DESIGN AND DEVELOPMENT OF A SOFT RECONFIGURABLE POWER ELECTRONIC CONTROL PROCESSOR

KEVIN S,
ER&DCI, CDAC, Trivandrum,
kevinivek92@gmail.com

AJEESH A,
Senior Engineer, Power Electronics Group, CDAC, Trivandrum,
ajeesh@cdac.in

DIVYA D.S,
Project Engineer, ER&DCI, CDAC, Trivandrum,
divyads@cdac.in

ABSTRACT
This paper proposes to build a soft, platform independent, reconfigurable power electronic control processor in HDL. Currently power electronic control processors are designed with digital signal processors/microcontrollers with limited re-configurability and input-output capabilities, also DSP/microcontrollers suffer processor obsolescence risk. This can be reduced with the use of a reconfigurable control processor design in HDL and implemented on FPGA. Currently FPGA are used along with DSP for glue logic purpose and input-output expansion. So it is economical to use a single FPGA along with a soft core CPU and control modules as hardware accelerators. Some of the soft cores available are licensed and are targeted to a particular FPGA, so a 32 bit soft, reconfigurable power electronic control processor is developed from scratch.

I. INTRODUCTION

In this paper development of a soft, reconfigurable power electronic control processor is discussed. The conventionally used soft cores in power electronics like NIOS II are highly efficient but they take 8-9 cycles for memory read or write instructions. So to reduce this memory read and write cycles, the need for a soft reconfigurable processor arises. This newly developed soft core is also a FPGA independent one, so that it can be implemented on any available FPGA. Two processor cores, NIOS II and the SAYEH processor are selected for study purpose. The datapath components, instruction set architecture etc are discussed in this paper.

II. BASIC PROCESSOR DESIGN

A processor is basically a digital circuit which incorporates the functions of a computer's central processing unit (CPU) on a single Integrated Circuit (IC), or at most a few integrated circuits. The internal arrangement of a processor varies depending on the age of the design and the intended purposes of the microprocessor. The complexity of an Integrated Circuit (IC) is bounded by physical limitations of the number of transistors that can be put onto one chip, the number of package terminations that can connect the processor to other parts of the system, the number of interconnections it is possible to make on the chip, and the heat that the chip can dissipate. Advancing technology makes more complex and powerful chips feasible to manufacture.

A minimal hypothetical microprocessor might only include an arithmetic logic unit (ALU) and a control logic section. The ALU performs operations such as addition and subtraction. Each operation of the ALU sets one or more flags in a status register, which indicate the results of the last operation (zero value, negative number, overflow, or others). The control logic retrieves instruction codes from memory and initiates the sequence of operations required for the ALU to carry out the instruction. A single operation code might affect many individual data paths, registers, and other elements of the processor. A minimal hypothetical microprocessor can do only a limited number of operations with less complexity. It may only

consist of the basic minimum components, and the most basic components needed for a processor are, an Arithmetic Unit, a Logic Unit, and a Multiplexer.
Logic unit: It does all the logical operations like AND, OR, XOR, NOT etc based on the select (SEL) input.
Arithmetic unit: It does all the arithmetic operations like addition, subtraction etc. This is also done based on the select (SEL) input.
Multiplexer (MUX): The multiplexer has 3 inputs, a select line (SEL), output lines each of one from the arithmetic unit and the logic unit. The MUX selects either one of the arithmetic output or the logic output depending upon the select line input. This select line values can also be called as the OPCODE (operation codes). We can define different opcodes for different functionality. There are two inputs a and b which is of 8 bit wide, the sel is of 4 bit wide and is also fed to both logic and arithmetic unit. When the inputs are fed both the arithmetic and logic units produces their outputs, depending on the MSB bit of sel line, the mux selects the output. Since the sel line is of 4 bit wide, when can assign 2^4 different operations. This is the basic working of a processor.
The arithmetic and logic unit works together to form the ALU. All the arithmetic and logical operations are done in ALU.

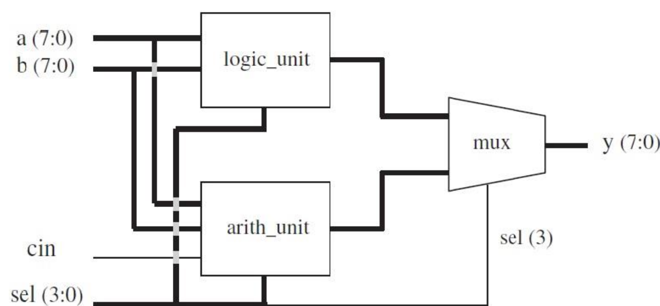

Figure. 1 : A minimal hypothetical microprocessor components.

A list of opcodes and their functionality is listed in Table 1. The different codes defines different functionality for each opcodes. The MSB bit decides whether the output should be from the arithmetic unit or the logic unit, if MSB is '0' then arithmetic unit's output is selected and if its '1' then logic unit's output is selected.

