Christian Heimlich

CH09243 Microprocessor Data Sheet

16bit, 256K memory, Harvard Based, Memory Mapped

Group: Taking Recruits

Submission Date: May 11, 2015

16-Bit, Synthesizable, Microprocessor

Single-Cycle Harvard Design:

- Harvard Memory Mapped
- 39 Single-Word Instructions
- All instructions are Single-Cycle
- 16-Bit Wide Instructions
- 16-Bit Wide Datapath
- 7-Slot Hardware Stack
- 2 Input ALU
- Store up to 65,536 Instructions
- 1Mb (64k x 16) Instruction ROM
- 4Mb (256k x 16) Data RAM
- 8 Regular Data Registers
- 3 Special Function Hardware Registers
- Variable Operating Speed; 24MHz max; 44ns cycles
- MUX based pathing

Special Features:

- Built in clock divider
- Programmable to devices that support Verilog
- Capable of outputting Stack usage to UI
- Capable of outputting instruction errors to UI
- Fully Asynchronous Reset
- 2 Input SFU; Simplified Single-Cycle Optional Instructions
- Capable of displaying Datapath contents
- Capable of Accepting Interrupts

Peripheral Features:

- 16 I/O Pins
- Logic and I/O controlled Input, Output, and Direction Special Function Registers
- Data storage display
- 7-Segment encoded stack usage (uses 3 displays)
- 7-Segment encoded stack usage (uses 4 displays)
- Expandable UI Design

Implemented Optional Instructions:

- Arithmetic: Multiply, Add with Carry, Move B, Arithmetic Shift Right
- Arithmetic/Logic Literals: Add Immediate, Subtract Immediate, XOR Immediate
- Stack: PUSH, POP
- PC Instructions: Return PC, Call, Jump Immediate, Jump Register
- Bit-Oriented: Bit Set, Bit Clear, Bit Test Skip if Set, Bit Test Skip if Clear

Table of Contents:

List of Figures:

List of Tables:

1) ALU Functions
2) SFR Overview
3) Stack Range
4) Control Variable Sizes & Definitions
5) Non-Control Word Variable Size & Definitions
6) Opcode Field Descriptions
7) Instructions with Control Word
8) Instruction Set Summary
9) Example Stack Test Program

Architecture Overview:

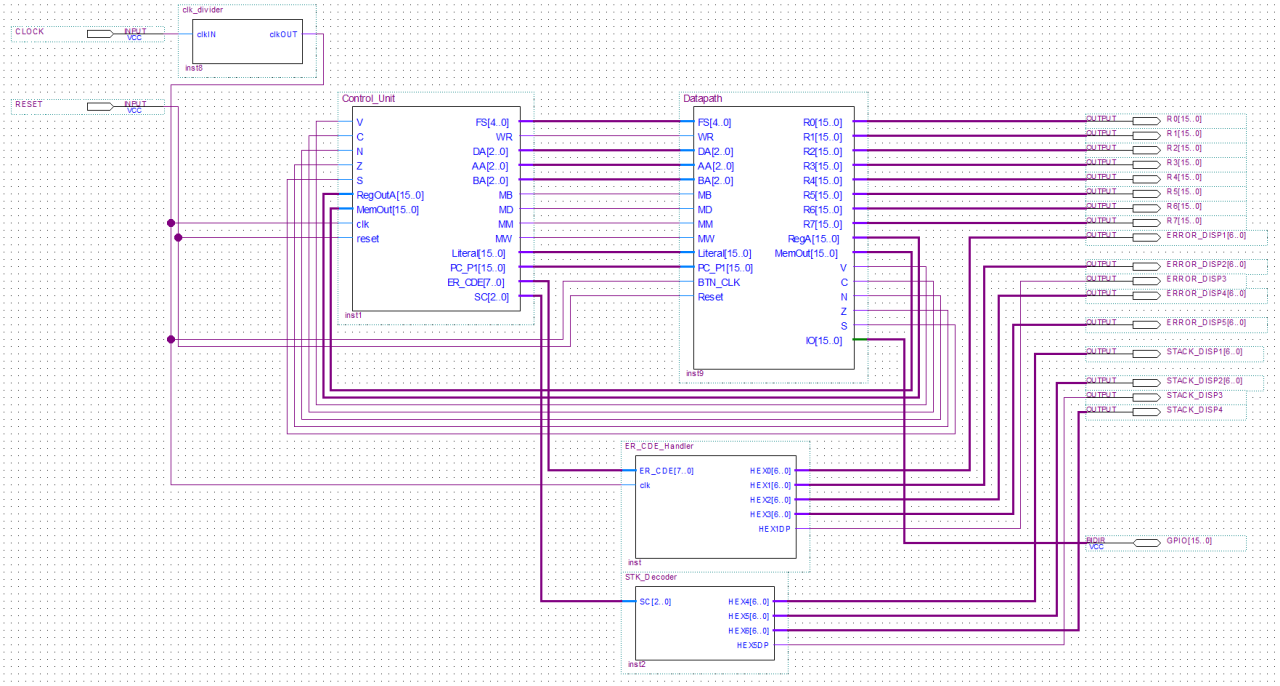

Figure 1: Simplified Block Diagram of Processor Design

Processor Description:

The CH09243 is a fully function single-cycle, Harvard based, Memory Mapped microprocessor primarily for use on an FPGA development board. The processor features a standard instruction decoder along with 1Mb ROM for feeding instructions to the processor. The instruction decoder allows the user to write programs entirely in the assembly language and the processor will interpret and execute the instructions fully on its own. The processor is limited to 65,536 preloaded instructions but is easily modifiable to support more instructions and real time instruction loading, and once the processor has completed its loaded instructions it will default to “No Operation”. Specifics of the processors instructions interpretation and execution are listed later in this document.

Some microprocessors in the same family have a limited instructions and features, such as no hardware stack, no advance arithmetic, and limited support for working with literals. This is not the case for the CH09243 which has full instruction set support, including single cycle multiply and advance program counter instructions, and a 7 slot hardware stack. Additionally the CH09243 is capable of outputting the current stack usage and an instruction specific error system for easier debugging.

As a Harvard architecture based processor the CH09243 consists primarily of a Control Unit and Datapath with additional minimalistic top-level components for adjusting clock speed, handling errors, and displaying information to a compatible user interface. The Control Unit and Datapath are made up of recognizable digital components and standard computer architecture units such as an ALU (Arithmetic Logic Unit), eight by 16-Bit register, etc. Additionally, the CH09243 contains proprietary components such as the SFU (Special Function Unit) for completing some advance operations. These parts and their operation are further described later in the data sheet.

This processor is 16-bit and therefore will experience overflow if values go above 65,536 or below -65,536. While it can work with negative numbers, it is an integer only processor. The CH09243 also has a clock speed limit of 24MHz and cannot function as a modern PC processor due to additional limitations, however; this processor is quite capable for more basic projects or machines.

Clocking Scheme:

The CH09243 does not have its own clock and is to be driven from an external/internal(virtually) signal depending on how it is installed. It contains an easily modifiable clock divider that is preset to run the processor at 24MHz assuming a 50MHz clock signal

8 x 16-bit Data Register:

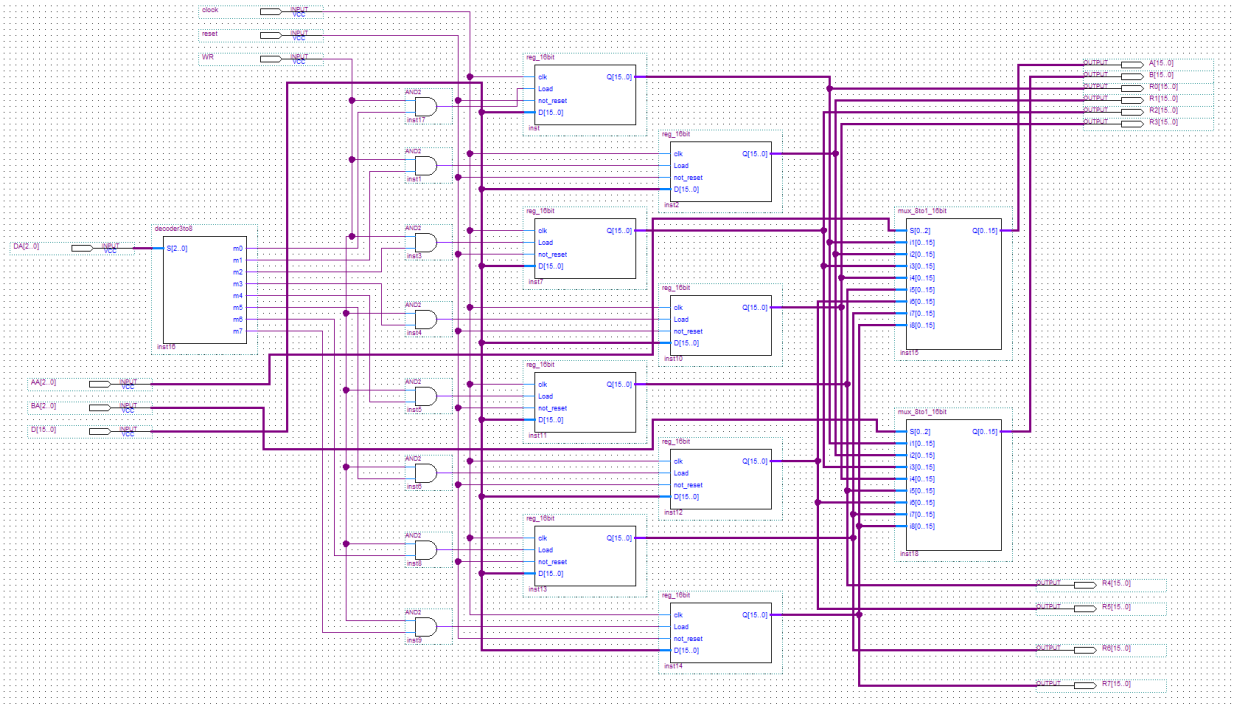

Figure 2: Internal design of the CH09243's 8 by 16bit register used in the Datapath

The CH09243's 16-bit registers are selected via a 3-8 decoder for writing and an 8 to 1 multiplexer for reading. Two multiplexers are used to allow reading from two registers at once or the same register twice for use in arithmetic or logical operations. Only one register can be written to per clock cycle. All eight registers are used for data, the stack and control unit components have their own counters and storage. Each 16-bit register is comprised of 16 D Flip-Flops and the appropriate wiring. Figure 3 shows an example of the register being simulated.

