Chapter 4
System Architecture

4.1 Introduction

The system under consideration for this Ph.D. work is a prototype model of 32-bit processor unit, which is designed considering the base of RISC principle and is targeted for implementation on Xilinx SPARTAN – 3E FPGA device. The embedded processors found in everyday appliances such as cell phones, personal digital assistants, and handheld game systems, are far more powerful compared to earlier ones. The embedded processor is a processor that has been embedded into a target device such as FPGA and it can be programmed to interact with different pieces of hardware. It is essential to have these embedded processors a low power processor as it becomes a part of everyday life and the expansion of battery life time, is an important aspect for the mobility of modern electronic gadgets, it also increases the device reliability [74] [75]. Embedded processors are accepted widely because they are small in size and are cost effective to fabricate, also very flexible as one can change the application or the specification very easily just by changing the software only. Low power design strategies, implemented at various abstraction levels of the system, also reduces the heat dissipation to a greater extent, which in turn reduces the packaging cost of the system as cooling arrangement may be simpler [76] [77].

This chapter includes the design of 32-bit processor using RISC principle with 4-stage pipeline [78] which allow the simpler implementation using the load / store mechanism and supports the predefined instruction set [79].

4.2 Processor Architecture

The processor to be discussed here is 4-stage pipelined processor with instruction and data memory within the FPGA chip and 32-bits are the width of instructions and data. Due to the
difference in time taken to access a register as compared to a memory location, it is much faster to perform the operation on-chip register rather than memory. To eliminate the latency of memory operations, MIPS (Microprocessor without interlocked pipeline stages) processor uses the load/store architecture where the access to memory is only through load and store instructions. There are total 16 general purpose registers identified as R0 to R15 and the size of each one is 32 – bit. Most of the instructions have two operands, one is a register operand (RD for destination) and the other is a register operand (RS for source) or an immediate or a direct or an indirect address in case of load store instruction. The operand result is written back to RD. The RD and RS registers can be one of the 16 general purpose 32 – bit registers i.e. from R0 to R15. Other special function registers discussed below are also designed which are essential for the pipeline operations.

**Program counter (PC):** It is a 32 – bit long register, holds the address of the next instruction which is to be fetched from the memory during the next clock cycle and to be executed. Normally PC is incremented by one during each clock cycle unless a branch instruction executed. In case of branch instruction is encountered the PC will jump to the branch offset address and start pointing to fetch the next instruction to be executed [80].

**Instruction Register (IR):** It is a 32 – bit register used by CPU itself. The instruction pointed by PC and fetched from the memory is loaded into this register. IR is not programmable and cannot be accessed.

**Two Registers (A and B):** The size of both the registers is 32 – bit, and used to hold two source operands.

**Register (C):** It is 32 – bit register used to hold the result of the operations, it is a destination operand.

**Single bit register (Z):** It is used to hold the zero flag for conditional branching. The branch instructions will be evaluated using this register.

The detailed architecture of a processor is given in Figure 4.1, it explains the 4 – pipeline stages of the processor, which are Instruction Fetch (IF), Decode and Operand Fetch (DC), Execution or Memory Access (EX) and Write Back (WB). All these stages [81]can be detailed hierarchically as shown below.

Pipelining structures are used in the processors to permit overlapping execution of multiple instructions within the same circuitry. This system is divided into the number of stages which
includes instruction decoding, arithmetic, registers fetching stages, and also has a pipeline structure, where one instruction is processed in each stage at a time.

Along with the pipeline structure, the processor architecture also incorporates the data forward unit and the hazard detection unit to maintain the proper data flow through the pipeline stages. Each of the stage of the pipeline along with the data forward and hazard detection unit are described in detailed as follows:

4.2.1 IF Stage

This stage consists of Program counter (PC), Instruction Memory and the Branch detection Unit. In this stage, the content of PC is sent to ROM location from which the next instruction to be executed is to be fetched. At the same time the PC predictor predicts the next instruction. If in previous instructions decode stage a branch taken is detected then, according to the sign, immediate value is incremented to or decremented from PC else PC is incremented by 1.

4.2.2 DC Stage

In this stage the instruction is now in the instruction register to be decoded and corresponding operand is to be fetched. The control unit generates the control signals, which are utilized for proper synchronization and operation of the overall system. The various signals decoded by the instruction decoder, which can be used for read and write operations for register bank and
program memory, which are as described in the following section. It also generates the signals which decide the usage of multiplier and ALU, and also generates the flags used by branch unit and produces the clock gating signals for ALU control. Instruction decoder decodes the following signals:

Figure 4.1: Detailed Architecture of 4 – Stage Pipelined Processor Under Consideration
• Btaken: 1 bit signal Branch taken to indicate that current instruction is one of the three
branch instruction.
• Regwrite: 1 bit signal indicating that current instruction will write to a destination register.
  Nop, store and all three branch instruction don’t generate this signal. For all other
  instructions this bit is decoded to logic 1.
• Load: 1 bit signal indicating that current instruction is a load instruction and data will be
  loaded from data memory to the destination register.
• Store: 1 bit signal indicating that current instruction is a store instruction and data will be
  stored to data memory from the source register.
• Op: 6 bit opcode is decoded in to this register. Opcode is detected to be either Mov, Add,
  Sub, Mul, Or, And, XOR, Ror, Rol, Srl, Sll, Inc, Dec, Cmp and Clr.
• a_depen: It’s a 1 bit destination register dependency signal indicating that current
  instruction destination register is same as the previous instruction source or destination
  register. It is used by ALU operand A as selection signal; and if it go high, the data of
  register C is selected and forwarded, otherwise, the data of register A is selected.
• b_depen: It’s a 1 bit source register dependency signal indicating that current instruction
  source register is same as the previous instruction source or destination register. It is used
  by ALU operand B as selection signal; and if go high, the content of register C is selected
  and forwarded, otherwise, the data of register B is selected.
• Immed: This 1 bit signal indicates that the current instruction is one of the instructions that
  operates on the immediate data using immediate addressing mode.
• Immed_data: This 32 bit register holds the 30 bit immediate value for the instruction
  involving operations on immediate data or 5 bit of shift or rotate for the two shift and two
  rotate instruction or 32 bit branch data for the three branch instruction.
• Sign: As Sign is a bit controllable signal, it indicates whether the branch will cause the
  program counter to be incremented or decremented by the immediate value.
• Ra: Destination register operand is decoded into this 4 bit register.
• Rb: Source register operand is decoded into this 4 bit register.

