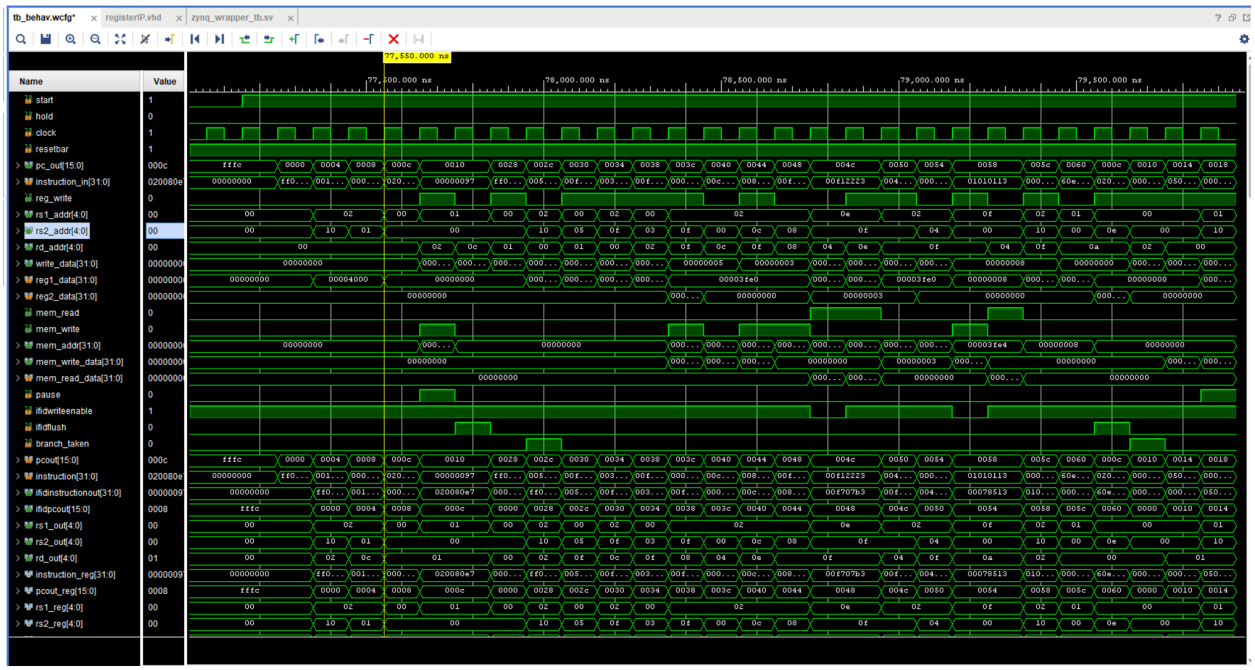
5.4 Simulations

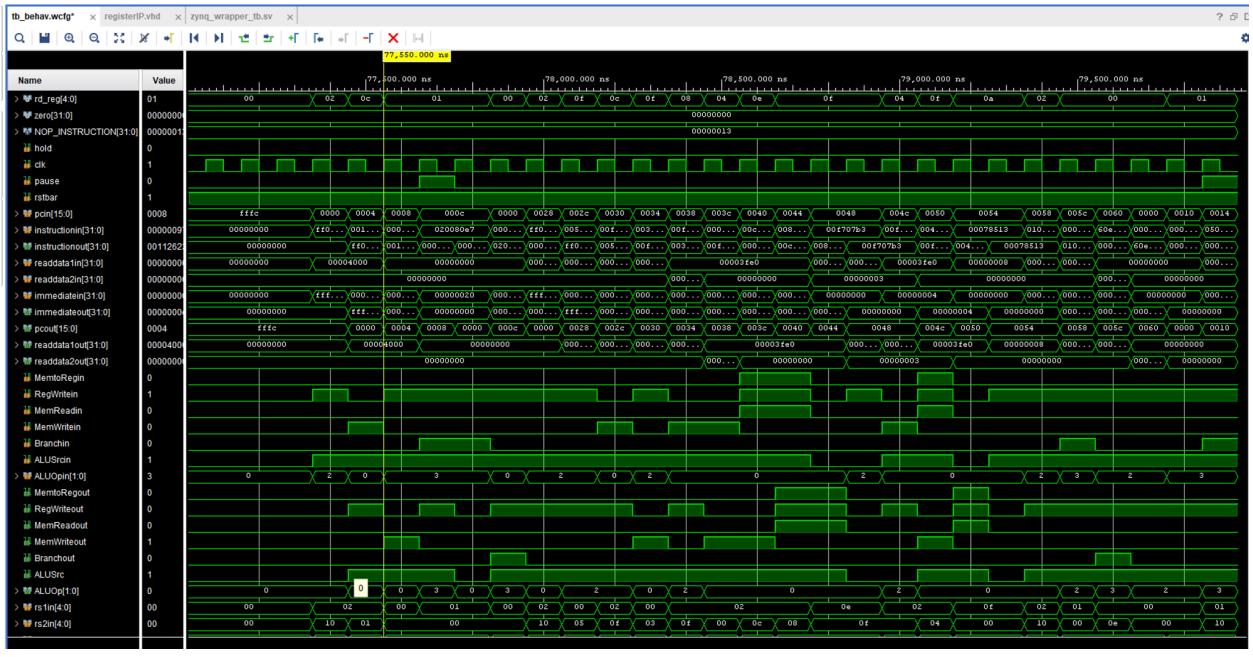
Simulations are essential for validating our processor's behavior and confirming the correctness of our design.

