Five-Stage Pipelined 32-Bit RISC-V Base Integer Instruction Set Architecture Soft Microprocessor Core in VHDL

Aaron Elson Phangestu
Computer Engineering Department
Faculty of Intelligent Electrical and Informatics Technology
Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia 60111
aaronelsonp@gmail.com

Dr. Ir. Totok Mujiono, M.I.Kom
Electrical Engineering Department
Faculty of Intelligent Electrical and Informatics Technology
Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia 60111
totok_m@ee.its.ac.id

Ahmad Zaini ST, M.T
Computer Engineering Department
Faculty of Intelligent Electrical and Informatics Technology
Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia 60111
zaini@te.its.ac.id

***Abstract*—Proprietary technologies with complicated licensing currently dominate the microprocessor industry. As a result, we must seek out a freely available, open-source alternative. In this paper, we discussed the implementation of a five-stage pipelined soft processor core. The core uses the RISC-V RV32I Base Integer Instruction Set Architecture. RISC-V is an open standard ISA that is freely available to use and modify. To implement our processor core, we followed the FPGA design methodology. First, we implemented the design specification with the VHDL Hardware Description Language. Then, we simulated the design in the ModelSim simulation environment. Following the verification, we analyzed the resource usage, critical path, and maximum frequency of the processor, then uploaded the processor core to an actual Cyclone IV EP4CE6E22C FPGA. Our CPU core successfully executed all the RV32I instructions, except FENCE, ECALL, and CSR instructions. The proposed processor core runs on 2115 LUTs, 558 flip-flops, and 67,608 memory bits, with a maximum frequency of 62.95 MHz.**

***Index Terms*—RISC-V, RV32I, FPGA, VHDL, soft processor core, five-stage pipeline.**

I. Introduction

RISC-V is an open standard Instruction Set Architecture designed for academic and industrial applications[1]. This architecture was originally used in 2010 at UC Berkeley for educational and research purposes in Computer Architecture. Over time, RISC-V gained industry traction, and in 2015 the RISC-V Foundation came into existence to encourage RISC-V adoption. The ISA has been designed to support 32-bit, 64-bit, and 128-bit address spaces to increase adoption, and is developed with the principle of Reduced Instruction Set Computer in mind.

The RISC-V Instruction Set Architecture is categorized as a basic integer instruction set and several other extended instruction sets[2]. Depending on the needs of our application, we can use the additional extensions. RV32I refers to the 32-bit base integer instruction set. Standard RISC-V extensions are defined as M for multiplication and division, F for floating-point instructions, D for double-point precision instructions, and A for atomic instructions. In addition, RV32G is a general-purpose 32-bit instruction set that includes all instructions from the M, F, D, and A extensions. There also exist the 64-bit and 128-bit variants of these instruction sets.

We focused on the RV32I base integer instruction set in this paper because it was sufficient for our needs. RV32I can emulate almost all other instructions from other extensions, except the A instruction sets. The main goal of this paper is to create an RV32I soft processor with five-stage pipelining that is simple to understand for academic purposes. And to see how it stacks up against a similar single-cycle RV32I soft processor.

We proposed a simple five-stage RV32I soft processor in this paper, which was tested on an EP4CE6E22C8N FPGA board. The microarchitecture is intended for academic purposes only since we haven't verified it formally. Our main contribution includes:

- We proposed an RV32I ISA soft processor with a five-stage pipelining. Our processor core can execute all RV32I instructions, except FENCE, environment call, and CSR instructions. The processor can run a program that we disassembled from an RV32I assembly code using the RISC-V GCC toolchain[3].
- We implemented said processor with VHDL Hardware Description Language and tested it within the ModelSim simulation environment. We uploaded the processor core into an FPGA after we validated the processor. Lastly, we evaluate the maximum frequency and resource allocation.
- We compared the proposed RV32I pipelined soft processor with our prior single-cycle RV32I soft processor.