Data dependencies are detected in this stage so accordingly data forwarding can be done from
write back stage to execute stage. Branch prediction is also done in this stage. For the data
dependency, consider the following sequence of operations of pipeline executions mentioned
as I1, I2, and I3.
I1: ADD R1, R2 ; R1 ← R1 + R2
I2: SUB R3, R1 ; R3 ← R3 - R1
I3: SUBI R1, 1 ; R1 ← R1 - 1

In case of instruction I2, it reads R3 and R1 from the register file in the Decode and Operand Fetch stage, and writes them to A and B registers respectively and at the same time, the instruction I1 add R1 and R2 and write it to C; then store the sum to R1 in the next stage. Hence, I2 will receive the previous data from register file and write it to B. If I2 uses it to perform subtraction, it will result into wrong output. But when I3 reading R1, there will not be any trouble. The remedy is to insert NOP instruction between I1 and I2 to introduce delay for the execution of I2, but the performance will be affected. Hence, a data path is to be designed in a way that it should solve this problem and a dependency detection block should be able to identify the dependencies if any. Once the data dependency is detected, the source data required for ALU operation is passed from C via multiplexer, instead of from A or B. The data dependencies occurs under certain conditions, which are like (a) the operand A / B of the current instruction is a register operand (cRD / cRS), (b) the result of the previous instruction will be written into register file (pRD) and (c) cRD/cRS and pRD are the same register.

The detection of dependency is done in EX stage. In this processor design, we do this operation in DC stage, and make use of pipeline registers to transfer to EX stage. This arrangement offers few advantages, which are (1) The dependency detection in DC stage will reduce the use of number of gates because a common logic can be shared with other decode circuits. (2) The time required by EX stage will be shortened because the signals a_depen and b_depen are made available immediately at the beginning of EX stage. For the control dependency, we use a delay branch method. In the proposed processor design, the branch target address is evaluated in DC stage and it introduce one delay cycle for which an additional adder is required for address evaluation. This technique is implemented in this work to achieve the optimization by rearranging the instruction codes described in detail in Chapter 5.

Operation fetch module fetches the data from the register file corresponding to source and destination operand.
• Sig_da: This 32 bit register holds the value of register pointed by destination operand.
• Sig_db: This 32 bit register holds the value of register pointed by source operand.
• B_select: This 32 bit register holds the value of register pointed by source operand or 32 bit immediate value depending on the status of immed signal.

4.2.3 EX stage

In the execution stage depending on the instructions either data is fetched from the data memory or stored into it or an ALU operation is performed. This stage includes ALU, ALU control Unit and Multiplier.

4.2.3.1 Data memory access

Block RAM of Xilinx is used as data memory. For writing data to memory, address is provided along with data, but the data reading operation has latency of one clock. So when there is load instruction to load data from memory to register, RAM address is generated one clock cycle earlier. Also depending on the dependency signal for RAM address, address is given by b_select or forwarded through c_forward, it depends on the status of b_depen signal. Similarly data to be written to memory is either given by sig_da or c_forward depending on a_depen signal if there is dependency between two consecutive instructions.

4.2.3.2 ALU

ALU is responsible for all arithmetic and logic operations that take place within the processor. These operations can have one or two operands, and these values are coming either from the registers or may be immediate value from the instruction directly. Either A or B register is selected according to the a_depen and b_depen signals and given to ALU. ALU then performs operation depending on the opcode and generate the result into register C. Zero flag is generated if the result in C is zero. A complete architecture of ALU unit is shown in Figure 4.2.
4.2.4 WB stage

During this stage the result generated by the instruction is written back into the one of the general purpose registers. In this stage the regwrite signal pipelined to this stage is checked for 1, if it is one indicating that destination register is to be updated. Thus the value in C register is updated to the corresponding destination register out of 16 general purpose registers.

4.3 Instruction Set Formation
The set of instructions is interpreted directly by the CPU. These instructions are encoded as bit strings in memory and are fetched and executed one by one by the processor. They perform primitive operations such as “add 2 to register i1”, “store contents of R6 into memory location 0xFF32”. This processor architecture supports mainly three types of the instructions such as (a) register type, for which both the operands are registers; (b) the immediate type, which consists of register as one of the operands and an immediate value as another operand; and the third one is (c) branch type instructions. The instruction formats for all these types of instructions are as given below. The format length is 40-bit.

(a) Register Type: Format for all instructions where both operands are registers. It includes the instructions such as
add, sub, mul,
or, and, xor, move,
load, store,
rotate right, rotate left, shift right, shift left.

(b) Immediate Type: Format for all instructions where one operand is register and other is an immediate data it includes the instructions such as
add, sub, mul,
or, and, xor,
move, load, store,
rotate right, rotate left, shift right, shift left.
Format for instructions given below which operates on a following register.

Increment, decrement,
Clear, Complement

<table>
<thead>
<tr>
<th>6 bits opcode</th>
<th>4 bits Operand</th>
<th>30 bits All 0’s</th>
</tr>
</thead>
</table>

(c) Branch Type: Below is the instruction format for the three predefined branch instruction and two update instructions.

<table>
<thead>
<tr>
<th>6 bits opcode</th>
<th>2 bits both 0</th>
<th>32 bits branch or update value.</th>
</tr>
</thead>
</table>

Following Table 4.1 summarizes all the instruction supported by this processor. It is not a complete instruction set for this processor, only a part of the complete instruction has been developed to deal with this processor to carry out the work in this thesis. In list op1, op2 and op are of 4 – bits indicating the register operand. The immediate and direct address is 30 bit value and the branch address and update are 32 – bit value.

