CPE - 3150
PROJECT 1
DATE: 03/12/2024

Microcontrollers Project 1
CPE - 3150 - Spring 2024

Project Title: **Creating Modified WIMP AVR Processor**

Team Members: **Trenton Cathcart (tcgyq@mst.edu, EE)**

Jamie Madison (jrmn64@mst.edu, EE)

Instructor: **Dua, Rohit (rdua@mst.edu, MS&T EE)**

Presented to the
Electrical & Computer Engineering Faculty of
Missouri University of Science and Technology
In partial fulfillment of the requirements for
Microcontrollers (CPE 3150)
March 2024 (SPRING 2024)

Table of Contents

1. INTRODUCTION

1.1. EXECUTIVE SUMMARY

In our project, we embarked on enhancing the WIMPAVR processor, a venture aimed at integrating new instructions into its existing framework while ensuring seamless operation on the Altera DE2 board. Our chosen instructions for this modification were CP (Compare), CPI (Compare with Immediate), and CPC (Compare with Carry), each designed to broaden the processor's operational capabilities and align with the advanced requirements of modern computing tasks.

The WIMPAVR processor, as we engaged with it, was initially configured with a set of instructions that allowed for basic processing tasks. Our preliminary tasks included a thorough review of the WIMPAVR resources and hands-on experience with the processor on the Altera board, which provided us with a solid foundation for our modification work. Our efforts were guided by the goal to create a version of the WIMPAVR processor in Quartus II, using Block Diagram Files (BDF), that not only incorporated our new instructions but also maintained the integrity and functionality of the original instruction set.

1.2. OUR INSTRUCTIONS

In our project, we delved into the implementation of three distinct yet fundamentally related instructions: CP (Compare), CPI (Compare with Immediate), and CPC (Compare with Carry), each leveraging the Atmel AVR Instruction Set Manual for guidance. Our objective was to enrich the WIMPAVR processor's capability with these operations, underpinning the modifications with a meticulously constructed status register. This register plays a pivotal role in the execution of compare commands, as it updates based on the computation flags: H (Half Carry), S (Sign), V (Two's complement overflow), N (Negative), Z (Zero), and C (Carry).

CP (Compare) serves as the foundational instruction among the trio, designed to compare the contents of two registers, Rd and Rr, without altering their values. The operation essentially subtracts Rr from Rd and updates the status register based on this result, setting flags to guide subsequent conditional branching. It's the bedrock for decision-making processes within the processor, enabling it to take different actions based on comparative results.

CPI (Compare with Immediate) extends the functionality of CP by introducing the ability to compare a register's content, Rd, with an immediate constant, K. This instruction enriches the processor's comparison operations by allowing direct comparison with fixed values, thus facilitating more versatile and immediate decision-making without the need for preloading the immediate value into a register. Like CP, CPI updates the status register based on the outcome of the comparison.

CPC (Compare with Carry) further expands the comparison capabilities by considering not only the contents of two registers, Rd and Rr, but also the carry flag from a previous operation. This nuanced addition makes CPC invaluable for extended precision arithmetic or sequences of comparisons where the cumulative result, including any carry from preceding operations, is critical. The instruction adeptly weaves together the immediate result of the comparison with the historical computational context provided by the carry flag.

These instructions can be observed in depth within **Figures 1-3** Below.

49. CP – Compare

49.1. Description

This instruction performs a compare between two registers Rd and Rr. None of the registers are changed. All conditional branches can be used after this instruction.

Operation:

(i) Rd - Rr

	Syntax:	Operands:	Program Counter:
(i)	CP Rd,Rr	0 ≤ d ≤ 31, 0 ≤ r ≤ 31	PC ← PC + 1

16-bit Opcode:

0001	01rd	dddd	rrrr

49.2. Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	⇔	⇔	⇔	⇔	⇔	⇔

Figure 1: CP Instruction

50. CPC – Compare with Carry

50.1. Description

This instruction performs a compare between two registers Rd and Rr and also takes into account the previous carry. None of the registers are changed. All conditional branches can be used after this instruction.

Operation:

(i) Rd - Rr - C

	Syntax:	Operands:	Program Counter:
(i)	CPC Rd,Rr	0 ≤ d ≤ 31, 0 ≤ r ≤ 31	PC ← PC + 1

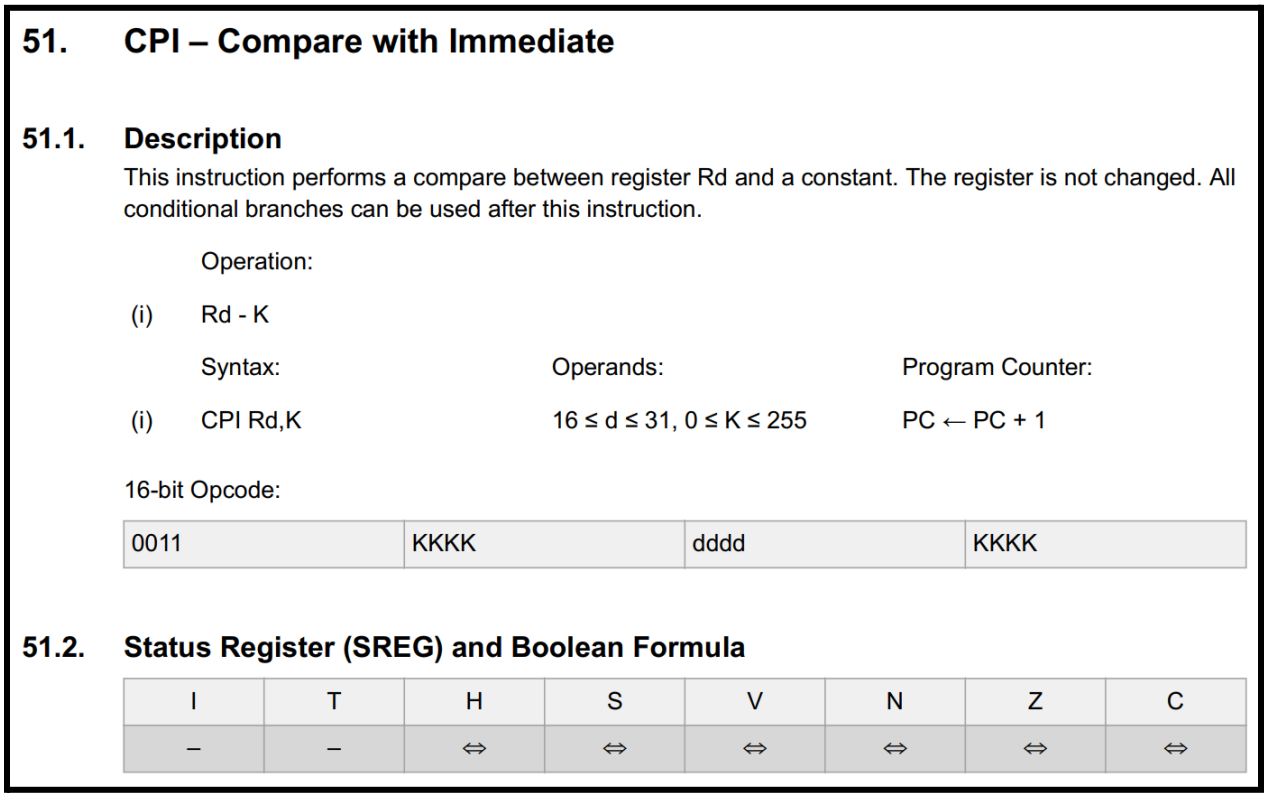
16-bit Opcode:

0000	01rd	dddd	rrrr

50.2. Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	⇔	⇔	⇔	⇔	⇔	⇔

Figure 2: CPC Instruction



51. CPI – Compare with Immediate

51.1. Description

This instruction performs a compare between register Rd and a constant. The register is not changed. All conditional branches can be used after this instruction.