Table 1 : Opcodes and their Functionality.

sel	Operation	Function	Unit
0000	y <= a	Transfer a	Arithmetic
0001	y <= a+1	Increment a	
0010	y <= a-1	Decrement a	
0011	y <= b	Transfer b	
0100	y <= b+1	Increment b	
0101	y <= b-1	Decrement b	
0110	y <= a+b	Add a and b	
0111	y <= a+b+cin	Add a and b with carry	
1000	y <= NOT a	Complement a	Logic
1001	y <= NOT b	Complement b	
1010	y <= a AND b	AND	
1011	y <= a OR b	OR	
1100	y <= a NAND b	NAND	
1101	y <= a NOR b	NOR	
1110	y <= a XOR b	XOR	
1111	y <= a XNOR b	XNOR	

III. NIOS II PROCESSOR

The Nios II processor is a general-purpose RISC processor core with has a full 32-bit instruction set, data path and address space. NIOS II has its own thirty 32 bit general purpose and thirty 32 bit control registers, optional shadow register sets, thirty two interrupt sources and an external interrupt controller interface for more interrupt sources. It also a single-instruction 32 x 32 multiply and divide producing a 32-bit result, dedicated instructions for computing 64-bit and 128-bit products of multiplication, optional floating-point instructions for single-precision floating-point operations, and a single-instruction barrel shifter and also access to a variety of on-chip peripherals, and interfaces to on-chip memories and peripherals.

It supports hardware-assisted debug module enabling processor start, stop, step, and trace under control of the Nios II software development tools and na optional memory management unit (MMU) to support operating systems that require MMUs.

A. NIOS II Architecture

A Nios II processor system is equivalent to a microcontroller or "computer on a chip" that includes a processor and a combination of peripherals and memory on a single chip. A Nios II processor system consists of a Nios II processor core, a set of on-chip peripherals, on-chip memory, and interfaces to off-chip memory, all implemented on a single Altera device. Like a microcontroller family, all Nios II processor systems use a consistent instruction set and programming model.
The NIOS II architecture defines the following functional
units: register file, arithmetic logic unit (ALU), interface to custom instruction logic, exception controller, internal or external interrupt controller, instruction bus, data bus, memory management unit (MMU), memory protection unit (MPU), instruction and data cache memories and tightly-coupled memory interfaces for instructions and data JTAG debug module.

1. Register file

The Nios II architecture supports a flat register file, consisting of thirty-two 32-bit general-purpose integer registers, and up to thirty-two 32-bit control registers. The architecture supports supervisor and user modes that allow system code to protect the control registers from errant applications.

2. Arithmetic and Logic Unit

The Nios II ALU operates on data stored in general-purpose registers. ALU operations take one or two inputs from registers, and store a result back in a register. The ALU supports the data operations described below. To implement any other operation, software computes the result by performing a combination of the fundamental operations. ALU supports addition, subtraction, multiplication, and division on signed and unsigned operands it also performs equal, not-equal, greater-than-or-equal, and less-than relational operations on signed and unsigned operands and supports AND, OR, NOR, and XOR logical operations. The ALU supports shift and rotate operations, and can shift/rotate data by 0 to 31 bit positions per instruction, supports arithmetic shift right and logical shift right/left. Also supports rotate left/right.

3. Operating Modes

Operating modes control how the processor operates, manages system memory, and accesses peripherals. The Nios II architecture supports two operating modes: Supervisor mode
and User mode.
Supervisor mode allows unrestricted operation of the processor. All code has access to all processor instructions and resources. The processor may perform any operation the Nios II architecture provides. Any instruction may be executed, any I/O operation may be initiated, and any area of memory may be accessed. Operating systems and other system software run in supervisor mode. In systems with an MMU, application code runs in user mode, and the operating system, running in supervisor mode, controls the application's access to memory and peripherals. In systems with an MPU, your system software controls the mode in which your application code runs. In Nios II systems without an MMU or MPU, all application and system code runs in supervisor mode. Code that needs direct access to and control of the processor runs in supervisor mode. For example, the processor enters supervisor mode whenever a processor exception (including processor reset or break) occurs. Software debugging tools also use supervisor mode to implement features such as breakpoints and watchpoints.
User mode is available only when the Nios II processor in your hardware design includes an MMU or MPU. User mode exists solely to support operating systems. Operating systems (that make use of the processor's user mode) run your application code in user mode. The user mode capabilities of the processor are a subset of the supervisor mode capabilities. Only a subset of the instruction set is available in user mode. The operating system determines which memory addresses are accessible to user mode applications. Attempts by user mode applications to access memory locations without user access enabled causes an exception. Code

running in user mode uses system calls to make requests to the operating system to perform I/O operations, manage memory, and access other system functionality in the supervisor memory.

IV. SAYEH PROCESSOR

Processors play a major role in the design of embedded systems. An embedded processor may be used as the central processing unit of an embedded system, or it may just be used as a convenient and fast way of implementing a hardware function. With embedded systems, understanding how a processor works, its software, and software utilities, such as compilers and assemblers, are key topics that a hardware designer should be familiar with.