Table 4.1: Summary of All the Instructions Supported by this Processor

<table>
<thead>
<tr>
<th>No</th>
<th>Instruction</th>
<th>Operation</th>
<th>Opcode</th>
<th>Instruction Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Nop</td>
<td>No operation</td>
<td>000000</td>
<td>(000000)(All 0’s)</td>
</tr>
<tr>
<td>2</td>
<td>Add</td>
<td>Add operand 2 to 1</td>
<td>000001</td>
<td>(000001)(op1)(op2)(all 0’s)</td>
</tr>
<tr>
<td>3</td>
<td>Sub</td>
<td>Subtract operand 2 to 1</td>
<td>000010</td>
<td>(000010)(op1)(op2)(all 0’s)</td>
</tr>
<tr>
<td>4</td>
<td>Mul</td>
<td>Multiply operand 2 to 1</td>
<td>000011</td>
<td>(000011)(op1)(op2)(all 0’s)</td>
</tr>
<tr>
<td>5</td>
<td>Or</td>
<td>Oring operand 2 to 1</td>
<td>000100</td>
<td>(000100)(op1)(op2)(all 0’s)</td>
</tr>
<tr>
<td>6</td>
<td>And</td>
<td>Anding operand 2 to 1</td>
<td>000101</td>
<td>(000101)(op1)(op2)(all 0’s)</td>
</tr>
<tr>
<td>7</td>
<td>Xor</td>
<td>Xoring operand 2 to 1</td>
<td>000110</td>
<td>(000110)(op1)(op2)(all 0’s)</td>
</tr>
<tr>
<td>8</td>
<td>Mov</td>
<td>Move operand 2 to 1</td>
<td>000111</td>
<td>(000111)(op1)(op2)(all 0’s)</td>
</tr>
<tr>
<td>9</td>
<td>Ror</td>
<td>Rotate right operand 1 by amount specified in operand 2</td>
<td>001000</td>
<td>(001000)(op1)(op2)(all 0’s)</td>
</tr>
<tr>
<td>10</td>
<td>Rol</td>
<td>Rotate left operand 1 by amount specified in operand 2</td>
<td>001001</td>
<td>(001001)(op1)(op2)(all 0’s)</td>
</tr>
<tr>
<td>11</td>
<td>Slr</td>
<td>Shift right operand 1 by amount specified in operand 2</td>
<td>001010</td>
<td>(001010)(op1)(op2)(all 0’s)</td>
</tr>
<tr>
<td>12</td>
<td>Sll</td>
<td>Shift left operand 1 by amount</td>
<td>001011</td>
<td>(001011)(op1)(op2)(all 0’s)</td>
</tr>
<tr>
<td></td>
<td>Operation</td>
<td>Specification</td>
<td>Instruction Code</td>
<td>Comment</td>
</tr>
<tr>
<td>---</td>
<td>-----------------------------------</td>
<td>----------------------------------------</td>
<td>------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>13</td>
<td>Load</td>
<td>Load value to register pointed by operand 1 from memory location pointed by operand 2</td>
<td>001100</td>
<td>(001100)(op1)(op2)(all 0’s)</td>
</tr>
<tr>
<td>14</td>
<td>Store</td>
<td>Store value from register pointed by operand 1 to memory location pointed by operand 2</td>
<td>001101</td>
<td>(001101)(op1)(op2)(all 0’s)</td>
</tr>
<tr>
<td>15</td>
<td>Addi</td>
<td>Add 30 bit immediate value to operand 1</td>
<td>001110</td>
<td>(001110)(op1)(immediate)</td>
</tr>
<tr>
<td>16</td>
<td>Subi</td>
<td>Subtract 30 bit immediate value to operand 1</td>
<td>001111</td>
<td>(001111)(op1)(immediate)</td>
</tr>
<tr>
<td>17</td>
<td>Muli</td>
<td>Multiply 30 bit immediate value to operand 1</td>
<td>010000</td>
<td>(010000)(op1)(immediate)</td>
</tr>
<tr>
<td>18</td>
<td>Ori</td>
<td>Or 30 bit immediate value to operand 1</td>
<td>010001</td>
<td>(010001)(op1)(immediate)</td>
</tr>
<tr>
<td>19</td>
<td>Andi</td>
<td>And 30 bit immediate value to operand 1</td>
<td>010010</td>
<td>(010010)(op1)(immediate)</td>
</tr>
<tr>
<td>20</td>
<td>Xori</td>
<td>Xor 30 bit immediate value to operand 1</td>
<td>010011</td>
<td>(010011)(op1)(immediate)</td>
</tr>
<tr>
<td>21</td>
<td>Movi</td>
<td>Move 30 bit immediate value to operand 1</td>
<td>010100</td>
<td>(010100)(op1)(immediate)</td>
</tr>
<tr>
<td>22</td>
<td>Rori</td>
<td>Rotate right operand 1 by amount specified by 5 bit immediate</td>
<td>010101</td>
<td>(010101)(op1)(immediate)</td>
</tr>
<tr>
<td>23</td>
<td>Roli</td>
<td>Rotate left operand 1 by amount specified by 5 bit immediate</td>
<td>010110</td>
<td>(010110)(op1)(immediate)</td>
</tr>
<tr>
<td>24</td>
<td>Slii</td>
<td>Shift right operand 1 by amount specified by 5 bit immediate</td>
<td>010111</td>
<td>(010111)(op1)(immediate)</td>
</tr>
<tr>
<td>25</td>
<td>Slli</td>
<td>Shift left operand 1 by amount specified by 5 bit immediate</td>
<td>011000</td>
<td>(011000)(op1)(immediate)</td>
</tr>
<tr>
<td>26</td>
<td>Loadi</td>
<td>Load value to register pointed by operand 1 from memory location indicated by 11 bit immediate</td>
<td>011001</td>
<td>(011001)(op1)(direct address)</td>
</tr>
<tr>
<td>27</td>
<td>Storei</td>
<td>Store value from register pointed by operand 1 to memory location indicated by 11 bit immediate</td>
<td>011010</td>
<td>(011010)(op1)(direct address)</td>
</tr>
<tr>
<td>28</td>
<td>Bz</td>
<td>Branch if Zero and start fetching instructions from location specified by 32 bit immediate address</td>
<td>011011</td>
<td>(011011)(2 bit 0’s)(Branch Addr)</td>
</tr>
<tr>
<td>29</td>
<td>Bnz</td>
<td>Branch if not Zero and start fetching instructions from location specified by 32 bit immediate address</td>
<td>011100</td>
<td>(011100)(2 bit 0’s)(Branch Addr)</td>
</tr>
<tr>
<td>30</td>
<td>Br</td>
<td>Branch and start fetching instructions from location specified by 32 bit immediate address</td>
<td>011101</td>
<td>(011101)(2 bit 0’s)(Branch Addr)</td>
</tr>
<tr>
<td>31</td>
<td>Sr0l</td>
<td>Update R0 register least significant word</td>
<td>011110</td>
<td>(011110)(2 bit 0’s)(update value)</td>
</tr>
<tr>
<td>32</td>
<td>Sr0h</td>
<td>Update R0 register most significant word</td>
<td>011111</td>
<td>(011111)(2 bit 0’s)(update value)</td>
</tr>
<tr>
<td>33</td>
<td>Inc</td>
<td>Increment operand by 1</td>
<td>100000</td>
<td>(100000)(op1)(All 0’s)</td>
</tr>
<tr>
<td>34</td>
<td>Dec</td>
<td>Decrement operand by 1</td>
<td>100001</td>
<td>(100001)(op1)(All 0’s)</td>
</tr>
<tr>
<td>35</td>
<td>Cmp</td>
<td>Complement operand</td>
<td>100010</td>
<td>(100010)(op1)(All 0’s)</td>
</tr>
<tr>
<td>36</td>
<td>Clr</td>
<td>Clear operand</td>
<td>100011</td>
<td>(100011)(op1)(All 0’s)</td>
</tr>
</tbody>
</table>