Operation:

(i) Rd - K

	Syntax:	Operands:	Program Counter:
(i)	CPI Rd,K	16 ≤ d ≤ 31, 0 ≤ K ≤ 255	PC ← PC + 1

16-bit Opcode:

0011	KKKK	dddd	KKKK

51.2. Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	⇔	⇔	⇔	⇔	⇔	⇔

Figure 3: CPI Instruction

2. EXISTING DESIGN AND METHODS

In this section

2.1. OVERVIEW OF EXISTING PROCESSOR

The WIMPAVR processor we worked with is composed of multiple key components, each crucial for executing the instructions and operations the processor is capable of. Here's an overview of the various parts of the existing WIMPAVR system, as well as the areas where we contributed our enhancements:

- Slow Clock: This component is pivotal for managing the rate at which the processor operates, ensuring that each instruction cycle is given enough time to be properly processed and executed.

- Control Unit: The nerve center of the processor, the control unit interprets the instruction register's contents and sends appropriate control signals to orchestrate the operation of other processor components.

- State Machine: Working closely with the control unit, the state machine assists in the sequencing of operations, governing the processor's flow through various states in the instruction cycle.

- Instruction Register: This register holds the current instruction being executed, essentially serving as the immediate point of reference for what the processor should do next.

- Register File: The register file contains a set of registers that store data and intermediary computation results, available for quick access by the processor.

- Instruction Displays and 7-Segment Displays: These provide a hexadecimal visual representation of the instructions being processed, allowing for an accessible readout of the current operations for debugging and demonstration purposes.

- Program Counter: This keeps track of the address of the next instruction to be executed, ensuring the processor's workflow progresses in a logical and ordered manner.

- Arithmetic Logic Unit (ALU): The ALU performs all arithmetic and logical operations. It's where the computations for our implemented CP, CPI, and CPC instructions take place.

- Data Load Multiplexer (MUX): The MUX is responsible for selecting which data should be loaded into the processor for the next operation, based on the instruction requirements.

- Pin I/O: These input/output connections facilitate the interface between the processor and the Altera DE2 board, allowing for real-world interaction and functionality.

- Onboard SRAM: For our project, we leveraged the DE2's onboard SRAM to program our instruction sets. This allowed us to sequentially feed instructions to the processor, testing our enhancements in a live environment.

2.2. EXISTING INSTRUCTIONS

The existing instruction set of the WIMPAVR processor, as seen in the provided image, represents a suite of operations fundamental to any computation on the platform. Let's delve into the details of each instruction as outlined:

- LDI (Load Data Immediate): This instruction loads immediate binary data into the lower register bank (16 - 31). It's crucial for initializing registers with constant values that are used throughout program execution.

- MOV (Move): A simple yet essential instruction that copies data from one register to another, allowing for the transfer of values within the processor's register file without altering the data.

- ADC (Add with Carry): It adds two registers together and includes the carry flag in the result. This is important for operations requiring extended precision or when carrying out consecutive addition operations.

- AND (Logical AND): Performs a bitwise AND operation between two registers. The result is stored in one of the original registers, and this instruction is often used for bit masking and conditional logic.

- OR (Logical OR): This instruction performs a bitwise OR between two registers, useful for setting specific bits in a register to '1' based on the condition of bits in another register.

- EOR (Exclusive OR): Executes a bitwise XOR between two registers. It is particularly useful for toggling bits or for operations where you need to invert specific bits conditionally.

- SWAP (Swap Nibbles): Swaps the lower and upper nibbles (4-bit segments) within a register. This can be handy for certain byte manipulation operations or for formatting data.

- RJUMP (Relative Jump): Alters the program counter by a relative amount, effectively jumping to a new code location. It's a control flow instruction that enables loops, skips, and function calls.

- BRBS (Branch if Bit in SREG Set): A conditional branch that takes place if a specific bit in the status register (like the Zero flag) is set. It's fundamental for decision-making in the processor based on the outcomes of previous operations.

- SEC (Set Carry): Sets the carry flag in the status register to '1'. It's used in multi-byte arithmetic operations or to prepare the processor for certain operations that require the carry flag to be set.

- CLC (Clear Carry): Clears the carry flag in the status register, setting it to '0'. This is necessary to reset the carry flag status before commencing operations that do not need or must negate the previous carry.

3. DESIGN IMPLEMENTATION:

3.1. PRELIMINARY

Before diving into the implementation of our compare instructions, we diligently prepared by thoroughly testing the existing instructions on the Altera DE2 board. We programmed the processor onto the FPGA board and meticulously executed instruction sets sequentially, ensuring a comprehensive understanding of the processor's capabilities and behavior.

As we dissected the processor's structure, we explored each component layer by layer. It soon became clear to us that all our modifications would be centered within the ALU_TOP of the top-level design. The rationale behind this was that the core functionality of compare instructions is fundamentally a subtraction operation, which can be construed as the 2's complement addition of register values. The ALU_TOP.bdf can be observed within **Figure 4.**

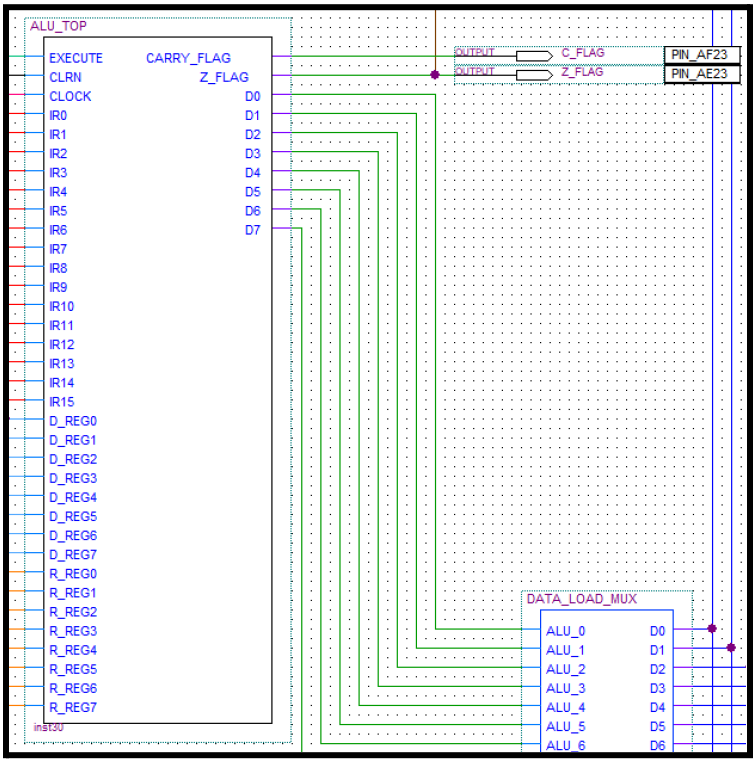


Figure 4: ALU_TOP.bdf

With the ADC instruction already implemented using an 8-bit ripple carry adder within this domain, we recognized that subtraction was inherently addition in 2's complement form. Given that addition was a native function of the processor, our task then pivoted to adapt this existing capability to serve our new instructions.

We also delved into the intricacies of the ALU top, particularly the ALU control and the existing status flags, which included the Z (Zero) and C (Carry) flags, established prior to our project. We had to understand how the original creators managed opcode transmission and instruction flow around the ALU top, and how computations were successfully executed. This involved a detailed examination of the existing data paths, control signals, and flag updates.

Our efforts were focused on ensuring that our compare instructions—CP, CPI, and CPC—would integrate seamlessly with the current architecture. This required not only an adaptation of the existing adder circuit but also a keen attention to the flag logic that dictates processor behavior post-computation. The flags, particularly, were crucial as they would determine the flow of operations following a compare instruction, influencing conditional branches and decision-making in subsequent instructions.

