Modeling the Execution Engine (Datapath) Portion of ARM Processor

Final Report

By: Saroj Shah

Date: May 06, 2025

Project Advisor: Professor James R. Moulic

Master's Project (IECE - 695)

Abstract:

This project focuses on modeling and simulating a simplified execution engine (datapath), main memory, and register-file of an ARM Cortex A76 processor using a selected subset of the ARMv8/ARMv7 instruction set architecture. It demonstrates the in-order execution of the integer control and datapath units by modeling the pipeline’s five stages, i.e., Instruction Fetch, Instruction Decode, Operand Fetch, Execute, and Write Back. The design involves the pre-loading of a short sequence of ARM instructions in machine code format into main memory (RAM) and then processes of loading data from memory and the register-file to and from the processor, and then evaluating those data within the processor's ALU, and retrieving the resultant output from it.

The code for the project is written first in the C programming language. Then the validated C model is synthesized into VHDL using Vitis High-Level Synthesis (HLS), and the resulting VHDL package is integrated into a testbench environment for simulation within Vivado. Although physical FPGA implementation is not part of this project, the behavioral simulation confirms correct pipeline behavior and control logic.

Introduction:

In modern computing systems, the processor’s execution engine or datapath is a critical component responsible for performing instruction execution and data processing tasks. It is responsible for fetching instructions and data from memory to the processor, decoding them (i.e., on those instructions), performing arithmetic and logical operations, and writing results back to memory (or any other destinations dictated). Thus, understanding the behavior of this unit is the fundamental key to processor architecture design and optimization.

The project focuses on modeling the execution unit of a simplified ARM processor, which is based on the LEGv8 instruction set architecture (a subset of ARMv8). The model implements the pipeline’s five stages, i.e., Instruction Fetch (IF), Instruction Decode (ID), Operand Fetch (OF), Execute (EX), and Write Back (WB) stages.

The processor model is initially developed in the C language and simulated using short sequences of ARMv7 assembly/machine code instructions. This C-based simulation allows functional verification of instruction flow, control logic, register file operations, and memory interactions. Then, the validated C model is synthesized into VHDL using Vivado High-Level Synthesis (HLS). After that, the generated VHDL hardware description is loaded into Vivado HLx and tested through behavioral simulation.

Although FPGA implementation of the VHDL representation of the model was not in the original scope of this project, the simulation-based approach provides a functional specification for verifying instruction flow, register behavior, and memory access. It also offers a clear framework for understanding ARM-based processor datapath and lays the foundation for potential future hardware implementation.

Background:

The ARM architecture is a family of RISC (Reduced Instruction Set Computer) architectures that are widely used in mobile devices, IoT systems, and even newer-generation MacBooks due to their power efficiency and high performance. ARM processors follow a load/store architecture with fixed-length instruction encoding, making them suitable for pipelined execution. To support educational use and simplify the complexity of ARMv8, a reduced and more approachable version known as the LEGv8 instruction set architecture was introduced for learning purposes.

In the processor, the control unit interprets the instruction and generates appropriate control signals that coordinate operations across the datapath components. The datapath consists of functional blocks such as the register file, ALU, and memory interface. Together, these units carry out the required operations as defined by the instruction being processed.

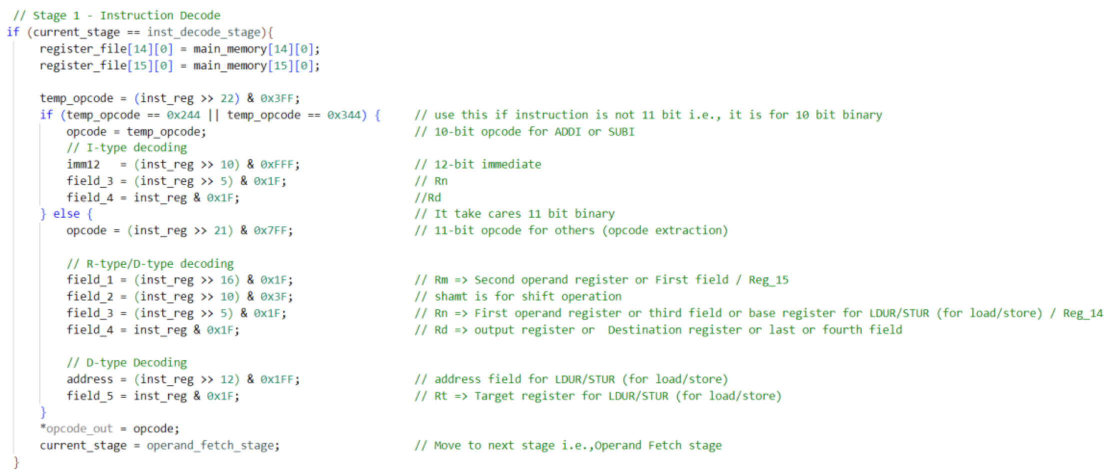
Since the model written in the C language can be directly translated into VHDL using Vitis/Vivado HLS, this project adopts the same approach to develop the processor model. The design is first simulated in C, and then a synthesizable VHDL representation is generated using HLS, which is subsequently verified through behavioral simulation in Vivado.