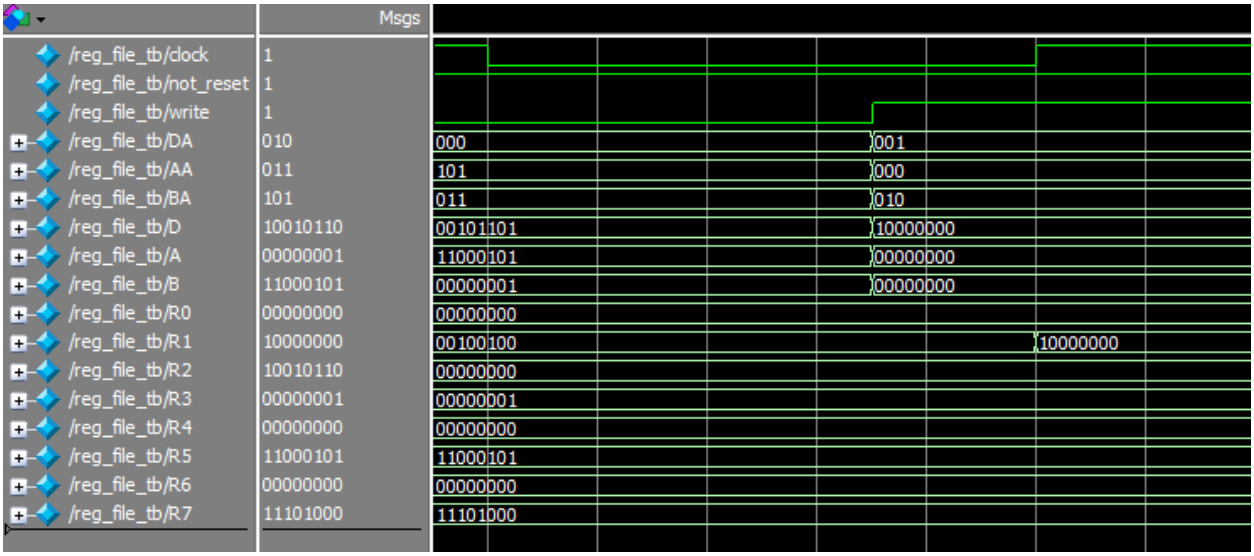

Figure 3: Simulation of the preceding 8x8bit register

The simulation showcases a write operation in which -128 is written to register one. The destination select (DA) is set to 1, write is set high, and the incoming data (D) is -128. At the next clock cycle after write is set high register one changes from its previous value of 36 to -128, which one can easily see is the intended result. The simulation is of an 8x8bit register, which is what the 16x8bit register was developed from, but the function is identical.

16-bit Arithmetic Logic Unit:

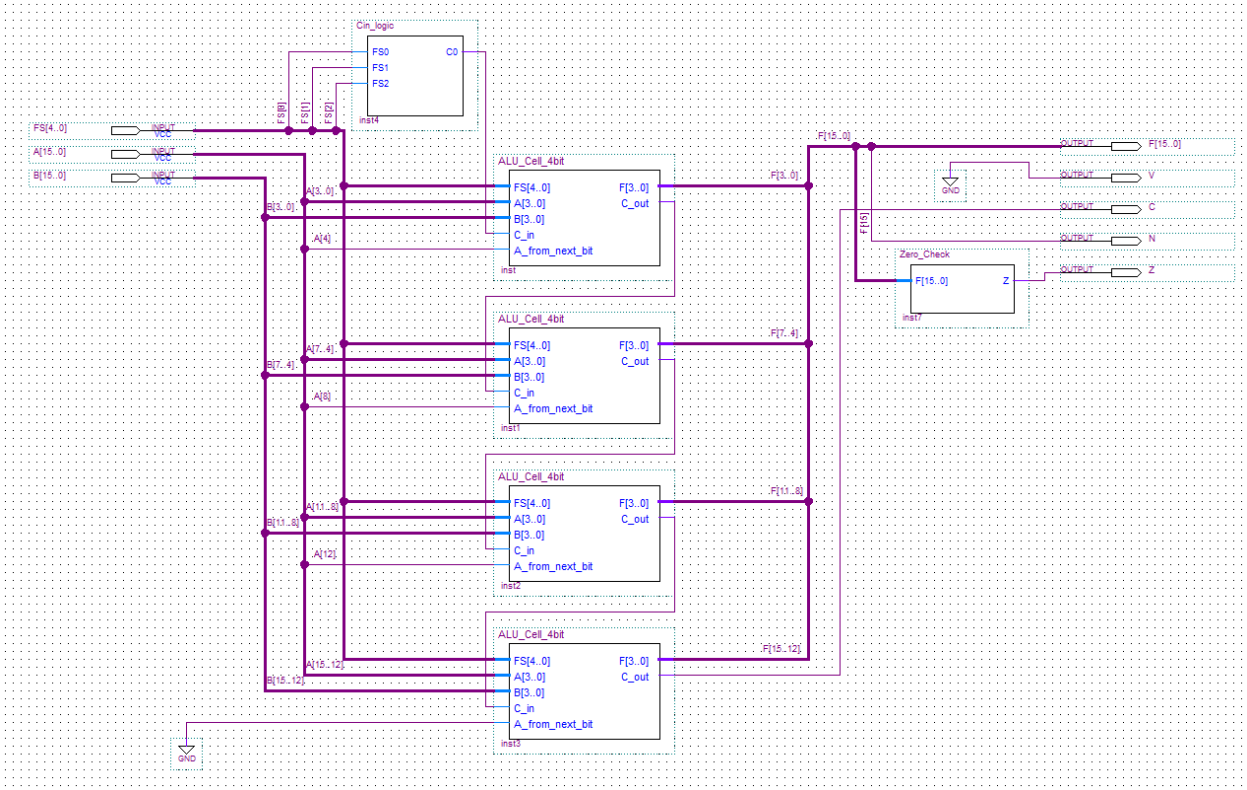

Figure 4: Internal design of the CH09243's Arithmetic Logic Unit used in the Datapath

The ALU accordingly computes any arithmetic or logic operations need by the processor for further operations, or user for a program. The chosen ALU design is an industry standard that was simply replicated and grouped to 4-bits and then 16-bits to work with 16-bit data. It works by receiving a “Function Select” (FS) signal and two 16-bit numbers and then computing the appropriate output based on the selected function. The ALU does this by utilizing a Full Adder and a series of MUX’s to manipulate bits for logic or complex arithmetic operations. The full size ALU is a series of 1-bit ALU cells that utilizes 16-bit busses. This can be seen in Figure 5 below:

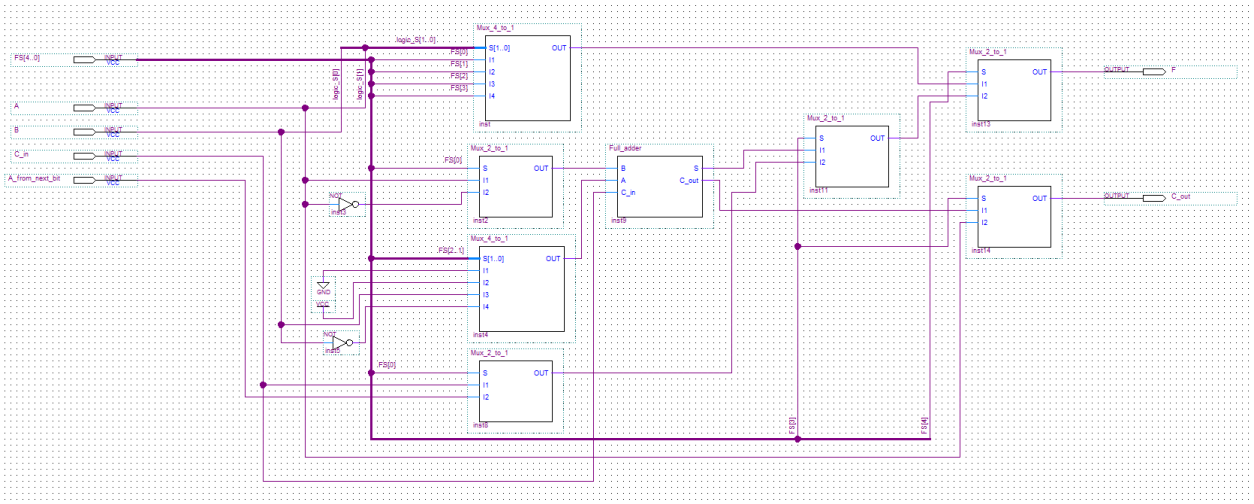

Figure 5: Internal design of a single ALU Cell

Note that while the ALU does a large majority of the processors calculations, it is not capable of advance operations such as arithmetic shifts or multiplications. Advanced computational functions are carried out by the Special Function Unit (SFU). Shown below is a simulation of the ALU:

Figure 6: Simulation of the ALU performing "Not A and Not B" with various B values

ALU development included testing all functions with a variety of values and checking for correct output. Note that in the full 16-bit ALU that there is no A_from_next_bit and that carry out (C_out) is used as the carry flag for the Control Unit which is described later in this document.

The following is a list of the functions the ALU is capable of along with their respect FS values and RTL results:

Table 1: ALU Functions

FS4	FS3	FS2	FS1	FS0	C0	Operation
0	0	0	0	0	x	0
0	0	0	0	1	x	notA and notB
0	0	0	1	0	x	notA and B
0	0	0	1	1	x	notA
0	0	1	0	0	x	A and notB
0	0	1	0	1	x	notB
0	0	1	1	0	x	A xor B
0	0	1	1	1	x	not AandB
0	1	0	0	0	x	A and B
0	1	0	0	1	x	not AxorB
0	1	0	1	0	x	B
0	1	0	1	1	x	notA or B
0	1	1	0	0	x	A
0	1	1	0	1	x	A or notB
0	1	1	1	0	x	A or B
0	1	1	1	1	x	1
1	0	0	0	0	1	A+1
1	0	0	0	1	1	negative A
1	0	0	1	0	0	A+1
1	0	0	1	1	1	A-1
1	0	1	0	0	0	A+B
1	0	1	0	1	1	B-A
1	0	1	1	0	1	A-B
1	0	1	1	1	0	negative (A+B)
1	1	x	x	0	x	shiftleft
1	1	x	x	1	x	Shiftright

16-bit Special Function Unit:

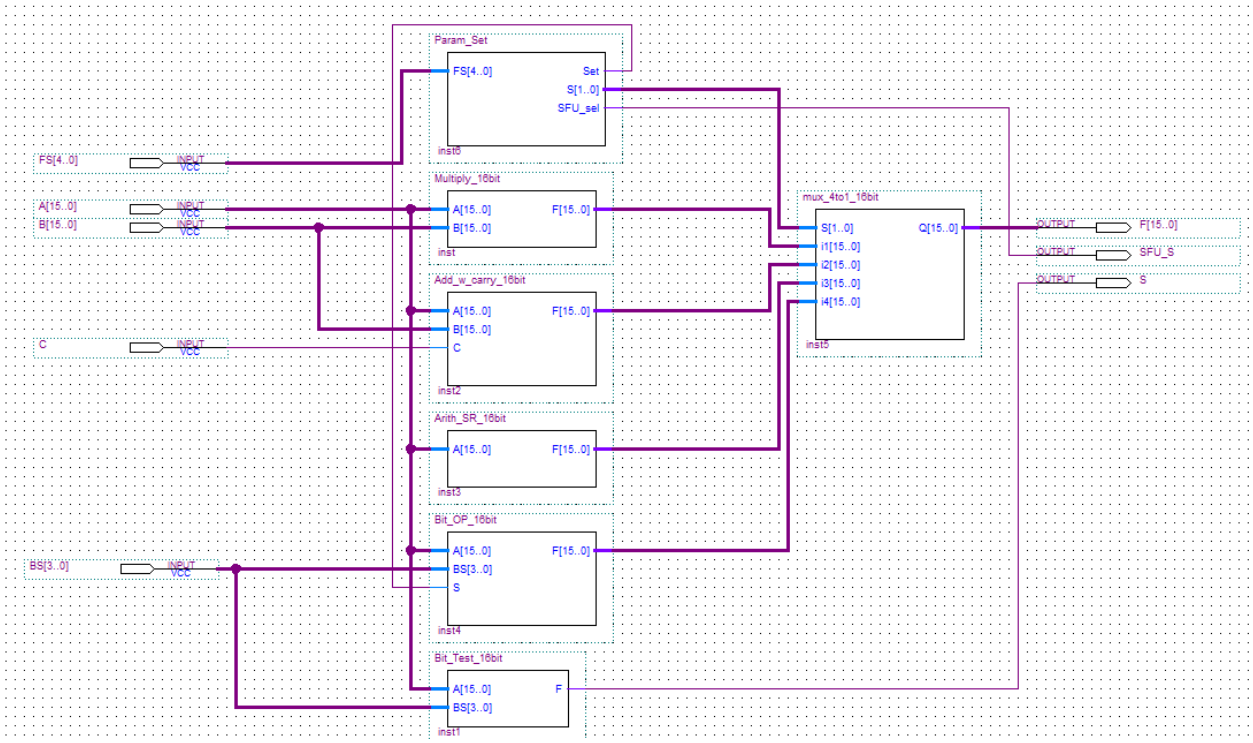

Figure 7: Internal design of the CH09243's Special Function Unit used in the Datapath (Code for each section of the SFU can be found in Appendix A.1 – A.6)

The SFU of the CH09243 run in parallel with the ALU and receives the same input data with the addition of a Bit Select (BS) signal and the carry output from the ALU itself. The SFU always computes its input data but its output is ignored unless an operation that requires it is being performed, in which case the data from the ALU is ignored. The SFU makes use of the remaining eight values of the 5-bit FS signal that the ALU does not use and is able to perform the following operations: Multiply, Add with Carry, Arithmetic Shift Right, Jump Immediate, Bitwise Clear. It also performs part of the Bitwise Test Skip Clear, and Bitwise Test Skip Set operations. The SFU was designed intuitively using a mix of Behavioral and Structural Verilog. Also, note that SFU was designed and tested entirely on a DE0 Nano development board so no simulation results are available for it.