3.2. DESIGN PHASE

3.2.1. ENABLE SIGNALS - ALU CONTROL

We initiated our first modification to the ALU top by crafting control signals for each of our new instructions: CP, CPI, and CPC. These enable signals are critical because they prevent instruction interference within the ALU top. Utilizing the unique 16-bit opcodes of each instruction, these signals determine exactly which instructions are being processed at any given value of the program counter.

For instance, CP's non-variable opcode portion is defined by the six most significant bits, IR15 to IR10. These bits are constant and provide a clear indication to the rest of the ALU to identify which instruction is in execution. All of our opcode bits differed by at least one fixed bit from all existing instructions. An example of this differentiation is shown in **Figure 5**, where it is observed that the difference between the ADC and CP opcode lies in the IR11 bit.

In our writing, we emphasize that the modification from the existing ALU_CONTROL Block Diagram File (BDF) to the new ALU_Control BDF, now including control enable signals that incorporate our unique opcodes for CP, CPI, and CPC, is evident in **Figure 6**. Moreover, the logic implementation for the CP_enable signal, which is a control signal built from logic gates, can be observed within **Figure 7**. These enable signals are vital for the ALU to function correctly with the newly introduced instructions, as they direct the flow of operations and ensure that each instruction is isolated and managed appropriately during its execution cycle.

CP – Compare

0001	01rd	dddd	rrrr

ADC – Add with Carry

16-bit Opcode:

0001	11rd	dddd	rrrr

Figure 5: Differing Opcode

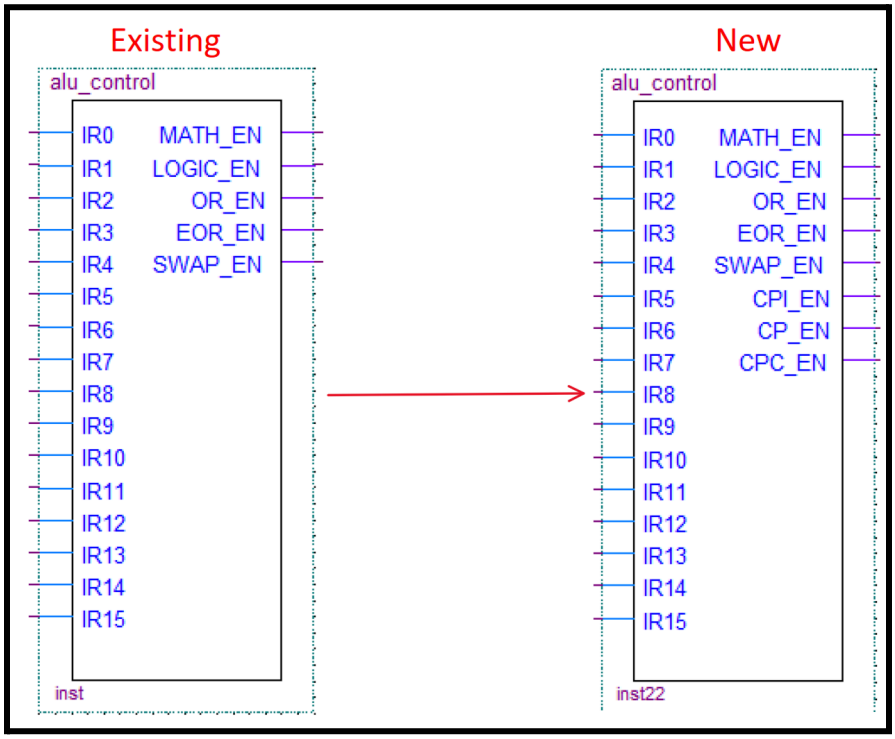


Figure 6: ALU_CONTROL Transformation

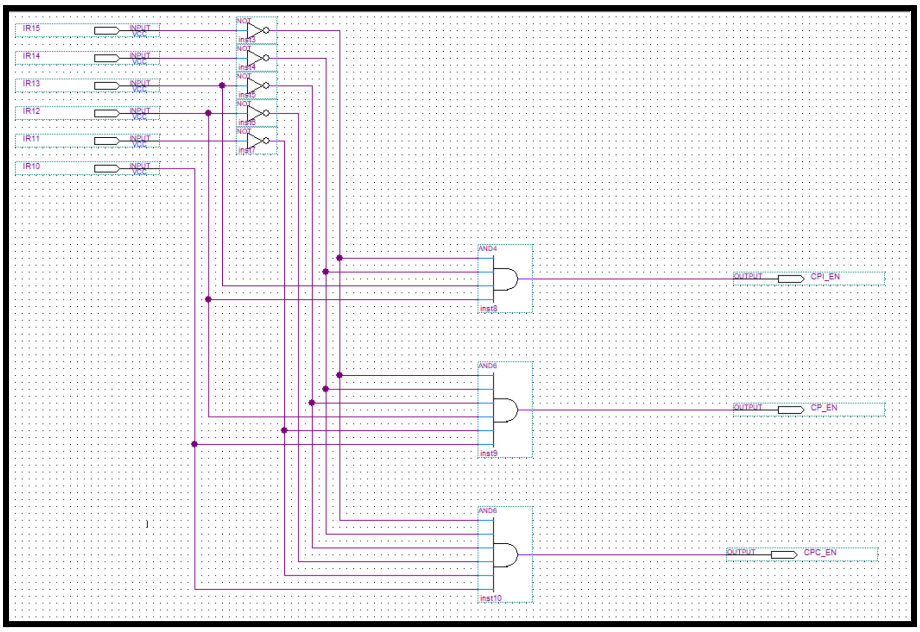


Figure 7: Compare Enable Logic

3.2.2. STATUS REGISTER CONSTRUCTION

In the construction of the status register for our WIMPAVR project, we leaned heavily on the ATMEL AVR instruction set manual, which meticulously details the Boolean algebra for each flag's output for the different instructions. We noted a significant divergence in the CPI instruction, where the Boolean expressions are dictated not by the values in the Rr register but by the opcode's immediate K values.

Most of these Boolean expressions, when translated from their verbose definitions, could be simplified. Take, for example, the carry flag—described by an extensive Boolean function in the manual, but in essence, it is set if the absolute value of the Rr register for CP and CPC, or the immediate K values for CPI, is larger than the value in register Rd. Essentially, if the result of a subtraction is negative, this indicates that the value of Rd is smaller than the absolute value of Rr or K. Consequently, we were able to distill the Boolean expressions to simply checking the most significant bit (MSB) of the subtraction result. These simplifications were applied to our project to streamline the logic needed within the ALU.

Furthermore, the details of the instruction status register descriptions and the associated Boolean functions are illustrated in **Figures 8 through 10**. It was clear to us that the flags needed to maintain their values for operations not associated with compare commands. Hence, within each flag's logic circuit, we incorporated D flip-flops to latch the flag outputs post-operation.

CP – Compare

H $\overline{Rd3} \cdot Rr3 + Rr3 \cdot R3 + R3 \cdot \overline{Rd3}$

Set if there was a borrow from bit 3; cleared otherwise.

S $N \oplus V$, for signed tests.

V $Rd7 \cdot \overline{Rr7} \cdot \overline{R7} + \overline{Rd7} \cdot Rr7 \cdot R7$

Set if two's complement overflow resulted from the operation; cleared otherwise.

N R7

Set if MSB of the result is set; cleared otherwise.

Z $\overline{R7} \cdot \overline{R6} \cdot \overline{R5} \cdot \overline{R4} \cdot \overline{R3} \cdot \overline{R2} \cdot \overline{R1} \cdot \overline{R0}$

Set if the result is $00; cleared otherwise.

