RISC-V FPGA Final Report
EECS 151
Spring 2022

Team 1: Fee-FIFO-Fum
Shrey Aeron and Neeraj Shenoy

May 9, 2022

Contents

1 Functional Description

Our main objective for this project involves designing a three-stage pipelined RISC-V CPU with a UART for tethering. Once we create a diagram of the datapath we will implement, noting down all intricacies between RAMs, stages, and control logic, we go on to create the datapath itself. We use Verilog, a hardware description language, to implement this system, targeting the Xilinx PYNQ platform (a PYNQ-Z1 development board with a Zynq 7000-series FPGA–see Figure 1).

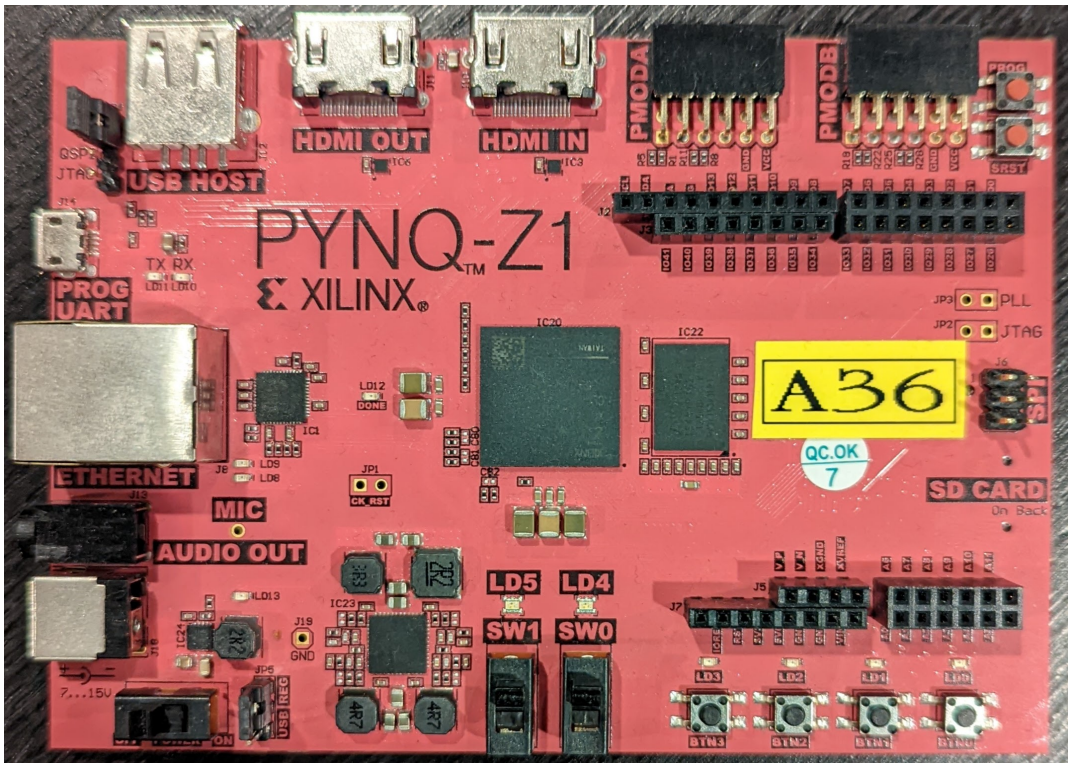


Figure 1: Xilinx PYNQ-Z1 Board

The design objectives for this project include minimizing the execution time of a matrix multiplication program. Execution time for the program is dependent on both CPI (cycles per instruction) and the CPU clock frequency, both of which we can optimize.

1.1 Pipelining

We are at liberty to choose where and how we want to delineate our three stages. To do this, we place pipeline registers where we want these separations to be so that data is is synchronously written on a rising clock edge. All of the logic within a given stage should be asynchronously read and written to our datapath components (ex. the branch comparator and ALU), with the exception of the RegFile, which is synchronously written to during the writeback stage, and our memories, which we will now describe.