II. Related Works

Maestro[4] is a five-stage RV32I ISA pipeline implementation based on David Patterson and John Hennessy's Computer Organization and Design RISC-V Edition. The project is entirely academic; it was developed at the Federal University of Rio Grande do Norte in Brazil, and the creator didn't intend to compete with complex implementations. It was created primarily for educational purposes about RISC-V, the ISA, and processor design. This project used VHDL as its Hardware Description Language.

We have found several drawbacks of the Maestro soft processor. First, we discovered a problem with the flushing unit when we simulated a couple of instructions in the ModelSim

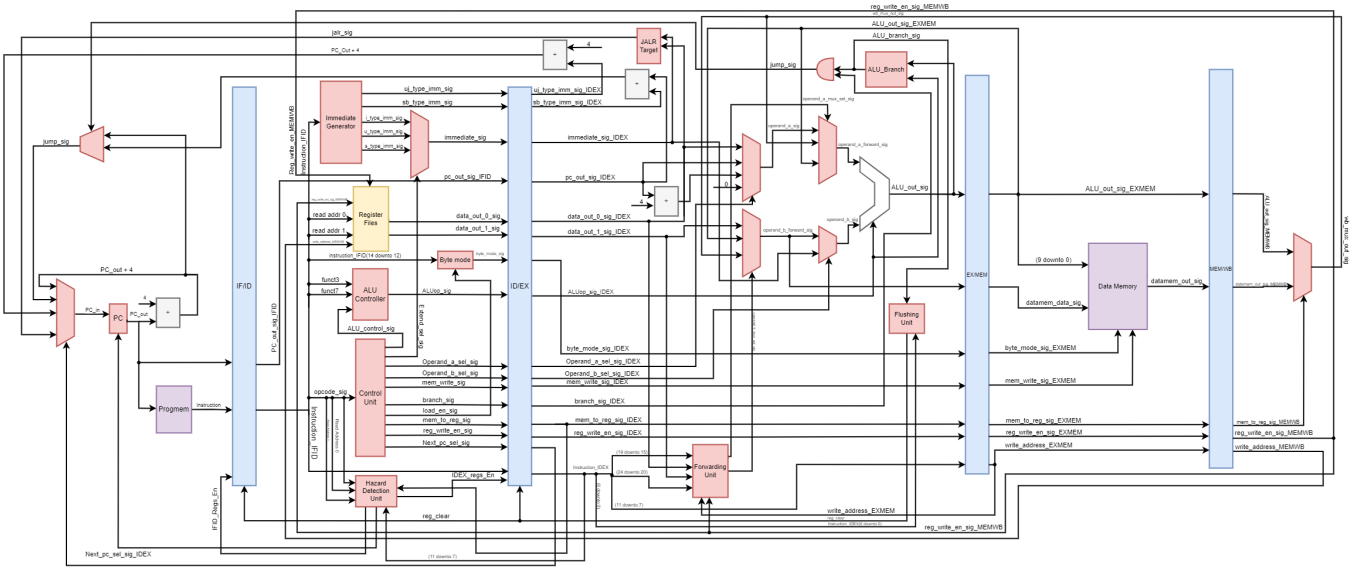

Fig. 1. The architecture of our proposed RISC-V soft processor with five-stage pipelining.

simulation environment. The glitch happened because of a timing error when the flushing unit detected a jump or branch. Furthermore, during the development, there was a timing problem with the overall processor core, and it was not able to be run in an actual FPGA. Lastly, several instructions remain unimplemented, such as AUIPC.

In addition to Maestro, we are also strongly inspired by John Hennessy's textbook[5] and our previous work on a single-cycle RV32I soft processor. The book discussed a five-stage pipeline microarchitecture, which includes: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (Mem), and Write Back (WB). In the book, there were several examples written in Verilog HDL. However, the architecture in the book only implemented several basic instructions instead of all of the RV32I instruction set. Aside from the book, our single-cycle soft processor could run most of the RV32I instructions. Due to the unoptimized nature of single-cycle processors, it runs on a slightly longer clock period.

