University of Victoria

Electrical and Computer Engineering Department

ECE 449 - Computer Systems and Architecture

16-Bit Harvard Architecture CPU

Khephren Gould V01012827,
Zarina Guzman V01057778

12th April 2026

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

ALU	Arithmetic Logic Unit
AMAT	Average Memory Access Time
AXI	Advanced eXtensible Interface
BIOS	Basic Input/Output System
BMG	Block Memory Generator
BTFN	Backward Taken, Forward Not-taken
CPI	Cycles Per Instruction
CPU	Central Processing Unit
DDS	Direct Digital Synthesis
DFT	Design For Test
DPDISTRAM	Dual-Port Distributed RAM
DSP	Digital Signal Processing
EX	Execute (pipeline stage)
FPGA	Field-Programmable Gate Array
FWD	Forwarding (Unit)
ID	Instruction Decode (pipeline stage)
IF	Instruction Fetch (pipeline stage)
ILA	Integrated Logic Analyzer
I/O	Input / Output
IP	Intellectual Property
ISA	Instruction Set Architecture
LED	Light-Emitting Diode
LR	Link Register
LUT	Look-Up Table
MEM	Memory Access (pipeline stage)
NCO	Numerically Controlled Oscillator
NOP	No Operation
PC	Program Counter
PMOD	Peripheral Module (interface)
PSR	Program Status Register
RAM	Random Access Memory
RAW	Read After Write
RM	Rate Monotonic
ROM	Read-Only Memory
RTL	Register-Transfer Level
SDR	Software-Defined Radio
USB	Universal Serial Bus
VGA	Video Graphics Array
VHDL	VHSIC Hardware Description Language
WB	Write Back (pipeline stage)
XDC	Xilinx Design Constraints
XPM	Xilinx Parameterized Macro

1 Abstract

This report presents the design, implementation, and verification of a 16-bit pipelined processor for the ECE 449 course project at the University of Victoria. The processor implements a Harvard architecture with separate instruction and data paths, a five-stage pipeline (Fetch, Decode, Execute, Memory, Write Back), and the full course-specified instruction set across the A, B, and L formats, including an extension instruction (BRR.OVERFLOW) that conditionally branches on signed arithmetic overflow. The datapath was developed in VHDL and synthesized for a Xilinx Artix-7 (xc7a35t) FPGA using Vivado, with a dual-ported distributed RAM serving as unified instruction and data memory, a separate ROM holding the bootloader, and memory-mapped I/O for board switches and LEDs. Data hazards are resolved through a combined forwarding unit and load-use hazard detection unit, while control hazards are handled by flushing the IF/ID and ID/EX stages on taken branches resolved in the Execute stage. Functional verification was performed both in simulation, using per-format assembly testbenches in Vivado, and on hardware, where a factorial program correctly computed 3! = 6 from switch input and successfully detected and branched on multiplication overflow. Post-synthesis utilization on the target device was 1653 LUTs (7.95%), 532 registers (1.28%), and a single DSP48E1 block, with timing analysis showing positive hold slack and a worst-case setup violation of approximately 5 ns at 100 MHz, limited primarily by the combinational multiplier feeding the falling-edge V flag register. The processor was successfully clocked and verified at frequencies up to 100 kHz. These results demonstrate a fully functional pipelined CPU meeting all baseline ISA requirements, while the timing analysis identifies the multiplier path and branch resolution stage as the most productive targets for future performance improvements such as sequential multiplication and static branch prediction.

2 Introduction

This report presents the design and implementation of a 16-bit pipelined processor for ECE 449 at the University of Victoria. The objective was to design a processor capable of executing any program written in the provided instruction set architecture, synthesize it onto an FPGA, and verify its correctness through simulation.

The design specifications required a Harvard architecture system using a dual-ported RAM (Xilinx XPM_MEMORY_DPDISTRAM) to separate instruction and data traffic. A 1024-byte ROM starting at address `0x0000` stores a provided bootloader responsible for loading user code into RAM and initiating execution. The 1024-byte RAM begins at address `0x0800`, and I/O is handled through memory-mapped ports, with the input port at `0xFFF0` and the output port at `0xFFF2`. Two reset modes are supported: Reset and Load, which vectors to address `0x0002` to load user code into RAM, and Reset and Execute, which vectors to address `0x0000` to begin execution. The processor uses a fixed-width 16-bit instruction encoding divided into three formats (A, B, and L), with eight general-purpose registers and two status flags.

This report first describes the instruction set architecture in detail, covering each format's bit-field layout and the complete set of supported instructions. It then presents the processor architecture, including the datapath components: ALU, register file, program counter, and memory interfaces, followed by the controller and its state machine implementation. The five-stage pipeline design is discussed next, walking through each stage and its associated pipeline register, along with the strategies used to handle data and control hazards. Finally, the report presents synthesis and simulation results, discusses design limitations and challenges encountered, and concludes with a summary of key takeaways.

3 Objective

The objective of this project is to design, implement, and verify a 16-bit pipelined processor on an FPGA. The processor must support the full instruction set provided, including arithmetic and logic operations, branching and subroutine control flow, and memory access. The design must follow a Harvard architecture with separate instruction and data memory paths, implement a five-stage pipeline to improve throughput, and interface with external I/O through memory-mapped ports. The final design is synthesized onto the FPGA and validated through simulation.

4 System Overview

A high-level schematic of the CPU architecture is shown in Figure **??**. The system consists of a 5 stage pipeline. Instructions is obtained from memory in the fetch stage, a controller unit decodes them in the decode stage, ALU operations and Branches are computed in the execute stage, and load/store operations are completed in the memory stage. The write back stage is used to place the evaluated result back into the register file.

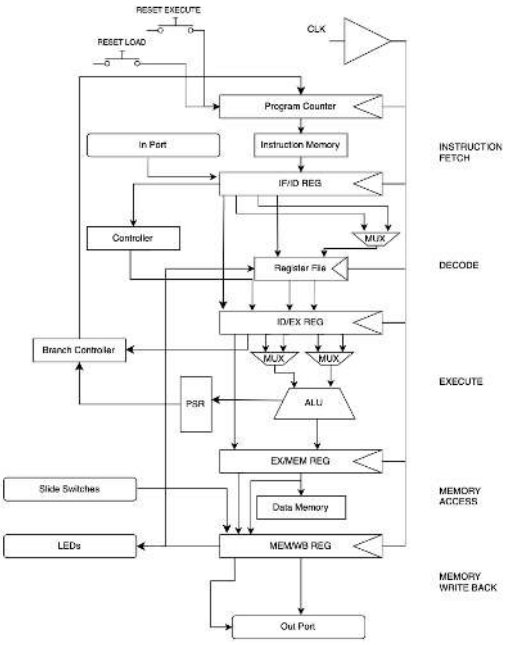

Figure 1: High Level System Overview

The modules contained within the CPU top level are listed below:

1. Program Counter
   - Features enable, reset, and parallel load capability
2. Single Port ROM
   - Xylinx Block Memory Generated 512 Word ROM
3. Dual Port RAM
   - Xylinx XPM Macro 512 Word Dual RAM
4. IF/ID Pipeline Register
   - 48 bit rising edge sensitive register with parallel load and enable
5. ID/EX Pipeline Register
   - 115 bit rising edge senstive with parallel load and enable
6. EX/MEM Pipeline Register
   - 42 bit rising edge senstive with parallel load and enable
7. MEM/WB Pipeline Register
   - 39 bit rising edge senstive with parallel load and enable
8. 8 Register Register File
   - dual read write ports, combinational read, falling edge write
9. ALU
   - Add, Subtract, Multiply, Barrel Shift, Nand, Magnitude Comparison and Overflow Detection Capabilities

10. PSR
    - Program Status Register to store n,z, and v flags, enable is separated between v and the z/n flags

11. Controller/Decoder
    - Purely cominational decoder to translate instructions into control signals

12. Branch Controller
    - Receives controller signals and PSR flags, resolves branch taken/not taken and computes branch target

13. Forwarding Unit
    -Receives destination and source register field from the ID, EX, and MEM stages, outputs control signals for ALU source mux

14. Hazard Detection Unit
    -Receives destination and source register field from the ID, EX, and MEM stages, outputs control signals for stalling on load data hazard.

5 Instruction Set Architecture

This processor uses 1 word long, word aligned, instruction set with three distinct instruction formats: A-format, B-format, and L-format. All instructions are encoded in a single 16-bit word, with bits 15:9 reserved for the opcode. The Following sections provide a detailed look into each format.

5.1 Format A

Table 1: A-Format Instruction Formats

Format	**Bits 15:9**	**Bits 8:6**	**Bits 5:3**	**Bits 2:0**
A0	op-code	xxxx		
A1	op-code	ra	rb	rc
A2	op-code	ra	xxxxx	cl
A3	op-code	ra	xxxx	

A-format instructions handle arithmetic, logic, shift, test, and I/O operations. All A-format instructions share the same 7-bit opcode field in bits 15:9, with the remaining 9 bits divided differently depending on the subtype.

Format A0 uses no operand fields and is reserved for instructions that require no register specification, such as NOP. All 9 lower bits are unused and ignored by the processor.

Format A1 encodes three register operands in bits 8:6 (ra), 5:3 (rb), and 2:0 (rc), and is used by the arithmetic and logic instructions ADD, SUB, MUL, and NAND. For ADD, SUB, and MUL, the operation is performed on the contents of rb and rc, with the result written to ra. NAND is a special case, it operates on ra and rb, writing the result back to ra. The rc field is still present in the encoding but is unused by NAND.

Format A2 is used by the shift instructions SHL and SHR. It encodes a destination register ra in bits 8:6 and a 4-bit shift count cl in bits 3:0. Bits 5:4 are unused. If the shift count is nonzero, the contents of ra are shifted left or right by that amount, with zeros filling the vacated bit positions. If the shift count is zero, no operation is performed.

Format A3 encodes only a single register ra in bits 8:6, with the remaining bits 5:0 unused. TEST reads the value in ra and sets the Z flag to 1 if the value is zero, or clears it to 0 otherwise. It also sets the N flag to 1 if the value is negative (i.e. the sign bit is set), or clears it to 0 otherwise. TEST is the only instruction that modifies the status flags. OUT writes the contents of ra to the output port, and IN reads from the input port into ra.

Table 2 lists all A-format instructions with their opcodes, functions, and syntax.