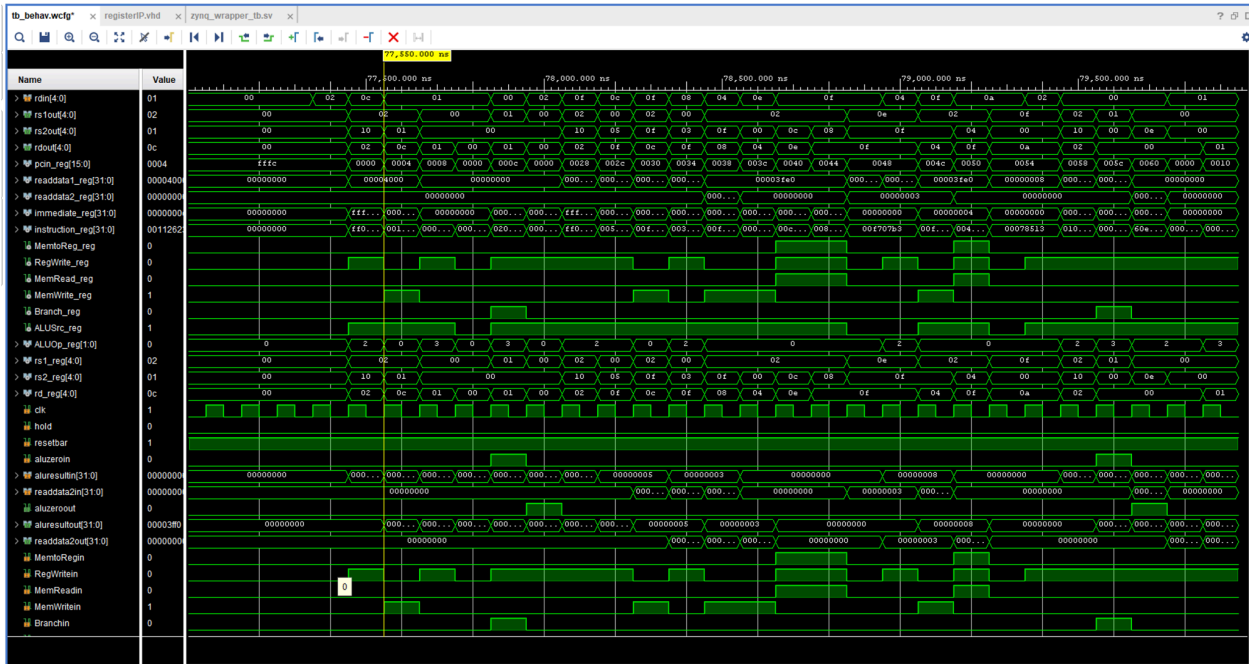
Snippet of Aldec Simulation for our 3+5 GCC Compiled program

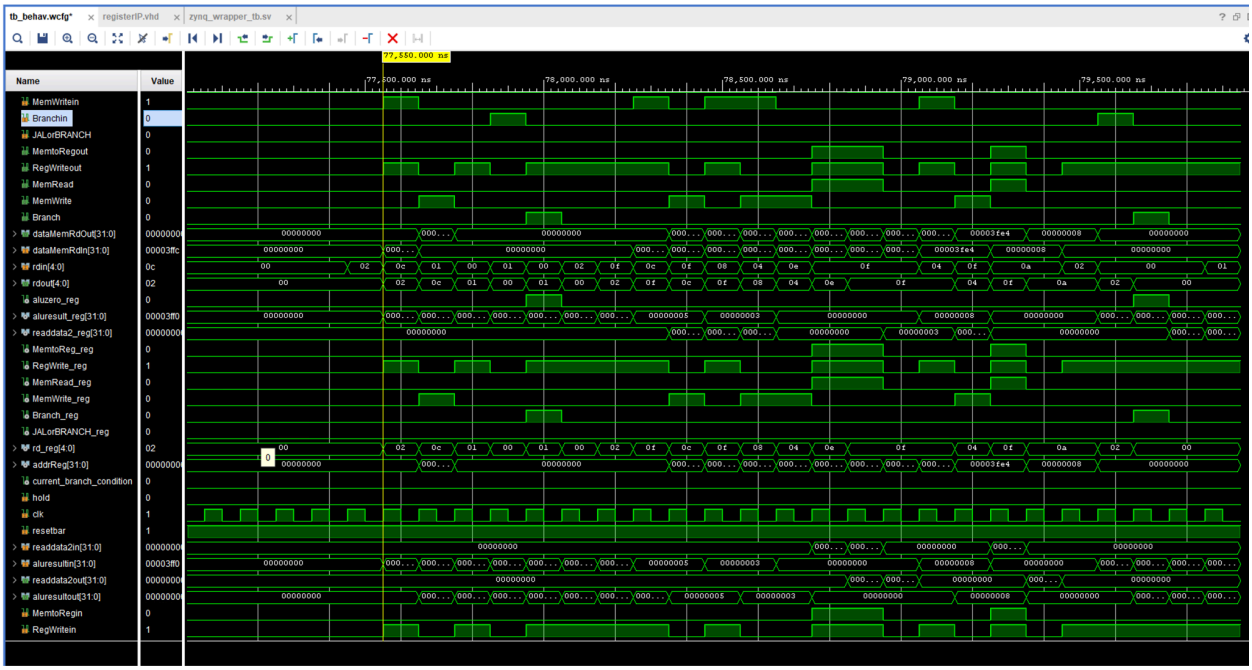


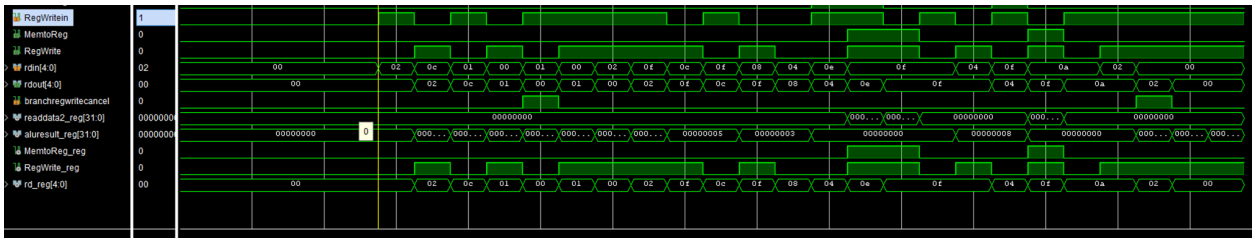
Entire Pipeline Register View of the Vivado Simulation Window


tb_behav.wcfg*
registerIP.vhd
zynq_wrapper_tb.sv
77,550.000 ns
Name
Value
start
hold
clock
resetbar
pc_out[15:0]
instruction_in[31:0]
reg_write
rs1_addr[4:0]
rs2_addr[4:0]
rd_addr[4:0]
write_data[31:0]
reg1_data[31:0]
reg2_data[31:0]
mem_read
mem_write
mem_addr[31:0]
mem_write_data[31:0]
mem_read_data[31:0]
pause
ifidwriteenable
ifidflush
branch_taken
pcout[15:0]
instruction[31:0]
ifidinstructionout[31:0]
ifidpcout[15:0]
rs1_out[4:0]
rs2_out[4:0]
rd_out[4:0]
instruction_reg[31:0]
pcout_reg[15:0]
rs1_reg[4:0]
rs2_reg[4:0]