The CPU is named as SAYEH (Simple Architecture, Yet Enough Hardware). This processor can be used as an embedded core for the design of an FIR filter. SAYEH is a processor with a set of instructions and an architecture that can be used for real embedded processor applications.

SAYEH components that are used by its instructions include the standard registers such as the Program Counter, Instruction Register, the Arithmetic Logic Unit, and Status Register. In addition, this processor has a register file forming registers R0, R1, R2 and R3 as well as a Window Pointer that defines R0, R1, R2 and R3 within the register file. The program counter and instruction register is of 16 bit wide. The instruction register can be loaded with a single 16 bit instruction, or with a single 8 bit instruction or with two 8 bit instructions. SAYEH processor has sixty four 16 bit general purpose registers.

A. **Instruction format**

The SAYEH processor works on both 16-bit and 8-bit instructions. 16-bit instructions have the Immediate field and the 8-bit instructions do not. The OPCODE filed is a 4-bit code that specifies the type of instruction. The Left and Right fields are two bit codes selecting R0 through R3 for source and/or destination of an instruction. Usually, Left is used for destination and Right for source. The Immediate filed is used for immediate data, or if two 8-bit instructions are packed, it is used for the second instruction. Processor has a total of 29 instructions. The instruction set is shown in figure 2

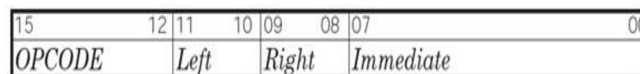

Figure 2 : Instruction set of SAYEH processor.

B. **Data path**

The processor has a data path and a controller. Data path components are Addressing Unit, IR, WP, Register File, Arithmetic Unit, and the Flags register. The Addressing Unit is further partitioned into the PC and Address Logic. The Addressing Logic is a combinational circuit that is capable of adding its inputs to generate a 16-bit output that forms the address for the processor memory. Program Counter and Instruction Register are 16-bit registers. Register File is a two-port memory and a file of 64 16-bit registers. The Window Pointer is a 6-bit register that is used as the base of the Register File. Specific registers for read and write (R0, R1, R2 or R3) in the Register File are selected by its 4-bit input bus coming from the Instruction Register. Two bits are used to select a source register and other two bits select the destination register. When the Window Pointer is enabled, it adds its 6-bit input to its current data.

The Flags register is a 2-bit register that saves the flag outputs of the Arithmetic Unit. The Arithmetic Unit is a 16-bit arithmetic and logic unit that has logical, shift, add and compare operations. A 9-bit input selects one of the nine functions of the ALU. This code is provided by the processor controller. Controller of SAYEH has eleven states for various reset, fetch, decode, execute, and halt operations. Signals generated by the controller control logic unit operations and register clocking in the datapath. SAYEH sequential data components and its controller are triggered on the rising edge of the main system clock. Control signals remain active after one rising edge through the next. This duration allows for propagation of signals through the busses and logic units in the data path.

Combinational and sequential SAYEH data components are described here. The combinational ones are like the ALU that performs arithmetic and logical operations. The function of such units is controlled by the controller. The sequential components are clocked with the negative edge of the main CPU clock. These components have functionalities like loading and resetting and are controlled by the controller

V. SOFT PROCESSOR

The soft processor is a 32-bit processor, which can handle all of the operations using 32 bit data. The processor works on 24 bit instructions and has an address bus width of 16 bit. This core is developed to handle data movement operations in minimum clock cycles. Soft core is based on Harvard architecture so it has separate busses for handling address and data, it is an asynchronous active low reset based system.

It has four 32 bit wide internal registers and an immediate memory which can store two hundred and fifty six 32 bit wide data values. Each time an instruction is executed the instruction status signal is made high for one clock cycle so that the next instruction is fetched from the program memory.

Figure 4 shows the basic structure of the soft processor.

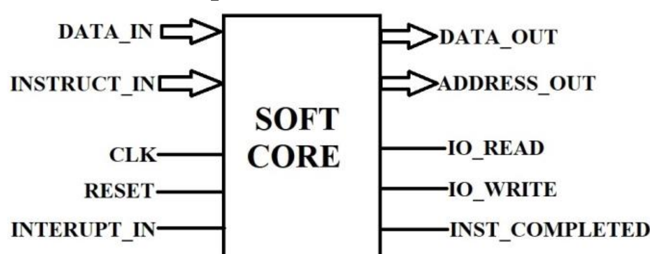

Figure 4: Soft processor

This soft processor is based on asynchronous ie, the system reset can occur at the exact time when the reset is asserted. This processor has four 32 bit wide internal registers to store the data coming from external devices. Figure 5 shows the block diagram of the soft core.