III. System Design

A. Top-Level Overview

Figure 1 contains the top level design of our proposed soft processor. The processor core implements a classical five-stage architecture, which includes the Instruction Fetch (IF) stage, Instruction Decode (ID) stage, Execute (EX) stage, Memory (Mem) stage, and Write Back (WB) stage. We have divided the components into different colors. Pipeline registers, colorized as blue, divide the processor into five stages and carry signals from one stage to the next one. Following, we color-coded Read-Only Memory (Program memory) and Random-Access Memory (Data memory) as purple. The program memory only accepts aligned data reads, while the ROM can read unaligned accesses. The Synthesizer implements them as memory bits. The register files unit that contains all `x0` to `x31` registers has a yellow color. It outputs the register contents asynchronously while reading input data synchronously. The gray blocks are either ALU or adders, and the rest are red blocks combinational units.

B. Data Memory (Random Access Memory)

```
1  entity data_memory is
2          port (
3                  clk: in std_logic;
4                  data: in std_logic_vector (31 downto 0);
5                  address: in std_logic_vector (9 downto 0);
6                  write_en: in std_logic;
7                  byte_mode: in std_logic_vector (2 downto 0)
                       ;
8                  q: out std_logic_vector (31 downto 0):= x
                       "00000000"
9          );
10 end data_memory;
```

Listing 1. Fibonacci loop under 9999 in RISC-V Assembly.

We implemented the soft processor using a data memory that allows misaligned memory access. We divide this data memory (Random Access Memory) into four bytes. Listing 1 is the simplified entity declaration of the proposed data memory. There is a three-bits mode input, which we use to distinguish between word operation, signed halfword operation, unsigned halfword operation, signed byte operation, and unsigned byte operation. The control unit decides this signal.

The amount of misalignment can be detected using the two bits of the input address. For example, if there is no byte misalignment, the last two bits of the address will be `00`. On the other hand, it will be `11` if there is three-byte misaligned access. The data memory will write the leftover misaligned write data to the registers with the address next to the current one. For instance, suppose that there are two empty registers with '0' as the default value: `reg(0)` and `reg(1)`. If there is a write access to address 00000011_{16} with data $\texttt{AABBCCDD}_{16}$, `reg(0)` will contain $\texttt{DD}000000_{16}$ and `reg(1)` will contain $\texttt{00AABBCC}_{16}$.

C. Control Unit

We modeled the control unit as the type-decode stage and the control-decode stage as in Figure 2. In the first stage, the

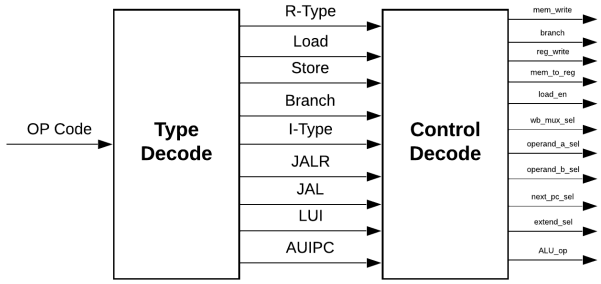

Fig. 2. The Control Unit of our proposed processor core.

TABLE I
PARTIAL TRUTH TABLE OF THE CONTROL UNIT.

Tipe Instruksi	Control Signal				
	MemWrite	Branch	RegWrite	MemtoReg	operand_B_sel
R-Type	0	0	1	0	0
Load	0	0	1	1	1
Store	1	0	0	0	1
Branch	0	1	0	0	0
I-Type	0	0	1	0	1
JALR	0	0	1	0	0
JAL	0	0	1	0	0
LUI	0	0	1	0	1
AUIPC	0	0	1	0	1

type-decoder will detect the type of the instruction by its OP code. After that, the control decoder will generate appropriate control signals depending on the instruction type.

