

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | |
|  | | 16-Bit Processor | | | | |  | |
|  |  | | | | | | |  |
|  | | | |  |  | | | |
|  | | | | Jawad AhmedHassan Ali |  | | | |
|  | | | | January 12, 2025—Information and Communication Technology—Prof. Asad Mansoor |  | | | |
|  | | |  | | |  | | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | | |  |  | | |  |
|  | INTRODUCTION | | | | | | |  |
|  |  | | |  |  | | |  |
|  |  | |  | | |  | |  |
|  |  |  | The evolution of digital systems has significantly influenced the modern technological landscape, with processors forming the backbone of computing devices. This project aims to delve into the intricacies of processor design by developing a custom 16-bit processor using Verilog Hardware Description Language (HDL). By leveraging Verilog HDL, this project enables a hands-on approach to understanding how processors operate at a fundamental level, focusing on the Harvard architecture. The design encapsulates key concepts such as instruction memory, data memory, a robust datapath, and an efficient control unit, all working harmoniously to execute a defined instruction set. | | |  |  |  |
| Decorative | | | | |
|  |  |  |  |
|  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Decorative | |  |  | | |  |  | |
|  | | PROJECT OVERVIEW | | | | |  | |
|  | |  |  | | |  |  | |
|  | **The processor is based on a Harvard Architecture with the following components:**  **Key Components:**  **Instruction Memory (ROM):**  A read-only memory module to store program instructions.  **Data Memory:**  A read-write memory module for handling data operations.  **Main Datapath:**  Includes 8 general-purpose registers.  Special-purpose registers for operations such as ALU tasks.  A robust ALU capable of executing various instructions.  **Control Unit:**  Manages the execution of instructions and interacts with the datapath via control and status signals. | | | | | | |  |
|  | Instruction Set: | | |  |  | | |  |
|  | The processor supports a diverse set of instructions categorized into R, I, and J types.  **Key instructions include:**   |  |  |  |  | | --- | --- | --- | --- | | **Instruction** | **Opcode** | **Type** | **Operation** | | Add | 0000 | R | Rd = Rs + Rt | | Sll | 0001 | R | Rd = Rs<<Shamt | | Lw | 0111 | I | Rd = Mem[Rs + Constant] | | Sw | 1000 | I | Mem[Rs + Constant] = Rd | | J | 1001 | J | PC = Address | | | | | | | |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Decorative | |  |  | | |  |  | |
|  | | **PROCESSOR ARCHITECTURE** | | | | |  | |
|  | |  |  | | |  |  | |
|  | The processor's architecture features:  **Datapath and Control Unit Interaction:**  Status and control signals ensure smooth execution of instructions.  **Special Registers**:   * **Program Counter (PC):**   An 8-bit register pointing to the current instruction address.   * **Status Register (SR):**   Tracks flags such as Zero, Negative, Carry, and Overflow.  **Status Register Working Example:**   * **Instruction:** sub $t0, $t1, $t2   Case 1: If $t1 == $t2, the Zero flag is set to 1.  Case 2: If $t1 < $t2, the Negative flag is set to 1.  Case 3: Overflow during operation sets the Overflow flag. | | | | | | |  |
|  | **Instruction Format:** | | |  |  | | |  |
|  | The processor supports three types of instruction formats, derived from the MIPS instruction set:  **Instruction Types:**  **Register Type (R):**  Format: Opcode | Rd | Rs | Rt | Shamt  Example: add Rd, Rs, Rt  **Immediate Type (I):**  Format: Opcode | Rd | Rs | Immediate  Example: addi Rd, Rs, Constant  **Jump Type (J):**  Format: Opcode | Address  Example: j Address | | | | | | |  |

|  |  |
| --- | --- |
|  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  |  | |