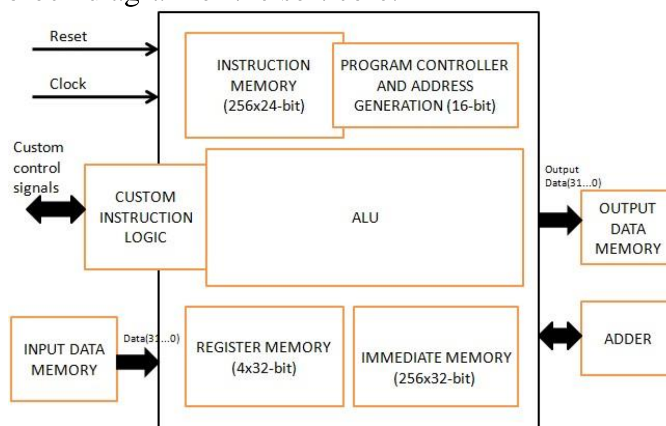

Figure 5: Block diagram of the Soft processor

The core consist of an ALU which works on custom instruction logic. Processor has its own instruction memory with a program controller and address generation block, this unit generates the address of the next instruction to be passed to the ALU for execution. The core consist of four internal 32 bit registers. It also has an immediate memory for storing immediate data. Immediate data needed for execution is fed from this memory.

A. Instruction Set

The core is based on a 24 bit wide instruction set. The MSB 8 bits are used to specify the OPCODE, the next 8 bits defines the destination address location and the LSB 8 bits gives the source address location. Figure 6 shows the instruction set of the soft core.

Figure 6: Instruction set

Table 2 gives a brief idea of the opcodes designed and their functionality. Soft core contains a total of nineteen custom instructions. It includes instructions to move data between different memories, move data between registers etc. Next type of instructions which are included are the addition instructions. These instructions are used to add two 32 bit data and stores the result back to a destination memory. The next type

of instructions are the Rotate/Shift instructions, which are used to shift a 32 bit data towards right or left. The instruction set also contains a Jump instruction. This instruction skips certain number of instructions. The number of instructions which has to be skipped can be specified in the instruction itself, the maximum skip length is limited to 256. Table 2 shows the opcodes and the functionality of different instructions.

OPCODE-01

This instruction is used to move a 32 bit data stored in an immediate memory location to register memory. The source immediate memory address can be specified on the 0-7 bits and the destination register memory address can be specified on the 8-15 bits of the instruction field respectively.

OPCODE-02

This instruction is used to move a 32 bit data stored in an external memory location to an immediate memory location. The source external memory location can be specified on the 0-7 bits and the destination immediate memory location be specified on the 8-15 bits of the instruction field respectively.

OPCODE-03

This instruction is used to move a 32 bit data stored in an immediate memory location to an external memory location. The source immediate memory location can be specified on the 0-7 bits and the destination external memory location be specified on the 8-15 bits of the instruction field respectively.

TABLE 2: Opcodes and functionality.

INSTRUCTION	OPCODE	OPERATION
MVIR	01	Move immediate data to register memory.
MVXI	02	Move data from external memory to immediate memory.
MVIX	03	Move data from immediate memory to external memory.
MVX	04	Move data between two external memories.
MVXR	05	Move data from external memory to register memory.
MVRX	06	Move data from register memory to external memory.
MVR	08	Move data between two registers.
MVRI	12	Move data from register to immediate memory.
MVI	13	Move data between two immediate memories.
ADDR	0A	Add data in registers and store in a register.
ADDI	1A	Add data in immediate memory and store in a immediate memory location.
ADDRI	2A	Add register and immediate data values and store in a register.
ADDIR	3A	Add register and immediate data values and store in a immediate memory location
JMP	F0	Unconditional jump. Jumps to an address location without any condition.
RL	B0	Rotate left by one bit.
RR	B1	Rotate right by one bit.
RLC	B2	Rotate left with carry bit.
RRC	B3	Rotate right with carry bit.
SWP	B4	Swap LSB and MSB bits.

OPCODE-04

This instruction is used to move data between two external memories connected to the core. The source memory address can be specified on the 0-7 bits, and the destination external memory address can be specified on the 8-15 bits of the instruction field respectively.

OPCODE-05

This instruction is used to move a 32 bit data stored in an external memory location to a register memory. The source external memory address can be specified on the 0-7 bits and the destination register memory address can be specified on the 8-15 bits of the instruction field respectively.

OPCODE-06

This instruction is used to move a 32 bit data stored in a register memory location to an external memory. The source register memory address can be specified on the 0-7 bits and the destination external memory address can be specified on the 8-15 bits of the instruction field respectively.

OPCODE-08

This instruction is used to move a 32 bit data between two register memories. The source register address can be specified on the 0-7 bits and the destination register memory address can be specified on the 8-15 bits of the instruction field respectively.

OPCODE - 0A

This instruction is used to add two 32 bit data stored in register memory location, and store the result back to the destination register memory location. The source register memory address can be specified on the 0-7 bits and the destination register memory address can be specified on the 8-15 bits of the instruction field respectively.

OPCODE - 1A

This instruction is used to add two 32 bit data stored in immediate memory location, and store the result back to the destination immediate memory location. The source immediate memory address can be specified on the 0-7 bits and the destination immediate memory address can be specified on the 8-15 bits of the instruction field respectively.

