Novel Design of 32-bit Asynchronous (RISC) Microprocessor & its Implementation on FPGA

Archana Rani
Faculty of Engineering and Technology, Manav Rachna International University, Faridabad, India
Email address: Archana.bhatia.pec@gmail.com

Dr. Naresh Grover
Faculty of Engineering and Technology, Manav Rachna International University, Faridabad, India
Email address: dean.academics@mriu.edu.in

Received: 20 July 2017; Accepted: 12 September 2017; Published: 08 January 2018

Abstract—As the efficiency and power consumption plays an important role in electronic system design, an asynchronous design is used to reduce such challenges faced in synchronous architectures. The asynchronous processors have a number of advantages, especially in SoC (System on chip) including reduced crosstalk between analog and digital circuits, ease of integrating multi-rate circuits, ease of component reuse and less power consumption as well. This paper deals with the novel design and implementation of such type of asynchronous microprocessor by using VHDL on Xilinx ISE tool wherein it has the capability of handling even I-Type, R-Type and Jump instructions with multiplier instruction packet. Moreover, it uses separate memory for instructions and data read-write that can be changed at any time.

Index Terms—Asynchronous design, Processor, VHDL, MIPS, Synthesis & Simulation, Instruction data path, EDA Tools

I. INTRODUCTION

Now-a-days computers are evolving using RISC (Reduced Instruction Set Computer) Architecture replacing stack architecture with the intention to displace the hypothetical, emulated computer by a real one. Instruction Set Architecture (ISA) design is the concept of CISC (Complex Instruction Set Computer) that emphasize more on each instruction using a wide range of addressing modes and number of operands in various locations in its Instruction Set. This leads to execution time to vary as instructions are not of fixed length, hence demanding a very complex Control Unit, which occupies a large area on chip. While on the other side, the RISC Processor has less number of Instructions. The fixed instruction length with more general purpose registers to support load-store architecture and simplified addressing modes. This makes individual instruction to execute faster in order to achieve a net gain in performance and an overall simpler design with less silicon consumption.

The choice of an RISC has become more obvious with the increase in size and complexity of modern processors and software. The hardware designer has a substantial amount of freedom for design by making use of FPGA being much more aware of availability of resources and of its limitations than the software developer.

Before commencing the design of an asynchronous processor we have to first focus on the architecture of Asynchronous processor as well as the various steps involved in such designs in terms of the program cycle. This paper presents processor architecture design, its implementation followed by processor instruction set, data path flow for fetching unit, Register type, I-type and load /store type instruction flow. Thereafter this paper illustrates control unit design of processor that shows the controlling of signals for different units in processor design. Further, a complete internal structure is shown followed by features of novel processor architecture. In the end, results have been shown using implementation windows. The complete design has been written using VHDL and then simulated and synthesized by XILINX ISE tool.

A. Processor Architecture

The asynchronous processor internal operation is segmented into five pipeline stages and in each of them the operations of the tasks will be performed in the normal cycle of an instruction, i.e. search of the instruction (identified with the IF block), decoding of The instruction (identified with the ID block), execution of the operation (identified with the EX block), memory access (identified with the MEM block) and storage of the operation result (identified with the WB block) [24].

The first stage is Instruction Fetch that comprises of Instruction Memory, Program Counter, and Instruction Register. In this stage, a program counter will extract the next instruction from a location in program memory. It updates the program counter value with the next instruction location sequentially or the location determined by a branch. The second stage is instruction decoding which comprises of register file and the extender (sign & Zero). This stage determines the values
on which the control lines should be set as per the instruction. The third stage is the instruction execution stage, where ALU and necessary parts will come into action. In this stage, the instruction is actually sent to the ALU and branch locations are also calculated. The fourth stage is Memory execution stage for accessing of data from system memory. Finally, in Write back stage the values/data written back to the register(s).