System Design and Methodology:

This project models a 32-bit processor based on the LEGv8 instruction set architecture. The processor's design includes both the control unit and datapath, working together to execute instructions sequentially, i.e., Instruction Fetch (IF), Instruction Decode (ID), Operand Fetch (OF), Execute (EX), and Write Back (WB).

- Instruction Format and Decoding:

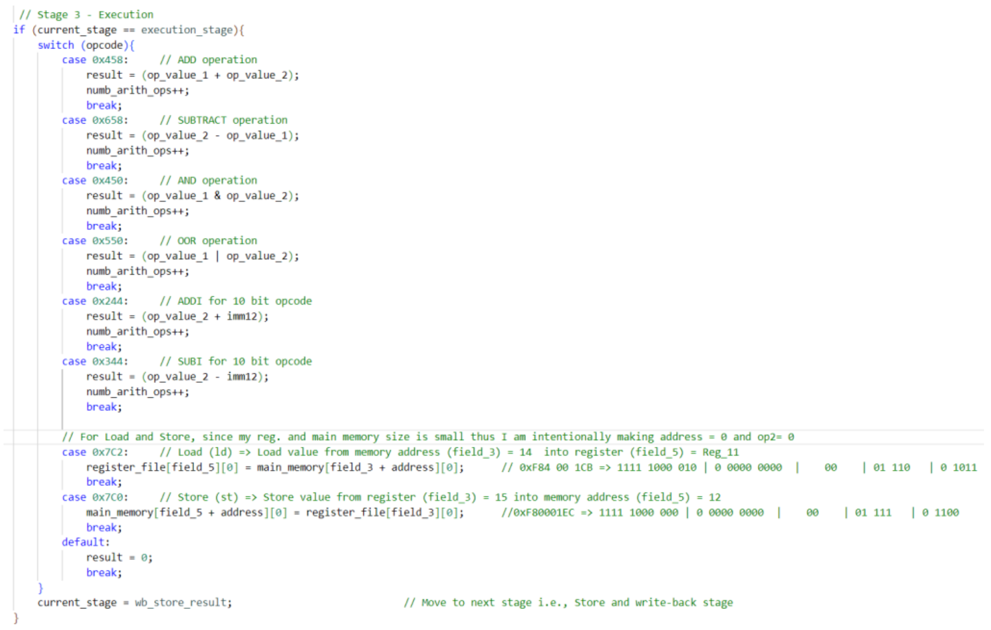
  The instructions are 32-bit wide and follow the LEGv8 instruction format structure. During the decode stage, each instruction is analyzed to extract the opcode and other fields, such as source and destination registers, immediate values, and memory addresses. Based on the opcode, control signals are derived to guide datapath operations, such as ALU control, memory access (read/write), and register selection.



```c
 // Stage 1 - Instruction Decode
if (current_stage == inst_decode_stage){
    register_file[14][0] = main_memory[14][0];
    register_file[15][0] = main_memory[15][0];

    temp_opcode = (inst_reg >> 22) & 0x3FF;
    if (temp_opcode == 0x244 || temp_opcode == 0x344) {     // use this if instruction is not 11 bit i.e., it is for 10 bit binary
        opcode = temp_opcode;                                // 10-bit opcode for ADDI or SUBI
        // I-type decoding
        imm12   = (inst_reg >> 10) & 0xFFF;                  // 12-bit immediate
        field_3 = (inst_reg >> 5) & 0x1F;                    // Rn
        field_4 = inst_reg & 0x1F;                           //Rd
    } else {                                                 // It take cares 11 bit binary
        opcode = (inst_reg >> 21) & 0x7FF;                   // 11-bit opcode for others (opcode extraction)

        // R-type/D-type decoding
        field_1 = (inst_reg >> 16) & 0x1F;                   // Rm => Second operand register or First field / Reg_15
        field_2 = (inst_reg >> 10) & 0x3F;                   // shamt is for shift operation
        field_3 = (inst_reg >> 5) & 0x1F;                    // Rn => First operand register or third field or base register for LDUR/STUR (for load/store) / Reg_14
        field_4 = inst_reg & 0x1F;                           // Rd => output register or  Destination register or last or fourth field

        // D-type Decoding
        address = (inst_reg >> 12) & 0x1FF;                  // address field for LDUR/STUR (for load/store)
        field_5 = inst_reg & 0x1F;                           // Rt => Target register for LDUR/STUR (for load/store)
    }
    *opcode_out = opcode;
    current_stage = operand_fetch_stage;                     // Move to next stage i.e.,Operand Fetch stage
}
```