C $\overline{Rd7} \cdot Rr7 + Rr7 \cdot R7 + R7 \cdot \overline{Rd7}$

Set if the absolute value of the contents of Rr is larger than the absolute value of Rd; cleared otherwise.

Figure 8: CP Status Register

CPI – Compare with Immediate

H $\overline{Rd3} \cdot K3 + K3 \cdot R3 + R3 \cdot \overline{Rd3}$

Set if there was a borrow from bit 3; cleared otherwise.

S $N \oplus V$, for signed tests.

V $Rd7 \cdot \overline{K7} \cdot \overline{R7} + \overline{Rd7} \cdot K7 \cdot R7$

Set if two's complement overflow resulted from the operation; cleared otherwise.

N R7

Set if MSB of the result is set; cleared otherwise.

Z $\overline{R7} \cdot \overline{R6} \cdot \overline{R5} \cdot \overline{R4} \cdot \overline{R3} \cdot \overline{R2} \cdot \overline{R1} \cdot \overline{R0}$

Set if the result is $00; cleared otherwise.

C $\overline{Rd7} \cdot K7 + K7 \cdot R7 + R7 \cdot \overline{Rd7}$

Set if the absolute value of K is larger than the absolute value of Rd; cleared otherwise.

Figure 9: CPI Status Register

CPC – Compare with Carry

H $\overline{Rd3} \bullet Rr3 + Rr3 \bullet R3 + R3 \bullet \overline{Rd3}$

Set if there was a borrow from bit 3; cleared otherwise.

S $N \oplus V$, for signed tests.

V $Rd7 \bullet \overline{Rr7} \bullet \overline{R7} + \overline{Rd7} \bullet Rr7 \bullet R7$

Set if two's complement overflow resulted from the operation; cleared otherwise.

N R7

Set if MSB of the result is set; cleared otherwise.

Z $\overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0} \bullet Z$

Previous value remains unchanged when the result is zero; cleared otherwise.

C $\overline{Rd7} \bullet Rr7 + Rr7 \bullet R7 + R7 \bullet \overline{Rd7}$

Set if the absolute value of the contents of Rr plus previous carry is larger than the absolute value of Rd; cleared otherwise.

Figure 10: CPC Status Register

However, we encountered a challenge with flags that were functions of other flags; directly using D flip-flops for these would cause the output to be dependent on two clock cycles, which was not acceptable for our design. To address this, we combined some of these interdependent flags into single input-output units that operated synchronously with the same clock, ensuring they would update together in a single clock cycle.

The construction of these status register flag logic BDF files is detailed in our documentation. Specifically, the H flag logic construction is depicted in **Figure 11**, while the logic for the N, S, and V flags can be seen in **Figure 12**. We also made modifications to the existing C and Z flag logic, which can be reviewed in **Figures 13 and 14**, respectively. By implementing these changes, we ensured that our compare instructions would correctly manipulate the status register flags and that the flags would reliably indicate the outcome of each operation.