### 4.4 Sub-modules of Processor
The processor is designed in modular fashion and includes ALU, Register File, Instruction Memory, Data Memory, Decoder, Pipeline registers, and multiplexors as major modules. The four pipeline stages are presented in the Figure 4.1. Following section discusses the major sub-modules incorporated in this processor.

### 4.4.1 ALU Design

The ALU of the processor implemented here performs 14 different operations. A 6-bit signal ALU_op selects the ALU operation shown in Figure 4.2. The 14 operations along with the possible variations defined which are to be performed by this ALU are listed below.

1. MOV (for MOV, MOVI and SR0H)
2. OR (for OR, ORI and SR0L)
3. XOR (for XOR and XORI)
4. AND (for AND and ANDI)
5. SUB (for SUB and SUBI)
6. ADD (for ADD and ADDI).
7. MUL (for MUL and MULI)
8. INC (for INC)
9. DEC (for DEC)
10. CMP (for CMP)
11. CLR (for CLR)
12. ROR (for ROR and RORI)
13. ROL (for ROL and ROLI)
14. SLR (for SLR and SLRI)
15. SLR (for SLL and SLLI)

### 4.4.2 Register File Design

Register is a storage location directly on the CPU, used for temporary storage of small amount of data during processing. In this processor implementation, we have constructed 16 general purpose registers \( R_0 - R_{15} \), each one is 32 bit wide. Hence, four bits are required to use for addressing the register file i.e. \( 16 \times 32 \) bits having two read ports and a write port. It can be implemented with 16, 32-bit registers and a pair of 16-to-1 multiplexors and each one is of 32 bits wide for read ports and uses a 4-to-16 decoder for write control.
4.4.3 Data Memory Design

Memory array is randomly accessible to memory bytes, each one identified by a unique address. Flat memory models, segmented memory models, and hybrid models exist which are distinguished by the way the locations are referenced and potentially divided into sections. As proposed processor development is based on Harvard architecture, there must be provisions for separated instruction memory module and data memory module. Data memory has 2048 x 32 bits. We use Xilinx block RAM modules to build on-chip memory. For this implementation the Xilinx coregen block memory generator is used for generating RAM. It made memory access possible once at each cycle. The load and store instructions are used to access this module.

4.4.4 Instruction Memory Design

This unit contains the instructions that are executed by the processor. Instruction memory has 2048 × 40 bits. For the instruction memory design, we have used Xilinx coregen block memory generator. The ROM is initialized to a known value with the coe file at the time of ROM generation. Coe file hold the instruction that are dumped into ROM at the time of core generation.

4.4.5 Instruction Decoder

The internal structure of instruction decoder is made up of multiplexers, comparators and the logic gates. Figure 4.4 explain the detailed internal architecture of instruction decoder, the operational role of the multiplexer is defined in Figure 4.5 and the useful signals which are utilized during the operations such as instruction fetch execution and write back as shown in Figure 4.6.
Figure 4.4: Detailed Internal Architecture of Instruction Decoder
Figure 4.5: Internal Architecture of Multiplexer
4.4.6 Control Unit Design

The control unit produce all the necessary control signals which are to be utilized for the synchronization among all the components of the processor. This unit also provides the signals that control all the read and write operations of the register file and also for decision about when to use the multiplier and when to use the ALU. It also generates appropriate branch flags that are used by the Branch Decide Unit. In addition to all these, this unit provides clock gating signals for the ALU control and the Branch Adder module. In short, this unit generates all the control signals for controlling all the data path activities. Summary of all the control signals is given in Table 4.2.

Table 4.2: Summary of Control Signals

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Signal</th>
<th>Description of Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>BTAKEN</td>
<td>If branch is taken, BTAKEN go high to select the branch address for instruction to be fetched in the next clock cycle.</td>
</tr>
<tr>
<td></td>
<td>Branch Taken</td>
<td></td>
</tr>
<tr>
<td>02</td>
<td>RA&lt;3:0&gt;</td>
<td>Used to point out the number of the destination register RD, but RD can also be source 1 register (RD = RD op RS); almost all the instructions keep the number of RD in a fixed position for the instruction format, but for SR0L and SR0H instructions, the number</td>
</tr>
</tbody>
</table>