Table 2: A-Format Instructions

Mnemonic	**Op-code**	**Function**	**Type**	**Syntax**
NOP	0	Nothing	A0	*NOP*
ADD	1	R[ra] ← R[rb] + R[rc];	A1	*ADD ra,rb,rc*
SUB	2	R[ra] ← R[rb] − R[rc];	A1	*SUB ra,rb,rc*
MUL	3	R[ra] ← R[rb] × R[rc];	A1	*MUL ra,rb,rc*
NAND	4	R[ra] ← R[ra] NAND R[rb];	A1	*NAND ra,rb,rc*
SHL	5	n:=c1⟨3...0⟩): (n|0) → (R[ra]⟨15...0⟩ ← R[ra]⟨15-n...0⟩#(n@0)): (n=0) → ();	A2	*SHL ra#n*
SHR	6	n:=c1⟨3...0⟩): (n|0) → (R[ra]⟨15...0⟩ ← (n@0)#R[ra]⟨15...n⟩): (n=0) → ();	A2	*SHR ra#n*
TEST	7	(R(ra) = 0) → Z ← 1; else → Z ← 0: (R[ra] < 0) → N ← 1; else → N ← 0;	A3	*TEST ra*
OUT	32	OUT.PORT ← R[ra];	A3	*OUT ra*
IN	33	R[ra] ← IN.PORT;	A3	*IN ra*

5.2 Format B

Table 3: B-Format Instruction Formats

Format	**Bits 15:9**	**Bits 8:6**	**Bits 5:0**
B1	op-code (BRR)	disp.l (bits 8:0)	
B2	op-code (BR)	ra	disp.s (bits 2:0)

Format B instructions handle relative and non-relative branching, conditional or otherwise, as well as sub routines.

B1 is used for PC relative branches (BRR), where all 9-bits are available for displacement since no no register operand is needed.

B2, on the other hand, computes the branch target relative to a register value in ra, which leaves only 6 bits for the displacement to make room for the register

field.

Table 4 lists all A-format instructions with their opcodes, functions, and syntax. The status flags are used for instructions BRR.N and BRR.Z as well as BR.Z and BR.N for the conditional branches.

Table 4: B-Format Instructions

Mnemonic	Op-code	Function	Type	Syntax
BRR	64	PC ← PC†+2*disp.l {sign extended 2's complement}	B1	*BRR +disp.l*
BRR.N	65	(N=1) → PC ← PC+2*disp.l {sign extended 2's complement}; (N=0) → PC ← PC+2 { 2's complement};	B1	*BRR.N +disp.l*
BRR.Z	66	(Z=1) → PC ← PC+2*disp.l {sign extended 2's complement}; (Z=0) → PC ← PC+2 { 2's complement};	B1	*BRR.Z +disp.l*
BR	67	PC ← R[ra]+2*disp.s {sign extended 2's complement}	B2	*BR ra+disp.s*
BR.N	68	(N=1) → PC ← R[ra] {word aligned}+2*disp.s {sign extended 2's complement}; (N=0) → PC ← PC+2 { 2's complement};	B2	*BR.N ra+disp.s*
BR.Z	69	(Z=1) → PC ← R[ra] {word aligned}+2*disp.s {sign extended 2's complement}; (Z=0) → PC ← PC+2 { 2's complement};	B2	*BR.Z ra+disp.s*
BR.SUB	70	r7 ← PC + 2; PC ← R[ra] {word aligned}+2*disp.s {sign extended 2's complement};	B2	*BR.SUB ra+disp.s*
RETURN	71	PC ← r7;	A0	*RETURN*
BRR.OVERFLOW	72	(V=1) → PC ← PC+2*disp.l {sign extended 2's complement}; (V=0) → PC ← PC+2 { 2's complement};	B1	*BRR.OVERFLOW + disp.l*

5.3 Format L

Table 5: L-Format Instruction Formats

Format	Bits 15:9	Bits 8:6	Bits 5:3	Bits 2:0
L1	op-code	m.l (bit 8) + imm (bits 7:0)		
L2	op-code	r.dest	r.src	xxx

L-format instructions handle memory access, register-to-register moves, and immediate loading. Like the other formats, the opcode occupies bits 15:9, with the remaining bits divided differently between the two subtypes.

Format L1 is used exclusively by the LOADIMM instruction, which loads an 8-bit immediate value into register r7. Bit 8 serves as the m.l flag, which selects whether the immediate is written to the upper byte (m.l=1) or the lower byte (m.l=0) of r7.

Format L2 encodes two register operands: r.dest in bits 8:6 and r.src in bits 5:3, with bits 2:0 unused. LOAD reads from memory at the address stored in r.src and writes the result into r.dest, while STORE writes the contents of r.src to the memory address stored in r.dest. MOV simply copies the value in r.src directly into r.dest without any memory access.

Table 6: L-Format Instructions

Mnemonic	Op-code	Function	Type	Syntax
LOAD	16	R[r.dest] ← M[R[r.src]];	L2	*LOAD r.dest, r.src*
STORE	17	M[R[r.dest]] ← R[r.src];	L2	*STORE r.dest, r.src*
LOADIMM	18	(m.l=1) → R7⟨15···8⟩ ← imm: (m.l=0) → R7⟨7···0⟩ ← imm;	L1	*LOADIMM.upper #nLOADIMM.lower #n*
MOV	19	R[r.dest] ← R[r.src];	L2	*MOV dest,src*

6 Processor Architecture

6.1 Data Path

The datapath contains all the components responsible for moving and manipulating data as instructions execute. It includes the program counter, instruction and data memory, the register file, and the arithmetic logic unit, along with the wiring and multiplexers that connect them. While the controller decides what each component should do at any given cycle, the datapath is where the actual work of computation happens. Values are read from registers, passed through the ALU, written to memory, and eventually stored back into the register file.

Each component in the datapath was designed and simulated individually before being integrated into the full processor. This made it easier to verify correctness at each level and isolate issues during testing. The following subsections describe each of the main datapath components in turn.

6.1.1 Program Counter

The program counter (PC) is a 16-bit register that holds the address of the next instruction to be fetched from memory. Its width matches the processor's address space, allowing it to reach any location in ROM, RAM, or the memory-mapped I/O region.

Listing 1: Program Counter

```
if rising_edge(clk) then
if(en = '0') then
    if rst_load = '1' then
        pc := std_logic_vector(to_unsigned(2,11));
    elsif rst_execute = '1' then
        pc := (others => '0');
    elsif load = '1' then
        pc := address_in;
    else
        pc := std_logic_vector(unsigned(pc) + 2);
    end if;
else
        pc := std_logic_vector(unsigned(pc) - 2);
end if;
```

Under normal operation, the PC increments by two each clock cycle, since instructions are 16 bits wide and memory is byte-addressed. This default behavior is overridden in three cases. The first is a synchronous reset: on Reset and Execute the PC is cleared to `0x0000`, and on Reset and Load the PC is set to `0x0002`. Both reset vectors point to locations in the ROM, where BRR instructions redirect execution to the appropriate handling routine. The second case is a branch or jump. When the `load` signal is asserted, the PC is loaded with

the target address provided on `address_in`, which is computed by the branch logic elsewhere in the datapath.

The third case handles pipeline stalls. Because the instruction memory has a one-cycle read latency, a naive stall (simply holding the PC value) causes the IF/ID register to miss an instruction when execution resumes. To compensate, when the enable signal is asserted (indicating a stall), the PC decrements by two rather than holding its value. This ensures that on the following cycle, the fetch stage receives the instruction that would have been fetched if the stall had not occurred. While unconventional, this approach simplifies stall handling in the presence of the memory's read delay.

6.1.2 Memory

The memory subsystem consists of three distinct regions accessed through a shared address bus: a read-only ROM holding the boot loader, a read/write RAM holding user code and data, and two memory-mapped I/O ports for communication with the outside world. Address decode logic inspects the high bits of each access and routes it to the appropriate destination, as summarized in the memory map below.

Table 7: Memory Map

Address Range	**Destination**
`0x0000`–`0x07FF`	ROM (boot loader)
`0x0800`–`0x0FFF`	RAM (user code and data)
`0xFFF0`	Input port
`0xFFF2`	Output port

6.1.3 RAM

The RAM is implemented using a single dual-ported block built around the Xilinx `xpm_memory_dpdistram` macro. Using two independent ports on the same physical memory provides the benefits of a Harvard architecture, namely simultaneous instruction fetch and data access, without duplicating storage. Port A handles data memory traffic for LOAD and STORE instructions and is configured for both reading and writing, while port B is read-only and dedicated to the Instruction Fetch stage.

The RAM itself is 8192 bits in size, organized as 512 words of 16 bits each, giving a total of 1024 bytes. Internal addresses are 11 bits to reflect the byte-addressable memory map, but since all accesses are word-aligned the lowest bit is discarded and only the upper 10 bits are passed to the XPM macro. Port A is configured with a read latency of zero, so data reads complete combinationally within the same cycle, while port B uses a read latency of one, placing fetched instructions at the output on the cycle following the address being presented. The signle cycle latency is required for compatibility with the ROM allowing the logic for reading instructions from RAM and ROM to be symmetrical.

To handle pipeline flushes caused by taken branches, the memory module includes a small counter that holds the port B output (instructions) at zero after two cycles after a flush signal is received. This is to accommodate for an issue arising from the single cycle latency present in the instruction memory. The

latency resulted in the 3rd instruction after the branch to fetch and execute. The counter ensures it is properly flushed. A similar solution was employed in the ROM.

6.1.4 ROM

The ROM is a separate 1024-byte block initialized with the provided boot loader image. It holds the reset vectors at addresses `0x0000` and `0x0002`, along with the load and execute handlers that run after each type of reset. Because the boot loader is never modified during execution, the ROM does not require a write port. The input and output ports are handled by a de-mux and a mux that intercepts accesses to `0xFFF0` and `0xFFF2` and redirects them to the corresponding top-level ports rather than to RAM. The mux and de-mux are shown in Figures and respectively.

The boot loader lives in ROM and is responsible for loading the program into RAM during a reset load. After the load is complete, a reset execute should then cause the program counter to jump to the start of the RAM memory, 0x0800. In our implementation, since we could not get the bootloader to properly store the the program into RAM, we instead encoded the program into RAM ourselves. In this way, we still had the ability to jump to the program during a reset execute, and carry out the factorial program.

6.1.5 Register File

The register file holds the eight general-purpose 16-bit registers (r0 through r7) available to the programmer. Registers are addressed using a 3-bit index, and the file is implemented as an array of eight 16-bit entries with two read ports and one write port. This configuration allows two source operands to be read simultaneously while another instruction writes back its result.

Reads from the register file are asynchronous: the contents of the register indexed by `rd_index1` or `rd_index2` appear on the corresponding output ports as soon as the index is applied. Writes, on the other hand, are synchronous and occur on the falling edge of the clock when `wr_enable` is asserted. Writing on the falling edge (rather than the rising edge) allows a register to be written in the first half of a clock cycle and read in the second half of the same cycle. This solution was employed as a means to eliminate a data hazard between instructions in the write back and the execute stage as described in Patterson, Hennesey (pp.408) [1]. The Figure that accompanies the discussion of this method is shown in Figure 2

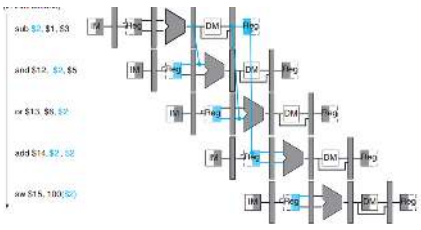

Figure 2: Data Hazard Between SUB and ADD

Register r7 serves a dual role as the link register for subroutine calls. When BR.SUB is executed, the return address (PC + 2) must be saved to r7 while the

branch target is loaded into the PC. To support this, the register file includes a `reg_rd_link` input that, when asserted, forces the second read port to return the contents of r7 regardless of the value on `rd_index2`. This provides the RETURN instruction with direct access to the link register without requiring a special instruction encoding.