4Mb (256K x 16-bit) Data RAM:

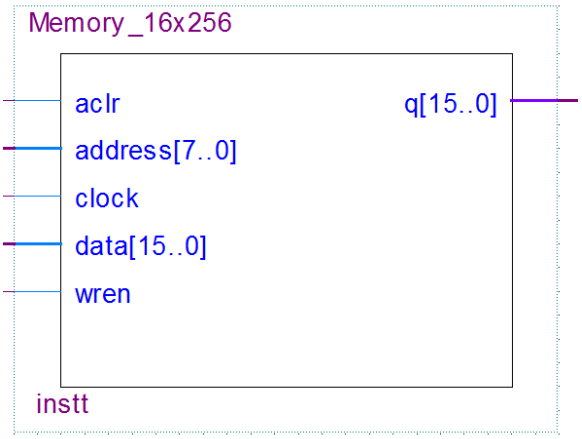

Figure 8: The CH09243 RAM Block used in the Datapath

The CH09243's RAM was developed using Altera's Megawizard which can quickly implement various common components that are usual for processor design. The RAM constantly outputs the selected address but its output is ignored unless memory operations are being performed. Additionally, it has a "Write Enable" input so that addresses are not overwritten accidently. Only one Address can be read from or written to per clock cycle. The RAM can only receive its address select from the Control Units literal output (only this is needed for all 39 instructions), but can receive its input data from the literal output, the Register's "B" output, or directly from the Program Counter (for Stack related instructions). Due to a Megawizard limitation, the RAM block accepts data on the negative edge of the clock in order to keep up with the system.

Addresses one, two, and three on RAM itself are unused because those addresses are reserved for the Special Function Registers (SFR). In addition, addresses 249 through 255 cannot be used directly because they are reserved for the 7 slot Hardware Stack that resides in the RAM itself. The stack works with a Stack Counter inside the Control Unit in order to operate.

The following is a view of the CH09243's RAM address space:

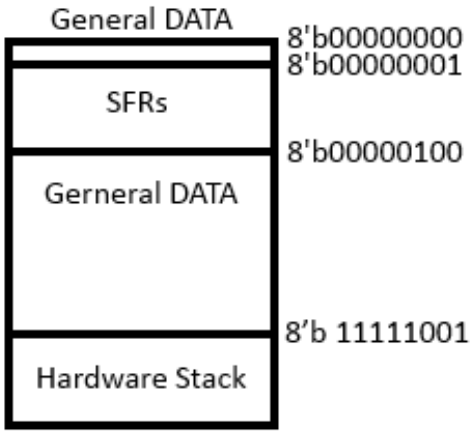

Figure 9: The CH09243's RAM Block address space with ranges

The following is a table of the SFRs contained in the CH09243. More information can be found in "GPIO Peripherals" on page 28:

Table 2: SFR Overview

SFR	Address	Usage
1	8'b 00000001	OUTPUT
2	8'b 00000002	DIRECTION SELECT
3	8'b 00000003	INPUT

The following is a table of the Stack location contained in the CH09243's RAM. More information can be found in "Control Unit" on page 24

Table 3: Stack Range

Slot	Address	Counter Value
(Empty Stack)	N/A	0
1	8'b 11111001	1
2	8'b 11111010	2
3	8'b 11111011	3
4	8'b 11111100	4
5	8'b 11111101	5
6	8'b 11111110	6
7	8'b 11111111	7

Datapath:

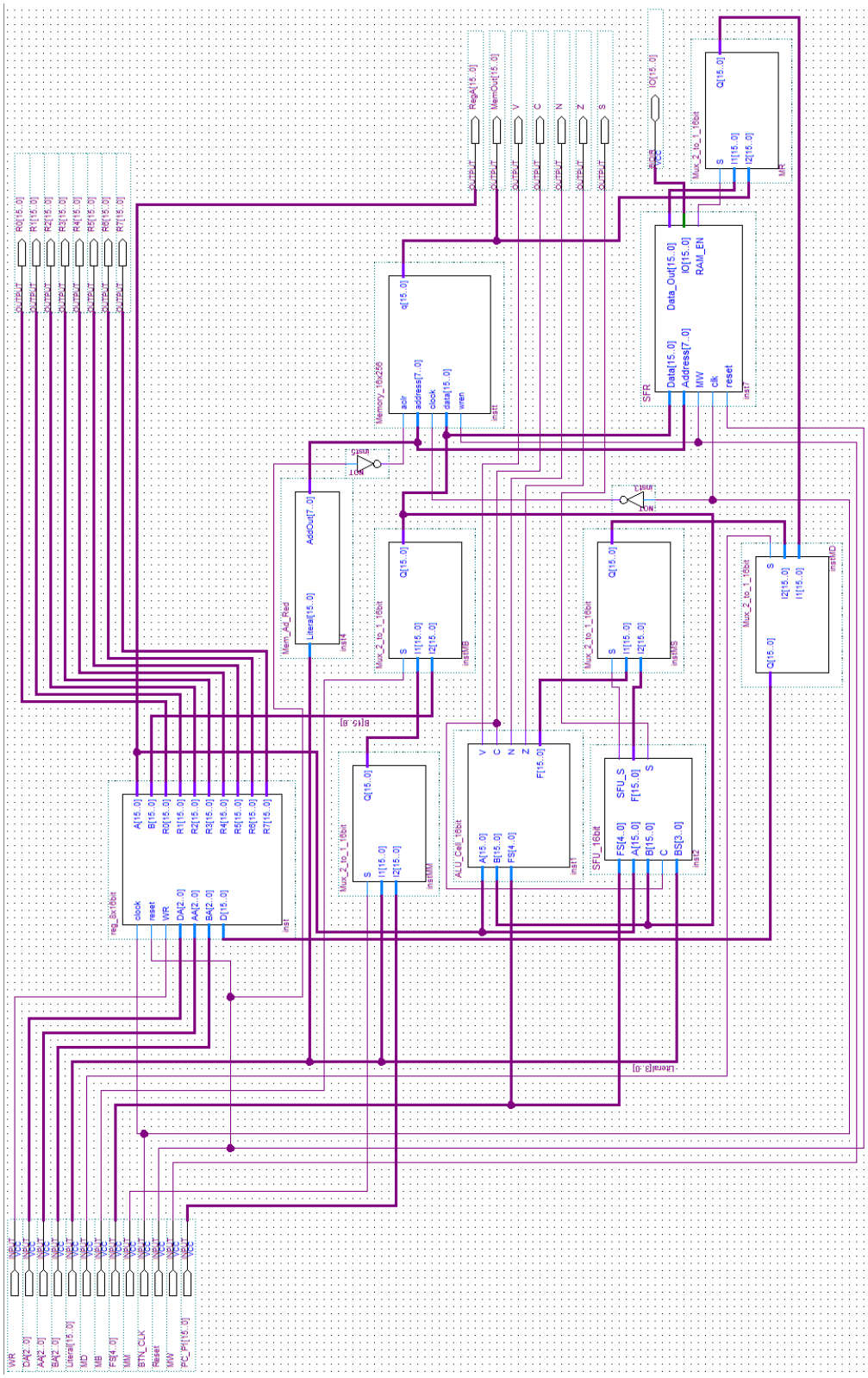

Figure 10: Internal Design of the CH09243's Datapath

The Datapath in the CH09243 manages the flow of all computational data throughout the processor. It receives the decoded Control Word from the Control Unit which affects MUX selects, write enables, FS, etc., which allows the data to essentially flow through the right components during the clock cycle to generate the correct output. The Datapath was designed with the desired Instruction Set always in-mind, and is, in the most simple definition, a connection of all computational components via appropriate combinational logic so that given the right Control Word the Datapath can complete any instruction. Other than Clock and Reset, the only additional input to the Datapath is PC+1 (the next state of the program counter) which is only used for one instruction and is only determined by the Program Counter. Therefore, it is not part of the Control Word, and because of this the Control Word is nearly responsible for all computation data changes that occur in the CH09243.

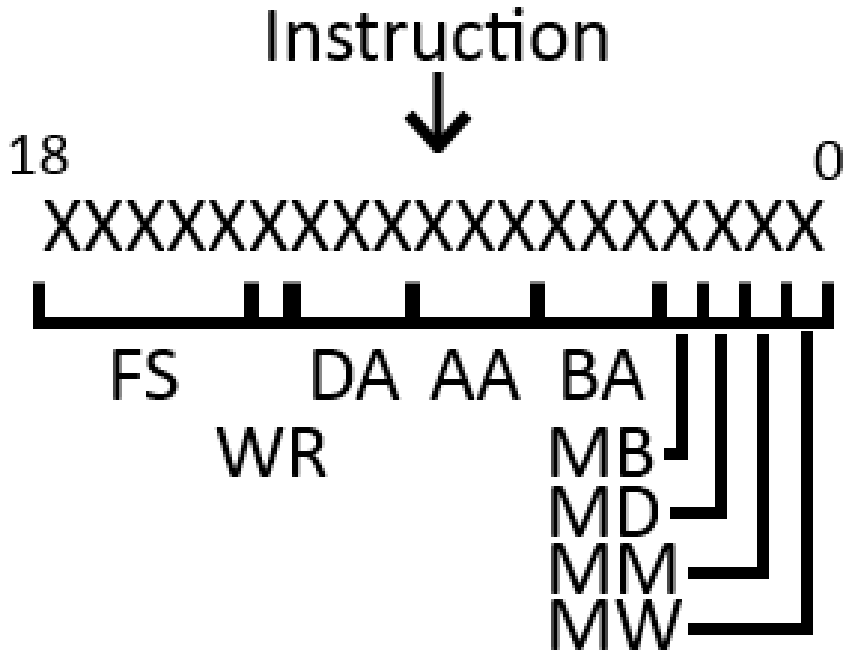

Figure 11: The CH09243's Control Word Format. Notice the convenient Control Variable Grouping

Figure 11 illustrates the Control Word format after being decoding in the Control Unit. The Control Word is 19-bits long and consists of nine different Control Variables that are sent to different component of the Datapath. Details of each are shown below in Table 4.