There are several control signals, which include: `mem_write`, `branch`, `reg_write`, `mem_to_reg`, `load_en`, `wb_mux_sel`, `operand_a_sel`, `operand_b_sel`, `next_pc_sel`, `extend_sel`, and `ALU_op`. The data memory uses `mem_write` to decide when to write the input data to the chosen address. ALU uses the `branch` control signal to make branching decisions. `Reg_write` enables writing to register files during the Writeback stage. `Mem_to_reg` decides the output data of the multiplexer in the Writeback stage. `Operand_a_sel` and `operand_b_sel` select the ALU input data alongside the forwarding unit. `Next_pc_sel` decides the next program counter value depending on the instruction and branching conditions. We use `extend_sel` to select an appropriate immediate value from the immediate generator. Lastly, ALU uses the `ALU_op` control signal to decide the correct operation to process its operands. Table I further describes the partial behavior of the Control Unit.

D. Forwarding Unit

The processor needs a Forwarding Unit that forwards the data from a previous instruction into the ALU input of the current instruction to fix a data hazard. We implemented this unit like in Table II. The forwarding unit will supply the ALU with the future value of a register by forwarding the result in the Memory stage or the Writeback stage. Because the result in the memory stage is more recent, the forwarding unit will prioritize it first. We can detect a data hazard by comparing the register read address in the Execution stage to the register destination address in the Memory stage or the Writeback stage.

TABLE II
DESCRIPTION OF THE PROPOSED FORWARDING UNIT.

Mux Control	Source	Explanation
ForwardA = 00	ID/EX	Operand A comes from Register Files.
ForwardA = 10	EX/MEM	Operand A is forwarded from the EX/MEM stage.
ForwardA = 01	MEM/WB	Operand A is forwarded from the MEM/WB stage.
ForwardB = 00	ID/EX	Operand B comes from Register Files.
ForwardB = 10	EX/MEM	Operand B is forwarded from the EX/MEM stage.
ForwardB = 01	MEM/WB	Operand B is forwarded from the MEM/WB stage.

TABLE III
BEHAVIORAL DESCRIPTION OF THE HAZARD DETECTION UNIT.

No	Current Instruction	Previous Instruction	Behavior
1	Load Upper Immediate	Load	Stall if Rs1=Rd
2	Store	Load	Stall if Rs1=Rd
3	Load	Load	Stall if Rs1=Rd
4	Immediate-Type	Load	Stall if Rs1=Rd
5	AUIPC	Load	Stall if Rs1=Rd
6	Other instructions	Load	Stall if Rs1=Rd or Rs2=Rd
7	Other instructions	Other than Load	Don't Stall

Because the result in the memory stage is more recent, the forwarding unit will prioritize it first. We can detect a data hazard by comparing the register read address in the Execution stage to the register destination address in the Memory stage or the Writeback stage. If `EX.rs1 = MEM.rd` or `EX.rs2 = MEM.rd`, the forwarding unit will forward the value in the Memory stage into the ALU. If there is no data hazard in the execution stage, the forwarding unit will bypass the value in the Writeback stage into the ALU. Otherwise, the ALU will use the value provided by the register files.

E. Hazard Detection Unit

Table III is the behavior of our proposed soft processor's Hazard Detection Unit. The Hazard Detection Unit will stall the Instruction Fetch and Instruction Decode stage of the pipeline to fix a data hazard caused by the load instruction. We need this unit because the forwarding unit can't forward the data from the Memory stage of the pipeline that's still in process. The processor will run the generated stalls as NOPs.

IV. EXPERIMENTS

A. Iterative Fibonacci Sequence Program Execution

```
1      lui x15 0x2
2      addi x10 x15 1807
3  start:
4      addi x6,x0,1
5      addi x5,x0, 0
6  up:
7      add x7,x5,x6
8      bge x7,x10,start
9      sw x7,0x34(x0)
10     sw x6,0x30(x0)
11     lw x5,0x30(x0)
12     sw x7,0x30(x0)
13     lw x6,0x30(x0)
14     jal up
```

Listing 2. Fibonacci loop under 9999 in RISC-V Assembly.