tb_behav.wcfg*
registerIP.vhd
zynq_wrapper_tb.sv
77,550.000 ns
Name
Value
rd_reg[4:0]
zero[31:0]
NOP_INSTRUCTION[31:0]
hold
clk
pause
rstbar
pcin[15:0]
instructionin[31:0]
instructionout[31:0]
readdata1in[31:0]
readdata2in[31:0]
immediatein[31:0]
immediateout[31:0]
pcout[15:0]
readdata1out[31:0]
readdata2out[31:0]
MemtoRegin
RegWritein
MemReadin
MemWritein
Branchin
ALUSrcin
ALUOpin[1:0]
MemtoRegout
RegWriteout
MemReadout
MemWriteout
Branchout
ALUSrc
ALUOp[1:0]
rs1in[4:0]
rs2in[4:0]


tb_behav.wcfg*
registerIP.vhd
zynq_wrapper_tb.sv
77,550.000 ns
Name
Value
rdin[4:0]
rs1out[4:0]
rs2out[4:0]
rdout[4:0]
pcin_reg[15:0]
readdata1_reg[31:0]
readdata2_reg[31:0]
immediate_reg[31:0]
instruction_reg[31:0]
MemtoReg_reg
RegWrite_reg
MemRead_reg
MemWrite_reg
Branch_reg
ALUSrc_reg
ALUOp_reg[1:0]
rs1_reg[4:0]
rs2_reg[4:0]
rd_reg[4:0]
clk
hold
resetbar
aluzeroin
aluresultin[31:0]
readdata2in[31:0]
aluzeroout
aluresultout[31:0]
readdata2out[31:0]
MemtoRegin
RegWritein
MemReadin
MemWritein
Branchin


tb_behav.wcfg*
registerIP.vhd
zynq_wrapper_tb.sv
77,550.000 ns
Name
Value
MemWritein
Branchin
JALorBRANCH
MemtoRegout
RegWriteout
MemRead
MemWrite
Branch
dataMemRdOut[31:0]
dataMemRdIn[31:0]
rdin[4:0]
rdout[4:0]
aluzero_reg
aluresult_reg[31:0]
readdata2_reg[31:0]
MemtoReg_reg
RegWrite_reg
MemRead_reg
MemWrite_reg
Branch_reg
JALorBRANCH_reg
rd_reg[4:0]
addrReg[31:0]
current_branch_condition
hold
clk
resetbar
readdata2in[31:0]
aluresultin[31:0]
readdata2out[31:0]
aluresultout[31:0]
MemtoRegin
RegWritein



6. Professional/Ethical Issues and Societal Impacts

The development of this FPGA based RISC-V processor highlights the importance of accessibility and innovation for the public. By using open-source architecture such as RISC-V, we promote the accessibility to modern computing technology, especially in educational and resource-constrained environments. Open hardware and software resources potentially could reduce the need for expensive proprietary systems, making it easier for students, researchers, and developers around the world to learn, experiment, and create innovative solutions for the upcoming future demands.

From an economic perspective, customizable and reprogrammable platforms like FPGAs would allow companies and institutions to prototype and iterate developing systems more efficiently. This flexibility can lower the cost of the development and reduce time-to-market for new products. The open-sourceness nature of RISC-V also helps in the long-term support for the products or system.

Environmental responsibility is also one of the major factors to consider, especially in an age where electronic waste (e-waste) is unfortunately abundant. In fact, 62 million tonnes of e-waste were generated in 2022 alone, and the volume is rising five times faster than the rate of documented recycling (United Nations Institute for Training and Research). By using FPGAs, we can help reduce e-waste by extending the lifecycle of a single hardware platform through reconfiguration rather than replacement.

Finally, as future engineers, we have a responsibility to uphold ethical standards in the system that we design. This includes ensuring our toolchains and resources are secure, well-documented, and reliable. alongside considering how our work might be used in the future. Every engineering decision we make, from choosing open source standards to integrating machine learning on embedded platforms, has the potential to impact society years to come, and that's why it's important that we approach our work with care, intention, and a strong sense of responsibility.

7. Method and Materials

6.1. Implementation Challenges and Improvements

While implementing memory within the processor we ran into issues using BRAM. The main issue we encountered was that BRAM is fully sequential, while most of our initial designs assumed combinational memory access. This led to several issues with control signals not updating stages of the pipeline properly and most of the pipeline timings to be thrown off completely. This along with tight time constraints led to the decision to implement most of the memory using LUTRAM. While using LUTRAM does provide for a simpler implementation, it does take up a much larger amount of space on the board. Depending on what application this processor would be used for, and what other changes would need to be made, switching to BRAM would be a likely first improvement. In order to do this we would need to rethink the pipeline behavior, cycle to cycle, to ensure the data flow would be ready after each clock cycle WHILE also allowing for sequential reads/writes to be tolerated by the pipeline.

There are also memory corruption issues within the ZYNQ that occur on program startup when running the CNN for object recognition. As we discussed previously, possible solutions to these problems would be to add extra latency at program start in order to give all pieces of our processor time to initialize and stabilize before running the MobileNetV2 model.

We also encountered issues with our compiler. In order to save time on the compiler we decided to use an open source disassembly program to convert C code into assembly language, and then convert it from there into hex. However, due to the way the disassembler operates on different sections of the prewritten code, global variables are discarded in compilation. This was discovered with only weeks left on our deadline, and thus we have decided to simply not support global variables, however a fix for this would be to simply write our own compiler from scratch, which shouldn't be too hard at this point, a little time consuming at worst.

8. Testbed for Verification

5.1. Verification Strategy

For our verification strategy we approached ISA compatibility as well as runtime/pipeline compliant through progressively more complicated programs, first hand assembled code to test results, than series of hazard based instructions, then compiled C code from GCC, then based on professor Milder's recommendations we started constructing C code with more complicated structure, variable initializations, coding constructs, and more. This method and flow of confirming our RISC-v ISA support and compatibility in the pipeline gave us opportunities to slowly introduce more and more complicated and niche pipeline hazards to debug and find program flow issues.

For our test bench, we created various programs to test/stress our pipeline and hazard support. These included:

- TEST 1: LOAD and ADD Forwarding
- TEST 2: R-Type Arithmetic Operations
- TEST 3: I-Type Arithmetic and Logical Operations
- TEST 4: Data Hazards
- TEST 5: Branch and Jump Instructions
- TEST 6: Modulo Program (47/5)
- TEST 7: GCC Compiled 3+5 Program

The test suite combines hand-assembled programs and a GCC-compiled C program to thoroughly evaluate our implementation. These individually set up the processor to execute operations in hazardous and control signal heavy situations dealing with program flow, delays, flushing, bubbles, pausing and all other situations engineers deal with in designing processors in a pipeline method. We chose/crafted these in order to test the range of RISC-V 32-IM required instruction types exhaustively (R, I, S, B, U, J) within our pipeline. As well as associated hazards when running combinations of them back to back in the pipeline.