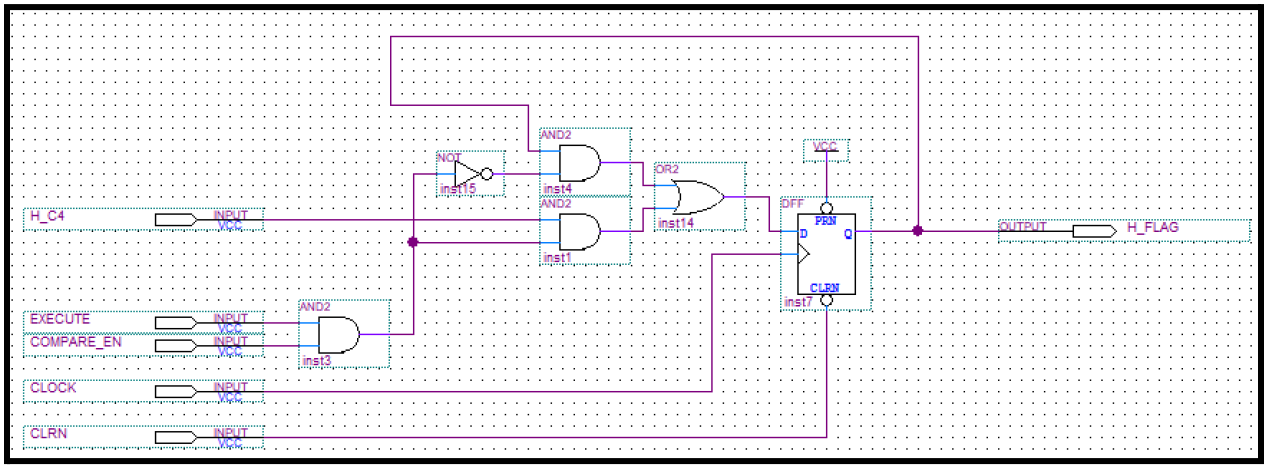


Figure 11: H Flag Construction

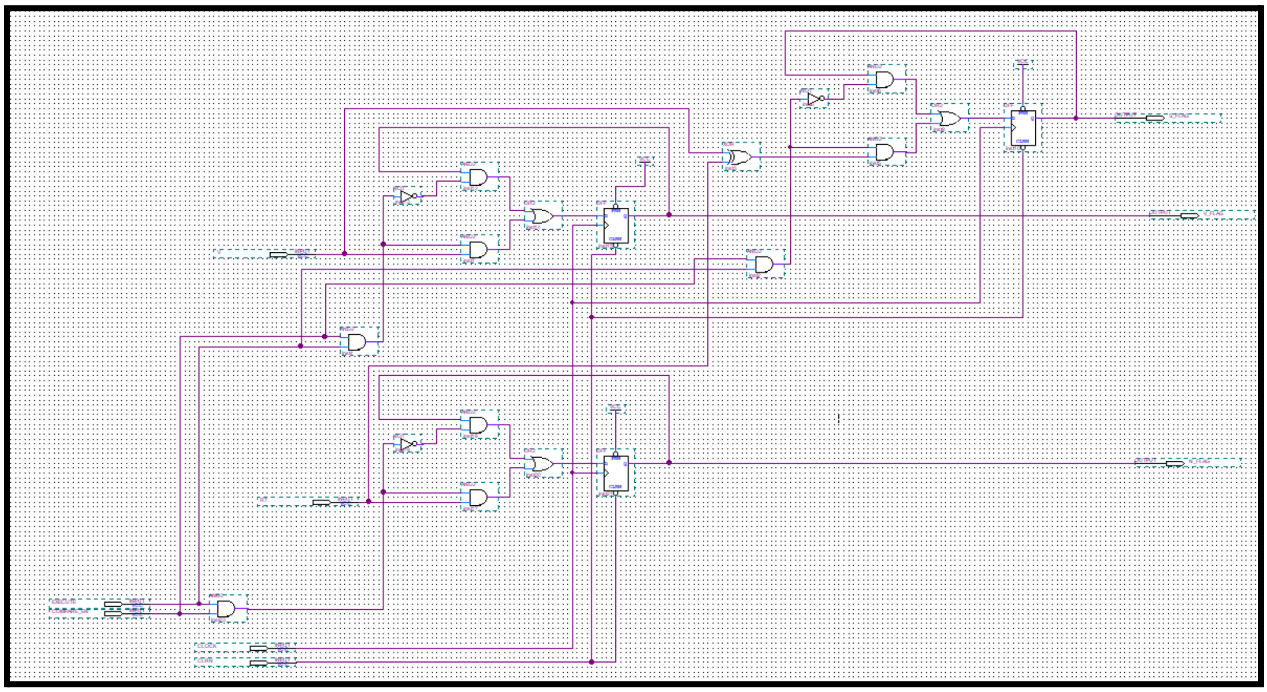


Figure 12: N,S, & V Flag Construction

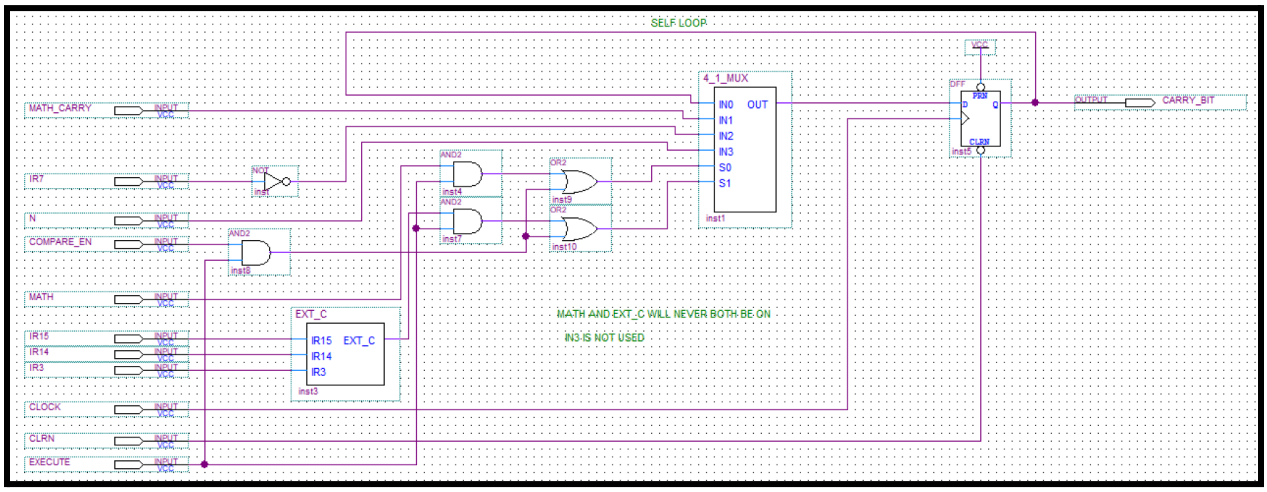


Figure 13: Carry Flag Construction

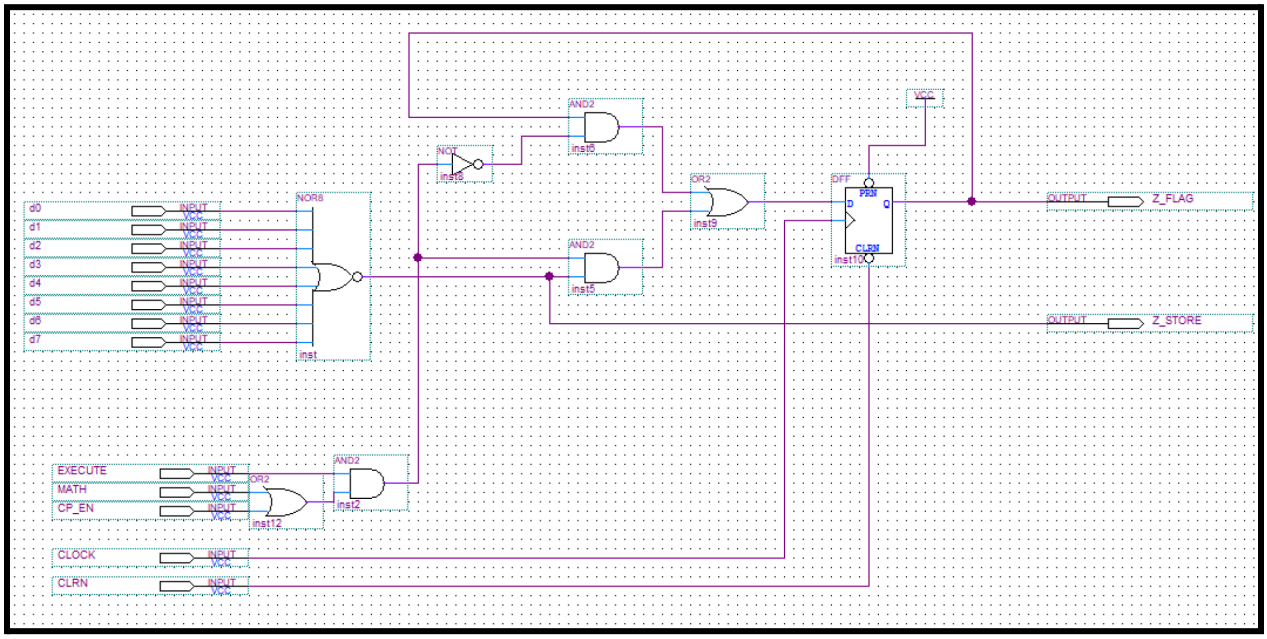


Figure 14: Z Flag Construction

3.2.3. ARITHMETIC IMPLEMENTATION

When we approached the arithmetic implementation, our aim was to utilize the existing 8-bit ripple carry adder to facilitate subtraction. To start, we modified the adder block to furnish additional outputs that would feed directly into the status register. Instead of relying on the ATMEL AVR instruction set's complex Boolean formulas, we opted for a more straightforward approach—for the H flag, which is triggered by a carry from the 4th to the 5th bit, we added an output on the carry line between these bits and routed it to serve as the input for the H flag. This modification, which we also applied for the V

flag output, enhanced the efficiency of flag status updates and can be seen detailed in **Figure 15**.

The next significant step was the creation of an add-subtract router. This involved leveraging the compare enable signal we previously constructed to control 2-to-1 multiplexers for each bit being compared. If any of the compare instructions were activated within the ALU Top, the router would trigger the 2's complement of the Rr register value, effectively inverting all bits and adding 1 to this inverted value. This 2's complemented value would then be routed to the 8-bit ripple adder to perform the subtraction between the two registers, or between the register and immediate data for CPI. If no compare instruction was detected, the operation would proceed as normal for any other instruction without passing through the bit inversion. The logic construction of this Add_Subtract_Router is demonstrated in **Figure 16**.

A complication arose when we recognized that we could not directly pass the bit values of the Rr register for all compare instructions, as the CPI instruction required immediate data encoded within the instruction opcode itself. To navigate this, we employed an 8-to-2-to-1 multiplexer, which used the CPI_enable control signal to determine whether to pass the relevant instruction bits or the Rr register values for the CP and CPC commands. The logic construction of this critical step is depicted in **Figure 17**.

This meticulous design approach ensured that our compare instructions worked seamlessly, leveraging the underlying hardware designed for addition and repurposing it for effective subtraction operations. Our alterations to the ALU's ripple carry adder and the thoughtful integration of the add-subtract router allowed for accurate computation of compare operations and proper flag status updates, which are integral to the processor's decision-making capabilities.

