EE533: Network Processor Design & Programming

Lab #7: ARM ISA Compatible Processor Core in Net FPGA

Instructor: Prof. Young Cho, PhD

Team Number: **#3**

Project Partners:

Member #1: **Sarthak Jain**

Member #2: **Archit Sethi**

Member #3: **Justin Santos**

Designed, Created, and Submitted by Team #3

University of Southern California

Los Angeles, CA 90007

Team #4 GitHub Repository Link: *Team#3 Link*

High-level Design Diagram (Datapath)

The datapath for the ARM-compatible processor was designed to support a basic subset of the ARMv7-A instruction set. The design follows a 5-stage pipeline architecture modeled on standard RISC pipeline design principles.

The five stages are:

Instruction Fetch (IF): Fetches the instruction from memory.
Instruction Decode (ID): Decodes the instruction and reads the necessary registers.
Execute (EX): Performs ALU operations or address calculations.
Memory Access (MEM): Reads or writes to memory for load/store instructions.
Write Back (WB): Writes results back into the register file.

The data-path includes the following main components:

1. Instruction Memory: Holds the machine code program.
2. Register File: 16 general-purpose registers (R0-R15) supporting ARM register conventions.
3. ALU: Performs arithmetic, logical, and shift operations.
4. Sign-Extend Unit: Extends 16-bit immediate values to 32 bits for certain instructions.
5. Data Memory: Used for load and store operations.
6. Control Unit: Generates control signals for ALU operations, memory access, and register writes.
7. Multiplexers: Select between register operands and immediate values for ALU input, and between ALU results and memory read data for register write-back.

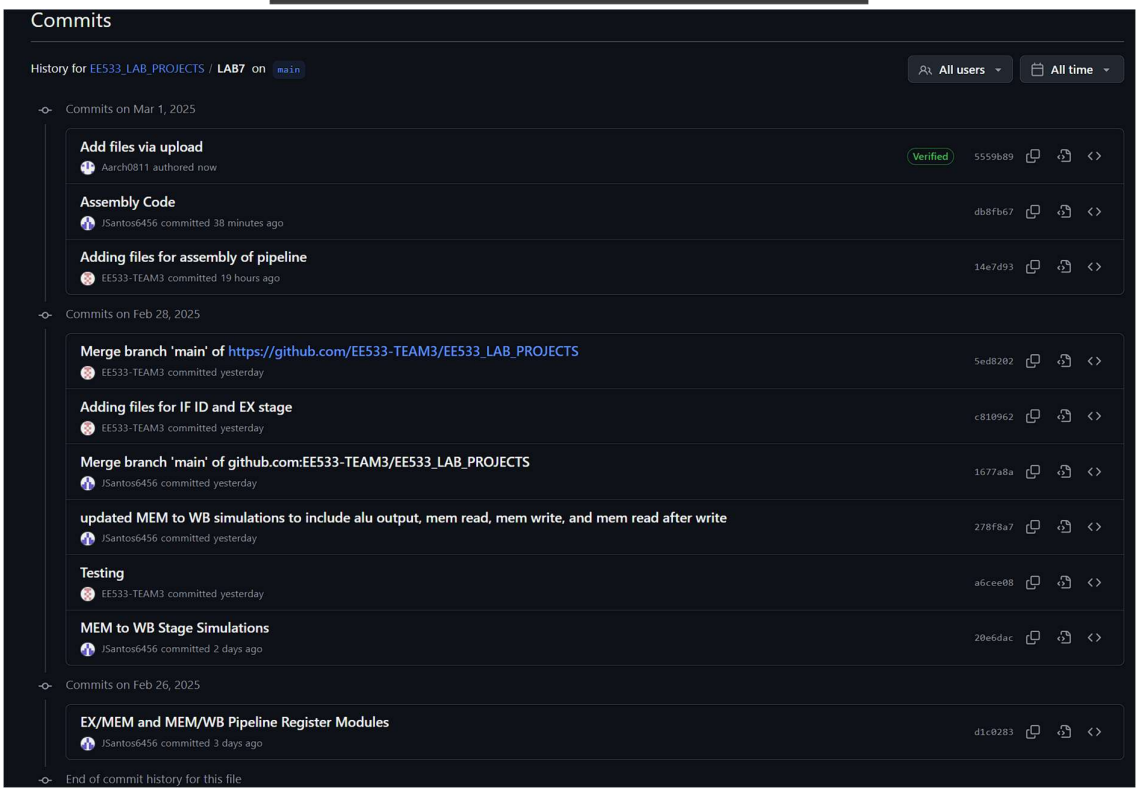
The processor supports:

1. Register-to-register arithmetic (ADD, SUB, AND, ORR)
2. Immediate arithmetic (ADDI, SUBI)
3. Load and store operations (LDR, STR)
4. Conditional branches (B)

Control signals such as ALUSrc, MemRead, MemWrite, RegWrite, and MemtoReg coordinate the operation of the datapath.

GitHub commits & Sequence of commands typed to interface

```
[team-3:fpga Lab6] ./pipeline read_imem 0
Found net device: nf2c0
Instruction 0 is: 0x40400000
[team-3:fpga Lab6] ./pipeline read_imem 1
Found net device: nf2c0
Instruction 1 is: 0x40600000
[team-3:fpga Lab6] ./pipeline read_dmem 0
Found net device: nf2c0
Found net device: nf2c0
Data at address 0 is: 0x0000000000000004
[team-3:fpga Lab6] ./pipeline read_dmem 4
Found net device: nf2c0
Found net device: nf2c0
Data at address 4 is: 0x0000000000000004
[team-3:fpga Lab6] ./pipeline write_imem 10 25
Wrote 0x19 to address: 10
[team-3:fpga Lab6] ./pipeline read_imem 10
Found net device: nf2c0
Instruction 10 is: 0x00000019
[team-3:fpga Lab6] ./pipeline write_dmem 50 36
Wrote 0x0000000000000024 to address: 50
[team-3:fpga Lab6] ./pipeline read_dmem 50
Found net device: nf2c0
Found net device: nf2c0
Data at address 50 is: 0x0000000000000024
```



NOTE: We pushed the codes and stuff from one system. So, name will be displayed once from the team#3 github account

```
./3cpu_pipeline read_dmem 4
./3cpu_pipeline read_dmem 4
./3cpu_pipeline read_dmem 4
./3cpu_pipeline read_dmem 0
nf_download 3cpu_
nf_download 3cpu_lab6.bit
./3cpu_pipeline read_dmem 4
./3cpu_pipeline write_dmem 4 0
./3cpu_pipeline read_dmem 4
./3cpu_pipeline read_dmem 4
./3cpu_pipeline write_dmem 4 100
./3cpu_pipeline read_dmem 4
./3cpu_pipeline write_dmem 0 100
./3cpu_pipeline read_dmem 4
./3cpu_pipeline read_dmem 0
./3cpu_pipeline read_dmem 4
./3cpu_pipeline write_dmem 0 77
./3cpu_pipeline read_dmem 4
./3cpu_pipeline read_dmem 4
./3cpu_pipeline read_dmem 4
nf_download 3cpu_lab6.bit
./3cpu_pipeline read_dmem 4
./3cpu_pipeline write_dmem 0 77
./3cpu_pipeline read_dmem 4
./3cpu_pipeline read_dmem 4
nf_download 3cpu_lab6.bit
./3cpu_pipeline write_dmem 4 6
./3cpu_pipeline read_dmem 4
./3cpu_pipeline read_dmem 0
nf_download 3cpu_lab6.bit
./3cpu_pipeline read_dmem 0
./3cpu_pipeline read_dmem 4
./3cpu_pipeline read_dmem 4
./3cpu_pipeline read_dmem 0
./3cpu_pipeline write_dmem 0 99
./3cpu_pipeline read_dmem 4
./3cpu_pipeline read_dmem 0
./3cpu_pipeline read_dmem 1
./3cpu_pipeline read_dmem 2
./3cpu_pipeline read_dmem 3
./3cpu_pipeline read_dmem 4
./3cpu_pipeline read_dmem 5
./3cpu_pipeline read_imem 0
./3cpu_pipeline read_imem 1
./3cpu_pipeline read_dmem 4
nf_download lab6_complete.bit
./3cpu_pipeline read_imem 0
./3cpu_pipeline read_imem 1
./3cpu_pipeline read_imem 2
```