- C-Based Simulation:

The processor is first modeled in the C language. A 2D memory array is used to store encoded instructions, while a register file is used to simulate the internal state. The instruction cycle is handled using a current_stage variable that mimics progression through the stages. This simulation validates the register interactions, memory access, and ALU operations for instructions such as ADD, SUB, ADDI, SUBI, LDUR, and STUR.

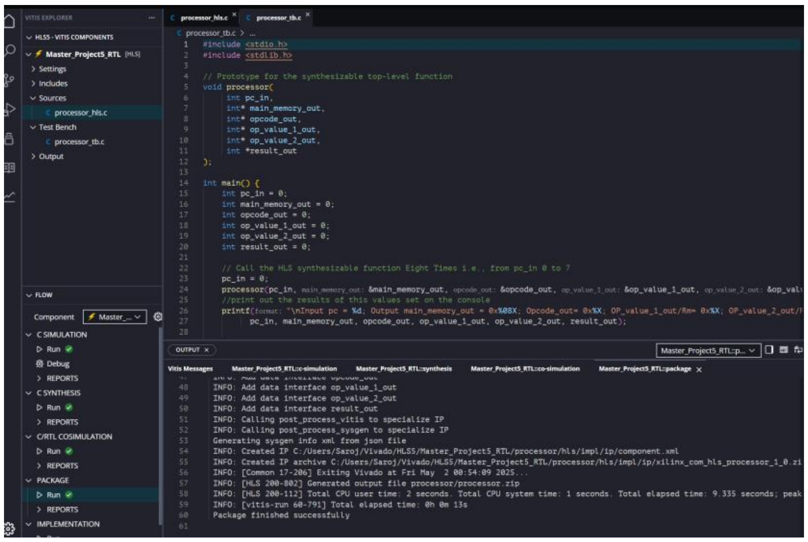


```c
 // Stage 3 - Execution
if (current_stage == execution_stage){
    switch (opcode){
        case 0x458:     // ADD operation
            result = (op_value_1 + op_value_2);
            numb_arith_ops++;
            break;
        case 0x658:     // SUBTRACT operation
            result = (op_value_2 - op_value_1);
            numb_arith_ops++;
            break;
        case 0x450:     // AND operation
            result = (op_value_1 & op_value_2);
            numb_arith_ops++;
            break;
        case 0x550:     // OOR operation
            result = (op_value_1 | op_value_2);
            numb_arith_ops++;
            break;
        case 0x244:     // ADDI for 10 bit opcode
            result = (op_value_2 + imm12);
            numb_arith_ops++;
            break;
        case 0x344:     // SUBI for 10 bit opcode
            result = (op_value_2 - imm12);
            numb_arith_ops++;
            break;

        // For Load and Store, since my reg. and main memory size is small thus I am intentionally making address = 0 and op2= 0
        case 0x7C2:     // Load (ld) => Load value from memory address (field_3) = 14  into register (field_5) = Reg_11
            register_file[field_5][0] = main_memory[field_3 + address][0];      // 0xF84 00 1CB => 1111 1000 010 | 0 0000 0000  |    00    | 01 110   | 0 1011
            break;
        case 0x7C0:     // Store (st) => Store value from register (field_3) = 15 into memory address (field_5) = 12
            main_memory[field_5 + address][0] = register_file[field_3][0];      //0xF80001EC => 1111 1000 000 | 0 0000 0000  |    00    | 01 111   | 0 1100
            break;
        default:
            result = 0;
            break;
    }
    current_stage = wb_store_result;                          // Move to next stage i.e., Store and write-back stage
}
```