Table 4: Control Variable Sizes & Definitions

Variable	Size (bits)	Full Name	Function
FS	5	Function Select	Determines ALU function to perform
WR	1	Write Enable (REG)	Determines whether or not the register block is to be written to during a given cycle
DA	3	Destination Select	Selects which register will receive incoming data if WR is on during a given cycle
AA	3	REG Output A Select	Selects which register's data is placed on bus A
BA	3	REG Output B Select	Selects which register's data is placed on bus B
MB	1	MUX B	Selects where the RAMs input data is coming from; Low = MUX M output, High = Bus B from register block output
MD	1	MUX D	Selects where the register blocks input data is coming from; Low = RAM (or Stack/SFRs depending on address), High = Bus B from register output
MM	1	MUX M	Further selects the RAMs input data (only applicable if MUX B is low; Low = Literal from Control Unit, High = PC+1 from Control Unit
MW	1	Write Enable (RAM)	Determines whether or not RAM is to be written to during a given cycle

In addition to the control word/variables there are two major internally controlled signals that are shown below in Table 5.

Table 5: Non-Control Word Variable Size & Definitions

Variable	Size (bits)	Full Name	Function
MS	1	MUX S	Selects whether the ALU's or SFU's output data is used (sent to MD). The value of this select bit is determined FS so that the SFU is only being used for operations that require it; Low = ALU, High = SFU
MR	1	MUX R	Selects whether or not the RAM block or SFRs are used for memory operations. The value of this select bit is determined by the memory address select so that the SFRs are only used when their address range is selected. Low = SFR, High = RAM block

Control Unit:

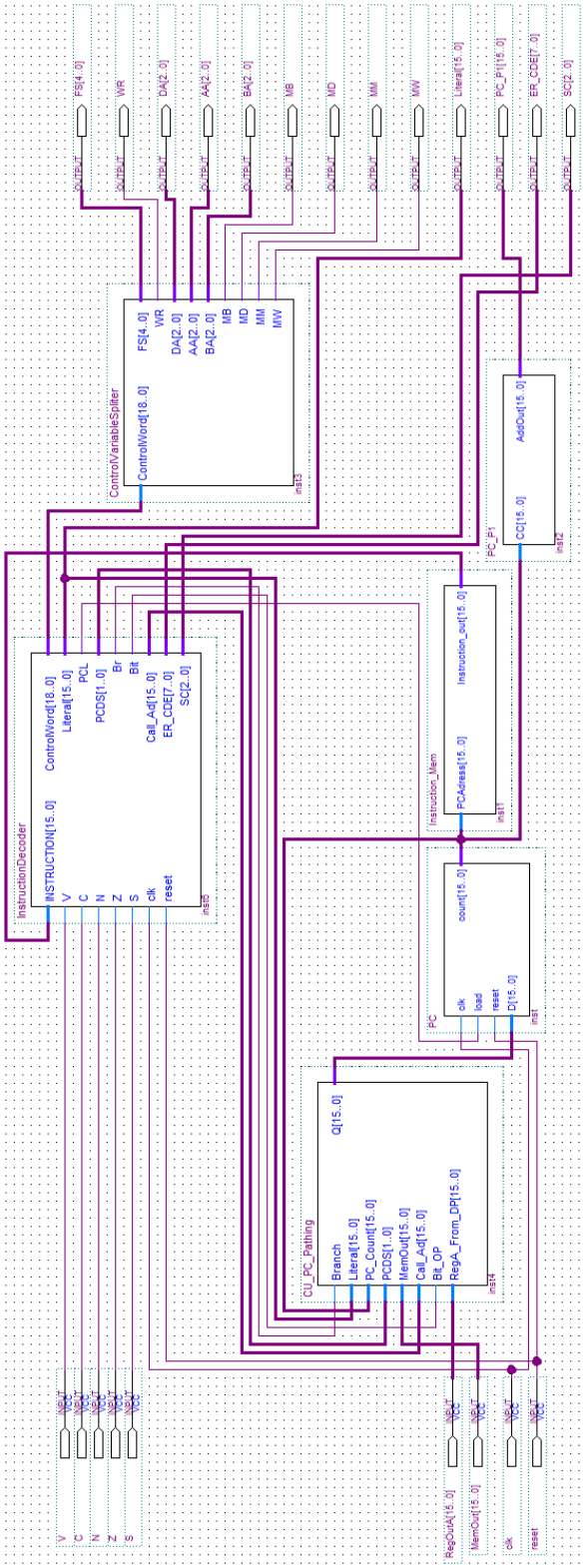

Figure 12: Internal design of the CH09243's Control Unit

The Control Unit of the CH09243 manages the pathing of information through the Datapath. Several parts work together by either calculating or keeping track of values relative to the Datapath and then feed them there through the appropriate data lines. While the Datapath was designed to simply pass data through the correct components so that calculations and outputs would be correct, the design of the Control Unit was much more complicated. The Datapath can be thought of as the human body, while the Control Unit can be thought of as the brain; a component that reduces complicated logic, states, and conditions down to a simple Control Word that can be processed by the Datapath.

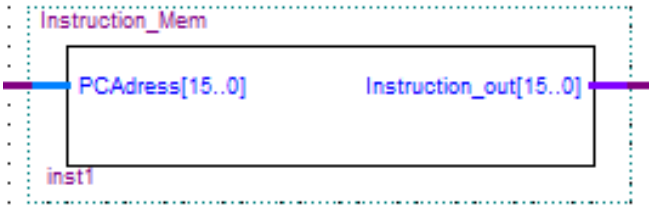

Figure 13: The Instruction Memory used in the Control Unit (Example code for the Instruction Memory can be found in Appendix A.8)

The Instruction Memory (shown above) contains the preloaded instructions that are executed by the processor once running. It is a simple ROM unit composed of case statements that runs down a list of instructions and outputs one at a time based on the current count of the Program Counter. While the unit itself can support a much higher amount of instructions, it is restricted to 65,536 instructions due to the 16-bit limitations of the Program Counter. When the Instruction Memory reaches the end of the instruction list it defaults to the “No Operation” instruction (shown in the Instruction Set on page 32).

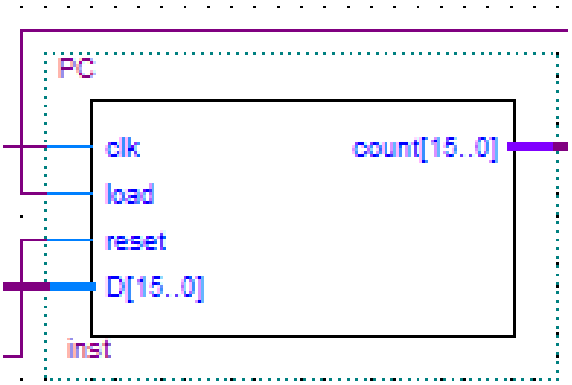

Figure 14: The Program Counter used in the Control Unit (Code for the Program Counter can be found in Appendix A.9)

The Program Counter is a simple 16-bit counter with parallel load and asynchronous reset that requires a few lines of behavioral Verilog. It simply increments the output “count” every clock cycle unless “load” is high, in which case it sets the count to whatever data is on “D” . If “reset” goes high the count is set to zero regardless of the clock input.

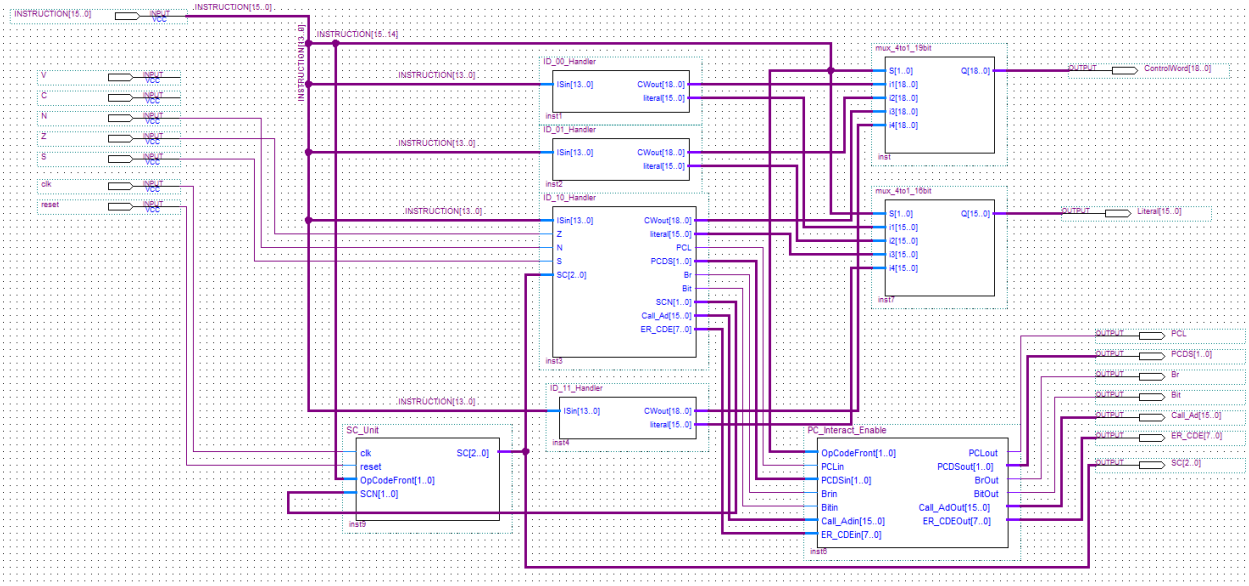

Figure 15: The internal design of the Instruction Decoder. (Example code for the "Handler" components of the decoder can be found in Appendix A.7)

The Instruction Decoder is the main component of the Control Unit as it determines the Control Word that is sent to the Datapath. The Instruction Decoder monitors various states of other CH09243 components such as the ALU status bits, or current Program Counter count, and generates the appropriate Control Word based on their state and the incoming Instruction. Some pieces of the instruction are mapped directly to the Control Word (i.e. Destination Select) since no changes need to be made to them, while other parts (i.e. MB, MD, etc.) are determined with a complex series of logical checks and calculations. Additionally, the “PC_Interact_Enable” block makes sure that no improper changes to the Program Counter will occur during the execution of instructions unrelated to it. The reason that the instruction interpretation is split into four methods is because it reduces the complexity of the case statements required for determining the Control Word. The decoder is also responsible for zero filing or sign extending any Literals, outputs some status bits for use elsewhere, and also keeps track of the hardware stack.

Another component of the Control Unit that has a simple responsibility but complex interworking is the “CU_PC_Pathing” block. It keeps track of numerous values across the whole CH09243 and outputs one counter value to the Program Counter so that the right count change is made any time a count load is needed.

Lastly, the “PC_P1” block simply outputs what the next Program Count will be for use in stack instructions, while the Control Variable Splitter just breaks up the Control Word into its components for easier use in the Datapath.

All of the components except the Instruction Decoder where codded in which one test could confirm proper function. Because of this the Instruction Decoder was the only unit to receive thorough testing through simulation. The results can be seen bellow in Figure 16.

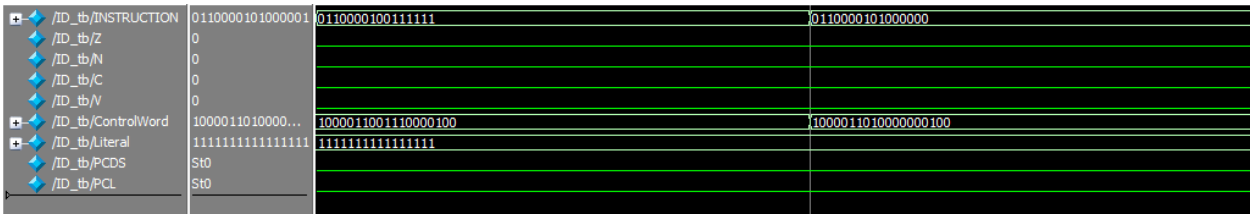

Figure 16: Sample result from the simulation of the Instruction Decoder

As seen above, the Instruction Decoder generates both the Control Word and makes any adjustments to the Literal that are needed. In the sample result the first instruction “Increment” is fed to the decoder and then the Control Word is set to the appropriate values such as an FS of 10000 for the ALU’s increment. The literal is set to all 1s when it is not used which is the case for Increment.