We used the program in Listing 2 to test the soft processor. This program counts the Fibonacci number from 1 to the last number below 9999 and repeats the sequence from the beginning. To use it with the processor core, we have to

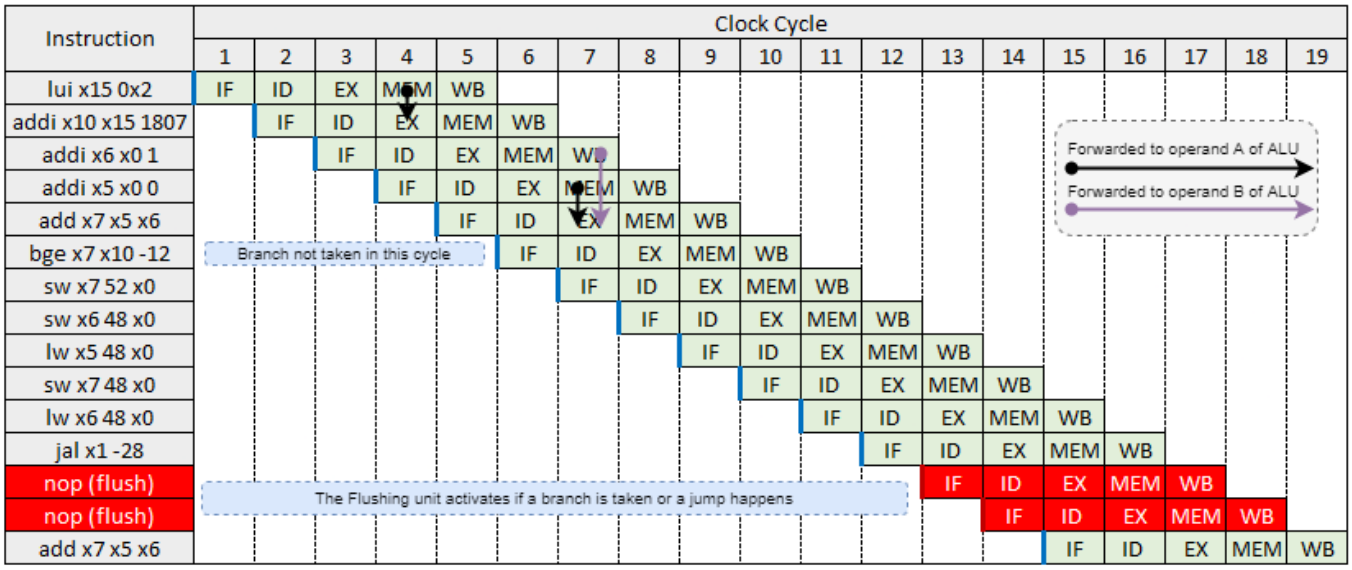

Fig. 3. The execution of the iterative Fibonacci number sequence program in Listing 2.

TABLE IV
PROGRAM MEMORY STORED VALUES

Address	Stored Value
0	00000000
4	000027b7
8	70f78513
12	00100313
16	00000293
20	006283b3
24	fea3dae3
28	02702c23
32	02602a23
36	03402283
40	02702a23
44	03402303
48	fe5ff0ef

use the RISC-V GCC toolchain to disassemble the RISC-V assembly code into its binary values[3]. The program memory stores these binary values as in Table IV. To test the functionality of the processor core, we compared the ModelSim simulation result to an external RISC-V simulator, such as Ripes[6]. After we finished the functional simulation, we did a timing simulation and then uploaded the processor core to an actual FPGA.

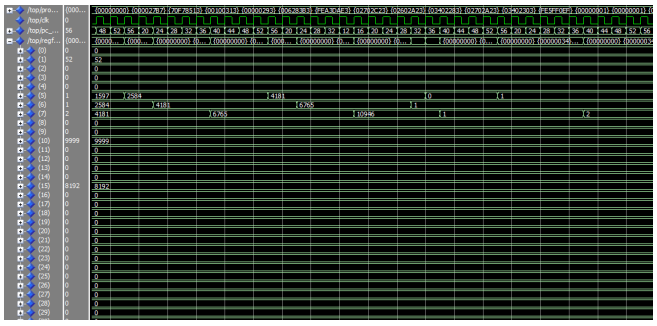

Fig. 4. The data values inside the registers, before and after branching.