5.2. Debugging Methodology

We utilized debugging within VIVADO Simulation with VIP System Verilog calls within the simulation software, simulation test benches within VHDL simulation of Aldec Student Edition, and then on Bare Metal Using test programs loaded in memory and running the actual FPGA. We have simulated, tested, and improved every step of the way following professor Milders guidance to catch complicated issues early, and being as thorough as possible with testing.

Debugging was a process of utilizing cycle by cycle reports within Aldec and Vivado to track pipeline progress, control signals, breakpoint tracking, data writeback, and cycle to cycle conformity. We would Assemble a program by hand and track which data was reaching the registers, pipeline buffer registers, the ALU, Program Counter, and Data Memory and compare it to the expected results and data fields which should be appearing there. We utilized highly detailed diagrams and pipeline maps we drew to diagnose and track how the program was

proceeding. This combined with the RISC-V ISA helped us to understand the gaps in our knowledge, the improper implementation pieces of the processor, and then a huge amount of creativity and problem solving got us across the finish line to ultimately have a compiled code ready and compatible processor.

9. Final Implementation and Design Goal

7.1. Design Specifications

The original project specifications were as follows:

- ISA: RV32IM (32-bit base integer + multiplication/division)
- Microarchitecture: Pipelined design (5-stage typical), hazard detection, static branch prediction
- Implementation: HDL (Verilog/VHDL/SystemVerilog), FPGA synthesis (Xilinx Vivado), bare-metal execution
- Toolchain: GCC-based C compilation, FPGA program loading/debugging infrastructure
- FPGA Integration: On-board memory management, I/O interfaces for program input/results output
- Validation: Testbenches, benchmark programs (performance metrics: CPI, clock speed)
- Extended Goals: Optional RV32IMV vector extensions, cache integration, custom instructions, ASIC cost analysis
- Evaluation: Power/area analysis, runtime measurements vs. theoretical CPI

7.2. Achievements

We successfully achieved the following design objectives:

- Implemented a RV32IM pipelined processor (excluding division) with hazard detection and early branch prediction.
- Developed the processor in VHDL, synthesized it using Vivado, and validated it with bare-metal execution.
- Successfully compiled and executed custom C code using the RISC-V GCC toolchain.
- Integrated Serial Control through the Zynq ARM core, allowing dynamic loading of the instruction buffer over UART without requiring board reimaging. This feature allowed the processor to run, clear, and display results directly from the FPGA.

For validation, we verified the processor's operation and compatibility through:

- VHDL simulation

- Vivado simulation
- Bare-metal execution of custom C programs

7.3. Extended Goals

As part of our extended goals:

- We implemented Google's MobileNetV2 object recognition CNN on the ARM core of the Zynq 7020 with the help of Aimagin's Waijung Matlab Simulink libraries and support. This CNN interacted with image data captured by the PCAM5C camera to predict the objects the camera was seeing.
- We built a complete RISC-V toolchain capable of compiling and automatically uploading code to the Zybo Z7 development board for FPGA execution, all through a UART/Serial Communication Terminal.

7.4. Evaluation and Performance

Based on the provided data from the Vivado hardware report, our implementation utilized approximately 32.1% of available LUTs, 31.47% of LUTRAM, 11.89% of flip-flops, and minimal BRAM (7.5%) and DSP (1.36%) resources on the Zynq 7020 FPGA. The design faced timing constraints with only 3 endpoints failing and a Worst Negative Slack of -2.45ns at the RISC-V Core's 10MHz clock domain. Overall, the implementation maintained moderate resource utilization.

In order to measure our processor's performance, we loaded a test program into the hardware simulation of Vivado. We then stepped through the program until our custom "Stop" instruction (Addi, x0, x0, 1550) had made its way through the pipeline , tracking the number of cycles as it ran. Our processor completed the 24th instruction of the program (the STOP instruction) in exactly 24 clock cycles, achieving perfect throughput.

We calculated the Cycles Per Instruction (CPI) by dividing the number of cycles by the number of instructions executed. Using our most complex test program, which includes function calls, memory operations, and data dependencies, we achieved a CPI of 1.0 (24 cycles / 24 instructions = 1.0 CPI). This represents the theoretical ideal for a pipelined processor.

Execution Time = # of cycles × Seconds per cycle = 24 cycles × 100ns/cycle = 2.4 μs on Vivado's hardware simulation.

Throughput = 10 MIPS (Million Instructions Per Second)

The CPI of 1.0 indicates that our pipeline is handling hazards with optimal efficiency. Our implementation successfully manages data dependencies through forwarding and control hazards through efficient branch/jump handling, allowing the processor to execute one instruction per cycle without stalls. For our 5-stage pipeline executing complex code with function calls,

memory operations, and various hazards, achieving the ideal CPI demonstrates excellent hazard management.

While our 10 MHz clock frequency provided adequate performance, there remains room for improvement. Future optimizations could target memory operations (LUTRAM access times), more efficient control signal generation, and optimized arithmetic execution paths. Additionally, addressing the timing constraints identified in our analysis (WNS of -2.45 ns) would enable higher clock frequencies, further improving overall performance.

10. Results & Discussion

In order to get any results from this project we had to design several different testbenches. For this we used Aldec simulations and the Vitis IDE. For this we first made several test programs, each of which had known expected results. We would then load these programs onto the Aldec simulated board and verify the final outputs after a test run. Assuming all outputs were as desired we would move to testing on board. Using Vitis, we were able to design another test benching program to be loaded onto the board, then we could then run the sample code on the board and ensure the final outputs are lining up with what they should appear as. The sample code was written in C and converted into RISC-V machine language using the existing compiler by making a wrapper to customize the compiled code. This allowed us to determine for certain that the board was behaving as intended. Each time we implemented something new onto the board, we would move through this cycle until every test was passed. All other instructions are working completely as intended.