Fig: Displaying Execution Stage

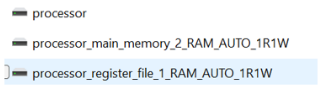
- High-Level Synthesis using Vitis HLS:

  After the initial C model validation, the C model is rewritten in an HLS-compatible form (i.e., using a call function or top-level function) for use with Vitis HLS. Then Vitis HLS converts the C source into a synthesizable VHDL design, preserving the C core simulation concept.



```
#include <stdio.h>
#include <stdlib.h>

// Prototype for the synthesizable top-level function
void processor(
        int pc_in,
        int* main_memory_out,
        int* opcode_out,
        int* op_value_1_out,
        int* op_value_2_out,
        int *result_out
);

int main() {
    int pc_in = 0;
    int main_memory_out = 0;
    int opcode_out = 0;
    int op_value_1_out = 0;
    int op_value_2_out = 0;
    int result_out = 0;

    // Call the HLS synthesizable function Eight Times i.e., from pc_in 0 to 7
    pc_in = 0;
    processor(pc_in, main_memory_out: &main_memory_out, opcode_out: &opcode_out, op_value_1_out: &op_value_1_out, op_value_2_out: &op_val
    //print out the results of this values set on the console
    printf(format: "\nInput pc = %d; Output main_memory_out = 0x%08X; Opcode_out= 0x%X; OP_value_1_out/Rm= 0x%X; OP_value_2_out/
           pc_in, main_memory_out, opcode_out, op_value_1_out, op_value_2_out, result_out);
```

```
INFO: Add data interface op_value_1_out
INFO: Add data interface op_value_2_out
INFO: Add data interface result_out
INFO: Calling post_process_vitis to specialize IP
INFO: Calling post_process_sysgen to specialize IP
Generating sysgen info xml from json file
INFO: Created IP C:/Users/Saroj/Vivado/HLS5/Master_Project5_RTL/processor/hls/impl/ip/component.xml
INFO: Created IP archive C:/Users/Saroj/Vivado/HLS5/Master_Project5_RTL/processor/hls/impl/ip/xilinx_com_hls_processor_1_0.zi
INFO: [Common 17-206] Exiting Vivado at Fri May  2 00:54:09 2025...
INFO: [HLS 200-802] Generated output file processor/processor.zip
INFO: [HLS 200-112] Total CPU user time: 2 seconds. Total CPU system time: 1 seconds. Total elapsed time: 9.335 seconds; peak
INFO: [vitis-run 60-791] Total elapsed time: 0h 0m 13s
Package finished successfully
```

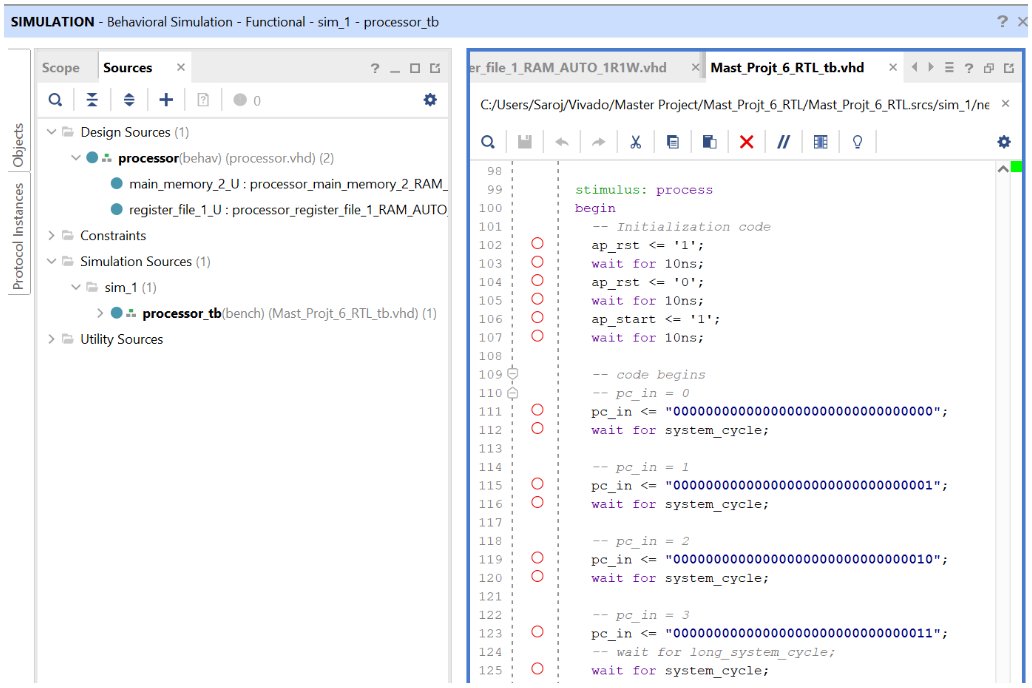


processor
processor_main_memory_2_RAM_AUTO_1R1W
processor_register_file_1_RAM_AUTO_1R1W