OPCODE - 2A

This instruction is used to add two 32 bit data stored in an immediate memory location and in a register memory, then stores the result back to the destination register memory location. The source immediate memory address can be specified on the 0-7 bits and the destination register memory address can be specified on the 8-15 bits of the instruction field respectively.

OPCODE - 3A

This instruction is used to add two 32 bit data stored in an immediate memory location and in a register memory, then stores the result back to the destination immediate memory location. The source register memory address can be specified on the 0-7 bits and the destination immediate memory address can be specified on the 8-15 bits of the instruction field respectively.

OPCODE-F0

This instruction is used to skip few instructions. Using this instruction we can skip upto 256 instructions, and jumping can be done only in forward direction.

OPCODE-B0

This instruction is used to shift left by one bit without using carry flag. Figure 7.1 shows the left shifting operation. When this instruction is executed the data is shifted left by one bit.

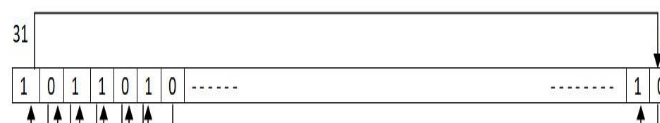

Figure 7.1: Shift left instruction.

OPCODE-B1

This instruction is used to shift right by one bit without using carry flag. Figure 7.2 shows the right shifting operation. When this instruction is executed the data is shifted right by one bit.

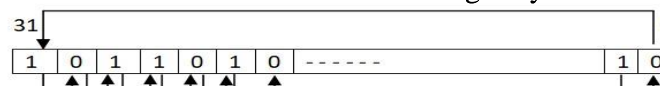

Figure 7.2: Shift right instruction.

OPCODE-B2

This instruction is used to shift left by one bit using carry flag. Figure 7.3 shows the left shifting operation with the carry flag. When this instruction is executed the data is shifted left by one bit.

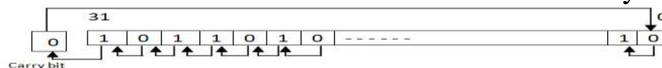

Figure 7.3: Shift left with carry instruction.

OPCODE-B3

This instruction is used to shift right by one bit using carry flag. Figure 7.4 shows the right shifting operation with the carry flag. When this instruction is executed the data is shifted right by one bit.

Figure 7.4: Shift right with carry instruction.

OPCODE-B4

This instruction is used to swap the LSB and MSB bits. Figure 7.5 shows the swapping mechanism.

Figure 7.5: Swap instruction.

VI. SIMULATION RESULTS.

A. Program memory

Program memory is used to store the instructions which has to be executed. Currently program memory can store up to two hundred and fifty six 24-bit instructions. Program memory is connected to the core, An instruction status (instruct_status) signal is used for the communication between. The core after execution of an instruction, the inst_cmplted(instruction status) signal of the core is made high for one clock cycle, now the program memory detects this change in the instruct_status signal and sends out the next instruction to be executed to the core. The connection diagram of core and program memory is shown in figure 8.1 and the simulation waveform is shown in figure 9.

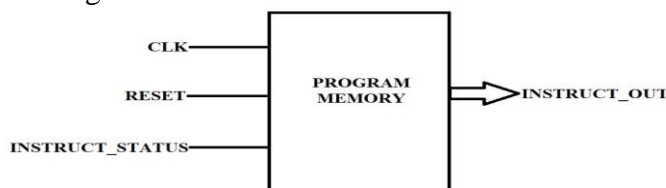

Figure 8: Program memory.

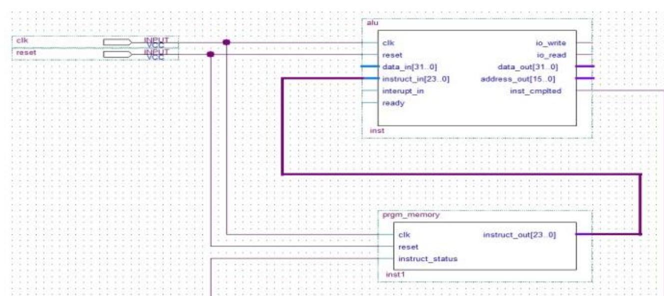

Figure 8.1: Connection diagram of program memory and the core

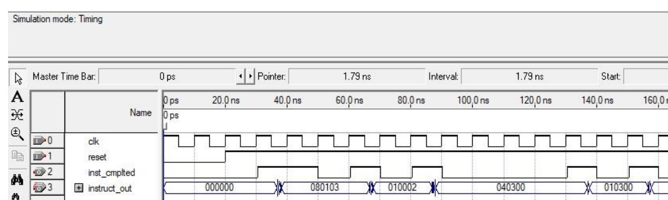

Figure 9: Simulation result.