RISC-V Test Programs

TEST 1: LOAD and ADD Forwarding

- `addi x1, x0, 42    # x1 = 42`
- `sw x1, 12(x0)      # Store 42 at address 12`
- `lw x2, 12(x0)      # Load from address 12 into x3`

Final State:

- `Memory = 42`
- `x1 = 42`
- `x2 = 42`

TEST 2: R-Type Arithmetic Operations

- `addi x1, x0, 10    # x1 = 10`
- `addi x2, x0, 20    # x2 = 20`
- `add x3, x1, x2     # x3 = x1 + x2 = 30`
- `sub x4, x2, x1     # x4 = x2 - x1 = 10`
- `mul x5, x1, x2     # x5 = x1 * x2 = 200`

Final State:

- `x1 = 10`
- `x2 = 20`
- `x3 = 30`
- `x4 = 10`
- `x5 = 200`

TEST 3: I-Type Arithmetic and Logical

- `addi x1, x0, 15    # x1 = 15`
- `slli x2, x1, 2     # x2 = 15 << 2 = 60`
- `srli x3, x2, 1     # x3 = 60 >> 1 = 30`
- `srai x4, x3, 1     # x4 = 30 >> 1 = 15 (arithmetic)`
- `andi x5, x1, 15    # x5 = 15 & 15 = 15`
- `ori x6, x1, 31     # x6 = 15 | 31 = 31`
- `xori x7, x1, 15    # x7 = 15 ^ 15 = 0`

Final State:

- x1 = 15 (0x0F)
- x2 = 60 (0x3C)
- x3 = 30 (0x1E)
- x4 = 15 (0x0F)
- x5 = 15 (0x0F)
- x6 = 31 (0x1F)
- x7 = 0 (0x00)

TEST 4: Data Hazards

```
addi x1, x0, 10    # x1 = 10
addi x3, x0, 20    # x3 = 20
mul x4, x1, x3     # x4 = x1 * x3 = 200
```

Final State:

- x1 = 10
- x3 = 20
- x4 = 200

TEST 5: BRANCH AND JUMP

```
addi x1, x0, 10    # x1 = 10
addi x2, x0, 20    # x2 = 20
beq x1, x2, label  # Won't branch (x1 ≠ x2)
bne x1, x2, label  # Branch to label (x1 ≠ x2)
addi x3, x0, 30    # Skipped
label:
jal x4, target     # Jump to target, save PC+4 in x4
addi x5, x0, 40    # Skipped
target:
addi x6, x0, 50    # x6 = 50
```

Final State:

- x1 = 10
- x2 = 20
- x3 = 0 (skipped)
- x4 = 20 (return address)
- x5 = 0 (skipped)
- x6 = 50

TEST 6: MODULO PROGRAM (47/5)

```
addi x10, x0, 47   # Load dividend
addi x11, x0, 5    # Load divisor
addi x12, x0, 0    # Initialize quotient
addi x13, x10, 0   # Copy dividend
loop:
sub x13, x13, x11  # Subtract divisor
blt x13, x11, end  # Exit if remainder < divisor
addi x12, x12, 1   # Increment quotient
jal x0, loop       # Repeat
End:
```

Final State:

- x10 = 47 (dividend)
- x11 = 5 (divisor)
- x12 = 8 (quotient - 1)
- x13 = 2 (remainder)

TEST 7: GCC Compiled 3+5 Program

```
# _start:
addi sp, sp, -16   # Allocate stack frame
sw ra, 12(sp)      # Save return address
auipc ra, 0x0      # Set up ra for call
jalr ra, 32(ra)    # Call main
auipc ra, 0x0      # Set up ra for call
jalr ra, 80(ra)    # Call finish
addi zero, zero, 0 # NOP
lw ra, 12(sp)      # Restore ra
addi sp, sp, 16    # Deallocate stack
jalr zero, 0(ra)   # Return
```

```
# main:
addi sp, sp, -16   # Allocate stack frame
addi a5, zero, 5   # a5 = 5
sw a5, 12(sp)      # Store 5 on stack
addi a5, zero, 3   # a5 = 3
sw a5, 8(sp)       # Store 3 on stack
sw zero, 4(sp)     # Initialize result to 0
```

- `lw a4, 12(sp)      # Load 5 into a4`
- `lw a5, 8(sp)       # Load 3 into a5`
- `add a5, a4, a5     # a5 = a4 + a5 = 8`
- `sw a5, 4(sp)       # Store result (8)`
- `lw a5, 4(sp)       # Load result`
- `addi a0, a5, 0     # Move result to a0`
- `addi sp, sp, 16    # Deallocate stack`
- `jalr zero, 0(ra)   # Return`

- `# finish (infinite loop):`
- `addi zero, zero, 1550  # NOP with comment`
- `addi zero, zero, 0     # NOP`
- `jal zero, -4           # Infinite loop`

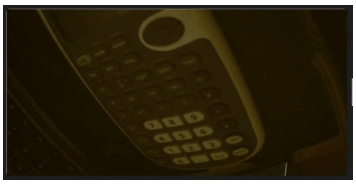
Final State:

- `x1 (ra) = 20 (return address)`
- `x2 (sp) = 16368 (stack pointer)`
- `x14 (a4) = 5`
- `x15 (a5) = 8 (result of 5+3)`
- `Memory[4089] = 8 (result)`
- `Memory[4090] = 3 (first operand)`
- `Memory[4091] = 5 (second operand)`

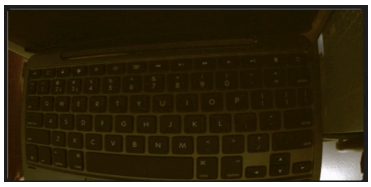
Once again we ran the following programs and saw the following results in the proper data slots. We believe this is an exhaustive and effective set of tests to show both the compatibility and the complexity of our pipelined processor.

MobileNetV2, as our model that we chose, is a lightweight convolutional neural network that achieves strong results on image classification tasks under the right circumstances. MobileNetV2 in Google's own testing and scenarios has a high accuracy, such as 94% in some experiments. In our own testing, we find that MOST of the time given the right lighting and clarity conditions, the objects and features which we show the model result in logical predictions (a calculator can be seen as a remote, phone cameras as projectors, pens as syringes) while not 100% accurate or near googles results, we may be testing with objects that don't have dedicated labels in the weight set. These results highlight MobileNetV2’s effectiveness in balancing

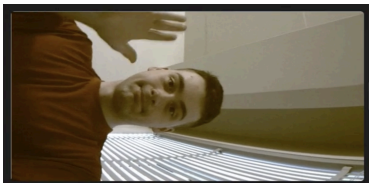
computational efficiency for running on a low power ZYNQ 7020 ARM Processor with high classification and/or logical accuracy, proving its use in limited resource/power mobile environments such as our dev board.