Fig: Depicting HLS Conversion

- Simulation in Vivado:

  Then the VHDL package generated by HLS is integrated into a testbench environment within Vivado. The testbench provides the clock, reset, and input stimuli such as pc_in, while observing outputs like main_memory_out, Rm, Rn, and result_out. The waveforms generated during simulation are used to verify correct instruction execution across all stages. This simulation confirms that the control and datapath units operate in a coordinated fashion and produce the expected results.


SIMULATION - Behavioral Simulation - Functional - sim_1 - processor_tb
Scope
Sources
Objects
Protocol Instances
Design Sources (1)
processor(behav) (processor.vhd) (2)
main_memory_2_U : processor_main_memory_2_RAM_
register_file_1_U : processor_register_file_1_RAM_AUTO
Constraints
Simulation Sources (1)
sim_1 (1)
processor_tb(bench) (Mast_Projt_6_RTL_tb.vhd) (1)
Utility Sources
er_file_1_RAM_AUTO_1R1W.vhd
Mast_Projt_6_RTL_tb.vhd

C:/Users/Saroj/Vivado/Master Project/Mast_Projt_6_RTL/Mast_Projt_6_RTL.srcs/sim_1/ne

```

stimulus: process
begin
  -- Initialization code
  ap_rst <= '1';
  wait for 10ns;
  ap_rst <= '0';
  wait for 10ns;
  ap_start <= '1';
  wait for 10ns;

  -- code begins
  -- pc_in = 0
  pc_in <= "00000000000000000000000000000000";
  wait for system_cycle;

  -- pc_in = 1
  pc_in <= "00000000000000000000000000000001";
  wait for system_cycle;

  -- pc_in = 2
  pc_in <= "00000000000000000000000000000010";
  wait for system_cycle;

  -- pc_in = 3
  pc_in <= "00000000000000000000000000000011";
  -- wait for long_system_cycle;
  wait for system_cycle;
```

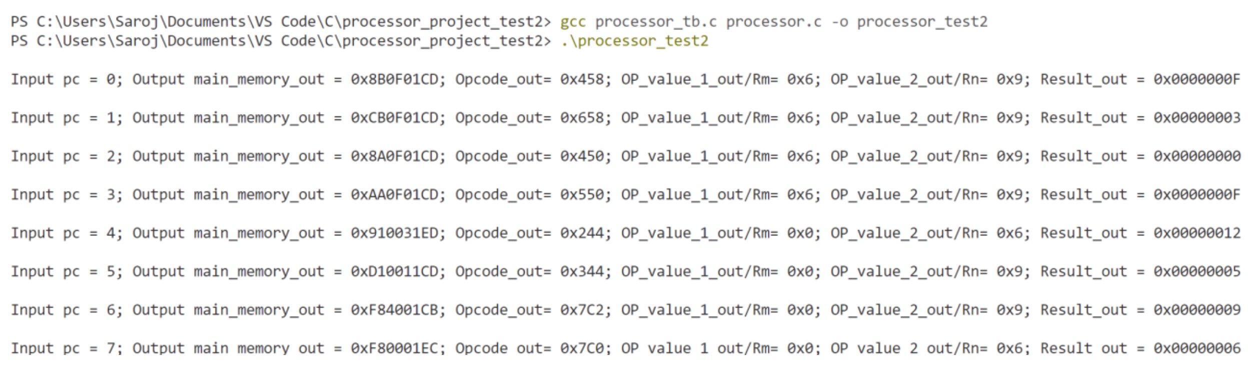
Fig: Displaying Vivado testbench and simulation

Although this project does not implement the design on an FPGA, it successfully demonstrates a complete high-level to RTL-level (register level data flow) design flow that can be applied on the FPGA platform in the future.