B. Interfacing memory modules

The soft core is capable of transferring 32 bit wide data between two external memory modules. A unique separate instruction was developed to do this operation in minimum clock cycles. This whole operation includes, passing the source address(16 bit) after making the read signal high for a one clock cycle, fetching data (32 bit)from the external memory, then passing the destination memory address after making the write signal high for one clock cycle, and finally storing the 32 bit data in the destination external memory.

An instruction MVX was designed for this operation. The instruction opcode is "04", opcode is represented in the 23-16 bits of the instruction field, bits 15-8 defines the destination memory address and bits 7-0 defines the source memory address. Below shows the block diagram representation of ALU & external memories.

The connection diagram is shown in figure 10 and the simulation result is shown in figure 11.

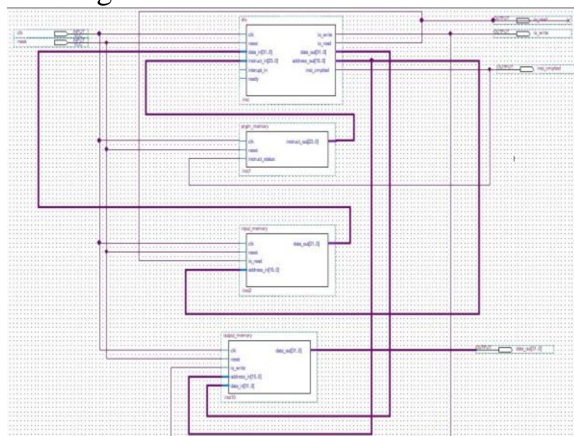

Figure 10: Connection diagram of external memory modules and the core.

The whole read and write operation takes about 5 clock cycle. The data from input memory is initially moved to the core and then the core moves this data to the output memory.

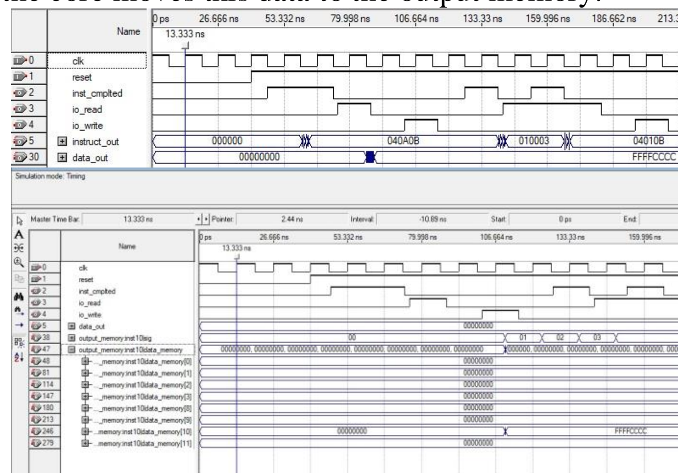

Figure 11: Simulation waveform

Figure 11 shows the simulation result of the instruction "040A0B". The core is connected to two external memory modules, input memory and the output memory. When this instruction is executed, the 32-bit data stored in the "0Bth" location of the input memory is transferred to the "0Ath" location of the output memory. Here in this case the data stored in the "0Bth" location is "FFFFCCCC". The whole instruction takes 6 clock cycles for the entire operation.

C. Interfacing adder module

A separate instruction was designed to connect an external adder module with the soft core. An 32 bit half adder block was designed which can take two 32-bit data as inputs and returns the sum back to the core. The connection diagram is shown in figure 12 and the simulation of different addition instruction is shown in figure 13.

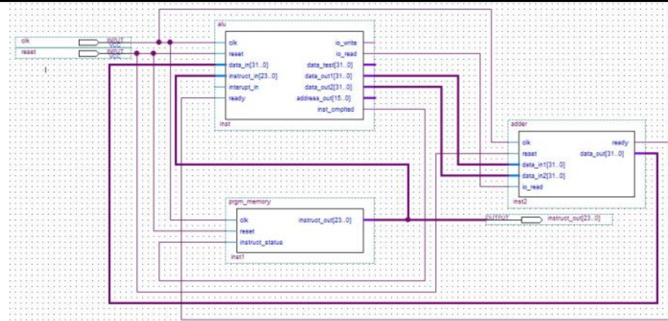

Figure 12: Connection diagram of adder and the core.

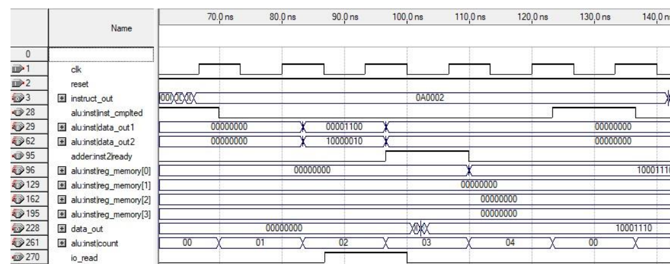

Figure 13: Simulation of ADDR instruction

In figure 13 the instruction executed is “0A0002”. This instruction is used to find sum of two 32-bit data stored in registers R0 and R2, and stores the sum in R2. Here R2 is the destination register.