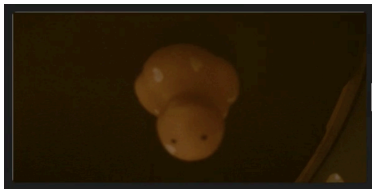
Remote control/Telephone



Waffle Iron/Laptop/Keyboard



Hamster/Mouse/Matchstick



Rubber Eraser

Our RISC-v Compatibility inherently is going to be far more limited in DATA Memory as well as total instruction memory size due to the dependency on LUTRAM baked into the combinational read, and non sequential read pace of our pipelined implementation.

Furthermore our processor must be loaded with the raw binary/hex data in its instruction memory contents, rather than loading the original ELF file as produced by many compilers. Labels are NOT supported, print statements are not supported, and there is NO I/O support that a mobile/embedded/desktop chip may have. There are also no exception/error flag reporting bits that leave the processor as referenced in the Patterson and Hennessy computer architecture book. We chose to remove this from our design for simplicity sake.

Works Cited

Aimagin. “Waijung 2 for Zynq 7000.” *Aimagin*, https://www.aimagin.com/waijung-2-for-zynq-7000.html. Accessed 7 May 2025.

MathWorks. “mobilenetv2.” *MathWorks*, 2022, www.mathworks.com/help/deeplearning/ref/mobilenetv2.html. Accessed 23 April 2025.

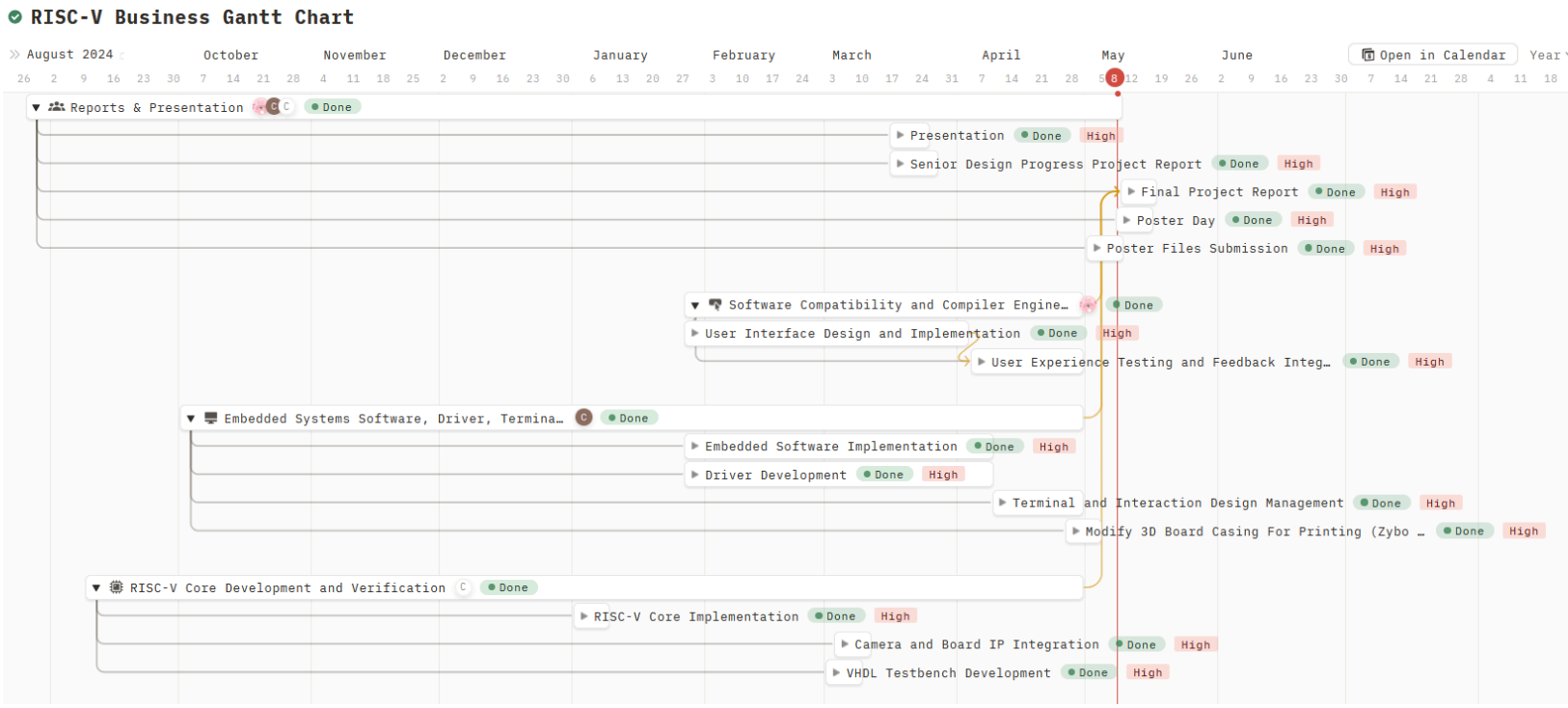
Patterson, David A., and John L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Elsevier Science, 2020.

RISC-V International. “RISC-V Ratified Specifications.” *RISC-V*, 2024, https://riscv.org/specifications/ratified/. Accessed 23 April 2025.

Sandler, Mark, et al. “MobileNetV2: Inverted Residuals and Linear Bottlenecks.” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

United Nations Institute for Training and Research. “Global e-Waste Monitor 2024: Electronic Waste Rising Five Times Faster than Documented E-waste Recycling.” *UNITAR*, 20 March 2024, https://unitar.org/about/news-stories/press/global-e-waste-monitor-2024-electronic-waste-rising-five-times-faster-documented-e-waste-recycling. Accessed 8 May 2025.

