**FINAL PROJECT REPORT**

**Team Members:**

**Abhishek Murlidhar Patil – amp170830**

**Shri Prakash – sxp175131**

**Reverse Engineer a 32-bit Arm processor with an 8-bit data bus and decoding of multiple ISA’s**

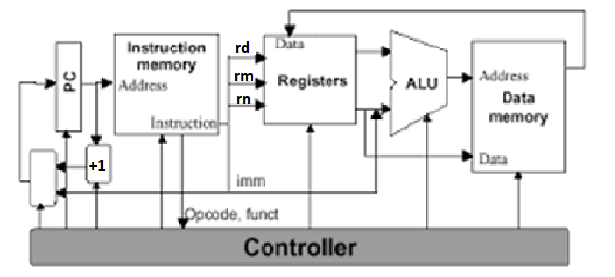
**Objective:**

The first objective of this project is to design, simulate, and implement a multicycle 32-bit microprocessor and 8-bit data bus with ARM architecture which supports the following instructions:

* Data processing
* Single data transfer
* Branch

The second objective of the project is to decode multiple ISAs in single platform which are ARM and RISC-V architectures

**General architecture of ARM**:



The data path consists of:

* **ALU**: performs all the necessary arithmetic/logic operations required to implement the ARM instruction set. The ALU also implements the conditions needed for branches and asserts the “Branch Taken” output if the condition is true.
* **Register File**: registers with two read ports and one write port.
* **IR**: The Instruction Register (IR) holds the instruction once it is fetched from memory
* **PC**: The Program Counter (PC) is a 10-bit register that contains the memory address of the next instruction to be executed.
* **Controller** which controls all the Datapath and the memory module.
* **Data Memory**: contains the RAM

The control signals include:

* rd\_en – Set read enable
* wr\_en – Set write enable
* Mem\_to\_reg – Set memory to register flag
* Reg\_write – Set register write
* Reg\_dest – Set register destination
* ALUC – Specifies to the ALU which type of instruction must be executed.

**Execution steps**

Since the instructions are 32-bit and only 8-bit data bus, we designed a instruction register of 512 entries of 8 bit each and increment the PC by 1 every clock cycle and decode the instruction once 32-bit instruction is determined.

* All instructions:
  + Step 1: - Fetch instruction, store in Instruction\_Reg, PC = PC + 1. Keep doing this until we get all 32 bits of one instruction
  + Step 2: - Decode instruction - Decode the 32 bit instruction to generate control signal and also source operands.
* Memory access:
  + Step 3: - Compute memory address to load from or store to.
  + Step 4: - If lw: Retrieve data from memory at specified address

- If sw: Write data in source3 operand to memory at specified address

* R-type:
  + Step 3: - Perform specified operation based on opcode value in the signal ALUC
  + Step 4: - After computation result value is stored in the register address specified by destination operand
* Branch:
  + Step 3: - If branch instruction jump to the address specified by offset.

**Design Environment:**

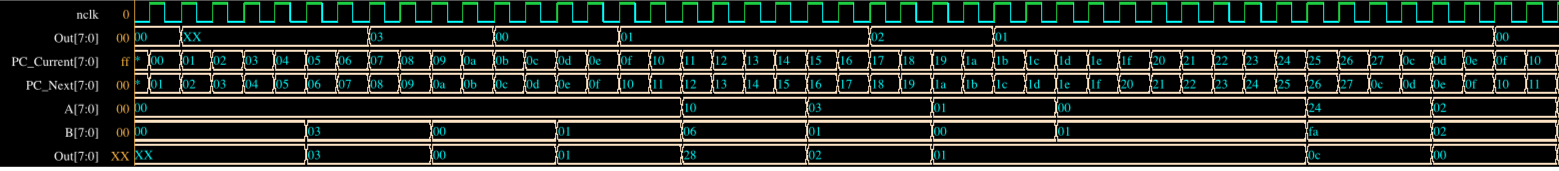
EDA playground was used to generate simulation file. After confirming the desired working operation of the ARM processor design, Xilinx ISE is used to synthesize the design and mapped to Nexys 3 Spartan 6 FPGA board to verify a simple program to find nth Fibonacci number.

**Simulation Results:**

These are the simulation results we got by running our Verilog code on EDA playground simulator to execute the following instructions which is the assembly code to find the nth Fibonacci number:

* MOV R4 #3 //n=3
* MOV R1 #0 //f1=0
* MOV R2 #1 //f2=1
* BZ #6 //Till n!=2 loop
* SUB R4 R4 #1 //n=n-1
* ADD R3 R2 R1 //f3=f1+f2
* MOV R1 R2 //f1=f2
* MOV R2 R3 //f2=f3
* B #-6
* CMP R4 #2
* B #-1
* MOV R2 R3

The first instruction takes 8 cycles to execute, the next set of instructions take 4 cycles each to execute.