Readable Assembly code (Bubble Sort)

1. Assembly Code sort.c

```
int main() {
    int array[10] = {323, 123, -455, 2, 98, 125, 10, 65, -56, 0};
    int i, j, swap;
    for (i = 0 ; i < 10; i++) {
    for (j = i+1 ; j < 10 ; j++) {
    if (array[j] < array[i]) {
    swap = array[j];
    array[j] = array[i];
    array[i] = swap;
    }
    }
    }
    }
```

2. Build and convert armv8.sh

```
#!/bin/bash

# Check if the source file was passed as an argument
if [ -z "$1" ]; then
  echo "Usage: $0 <source_file.c>"
  exit 1
fi

SOURCE_FILE=$1
BASE_NAME=$(basename "$SOURCE_FILE" .c)  # Extract the base name without extension

# Step 1: Generate Assembly Code
echo "Generating Assembly Code for $SOURCE_FILE..."
arm-none-eabi-gcc -S -march=armv8-a "$SOURCE_FILE" -o "$BASE_NAME.s"

# Step 2: Convert Assembly Code to Machine Code
echo "Converting Assembly Code to Machine Code..."
arm-none-eabi-as -o "$BASE_NAME.o" "$BASE_NAME.s"

# Step 3: Produce ELF file
echo "Producing ELF file..."
arm-none-eabi-gcc -o "$BASE_NAME.elf" "$BASE_NAME.o" -nostdlib

# Step 4: View ELF in readable hex format (Disassembly)
echo "Disassembling ELF to readable format..."
arm-none-eabi-objdump -d "$BASE_NAME.elf" > "$BASE_NAME"_disassembly.txt

# Step 5: Generate Machine Binary
echo "Generating Machine Binary..."
arm-none-eabi-objcopy -O binary "$BASE_NAME.elf" "$BASE_NAME.bin"

# Step 6: Generate Binary Output Text in Binary and Hex Formats
echo "Generating Binary Output in Binary and Hex formats..."

# Binary Output
xxd -b "$BASE_NAME.bin" > "$BASE_NAME"_binary.txt

# Hexadecimal Output
xxd "$BASE_NAME.bin" > "$BASE_NAME"_hexadecimal.txt

# Inform the user that the process is complete
echo "Process complete. Files generated:"
echo "$BASE_NAME"_disassembly.txt "(disassembly of ELF)"
echo "$BASE_NAME"_binary.txt "(binary output)"
echo "$BASE_NAME"_hexadecimal.txt "(hexadecimal output)"
```

3. Sort.s

```
        .arch armv8-a
        .eabi_attribute 20, 1
        .eabi_attribute 21, 1
        .eabi_attribute 23, 3
        .eabi_attribute 24, 1
        .eabi_attribute 25, 1
        .eabi_attribute 26, 1
        .eabi_attribute 30, 6
        .eabi_attribute 34, 1
        .eabi_attribute 18, 4
        .file   "sort.c"
        .text
        .section        .rodata
        .align  2
.LC0:
        .word   323
        .word   123
        .word   -455
        .word   2
        .word   98
        .word   125
        .word   10
        .word   65
        .word   -56
        .word   0
        .text
        .align  2
        .global main
        .syntax unified
        .arm
        .fpu softvfp
        .type   main, %function
main:
        @ args = 0, pretend = 0, frame = 56
        @ frame_needed = 1, uses_anonymous_args = 0
        push    {fp, lr}
        add     fp, sp, #4
        sub     sp, sp, #56
        movw    r3, #:lower16:.LC0
```

```
39              movw    r3, #:lower16:.LC0
40              movt    r3, #:upper16:.LC0
41              sub     ip, fp, #56
42              mov     lr, r3
43              ldmia   lr!, {r0, r1, r2, r3}
44              stmia   ip!, {r0, r1, r2, r3}
45              ldmia   lr!, {r0, r1, r2, r3}
46              stmia   ip!, {r0, r1, r2, r3}
47              ldm     lr, {r0, r1}
48              stm     ip, {r0, r1}
49              mov     r3, #0
50              str     r3, [fp, #-8]
51              b       .L2
52      .L6:
53              ldr     r3, [fp, #-8]
54              add     r3, r3, #1
55              str     r3, [fp, #-12]
56              b       .L3
57      .L5:
58              ldr     r3, [fp, #-12]
59              lsl     r3, r3, #2
60              sub     r3, r3, #4
61              add     r3, r3, fp
62              ldr     r2, [r3, #-52]
63              ldr     r3, [fp, #-8]
64              lsl     r3, r3, #2
65              sub     r3, r3, #4
66              add     r3, r3, fp
67              ldr     r3, [r3, #-52]
68              cmp     r2, r3
69              bge     .L4
70              ldr     r3, [fp, #-12]
71              lsl     r3, r3, #2
72              sub     r3, r3, #4
73              add     r3, r3, fp
74              ldr     r3, [r3, #-52]
75              str     r3, [fp, #-16]
76              ldr     r3, [fp, #-8]
77              lsl     r3, r3, #2
78              sub     r3, r3, #4
79              add     r3, r3, fp
80              ldr     r2, [r3, #-52]
81              ldr     r3, [fp, #-12]
82              lsl     r3, r3, #2
83              sub     r3, r3, #4
84              add     r3, r3, fp
85              str     r2, [r3, #-52]
```

```
69              bge     .L4
70              ldr     r3, [fp, #-12]
71              lsl     r3, r3, #2
72              sub     r3, r3, #4
73              add     r3, r3, fp
74              ldr     r3, [r3, #-52]
75              str     r3, [fp, #-16]
76              ldr     r3, [fp, #-8]
77              lsl     r3, r3, #2
78              sub     r3, r3, #4
79              add     r3, r3, fp
80              ldr     r2, [r3, #-52]
81              ldr     r3, [fp, #-12]
82              lsl     r3, r3, #2
83              sub     r3, r3, #4
84              add     r3, r3, fp
85              str     r2, [r3, #-52]
86              ldr     r3, [fp, #-8]
87              lsl     r3, r3, #2
88              sub     r3, r3, #4
89              add     r3, r3, fp
90              ldr     r2, [fp, #-16]
91              str     r2, [r3, #-52]
92      .L4:
93              ldr     r3, [fp, #-12]
94              add     r3, r3, #1
95              str     r3, [fp, #-12]
96      .L3:
97              ldr     r3, [fp, #-12]
98              cmp     r3, #9
99              ble     .L5
100             ldr     r3, [fp, #-8]
101             add     r3, r3, #1
102             str     r3, [fp, #-8]
103     .L2:
104             ldr     r3, [fp, #-8]
105             cmp     r3, #9
106             ble     .L6
107             mov     r3, #0
108             mov     r0, r3
109             sub     sp, fp, #4
110             @ sp needed
111             pop     {fp, pc}
112             .size   main, .-main
113             .ident  "GCC: (GNU Arm Embedded Toolchain 10.3-2021.07) 10.3.1 20210621 (release)"
```