|  | | **IMPLEMENTATION** | | |  | |
|  | |  |  |  |  | |
|  | **Verilog Modules**  **The processor design is divided into modular components, each implemented in Verilog HDL for functionality and clarity.**  **Instruction Memory**  **Purpose**: Stores the program's instructions in a Read-Only Memory (ROM) format.  **Implementation:**  The ROM is initialized with pre-defined instructions.  The Program Counter (PC) fetches instructions sequentially from the ROM.  **Example Functionality:**  When the PC points to a specific address, the corresponding instruction is fetched and sent to the control unit for decoding.  **Data Memory**  **Purpose:**  Acts as a Read-Write memory, storing and retrieving data used during program execution.  **Implementation:**  Contains multiple memory locations (16-bit wide) for storing intermediate or final results.  Data is accessed using memory addresses provided by instructions like lw (load word) and sw (store word).  **Example Functionality:**  For the instruction lw $Rd, offset($Rs), the memory address is calculated using $Rs + offset, and the value at that memory address is loaded into $Rd.  **ALU (Arithmetic Logic Unit)**  **Purpose:** Performs all arithmetic and logical operations specified in the instruction set.  **Implementation:**  Inputs: Operands from the registers and control signals from the control unit.  **Outputs:** Results of operations and status flags (e.g., Zero, Negative, Overflow).  **Operations Supported:**   * Arithmetic: add, sub, addi * Logical: and, or * Shifts: sll (shift left logical), srl (shift right logical) * Multiplication: Generates results in special-purpose registers Hi and Lo.   **Example Functionality:**  For add $Rd, $Rs, $Rt, the ALU adds the values from $Rs and $Rt and stores the result in $Rd.  **Control Unit**  **Purpose:**  The brain of the processor, responsible for decoding instructions and generating control signals to guide the datapath components.  **Implementation:**  Decodes the opcode and identifies the type of instruction (R, I, or J). Generates control signals to enable specific operations in the ALU, data memory, and register file.  **Example Functionality:**  For the lw instruction, the control unit enables:  Memory read signal for data memory.  Write signal for the destination register.  Address calculation logic using $Rs and the offset.  **Register File**  **Purpose:**  A collection of 16-bit registers used for storing temporary data during execution.  **Implementation:**  Consists of 8 general-purpose registers.  Special-purpose registers:  Hi and Lo: Store results from multiplication operations.  Status Register (SR): Stores flags like Zero, Negative, Overflow, and Carry.  **Example Functionality:**  The add $Rd, $Rs, $Rt instruction retrieves values from $Rs and $Rt, performs the addition in the ALU, and writes the result to $Rd | | | | |  |
|  |  |
|  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  | | |  |  | |
|  | | **INSTRUCTION EXECUTION FLOW** | | | | |  | |
|  | |  |  | | |  |  | |
|  | The execution of an instruction in the processor follows a systematic flow:   1. **Fetch**   The Program Counter (PC) points to the memory address of the next instruction.  The instruction is fetched from the ROM and sent to the control unit.   1. **Decode**   The control unit decodes the fetched instruction to determine:  The operation (e.g., add, load, store).  The type of instruction (R, I, or J).  The operands involved (registers or memory addresses).   1. **Execute**   The ALU performs the operation based on the decoded instruction and control signals.  For example:  **add $Rd, $Rs, $Rt**: ALU adds the values from $Rs and $Rt.  **sll $Rd, $Rs, shamt**: ALU shifts the value in $Rs left by shamt.   