Figure 4.6: Diagram of Instruction Decoder with all relevant Signals
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>RB&lt;3:0&gt;</td>
<td>It shows the register number of RS which is the source 2 register. The Most of the instructions put the number of RS in a fixed position, but few instructions use it as immediate data in that position.</td>
</tr>
<tr>
<td>04</td>
<td>IMMED Immediate Selection</td>
<td>When the source 2 operand is an immediate, IMMED go high for the selection of immediate, not register operand.</td>
</tr>
<tr>
<td>05</td>
<td>IMMED_DATA&lt;31:0&gt; Immediate</td>
<td>When it is 32-bit immediate data. Depending upon instructions, it can be produced with different extension techniques. The IMMED_DATA&lt;31:0&gt; is used for the source 2 operand of ALU operations and also for the branch target address calculation.</td>
</tr>
<tr>
<td>06</td>
<td>OP&lt;5:0&gt; ALU Operation Control</td>
<td>This signal will be generated based on instructions.</td>
</tr>
<tr>
<td>07</td>
<td>ADEPEN (A Dependent)</td>
<td>It is the selection signal for ALU operand A, if high; the forwarding data of register C is selected, else the data of register A is selected.</td>
</tr>
<tr>
<td>08</td>
<td>BDEPEN (B Dependent)</td>
<td>It is the selection signal for ALU operand B; if high, the forwarding data of register C is selected, else the data of register B is selected.</td>
</tr>
<tr>
<td>09</td>
<td>STORE Store</td>
<td>It is called memory-write control signal, if it go high, a memory write operation performed.</td>
</tr>
<tr>
<td>10</td>
<td>LOAD Load</td>
<td>When this signal LOAD is high, the register C will be written with the loaded data from data memory.</td>
</tr>
<tr>
<td>11</td>
<td>ZERO_WRITE Zero Register Write</td>
<td>When ZR WRITE is high, zero flag register will be updated.</td>
</tr>
<tr>
<td>12</td>
<td>RA2&lt;3:0&gt;</td>
<td>It is the same signal as RA&lt;3:0&gt; of DC stage and is pipelined to retain the number of destination register for the purpose to be written in the write back stage.</td>
</tr>
<tr>
<td>13</td>
<td>REGWRITE (Register Write)</td>
<td>It is the register-write control signal. If it goes high, then the register C content will be shifted into register file.</td>
</tr>
</tbody>
</table>

All of the control signals are generated within the DC stage. However, few of them are actually utilized in EX and WB stages. To handle this situation, pipeline registers are used to assign a proper signal to the corresponding stages. In our processor design the control unit is divided into five blocks and each block generates the relevant signal. These blocks are described as follows with related signals.

- Branch block generates BTAKEN,
- Register address block generates RA<3:0> and RB<3:0>,
- ALU control block generates OP<5:0>, ZERO_WRITE, and REGWRITE,
- Immediate block generates IMMED and IMMED_DATA <31:0>,
- Dependent block generates A_DEPEN and B_DEPEN.
4.5 Multiplier Unit & Its Logic

For multiplication to be done in the Spartan 3e FPGA, the dedicated multipliers are used, it provides 4 to 36 dedicated multiplier blocks per device. The multipliers are located together with the block RAM in one or two columns depending on device density. The multiplier blocks primarily perform two’s complement numerical multiplication and it can also perform some less obvious applications, such as simple data storage and barrel shifting. Logic slices also implement efficient small multipliers and thereby supplement the dedicated multipliers. Each multiplier performs the principle operation \( P = A \times B \); where ‘A’ and ‘B’ are 18 – bit words in two’s complement form, and ‘P’ is the full-precision 36 – bit product, also in two’s complement form. The 18 – bit inputs represent values ranging from \(-131,072_{10}\) to \(+131,071_{10}\) with a resulting product ranging from \(-17,179,738,112_{10}\) to \(+17,179,869,184_{10}\).

Wider multiplication operation can be possible by combining the dedicated multipliers and slice-based logic in any viable combination or by time-sharing a single multiplier, perform unsigned multiplication by restricting the inputs to the positive range. Tie the most-significant bit low and represent the unsigned value in the remaining 17 lesser-significant bits. Figure 4.7 show that each multiplier block has optional registers on each of the multiplier inputs and the output. The registers are named AREG, BREG and PREG and can be used in any combination. The clock input is common to all the registers within a block, but each register has an independent clock enable and synchronous reset controls making them

![Figure 4.7: Main Features of Multiplier Block](image-url)
ideal for storing data samples and coefficients. When used for pipelining, the registers boost the multiplier clock rate, which is beneficial for higher performance applications. The MULTI8X18SIO primitive shown in Figure 4.8 is used to instantiate a multiplier within a design. Although high-level logic synthesis software usually automatically infers a multiplier, adding the pipeline registers might require the MULTI8X18SIO primitive. Connect the appropriate signals to the MULTI8X18SIO multiplier ports and set the individual AREG, BREG, and PREG attributes to ‘1’ to insert the associated register, or to 0 to remove it and make the signal path combinatorial.

The MULTI8X18SIO primitive has two additional ports called BCIN and BCOUT to cascade or share the multiplier’s ‘B’ input among several multiplier blocks. The 18-bit BCIN “cascade” input port offers an alternate input source from the more typical ‘B’ input. The B_INPUT attribute specifies whether the specific implementation uses the BCIN or ‘B’ input path. Setting B_INPUT to DIRECT chooses the ‘B’ input. Setting B_INPUT to CASCADE selects the alternate BCIN input. The BREG register then optionally holds the selected input value, if required. BCOUT is an 18-bit output port that always reflects the value that is applied to the multiplier’s second input, which is either the ‘B’ input, the cascaded value from the BCIN input, or the output of the BREG if it is inserted. Figure 4.9 illustrates the four possible configurations using different settings for the B_INPUT attribute and the BREG attribute.
The BCIN and BCOUT ports have associated dedicated routing that connects adjacent multipliers within the same column. Via the cascade connection, the BCOUT port of one multiplier block drives the BCIN port of the multiplier block directly above it. There is no connection to the BCIN port of the bottom-most multiplier block in a column or a connection from the BCOUT port of the top-most block in a column.

### 4.6 Clock Distribution Network

In this processor design Spartan 3e xc3s500e FPGA device is used for its implementation and its clock distribution network is utilized for the functionality. The Spartan-3E clock distribution network is as shown in Figure 4.10, it presents a series of low-capacitance and low-skew interconnect lines which is suitable to carry high-frequency signals throughout the FPGA’s blocks. The infrastructure also includes the clock inputs and BUFGMUX clock buffers/multiplexers. The Xilinx Place-and-Route (PAR) software automatically routes high-fan out clock signals using these resources. Clock pins can be connected directly to external clock signals and also to DCMs and BUFGMUX elements. Each Spartan-3E FPGA has 16 Global Clock inputs (GCLK0 through GCLK15) located along the top and bottom edges of the FPGA. Out of which 8 are Right-Half Clock inputs (RHCLK0 through RHCLK7) located along the right edge and 8 are Left-Half Clock inputs (LHCLK0 through LHCLK7) located
along the left edge. Each clock input is also optionally a user-I/O pin and connects to internal interconnect. Some clock pad pins are input-only pins.