4. Sort binary.txt

```
00000000: 00000000 01001000 00101101 11101001 00000100 10110000  .H-...
00000006: 10001101 11100010 00111000 11010000 01001101 11100010  ..8.M.
0000000c: 00011000 00110001 00001000 11100011 00000000 00110000  .1...0
00000012: 01000000 11100011 00111000 11000000 01001011 11100010  @.8.K.
00000018: 00000011 11100000 10100000 11100001 00001111 00000000  ......
0000001e: 10111110 11101000 00001111 00000000 10101100 11101000  ......
00000024: 00001111 00000000 10111110 11101000 00001111 00000000  ......
0000002a: 10101100 11101000 00000011 00000000 10011110 11101000  ......
00000030: 00000011 00000000 10001100 11101000 00000000 00110000  .....0
00000036: 10100000 11100011 00001000 00110000 00001011 11100101  ...0..
0000003c: 00101110 00000000 00000000 11101010 00001000 00110000  .....0
00000042: 00011011 11100101 00000001 00110000 10000011 11100010  ...0..
00000048: 00001100 00110000 00001011 11100101 00100100 00000000  .0..$.
0000004e: 00000000 11101010 00001100 00110000 00011011 11100101  ...0..
00000054: 00000011 00110001 10100000 11100001 00000100 00110000  .1...0
0000005a: 01000011 11100010 00001011 00110000 10000011 11100000  C..0..
00000060: 00110100 00100000 00010011 11100101 00001000 00110000  4 ...0
00000066: 00011011 11100101 00000011 00110001 10100000 11100001  ...1..
0000006c: 00000100 00110000 01000011 11100010 00001011 00110000  .0C..0
00000072: 10000011 11100000 00110100 00110000 00010011 11100101  ..40..
00000078: 00000011 00000000 01010010 11100001 00010101 00000000  ..R...
0000007e: 00000000 10101010 00001100 00110000 00011011 11100101  ...0..
00000084: 00000011 00110001 10100000 11100001 00000100 00110000  .1...0
0000008a: 01000011 11100010 00001011 00110000 10000011 11100000  C..0..
00000090: 00110100 00110000 00010011 11100101 00010000 00110000  40...0
00000096: 00001011 11100101 00001000 00110000 00011011 11100101  ...0..
0000009c: 00000011 00110001 10100000 11100001 00000100 00110000  .1...0
000000a2: 01000011 11100010 00001011 00110000 10000011 11100000  C..0..
000000a8: 00110100 00100000 00010011 11100101 00001100 00110000  4 ...0
000000ae: 00011011 11100101 00000011 00110001 10100000 11100001  ...1..
000000b4: 00000100 00110000 01000011 11100010 00001011 00110000  .0C..0
000000ba: 10000011 11100000 00110100 00100000 00000011 11100101  ..4 ..
000000c0: 00001000 00110000 00011011 11100101 00000011 00110001  .0...1
000000c6: 10100000 11100001 00000100 00110000 01000011 11100010  ...0C.
000000cc: 00001011 00110000 10000011 11100000 00010000 00100000  .0...
000000d2: 00011011 11100101 00110100 00100000 00000011 11100101  ..4 ..
000000d8: 00001100 00110000 00011011 11100101 00000001 00110000  .0...0
```

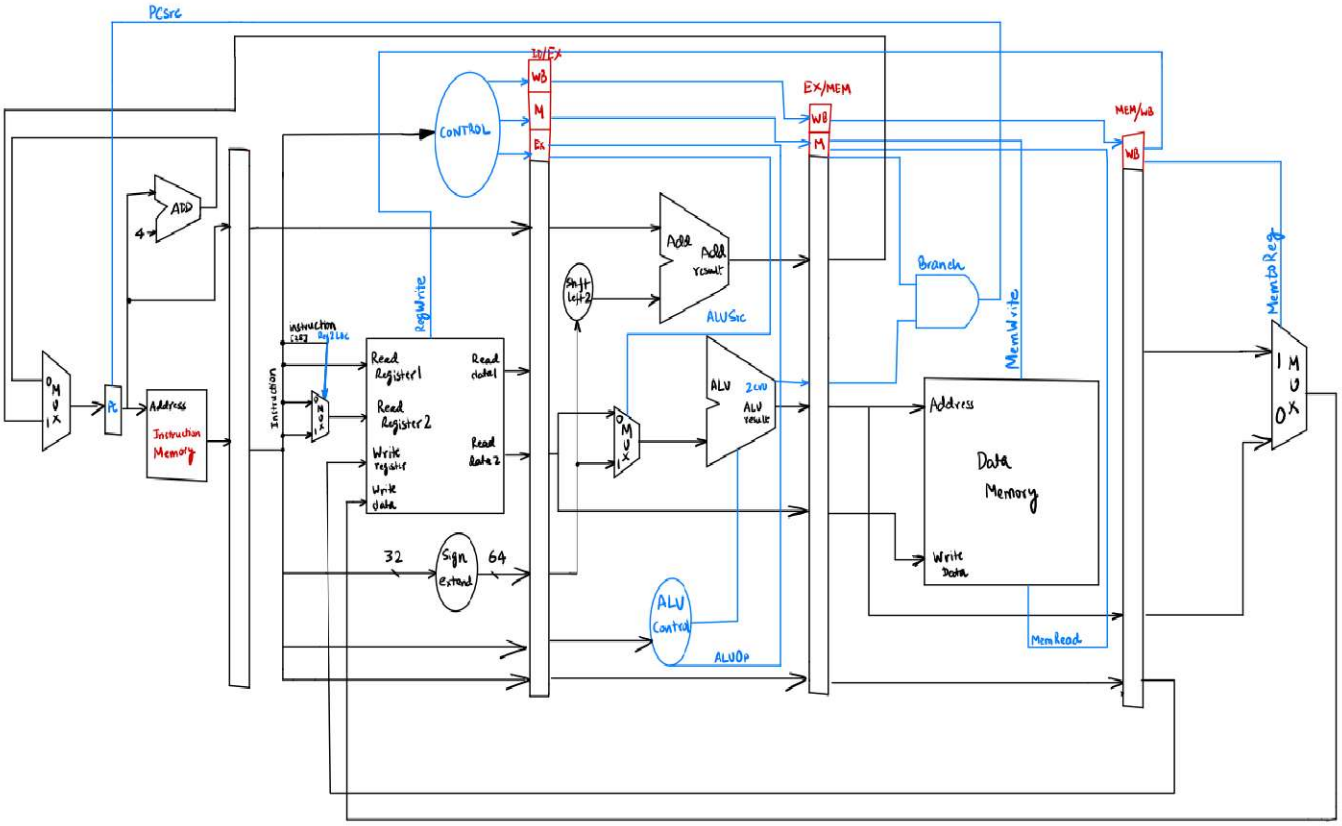
5. Sort headecimal.txt

```
00000000: 0048 2de9 04b0 8de2 38d0 4de2 1831 08e3  .H-.....8.M..1..
00000010: 0030 40e3 38c0 4be2 03e0 a0e1 0f00 bee8  .0@.8.K.........
00000020: 0f00 ace8 0f00 bee8 0f00 ace8 0300 9ee8  ................
00000030: 0300 8ce8 0030 a0e3 0830 0be5 2e00 00ea  .....0...0......
00000040: 0830 1be5 0130 83e2 0c30 0be5 2400 00ea  .0...0...0..$...
00000050: 0c30 1be5 0331 a0e1 0430 43e2 0b30 83e0  .0...1...0C..0..
00000060: 3420 13e5 0830 1be5 0331 a0e1 0430 43e2  4 ...0...1...0C.
00000070: 0b30 83e0 3430 13e5 0300 52e1 1500 00aa  .0..40....R.....
00000080: 0c30 1be5 0331 a0e1 0430 43e2 0b30 83e0  .0...1...0C..0..
00000090: 3430 13e5 1030 0be5 0830 1be5 0331 a0e1  40...0...0...1..
000000a0: 0430 43e2 0b30 83e0 3420 13e5 0c30 1be5  .0C..0..4 ...0..
000000b0: 0331 a0e1 0430 43e2 0b30 83e0 3420 03e5  .1...0C..0..4 ..
000000c0: 0830 1be5 0331 a0e1 0430 43e2 0b30 83e0  .0...1...0C..0..
000000d0: 1020 1be5 3420 03e5 0c30 1be5 0130 83e2  . ..4 ...0...0..
000000e0: 0c30 0be5 0c30 1be5 0900 53e3 d7ff ffda  .0...0....S.....
000000f0: 0830 1be5 0130 83e2 0830 0be5 0830 1be5  .0...0...0...0..
00000100: 0900 53e3 cdff ffda 0030 a0e3 0300 a0e1  ..S......0......
00000110: 04d0 4be2 0088 bde8 4301 0000 7b00 0000  ..K.....C...{...
00000120: 39fe ffff 0200 0000 6200 0000 7d00 0000  9.......b...}...
00000130: 0a00 0000 4100 0000 c8ff ffff 0000 0000  ....A...........
```