D. MVI Instruction

The MVI instruction is used to move data between two immediate memory locations. The simulation result is shown below.

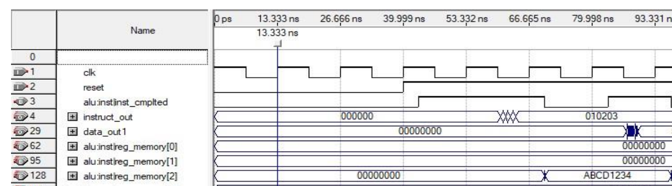

Figure 14: Simulation of MVI instruction.

The instruction used here is “010203”. Here the 32-bit data stored in the 03th location is copied to 02th location of the immediate memory. This whole instruction is carried out in 2 clock cycles. The data stored in the 03th location of immediate memory here is “ABCD1234”.

E. MVR Instruction

The MVR instruction is used to move data between two register memories. The simulation result is shown below.

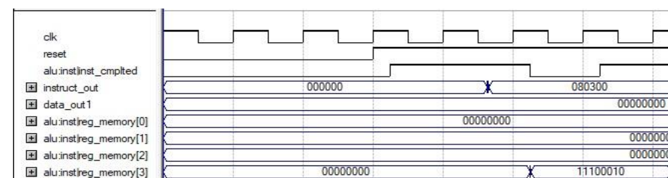

Figure 15: Simulation of MVR instruction.

The instruction used here is “080300”. Here the 32-bit data stored in the R0 register is copied to R0 register. This whole instruction is carried out in 2 clock cycles. The data stored in R0 in this case is “11100010”.

F. Rotate and Shift Instructions.

Rotate and shift operations are done to shift a 32 bit data towards left or right by one bit. There are four different shifting operations right shifting, left shifting, right shift with carry bit, left shift with carry bit and an instruction for swapping LSB and MSB bits. The data that should be shifted must be loaded into a

register first. Any register (R0,R1,R2,R3) can be used for shifting/swapping operating. An example of right and left shifting is shown below. The values initially stored in the registers is shown in Figure 16.

```
reg_memory(0)      <=X"12345678";
reg_memory(1)      <=X"00010001";
reg_memory(2)      <=X"0F0F0F0F";
reg_memory(3)      <=X"F0F0F0F0";
```

Figure 16: Register memory.

As an example, two instructions "B00001" and "B10002" are illustrated below. "B00001" is used for left shift operation of data stored in R1 register and "B10002" is used for right shift operation of data stored in R2 register. Register R1 is initially stored with a hex value 00010001 and after left shifting the value becomes 00020002, while register R2 is initially stored with a hex value 0F0F0F0F and after left shifting the value becomes 87878787. The simulation waveform is shown in figure 17.

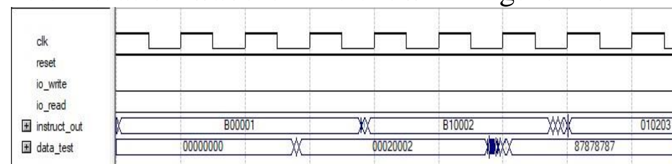

Figure 17:Left and right shifting simulation waveform.

G. Jump Instruction

Instruction having opcode "F0" is used for skipping or jumping successive instructions without any condition. This instruction is basically an unconditional jump instruction, which skips a certain number of instructions specified in the instruction along with the opcode. For example if the instruction is like F00107, the opcode is the MSB two hex values (here it is F0) and the LSB two hex values is the number of instructions which has to be skipped(here it is 7). So while executing this instruction the next seven instructions after this instruction will be skipped. The middle two hex values in this instruction are of no importance.

```
main_memory(0)  <=X"B40000";
main_memory(1)  <=X"B40001";
main_memory(2)  <=X"010203";
main_memory(3)  <=X"F00107";
main_memory(4)  <=X"010301";
main_memory(5)  <=X"010200";
main_memory(6)  <=X"010002";
main_memory(7)  <=X"010003";
main_memory(8)  <=X"010101";
main_memory(9)  <=X"010200";
main_memory(10)  <=X"010302";
main_memory(11)  <=X"010003";
main_memory(12)  <=X"010204";
main_memory(13)  <=X"F00004";
main_memory(14)  <=X"010302";
main_memory(15)  <=X"010003";
main_memory(16)  <=X"010201";
main_memory(17)  <=X"010300";
main_memory(18)  <=X"010002";
main_memory(19)  <=X"010103";
```

Figure 18 : Program Memory.

The instruction memory is shown in figure 18. It contains two jump instructions. Instruction F00107 skips next seven instructions and F00004 skips the next four instructions. The simulation result is shown in figure 19.