II. INSTRUCTION IMPLEMENTATION

Every instruction propagates in a specified sequence such as fetch, decode, execution & write back. There are three types of instruction(s) in processor/controller based system. These are I-Type, R-Type and Jump type instruction. The different type of instruction data paths has been depicted in figures 3, 4, 5 and 6. Figure 3 shows the data path for the fetching process. It depicts the flow of the various path involved in fetching. The figure 4 and 5 shows the register/memory pattern follows during the Register/Immediate type instruction execution. And at last figure 6 depicts the overall combined path flow for all type of instruction execution data path.

A. Fetch Data path Flow

Figure 3 shows the Fetching of an instruction from memory. The PC (Program Counter) always indicates the location of the instruction to execute. The location is basically an Internal ROM where all instruction (to be executed) are stored at different locations. Once the PC fetches an instruction from its current value, the PC address automatically advances by 4. The order of the PC can be changed by the occurrence of the instruction stored in the memory block.

B. Register Type Data path Flow

The register type instruction(s) is the instruction involved with the data to be communicated to and from the internal (general purpose registers)/external registers (RAM). After being fetched, the instruction is bifurcated into fields i.e. Opcode (Store/shift/Add etc.), source and destination register location/data with some controlling signals named register write or read registers. Then the data or value of the specified register location will propagate through ALU for different operations to be performed.
C. Load/Store Datapath Flow

Figure 5 shows the data path flow for the immediate instructions to load and store at various location, these locations may be any general purpose or any memory locations that can be addressed randomly.

D. Multiple Instruction Datapath Flow

![Fig.6. processing paths for all instructions](image)

Table 1. Action for all instructions

<table>
<thead>
<tr>
<th>Step</th>
<th>R-Type Actions</th>
<th>Memory reference Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>IR = Memory[PC]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PC = PC + 4</td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td>A = Reg(IR[25-21])</td>
<td>B = Reg(IR[20-16])</td>
</tr>
<tr>
<td>Execution</td>
<td>ALUOut = A op B</td>
<td>ALUOut = A = Sign_extend(IR[15-0])</td>
</tr>
<tr>
<td>Memory</td>
<td>Reg[IR[15-11]] = ALUOut</td>
<td>Load MDR = Memory[ALUOut]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Or Store: Memory( ALUOut) = B</td>
</tr>
<tr>
<td>Write Back</td>
<td></td>
<td>Load Reg(IR[20-16]) = MDR</td>
</tr>
</tbody>
</table>

Table 1 shows all actions necessary for the execution of an instruction (of any type) during each stage(s). To differentiate the type of instruction we had design one control unit which takes care of overall operation execution cycle.

E. Control Unit Design

The design of control unit has been divided into various states through which it is going to generate the control signals for other components in the design, based on the current state and on the instruction code. Following are the control signals:

- **MemRead**: if 1, read from memory;
- **MemWrite**: if 1, write to memory;
- **RegDst**: if 1, the destination number for the Write register comes from the Rd field; if 0, it comes from Rt field;
- **RegWrite**: if 1, the general-purpose register selected by the Write register number is written with the value of the Write data input;
- **AluSrcA**: (ALU Source A) if 1, the operand is A register; if 0, the operand is PC;
- **MemtoReg**: if 1, it comes from Memory data register (MDR) and if 0, the value fed to the register file Write data input comes from ALUOut;
- **IRWrite**: if 1, write instruction is performed in IR;
- **PCWrite**: if 1, update the PC;

The control unit is basically the part of our 32-bit asynchronous processor. The complete designs of processor architecture have been distributed into small units. Proposed design has various sub modules which are basically named as ALU control unit, RAM / ROM blocks, PC unit, Shift, Add, Multiplier units for various arithmetic and logical instruction. In the end, interfacing module for all the sub units is designed. As each unit has their own significance, the major role played by the control unit is to command/control the overall operation. Hence more emphasis has been given to this unit.