Complete CPU:

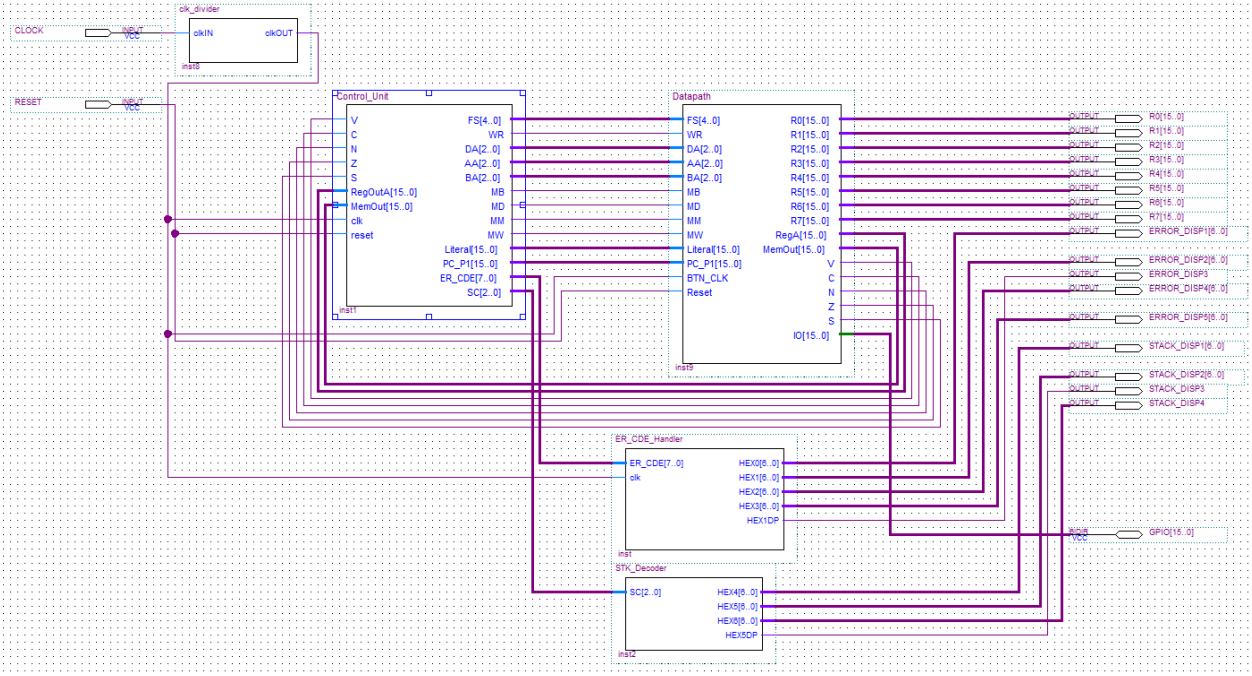

Figure 17: Assembly of all processor components into a functioning CPU

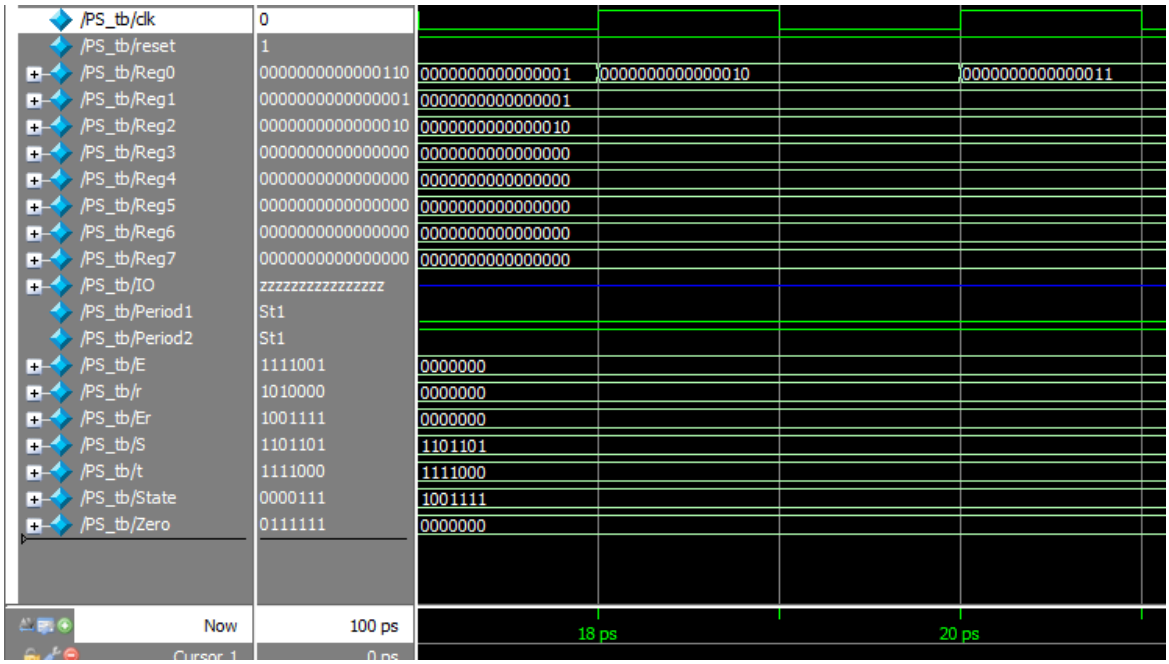

Figure 18: Result of incrementing Register 0 on the CPU

The complete CPU is simply the connection of the Control Unit and Datapath plus minor logic for the Error Handler and output of the hardware stack count.

Figure 14 shows the extensive testing done on the CPU after its assembly, and in particular shows register zero being incremented. Many sample programs were written in the Assembly language and simulating using the CPU test bench. A simple clock signal is the only required input as the CPU should handle each instruction every clock cycle and change each register as needed, as well as output the stack usage and any errors. Every single instruction with many possible variable choices (i.e. source register, destination register, etc.) was tested.

As mentioned previously the CPU will output the current stack usage and any errors it encounters. Assuming seven-segment displays are used, the stack usage is displayed as “St. X” where “X” is the highest slot used in the stack, and any error is displayed as “Er. YY” where YY is the error code. The stack usage info is always displayed, while error information is only displayed for one clock cycle during the cycle the error occurred. As of this revision the error codes are as follows:

- Error 00: No Error – Internally determined but not actually outputted
- Error 01: User tried to load from memory at an address that was reserved for the hardware stack
- Error 02: User tried to write to memory at an address that was reserved for the hardware stack
- Error 03: (Stack Overflow) User tried to PUSH to the stack while it was full
- Error 04: User tried to POP from the stack while it was empty
- Error 05: The CPU attempted to answer a CALL, but could not record the next count because the stack was full
- Error 06: The CPU attempted to return to a recorded address, but failed because the stack was empty

GPIO:

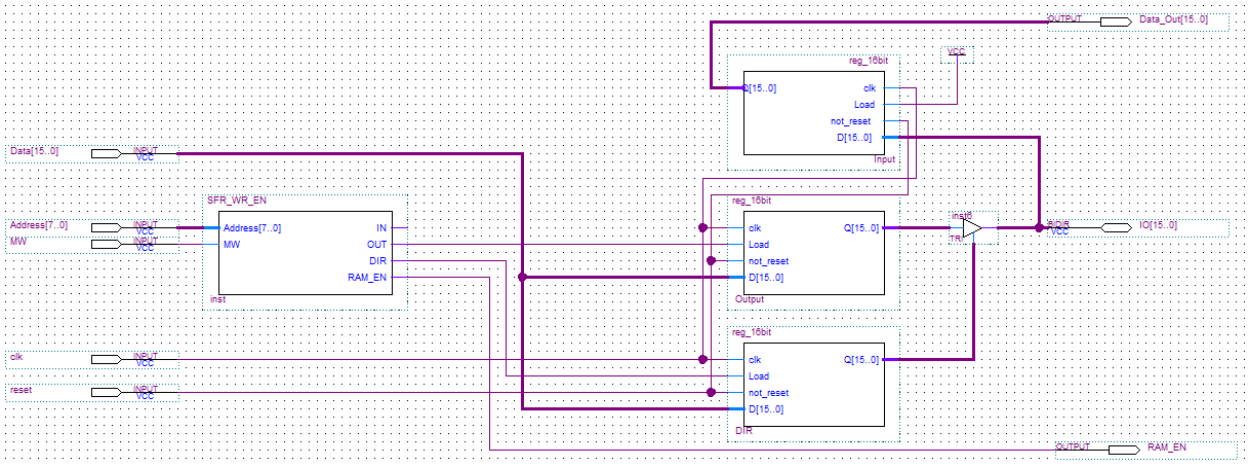

Figure 19: The GPIO interface of the CH09243

As shown in Figure 19, the CH09243 has three memory-mapped Special Function Registers (SFRs) responsible for interfacing with the GPIO. The memory mapping allows the SFRs to be controlled via memory instructions on address 1 through 3. The DIR register decides whether the 16-bit GPIO will be an input or output and can change each bit independently. The Input register send any bits acting as input into the data line of the Datapath and to the 16-bit data registers, while the Output register simply outputs any bits acting as output.

Instruction Set:

Overview:

Table 6: Opcode Field Descriptions

Field	Description
Mnemonic	2 to 4 letter abbreviation of an instruction
DR	3-bit Destination register address
SA	3-bit Source register A address
SB	3-bit Source register A address
Literal	User defined constant
←	Assigned to
b	4-bit Bit operation selection
zf	Zero filled
se	Sign extended
R[x]	Register x
XXX	Field not used for instruction