The first iteration of the program is illustrated in Figure 3. We used some arrows to signify operand occurrences. For example, in the first and second instruction, the fist operand of instruction `addi x10 x15 1807` has not been rewritten back to the register files. In order to maintain a correct result, we have to forward the data from the Memory stage of the previous instruction.A similar thing happened between the fifth instruction (`add x7 x5 x6`) with the third instruction (`addi x6 x0 1`) and fourth instruction (`addi x5 x0 0`). Between the third and the fifth instructions, we need to forward the calculation result of register x6 in the Write Back stage into the second operand of the ALU. Between the fourth and the fifth instruction, we need to forward the ALU result in the Memory stage into the first operand of the ALU. The sixth operation of the program (`bge x7 x10 -12`) will restart the program and all the stored registers. In the first loop, the program will continue as usual, since the `x7` is still less than `x10`. We can see the change of register values before and after branching in Figure 4. When a jump or a branch is detected, such as the operation `jal x1 -28`, the flushing unit will clear the content of the IF/ID and ID/EX pipeline registers. This will cause the next two cycles to be executed as no-operations.

```
1 lw x5,20(x1)
2 and x4,x2,x5
3 or x5,x4,x6
4 lw x5,24(x2)
5 add x9,x4,x2
6 sub x1,x6,x7
```

Listing 3. Hazard Detection Unit Testing Program.

B. Recursive Fibonacci Sequence Program Execution

```
1 asm("auipc␣sp␣0x1");  // Set stack pointer
2 asm("jal␣main");          // call main
3 asm("NOP");         // End of Program
4
5 int fibbonacci(int n) {
6    if(n == 0){
7       return 0;
8    } else if(n == 1) {
```

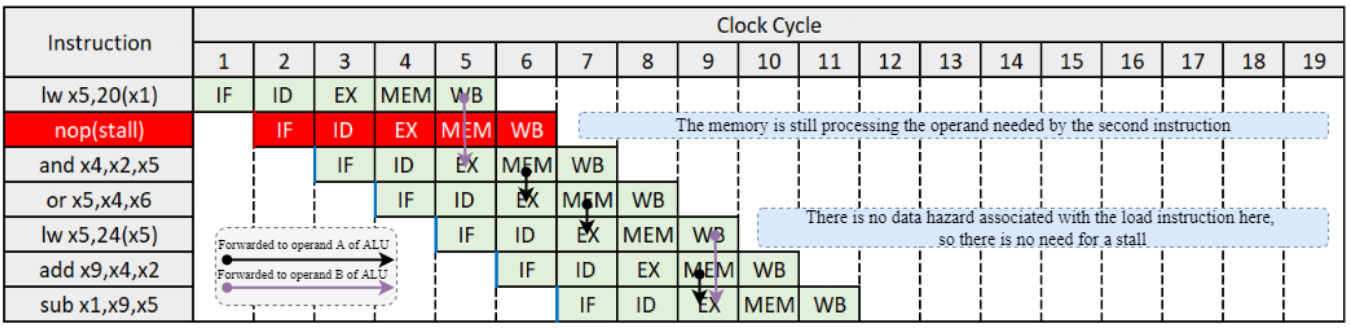

Fig. 5. The execution of the Hazard Detection Unit testing program in Listing 3.

```
9        return 1;
10     } else {
11        return (fibbonacci(n-1) + fibbonacci(n-2));
12     }
13  }
14
15  int main() {
16      int x = 0;
17       x = fibbonacci(5);
18      return 0;
19  }
```

Listing 4. Recursive Fibonacci Program.

We also tested the soft processor to run a recursive version of the Fibonacci sequence program. Unlike the iterative variant, this program requires the processor to utilize its data memory and stack pointer. We started by writing the program in C, like in Listing 4, compiling it into RISC-V machine code, and storing it in the program memory, similar to Table IV].