Clock Buffers/Multiplexers either drive clock input signals directly onto a clock line (BUFG) or optionally provide a multiplexer to switch between two unrelated, possibly asynchronous clock signals (BUFGMUX). Each BUFGMUX element, shown in Figure 4.10, is a 2-to-1 multiplexer. The select line, S, chooses which of the two inputs, I0 or I1, drives the BUFGMUX’s output signal, O. The switching from one clock to the other is glitch-less, and done in such a way that the output High and Low transition times are never shorter than the shortest High or Low time of either input clock. The two clock inputs can be asynchronous with regard to each other, and the S input can change at any time, except for a short setup time prior to the rising edge of the presently selected clock (I0 or I1).

Figure 4.10: Xilinx SPARTAN – 3E Clock Distribution Network (Courtesy of Xilinx™ Co.)
The BUFG clock buffer primitive drives a single clock signal onto the clock network and is essentially the same element as a BUFGMUX, just without the clock select mechanism. Similarly, the BUFGCE primitive creates an enabled clock buffer using the BUFGMUX select mechanism. The I0 and I1 inputs to BUFGMUX element originate from clock input pins, DCMs, or Double-Line interconnect, as shown in above Figure 4.11. As depicted in Figure 4.10, there are total 24 BUFGMUX elements, which are distributed around the four edges of the device. Clock signals from the four BUFGMUX elements at the top edge and the four at the bottom edge are truly global and connect to all clocking quadrants. The eight left-edge BUFGMUX elements only connect to the two clock quadrants in the left half of the device. Similarly, the eight right-edge BUFGMUX elements only connect to the right half of the device. BUFGMUX elements are organized in pairs and share I0 and I1 connections with adjacent BUFGMUX elements from a common clock switch matrix as shown in Figure 4.11. For example, the input on I0 of one BUFGMUX is also a shared input to I1 of the adjacent BUFGMUX. The clock switch matrix for the left- and right-edge BUFGMUX elements receives signals from any of the three following sources: an LHCLK or RHCLK pin as appropriate, a Double-Line interconnect, or a DCM in the XC3S1200E and XC3S1600E devices.

By contrast, the clock switch matrixes on the top and bottom edges receive signals from any of the five following sources: two GCLK pins, two DCM outputs, or one Double-Line interconnect. The four BUFGMUX elements on the top edge are paired together and share
inputs from the eight global clock inputs along the top edge. Each BUFGMUX pair connects
to four of the eight global clock inputs, as shown in Figure 4.10. This optionally allows
differential inputs to the global clock inputs without wasting a BUFGMUX element. The
connections for the bottom-edge BUFGMUX elements are similar to the top-edge
connections (see Figure 4.11). On the left and right edges, only two clock inputs feed each
pair of BUFGMUX elements.

The clock routing within the FPGA is quadrant-based, as shown in Figure 4.10. Each clock
quadrant supports eight total clock signals, labelled ‘A’ through ‘H’ as shown in Figure 4.12.
The clock source for an individual clock line originates either from a global BUFGMUX
element along the top and bottom edges or from a BUFGMUX element along the associated
edge, as shown in Figure 4.10. The clock lines provide the synchronous resource elements
(CLBs, IOBs, Block RAM, multipliers and DCMs) within the quadrant. The four quadrants
of the device are: Top Right (TR), Bottom Right (BR), Bottom Left (BL) and Top Left (TL).

The outputs of the top or bottom BUFGMUX elements connect to two vertical spines, each
comprising four vertical clock lines as shown in Figure 4.10. At the centre of the die, these
clock signals connect to the eight-line horizontal clock spine. Outputs of the left and right
BUFGMUX elements are routed onto the left or right horizontal spines, each comprising
eight horizontal clock lines. Each of the eight clock signals in a clock quadrant derives either
from a global clock signal or a half clock signal. In other words, there are up to 24 total
potential clock inputs to the FPGA, eight of which can connect to clocked elements in a
single clock quadrant. Figure 4.12 shows how the clock lines in each quadrant are selected from associated BUFGMUX sources. For example, if quadrant clock ‘A’ in the bottom left (BL) quadrant originates from BUFGMUX_X2Y1, then the clock signal from BUFGMUX_X0Y2 is unavailable in the bottom left quadrant. However, the top left (TL) quadrant clock ‘A’ can still solely use the output from either BUFGMUX_X2Y1 or BUFGMUX_X0Y2 as the source. To minimize the dynamic power dissipation of the clock network, the Xilinx development software automatically disables all clock segments which are not in use.

4.7 Data Forwarding and Data Dependency

This unit is responsible for proper data forwarding to ALU and multiplier. The primary function of this unit is to compare the destination register address of the data waiting in the EX and WB pipeline registers to be written back to the register file with the current data needed by the ALU or multiplier and forward the most up-to-date data to these units. By forwarding the data at the appropriate time, this unit makes sure that the pipeline works smoothly and does not stall as a result of data dependencies [82]. All pipeline registers shown below are involved to have smooth flow through different stages of pipeline.