1.2 Memory Hierarchy

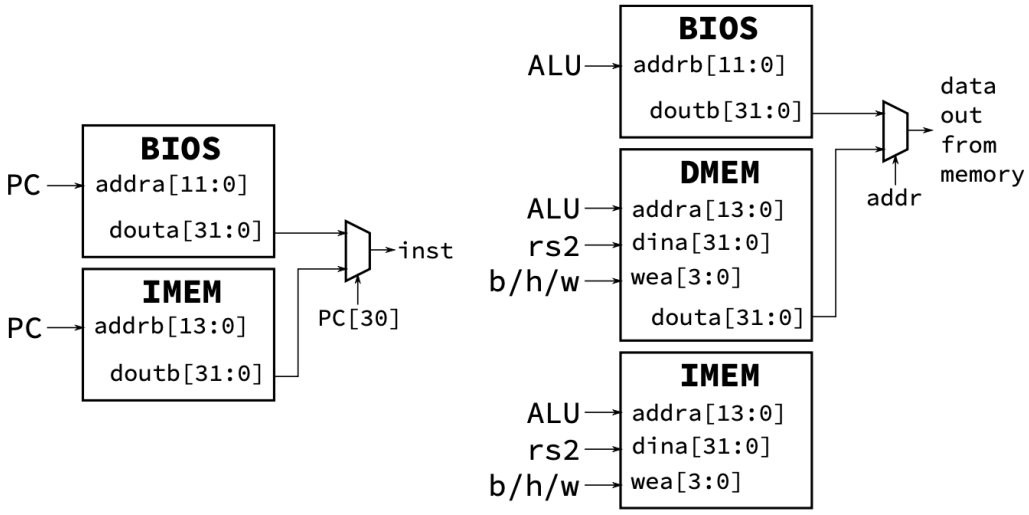


Figure 2: Memory Hierarchy in Datapath

See Figure 2 for a layout of the memory hierarchy. All of our memories have synchronous reads and synchronous writes.

1.2.1 BIOS

The Basic Input/Output System (BIOS) memory is where the processor begins execution. It is initialized by the BIOS program, which should be able to read from the BIOS memory (to fetch static data and instructions) and read and write from the instruction and data memories. User programs are uploaded through a UART ready/valid interface to the BIOS memory.

1.2.2 IMEM

The Instruction Memory (IMEM) is where all instructions for the currently loaded program are stored.

1.2.3 DMEM

The Data Memory (DMEM) is where all loads and stores over the course of a user program occur.

1.2.4 Memory Mapped I/O

The Memory Mapped I/O is a technique in which registers of I/O devices are assigned memory addresses. This enables load and store instructions to access the I/O devices as if they were memory. The map includes both instruction and cycle counters so that CPI can be determined.