A synchronous reset clears all eight registers to zero, ensuring a known state at startup. A block diagram of the register file is shown in Figure.

6.1.6 Arithmetic Logic Unit

Listing 2: ALU Logic

```
   process(all)
begin
    temp_result <= (others => '0');
    flag_z <= '0';
    flag_n <= '0';
    flag_v <= '0';
    enable_psr <= '0';
    enable_v <= '0';
    if(alu_rst = '0') then
        case alu_mode is
            when "0000" =>
                --NOP
                temp_result <= std_logic_vector(signed(op1));
            when "0001" =>
             -- ADD
                enable_v <= '1';
                add_sub_ext <= resize(signed(op1), 17) + resize(signed(op2), 17);
                temp_result <= std_logic_vector(add_sub_ext(15 downto 0));
                flag_v <= add_sub_ext(16) xor add_sub_ext(15);

            when "0010" =>
            -- SUB
                enable_v <= '1';
                add_sub_ext <= resize(signed(op1), 17) - resize(signed(op2), 17);
                temp_result <= std_logic_vector(add_sub_ext(15 downto 0));
                flag_v <= add_sub_ext(16) xor add_sub_ext(15);

            when "0011" =>
            -- MUL
                enable_v <= '1';
                mul_full <= signed(op1) * signed(op2);
                temp_result <= std_logic_vector(mul_full(15 downto 0));
                for i in 16 to 31 loop
                    if mul_full(i) /= mul_full(15) then
                        flag_v <= '1';
                    end if;
                end loop;
            when "0100" =>
                --NAND
                temp_result <= op1 nand op2;
            when "0101" =>
                --SLL
                temp_result <=
                     std_logic_vector(shift_left(unsigned(op2),to_integer(unsigned(op1))));
            when "0110" =>
                --SRL
                temp_result <=
                     std_logic_vector(shift_right(unsigned(op2),to_integer(unsigned(op1))));
```

```
            when "0111" =>
                --TEST
                enable_psr <= '1';
                if to_integer(signed(op2)) = 0 then
                    flag_z <= '1';
                end if;
                if to_integer(signed(op2)) < 0 then
                    flag_n <= '1';
                end if;
            when "1000" =>
                --out
                temp_result <= std_logic_vector(signed(op2));
            when "1001" =>
                -- Loadimm upper
                temp_result <= op1(7 downto 0) & op2(7 downto 0);
            when "1010" =>
                -- Loadimm lower
                temp_result <= op2(15 downto 8) & op1(7 downto 0);
            when others =>
                temp_result <= (others => '0');
        end case;
    end if;
    alu_result <= temp_result;

end process;
```

The arithmetic logic unit (ALU) performs all arithmetic, logic, and shift operations required by the processor's instruction set. It accepts two 16-bit operands and a 4-bit `alu_mode` control signal from the controller, and produces a 16-bit result along with status flags for zero (Z), negative (N), and overflow (V).

The supported operations include addition, subtraction, and signed multiplication for the arithmetic group, bitwise NAND for the logic group, and logical shift left and shift right for the shift group. Addition and subtraction are implemented by sign-extending both operands to 17 bits before performing the operation, which makes overflow detection straightforward: the overflow flag is set when the carry into the sign bit differs from the carry out of it. Multiplication uses a full 32-bit signed product, and overflow is flagged whenever any of the upper 16 bits disagree with the sign bit of the lower 16 bits, indicating that the result cannot be represented in 16 bits. The overflow flag extends the base instruction set specification, which originally defined only the Z and N flags.

The ALU also handles the TEST, OUT, and LOADIMM instructions. TEST inspects the second operand and sets the Z flag if the value is zero or the N flag if the value is negative, asserting `enable_psr` to signal the program status register to latch these flags. TEST is the only instruction that updates Z and N, which preserves the flag state across unrelated instructions between a TEST and its dependent conditional branch. OUT passes its operand through unchanged for delivery to the output port. LOADIMM is split into upper and lower variants that assemble an 8-bit immediate into the appropriate half of the target register by concatenating bytes from the two operand inputs, leaving the other half untouched.

Shift operations treat `op1` as the shift count and `op2` as the value to be shifted, reflecting the A2 format encoding where the shift count is extracted from the lower bits of the instruction word rather than from a register.

6.2 Program Status Register

The program status register latches the status flags for use in the branch controller to compute conditional branches. These flags are Z, for zero, N for negative, and V for overflow. The PSR is divided into a two bit register and an individual flip flop. The division was necessary to allow for separate enable signals to be used for the N/Z flags and the V flag. The enable signals are asserted by the ALU in a mutually exclusive manner. That is, the N and Z flags are only enabled for writing during a TEST instruction and the v flag is only enabled for writing during MUL, SUB, and ADD instructions. This allows the flags to be individually preserved allowing the programmer more flexibility. Using this scheme allows for code as shown in listing to be properly executed.

Listing 3: PSR Flag Scheme Example

```
TEST r1
MUL r1,r2,r3
br.n r1
```

This code would execute properly despite the BR and TEST instructions being interleaved by the MUL. If all flags were controlled by a single clock enable, the MUL would reset the N flag when writing the V flag.

Finally, the PSR is clocked on the falling edge. This allows the output to be made available in the EX stage which is required due to the use of the flags by the Branch Controller directly (they are not fed through the pipeline).

6.3 Controller

Listing 4: Controller Case Snippet

```
when OP_BR_N =>
br_cond <= "01";
br_en <= '1';
ra_op <= '1';
```

The controller is responsible for generating all control signals required by the datapath based on the current instruction's opcode. Unlike a traditional multi-cycle controller implemented as a state machine, this design uses a purely combinational decoder, which is well suited to a pipelined architecture where each instruction progresses through a fixed sequence of stages on every clock cycle. The controller examines the 7-bit opcode from the IF/ID register and asserts the appropriate control signals for that instruction, which are then latched into the subsequent pipeline registers and carried forward to the stages that consume them.

The generated control signals fall into several groups. Register file signals (`reg_wr_en`, `reg_wr_en_pc`, `reg_rd_link`, and `reg_rst`) determine when and what to write back, as well as when to force the second read port to return r7 for RETURN instructions. ALU signals (`alu_mode`, `alu_src`, and `alu_rst`) select the operation to perform and the source of its operands, with `ra_op` selecting whether the second operand comes from the ra field for instructions like TEST,

OUT, and the shifts. Branching signals (`brr_en`, `br_en`, and `br_cond`) indicate whether the current instruction is a relative or register-based branch.

A full list of controller output signals per instruction can be found in A.

6.4 Branch Controller

The branch controller is responsible for loading the program counter with the appropriate value based on the branch instruction given. It is responsible for both computing the branch target and resolving conditional branches. For any conditional branches resolved as taken it asserts the flush signal which clears the next 3 instructions in the pipeline to be executed. This results in a branch penalty of 3 cycles. A snippet demonstrating the conditional branch resolution and target address computation is shown in listing 5.

Listing 5: Branch Controller Snippet

```
when "01" =>
    if flag_n = '1' and br_en = '1' then
        pc_load <= '1';
        flush <= '1';
        pc_branch_address <= std_logic_vector(
            resize(signed(absolute_addr), pc_branch_address'length) +
            shift_left(resize(signed(disp_s), pc_branch_address'length), 1));
```

The listing shows the portion of the branch controller that executes when the br_cond control signal is set to "01" which occurs when a br.n instruction is decoded. The branch controller first checks the n flag which it reads from the PSR. If it is asserted the branch is taken. The branch controller asserts the pc_load signal to enable the pc for parallel loading and computes the branch target.

6.4.1 BRR.OVERFLOW

This section discusses the implementation of the BRR.OVERFLOW instruction including the required hardware/software co-design.

Software The listing for the test program used for testing overflow is shown in the snipped below:

Listing 6: Branch On Overflow Test Code

```
LOOP:
    mul   r4, r4, r5
    brr.overflow  OVERFLOW    ; branch if multiply overflowed
    add   r5, r5, r3
    sub   r6, r6, r3
    test    r6
    brr.z   DONE
    brr   LOOP
OVERFLOW:
    loadimm.upper 0x00 ; output 0x0000 on overflow
    loadimm.lower 0x00
    mov   r4, r7
```

The software simply inserts the special branch instruction after each multiply. It sets the branch target to a simple block that outputs an error code to the LEDs.

Hardware Hardware modifications and additions were required for the BRR.OVERFLOW instruction. As shown in Appendix A control signals table, the brr.overflow function has a dedicated value for brr.cond. This signal allows the controller to indicate to the branch controller that the conditional branch condition to be verfied for this instruction is that flag v is set. The branch controller was also modified. The added logic is shown in Figure **??**. The conditional branching logic is identical to brr.z and brr.n except the v flag is checked. Finally, a dedicated flip flop to store the v flag was added. Two notable design decisions were to make the flip flop falling edge sensitive and clock enabled by the alu. The falling edge sensitivity is required for the v flag to be available in the ex stage (which is where conditional branching is handled) and the clock enable is only set by the ALU when the flag may require writing (ADD, SUB, MULT). In this way, the flag is preserved until the next instruction for which overflow is possible executes. Without the clock enable the flip flop would be overwritten by the flag v output of the ALU on every clock cycle.

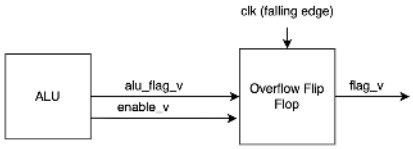

Figure 3: BRR.OVERFLOW Block Diagram

*OUT also asserts `out_port_en`. LOADIMM selects `alu_mode` = 1001 when `loadimm_m1` is high (upper byte) and 1010 when low (lower byte), and also asserts the `loadimm` signal.

6.5 Pipeline

To improve instruction throughput, the processor is implemented as a five-stage pipeline rather than a single-cycle or multi-cycle design. In a non-pipelined processor, each instruction must complete entirely before the next one can begin, meaning the hardware spends most of its time idle. The ALU sits unused during memory access, the register file is idle during execution, and so on. Pipelining addresses this by breaking instruction execution into sequential stages and allowing multiple instructions to be in flight simultaneously, with each stage working on a different instruction during the same clock cycle.