This project implements a 5-stage pipelined processor based on a subset of the ARMv8 (LEGv8) instruction set architecture (ISA), tailored to execute a simple bubble sort program. The design follows the principles outlined in Patterson and Hennessy's "Computer Organization and Design", with modifications to account for the LEGv8 instruction format and NetFPGA deployment requirements.

The pipeline stages implemented are:

- Instruction Fetch (IF) — Fetches 32-bit instruction from instruction memory.
- Instruction Decode (ID) — Decodes instruction, generates control signals, reads registers, and sign-extends immediate.
- Execution (EX) — Performs arithmetic, logic, or address calculations via the ALU.
- Memory Access (MEM) — Reads or writes to data memory.
- Writeback (WB) — Writes results back to the register file.



Instruction Subset Supported

The following LEGv8 instructions were fully implemented and supported in the data-path to run the bubble sort algorithm:

Instruction	Description
ADD	Add two registers
SUB	Subtract two registers
AND	Bitwise AND
ORR	Bitwise OR
ADDI	Add immediate
SUBI	Subtract immediate
ANDI	Bitwise AND with immediate
ORRI	Bitwise OR with immediate
LDUR	Load from data memory
STUR	Store to data memory
CBZ	Conditional branch if zero
B	Unconditional branch
BITWISE_XNOR	Custom instruction added for testing
COMPARE	Custom instruction for comparisons
LOGICAL_SHIFT_LEFT	Custom shift instruction
LOGICAL_SHIFT_RIGHT	Custom shift instruction
SHIFT_THEN_COMPARE	Custom composite instruction
SUBSTRING_COMPARISON	(Placeholder for future extension)

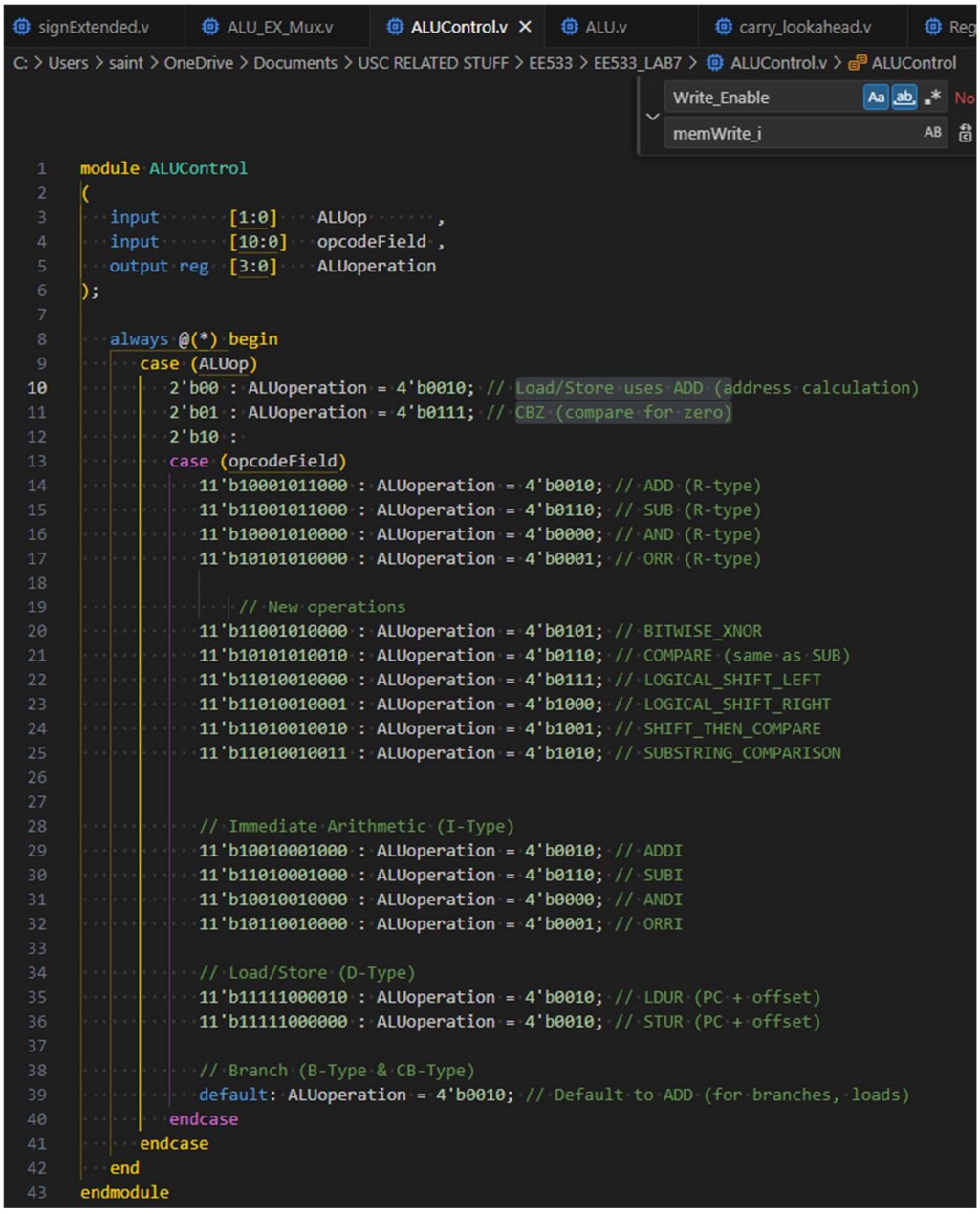
The custom instructions were added to extend the basic processor for future projects and may not be used directly in the bubble sort test.