F. Processor Architecture

This section deals with the complete internal structure of our 32-bit Asynchronous Processor. All the various stage(s) required for an instruction processing is been identified by the call out(s) box(es). Figure 7 shows the internal architecture of Asynchronous processor. In this processor, fetching the instruction pointed by the Program counter goes to the next unit called decoder which generates the different values of the memory location, as per the instruction fetched from the previous unit. During this control unit has been designed in order to synchronize the various other units such as ALU, data memory or general purpose registers to properly execute the desired instruction. There are other units named as ALU, data memory, and some multiplexers to complete the execution cycle. The entire major units have been discussed previously.
III. FEATURES OF NOVEL PROCESSOR

The 32-bit asynchronous processor has the capability of handling all types of instructions i.e. I-Type, R-Type, Jump Instructions and also multiply instructions packet. The multiplied result is stored until is needed irrespective of other instructions follows.

The proposed processor is using separate memory for instructions and Data. The capacity of instruction memory i.e. ROM is of 8192*32 in which 8192 are representing the locations where instructions are to be stored with 32-bit data. The structures of instructions are as per ISA (instruction set Architecture). For all stages, there is only one clock cycle needed, while the data memory has the capacity of 64K. Both memories are functioning in falling pulse. The other pulses are used for developing the necessary functions just like pipelining in order to make our processor core faster and much flexible.

All I-type instructions are part decoded in the first stage and all R-type instructions are part decoded in ALU control unit. This reduces the complexity in main control unit. The complete processor core is designed in Xilinx ISE 14.x tool.

The so far studied processor architectures do not contain all instruction in a single architecture especially (Jump & Multiply instructions). Also while designing the whole processor core much more attention is given to design the proposed processor in such a way to optimize the core for much better results in terms of Area, Power or Delay.

IV. DESIGN IMPLEMENTATION

The complete processor has been implemented on the Xilinx ISE tool. All the coding have been done in VHDL and simulated or verified by the XILINX ISIM. The following windows come from Xilinx simulator for various instructions Stored in Instruction memory. Figure 8 shows the project settings i.e. the target family Virtex-6, with device name XC6Vlx240T.

A. Instruction Memory Window

Figure 9 shows the various instructions written into internal memory. From which PC fetches the instruction to be executed. In figure 10 the table shows the complete structure of the mnemonics for asynchronous processor instruction. The order of instruction execution can be changed as per the designers or consumer requirement. Although once written in Instruction memory the order of
the execution of instruction will be as per the order of the
mnemonics appears. The instruction memory is also
known as code memory /ROM where all the instructions
are going to be stored permanently. The overall execution
of any instruction is always been started by the
code/instruction memory. In the below figure we have
shown some instruction mnemonics although we can
store /provide various instruction up to 32 KB i.e. 8192
memory location available and on each location one can
store up to 32 bit of data. Also, we are having 32*32
register for temporary storage of the output or any general
purpose work. This is shown in figure 10.

V. SIMULATION RESULTS

The simulation is carried out on Xilinx ISIM tool and
results are shown in this section. Figure 12 shows the
result with initial conditions i.e. Rst=’0’. So that the
processor comes into its initial state with the initial values
in all types of the registers.

B. Register Bank Window

Figure 11 shows the internal register bank
arrangements of the designed processor. By default, all
the values in registers are (in 32-bit Binary)
"0000000000000000000000000000000000"
We kept some
values during the operation of addition and subtraction so
that we never get zero as an output. There are thirty-two,
32-Bit registers to temporary hold the data. These
registers can be used in any type of addressing modes i.e.
Immediate, Register Direct or Indirect addressing modes.
The values in the corresponding registers will
automatically update once the instruction will be fetched
and executed.

Figure 13 & 14 shows the simulation result for loading
of a number from memory to a register. The instruction
for this is 8F890064
(100011111100010010000000110100b), where
100011b is the Opcode, 11100 is source location, 01001b
is Destination register location and rest is the immediate
address from where data to be read. The instruction
works as:
Rt = mem [rs(data)+64h] where rt= 1001b or 9(h/d), Rs =11100b or 1ch or 28d in register bank and having zero value. So the resultant address from where data to be fetched would be 64h+0h= 64h. The content of this 64h will be loaded into the 9 register in register bank.