The five stages used in this design are Instruction Fetch (IF), Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). Between each pair of stages sits a pipeline register (IF/ID, ID/EX, EX/MEM, and MEM/WB) that latches the intermediate results and control signals, allowing each stage to operate independently on its own instruction. Splitting the datapath this way also shortens the critical path, since the longest combinational delay in a single clock cycle is now limited to one stage rather than the entire instruction

execution. This translates directly into a higher maximum clock frequency and, in the ideal case, one completed instruction per cycle once the pipeline is full.

Pipelining introduces its own challenges, however. Data hazards arise when an instruction depends on the result of a previous instruction that has not yet written its result back to the register file, and control hazards arise when a branch instruction changes the program counter after subsequent instructions have already been fetched. These issues, and the strategies used to resolve them, are discussed in Section 6.6.

The following subsections describe each pipeline stage in detail, including the operations performed and the role of the associated pipeline register. The full block diagram can be found in appendix C

6.5.1 Instruction Fetch (IF)

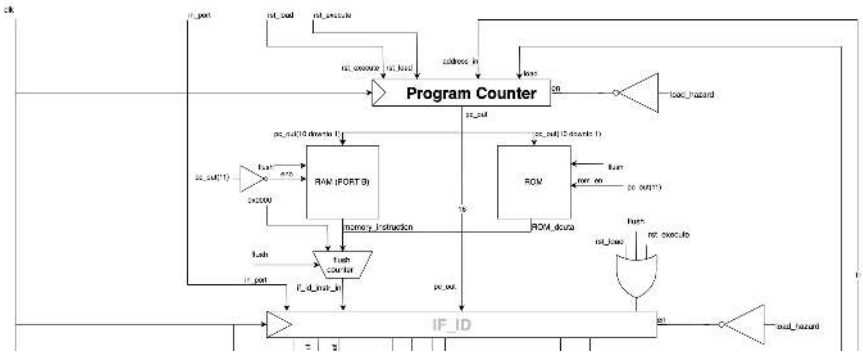

Figure 4: Fetch Stage Schematic

The Instruction Fetch stage is responsible for retrieving the next instruction from memory and presenting it to the Decode stage on the following clock cycle. At the start of each cycle, the program counter drives the read address into port B of the memory subsystem. Because port B is configured with a one-cycle read latency, the addressed instruction word does not appear at the output until the following clock edge, at which point it is captured by the IF/ID pipeline register together with the value of the PC that produced it. Carrying the PC forward is necessary because branch instructions in the Execute stage need to compute their targets relative to the address of the branch itself, not the address of whatever instruction is currently being fetched.

As shown in Table 8, the bit fields for each signal being passed through to the next stage of the pipeline. Note that some of these values overlap the same bit-ranging depending on the instruction format. Those bits encode register indices (`rB`, `rC`), a shift count (`cl`), a long or short branch displacement (`displ`, `disps`), or a LOADIMM immediate and byte-select flag. Rather than storing these fields separately, the register holds the raw bits and exposes them through overlapping VHDL aliases. The actual field selection happens downstream in the ID stage via a set of multiplexers driven by control signals from the decoder. The `ra_op` signal selects between `rC` and `rA` as the second register file read index, while `alu_src` routes either a register file output, a sign-extended displacement, or a zero-extended LOADIMM immediate to the ALU's second operand. A separate

mux feeds bits [3 : 0] into the shifter as `cl` for shift and rotate instructions. In this way the IF/ID register itself remains a dumb 43-bit latch, and the opcode decoded in the same stage is what determines which interpretation of the lower bits is actually consumed by the execute stage.

Table 8: IF/ID Pipeline Register Fields (43 bits total)

Field	Bit Range	Width	Description
`in_port`	[42:27]	16	Input port data passed through to ID
`pc`	[26:16]	11	Program counter value of the fetched instruction
`opcode`	[15:9]	7	Instruction opcode
`ra`	[8:6]	3	Register A index (R-type / I-type destination)
`rb`	[5:3]	3	Register B index (first source)
`rc`	[2:0]	3	Register C index (second source)
Overlapping aliases (instruction-format dependent, share bits [8:0])			
`loadimm_m1`	[8]	1	LOADIMM byte-select flag (high/low byte)
`loadimm_imm`	[7:0]	8	LOADIMM 8-bit immediate
`displ`	[8:0]	9	Long branch displacement (BR-type)
`disps`	[5:0]	6	Short branch displacement (BRR-type)
`cl`	[3:0]	4	Shift/rotate count (SHL/SHR/ROL/ROR)

6.5.2 Decode (ID)

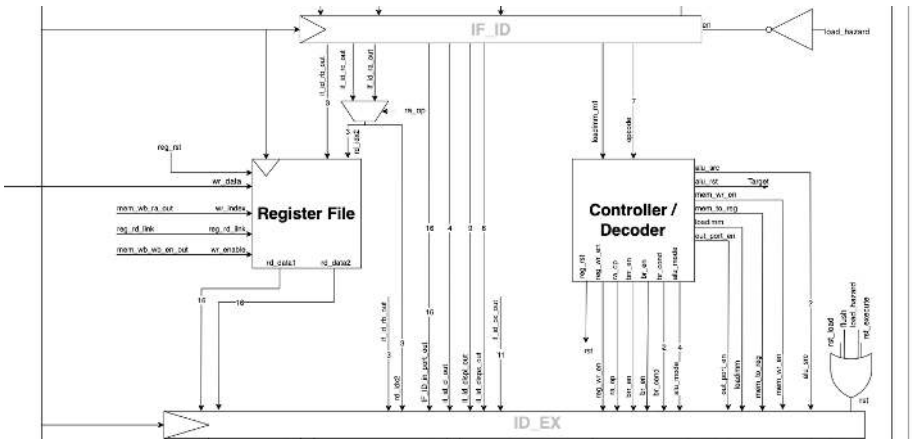

Figure 5: Decode Stage Schematic

The Instruction Decode (ID) stage is where the raw instruction word fetched in the previous stage gets turned into something the rest of the pipeline can actually act on. The opcode and `loadimm_m1` flag are handed off to the Controller/Decoder, which generates all of the control signals that downstream stages need, such as `alu_mode`, `alu_src`, `reg_wr_en`, `mem_wr_en`, `mem_to_reg`, the branch controls (`br_en`, `brr_en`, `br_cond`), and `out_port_en`. At the same time, the register file is read using `if_id_rb_out` on the first read port and `rd_idx2` on the second, where `rd_idx2` is chosen by the `ra_op` mux between `if_id_rc_out` and `if_id_ra_out`

depending on whether the instruction treats rA as a source. Writes coming back from the MEM/WB stage are committed to the register file in this same stage, with `reg_rd_link` handling the link register for branch-and-link instructions. Once the operands and control signals are ready, everything is latched into the ID/EX pipeline register so the execute stage can pick it up on the next clock edge. The ID/EX latch can also be cleared by `flush`, `load_hazard`, or `rst_execute`, which is how the hazard detection and branch resolution logic inserts a bubble when an instruction needs to be squashed without disturbing anything earlier in the pipeline.

This ID/EX register holds the most bits in our pipeline, at 108 bits total. The following table shows each signal and corresponding bit range and width for reference.

Table 9: ID/EX Pipeline Register Fields (110 bits total)

Field	**Bit Range**	**Width**	**Description**
`loadimm`	[109]	1	LOADIMM instruction flag
`loadimm_imm`	[108 : 101]	8	LOADIMM 8-bit immediate value
`mem_wr_en`	[100]	1	Data memory write enable
`mem_to_reg`	[99]	1	Selects memory data for writeback
`rc`	[98 : 96]	3	Register C index
`rb`	[95 : 93]	3	Register B index
`displ`	[92 : 84]	9	Long branch displacement
`disps`	[83 : 78]	6	Short branch displacement
`pc`	[77 : 67]	11	Program counter passed through
`br_cond`	[66 : 65]	2	Branch condition code
`br_en`	[64]	1	Absolute branch enable
`brr_en`	[63]	1	Relative branch enable
`out_port_en`	[62]	1	Output port enable
`ra`	[61 : 59]	3	Register A index (writeback destination)
`wb_en`	[58]	1	Register file writeback enable
`alu_src`	[57 : 56]	2	ALU second operand select
`alu_mode`	[55 : 52]	4	ALU operation select
`cl`	[51 : 48]	4	Shift/rotate count
`in_port`	[47 : 32]	16	Input port data
`d2`	[31 : 16]	16	Register file read port 2 data
`d1`	[15 : 0]	16	Register file read port 1 data

6.5.3 Execute (EX)

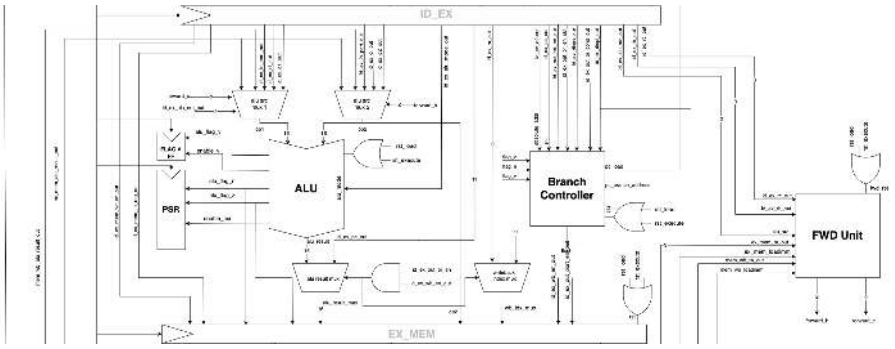

Figure 6: Execute Stage Schematic

The Execute (EX) stage performs the arithmetic, logical, and branch computations for each instruction. Two operand muxes (`alu_src_mux_1` and `alu_src_mux_2`) feed the ALU, with their selects driven by `id_ex_alu_src_out` together with the forwarding controls `forward_c` and `forward_b` from the FWD Unit. Operand one can be the register file read on port one (`d1`), the zero-extended LOADIMM immediate, the shift count `cl`, or the input port data, while operand two is normally `d2` from the register file. The Forwarding Unit watches the `ra` fields of the EX/MEM and MEM/WB stages against the current `rb` and `rc` indices, and when it detects a RAW hazard it overrides the appropriate operand with `ex_mem_alu_result_out` or `mem_wb_alu_result_out` so that back-to-back dependent instructions can run without stalling. The ALU itself is driven by `id_ex_alu_mode_out` and produces a 16-bit result along with the zero, negative, and overflow flags, the latter of which is latched into a dedicated Flag V flip-flop while the full set of flags is captured in the PSR under `enable_psr`. Note that the PSR and Flag V flip-flop sit outside the EX/MEM pipeline register and act as shared state across the execute stage rather than per-instruction state, which means a branch instruction reads whichever flags were most recently committed by the ALU.