2 Technical Details

2.1 CPU Organization

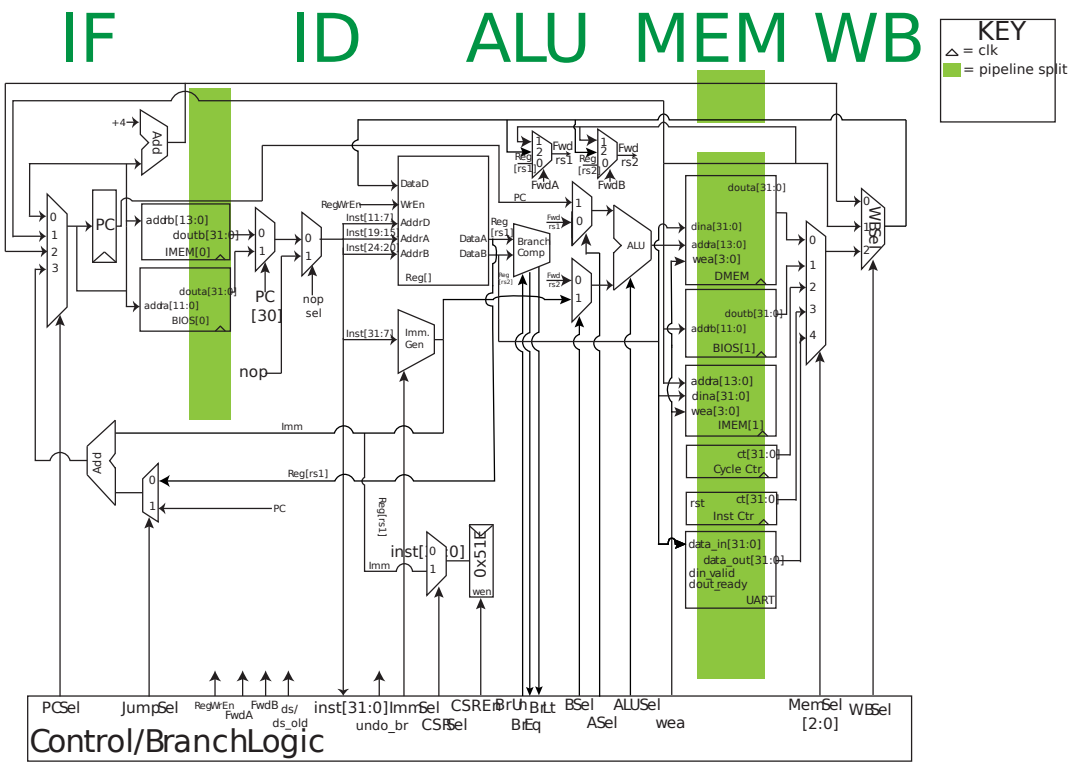


Figure 3: RISC-V Final Datapath

See Figure 3 for a high level overview of the datapath. We choose to break down our Datapath into three main clocked portions containing the following parts of the processor:

- Instruction Fetch
- Instruction Decode, ALU
- Memory, Writeback

We chose this split due to the way memory was read from the RAMs into the processor. By putting most of our memory operations on the clock edge, we made our design simpler and easier to manage. Additionally, the central ALU and Decode module was split into the smaller modules for ALU and Branch comparator, which were wired together with a connector module linking the stage up before adding it to Riscv151.

The control logic was clocked separately without the registers provided to us, and it kept track of the current and previous instructions to send the appropriate control signals to different stages of the CPU. This allows us to send crucial signals for the program counter, branching, and saving data.

2.2 CPU Subsections

2.2.1 Instruction Fetch

In the Instruction Fetch Stage, we take in the PCSel value and choose whether to read from ALU, PC, PC4, or Add. This last add command is special because it aims to reduce the critical path for the CPU in the case of Jump and Branch instructions. Instead of going through the slower and and complex ALU stage, it grabs for imm, rs2, and PC from the second stage continuously and sends it to the MUX for selection. Thus, branches are calculated without violating any timing constraints with a short critical path internally in the fetch stage.

In addition, the actual Program Counter is clocked to update itself every cycle internally. However, to reduce errors with reading the instructions from IMEM/BIOS, the calculated PC is continuously passed into the memories, which only execute on clock.

2.2.2 Decode and Execute

At the start of the Decode and Execute Stage, we use PC[30] as a selector bit between the IMEM and BIOS for our instruction. Then, we use the nopsel signal to decide whether or not to insert a nop instruction (hard-coded as addi x0, x0, 0 in our design). More on this nopsel signal will be explained later. The entire instruction is passed to the control logic so the correct control signals can be passed back to the datapath later in the second and third stages. We then pass parts of this instruction to the RegFile and also to the immediate generator, where the appropriate immediate will be generated based on the ImmSel from the control logic.

Now we get to the execution phase of this stage. Our branch comparator and ALU are separate modules with asynchronous inputs and outputs for simpler organization. The branch comparator module takes in as inputs the rs1 and rs2 signals from the RegFile, as well as whether or not a branch instruction is unsigned via the BrUn signal. It outputs to the control logic whether the value in rs1 is less than (via signal BrLt) or equal to (via signal BrEq) the value in rs2. As for our naïve branch prediction scheme, we assume a branch is taken for any branch instruction. Looking back in the stage, the nopsel signal is high either on rst or when we need to flush the pipeline if we mispredict a branch instruction.

This brings us to our forwarding scheme. To account for data hazards, we forward the ALU result from the third stage, which is handling the previous instruction in the pipeline. The FwdA and FwdB MUXes decide whether or not to feed the forwarded ALU result, the current rs1 and rs2 values, or the writeback stage output to the RegFile to the ASel and BSel MUXes. The determination is made in the control logic. At this point, we have two inputs to the ALU module, which will choose which operation to perform on the inputs based on ALUSel. The ALU output is then passed to the DMEM, IMEM, and UART.

2.2.3 Memory and Writeback

The Memory Stage is laid out such that it is right on the clock period, ensuring that the outputs from the previous stages are immediately written in upon clock tick. The memories are laid out to take data from rs2 and store it into the address in alu_out. This data is already masked, such that it can execute store instructions on the memory cells correctly. The memories include the standard Instruction Memory and Data Memory, but also BIOS and UART for communication and a single Counter module. This

module contains the outputs for the Instruction and Cycle Counters. The Instruction counter simply increments 1 instruction after 2 clocks have executed (since this is a 3-stage processor) and does not increment if it detects that a nop signal was inserted by the previous stages.