The GCC compiler uses register `x2` as the stack pointer by default. We declared the address 1000_{16} as the highest stack address at the beginning of the program. The stack pointer needs to be set accordingly to the necessities to avoid an overflow. Right after that, the program will jump into the main section of the algorithm. The processor will repeatedly run the recursive Fibonacci subroutine from the declared `N` number until it reaches the base case.

After around 15,000 clock cycles, the processor finishes computing the 12th Fibonacci number recursively. Unlike the iterative algorithm, which runs with linear complexity, this recursive variant requires much more computing power and resources because of its quadratic complexity nature. We can observe the result in Figure 6.

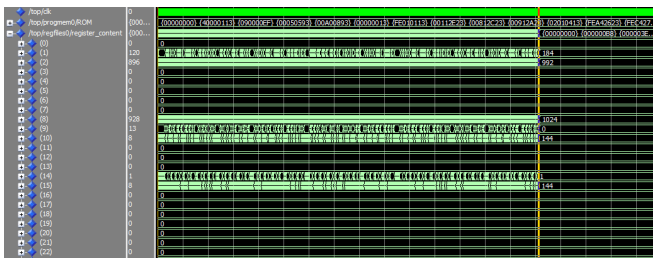

Fig. 6. The data values inside the registers, before and after branching.

TABLE V
RESOURCE ALLOCATION AFTER COMPILATION.

Device	EP4CE6E22C8
Total logic elements	2,115/6,273 (34%)
Total registers	558
Total memory bits	67,608/276,480 (7%)

C. Hazard Detection Unit Testing

To test the hazard detection unit, we use the program specified in Listing 3. When the unit detects a data hazard that can't be fixed merely by forwarding, it will stall the Instruction Fetch and Instruction Decode Stage by one cycle. This failure happens when an operand of current instruction needs the data that the previous load instruction is still processing. Since the required operand from data memory is still in process, we need to stall the processor by inserting NOPs. Figure 5 illustrates how the processor handles a data hazard caused by the load instruction. In clock cycle 4, the operand `x5` required by the instruction `add x4,x2,x5` is still unavailable. After adding a one-cycle NOP, ALU can access this operand by forwarding the value in the Writeback stage.

D. Resource Usage

Table V shows the resources needed to synthesize the soft processor into an FPGA. We need 1,693 logic elements on an EP4CE6E22C8 Altera FPGA to implement the combinational units and functional blocks. Also, we need 359 registers and 18,432 memory bits to integrate the pipeline registers, program memory, and data memory. If the program requires more memory, we can increase the size of our data memory and program memory, thus increasing the usage of memory bits. According to the synthesis results, the proposed processor core with five-stage pipelining consumes fewer resources than our previous single-cycle design. Such disparity can occur because we optimized the processor's overall architectural design, or the compiler and fitter could optimize it better.

E. Performance

Table VI compares the performance of our proposed core and several other RISC-V processor cores[7]. The proposed processor core can run up to 62.95 MHz, which slightly improves over our previous single-cycle design that runs at 46 MHz, which is a 37% increase in throughput. We also have improved our performance compared to the Maestro core, which inspired this core. However, other RISC-V cores

TABLE VI
PERFORMANCE COMPARISON BETWEEN THE PROPOSED CORE AND SEVERAL OTHER RISC-V CORES.

Core	LUT	FF	MaxFreq
This core	2115	558	62.95
Our single-cycle core	2660	1168	46
Maestro	3583	1513	31
PicoRV	1291	568	367
DarkRISCV	1177	246	150
VexRISCV	1993	1345	214
PicoRV32	1291	568	309
SERV	217	174	367
RPU	2943	1103	111
UE RISC-V	3676	2289	124
UE BiRISC-V	15667	6324	87

with five-stage pipelining, such as DarkRISCV, PicoRV32, and VexRISCV, are much faster than the proposed design. Those RISC-V cores are highly optimized and have been used in FPGAs for an actual projects. Also, we consider the throughput increase compared to the single-cycle RISC-V to be minimal because the theoretical improvement should be much higher.