At the same time, the Branch Controller takes `br_en`, `brr_en`, `br_cond`, the displacements, the absolute target from `d2`, and the current PC, and uses the ALU flags to decide whether the branch is taken. When it is, it asserts `pc_load` with the computed `pc_branch_address` and raises `flush` so that any instructions already in IF/ID and ID/EX get squashed. For branch-and-link instructions, the `alu_result_mux` substitutes the current PC for the ALU result so the return address is what gets written back, and the writeback index mux forces the destination to register seven. Everything the MEM stage will need, including the ALU result, the captured flags, the writeback controls, and the precomputed store address, is then latched into the EX/MEM pipeline register at the end of the cycle.

Table 10 shows the signals being passed through to the next section of the pipeline.

Table 10: EX/MEM Pipeline Register Fields (37 bits total)

Field	Bit Range	Width	Description
`loadimm`	[36]	1	LOADIMM instruction flag
`mem_wr_en`	[35]	1	Data memory write enable
`mem_to_reg`	[34]	1	Selects memory data for writeback
`store_addr`	[33 : 23]	11	Computed store address for memory writes
`out_port_en`	[22]	1	Output port enable
`ra`	[21 : 19]	3	Writeback destination register index
`wb_en`	[18]	1	Register file writeback enable
`flag_n`	[17]	1	ALU negative flag
`flag_z`	[16]	1	ALU zero flag
`alu_result`	[15 : 0]	16	16-bit ALU result

6.5.4 Memory Access (MEM)

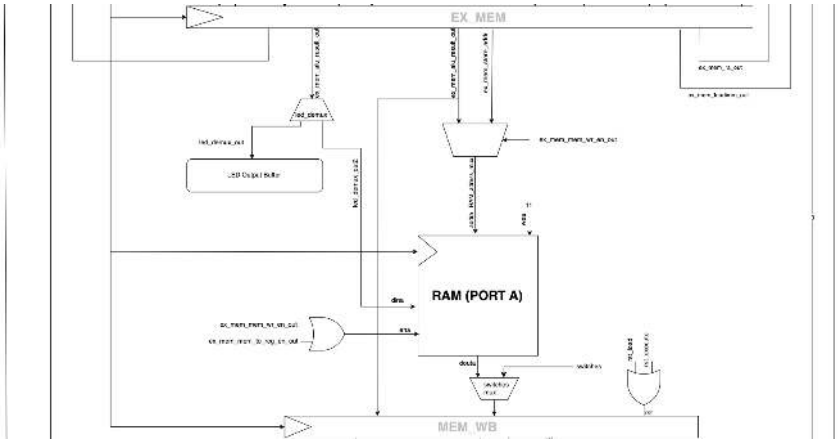

Figure 7: Memory Stage Schematic

The Memory (MEM) stage is where the CPU talks to data RAM. Most instructions just pass straight through this stage without doing anything, but load and store instructions use it to read from or write to memory. A small mux on the RAM address input picks where the address actually comes from, depending on whether the current instruction is a load or a store. On a store, the address is taken from the dedicated `store_addr` field that was computed back in EX and carried forward in the EX/MEM register, while the value being stored comes from the ALU result. On a load, the ALU result itself is the address, so the mux instead routes the lower 11 bits of `alu_result` to the RAM. The RAM's write enable is also tied to `mem_wr_en` so that the memory only commits a write on store cycles and behaves as read-only the rest of the time. Whatever the load returns, along with the ALU result and all the writeback control signals, is then latched into the MEM/WB pipeline register so the writeback stage has everything it needs on the next clock edge.

Table 11: MEM/WB Pipeline Register Fields (39 bits total)

Field	Bit Range	Width	Description
`loadimm`	[38]	1	LOADIMM instruction flag
`mem_to_reg`	[37]	1	Selects memory data over ALU result for writeback
`load_data`	[36 : 21]	16	Data read from data memory
`out_port_en`	[20]	1	Output port enable
`ra`	[19 : 17]	3	Writeback destination register index
`wb_en`	[16]	1	Register file writeback enable
`alu_result`	[15 : 0]	16	ALU result passed through from EX/MEM

6.5.5 Write Back (WB)

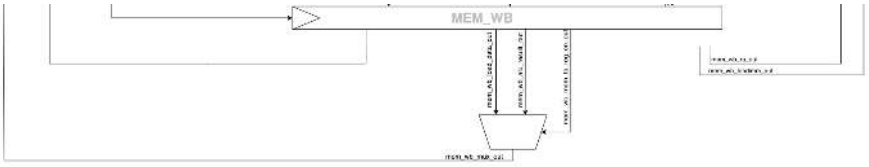

Figure 8: Write Back Stage Schematic

Finally, the Writeback (WB) stage is the last stage in our pipeline and, as its name suggests, is responsible for committing results back to the register file. To achieve this, the CPU uses the control lines passed down through the MEM/WB pipeline register. A small mux at the writeback input selects between the ALU result forwarded from EX/MEM and the data loaded from memory in the MEM stage, with `mem_wb_mem_to_reg_en_out` acting as the select line. On a load instruction this select is high so that the value read from RAM (`mem_wb_load_data_out`) is what gets written back, while for every other instruction it is low and the ALU result (`mem_wb_alu_result_out`) is written instead. The output of this mux, `mem_wb_mux_out`, is then routed to the register file write port and committed to the register indexed by `mem_wb_ra_out` whenever `mem_wb_wb_en_out` is asserted. The same value is also fed back to the Forwarding Unit so that an instruction currently in EX can pick it up if it depends on the result being written this cycle, and it drives the external `out_port` latch whenever `mem_wb_out_port_en_out` is high so that OUT instructions can present data to the outside world.