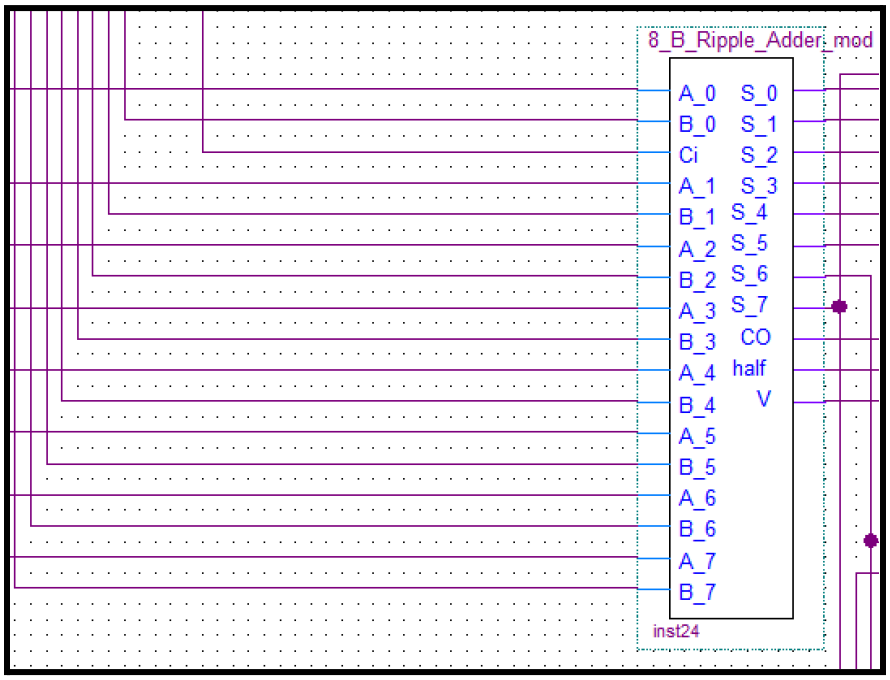


Figure 15: 8 Bit Ripple Modification

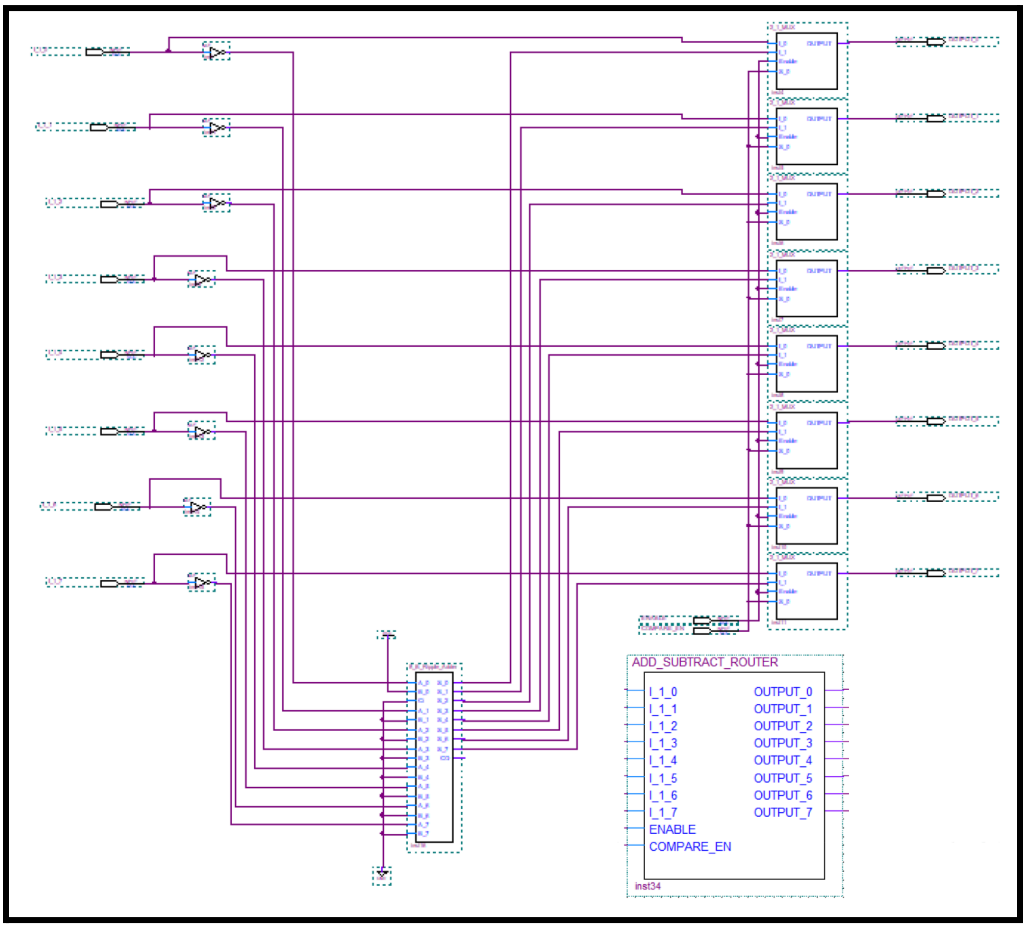


Figure 16: Add Subtract router

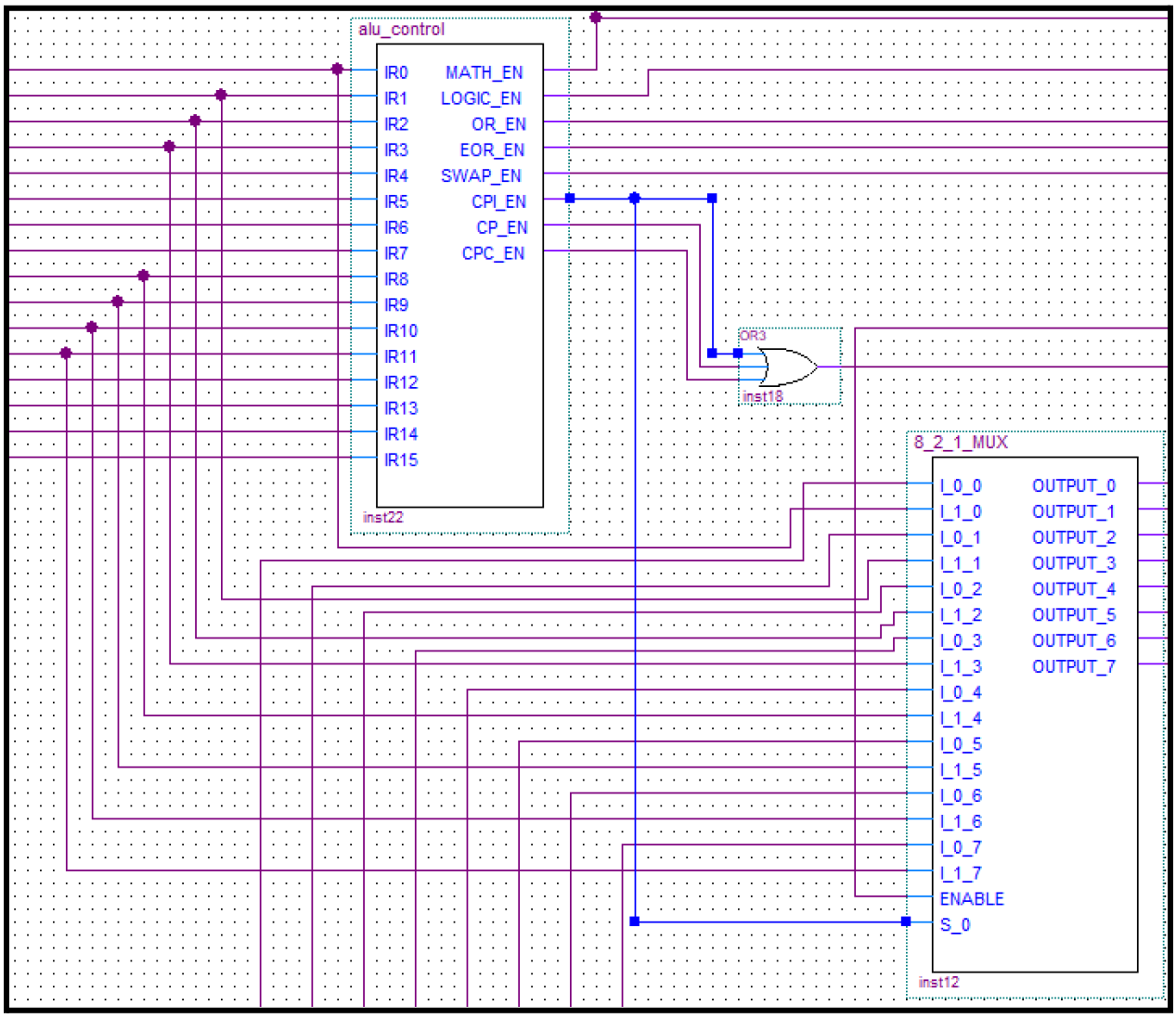


Figure 17: CPI Inclusion

3.2.4. SWAP CONFLICT AVOIDANCE

Our final task in implementing the compare instructions was to address the issue of storing the result of the compare operation. Typically, the output of an add operation from the ripple carry adder would be stored directly in the destination register, Rd. However, for the compare operations—CP, CPI, and CPC—this behavior wasn't desirable as we did not want to alter the contents of the Rd register; the compare operations are meant to affect the status flags without changing the registers being compared.