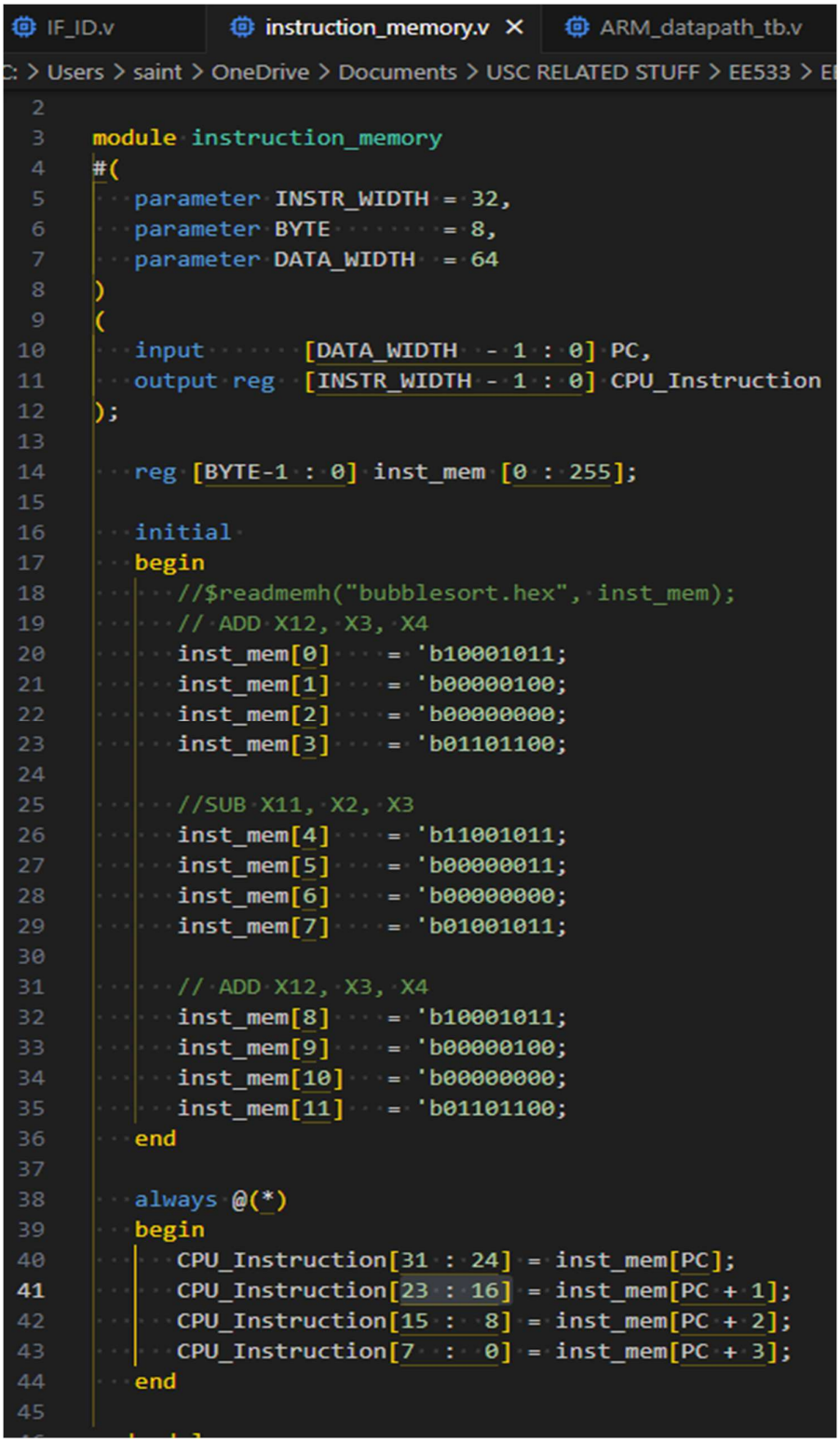
Key Datapath Modifications

To support these instructions, the following **major changes** were made to the initial basic pipeline:

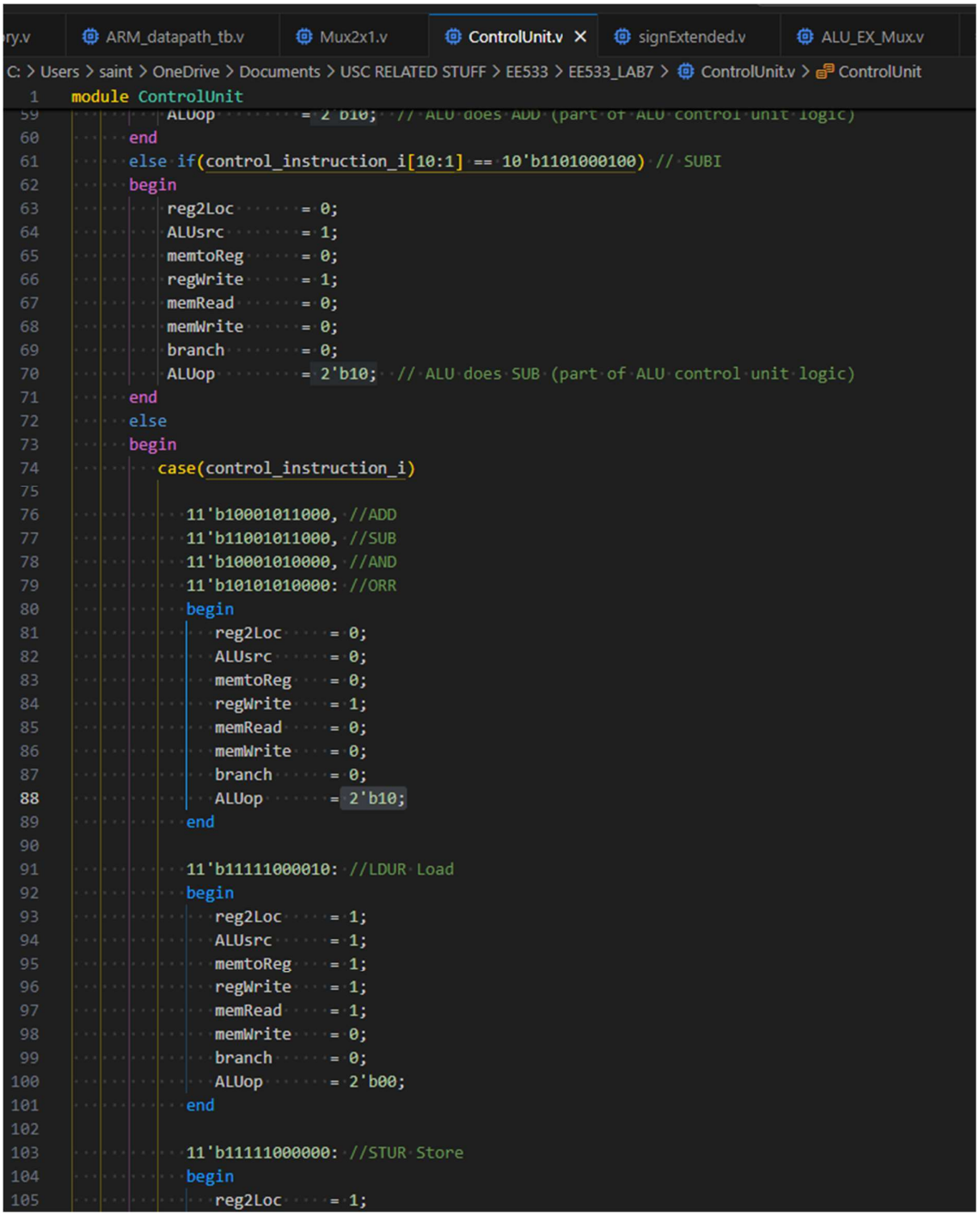
- **ALU Control Module:** Added logic to decode the LEGv8 opcode and generate correct ALU operation codes.
- **Control Unit:** Modified to generate all control signals for instruction types including Load/Store, Branches, and Arithmetic.
- **Pipeline Registers:** Created proper IF/ID, ID/EX, EX/MEM, and MEM/WB registers to support forwarding, stalling, and flushing.
- **Hazard Detection Unit:** Added to detect data hazards (load-use hazards).
- **Forwarding Unit:** Added to resolve data dependencies between EX, MEM, and WB stages.
- **Instruction Memory Parsing:** Modified to support reading instruction binaries in LEGv8 format.
- **Data Memory Module:** Created a separate memory module to store and sort data arrays.
- **Sign Extension Module:** Added support for immediate-based instructions, properly handling sign extension for branch offsets and load/store immediates.