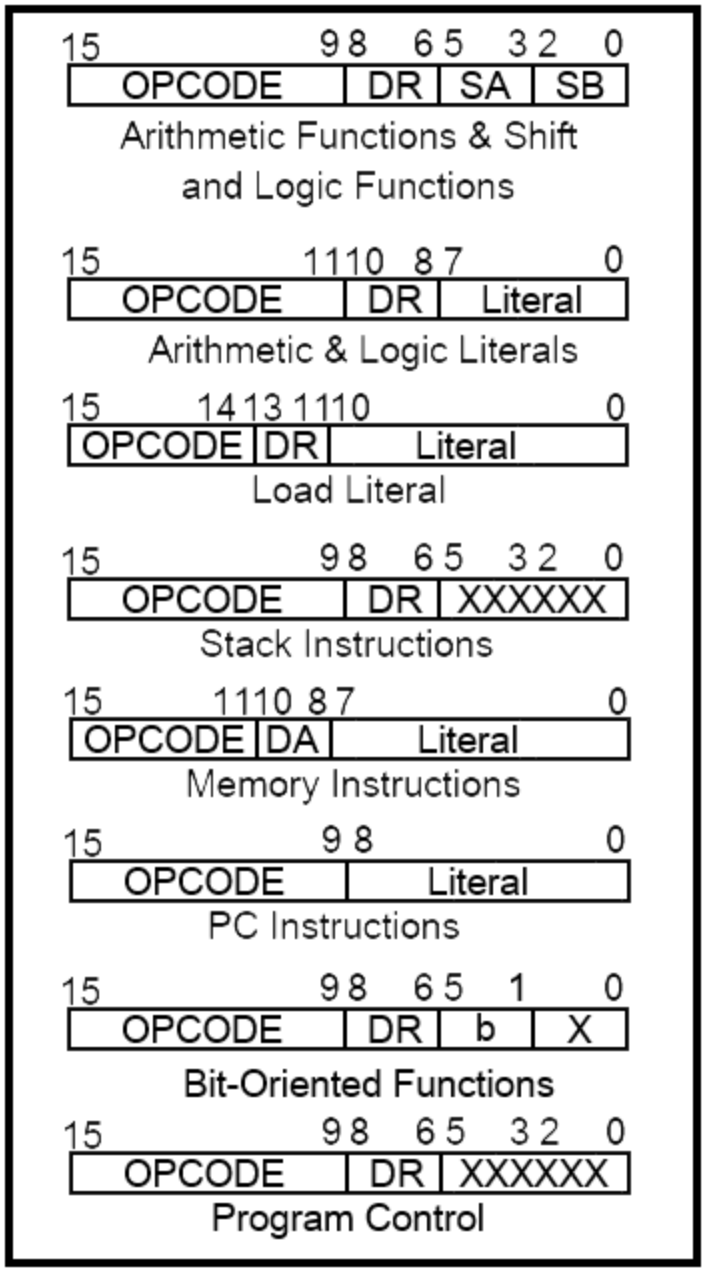

Figure 20: Figure of Instruction Formats

Table 7: Instructions with Control Word

Operation	Op Code	FS	WR	DA	AA	BA	MB	MD	MM	MW	Literal (16 bit)
No Operation	00|00000	xxxxx	0	xxx	xxx	xxx	x	x	x	0	xxxxxxxxxxxxxxxx
Increment	01|10000	10000	1	DA	AA	xxx	x	1	x	0	xxxxxxxxxxxxxxxx
Add	01|10100	10100	1	DA	AA	BA	1	1	x	0	xxxxxxxxxxxxxxxx
Subtract	01|10110	10110	1	DA	AA	BA	1	1	x	0	xxxxxxxxxxxxxxxx
Decrement	01|10010	10011	1	DA	AA	xxx	x	1	x	0	xxxxxxxxxxxxxxxx
Negative	01|10001	10001	1	DA	AA	xxx	x	1	x	0	xxxxxxxxxxxxxxxx
Move A	01|01100	01100	1	DA	AA	xxx	x	1	x	0	xxxxxxxxxxxxxxxx
Move B*	01|01010	01010	1	DA	xxx	BA	1	1	x	0	xxxxxxxxxxxxxxxx
Shift Right	01|11001	11001	1	DA	AA	xxx	x	1	x	0	xxxxxxxxxxxxxxxx
Shift Left	01|11000	11000	1	DA	AA	xxx	x	1	x	0	xxxxxxxxxxxxxxxx
Clear	01|00000	00000	1	DA	AA = DA	BA = DA	x	1	x	0	xxxxxxxxxxxxxxxx
Set	01|01111	01111	1	DA	AA = DA	BA = DA	x	1	x	0	xxxxxxxxxxxxxxxx
NOT	01|00011	00011	1	DA	AA	xxx	x	1	x	0	xxxxxxxxxxxxxxxx
AND	01|01000	01000	1	DA	AA	BA	1	1	x	0	xxxxxxxxxxxxxxxx
OR	01|01110	01110	1	DA	AA	BA	1	1	x	0	xxxxxxxxxxxxxxxx
XOR	01|00110	00110	1	DA	AA	BA	1	1	x	0	xxxxxxxxxxxxxxxx
Add Immediate*	00|001	10100	1	DA	AA	xxx	0	1	x	0	Literal (16 bit)
Subtract Immediate*	00|010	10110	1	DA	AA	xxx	0	1	x	0	Literal (16 bit)
AND Immediate	00|011	01000	1	DA	AA	xxx	0	1	x	0	Literal (16 bit)
OR Immediate	00|101	01110	1	DA	AA	xxx	0	1	x	0	Literal (16 bit)
XOR Immediate*	00|110	00110	1	DA	AA	xxx	0	1	x	0	Literal (16 bit)
Load Immediate	11|	01010	1	DA	xxx	xxx	0	1	x	0	Literal (16 bit)
Load Memory	10|100	xxxxx	1	DA	xxx	xxx	x	0	MM	0	Literal (16 bit)
Store Memory	10|101	xxxxx	0	xxx	xxx	BA	1	x	MM	1	Literal (16 bit)
Jump Register	10|01101	xxxxx	0	xxx	AA	xxx	x	x	x	0	xxxxxxxxxxxxxxxx
Branch if Zero	10|110	01100	1	DA = AA	AA	xxx	x	1	x	0	Literal (16 bit)
Branch if Negative	10|111	01100	1	DA = AA	AA	xxx	x	1	x	0	Literal (16 bit)
Multiply*	01|10111	11010	1	DA	AA	BA	1	1	x	0	xxxxxxxxxxxxxxxx
Add With Carry*	01|10101	11011	1	DA	AA	BA	1	1	x	0	xxxxxxxxxxxxxxxx
Arithmetic Shift Right *	01|11011	11100	1	DA	AA	xxx	x	1	x	0	xxxxxxxxxxxxxxxx
Jump Immediate*	10|01100	xxxxx	0	xxx	xxx	xxx	x	x	x	0	Literal (16 bit)
Bitwise Clear*	10|01000	11101	1	DA	AA = DA	BA = DA	x	1	x	0	xxxxxxxxxxxxLiteral (4 bit)
Bitwise Set*	10|01001	11110	1	DA	AA = DA	BA = DA	x	1	x	0	xxxxxxxxxxxxLiteral (4 bit)
Bitwise Test Skip Clear*	10|01011	11111	0	DA	AA = DA	xxx	x	1	x	0	xxxxxxxxxxxxLiteral (4 bit)
Bitwise Test Skip Set*	10|01010	11111	0	DA	AA = DA	xxx	x	1	x	0	xxxxxxxxxxxxLiteral (4 bit)

Table 8: Instruction Set Summary

INSTRUCTION SET – Computer Architecture Lab – Spring 2015							
Unless a modification to PC is specified then PC <-- PC + 1 is implied							
Description	**Mnemonic**	**Opt.**	**Opcode**	**Field 1**	**Field 2**	**Field 3**	**RTL**
			7-bit opcode	**3-bit Addr.**	**3-bit Addr.**	**3-bit Addr.**	
No Operation	NOP		0000000	XXX	XXX	XXX	No Action
Arithmetic Functions & Shift							
Increment	INC		0110000	DR	SA	XXX	R[DR] ← R[SA] + 1
Add	ADD		0110100	DR	SA	SB	R[DR] ← R[SA] + R[SB]
Subtract	SUB		0110110	DR	SA	SB	R[DR] ← R[SA] - R[SB]
Decrement	DEC		0110010	DR	SA	XXX	R[DR] ← R[SA] - 1
Multiply	MUL	X	0110111	DR	SA	SB	R[DR] ← R[SA] * R[SB]
Add w/ Carry	ADDC	X	0110101	DR	SA	SB	R[DR] ← R[SA] + R[SB] + C
Negative	NEG		0110001	DR	SA	XXX	R[DR] ← ~R[SA] + 1
Move A	MOVA		0101100	DR	SA	XXX	R[DR] ← R[SA]
Move B	MOVB	X	0101010	DR	XXX	SB	R[DR] ← R[SB]
Shift Right	SHR		0111001	DR	SA	XXX	R[DR] ← sr R[SA]
Shift Left	SHL		0111000	DR	SA	XXX	R[DR] ← sl R[SA]
Arithmetic Shift Right	ASHR	X	0111011	DR	SA	XXX	R[DR] ← arithmetic sr R[SA] (sign extend)
Logic Functions			**7-bit opcode**	**3-bit Addr.**	**3-bit Addr.**	**3-bit Addr.**	
Clear	CLR		0100000	DR	XXX	XXX	R[DR] ← 0
Set	SET		0101111	DR	XXX	XXX	R[DR] ← FFFF
NOT	NOT		0100011	DR	SA	XXX	R[DR] ← ~R[SA]
AND	AND		0101000	DR	SA	SB	R[DR] ← R[SA] ^ R[SB]
OR	OR		0101110	DR	SA	SB	R[DR] ← R[SA] v R[SB]
XOR	XOR		0100110	DR	SA	SB	R[DR] ← R[SA] XOR R[SB]
Arithmetic & Logic Literals			**5-bit opcode**	**3-bit Addr.**	**8 -bit Field**		**[se -- sign extend] [zf -- zero fill]**
Add Immediate	ADDI	X	00001	DR (SA = DR)	Literal		R[DR] ← R[SA] + zf-lit
Subtract Immediate	SUBI	X	00010	DR (SA = DR)	Literal		R[DR] ← R[SA] - zf-lit
AND Immediate	ANDI		00011	DR (SA = DR)	Literal		R[DR] ← R[SA] ^ zf-lit
OR Immediate	ORI		00101	DR (SA = DR)	Literal		R[DR] ← R[SA] v zf-lit
XOR Immediate	XORI	X	00110	DR (SA = DR)	Literal		R[DR] ← R[SA] XOR zf-lit
Load Literal			**2-bit opcode**	**3-bit Addr.**	**11-bit literal**		
Load Immediate	LDI		11	DR	literal		R[DR] ← se lit
Stack Instructions			**7-bit opcode**	**3-bit Addr.**	**3-bit Addr.**	**3-bit Addr.**	
PUSH	PUSH	X	1000000	SA	XXX	XXX	PUSH R[SA]
POP	POP	X	1000001	DR	XXX	XXX	R[DR] ← POP
Memory Instructions			**5-bit opcode**	**3-bit Addr.**	**8-bit address**		
Load Memory	LDM		10100	DR	Address		R[DR] ← M[AD]
Store Memory	STM		10101	SA	Address		M[AD] ← R[SA]
PC Instructions			**7-bit opcode**	**9-bit Addr.**			
Return PC	RET	X	1001111	XXXXXXXXX			PC ← POP
Call	CALL	X	1001110	Address (AD)			PUSH PC+1, PC ← zf AD
Jump Immediate	JMPI	X	1001100	Address			PC ← zf AD

				(AD)			
				3-bit Addr.			
Jump Register	JMPR		1001101	SB	XXX	XXX	PC ← R[SB]
Bit-Oriented Functions			**7-bit opcode**	**3-bit Addr.**	**4-bit bit field index**		
Bit Set	BSET	X	1001001	DR	b		R[DR].b = 1
Bit Clear	BCLR	X	1001000	DR	b		R[DR].b = 0
Bit Test Skip If Set	BTSC	X	1001010	SA	b		if(R[SA].b == 1) PC ← PC + 2; else PC ←PC + 1
Bit Test Skip If Clear	BTSS	X	1001011	SA	b		if(R[SA].b == 0) PC ← PC + 2; else PC ← PC + 1
Program Control			**5-bit opcode**	**3-bit Addr.**	**8-bit address (AD)**		
Branch If Zero	BRZ		10110	SA	Signed Address Offset		if(R[SA]==0) PC ←PC + se AD
Branch If Negative	BRN		10111	SA	Signed Address Offset		if(R[SA]<0) PC ← PC + se AD
Instructions below are required for non-memory mapped I/O							
			7-bit opcode	**3-bit Addr.**	**3-bit 3-bit Addr.**	**3-bit Addr.**	
Read GPIO In	IN		1000010	DR	XXX	XXX	R[DR] ← IN
Write GPIO Out	OUT		1000011	SA	XXX	XXX	OUT ← R[SA]
Write GPIO Dir	DIR		1000100	SA	XXX	XXX	DIR ← R[SA]

Detailed Instruction Set:

NOP — No Operation

Syntax : 0000000XXXXXXXXX

Operands: XXX

Operation: No Action

Status Affected: None

Description: The processor does nothing

INC — Increment

Syntax : {0110000,DR[2:0],SA[2:0],XXX}

Operands: DR, SA – 0 to 7

Operation: R[DR] ← R[SA] + 1

Status Affected: V, C, Z

Description: The selected register is incremented by 1 and is placed in the same or another selected register

ADD — Add

Syntax : {0110100,DR[2:0],SA[2:0],SB[2:0]}

Operands: DR, SA, SB – 0 to 7

Operation: R[DR] ← R[SA] + R[SB]

Status Affected: V, C, N, Z

Description: The two selected registers are added together and the result is placed in the same or another selected register

SUB — Subtract

Syntax : {0110110,DR[2:0],SA[2:0],SB[2:0]}

Operands: DR, SA, SB – 0 to 7

Operation: R[DR] ← R[SA] - R[SB]

Status Affected: V, C, N, Z

Description: Register A is subtracted from Register B and the result is placed in the same or another selected register

DEC — Decrement

Syntax : {0110010,DR[2:0],SA[2:0],XXX}

Operands: DR, SA – 0 to 7

Operation: R[DR] ← R[SA] - 1

Status Affected: V, C, N, Z

Description: The selected register is decremented by 1 and is placed in the same or another selected register

MUL — Multiply

Syntax : {0110111,DR[2:0],SA[2:0],SB[2:0]}

Operands: DR, SA, SB – 0 to 7

Operation: R[DR] ← R[SA] * R[SB]

Status Affected: V, C, N, Z

Description: The two selected registers are multiplied together and the result is placed in the same or another selected register

ADDC Add w/ Carry

Syntax : {0110101,DR[2:0],SA[2:0],SB[2:0]}

Operands: DR, SA, SB – 0 to 7

Operation: R[DR] ← R[SA] + R[SB] + 1

Status Affected: All

Description: The two selected registers are added together along with the carry bit (if high) and the result is placed in the same or another selected register

NEG Negative

Syntax : {0110001,DR[2:0],SA[2:0],XXX}

Operands: DR, SA – 0 to 7

Operation: R[DR] ← ~R[SA] + 1

Status Affected: All

Description: The selected register is made negative and is placed in the same or another selected register

MOVA MOVE A

Syntax : {0101100,DR[2:0],SA[2:0],XXX}

Operands: DR, SA – 0 to 7

Operation: R[DR] ← R[SA]

Status Affected: All

Description: The selected register is copied and moved to the same or another selected register using output A

MOVB MOVE B

Syntax : {0101101,DR[2:0],XXX,SB[2:0]}

Operands: DR, SB – 0 to 7

Operation: R[DR] ← R[SB]

Status Affected: All

Description: The selected register is copied and moved to the same or another selected register using output B

SHR Shift Right

Syntax : {0111001,DR[2:0],SA[2:0],XXX}

Operands: DR, SA – 0 to 7

Operation: R[DR] ← sr R[SA]

Status Affected: All

Description: The selected register is shifted right and moved to the same or another selected register using output A

SHL Shift Left

Syntax : {0111000,DR[2:0],SA[2:0],XXX}

Operands: DR, SA – 0 to 7

Operation: R[DR] ← sl R[SA]

Status Affected: All

Description: The selected register is shifted left and moved to the same or another selected register using output A

ASHR Arithmetic Shift Right

Syntax : {0111011,DR[2:0],SA[2:0]XXX}

Operands: DR, SA – 0 to 7

Operation: R[DR] ← arithmetic sr R[SA]

(sign extend)

Status Affected: All

Description: The selected register is set to zero

CLR Clear

Syntax : {0100000,DR[2:0],XXXXXX}

Operands: DR – 0 to 7

Operation: R[DR] ← 0

Status Affected: All

Description: The selected register is set to zero

SET Set

Syntax : {0101111,DR[2:0],XXXXXX}

Operands: DR – 0 to 7

Operation: R[DR] ← FFFF

Status Affected: All

Description: The selected register is set to 65,536

NOT NOT

Syntax : {0100011,DR[2:0],SA[2:0],XXX}

Operands: DR, SA – 0 to 7

Operation: R[DR] ← ~R[SA]

Status Affected: All

Description: The selected register is set to NOT itself or another selected register

AND AND

Syntax : {0101000,DR[2:0],SA[2:0],SB[2:0}}

Operands: DR, SA, SB – 0 to 7

Operation: R[DR] ← R[SA] ^ R[SB

Status Affected: All

Description: The selected two registered are ANDed together and the result is placed in the selected register

OR OR

Syntax : {0101110,DR[2:0],SA[2:0],SB[2:0}}

Operands: DR, SA, SB – 0 to 7

Operation: R[DR] ← R[SA] v R[SB

Status Affected: All

Description: The selected two registered are ORed together and the result is placed in the selected register

XOR XOR

Syntax : {0100110,DR[2:0],SA[2:0],SB[2:0}}

Operands: DR, SA, SB – 0 to 7

Operation: R[DR] ← R[SA] XOR R[SB]

Status Affected: All

Description: The selected two registered are XORed together and the result is placed in the selected register

ADDI Add Immediate

Syntax : {00001,DR[2:0],Literal[7:0]}

Operands: DR – 0 to 7, Literal – 8-bit

Operation: R[DR] ← R[SA] + zf-lit

Status Affected: All

Description: The selected register is added to the literal and the result is placed in the same or another selected register

SUBI Subtract Immediate

Syntax : {00010,DR[2:0],Literal[7:0]}

Operands: DR – 0 to 7, Literal – 8-bit

Operation: R[DR] ← R[SA] - zf-lit

Status Affected: All

Description: The literal is subtracted from the selected register and the result is placed in the same or another selected register

ADDI Add Immediate

Syntax : {00001,DR[2:0],Literal[7:0]}

Operands: DR – 0 to 7, Literal – 8-bit

Operation: R[DR] ← R[SA] + zf-lit

Status Affected: All

Description: The selected register is added to the literal and the result is placed in the same or another selected register

ANDI AND Immediate

Syntax : {00011,DR[2:0],Literal[7:0]}

Operands: DR – 0 to 7, Literal – 8-bit

Operation: R[DR] ← R[SA] ^ zf-lit

Status Affected: All

Description: The selected register is ANDed with the literal and the result is placed in the same or another selected register

ORI OR Immediate

Syntax : {00101,DR[2:0],Literal[7:0]}

Operands: DR – 0 to 7, Literal – 8-bit

Operation: R[DR] ← R[SA] v zf-lit

Status Affected: All

Description: The selected register is ORed with the literal and the result is placed in the same or another selected register

XORI XOR Immediate

Syntax : {00110,DR[2:0],Literal[7:0]}

Operands: DR – 0 to 7, Literal – 8-bit

Operation: R[DR] ← R[SA] XOR zf-lit

Status Affected: All

Description: The selected register is XORed with the literal and the result is placed in the same or another selected register

LDI Load Immediate

Syntax : {11,DR[2:0],Literal[10:0]}

Operands: DR – 0 to 7, Literal – 11-bit

Operation: R[DR] ← se lit

Status Affected: All

Description: The selected register is replaced with the literal

PUSH PUSH

Syntax : {1000000,SA[2:0],XXXXXX}

Operands: SA – 0 to 7

Operation: PUSH R[SA]

Status Affected: None

Description: The selected register is pushed into the stack

PUSH PUSH

Syntax : {1000001,DR[2:0],XXXXXX}

Operands: DR – 0 to 7

Operation: R[DR] ← POP

Status Affected: None

Description: The top of the stack is POPed onto the selected register

LDM Load Memory

Syntax : {10100,DR[2:0],Literal[7:0]}

Operands: DR – 0 to 7, Literal – 8-bit

Operation: R[DR] ← M[AD]

Status Affected: All

Description: The selected register is replaced by what is in RAM at the selected address

STM Store Memory

Syntax : {10101,SA[2:0],Literal[7:0]}

Operands: SA – 0 to 7, Literal – 8-bit

Operation: M[AD] ← R[SA]

Status Affected: None

Description: The selected register is saved to RAM at the selected address

LDM Load Memory

Syntax : {10100,DR[2:0],Literal[7:0]}

Operands: DR – 0 to 7, Literal – 8-bit

Operation: R[DR] ← M[AD]

Status Affected: All

Description: The selected register is replaced by what is in RAM at the selected address

RET Return PC

Syntax : {1001111,XXXXXXXXX}

Operands: None

Operation: PC ← POP

Status Affected: None

Description: The top of the stack is POPed out into the program counter

CALL — Call

Syntax : {1001110,Literal[8:0]}

Operands: Literal - 9-bit

Operation: PUSH PC+1, PC ← zf AD

Status Affected: None

Description: The next program count is pushed into the stack and the program counter goes to the specified value

JMPI — Jump Immediate

Syntax : {1001100,Literal[8:0]}

Operands: Literal - 9-bit

Operation: PC ← zf AD

Status Affected: None

Description: The program counter goes to the specified value

JMPR — Jump Register

Syntax : {1001101,SB[2:0],XXXXXX}

Operands: SB – 0 to 7

Operation: PC ← R[SB]

Status Affected: None

Description: The program counter goes to the value in the selected register

BSET — Bit Set

Syntax : {1001001, DR[2:0],b[3:0],XX}

Operands: DR – 0 to 7, b – 0 to 15

Operation: R[DR].b = 1

Status Affected: V, Z

Description: The selected bit of the selected register becomes 1

BCLR — Bit Clear

Syntax : {1001000, DR[2:0],b[3:0],XX}

Operands: DR – 0 to 7, b – 0 to 15

Operation: R[DR].b = 0

Status Affected: V, Z

Description: The selected bit of the selected register becomes 0

BTSS — Bit Test Skip If Set

Syntax : {1001010, SA[2:0],b[3:0],XX}

Operands: SA – 0 to 7, b – 0 to 15

Operation: if(R[SA].b == 1) PC ← PC + 2;
else PC ←PC + 1

Status Affected: None

Description: Checks to see if the selected bit of the selected register is 1. If so, the program counter skips the next instruction, if not, nothing happens

BTSC Bit Test Skip If Clear

Syntax : {1001011, SA[2:0],b[3:0],XX}

Operands: SA – 0 to 7, b – 0 to 15

Operation: if(R[SA].b == 0) PC ← PC + 2;

else PC ←PC + 1

Status Affected: None

Description: Checks to see if the selected bit of the selected register is 0. If so, the program counter skips the next instruction, if not, nothing happens

BRZ Branch If Zero

Syntax : {10110, SA[2:0],Literal[7:0]}

Operands: SA – 0 to 7, Literal 16-bit

Operation: if(R[SA]==0) PC ←PC + se AD

Status Affected: None

Description: Checks to see if the status bit Z is high, if so the program counter goes to the specified address, if not nothing happens

BRN Branch If Negative

Syntax : {101101 SA[2:0],Literal[7:0]}

Operands: SA – 0 to 7, Literal 16-bit

Operation: if(R[SA]<0) PC ←PC + se AD

Status Affected: None

Description: Checks to see if the status bit N is high, if so the program counter goes to the specified address, if not nothing happens

Example Program:

Table 9: Example Stack Test Program

Instruction	Assembly Instruction	Binary Instruction Word
1		1100100000000001
2		1101000000000010
3		0110000000000000
4		1000000001000000
5		1000000010000000
6		1001110000001101
7		1000001000000000
8		1000001000000000
9		1000001000000000
10		1000001000000000
11		1000001000000000
12		1000001000000000
13		0110000000000000
14		0110000000000000
15		0110000000000000
16		1001111000000000

The above is an example of a program that is preloaded on the CH09243. It checks to make sure the stack is working properly by confirming that items go in and out of the stack in the correct order. It also checks that the stack cannot overflow and that POPing with an empty stack doesn't work. Finally, it also tests Call and Return PC by making sure that the POP instructions are skipped and not completed until Return PC is executed.