Materials:

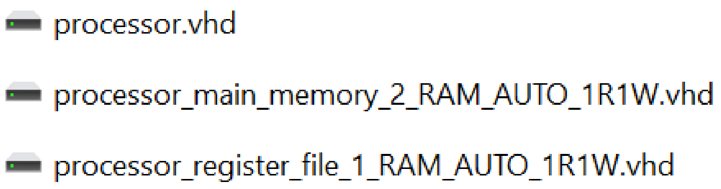
Computer

Results:



```
PS C:\Users\Saroj\Documents\VS Code\C\processor_project_test2> gcc processor_tb.c processor.c -o processor_test2
PS C:\Users\Saroj\Documents\VS Code\C\processor_project_test2> .\processor_test2

Input pc = 0; Output main_memory_out = 0x8B0F01CD; Opcode_out= 0x458; OP_value_1_out/Rm= 0x6; OP_value_2_out/Rn= 0x9; Result_out = 0x0000000F

Input pc = 1; Output main_memory_out = 0xCB0F01CD; Opcode_out= 0x658; OP_value_1_out/Rm= 0x6; OP_value_2_out/Rn= 0x9; Result_out = 0x00000003

Input pc = 2; Output main_memory_out = 0x8A0F01CD; Opcode_out= 0x450; OP_value_1_out/Rm= 0x6; OP_value_2_out/Rn= 0x9; Result_out = 0x00000000

Input pc = 3; Output main_memory_out = 0xAA0F01CD; Opcode_out= 0x550; OP_value_1_out/Rm= 0x6; OP_value_2_out/Rn= 0x9; Result_out = 0x0000000F

Input pc = 4; Output main_memory_out = 0x910031ED; Opcode_out= 0x244; OP_value_1_out/Rm= 0x0; OP_value_2_out/Rn= 0x6; Result_out = 0x00000012

Input pc = 5; Output main_memory_out = 0xD10011CD; Opcode_out= 0x344; OP_value_1_out/Rm= 0x0; OP_value_2_out/Rn= 0x9; Result_out = 0x00000005

Input pc = 6; Output main_memory_out = 0xF84001CB; Opcode_out= 0x7C2; OP_value_1_out/Rm= 0x0; OP_value_2_out/Rn= 0x9; Result_out = 0x00000009

Input pc = 7; Output main_memory_out = 0xF80001EC; Opcode_out= 0x7C0; OP_value_1_out/Rm= 0x0; OP_value_2_out/Rn= 0x6; Result_out = 0x00000006
```