****

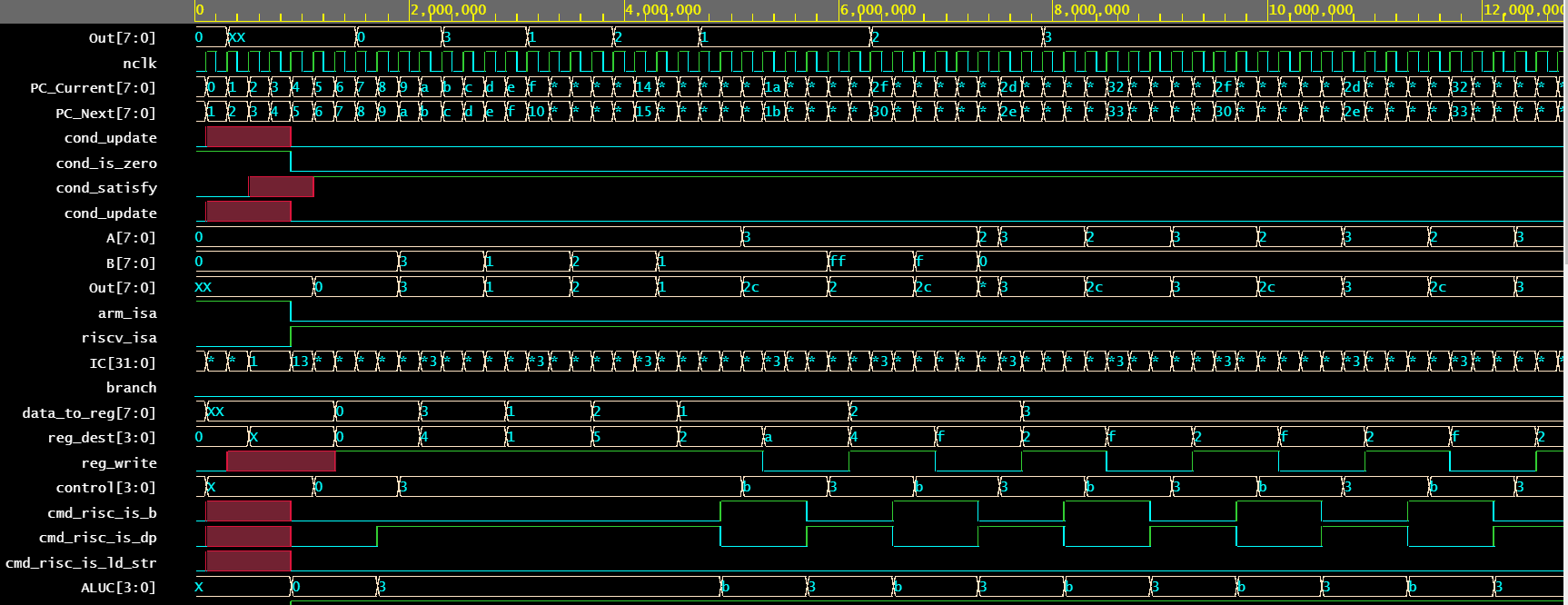
**Synthesis and Mapping to FPGA:**

The. ucf file was modified to map the Verilog output data which is the nth Fibonacci number, to the seven-segment display and bitstream file was generated which was dumped to the FPGA board

**RISC-V ISA and decoding multiple ISAs:**

RISC-V is an open instruction set architecture (ISA) based on established reduced instruction set computing (RISC) principles. In contrast to most ISAs, the RISC-V ISA can be freely used for any purpose, permitting anyone to design, manufacture and sell RISC-V chips and software. It is designed to be useful in modern computerized devices such as warehouse-scale cloud computers, high-end mobile phones and the smallest embedded systems. One of the key features of this ISA is that it is separated into a small base integer ISA, usable by itself as a base for customized accelerators or for educational purposes, and optional standard extensions, to support general-purpose software development.

To decode both ARM and RISC-V architecture-based instruction set we include an initial instruction which indicates the next following set of is of ARM architecture or RISC-V architecture. Specifically if the 32bit instruction is all zeros we set **arm\_isa** bit to ‘1’ and **risc\_v\_isa**  to ‘0’ else if only the first bit of 32 bit instruction is ‘1’ and rest zeros we **risc\_v\_isa bit** to ‘1’ and **arm\_isa** to ‘0’. This way we were able to decode both arm and risc-v architecture based instructions.



This simulation shows the finding the nth fibonacci number program written for RISC-V architecture run on the same platform. This simulation is similar to the previous one

**Challenges Faced:**

* Output data sent to the FPGA board was stable enough to print correctly on the FPGA board.
* Difficulty in handling the control signals while decoding the multiple ISAs
* Since the binary translation for assembly code was hardcoded there were some inevitable mistakes while binary translation