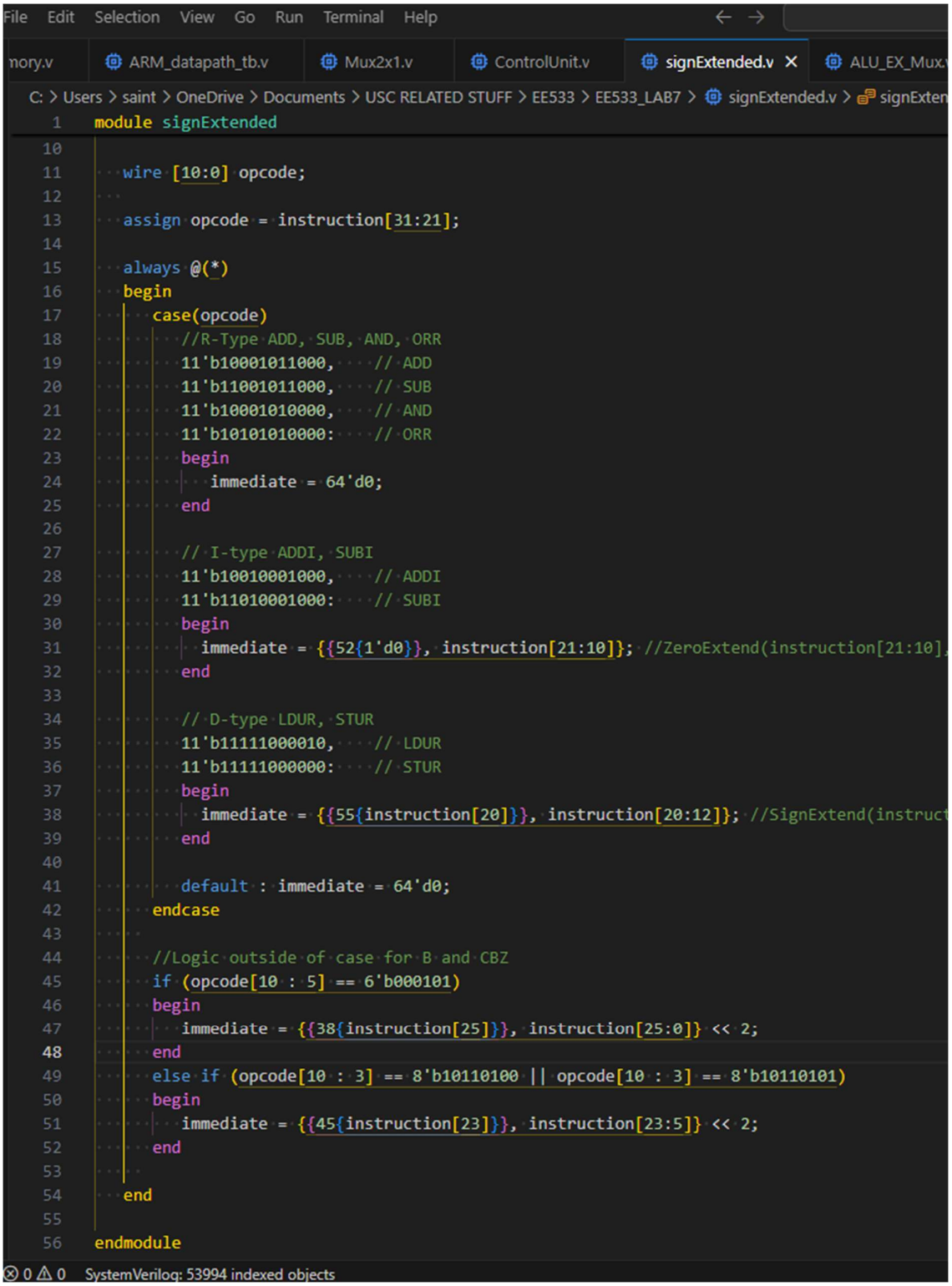
The snippets of the code and the changes we made are below:





We made the CPU instructions to parse the instructions in a byte addressable manner as we wanted to make it LEGV8 compatible.


ry.v
ARM_datapath_tb.v
Mux2x1.v
ControlUnit.v
signExtended.v
ALU_EX_Mux.v
C: > Users > saint > OneDrive > Documents > USC RELATED STUFF > EE533 > EE533_LAB7 > ControlUnit.v > ControlUnit
1 module ControlUnit
59 ALUop = 2'b10; // ALU does ADD (part of ALU control unit logic)
60 end
61 else if(control_instruction_i[10:1] == 10'b1101000100) // SUBI
62 begin
63 reg2Loc = 0;
64 ALUsrc = 1;
65 memtoReg = 0;
66 regWrite = 1;
67 memRead = 0;
68 memWrite = 0;
69 branch = 0;
70 ALUop = 2'b10; // ALU does SUB (part of ALU control unit logic)
71 end
72 else
73 begin
74 case(control_instruction_i)
75
76 11'b10001011000, //ADD
77 11'b11001011000, //SUB
78 11'b10001010000, //AND
79 11'b10101010000: //ORR
80 begin
81 reg2Loc = 0;
82 ALUsrc = 0;
83 memtoReg = 0;
84 regWrite = 1;
85 memRead = 0;
86 memWrite = 0;
87 branch = 0;
88 ALUop = 2'b10;
89 end
90
91 11'b11111000010: //LDUR Load
92 begin
93 reg2Loc = 1;
94 ALUsrc = 1;
95 memtoReg = 1;
96 regWrite = 1;
97 memRead = 1;
98 memWrite = 0;
99 branch = 0;
100 ALUop = 2'b00;
101 end
102
103 11'b11111000000: //STUR Store
104 begin
105 reg2Loc = 1;