Appendix A: Completed Gantt Chart



Appendix B: Record of Team Meetings and Faculty Meetings

October 21, 2024

Attendees: Chris Nielsen, Hilary Ng, Christopher Shamah
Duration: 2 hours
Key Points:

- Discussed processor theorem
- Aligned team on project goals
- Assigned roles and responsibilities
- Scheduled regular meetings
- Discussed required software installation

October 28, 2024

Attendees: Chris Nielsen, Hilary Ng, Christopher Shamah, Peter Milder
Duration: 1 hour
Key Points:

- Discussed project design
- Addressed knowledge gaps
- Outlined steps moving forward
- Addressed project proposal concerns

November 4, 2024

Attendees: Chris Nielsen, Hilary Ng, Peter Milder
Duration: 1 hour
Key Points:

- Decided to start single-cycle design
- Assigned ARM chip implementation research

November 11, 2024

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 30 minutes
Key Points:

- Discussion of single-cycle considerations

November 18, 2024

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 30 minutes
Key Points:

- Discussion of branching/jump support in single-cycle structure

November 25, 2024

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 1 hour
Key Points:

- Discussion of multi-cycle structure breakdown based on textbook

December 2, 2024

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 1 hour
Key Points:

- Discussion of Vivado difficulties
- Bugs with test benching
- Proper ISA reference

December 9, 2024

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 1 hour
Key Points:

- Continued Vivado and Vitis IDE difficulties
- Recommendations for software IDE version

December 16, 2024

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 1 hour
Key Points:

- Pipeline structure finished and working for basic instructions
- Recommendations for further complicated situations and stress testing the processor

December 23, 2024

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 1 hour
Key Points:

- Difficulties with launching ELF files on MiniZed
- Developing within Vitis
- Running example code
- Ideas and concepts behind AXI examples and hardware

WINTER BREAK

January 27, 2025

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 1 hour
Key Points:

- Refreshed team on main objectives and goals
- Questions on SystemVerilog test benching in Vivado

February 3, 2025

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 1 hour
Key Points:

- Inability to testbench in Vivado
- Issues with mounting the VIP on the module path
- Professor Milder to research mounting VIP in DUT design
- *Note: Later resolved via email; professor sent proper DUT file path*

February 10, 2025

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 1 hour
Key Points:

- Python 1300 C EMBV development kit questions
- Ideas for starting vision extra features of the project

February 17, 2025

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 1 hour
Key Points:

- Discussion of AXI reads and writes example code
- Vivado AXI read/write tests with BRAM inside AXI entities (theoretical)
- Next steps

February 24, 2025

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 1 hour
Key Points:

- Professor requested expansion of AXI instruction and data memory

- Aim to make memory more usable and realistic compared to contemporary small processors

March 3, 2025

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 1 hour
Key Points:

- Tested in Vivado testbench (successful): can read instruction and register memory
- Used block RAMs for memory
- Register file not using block RAM (acceptable)
- Reviewed synthesis warnings
- No need for AXI calls for each individual operation
- Explored using Linux VM for Python script and RISC-V GCC install

March 10, 2025

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 1 hour
Key Points:

- Able to use BRAM
- Not recommended to use a separate clock for memory; change clock signal
- Explored function definition with parameter input
- Investigated global variables and configuration
- Discussed linker scripts and configuration for compilation
- Considered separating RISC-V extensions (multiplication, no division)

March 17, 2025

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 1 hour
Key Points:

- Demonstrated expanded and working CNN AI camera demo on ZYBO Z7-20 board
- Showed remote desktop camera feed and demo video
- Discussed Python application for compiling RISC-V hex (512 instructions max)

March 24, 2025

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 1 hour
Key Points:

- Explained new UART instruction loading technique for dynamic instruction data transfer to processor

March 31, 2025

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 1 hour
Key Points:

- Completed C code implementation of testbench operations
- Initialized memory and processor control signals (START, HOLD, RESET)

April 7, 2025

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 1 hour
Key Points:

- MiniZed successfully placed and routed with RISC-V hardware
- Ran basic program with known results
- Achieved successful operation on board after adjusting clock frequency and constraints

April 14, 2025

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 1 hour
Key Points:

- JALR discussion and extra hardware strategies
- Implementing RTL diagram of processor for board/report
- Recommendation to use Draw.io for diagram creation

April 22, 2025 *(Rescheduled for Tuesday)*

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 1 hour
Key Points:

- Plan to email Prof. Milder when report is ready for evaluation
- Adjusted diagram in Draw.io for clarity (darker line, emphasize architectural differences)
- Recommendation to include diagram in landscape at end of report

April 28, 2025

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 1 hour
Key Points:

- Discussion of JALR successful operation
- Further operational compatibility of the processo
- Markup to final report
- Further scale of the project

May 5, 2025

Attendees: Chris Nielsen, Chris Shamah, Hilary Ng, Peter Milder
Duration: 1 hour online zoom
Meeting notes 5/5

- Deliverables & Demo to Prof. Milder at Friday 5/9 via Zoom at 2:30PM
- Priority: Memory test, figuring out stack pointer and memory, fix the data mem read back issues and why the processor results are different on hardware.

Hillary C. Ng

East Islip, NY
631-415-2023 HillaryChandra13@gmail.com https://www.linkedin.com/in/hillary-c-ng github.com/hillabunny

Education

Stony Brook University **2021 – Expected 2026**
Accelerated BE/MS Program in Computer Engineering (GPA: 3.5) *Stony Brook, NY*

Suffolk County Community College **2019 – 2021**
Associated of Applied Arts and Sciences (GPA: 4.0) *Suffolk County, NY*

Relevant Coursework

- Electrical Circuit Analysis
- Electronics
- Embedded Systems Design
- Digital Logic Design
- Signals and Systems
- Random Signals and Systems
- Digital Design using VHDL and PLDs
- Software Techniques for Engineers
- Data Structures
- Computer Vision
- Real time Operating Systems
- Computer Communications
- Fundamental Algorithms for Automated Electronic Design
- Discrete Math
- Calculus I/II/III/IV
- Physics I/II
- Linear Algebra

Experience

Zebra Technologies **2024**
Advanced Data Capture Software Engineering Intern *Holtsville, NY*

- Gained hands-on experience with various tools used by the software engineering team..
- Contributed to the development and maintenance of real-time operating embedded systems.
- Collaborated closely with senior engineers to troubleshoot and resolve software issues.
- Improved understanding of embedded systems architecture and real-time operating systems (RTOS).
- Developed and documented scripts and utilities to automate common tasks.
- Enhanced problem-solving skills by working on real-world projects in a fast-paced environment.

CVS Pharmacy **2021 – 2023**
Pharmacy Technician *Islip Terrace, NY*

- Completed various tasks throughout the day, including customer service, transaction handling, and keeping stock of pharmaceutical supplies.
- Learned and utilized an internal database system to keep inventory of pharmacy supplies.
- Aided on the front lines of the COVID-19 pandemic by testing individuals using Epic and Lumira.
- Aided in the fight against COVID-19 by being a member of the vaccination team, and showcasing a great teamwork to make the vaccination process more efficient.

Projects

Scanner Frame Injection | *C, C++, & Python*

- Developed a code wrapper to convert images into files suitable for uploading to an existing Zebra scanner.
- Integrated with a large existing codebase to build the program using existing APIs.
- Designed and implemented a new state machine to simulate image scanning, aiding the engineering team in debugging by injecting images into the scanner's memory buffer.
- Created comprehensive documentation detailing how to use the program, ensuring ease of understanding and future maintenance.
- Collaborated with team members to test and refine the program, improving its reliability and performance.
- Enhanced technical skills in embedded systems and API integration.