Fig: Showing Result of C-Based Simulation

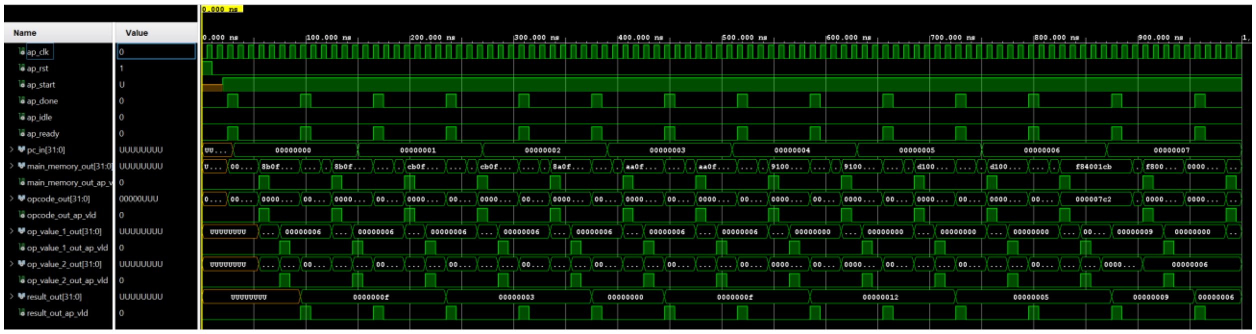


processor.vhd

processor_main_memory_2_RAM_AUTO_1R1W.vhd

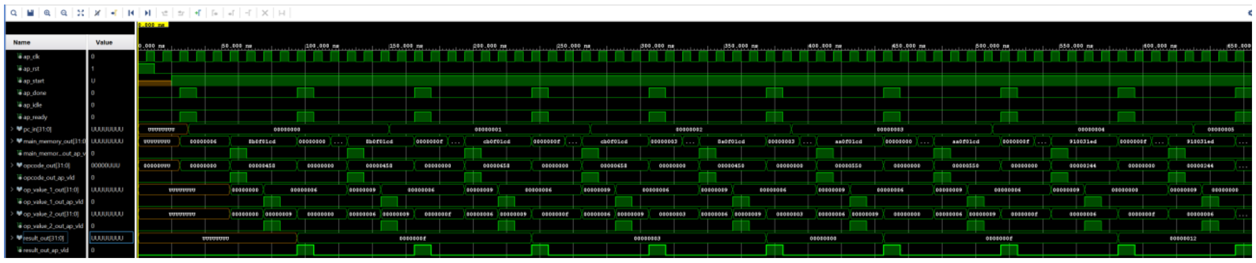
processor_register_file_1_RAM_AUTO_1R1W.vhd

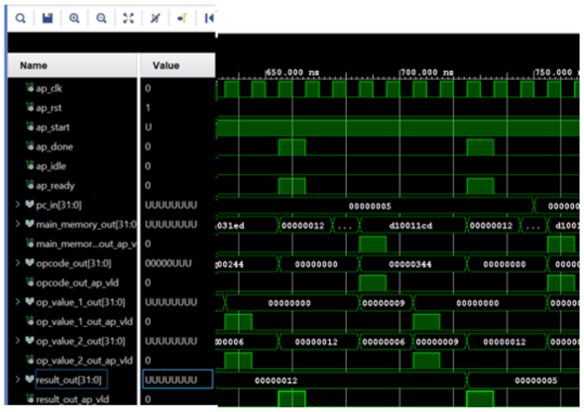
Fig: Showing the packages/files received after running Vitis



```
Name                      Value
ap_clk                    0
ap_rst                    1
ap_start                  U
ap_done                   0
ap_idle                   0
ap_ready                  0
pc_in[31:0]               UUUUUUUU
main_memory_out[31:0]     UUUUUUUU
main_memory_out_ap_v      0
opcode_out[31:0]          00000UUU
opcode_out_ap_vld         0
op_value_1_out[31:0]      UUUUUUUU
op_value_1_out_ap_vld     0
op_value_2_out[31:0]      UUUUUUUU
op_value_2_out_ap_vld     0
result_out[31:0]          UUUUUUUU
result_out_ap_vld         0
```

Fig: Showing Vivado Simulation Result





```
Name                      Value
ap_clk                    0
ap_rst                    1
ap_start                  U
ap_done                   0
ap_idle                   0
ap_ready                  0
pc_in[31:0]               UUUUUUUU
main_memory_out[31:0]     UUUUUUUU
main_memor...out_ap_v     0
opcode_out[31:0]          00000UUU
opcode_out_ap_vld         0
op_value_1_out[31:0]      UUUUUUUU
op_value_1_out_ap_vld     0
op_value_2_out[31:0]      UUUUUUUU
op_value_2_out_ap_vld     0
result_out[31:0]          UUUUUUUU
result_out_ap_vld         0
```