We suspect that the bottleneck comes from an unoptimized flushing unit datapath. The critical datapath that determines the maximum frequency of the clock is the path that selects the next program counter input signal between all the possibilities. This dapath includes the path between the mem/wb register, the mem/wb multiplexer, the ALU multiplexer, ALU, ALU branch detector, jump/branch detector, PC multiplexer, and finally program counter. Generally, the ALU is the main factor of the delay as it contains a long combinational path. We can increase the maximum synthesis frequency of this processor core by improving the design of the ALU and balancing the levels of combinational logic.

F. Synthesis Into an FPGA

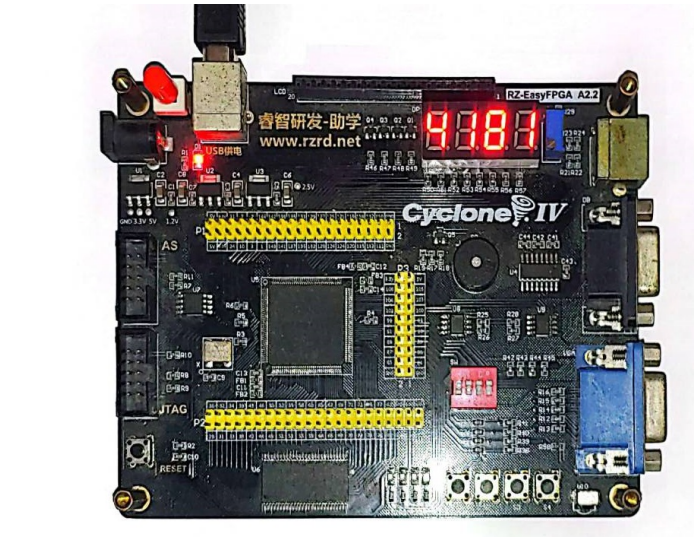

Fig. 7. The synthesized processor core running the program in Listing 2 on an EP4CE6E22C8 FPGA board.

After analyzing the resource usage and performance, we uploaded the proposed processor core into an FPGA. The interface that connects the soft processor and the FPGA will convert the value stored in register `x7` into a binary coded decimal that we will multiplex into the seven segments. We set the clock to 10 Hz using a clock divider so we can manually verify the Fibonacci sequence. The FPGA displayed the number 4181, one of the Fibonacci numbers, as shown in Figure 7.

V. SUMMARY

We propose a RISC-V soft processor with a five-stage pipelining for educational purposes. We implemented this processor core in VHDL Hardware Description Language and verified the logic using ModelSim. Then, we evaluated the resource usage, critical path, and maximum frequency compared to our previous single-cycle design and other soft processors. The processor core could run the test program when we run it on an EP4CE6E22C FPGA board.

The testing result shows that the proposed soft processor with five-stage pipelining runs around 37% faster than our previous single-cycle design. Despite the increase in throughput, we believe that we can improve this design further. We consider this improvement to be insignificant. The critical path that includes the datapath from MEM/WB register, ALU multiplexer, ALU, Branching unit, PC Multiplexer, and PC is probably the cause of this bottleneck. We can improve the maximum frequency of the proposed processor core by balancing the long combinational path of that critical path.

REFERENCES

[1] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146, Aug 2014. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html

[2] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The risc-v instruction set manual. volume 1: User-level isa, version 2.0," 2014.

[3] M. Poorhosseini, W. Nebel, and K. Grüttner, "A compiler comparison in the risc-v ecosystem," 09 2020.

[4] J. Vitor Rafael Chrisóstomo, "Entendendo a necessidade de uma isa aberta e implementação de um núcleo risc-v," *Github*, 2018. [Online]. Available: https://github.com/Artoriuz/maestro/blob/master/papers/

[5] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[6] M. B. Petersen, "Ripes: A visual computer architecture simulator," in *2021 ACM/IEEE Workshop on Computer Architecture Education (WCAE)*, 2021, pp. 1–8.

[7] N. M. Qui, C. H. Lin, and P. Chen, "Design and implementation of a 256-bit risc-v-based dynamically scheduled very long instruction word on fpga," *IEEE Access*, vol. 8, pp. 172 996–173 007, 2020.