Performance:

The maximum clock speed that the CH09243 can achieve without error is almost exactly 24MHz, which was determined using development board testing and compiler analysis (Timing Analyzer). There are a few reasons for this lower speed and if there was additional time to make improvements the speed could be increased specifically. The first major reason that the processor is slower is that a good amount of it was codded in behavioral Verilog as well as with conditional statements, both of which are difficult to synthesize onto an FPGA and therefore add larges amount of propagation delay. Secondly, this particular model has every function implemented which adds more complexity to the circuit. Finally, the CH09243 uses multiplexers for almost all selection and some speed could be gained if they were changed to a tri-state buffer design.

Appendix:

A.1: Param_Set in SFU Code

```
1   module Param_Set(Set, S, SFU_sel, FS);
2
3   input[4:0] FS;
4   output reg Set, SFU_sel;
5   output reg[1:0] S;
6
7   always @(FS) begin
8   case (FS[4:0])
9
10  5'b11010: begin  Set = 1'b0; //Multiply
11  S = 2'b00;
12  SFU_sel = 1'b1;
13  end
14
15  5'b11011: begin  Set = 1'b0; //Add with Carry
16  S = 2'b01;
17  SFU_sel = 1'b1;
18  end
19
20  5'b11100: begin  Set = 1'b0; //Arithmetic Shift Right
21  S = 2'b10;
22  SFU_sel = 1'b1;
23  end
24
25  5'b11101: begin  Set = 1'b0; //Bitwise Clear
26  S = 2'b11;
27  SFU_sel = 1'b1;
28  end
29
30  5'b11110: begin  Set = 1'b1; //Bitwise Set
31  S = 2'b11;
32  SFU_sel = 1'b1;
33  end
34
35  5'b11111: begin  Set = 1'b0; //Bitwise Test
36  S = 2'b11;
37  SFU_sel = 1'b1;
38  end
39
40  default: begin  Set = 1'b0; //Non SFU Function
41  S = 2'b00;
42  SFU_sel = 1'b0;
43  end
44
45
46  endcase
47
48  end
49
50  endmodule
```

A.2: Multiply_16bit in SFU Code

```
module Multiply_16bit(F, A, B);

input signed[15:0] A, B;
output signed[15:0] F;

assign F = A*B;

endmodule
```

A.3: Add_w_carry_16bit in SFU Code

```
module Add_w_carry_16bit(F, A, B, C);

input signed[15:0] A, B;
input C;
output reg signed[15:0] F;

always @(A) begin
    if (C)
        F = (A+B)+1;
    else
        F = A+B;
end
endmodule
```

A.4: Arith_SR 16bit in SFU Code

```
module Arith_SR_16bit(F, A);

input signed[15:0] A;
output reg signed[15:0] F;

always @(A)
begin
F = A >>> 1;
end

endmodule
```

A.5: Bit_OP_16bit in SFU Code

```
module Bit_OP_16bit(F, A, BS, S);

input[15:0] A;
input[3:0] BS;
input S;
output reg[15:0] F;

always @(A) begin
case (BS[3:0])
4'b0000: begin  F = {A[15:1],S}; //Bit 0
end

4'b0001: begin  F = {A[15:2],S,A[0]}; //Bit 1
end

4'b0010: begin  F = {A[15:3],S,A[1:0]}; //Bit 2
end

4'b0011: begin  F = {A[15:4],S,A[2:0]}; //Bit 3
end

4'b0100: begin  F = {A[15:5],S,A[3:0]}; //Bit 4
end

4'b0101: begin  F = {A[15:6],S,A[4:0]}; //Bit 5
end

4'b0110: begin  F = {A[15:7],S,A[5:0]}; //Bit 6
end

4'b0111: begin  F = {A[15:8],S,A[6:0]}; //Bit 7
end

4'b1000: begin  F = {A[15:9],S,A[7:0]}; //Bit 8
end

4'b1001: begin  F = {A[15:10],S,A[8:0]}; //Bit 9
end

4'b1010: begin  F = {A[15:11],S,A[9:0]}; //Bit 10
end

4'b1011: begin  F = {A[15:12],S,A[10:0]}; //Bit 11
end

4'b1100: begin  F = {A[15:13],S,A[11:0]}; //Bit 12
end

4'b1101: begin  F = {A[15:14],S,A[12:0]}; //Bit 13
end

4'b1110: begin  F = {A[15],S,A[13:0]}; //Bit 14
end

4'b1111: begin  F = {S,A[14:0]}; //Bit 15
end

endcase

end
```

A.6: Bit_Test is SFU Code

```
module Bit_Test_16bit(F, A, BS);

input[15:0] A;
input[3:0] BS;
output reg F;

always @(A) begin
case (BS[3:0])
4'b0000: begin  F = A[0]; //Bit 0
end

4'b0001: begin  F = A[1]; //Bit 1
end

4'b0010: begin  F = A[2]; //Bit 2
end

4'b0011: begin  F = A[3]; //Bit 3
end

4'b0100: begin  F = A[4]; //Bit 4
end

4'b0101: begin  F = A[5]; //Bit 5
end

4'b0110: begin  F = A[6]; //Bit 6
end

4'b0111: begin  F = A[7]; //Bit 7
end

4'b1000: begin  F = A[8]; //Bit 8
end

4'b1001: begin  F = A[9]; //Bit 9
end

4'b1010: begin  F = A[10]; //Bit 10
end

4'b1011: begin  F = A[11]; //Bit 11
end

4'b1100: begin  F = A[12]; //Bit 12
end

4'b1101: begin  F = A[13]; //Bit 13
end

4'b1110: begin  F = A[14]; //Bit 14
end

4'b1111: begin  F = A[15]; //Bit 15
end

endcase

end
```

A.7: One Instruction Decoder Handler Code

```
1   module ID_00_Handler (ISin, CWout, literal);
2   input [13:0] ISin;
3   output reg [15:0] literal;
4   output reg [18:0] CWout;
5   always @(ISin) begin
6   casex (ISin[13:9])
7
8   5'b00000: begin  CWout = 19'b0000000000000000000; //No Operation
9    literal= 16'b1111111111111111; end
10
11  5'b001xx: begin  CWout = {6'b101001,ISin[10:8],ISin[10:8],7'b0000100}; //Add Immediate
12   literal = {8'b00000000,ISin[7:0]}; end
13
14  5'b010xx: begin  CWout = {6'b101101,ISin[10:8],ISin[10:8],7'b0000100}; //Subtract Immediate
15   literal = {8'b00000000,ISin[7:0]}; end
16
17  5'b011xx: begin  CWout = {6'b010001,ISin[10:8],ISin[10:8],7'b0000100}; //AND Immediate
18   literal = {8'b00000000,ISin[7:0]}; end
19
20  5'b101xx: begin  CWout = {6'b011101,ISin[10:8],ISin[10:8],7'b0000100}; //OR Immediate
21   literal = {8'b00000000,ISin[7:0]}; end
22
23  5'b110xx: begin  CWout = {6'b001101,ISin[10:8],ISin[10:8],7'b0000100}; //XOR Immediate
24   literal = {8'b00000000,ISin[7:0]}; end
25
26  endcase
27
28  end
29
30
31  endmodule
```

A.8: Section of Instruction Memory Code

```
module Instruction_Mem(Instruction_out, PCAdress);
  output reg [15:0] Instruction_out;
  input  [15:0] PCAdress;

  always @(PCAdress)
  begin
    case (PCAdress)

      8'h0:  Instruction_out = 16'hc000; // LDI R0,0
      8'h1:  Instruction_out = 16'ha802; // STM R0,2 #set direction as input CHECK ADDRESS
      8'h2:  Instruction_out = 16'hc66b; // LDI R0,1643 #11 bit -405
      8'h3:  Instruction_out = 16'hec00; // LDI R5,1024 #11 bit -1024
      8'h4:  Instruction_out = 16'hd119; // LDI R2,281
      8'h5:  Instruction_out = 16'h6895; // ADD R2,R2,R5
      8'h6:  Instruction_out = 16'h6805; // ADD R0,R0,R5
      8'h7:  Instruction_out = 16'h696d; // ADD R5,R5,R5
      8'h8:  Instruction_out = 16'h68aa; // ADD R2,R5,R2
      8'h9:  Instruction_out = 16'h7168; // SHL R5,R5
      8'ha:  Instruction_out = 16'h6805; // ADD R0,R0,R5
      8'hb:  Instruction_out = 16'h7168; // SHL R5,R5
      8'hc:  Instruction_out = 16'h6895; // ADD R2,R2,R5
      8'hd:  Instruction_out = 16'h6368; // NEG R5,R5
      8'he:  Instruction_out = 16'h6802; // ADD R0,R0,R2
      8'hf:  Instruction_out = 16'h6802; // ADD R0,R0,R2
      8'h10:  Instruction_out = 16'ha818; // STM R0,24 #store special values
      8'h11:  Instruction_out = 16'haa19; // STM R2,25
      8'h12:  Instruction_out = 16'had1a; // STM R5,26
      8'h13:  Instruction_out = 16'hf014; // LDI R6,20 #load R6 with jump address, main loop
      8'h14:  Instruction_out = 16'ha003; // LDM R0,3 #load input to R0 CHECK ADDRESS
      8'h15:  Instruction_out = 16'h4640; // NOT R1,R0
      8'h16:  Instruction_out = 16'h1901; // ANDI R1,1 #mask bit 1 of input
      8'h17:  Instruction_out = 16'hb108; // BRZ R1,8 #if bit1 of input was on then all off
      8'h18:  Instruction_out = 16'h4640; // NOT R1,R0
      8'h19:  Instruction_out = 16'h1902; // ANDI R1,2 #mask bit 2 of input
      8'h1a:  Instruction_out = 16'hb110; // BRZ R1,16 #if bit2 of input was on then all on
      8'h1b:  Instruction_out = 16'h4640; // NOT R1,R0
      8'h1c:  Instruction_out = 16'h1904; // ANDI R1,4 #mask bit 3 of input
      8'h1d:  Instruction_out = 16'hb119; // BRZ R1,25 #if bit3 of input was on then display
      8'h1e:  Instruction_out = 16'h9b80; // JMPR R6 #end main loop
      8'h1f:  Instruction_out = 16'hc000; // LDI R0,0 #start all off
      8'h20:  Instruction_out = 16'h5840; // MOVA R1,R0
      8'h21:  Instruction_out = 16'h5888; // MOVA R2,R1
      8'h22:  Instruction_out = 16'h58d0; // MOVA R3,R2
      8'h23:  Instruction_out = 16'h5918; // MOVA R4,R3
      8'h24:  Instruction_out = 16'h5960; // MOVA R5,R4 #end all off
      8'h25:  Instruction_out = 16'ha703; // LDM R7,3 #load input to R7 CHECK ADDRESS
      8'h26:  Instruction_out = 16'h47f8; // NOT R7,R7
      8'h27:  Instruction_out = 16'h1f01; // ANDI R7,1
      8'h28:  Instruction_out = 16'hb7fd; // BRZ R7,-3 #while input bit 1 is on
```

A.9: Program Counter Code

```
1   module PC(clk, load, reset, D, count);
2       input clk, load, reset;
3       input [15:0] D;
4       output reg [15:0] count;
5
6       always @(posedge clk or negedge reset)
7            if (~reset)
8               count <= 16'b0;
9            else if (load)
10              count <= D;
11           else
12              count <= count + 1'b1;
13  endmodule
```