File Edit Selection View Go Run Terminal Help
mory.v
ARM_datapath_tb.v
Mux2x1.v
ControlUnit.v
signExtended.v
ALU_EX_Mux.v
C: > Users > saint > OneDrive > Documents > USC RELATED STUFF > EE533 > EE533_LAB7 > signExtended.v > signExten
1 module signExtended
10
11 wire [10:0] opcode;
12
13 assign opcode = instruction[31:21];
14
15 always @(*)
16 begin
17 case(opcode)
18 //R-Type ADD, SUB, AND, ORR
19 11'b10001011000, // ADD
20 11'b11001011000, // SUB
21 11'b10001010000, // AND
22 11'b10101010000: // ORR
23 begin
24 immediate = 64'd0;
25 end
26
27 // I-type ADDI, SUBI
28 11'b10010001000, // ADDI
29 11'b11010001000: // SUBI
30 begin
31 immediate = {{52{1'd0}}, instruction[21:10]}; //ZeroExtend(instruction[21:10],
32 end
33
34 // D-type LDUR, STUR
35 11'b11111000010, // LDUR
36 11'b11111000000: // STUR
37 begin
38 immediate = {{55{instruction[20]}}, instruction[20:12]}; //SignExtend(instruct
39 end
40
41 default : immediate = 64'd0;
42 endcase
43
44 //Logic outside of case for B and CBZ
45 if (opcode[10 : 5] == 6'b000101)
46 begin
47 immediate = {{38{instruction[25]}}, instruction[25:0]} << 2;
48 end
49 else if (opcode[10 : 3] == 8'b10110100 || opcode[10 : 3] == 8'b10110101)
50 begin
51 immediate = {{45{instruction[23]}}, instruction[23:5]} << 2;
52 end
53
54 end
55
56 endmodule
⊗ 0 ⚠ 0 SystemVerilog: 53994 indexed objects

pipe_EXMEM.v | pipe_MEMWB.v | Data_Mem.v | IF_I

C: > Users > saint > EE533_LAB_PROJECTS > LAB7 > MEMtoWB modules > p

```
2   module pipe_EXMEM
7       (
8           input                       clock, reset    ,
9           input                       memRead_i       ,
10          input                       memWrite_i      ,
11          input                       regWrite_i      ,
12          input                       mem2Reg_i       ,
13          input       [4:0]           writeReg_i      ,
14          input       [DATA_WIDTH-1 : 0] writeData_mem  ,
15          input       [DATA_WIDTH-1 : 0] aluResult_in   ,
16          input                       aluZero_i       ,
17
18          output reg                  memRead_o       ,
19          output reg                  memWrite_o      ,
20          output reg                  regWrite_o      ,
21          output reg                  mem2Reg_o       ,
22          output reg [4:0]            writeReg_o      ,
23          output reg [DATA_WIDTH-1 : 0] writeData_mem_o,
24          output reg [DATA_WIDTH-1 : 0] aluResult_out   ,
25          output reg                  aluZero
26      );
27
28      always @(posedge clock or negedge reset)
29      begin
30          if (!reset)
31          begin
32              memRead_o       <= 'd0;
33              memWrite_o      <= 'd0;
34              regWrite_o      <= 'd0;
35              mem2Reg_o       <= 'd0;
36              writeReg_o      <= 'd0;
37              writeData_mem_o <= 'd0;
38              aluResult_out   <= 'd0;
39              aluZero_i       <= 'd0;
40          end
41          else if (enable)
42          begin
43              memRead_o       <= memRead_i    ;
44              memWrite_o      <= memWrite_i   ;
45              regWrite_o      <= regWrite_i   ;
46              mem2Reg_o       <= mem2Reg_i    ;
47              writeReg_o      <= writeReg_i   ;
48              writeData_mem_o <= writeData_mem;
49              aluResult_out   <= aluResult_in ;
50              aluZero_o       <= aluZero_i    ;
51          end
```

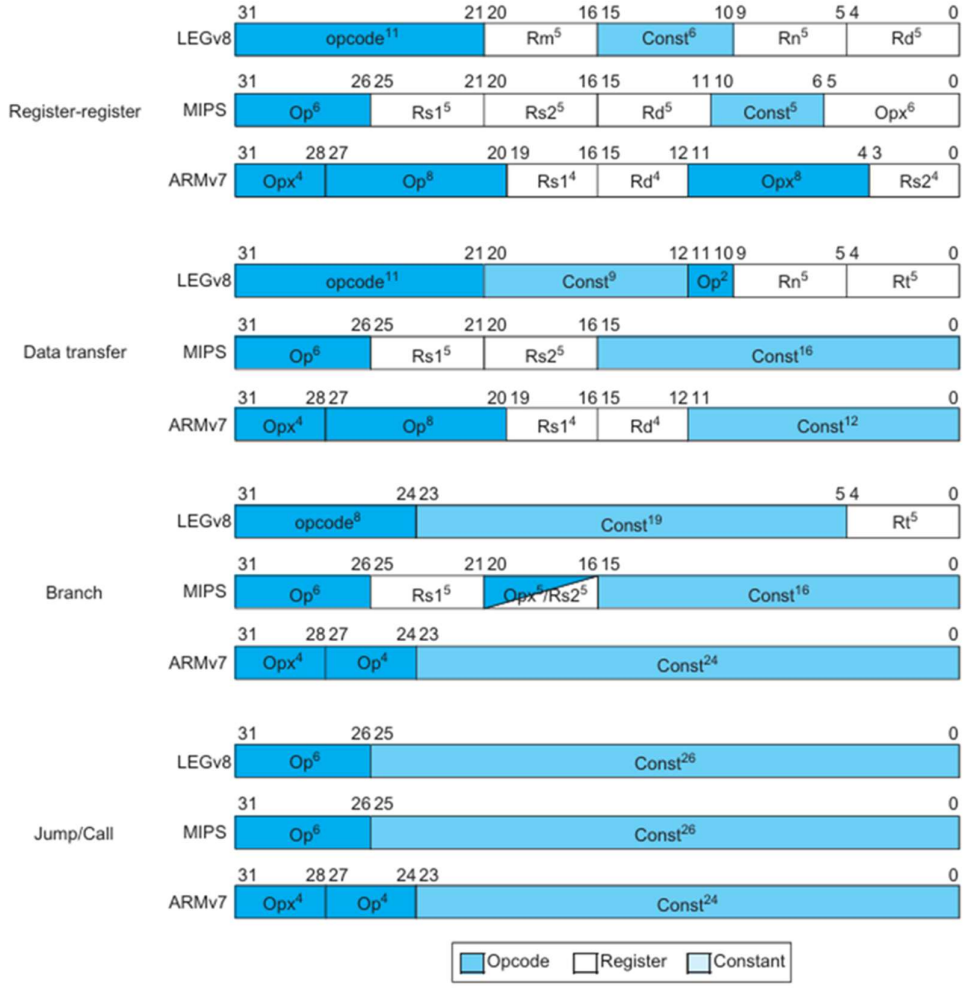
This is the reference we used to generate our opcodes and the alu control:-

Instruction	ALUOp	Instruction operation	Opcode field	Desired ALU action	ALU control input
LDUR	00	load register	XXXXXXXXXXX	add	0010
STUR	00	store register	XXXXXXXXXXX	add	0010
CBZ	01	compare and branch on zero	XXXXXXXXXXX	pass input b	0111
R-type	10	ADD	10001011000	add	0010
R-type	10	SUB	11001011000	subtract	0110
R-type	10	AND	10001010000	AND	0000
R-type	10	ORR	10101010000	OR	0001

ALUOp		Opcode field											
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[24]	I[23]	I[22]	I[21]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	X	0111
1	X	1	0	0	0	1	0	1	1	0	0	0	0010
1	X	1	1	0	0	1	0	1	1	0	0	0	0110
1	X	1	0	0	0	1	0	1	0	0	0	0	0000
1	X	1	0	1	0	1	0	1	0	0	0	0	0001

Name	Fields							Comments
Field size		6 to 11 bits	5 to 10 bits	5 or 4 bits	2 bits	5 bits	5 bits	All LEGv8 instructions are 32 bits long
R-format	R	opcode	Rm	shamt		Rn	Rd	Arithmetic instruction format
I-format	I	opcode	immediate			Rn	Rd	Immediate format
D-format	D	opcode	address		op2	Rn	Rt	Data transfer format
B-format	B	opcode	address					Unconditional Branch format
CB-format	CB	opcode	address				Rt	Conditional Branch format
IW-format	IW	opcode	immediate				Rd	Wide Immediate format

FIGURE 2.21 LEGv8 Instruction formats.