Embedded System | *AVR Assembly & Hardware wiring*

- Leveraged Atmel/Microchip AVR128DB48 I/O ports to architect and develop embedded systems with microcontroller programming.
- Utilized Microchip Studio to code AVR assembly programs and debugging, ensuring seamless integration into the microcontroller for optimal system functionality.
- Successfully integrated a 4x4 matrix keypad into the system, enhancing user interaction and feedback by incorporating LED display to visually represent input information.

Technical Skills

Languages: C, C++, AVR Assembly, MATLAB, Python
Developer Tools: Visual Studio Code, Visual Studio, Dev C++, Microchip Studio, Aldec Active HDL
Technologies/Frameworks: Microsoft Office, Google Suite, Notion
Foreign Languages: English, Mandarin Chinese, Bahasa Indonesia

CHRISTOPHER J. SHAMAH

61 S. Robert Damm Street
Bethpage, NY 11714

Phone: 516-350-4709
ChristopherShamah@gmail.com

Skills

Proficient in C++, Python, Java
Networking
Basic HTML Knowledge
Troubleshooting
Identity Access Management
UI Design

Summary of Qualifications

Computer Engineer with a background in Java, C++, and Python. Experienced in software development, troubleshooting and networking. Adept at collaborating with cross-functional teams and driving projects to successful completion. Proven ability to connect identity access management for high-value websites. Eager to leverage expertise in software, logic, and advanced analytical thinking skills to deliver high-quality solutions.

Education

SUNY STONY BROOK, Stony Brook, NY
Bachelors of Science Computer Engineering 2025

Experience

SOHO SOLUTIONS INC October 2017 – August 2023
Cable Technician/Software Engineer
Glendale, NY

- Involved in the development and maintenance of various software applications
- Collaborated with cross-functional teams to analyze requirements, design solutions, and deliver high-quality software products within project timelines.
- Implemented troubleshooting protocols to identify and resolve software issues, ensuring smooth functionality.
- Headed the development of the identity access management portion of a website containing millions of dollars worth of critical information, enhancing security and accessibility.
- Utilized Microsoft SharePoint for efficient collaboration and document management within the team.
- Continuously improved processes and systems to enhance software performance and user experience.
- Replaced defective parts or components within a cable system when necessary.
- Ran Cat 5 Cable, Installed Cat 5 ports, and other related equipment.
- Repaired faulty connections by removing old wiring and replacing it with new wiring as needed.

STONY BROOK UNIVERSITY SCIENCE FICTION FORUM December 2023 – Present
President/Head Activity Coordinator
Stony Brook, NY

- Led and managed a team of 12 executives, ensuring effective communication and collaboration
- Performed Various duties to keep the organization running including; filling out paperwork, recruiting members, allocating $13k yearly Budget
- Collaborated with school staff to develop and implement extracurricular activities
- Planned Multiple Events per week, the largest of which had over 200 attendees

Christopher Nielsen

36 Herbert Avenue, Port Washington, NY 11050
516-946-8995 Christopher.R.Nielsen@protonmail.com
LinkedIn - christopher-nielsen-ab6233205 nielsenengineering.netlify.app Github - CypherVault

Education

Stony Brook University **May 2025**
Major in Computer Engineering, Bachelor of Engineering - 3.05 GPA
Eagle Scout - Class of 2021

Chaminade High School **2017 - 2021**
4 Year Honor Roll Student *Mineola, New York*

Relevant Coursework

Computer Architecture, Machine Learning/Vision, PCB Design, Digital Design, MicroController Systems, Schematic Drawing, VLSI

Technical Skills

Languages: C, C++, VHDL(FPGAS), MIPS/AVR Assembly, Shell, Rust, Python
Developer Tools: Xilinx SDK, AutoDesk, Fusion360, Aldec, Cadence, GitHub, VS Code, OrCAD, LT Spice, Cube STM, Solidworks Electrical, Jira, Odoo
Expertise: Embedded Systems, IT/AV, Electronics Lab Equipment, Soldering, Board Level Troubleshooting, IoT

Work Experience

Unique Technical Solutions LLC **May 2024 – Present**
Electrical Engineer Part Time - 50 hr/week *Holbrook, NY*

- Responsible for the design/layout/database of schematics, embedded code, and firmware of FORD/THOMAS Electric Vehicle conversion kits, HVAC systems, and contracted aquatic projects worth $300 million.
- Collaborated effectively with a diverse team of electrical and mechanical engineers/installers in a high pace environment, fostering productivity and innovation. Regular review sessions, revision communication, JIRA ticketing, and proprietary documentation handling.

North Atlantic Industries **June 2023 – December 2023**
Test Engineer Part Time - 40 hr/week *Bohemia, NY*

- Verified functional operations and repaired 20+ NAI models (including Analog Electronics, Single Board Computers, and Function Modules) through collaboration with various multi-disciplinary teams and use of layouts/schematics/automated test scripts.
- Executed testing and debugging on 30+ unique IC controllers and power supply circuitry (PCIE/Serial/I2C/Buck/Boost/I2C/SPI/UART/RS232).

Stony Brook DoIT CES: Media Systems Engineer Assistant **August 2021 – Present**

- Helped development of $50,000 165-inch video info graphic wall system, encompassing software interface creation, technical documentation, wiring schematics, and installation oversight for lecture hall installation.
- Required recurring cross-team collaborative meetings for upkeep of 140+ university AV installations.

Projects

RISCV and NPU Dual Core Processor

- Engineered a five-stage pipelined processor with RISCV32I ISA compatibility on a Zynq XC7Z020 FPGA.
- Implemented and ran MobileNetV2 Nueral Network image recognition on bare metal using Vivado video IP, VITIS SDK, and Matlab Simulink.

Lidar-Guided Automated Driving Internship Project

- Integrated RS232 serial communication and DC-to-AC modules as well as software on Zynq 7015 Soc ARM-based board for internship project, showcasing product application range to potential partners.

45nm CMOS D Flip Flop - VLSI

- Designed and simulated the layout of a 45nm CMOS based D-Flip-Flop with an asynchronous RESET.

Leadership / Extracurricular

Order of the Arrow - Scouting's National Honor Society **Spring 2020 - Spring 2023**
Council Lodge Treasurer *Buckskin Lodge, Theodore Roosevelt Council 386*