Fig: Displaying Vivado Simulation Result of ADD-SUBI

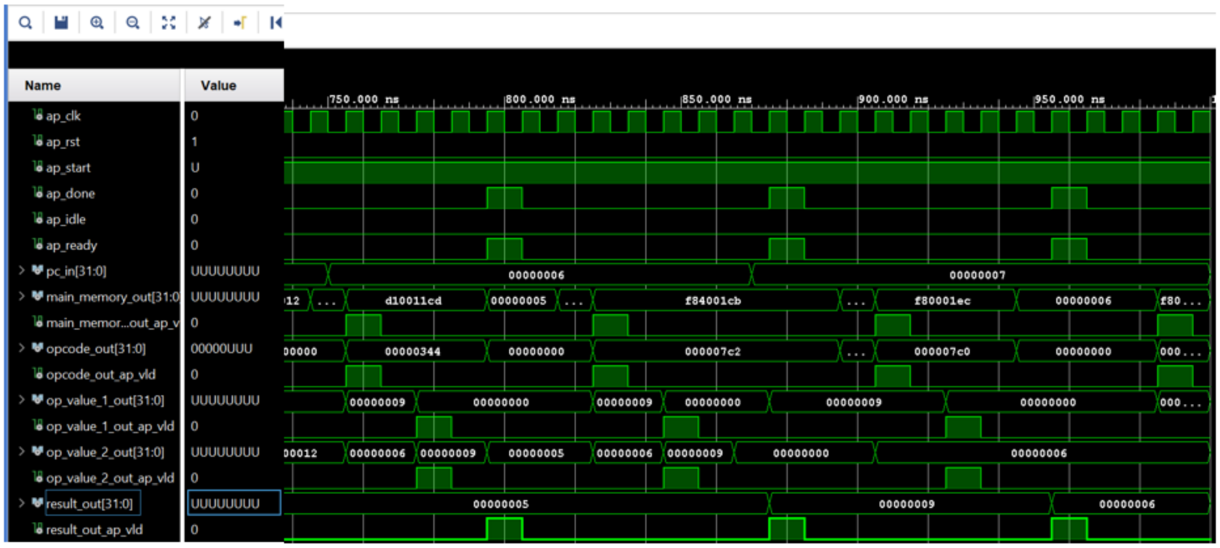


Fig: Showing Vivado Simulation Result of LDUR and STUR

After Simulating, synthesizing the C-based model using Vitis HLS, the resulting VHDL design was simulated using the Vivado environment. The testbench provided essential input signals such as clock, reset, and pc_in. During simulation, output signals such as main_memory_out, op_value_1_out (Rm), op_value_2_out (Rn), and result_out were monitored.

The waveform generated during simulation confirmed the correct operation of all five stages. Visually, we can see on the waveform that the pc_in signal progressed correctly, showing sequential instruction fetching. Additionally, the register values were updated during the correct write-back stage, and memory interactions through LDUR and STUR instructions accessed the correct addresses and data. Moreover, the ALU produced correct outputs during arithmetic operations, i.e., outputs matched the expected results for the given instructions: ADD, SUB, ADDI, SUBI, LDUR, and STUR.

Thus, these waveforms demonstrated that each instruction passed through the fetch, decode, operand fetch, execute, and write-back stages correctly. This validated both the control logic and the datapath design.

Recommendation for Future Work:

- In the future, to support the border or full implementation of the ARMv8/ARMv7 instruction set architecture, the current C code needs to be expanded. It should expand to incorporate floating-point operations, conditional branches, support for both integer and floating-point data types, and implementation of mode-switching capability between 32-bit (ARMv7) and 64-bit (ARMv8) execution states. Since these enhancements would significantly increase code complexity and size, careful attention would be required to maintain modularity and simulation accuracy.
- Furthermore, the design can be synthesized and implemented on an FPGA board. However, before deployment, the current simplified model must be evaluated for hardware feasibility, particularly with respect to resource availability on platforms such as the Basys3. Preliminary analysis suggests that, the current model, while simplified, is still not well-suited for implementation on the Basys3 board because it has a limited number logic resources (LUTs and flip-flops), it has limited number of I/O pins, and also lacks native 64-bit data path support, which makes mapping a wide datapath architecture (as used in ARMv8) resource-intensive. As a result, even for the implementation of this limited model, a more advanced FPGA (e.g., Artix-7 or Zynq-based boards) is required. So, if the system is expanded to support additional features (e.g., floating-point operations or 64-bit mode switching), the enhanced hardware platforms will be required. Thus, to support deployment on advanced FPGA platforms, the C code and synthesized VHDL will need to be modified to align with the target hardware's architecture and resource constraints.

Conclusion:

Hence, we say that this project successfully demonstrated the execution engine of a 32-bit ARM processor using the LEGv8 instruction set architecture. By implementing the five stages, i.e., Instruction Fetch, Instruction Decode, Operand Fetch, Execute, and Write Back, the ARM processor's core behavior was accurately simulated.

Even though the design was not implemented on physical hardware, this simulation closely emulated actual Arm processor behavior. The successful verification of instruction flow, register operations, and memory handling indicates that the model is accurate and suitable for future FPGA implementation.

Works Cited

Moulic, James R. Class Notes, "---."

Roth, Charles H., and Lizy Kurian John. "Design of a RISC Microprocessor." *Digital Systems Design Using VHDL*. Thomson Learning, Inc., 2008, pp. 429-467.

"Software Optimization Guide." *Arm Cortex-A75 Core*. Revision: r3p1, 2024 ARM Limited (or its affiliates), 2018, Issue 3.0.

"LEGv8 Reference Data Card ("Green Card")." Elsevier Inc., 2017 https://booksite.elsevier.com/9780128017333/content/Green%20Card.pdf

Patterson, David A., and John L. Hennessy. *Computer Organization and Design ARM Edition: The Hardware Software Interface*. 1st ed., Morgan Kaufmann, 2017.