6.6 Data Hazard Handling

6.6.1 Forwarding Unit

Listing 7: Forwarding Unit Pseudocode

```
// EX Hazard

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd /= 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10
if (EX/MEM.RegWrite
```

```
and (EX/MEM.RegisterRd =/ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10

// MEM Hazard Logic

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd /= 0)
and (EX/MEM.RegisterRd /= ID/EX.RegisterRs)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd /= 0)
and (EX/MEM.RegisterRd /= ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

The Forwarding Unit is what lets back-to-back dependent instructions run through the pipeline without stalling. It sits alongside the Execute stage and watches the destination register indices of the two instructions ahead of the current one, namely `ex_mem_ra_out` for the instruction in MEM and `mem_wb_ra_out` for the instruction in WB, and compares them against the source register indices `id_ex_rb_out` and `id_ex_rc_out` of the instruction currently entering EX. Whenever it sees a match and the producing instruction is actually going to write back to the register file, it raises one of two select signals, `forward_b` or `forward_c`, which tell the operand muxes in the EX stage to grab the value directly from the later pipeline register instead of from the register file. A value of `"10"` forwards from EX/MEM and a value of `"01"` forwards from MEM/WB, with EX/MEM taking priority since it holds the more recent result. The unit also handles the LOADIMM corner case by checking `ex_mem_loadimm` and `mem_wb_loadimm`, since LOADIMM always writes its result into register seven and needs to be forwarded the same way as a normal ALU result when a following instruction reads that register.

6.6.2 Hazard Detection Unit

The Hazard Detection Unit handles the one situation that the Forwarding Unit cannot fix on its own, namely the load-use hazard. When a load instruction is sitting in the EX stage and the instruction immediately behind it in ID needs to read the value that load is about to fetch, forwarding cannot save us, since the loaded data does not actually exist yet at the end of EX. It only becomes available one cycle later at the end of MEM, which means the consuming instruction has to wait one cycle before it can proceed. The hazard detection unit is the small block of combinational logic that recognizes this case and inserts that stall.

It sits between the IF/ID and ID/EX pipeline registers and watches for three things. The destination register of the instruction currently in ID/EX (`id_ex_ra`), the `mem_to_reg` bit of that same instruction (which is high only for loads), and the two source register indices of the instruction currently in IF/ID (`if_id_rb` and `if_id_rc`). If a load is in EX and either of the source registers of the following instruction matches the load's destination, the unit raises `load_hazard`, which the datapath uses to implement a stall and bubble in the style described in Patterson and Hennessy [1]. The stall is achieved by disabling the program counter so the same instruction is fetched again next cycle, disabling the IF/ID register so the dependent instruction stays put in decode, and resetting the ID/EX register so the load's slot in EX is replaced with a NOP. After one cycle of stalling the load reaches MEM, its result becomes available on

the MEM/WB forwarding path, and the dependent instruction can finally enter EX with the correct operand routed in by the Forwarding Unit. Together the two units handle every RAW hazard in the pipeline, with forwarding covering the common case at zero cost and hazard detection paying a one-cycle penalty only when a load is immediately consumed.

7 Console Display

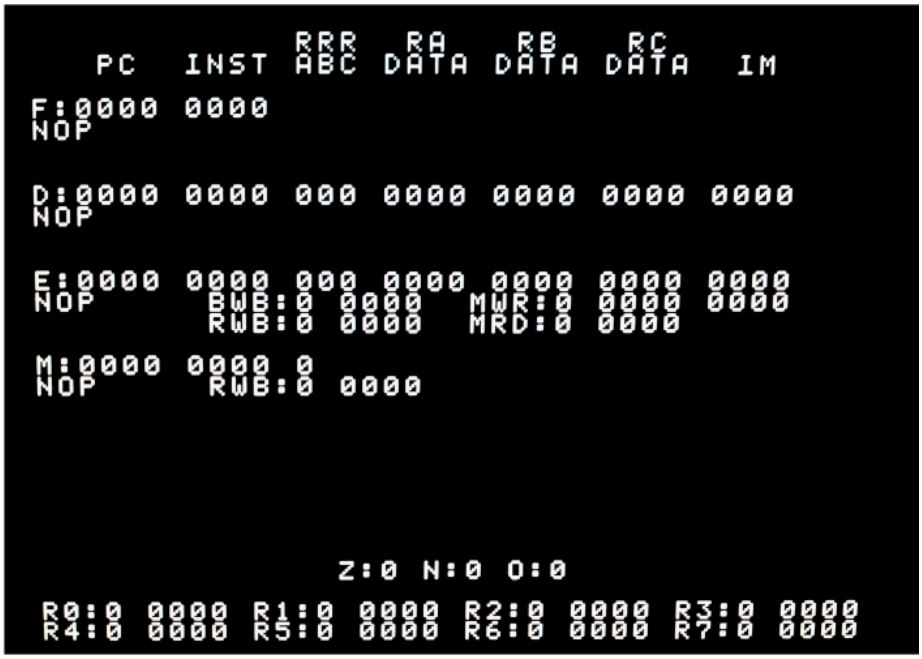

Figure 9: Console Display Output

The Console Display is a text-to-VGA module provided to students for debugging purposes. It displays the live state of the CPU pipeline onto a standard 640×480 monitor, showing the program counter, raw instruction word, register indices, register file contents, immediates, writeback signals, memory read and write activity, the Z/N/V status flags, and all eight architectural registers, broken out by stage so each one can be inspected independently. It also decodes opcodes into their mnemonic text descriptions (with the exception of the `BRR.O` instruction, which is not in the lookup table), and exposes a memory-mapped character buffer that lets the CPU itself write custom text to the screen via `OUT`-style instructions. This VHDL module was instantiated in our top-level entity and used throughout the hardware implementation phase to debug the bootloader, since being able to watch the fetched bytes, packet decoding, and RAM writes happen in real time made it much easier to identify where the bootloader was breaking down than waveform inspection alone would have allowed.

8 System Integration and Test Setup

This section documents the test environment the processor was deployed into, for future reference. The processor runs on a Basys 3 FPGA board paired with an STM32F0DISCOVERY board that provides the bootloader and clock control infrastructure used throughout development. Both are mounted on a shared carrier board along with a keypad and physical step/run buttons.

8.1 Toolchain

Test programs are written in the provided assembly language and assembled using the command-line `assembler` tool, which produces three output files: a `.lst` listing file used by the FPGA Loader, and `.hex` and `.coe` files used to initialize the ROM block in Vivado. An example command is `assembler -o factorial factorial.asm`, which omits the extension from the output name.

8.2 FPGA Loader

The FPGA Loader is a desktop application that transfers an assembled `.lst` file to the STM32F0 over USB using OpenOCD. It also selects the CPU clock frequency (1 Hz, 10 Hz, 100 Hz, 1 kHz, 10 kHz, or 100 kHz) and reports the number of instructions, origins, and packets in the loaded program. A byte-addressable or word-addressable mode can be selected from the File menu. Once the file is loaded, the STM32F0 streams the program into RAM through the memory-mapped input port.

8.3 Load and Execute Sequence

To run a program, the test CPU is first placed in Reset and Load mode, which vectors the PC to address `0x0002` and runs the load handler in the BIOS ROM. The handler reads instruction and address packets from the input port and writes them into RAM. Once loading completes, the CPU is switched to Reset and Execute mode, which vectors the PC to `0x0000` and begins executing from the start of the loaded program. The first three words of RAM are reserved by the bootloader to hold a jump to the actual program entry point, which is determined by the final `ORG` directive in the source file.

8.4 Board Controls

Three physical buttons provide manual clock control for debugging. STEP manually toggles the clock, advancing the pipeline by one edge per press. NEXT runs the clock at the selected frequency until an acknowledgement signal is received, which is useful for advancing past a single instruction or bootloader packet. RESUME runs the clock continuously at the selected frequency until pressed again. An external clock can also be applied by relocating the clock source jumper and connecting a signal to the external clock loops on the carrier board.

8.5 Pin Assignments

The top-level CPU signals are mapped to Basys 3 pins through the Vivado XDC constraints file. A summary of the top-level I/O and their assigned pins is provided in

Table 12: Boot Loader Pin Assignments (STM32F0 to FPGA)

STM32F0 Signal	Direction	Logic Vector	PMOD Pin	FPGA Pin
System Clock	Output	`std_logic`	JC1	K17
Load	Output	`input_port(7)`	JC10	R18
Address/Data	Output	`input_port(6)`	JC9	P17
Acknowledge	Input	`output_port(0)`	JC2	M18
Data Bit 7	Output	`input_port(15)`	JB10	C16
Data Bit 6	Output	`input_port(14)`	JB9	C15
Data Bit 5	Output	`input_port(13)`	JB8	A17
Data Bit 4	Output	`input_port(12)`	JB7	A15
Data Bit 3	Output	`input_port(11)`	JB4	B16
Data Bit 2	Output	`input_port(10)`	JB3	B15
Data Bit 1	Output	`input_port(9)`	JB2	A16
Data Bit 0	Output	`input_port(8)`	JB1	A14

9 Results

Simulations and Waveforms Throughout the design process, the architecture was tested incrementally. Each instruction format was tested individually to confirm proper operation. The initial tests were conducted using Vivado's simulation tools. The incremental testing process consisted of running specific assembly code testbenches confined to use the instructions relevant to the format being tested. During the early design stage, all instructions were separated by four NOP instructions due to the lack of RAW hazard detection and resolution capabilities. Listings containing sample testbenches are contained in Appendix B for reference. Additionally a sample waveform used for verification is shown. Figures 10 and 11 show the contents of the register file after the first and second SUB instructions (FormatBTest1 listing in Appendix B) demonstrating proper forwarding. This test was conducted during the Format B stage of the project.

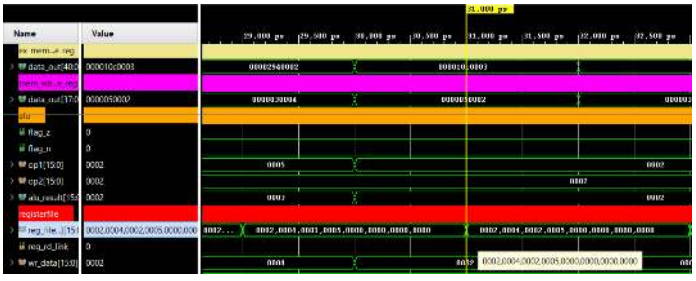

Figure 10: Register File Contents After First SUB

Figure 11: Register File Contents After Second SUB

Hardware Testing The final stage of the project consisted of testing the CPU on hardware. These tests were conducted on an Artix-7CPG236 model FPGA. As discussed in the design limitations section, the bootloader's reset load function was not rendered operational for this project. Thus, tests were conducted by generating the bitsream with the RAM contents pre-populated. The tests were conducted with the software Toolchain and STM32 which are discussed in section 8. This allowed the CPU to be tested at varying frequences and allowed for "step-through" debugging to be employed. The results of the program shown in listing 11 are included in this section. Figure 12 shows the configuration of the external switches used to test the overflow functionality. The screenshot was taken after program execution. The LED value of zero indicates overflow was detected (the error code of 0 is shown).

Figure 12: Switch Configuration for Overflow Test

As shown in Figure 13, the test successfully set the O flag and took the conditional overflow branch.

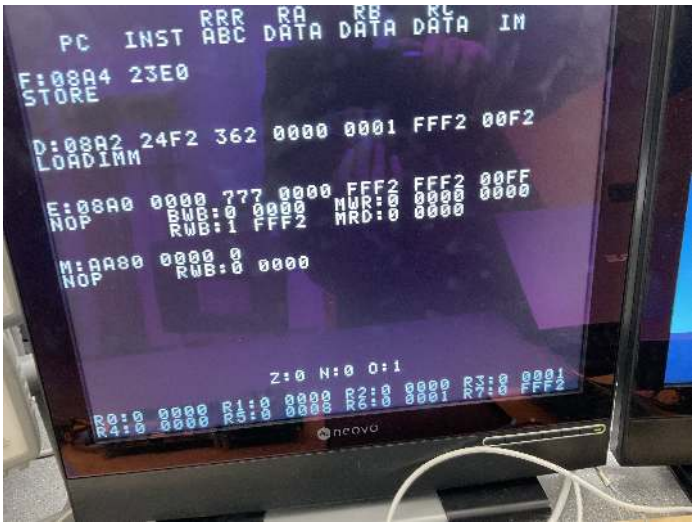

Figure 13: Overflow Test (0 = error) on Console Display

Figure 14 shows the output of 6 after an input of 3 is placed on the slide switches confirming proper operation in the non-overflow case. The success of these two tests also demonstrate the proper functionality of the memory map and reset and execute functionality.

Figure 14: Regular Factorial Test (3! = 6)

After functional testing was conducted, a timing and power analysis was performed on the completed design. The results are shown in Figures 15, 16 and 17 respectively. The results of the timing simulation indicate that no hold time violations are present in the design. This agrees with the results of the functional test as hold time violations are irrespective of the clock frequency. Additionally, the report shows that clocking the CPU with a 100MHz clock (the default used for the report is the onboard clock frequency) results in a 5ns setup time violation.

A more detailed examination of these results is present in the discussions section. The power estimate results are simply included for reference and documentation. Power consumption is a major constraint of modern processors, however due to the simplicity of the CPU implemented, is not a concern in this case.

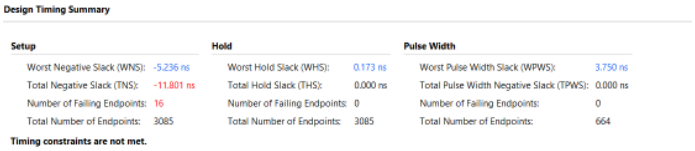

Figure 15: Timing Report Provided By Vivado

Figure 16: Worst Case Setup Time Violations

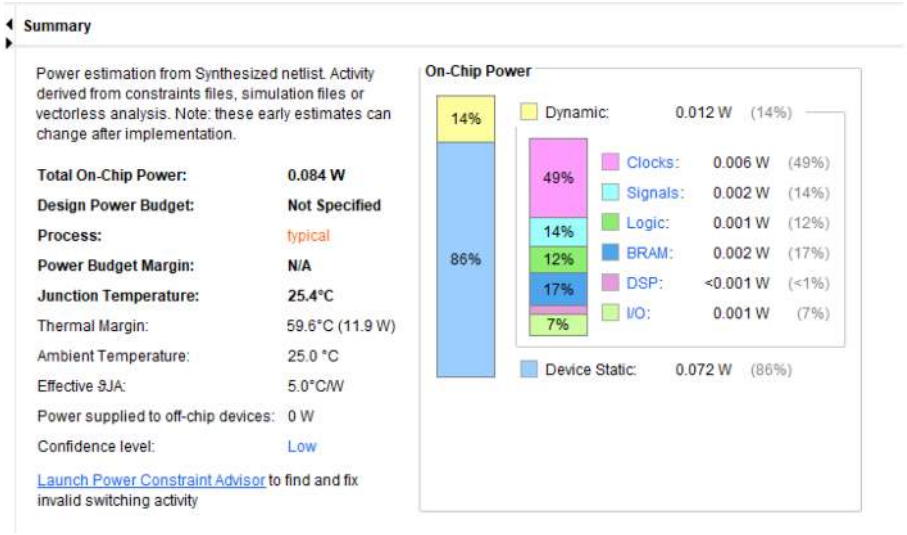

Figure 17: On-Chip Power Summary with 10ns sys clk

Finally, a utilization report was generated. The complete contents of the report are contained in listing 12 in Appendix D along with the implementation schematic. Notable is the utilization of 1653 total LUT primitives, 532 Register Primitives, and 247 Mux primitives. Additionally a single DPSP48E1 block (DSP block) was used. It is likely that the DSP Block was utilized to implement the combinational multiplier.

10 Discussion

To describe your design limitations, shortcomings, and extra features.
(b) To summarize errors, pitfalls, and problems encountered while doing the project and how did you handle them.
(c) To comment on the results.

Design Limitations Although the minimum requirements specified by the ISA were fulfilled by our design, the CPU is still very limited in its capabilities.

A significant limitation is the inability to load user code from an external source. This arises from the lack of functional reset and load capability. This resulted from a failure to implement and test the CPU with a bootloader capable of this functionality. As discussed in the bootloader section, a minimalistic bootloader was employed that does not feature reset and load capabilities. The bootloader simply assumes the RAM contains the proper instructions and executes an absolute branch instruction to the pre-configured address of 0x0824.

Another limitation is the performance of our processor. As discussed in the lecture slides, the basic equation for CPU performance is:

$$CPU\_Time = \frac{InstructionCount \times CPI}{ClockRate} \tag{1}$$

Optimization of the instruction count parameter is beyond the scope of this project and is mainly the domain of software design and optimization. However, both the CPI and clock rate of our processor could be improved.

One option to improve the CPI of our processor is to introduce a branch prediction scheme. As discussed in the Branch Controller section (6.4), all branches are evaluated in the execute stage of the pipeline. This is not optimal for two main reasons:

1. Theoretically, unconditional branches can be resolved in the Decode stage.
2. Even simple prediction schemes can reduce the CPI penalty for branches significantly.

A simple yet popular scheme for static branch prediction is BTFN [2]. BTFN consists of predicting forward branches as taken and backward branches as untaken. A forward branch consists of a positive relative branch address whereas a backwards branch consists of a negative branch address. Thus, all that is needed to implement this scheme is the addition of a magnitude comparator in the decode stage, additional gates in the execute stage to control the flush command, and the addition of logic to compute the branch target in the decode stage. The logic should be such that the flush command is only pulled high when the prediction is wrong. A flowchart complete with the stage the additional logic would be required in is shown in Figure 18

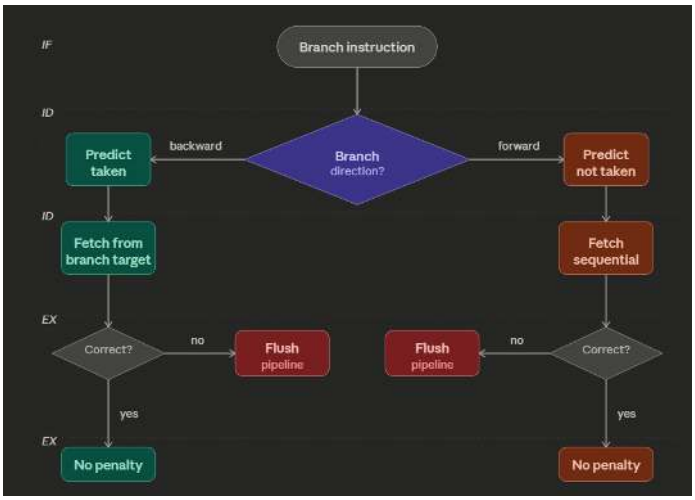

Figure 18: BTFN Branch Prediction Algorithm

The second factor that could be modified to improve performance is the clock frequency. As discussed in the results section, the maximum speed for which the processor was successfully tested was 100kHz. This result has many implications on the success of our design. First, it implies that the post route implementation of our VHDL source is such that the CPU can be implemented on the FPGA without hold time violations. As discussed in [3], hold time violations occur as a result of either:

1. Large combinational logic delays between synchronous registers
2. Large clock skew between synchronous registers

Hold time violations occur independent of the clock frequency and result in a non functional design. The confirmed functionality of our processor on an FPGA indicates that the design effectively separated the combinational logic delay between pipeline stages. Additionally, it indicates that the source code followed best practices for synthesizing clock distribution circuits on an FPGA to minimize clock skew. This is also reflected in the Vivado timing report which shows positive hold time slack.

As opposed to hold time violations, setup time violations are frequency dependent. The occurrence of setup time violations is often the limiting factor for clock frequency. To increase the setup slack (and therefore are maximum clock frequency), many strategies could be employed. Namely, large blocks of combinational logic such as the multiplier could be made sequential. This would consist of modifying the signed multiplication logic to implement a sequential algorithm (such as Booth's algorithm [4]). This would reduce combinational logic delay and increase the slack time. Interestingly the timing reports show that this specific optimization would have a direct effect on the maximum clock frequency (shown in Figures 16 and 15). The timing report uses the board's 100MHz clock for violation reports. Note that some signals have a requirement of 5ns and some have a requirement of 10ns. This is a direct result of our design decision to clock these registers on the falling edge (cutting the slack time in

half). The worst case delay is the delay between the ID_EX and flag_v registers. This path contains the large combinational multiplier and suffers from the fact that the flag_v register is clocked on the falling edge so that the flag is available in the EX stage where branches are evaluated.

Errors and Pitfalls During the design process, numerous pitfalls hampered progress on the project.

The first main pitfall was attempting to design the ALU logic at the gate level. This revealed the importance of hierarchical design when designing large systems. By using existing libraries (numeric_std), the complexity of the arithmetic function was abstracted and the design was simplified.

Secondly, the concept of pipelining was originally misunderstood. Originally, all the control logic was placed in a sequential state-machine based controller. This is counter to the pipelining design methodology which consists of placing the control signals together with the instruction within the pipeline registers. In this way, all the internal state (memory) required for the functional units to properly process the instruction is stored together with the instruction. This allows multiple instructions to pass through the pipeline simultaneously.

11 Conclusion

This project successfully demonstrated the design, implementation, and hardware verification of a 16-bit pipelined Harvard architecture processor implementing the full ECE 449 instruction set. The final design supports all required A, B, and L format instructions, resolves data hazards through a combination of forwarding and load-use stall insertion, handles control hazards by flushing the IF/ID and ID/EX pipeline registers on taken branches, and extends the base ISA with a BRR.OVERFLOW instruction supported by a dedicated overflow flag in the program status register. The processor was synthesized onto a Xilinx Artix-7 FPGA and verified in hardware by executing a factorial program that exercised arithmetic, conditional branching, memory-mapped I/O, and the overflow detection extension, with all test cases producing the expected results.

The most significant lesson was the shift in mindset required to move from a sequential, state-machine-based controller to a pipelined design where control signals travel alongside their instructions through the pipeline registers. Equally instructive was the role of hierarchical design and the importance of leveraging existing libraries such as `numeric_std` rather than implementing arithmetic units at the gate level. Debugging the design end-to-end, from individual component simulation in Vivado through to hardware testing on the Basys 3 board with the live VGA console display, reinforced the value of incremental verification at every level of the design hierarchy.

The design also exposed several opportunities for improvement that were beyond the scope of this project. The combinational multiplier dominates the critical path and limits the achievable clock frequency; replacing it with a sequential implementation such as Booth's algorithm would directly improve the worst-case setup slack reported by Vivado. Resolving branches in the Execute stage incurs a three-cycle penalty on every taken branch, which a simple static prediction scheme such as BTFN could substantially reduce. Finally, the lack of a working bootloader load path remains the most significant functional limitation,

since user code must currently be embedded into the RAM initialization at synthesis time rather than streamed in at runtime. Addressing these limitations would transform the processor from a working academic exercise into a more practically usable embedded core, while the foundation built through this project, namely the pipeline structure, hazard handling, and verification infrastructure, provides a solid starting point for that future work.

References

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 5th ed. Waltham, MA: Morgan Kaufmann, 2014.

[2] V. C. Hamacher, Z. G. Vranesic, and S. G. Zaky, *Computer Organization*, 5th ed. USA: McGraw-Hill, Inc., 2001.

[3] P. P. Chu, *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability.* Wiley-IEEE Press, 2006.

[4] A. D. Booth, "A signed binary multiplication technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

Appendix A Control Line Mapping

Table 13: Controller Output Signals per Instruction

Instr.	alu_mode	alu_src	ra_op	reg_wr_en	reg_rd_link	brr_en	br_en	br_cond	mem_wr_en	mem_to_reg
NOP	0000	00	0	0	0	0	0	00	0	0
ADD	0001	00	0	1	0	0	0	00	0	0
SUB	0010	00	0	1	0	0	0	00	0	0
MUL	0011	00	0	1	0	0	0	00	0	0
NAND	0100	00	0	1	0	0	0	00	0	0
SHL	0101	01	1	1	0	0	0	00	0	0
SHR	0110	01	1	1	0	0	0	00	0	0
TEST	0111	00	1	0	0	0	0	00	0	0
OUT	1000	00	1	0	0	0	0	00	0	0
IN	0000	10	0	1	0	0	0	00	0	0
BRR	0000	00	0	0	0	1	0	00	0	0
BRR.N	0000	00	0	0	0	1	0	01	0	0
BRR.Z	0000	00	0	0	0	1	0	10	0	0
BR	0000	00	1	0	0	0	1	00	0	0
BR.N	0000	00	1	0	0	0	1	01	0	0
BR.Z	0000	00	1	0	0	0	1	10	0	0
BR.SUB	0000	00	1	1	0	0	1	00	0	0
RETURN	0000	00	0	0	1	0	1	00	0	0
BRR.OVF	0000	00	0	0	0	1	0	11	0	0
LOAD	0000	00	0	1	0	0	0	00	0	1
STORE	0000	00	1	0	0	0	0	00	1	0
LOADIMM	1001/1010	00	0	1	1	0	0	00	0	0
MOV	0000	00	0	1	0	0	0	00	0	0

Appendix B Test Code Listings

Listing 8: Format A Test Program