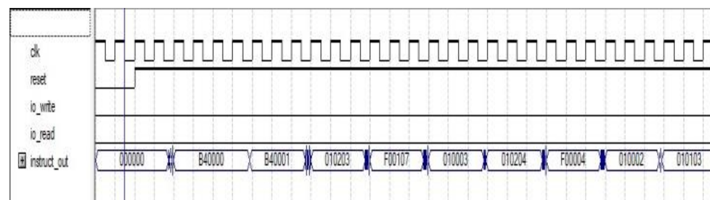

Figure 19: Simulation of Jump Instruction.

VII. TIMING REPORT

The timing was checked using Classic Timing Analyzer Tool in Quartus II. The soft core works at maximum frequency of 133.69 MHz. Figure 20 shows the timing report.

Figure 20: Timing Report in Quartus II.

VIII. ASSEMBLER

Assembler is a program which converts instructions written in low-level language to machine language. Machines can understand only binary values, ie either `1' or `0'. So it is necessary to convert the assembly code to binary form. This job is done by an assembler. Here an assembler was developed as a windows form application using Visual studio 2018. The windows form application is shown in Figure 21.

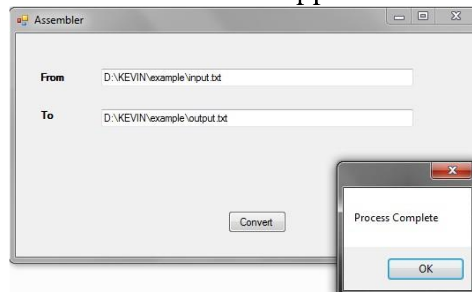

Figure 21: Windows form application : Assembler

The assembly level instructions which have to be executed are saved in a text file. This file's path must be specified in the text box `From'. Refer figure 20, the file `input.txt' contains the assembly codes. And the output file's path must be specified in the `To' text box. Now in this specified path a file named `output.txt', will be created which contains the corresponding binary values. The contents of the input file `input.txt' is shown in figure 21.

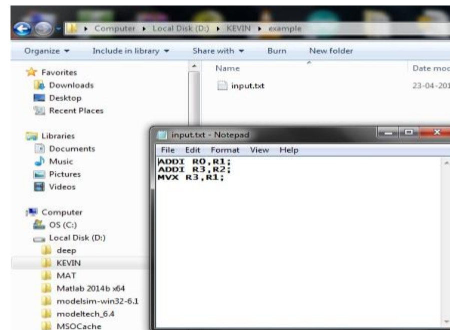

Figure 21: Input Assembly File.

Input.txt contains three assembly instructions. The assembler reads these instructions line by line, and converts to corresponding binary form. And finally this binary values is written to `output.txt'. This output file contains only binary values. This is shown in figure 22.

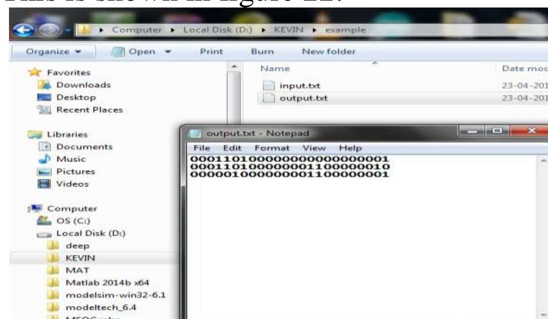

Figure 22: Output Binary File.

IX. CONCLUSION

In this paper have discussed about NIOS II processor and SAYEH processor. Have conducted studies on basics of processor designing and the different steps which has to be followed. Have discussed about

designing a 32 bit processor with 24 bit instruction set architecture. The coding of this core was done in VHDL using QUESTASIM.

Interfacing with memory modules and adder module has been done. The timing was checked using QUARTUS II Classic time analyzer. Have completed the design of adder module.

Designed and developed nineteen custom instructions. The developed Soft Core was found to run at a maximum frequency of 133.69 MHz. Developed an assembler program as a windows form application using visual studio. Assembler reads a text file containing assembly language instructions and generates a text file with binary values.

REFERENCES

I. Raj Prakash Singh, Ankit K Vashishtha,Krishna R, 32 bit Re-configurable RISC Processor Design and Implementation for BETA ISA with Inbuilt Matrix Multiplier, Sixth International Symposium on Embedded Computing and System Design, pages 112-116.

II. Peter Yiannacouras, J. Gregory Ste_an,Jonathan Rose, Exploration and customization of FPGA-Based Soft Processors, IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 26, NO. 2 FEBRUARY, 2007.

III. Michael Gschwind,Valentina Salapura, and Dietmar Maurer, FPGA Prototyping of a RISC Processor Core for Embedded Applications., IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, VOL. 9, NO. 2, APRIL 2001.

IV. Volnei A. Pedroni, “Circuit Design with VHDL,” MIT press Cambridge, Massachusetts London, England, 2004.

V. Zainalabedin Navabi, “VHDL: Modular Design and Synthesis of Cores and Systems”. Third edition, The McGraw-Hill Companies, 2007.

VI. Altera, NIOS II Datasheet,Nios II Classic Processor Reference Guide, 101,Innovation Drive San Jose, CA 95134,www.altera.com.