Once the data is written/read into the respective memories, the writeback stage aims to choose which memory/output to write from back into the RegFile on the next clock cycle. It continuously assigned the next value to write back and assigns to a master output register on clock edge that goes to the RegFile. The data is chosen by a MUX Cascade, to first choose which type of memory we are writing back from (DMEM, IMEM, BIOS, UART, Counters) and then specifically chooses the type for the Memory Mapped IO Modules. Ideally, a 1D MUX structure with 1 layer would have greatly benefited performance due to scaling at higher dimensions and a shorter critical path. In this portion of the processor, the data is also masked as it is read back from the memories. For LOAD type instructions, the relevant masking scheme for the load type is passed in from the previous stage (HALF, BYTE, or WORD) and the data is shifted using the address bits calculated in the ALU. This same process is calculated for signed loads, with the only difference being the sign extensions on the MSB of the masked input.

2.2.4 Control Logic

The Control Logic is arguably the most complicated part of the processor, as it outputs multiple signals which are quintessential to the operation of each stage in the process and links the stages together by passing the correct interpretation of the instruction and what needs to be done to successfully execute it given the current environment of the processor, taking into account the previous instructions, hazards, branch status, writebacks, and more.

The module we created consists of a series of continuously assigned functions which take the current inputs and asynchronously output the correct control signal. Somewhat non-ideally, we instantiate registers that hold the last instruction in order to provide control signals for the third stage, as the instruction from the previous clock cycle enters this stage and requires the correct set of signals to be sent for valid operations. When the CPU starts, the input signals tend to be haywire or all zeros, so we prevent any odd behaviors in PCSel by gating the real output through a ternary operator driven by the instruction's opcode.

The logic also takes care on indicating whether the current instruction is a jump or branch type instruction, which triggers the selector bit in the MUX in IFetch to use imm + rs2 or imm + pc. This, with the PCSel, helps jump and branch address calculations occur much faster, diverting from the slower ALU path. Many seemingly random control signals, such as data_size and data_type, help determine actions for specific types of instructions, such ad LOAD or STORE. Similarly, forwarding logic for writeback, is determined here, which feeds MUXes in the Execute stage which choose which data needs to chose from the previous writeback cycle and which of rs1 or rs2 would require this data.

The main drawback in the control logic stage is the extremely significant logic delays that are faced during instruction execution. Due to the large MUX sizes used, with up to 5 bit selectors, the output update time of this portion of the processor acted as a hindrance to increased clock speeds. Much of the naïve control code was simplified using Boolean Algebra, but at the expense of readability. Many of the heavy hitting MUXes still remain, but this could be fixed for future iterations by simplifying logic for multi-bit outputs by calculating logic for each output bit, again at the expense of readability and difficulty if the ENUM definitions change.

3 Status and Results

All the critical components of the processor from a basic standpoint work as needed. It can branch, jump, flush instructions, and the RISCV 32-bit base instruction set is supported completely. We are still working on a branch predictor, modeled from the design proposed in the document here: Link to UT Austin Branch Predictor Slides.

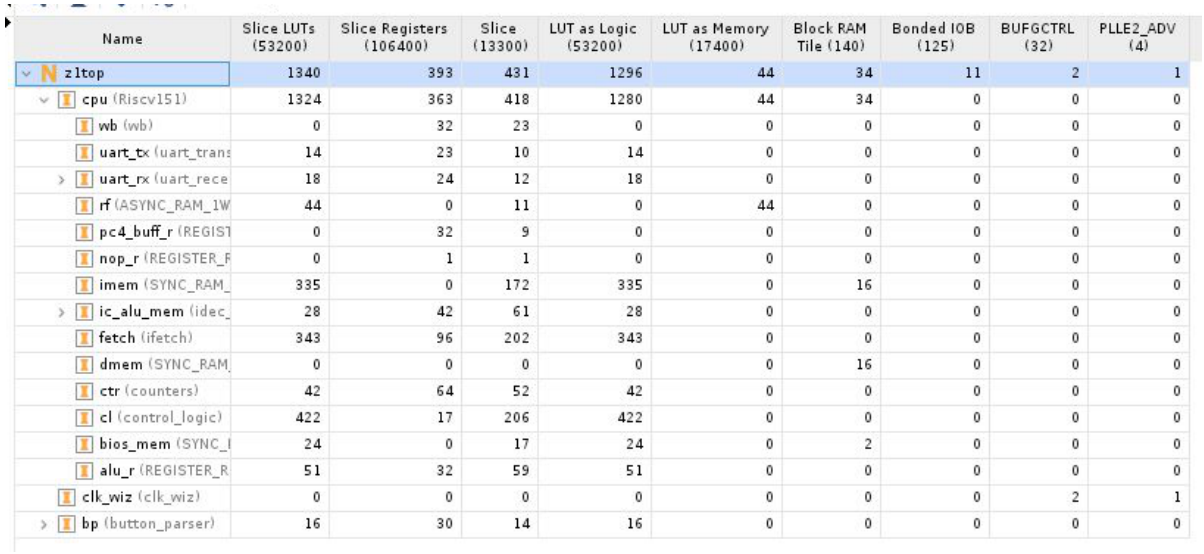
The processor runs with a clock period of $17ps$, meaning that the new clock frequency is $125MHz \cdot \frac{8}{1\times17} = 58.8MHz$, being $\approx$ 16% better than the existing and base clock speed. All checkpoints work at this speed, since most of the critical path delay arises from the control logic, which is a crucial part for every processor checkpoint. Additionally, later checkpoints added minimal or no delay because the asynchronously calculated logic ran faster than other existing logic.