To circumvent this, we introduced an additional 8-to-2-to-1 multiplexer to control the routing of the adder's output. This multiplexer was designed to allow the result of the adder to be stored in the Rd register only when the compare instructions were not active. This was determined by the compare enable control signal, which, when indicating the presence of a compare instruction, would prevent the storage of the result. This means that during a compare operation, while the status flags are updated based on the result of the subtraction (2's complement addition), the Rd register remains unaffected—preserving its original value for use in subsequent operations.

The logic construction for this control mechanism, which ensures the selective storage of computational results, is showcased in **Figure 18**. By implementing this design, we ensured that the core functionality of the compare instructions was upheld and that the integrity of the data within the registers was maintained throughout the computational process.

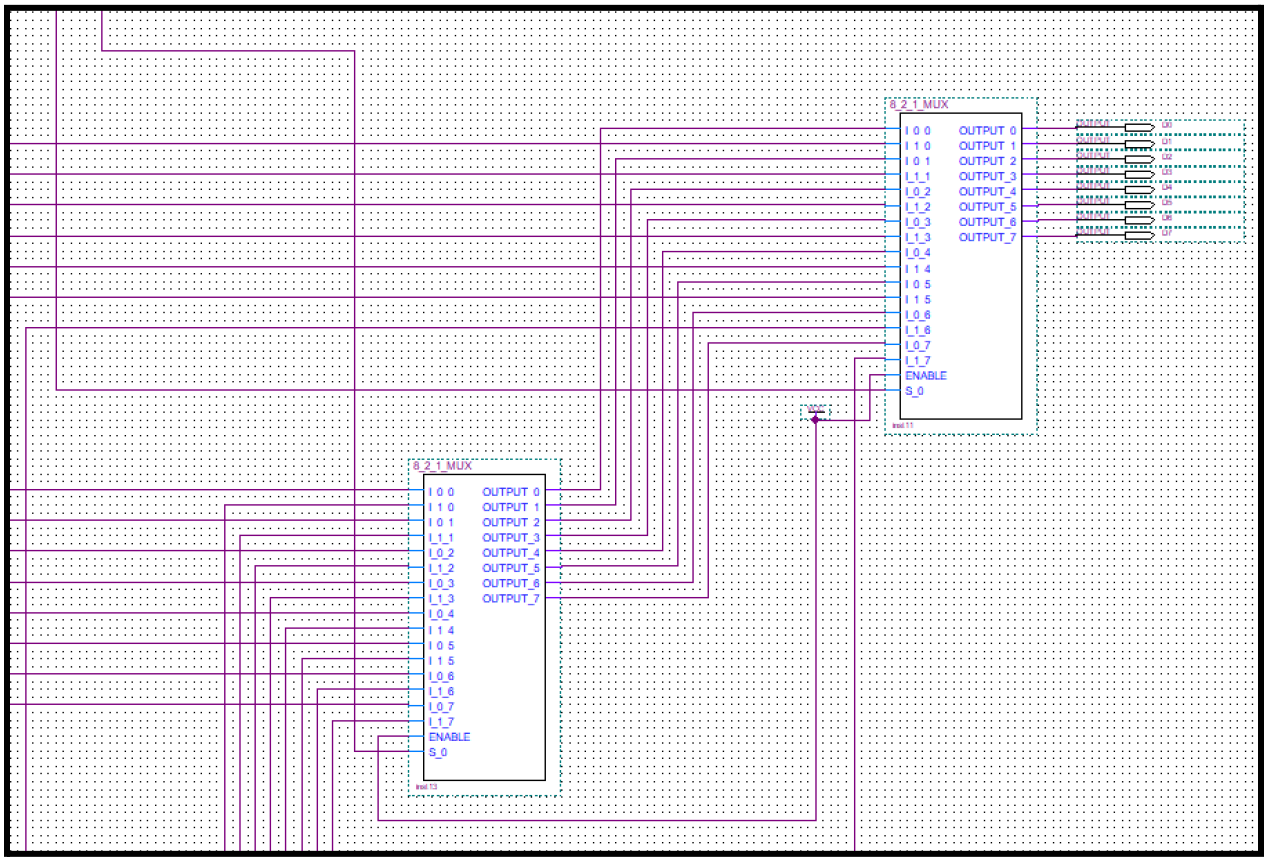


Figure 18: Register Storing Fix

4. PRESENTING

Presenting our project to the class and our instructor involved a crucial step: the development of a master test program. This program was comprehensive, encompassing all the existing instructions and integrating our new additions—CP, CPI, and CPC. We used the Quartus II Programmer to load this program onto the onboard SRAM of the Altera DE2 board.

Crafting this master test program required meticulous calculation to predict the expected outcomes and the status of the flags for each operation. We had to ensure that for every test case, we knew in advance which flags would be set or cleared, and what the results of the operations should be. This preemptive work was crucial for us to confidently demonstrate the functionality of our implementation and to validate the modifications made to the ALU and the status register logic.

The detailed outline of the master test program and its comprehensive coverage is documented in **Figure 19**. To complement this, we prepared a reference for all the corresponding flag outputs, which can be examined in **Figure 20**. This preparation allowed us to walk through each instruction set systematically, demonstrating the correct flag responses and the proper execution of each command in real-time.

During the presentation, we showcased the SRAM master test code's execution on the Altera DE2 board. By stepping through the test cases, we were able to illustrate the precise working of the compare instructions and the accurate status flag updates. Our instructor was able to observe firsthand the results displayed on the board, corresponding to the expectations we had laid out in our documentation. This live demonstration not only validated our design but also exhibited our understanding of the processor's architecture and our ability to extend its functionality through careful and deliberate engineering.