**Hex code: 8F890064, LW $s1, 100($s2) Load word**

Figure 13. Register bank updated Value

Figure 14. Load data from memory

Figure 15, shows the simulation for the instruction to store some number from register to memory location. “AF890060”, is the mnemonics for this instruction. This will perform the storing the number from the specified location i.e. 1001 (9) to the address hold by the rs and constant value. The yellow marker shows this happening in the simulation cycle.

**Hex Code: AF890060 SW $s1, 96($s2) Store word**

mem [rs+60h]= rt where rt= 1001b or 9(h/d), Rs =11100b or 1ch or 28d in register bank and having zero value. Since Rt holds the value 30303030H, now this value is going to be written in the memory location.

Figure 16. Addition of values from register bank
Figure 16 shows the simulation output of the Addition operation of the values stored into internal register bank. The opcode for this instruction is 2538820. The register location(s) are marked by blue color in the simulation diagram.

The above figure shows the complete summary of our implemented design, which carries the information(s) about the number of resources i.e. inferred gates/FFs/dedicated LUTs, has been used by the implemented designed. Through this information, we can also put some extra efforts in the direction of optimizing our design for the best possible result.

A. Overall processor Simulation Results

Figure(s) 19 and 20 are showing the overall processor results. While initializing the processor with \( \text{rst} = 1 \)''. Our processor starts fetching the instruction from ROM/Instruction memory which contains the Store, Load word and Addition and Subtraction operation. Due to large simulation signals we had divided our simulation windows in two halves. The first half i.e. figure 19 shows the signals like clk, rst, Aluop, Alusw, AlusrcA, IRwrite etc. These signals are basically coming from the control Unit to bind the overall instruction execution. For simplicity we had taken out Bus\(_{r}(31:0)\) for the final output. Although we can verify these data into the internal memory location(s) or Register Memory Window.
The instr(31:26) signal shows the process of opcode fetching from Instruction memory, it starts from 8F, i.e. opcode for the loading of data into the memory, the next is AF for storing the data from memory, then 02 for Addition and so on. On every clock pulse the new instruction will be fetched and update automatically the instr(31:26) signal. This signal in simulation waveform is highlighted by the color RED.

The immed_addr(15:0) signal shows the immediate value for the use of ALU operations. This signal is highlighted by Yellow color in simulation waveform. The intermediate Bus (31:0) is showing the data output from the ALU after completion of any instruction. This is highlighted by the color white. At last the output from A showing by ALU_OUT(31:0) highlighted by the color Purple.

VI. CONCLUSION

The 32-bit fully functional asynchronous processor has been designed using VHDL. A fully asynchronous processor has been implemented that is comprised of five stages. The work can be potentially improved by reducing I and R-type instructions. The functional simulation shows that proposed processor executes all the various instructions efficiently. The proposed design being an open core is more advantageous as compared to the existing commercial microprocessor for better understanding of internals of the asynchronous microprocessor.

FUTURE SCOPE

The design of this 32-bit asynchronous microprocessor implemented using FPGA can be further optimized to reduce its power and area by using any of the following optimization techniques Technology mapping & Logic Optimization.

REFERENCES


Authors’ Profiles

Ms. Archana Rani Bhatia is a Ph.D. A scholar from Manav Rachna International University, Faridabad. Had completed Post Graduation in Electronics Product Design and Technology from Punjab Engineering College Chandigarh in 2008 and did Graduation in Electronics and Communication Engineering, with sound working experience over 6.6 years in various electronic works, related to Teaching, Research & Industry side.

Prof. (Dr.) Naresh Grover did his B.Sc (Engg.) in 1984 and M.Tech in Electronics and Communication Engineering in 1998 from REC Kurukshetra (Now NIT Kurukshetra). He has a rich experience of 33 years in academics. He has authored two books on Microprocessors and is a co-author of a book on Electronic Components and Materials. His core area of interest is Microprocessors and Digital System Design. Presently he is Dean-Academics at Manav Rachna International University, Faridabad.


Copyright © 2018 MECS I.J. Information Engineering and Electronic Business, 2018, 1, 39-47