Here is the relevant Hardware Utilization for the z1top run:

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	1327	0	0	53200	2.49
LUT as Logic	1283	0	0	53200	2.41
LUT as Memory	44	0	0	17400	0.25
LUT as Distributed RAM	44	0			
LUT as Shift Register	0	0			
Slice Registers	393	0	0	106400	0.37
Register as Flip Flop	329	0	0	106400	0.31
Register as Latch	64	0	0	106400	0.06
F7 Muxes	0	0	0	26600	0.00
F8 Muxes	0	0	0	13300	0.00

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	34	0	0	140	24.29
RAMB36/FIFO*	34	0	0	140	24.29
RAMB36E1 only	34				
RAMB18	0	0	0	280	0.00

Site Type	Used	Fixed	Prohibited	Available	Util%
BUFGCTRL	2	0	0	32	6.25
BUFIO	0	0	0	16	0.00
MMCME2_ADV	0	0	0	4	0.00
PLLE2_ADV	1	0	0	4	25.00
BUFMRCE	0	0	0	8	0.00
BUFHCE	0	0	0	72	0.00
BUFR	0	0	0	16	0.00



Name	Slice LUTs (53200)	Slice Registers (106400)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	Block RAM Tile (140)	Bonded IOB (125)	BUFGCTRL (32)	PLLE2_ADV (4)
z1top	1340	393	431	1296	44	34	11	2	1
cpu (Riscv151)	1324	363	418	1280	44	34	0	0	0
wb (wb)	0	32	23	0	0	0	0	0	0
uart_tx (uart_trans	14	23	10	14	0	0	0	0	0
uart_rx (uart_rece	18	24	12	18	0	0	0	0	0
rf (ASYNC_RAM_1W	44	0	11	0	44	0	0	0	0
pc4_buff_r (REGIST	0	32	9	0	0	0	0	0	0
nop_r (REGISTER_R	0	1	1	0	0	0	0	0	0
imem (SYNC_RAM_	335	0	172	335	0	16	0	0	0
ic_alu_mem (idec_	28	42	61	28	0	0	0	0	0
fetch (ifetch)	343	96	202	343	0	0	0	0	0
dmem (SYNC_RAM_	0	0	0	0	0	16	0	0	0
ctr (counters)	42	64	52	42	0	0	0	0	0
cl (control_logic)	422	17	206	422	0	0	0	0	0
bios_mem (SYNC_I	24	0	17	24	0	2	0	0	0
alu_r (REGISTER_R	51	32	59	51	0	0	0	0	0
clk_wiz (clk_wiz)	0	0	0	0	0	0	0	2	1
bp (button_parser)	16	30	14	16	0	0	0	0	0

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	422	0	0	13300	3.17
SLICEL	254	0			
SLICEM	168	0			
LUT as Logic	1283	0	0	53200	2.41
using O5 output only	0				
using O6 output only	1122				
using O5 and O6	161				
LUT as Memory	44	0	0	17400	0.25
LUT as Distributed RAM	44	0			
using O5 output only	0				
using O6 output only	0				
using O5 and O6	44				
LUT as Shift Register	0	0			
Slice Registers	393	0	0	106400	0.37
Register driven from within the Slice	200				
Register driven from outside the Slice	193				
LUT in front of the register is unused	55				
LUT in front of the register is used	138				
Unique Control Sets	16		0	13300	0.12

Here are a few more performance numbers on mmult in terms of the various CPIs achieved:

- Original: 1.33
- JAL optimization: 1.29
- Branch optimization: 1.14

The slowest clock period was reported as: 14.058ps, with the slowest logic being: 15.071ps.