Labels	Memory Address	Assembly	Binary	Machine Code (HEX)	Description of Operation
	00H	LDI	1110 0000 1111 1000	E0F8	Loads 08 H into register 31, r31 = 08 H --- Register 31 Position = F = 1111
	01H	LDI	1110 0000 1110 1100	E0EC	Loads 0C H into register 30, r30 = 0C H --- Register 30 Position = E = 1110
	02H	LDI	1110 0000 1101 0011	E0D3	Loads 03 H into register 29, r29 = 03 H --- Register 29 Position = D = 1101
	03H	LDI	1110 0000 1100 0000	E0C0	Loads 00 H into register 28, r28 = 00 H --- Register 28 Position = C = 1100
	04H	LDI	1110 0000 1011 0000	E0B0	Loads 00 H into register 27, r27 = 00 H --- Register 27 Position = B = 1011
	05H	CLC	1001 0100 1000 1000	9488	Clears carry flag, C4 = 0
	06H	CP	0001 0101 1100 1011	15CB	Compares r28 (00 H) and r27 (00 H) and updates status register --- B=A --- Compares ZERO
	07H	CPI	0011 0000 1100 0111	30C7	Compares r28 (00 H) and rK (07 H) and updates status register --- A < B --- Compares ZERO
	08H	CLC	1001 0100 1000 1000	9488	Clears carry flag, C4 = 0
HERE:	09H	ADC	0001 1111 1110 1101	1FED	Adds r30 (0C H) and r29 (03 H) and saves to r30, r30 = 0F H
	10---0AH	CPC	0000 0111 1110 1100	07EC	Compares r30 (0F H) and r28 (00 H) with addition of a carry bit=0, and updates the status register --- A>B --- Compares ZERO
	11---0BH	CP	0001 0111 1111 1110	17FE	Compares r31 (08 H) and r30 (0F H) and updates status register --- B>A
	12---0CH	CPI	0011 0000 1111 0111	30F7	Compares r31 (08 H) and rK (07 H) and updates status register --- A = B
	13---0DH	SEC	1001 0100 0000 1000	9408	Sets carry flag, C4 = 1
	14---0EH	CPC	0000 0111 1110 1111	07EF	Compares r30 (0F H) and r31 (08 H) with addition of a carry bit, and updates the status register --- A>B
	15---0FH	AND	0010 0011 1111 1110	23FE	AND r31 (08 H) and r30 (0F H) and saves to r31, r31 = 08 H
	16---10H	OR	0010 1011 1111 1101	2BFD	OR r31 (08 H) and r29 (03 H) and saves to r31, r31 = 0B H
	17---11H	EOR	0010 0111 1111 1101	27FD	EXOR r31 (0B H) and r29 (03 H) and saves to r31, r31 = 08 H
	18---012H	SWAP	1001 0101 1101 0010	95D2	SWAPS Upper and Lower nibbles of r29 (03 H) and saves to r29, r29 = 30 H
	19---013H	CP	0001 0111 1111 1101	17FD	Compares r31 (08 H) and r29 (30 H) and updates status register --- A<B
	20---014H	CPI	0011 0000 1110 1000	30E8	Compares r30 (0F H) and rK (08 H) and updates status register --- A<B
	21---015H	CPC	0000 0111 1111 1101	07FD	Compares r31 (08 H) and r29 (30 H) and updates status register --- A<B
	22---016H	MOV	0010 1111 1111 1110	2FFE	Moves (Copies) r30 (0F H) to r31, r31 = (0F H)
	23---017H	CP	0001 0101 1100 1011	15CB	Compares r28 (00 H) and r27 (00 H) and updates status register --- B=A --- Compares ZERO
	24---018H	BRBS (Z)	1111 0000 0000 1001	F009	JUMPS TO PC + k + 1, if zero flag is set, PC = 23 + 1 + 1 = 25 --- 1111 00kk kkkk k001
	25---019H	RJMP	1100 1111 1110 1111	CFEF	JUMPS To PC + k, PC = 25 + -19 + 1 = 07 --- 1100 kkkk kkkk kkkk
STOP:	26---01AH	RJMP	1100 1111 1111 1111	CFFF	JUMPS To PC + k, PC = 23 + -1 + 1 = 23 --- 1100 kkkk kkkk kkkk

Figure 19: Master Test Program

Machine Code (HEX)	Description of Operation	C_FLAG - LEDR1	Z_FLAG - LEDR0	S_FLAG - LEDG7	N_FLAG - LEDG6	H_FLAG - LEDG5	V_FLAG - LEDG4
E0F8	Loads 08 H into register 31, r31 = 08 H --- Register 31 Position = F = 1111	0	0	0	0	0	0
E0EC	Loads 0C H into register 30, r30 = 0C H --- Register 30 Position = E = 1110	0	0	0	0	0	0
E0D3	Loads 03 H into register 29, r29 = 03 H --- Register 29 Position = D = 1101	0	0	0	0	0	0
E0C0	Loads 00 H into register 28, r28 = 00 H --- Register 28 Position = C = 1100	0	0	0	0	0	0
E0B0	Loads 00 H into register 27, r27 = 00 H --- Register 27 Position = B = 1011	0	0	0	0	0	0
9488	Clears carry flag, C4 = 0	0	0	0	0	0	0
15CB	Compares r28 (00 H) and r27 (00 H) and updates status register --- B=A --- Compares ZERO	0	1	0	0	0	0
30C7	Compares r28 (00 H) and rK (07 H) and updates status register --- A < B --- Compares ZERO	1	0	1	1	0	0
9488	Clears carry flag, C4 = 0	0	0	1	1	0	0
1FED	Adds r30 (0C H) and r29 (03 H) and saves to r30, r30 = 0F H	0	0	1	1	0	0
07EC	Compares r30 (0F H) and r28 (00 H) with addition of a carry bit=0, and updates the status register --- A>B --- Compares ZERO	0	0	0	0	0	0
17FE	Compares r31 (08 H) and r30 (0F H) and updates status register --- B>A	1	0	1	1	0	0
30F7	Compares r31 (08 H) and rK (07 H) and updates status register --- A = B	0	0	0	0	0	0
9408	Sets carry flag, C4 = 1	1	1	0	0	1	0
07EF	Compares r30 (0F H) and r31 (08 H) with addition of a carry bit, and updates the status register --- A>B	0	0	0	0	1	0
23FE	AND r31 (08 H) and r30 (0F H) and saves to r31, r31 = 08 H	0	0	0	0	1	0
2BFD	OR r31 (08 H) and r29 (03 H) and saves to r31, r31 = 0B H	0	0	0	0	1	0
27FD	EXOR r31 (0B H) and r29 (03 H) and saves to r31, r31 = 08 H	0	0	0	0	1	0
95D2	SWAPS Upper and Lower nibbles of r29 (03 H) and saves to r29, r29 = 30 H	0	0	0	0	1	0
17FD	Compares r31 (08 H) and r29 (30 H) and updates status register --- A<B	1	0	1	1	0	0
30E8	Compares r30 (0F H) and rK (08 H) and updates status register --- A<B	0	0	0	0	1	0
07FD	Compares r31 (08 H) and r29 (30 H) and updates status register --- A<B	1	0	1	1	0	0
2FFE	Moves (Copies) r30 (0F H) to r31, r31 = (0F H)	1	0	1	1	0	0
15CB	Compares r28 (00 H) and r27 (00 H) and updates status register --- B=A --- Compares ZERO	0	1	0	0	0	0
F009	JUMPS TO PC + k + 1, if zero flag is set, PC = 23 + 1 + 1 = 25 --- 1111 00kk kkkk k001	0	0	0	0	0	0
CFEF	JUMPS To PC + k, PC = 25 + -19 + 1 = 07 --- 1100 kkkk kkkk kkkk	0	0	0	0	0	0
CFFF	JUMPS To PC + k, PC = 23 + -1 + 1 = 23 --- 1100 kkkk kkkk kkkk	0	0	0	0	0	0

Figure 20: Flag Updates

5. CONCLUSION

In conclusion, our project set out to enhance the capabilities of the WIMPAVR processor by integrating three new instructions—CP, CPI, and CPC—into its existing architecture. Through meticulous planning, rigorous testing, and innovative design, we successfully embedded these instructions within the ALU_TOP of the processor without compromising the functionality of the original instruction set.

Our modifications to the ALU control signals, the status register construction, and the arithmetic implementation were executed with precision, allowing us to present a robust system that performed compare operations effectively while maintaining the integrity of the data within the registers. We demonstrated the successful integration of our work through a comprehensive master test program, loaded onto the Altera DE2 board's onboard SRAM, which not only tested the new instructions but also ensured that all existing operations functioned as intended.

The project culminated in a live demonstration, where we confidently presented our results to our instructor and peers, showcasing our ability to extend the processor's functionality through thoughtful engineering and a deep understanding of the system's architecture. Our efforts not only reaffirmed the processor's operational capabilities but also set a precedent for future enhancements that could further expand its computational potential.