```
ORG 0x0000
IN r1   ; 03
IN r2 ; 05
NOP
```

```
NOP
NOP
NOP
ADD r3, r2, r1
NOP
NOP
NOP
NOP
SHL r3, 2
NOP
NOP
NOP
NOP
MUL r2, r1, r3
NOP
NOP
NOP
NOP
OUT r2
NOP
END
```

Listing 9: Format B Test Program

```
ORG  0x0000
IN R0 ; 02  ; This example tests how data dependencies are handled
IN R1 ; 03  ; The values to be loaded into the corresponding resgister.
IN R2 ; 01
IN R3 ; 05  ;  End of initialization
ADD R1, R1, R2 ; r1 should be 4
SUB R2, R1, R0 ; r2 should be 6
SUB R1, R3, R2 ; r1 should be -1
END
```

Listing 10: Format C Test Program

```
ORG 0x08
IN R1    ; 5
LOADIMM.LOWER 5
LOADIMM.UPPER 0
MOV R2, R7
LOADIMM.LOWER 2
LOADIMM.UPPER 0
MOV R3, R7
LOADIMM.LOWER 1
LOADIMM.UPPER 0
MOV R4, R7
NAND R1, R1, R3
SHL R1 1
SUB R2, R2, R4
TEST R2
BRR.Z 2
BRR -5
OUT R1
BRR 0
END
```

Listing 11: Factorial Program with Overflow Detection