4 Learning and Future Work

Overall, our project was certainly a success. We were able to turn in every checkpoint of the project on time, which is stellar considering the rigor of this project typically leaves some people with an unfinished datapath. Having the ability to test at home with our own boards and off-chip UARTs certainly saved time, rather than having to walk to the lab (see Figure 4). We had to make use of slip days for checkpoint 2 due to starting a bit later than we would have liked, but our extensive datapath diagram and thought put into the checkpoint 1 questions paid dividends, in our opinion, in saving us time constructing our first implementation of the datapath.

If we were to redo the project, we would probably start a bit earlier in the checkpoint 2 time frame. Also, we were only using the iverilog tests at first to debug. Then, we realized using the Vivado tests were a lot more insightful, as they pointed out latches and unreachable cases in our logic. Once we passed the Vivado tests, it was a pretty good indication the implementation would work on the board.

In terms of future work, implementing a five-stage datapath is certainly feasible–we would simply need to create two more modules and make sure to split our inputs and outputs accordingly. And as aforementioned, an updated branch predictor could certainly be in the works as well; we would make a two-bit saturating counter by using an FSM with two-bit states–this would reduce our CPI even further.

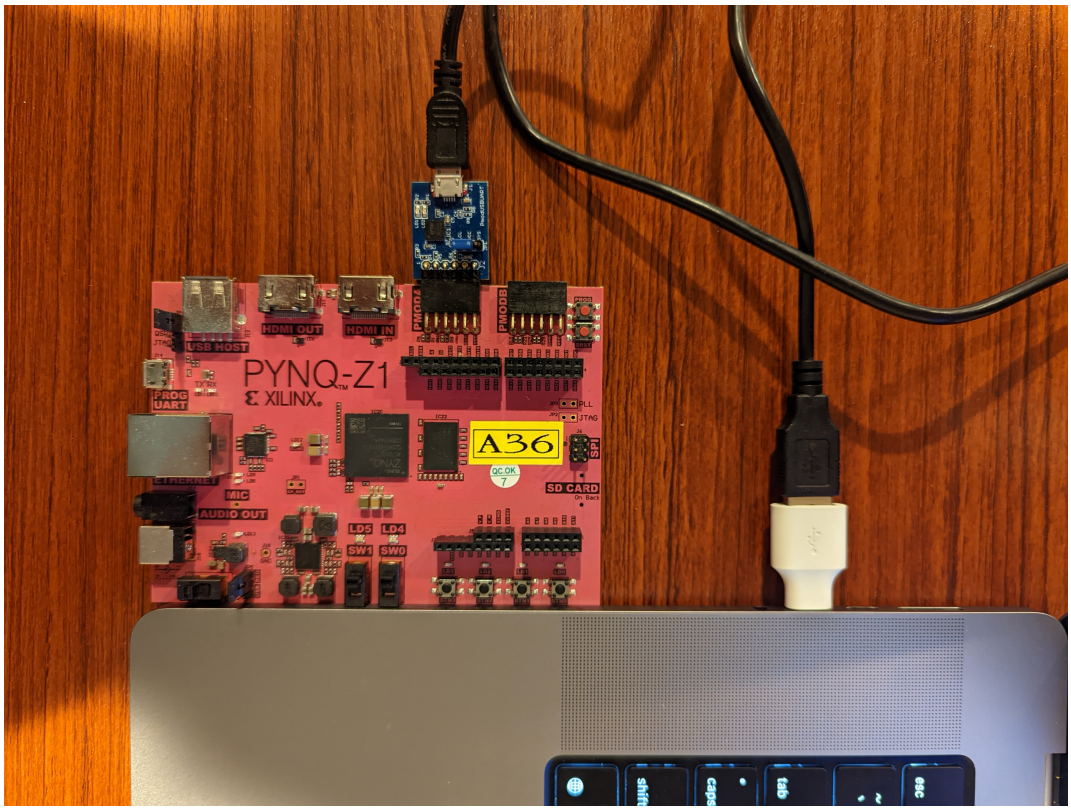


Figure 4: FPGA Connected to Laptop via Off-Chip UART