<table>
<thead>
<tr>
<th>IF Stage</th>
<th>DC Stage</th>
<th>EX Stage</th>
<th>WB Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zerowrite</td>
<td>Zerowrite</td>
<td>Zerowrite1</td>
<td></td>
</tr>
<tr>
<td>Regwrite</td>
<td>Regwrite</td>
<td>Regwrite1</td>
<td>Regwrite2</td>
</tr>
<tr>
<td>Ra(3 : 0)</td>
<td>Ra(3 : 0)</td>
<td>Ra1(3 : 0)</td>
<td>Ra2(3 : 0)</td>
</tr>
<tr>
<td>Rb(3 : 0)</td>
<td>Rb(3 : 0)</td>
<td>Rb1(3 : 0)</td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>Load</td>
<td>Load1</td>
<td></td>
</tr>
<tr>
<td>Store</td>
<td>Store</td>
<td>Store1</td>
<td></td>
</tr>
<tr>
<td>Op(5 : 0)</td>
<td>Op(5 : 0)</td>
<td>Op1(5 : 0)</td>
<td></td>
</tr>
<tr>
<td>A_depen</td>
<td>A_depen</td>
<td>A_depen1</td>
<td></td>
</tr>
<tr>
<td>B_depen</td>
<td>B_depen</td>
<td>B_depen1</td>
<td></td>
</tr>
<tr>
<td>Immed</td>
<td>Immed</td>
<td>Immed1</td>
<td></td>
</tr>
<tr>
<td>Immed_data(31 : 0)</td>
<td>Immed_data(31 : 0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hazards are the situations in which a pipeline may produce wrong answers by providing incorrect data. The remedy to this is instructions stalls and/or data forwarding to be used to overcome such pipeline hazards. Hazard detection unit detects such scenario in which correct data forwarding is not possible and stalls the pipeline for one or two clock cycles in order to assure that instructions are executed with the correct data set. If it found that a stall is necessary, it disables the operation in the instruction decode pipeline registers, stops the program counter from incrementing, and clears all the control signals generated by the control unit. There are mainly three types of hazards discussed below.

### 4.7.1 Structural Hazards

Structural Hazards occur when more than one instruction attempt simultaneously to make use of same resources or wrong inputs are given to hardware. Specifically, branch instructions could make use of the same ALU to figure out the target branch address. If the ALU were to use in the decode stage for the same purpose, an ALU related instructions followed by a branch would have seen that both the types of instructions tries to use the ALU concurrently. This conflict is resolved by designing a specialized branch target adder into decode stage. Float value given to integer instruction is also a structural hazard.

### 4.7.2 Data Hazards

These types of Hazards observed when an instruction scheduled blindly and if it tries to make use of data before the data is actually available in the register file. Data dependencies are mainly of three types for which there could be three types of possibilities for data hazards.

1) **Read after Write:**

Suppose instruction 1 writes a value which is used later by the next instruction 2. Instruction 1 must come first otherwise instruction 2 will read the older value instead of the newer one. For example, Instruction J tries to read operand before Instruction I writes it. Caused by Dependences occur in compiler nomenclature.

I: add r1,r2,r3
J: sub r4,r1,r3

2) **Write after Read:**
Suppose the situation is such that an instruction 1 reads a location which is to be later overwritten by instruction 2. It is mandatorily required that the instruction must come first, or it should read the newer updated value instead of the previous one. For example, Instruction J writes operand before Instruction I read it which is called an anti dependence by compiler writers. This results because of reusing the name r1.

I: sub r4,r1,r3  
J: add r1,r2,r3  
K: mul r6,r1,r7

3) Write after Write:

Consider two instructions and both of which tries to write the same location. It is mandatory requirement that instructions must take place as per their original order. For example, Instruction J writes operand before the instruction I writes it, Called an output dependence by compiler writers. This also results due to repetition of name r1.

I: sub r1,r4,r3  
J: add r1,r2,r3  
K: mul r6,r1,r7

4.7.3 Control Hazards

Control Hazards take place because of conditional and unconditional branching and jumps. The pipeline structure based on RISC provides solution for branches in the Decode stage, in which the branch resolution and reoccurrences are two cycles long. For any branch taken, the instruction which is immediately to be executed after the branch should always be fetched from instruction memory. If this instruction is being ignored, there will be an introduction of one cycle per taken branch as penalty, which is quite large. There are four techniques used to solve this performance problem with branches which are named as (1) Predict not taken; (2) Branch likely; (3) Branch delay slot and (4) Branch prediction.

In the proposed processor design we have used branch delay slot scheme to prevent control hazard. After every branch instruction a NOP instruction is there to provide a delay slot in the pipeline. Also this processor as being a fixed pipeline processor has only read after write data dependency. This dependency has been identified in the decode stage and data forwarding is
done from write back stage to execution stage to eliminate this hazard. The data dependency and data forwarding are explained by Figure 4.13.

### 4.8 External Interface

This processor is given two signals as an external input signals which are Clk (System Clock) and reset_n (Active high Asynchronous Reset) and the interface is as shown in Figure 4.14.

### 4.9 Instruction Simulation and Verification

All the instructions supported by the processor are simulated using VHDL coding and verified through the generation of waveforms. Thus the performance of the processor as whole system has been checked, tested and verified by simulating variety of instructions. The overall performance of the CPU is discussed in detailed in the Chapter 5.

### 4.10 FPGA Design Flow
Whole system has been developed by using the VHDL and then implemented into the FPGA. A typical FPGA design flow has been followed and each and every step of the flow is shown in the following Figure 4.15.

![FPGA Design Flow](image)

**Schematic Entry:** The design is entered into a synthesis design system using a hardware description language. The language used for this research work was VHDL and editor used for this purpose is one which is provided by Xilinx Integrated Environment (ISE Version 13.1).  

**Synthesis:** A netlist is generated using VHDL code and Xilinx synthesis tool.  

**Place and Route:** The place process decides the best location of the cells, the best routing strategy for the given design and for desired performance. The route process makes the connections between the cells and the blocks. This process is done by using Xilinx ISE.  

**Configuration:** Creates the PLD configuration data and downloads the configuration data to the PLD (FPGA) and enables the configuration on the PLD to be verified for correctness.  

**Verification:** At each and every step of the design process, the processor architecture has been verified using the software simulation. ModelSim XE – II has been used for simulating the VHDL coding.