Register-register
LEGv8
31 21 20 16 15 10 9 5 4 0
opcode11 Rm5 Const6 Rn5 Rd5
MIPS
31 26 25 21 20 16 15 11 10 6 5 0
Op6 Rs15 Rs25 Rd5 Const5 Opx6
ARMv7
31 28 27 20 19 16 15 12 11 4 3 0
Opx4 Op8 Rs14 Rd4 Opx8 Rs24
Data transfer
LEGv8
31 21 20 12 11 10 9 5 4 0
opcode11 Const9 Op2 Rn5 Rt5
MIPS
31 26 25 21 20 16 15 0
Op6 Rs15 Rs25 Const16
ARMv7
31 28 27 20 19 16 15 12 11 0
Opx4 Op8 Rs14 Rd4 Const12
Branch
LEGv8
31 24 23 5 4 0
opcode8 Const19 Rt5
MIPS
31 26 25 21 20 16 15 0
Op6 Rs15 Opx5/Rs25 Const16
ARMv7
31 28 27 24 23 0
Opx4 Op4 Const24
Jump/Call
LEGv8
31 26 25 0
Op6 Const26
MIPS
31 26 25 0
Op6 Const26
ARMv7
31 28 27 24 23 0
Opx4 Op4 Const24
Opcode
Register
Constant

```
loop:
   LDUR X1, [X20, #0]
   LDUR X2, [X20, #8]
   SUB X3, X1, X2
   CBZ X3, noswap
   STUR X2, [X20, #0]
   STUR X1, [X20, #8]
noswap:
   ADD X20, X20, #8
   SUB X5, X5, #1
   CBZ X5, end
   B loop
end:
```

This is a direct mapping of the bubble sort logic into the subset of supported LEGv8 instructions.

Internal Memory Initialization and Verification

1. **Memory Initialization:** The unsorted array is pre-loaded into data memory using a **hex file loader written in Verilog**. The array can be observed at program start.
2. **Memory Verification:** After program execution, the sorted array is dumped from data memory for verification. This ensures the processor performed correct comparisons and swaps.

Testing and Simulation Process

Simulation: All components (ALU, Control Unit, Pipeline Registers, etc.) were tested individually and together in a datapath testbench.

Tool: All simulations were conducted in ModelSim to verify instruction and data flow across the pipeline.

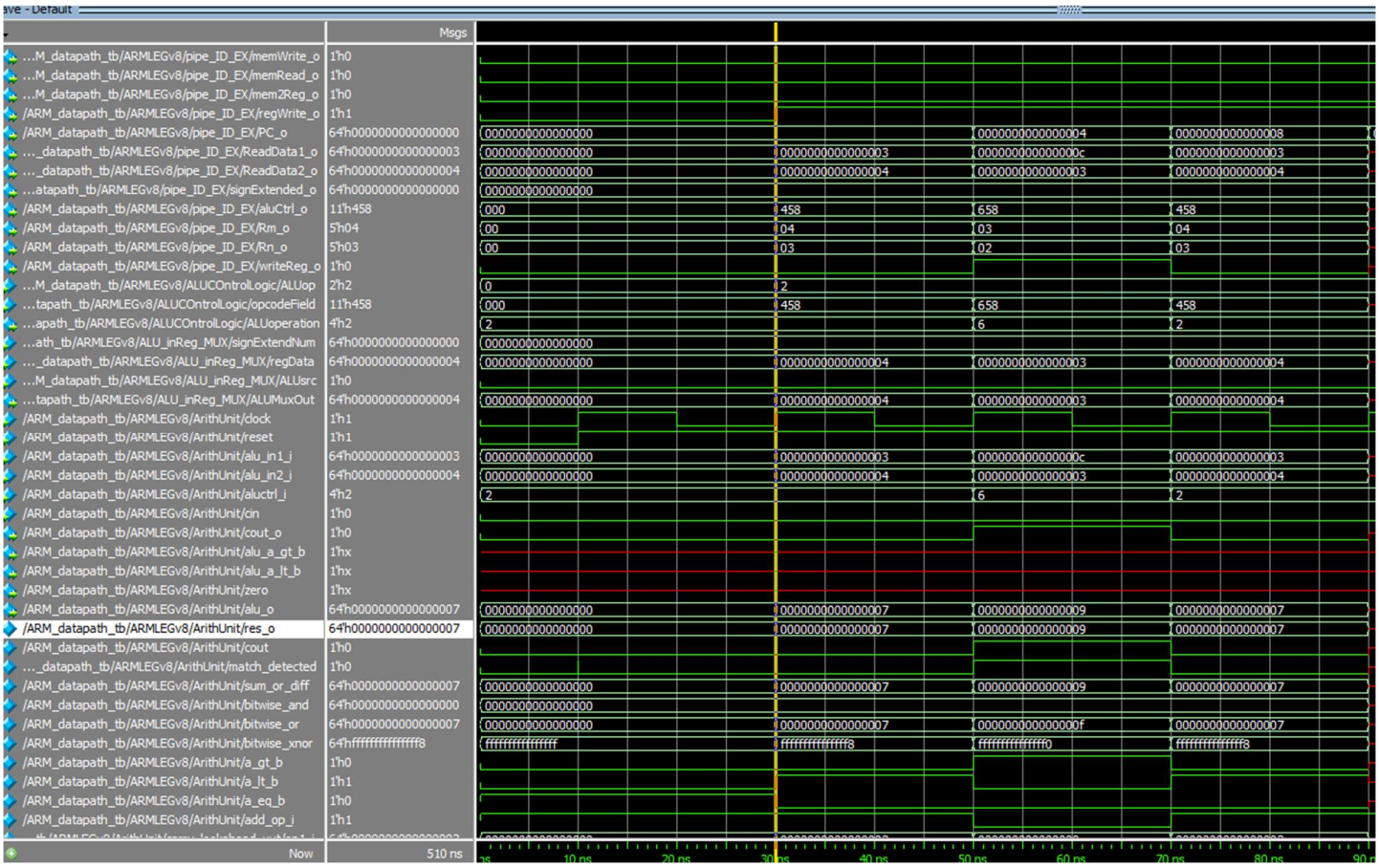
Waveform Analysis: Full pipeline tracing was performed, capturing:

- Instruction fetch and decode.
- Data hazards (stalls).
- ALU operations.
- Correct forwarding paths.
- Correct memory access.

NetFPGA Testing: Once simulation passed, the design was synthesized and deployed to NetFPGA hardware.

Runtime Behavior: The same bubble sort program was run on NetFPGA, with the internal memory contents captured at the end, confirming correct sorting.

Basic simulation run on modelsim:



vsim work.ARM_datapath_tb
run 500ns
 # Time=0ns: Reset asserted
 # Time=10ns: Reset deasserted, instruction fetch starts
 # Time=50ns: First compare instruction executed
 # Time=200ns: First swap executed
 # Time=450ns: Final sorted array = [2, 3, 5, 8, 9]