```
LedDisplay: equ   0xFFF2
DipSwitches:  equ   0xFFF0
DipSwitchMask:  equ   0x07
    org   0x0820
MAIN:
    loadimm.upper DipSwitches.hi
    loadimm.lower DipSwitches.lo
    load    r6, r7
    loadimm.upper 0x00
    loadimm.lower 0x01
    mov   r4, r7
    mov   r3, r7
    test    r6
    brr.z   DONE
    sub   r6, r6, r3    ; 0 and 1 factorial should return 1
    test    r6
    brr.z   DONE
    loadimm.upper 0x00
    loadimm.lower 0x02
    mov   r5, r7
LOOP:
    mul   r4, r4, r5
    brr.overflow  OVERFLOW    ; branch if multiply overflowed
    add   r5, r5, r3
    sub   r6, r6, r3
    test    r6
    brr.z   DONE
    brr   LOOP
OVERFLOW:
    loadimm.upper 0x00 ; output 0x0000 on overflow
    loadimm.lower 0x00
    mov   r4, r7
DONE:
    loadimm.upper LedDisplay.hi
    loadimm.lower LedDisplay.lo
    store   r7, r4
    brr   DONE
```

Appendix C Complete CPU Block Schematic

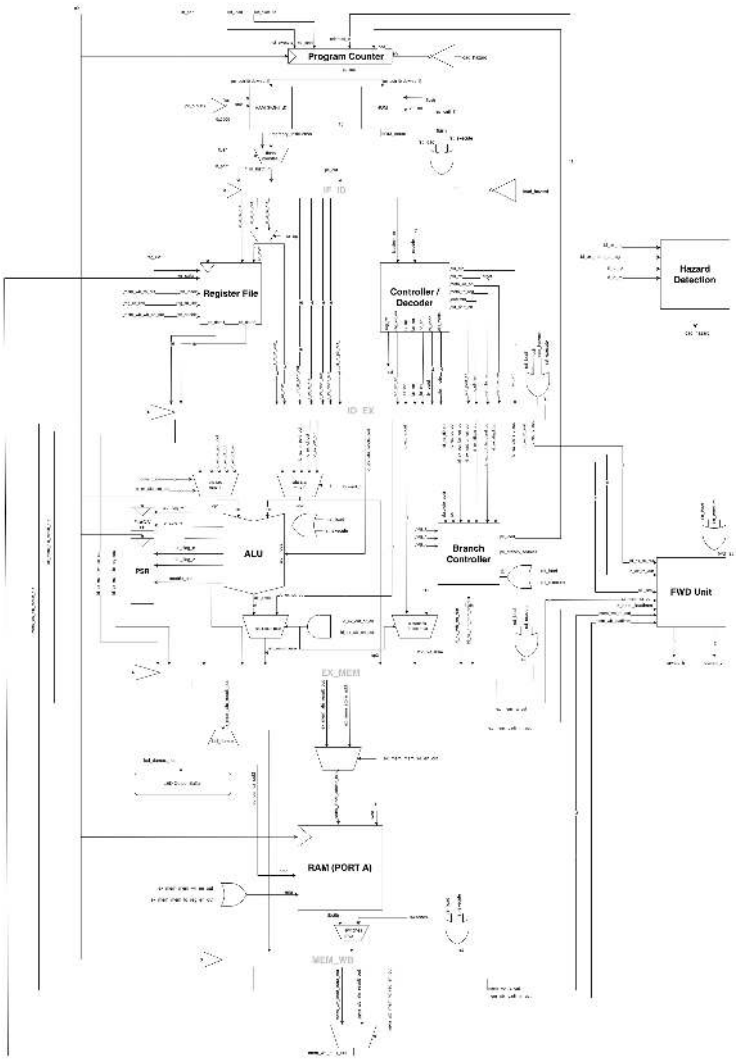

Figure 19: Detailed Datapath Block Diagram

Appendix D Full Utilization Report

Listing 12: Uilizarion Report

```
Copyright 1986-2022 Xilinx, Inc. All Rights Reserved. Copyright 2022-2024 Advanced
     Micro Devices, Inc. All Rights Reserved.
-------------------------------------------------------------------------------------------------------
| Tool Version : Vivado v.2024.2 (win64) Build 5239630 Fri Nov 08 22:35:27 MST 2024
| Date         : Sun Apr 12 12:17:52 2026
| Host         : ws11-22 running 64-bit major release  (build 9200)
| Command      : report_utilization -file cpu_top_level_utilization_synth.rpt -pb
     cpu_top_level_utilization_synth.pb
| Design       : cpu_top_level
| Device       : xc7a35tcpg236-1
| Speed File   : -1
| Design State : Synthesized
-------------------------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Memory
3. DSP
4. IO and GT Specific
5. Clocking
6. Specific Feature
7. Primitives
8. Black Boxes
9. Instantiated Netlists

1. Slice Logic
--------------

+----------------------------+------+-------+------------+-----------+-------+
|          Site Type         | Used | Fixed | Prohibited | Available | Util% |
+----------------------------+------+-------+------------+-----------+-------+
| Slice LUTs*                | 1653 |     0 |          0 |     20800 |  7.95 |
|   LUT as Logic             | 1397 |     0 |          0 |     20800 |  6.72 |
|   LUT as Memory            |  256 |     0 |          0 |      9600 |  2.67 |
|     LUT as Distributed RAM |  256 |     0 |            |           |       |
|     LUT as Shift Register  |    0 |     0 |            |           |       |
| Slice Registers            |  532 |     0 |          0 |     41600 |  1.28 |
|   Register as Flip Flop    |  483 |     0 |          0 |     41600 |  1.16 |
|   Register as Latch        |   49 |     0 |          0 |     41600 |  0.12 |
| F7 Muxes                   |  274 |     0 |          0 |     16300 |  1.68 |
| F8 Muxes                   |   11 |     0 |          0 |      8150 |  0.13 |
+----------------------------+------+-------+------------+-----------+-------+
* Warning! The Final LUT count, after physical optimizations and full implementation,
     is typically lower. Run opt_design after synthesis, if not already completed, for
     a more realistic count.
Warning! LUT value is adjusted to account for LUT combining.
Warning! For any ECO changes, please run place_design if there are unplaced instances


1.1 Summary of Registers by Type
--------------------------------

+-------+--------------+-------------+--------------+
```

Total	Clock Enable	Synchronous	Asynchronous
0	_	-	-
0	_	-	Set
0	_	-	Reset
0	_	Set	-
0	_	Reset	-
0	Yes	-	-
0	Yes	-	Set
49	Yes	-	Reset
0	Yes	Set	-
483	Yes	Reset	-

2. Memory

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	0	0	0	50	0.00
RAMB36/FIFO*	0	0	0	50	0.00
RAMB18	0	0	0	100	0.00

* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E1 or one FIFO18E1. However, if a FIFO18E1 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E1

3. DSP

Site Type	Used	Fixed	Prohibited	Available	Util%
DSPs	1	0	0	90	1.11
DSP48E1 only	1				

4. IO and GT Specific

Site Type	Used	Fixed	Prohibited	Available	Util%
Bonded IOB	61	0	0	106	57.55
Bonded IPADs	0	0	0	10	0.00
Bonded OPADs	0	0	0	4	0.00
PHY_CONTROL	0	0	0	5	0.00
PHASER_REF	0	0	0	5	0.00
OUT_FIFO	0	0	0	20	0.00
IN_FIFO	0	0	0	20	0.00
IDELAYCTRL	0	0	0	5	0.00
IBUFDS	0	0	0	104	0.00
GTPE2_CHANNEL	0	0	0	2	0.00
PHASER_OUT/PHASER_OUT_PHY	0	0	0	20	0.00
PHASER_IN/PHASER_IN_PHY	0	0	0	20	0.00
IDELAYE2/IDELAYE2_FINEDELAY	0	0	0	250	0.00
IBUFDS_GTE2	0	0	0	2	0.00
ILOGIC	0	0	0	106	0.00

| OLOGIC | 0 | 0 | 0 | 106 | 0.00 |

5. Clocking

Site Type	Used	Fixed	Prohibited	Available	Util%
BUFGCTRL	4	0	0	32	12.50
BUFIO	0	0	0	20	0.00
MMCME2_ADV	0	0	0	5	0.00
PLLE2_ADV	0	0	0	5	0.00
BUFMRCE	0	0	0	10	0.00
BUFHCE	0	0	0	72	0.00
BUFR	0	0	0	20	0.00

6. Specific Feature

Site Type	Used	Fixed	Prohibited	Available	Util%
BSCANE2	0	0	0	4	0.00
CAPTUREE2	0	0	0	1	0.00
DNA_PORT	0	0	0	1	0.00
EFUSE_USR	0	0	0	1	0.00
FRAME_ECCE2	0	0	0	1	0.00
ICAPE2	0	0	0	2	0.00
PCIE_2_1	0	0	0	1	0.00
STARTUPE2	0	0	0	1	0.00
XADC	0	0	0	1	0.00

7. Primitives

Ref Name	Used	Functional Category
LUT6	721	LUT
FDRE	483	Flop & Latch
LUT3	329	LUT
MUXF7	274	MuxFx
RAMD64E	256	Distributed Memory
LUT5	241	LUT
LUT4	241	LUT
LUT2	117	LUT
LDCE	49	Flop & Latch
OBUF	31	IO
IBUF	30	IO
CARRY4	25	CarryLogic
MUXF8	11	MuxFx
LUT1	10	LUT
BUFG	4	Clock
DSP48E1	1	Block Arithmetic

```
8. Black Boxes
--------------

+----------+------+
| Ref Name | Used |
+----------+------+
| CPU_ROM  |    1 |
+----------+------+


9. Instantiated Netlists
------------------------

+----------+------+
| Ref Name | Used |
+----------+------+
```

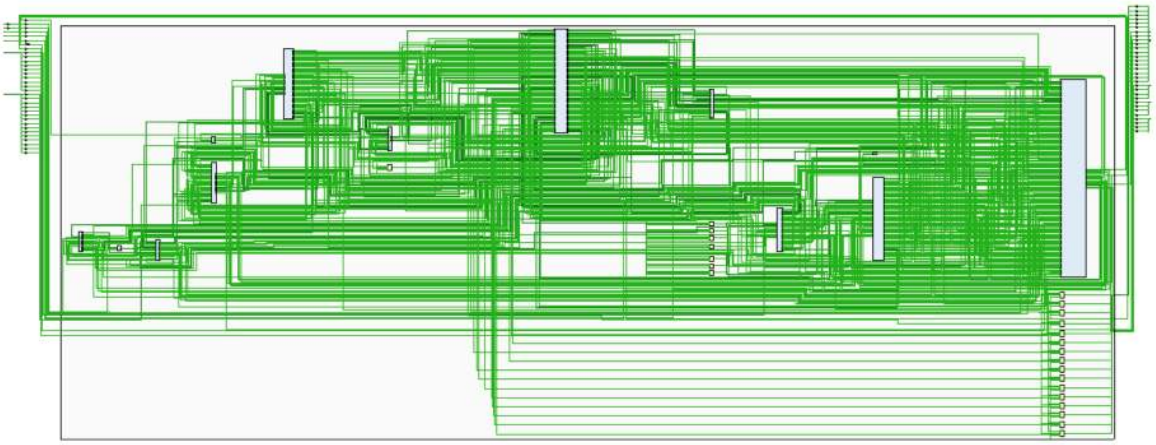

Figure 20: Implementation Schematic