### 4.11 Summary of Synthesis Report

Whole architecture has been synthesized for FPGA implementation using the Xilinx ISE – 13.1 suit and the summary of synthesis report has been presentation in Table 4.4.
Table 4.4: Summary of Synthesis Report

<table>
<thead>
<tr>
<th>Devices</th>
<th>Device Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Devices Used</td>
</tr>
<tr>
<td>Number of Slices</td>
<td>2017</td>
</tr>
<tr>
<td>Number of Slice Flip Flops</td>
<td>697</td>
</tr>
<tr>
<td>Number of 4 Input LUTs</td>
<td>3066</td>
</tr>
<tr>
<td>Number of bonded IOBs</td>
<td>2</td>
</tr>
<tr>
<td>Number of BRAMs</td>
<td>9</td>
</tr>
<tr>
<td>Number of MULT18X18SIOs</td>
<td>1</td>
</tr>
<tr>
<td>Number of GCLKs</td>
<td>1</td>
</tr>
<tr>
<td>Number of IOs</td>
<td></td>
</tr>
</tbody>
</table>

Timing Summary

<table>
<thead>
<tr>
<th>Speed Grade</th>
<th>Minimum Period</th>
<th>Maximum Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>23.934ns</td>
<td>41.782 MHz</td>
</tr>
</tbody>
</table>

4.12 Power Estimation Reports of Complete Architecture

The XPower Estimator (XPE) – 11.1spreadsheet is a power estimation tool typically used in the pre-design and pre-implementation phases of a project. XPE assists with architecture evaluation, device selection, appropriate power supply components and thermal management components specific for the targeted application. XPE considers design’s resource usage, toggle rates, I/O loading, and many other factors which it combines with the device models to calculate the estimated power distribution. The accuracy of XPE is dependent on two primary sets of inputs (1) Device utilization, component configuration, clock, enable, and toggle rates, and other information related to project entered into the tool and (2) Device data models integrated into the tool. It is a pre-implementation tool to be used in the early stages of a design cycle or when the RTL description is incomplete. After implementation, the XPower Analyzer (XPA) tool (available in the ISE® Design Suite software) can be used for more accurate estimations and power analysis.

The Summary sheet shown in Figure 4.16 is the default sheet on launch and allows designer to enter all device and environment settings. On this sheet the tool also reports estimated power; rail-wise and block-wise, so one can quickly review thermal and supply power.
distribution required within the design. It also gives summary related to clock, logic, IO, BRAM, DCM and MUL. Important factors in dynamic power calculation are the activity and the load capacitance that needs to be switched by each net in the design. Some of the factors in determining the loading capacitance are fanout, wire length, etc. With clocks typically having higher activity and fanouts, the power associated with clock nets can be significant and also can be reported in a separate worksheet sheet along with many other reports. The consolidated summary of estimation report for the design under consideration is shown in Figure 4.16.

Clock Fanout Column is the number of synchronous elements driven by this clock and for this design the maximum achievable frequency is 54.7 Mhz and clk fanout is 668. Using XPower Estimator – 11.1 one can have the separate sheets for each components such as Logic sheet is used to estimate the power consumed in the CLB resources. The estimated power accounts for both the logic components and the routing. Here two types of information are to be entered (1) Utilization – Enter the number of LUTs, Shift Registers and LUT-based
RAMs and ROMs and (2) Activity – Enter the Clock domain as per the logic and then enter the Toggle Rate the logic is expected to switch and the Average Fanout.

Note: The default setting for Toggle Rate (12.5%) and Average Fanout (3) are based on an average extracted from a suite of customer designs. In the absence of a better estimation for a specific design, Xilinx recommends to use the default setting provided in Xpower Estimator.

In our design total no of flip flops are 659 and LUT’s are 3423 used.

With high switching speed and capacitive load, there will be major contribution to the total power consumption of an FPGA is from switching I/O power. Because of this, it is important to accurately define all I/O related parameters. Using the I/O sheet, the XPE helps to calculate the on-chip and, eventually, off-chip power for the system I/O interfaces. There are three main types of information entered on the I/O sheet (1) IO Settings, (2) Activity and (3) bank voltage levels and voltage standards. In this design only two input pins of the processor are used, which are the clock and the reset pin. FPGA devices have dedicated block RAM resources. The details about the Enable Rate and Write Rate columns used in the Block RAM sheet can be described as (1) Use the Enable Rate to specify the percentage of time each block RAM’s ports are enabled for reading and/or writing. To save power, the RAM enable can be driven low on clock cycles when the block RAM is not used in the design.

![Figure 4.17: Graphical Representation of Estimated Power Requirements for Internal Modules and Effect of Various Parameters on Power Consumption](image)

Enable Rate, together with Clock rate is important parameters that must be considered for
power optimization. (2) The Write Rate represents the percentage of time that each block RAM port performs write operations. The read rate is understood to be 100% – write rate. We have used in all 9 block RAMs of 18K: 4 for ROM and 5 for RAM and only 1 Digital Clock Manage (DCM) is used as there is only one clock source and also present its related electrical parameters and one multiplier is used as there is only one multiplication instruction. Graphs are plotted according to the data given as input and are well described in Figure 4.17, which represents the need of power of each individual module within the system and also provides the information for power need with reference to junction temperature and the voltage supplied to the device. It is noted that power consumption increases with the junction temperature and the increase in the voltage as well.

4.13 Conclusion

This chapter provides the complete details of construction of 4 – stage pipeline processor architecture and also discusses the types of instruction and their construction along with the detailed formation of instruction.

Processor’s architecture design and its instruction formation are thoroughly tested using number of programs and validated through the simulated results in terms of waveform to solidify the design and observed that the correct functionality is achieved and the simulated waveforms are discussed in Chapter 5.

It covers the complete construction of decoder unit and multiplier unit along with many other useful modules such as data forwarding and data hazard detection unit along with the types of various data hazards and provides the remedies for the same.

We also included the complete clock distribution network and the structure of multiplier unit along with the details of all the related signals, as the system is targeted to be implemented up on SPARTAN – 3E FPGA.

At last the power estimation is done using Xpower Estimator -11.1, which will be used for power comparisons to be done in the next chapter.
The following chapter describes the conventional 5 – stage CPU, its power analysis and then its conversion to 4 – stage CPU along with its power analysis and suggests the newly developed power reduction techniques such as Memory Access Stage Removal, Resource Sharing, RAM Addressing Scheme and Clock Gating in order to reduce the dynamic power consumption and are implemented successfully on FPGA. Also the outcomes are analysed and verified in order to lower the overall system power consumption.

These techniques are applied at the hardware design level and analyzed for power requirements. Also the power consumption comparison between conventional 5 – stages CPU with the newly developed low power CPU with 4 – pipeline stages has been made by generating the power reports.