1. **Memory Access**   For instructions like lw and sw:  The memory module is accessed to load or store data.  Example:  **lw $Rd, offset($Rs):** Data is loaded from Memory[$Rs + offset] into $Rd.   1. **Write Back**   Results are written back to the destination register.  Special-purpose registers (e.g., Status Register) are updated with flags like Zero, Negative, Overflow, or Carry when applicable.   1. **PC Update**   For sequential instructions, the PC increments to point to the next instruction.  For jump and branch instructions, the PC is updated based on the address or condition. | | | | | | |  |
|  |  | | |  |  | | |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  |  | |
|  | | **CODES** | | |  | |
|  | |  |  |  |  | |
|  | **ALU**  module ALU(op\_code,rd,rs,rt,shamt,constant,hi\_in,lo\_in,hi\_out,lo\_out);  //inputs  input [3:0] op\_code; // op\_code is of 4 bits  input [15:0]rs,rt,hi\_in,lo\_in,constant; // all inputs are of 16 bits  input [2:0] shamt; // shamt is of 3 bits  //temp regs  reg [8:0] address;  reg [31:0] TEMP;  //outputs // temporary wire for saving 32 bit answer of multiplication  output reg[15:0] rd,hi\_out,lo\_out; // outputs are also 16 bits  //////////////////////////////////////////RAM//////////////////////////////////////////////////  reg [15:0] RAM [0:7];  initial  begin  RAM[0] = 16'b 0000000000000001;  RAM[1] = 16'b 0000000000000101; // Memory location  RAM[2] = 16'b 0000000000000000;  RAM[3] = 16'b 0000000000000000;  RAM[4] = 16'b 0000000000000011;  RAM[5] = 16'b 0000000000000001;  RAM[6] = 16'b 0000000000000001;  RAM[7] = 16'b 0000000000000001;  end  ///////////////////////////////////////////////////////  always @ (\*)  begin  if(op\_code== 0) // if op\_code is 0 then add rs and rt  begin  rd = rs + rt;  end    else if (op\_code==1) // if op\_code is 1 then shift the rs left by shamt amount  begin  rd = rs << shamt;  end  else if (op\_code==2) // if op\_code is 2 then shift the rs right by shamt amount  begin  rd = rs >> shamt;  end  else if (op\_code==3) // if op\_code == 3 then then multiply the answer will come 32 bit which we will store in TEMP  begin  TEMP = rs \* rt;  hi\_out = TEMP[31:16]; //saving 16 msb in hi  lo\_out = TEMP[15:0]; //saving rest 16 lsb in lo  end  else if (op\_code==4) //if op\_code == 4 then rd = lo\_in  begin  rd = lo\_in;  end  else if (op\_code==5) //if op\_code == 5 then rd = hi\_in  begin  rd = rs + constant;  end  else if (op\_code==6) //if op\_code == 6 then rd = rs | rt  begin  rd = rs | rt;  end  else if (op\_code==7) //if op\_code == 7 then rd = rs & rd  begin  rd = rs & rt;  end  else if (op\_code==8) //if op\_code == 8 then rd = rs + constant(addi)  begin  rd = rs + constant;  end  else if (op\_code==9) //if op\_code == 9 then rd = constant(li)  begin  rd = constant;  end  else if (op\_code==10) //if op\_code == 10 then rd = RAM [rs + constant]  begin  rd = RAM [rs + constant];  end  else if (op\_code==11) //if op\_code == 11 then RAM [rs + constant] = rd  begin  RAM [rs + constant] = rd;  end    else  begin  rd = 0;  end    end  endmodule  **DATAPATH**  module data\_path(op\_code,rd\_index,rs\_index,rt\_index,shamt,constant,rd\_value,rs\_value,rt\_value,low\_out,high\_out,SR);  //inputs  input [3:0] op\_code;  input [2:0] rd\_index, rs\_index, rt\_index, shamt;  input [5:0] constant;  //outputs  output [15:0] rd\_value;  output reg [15:0] rs\_value,rt\_value,low\_out,high\_out;  output [16:0]SR;  //wires  wire [15:0] hi\_in,lo\_in,hi\_out,lo\_out;  //regs  reg [16:0]HI,LO,SR;  ///////////////////////// REG FILE ///////////////////  reg [15:0] regfile[0:7];  initial  begin  regfile[0]=7;  regfile[1]=6;  regfile[2]=5;  regfile[3]=4;  regfile[4]=3;  regfile[5]=2;  regfile[6]=1;  regfile[7]=0;  //special registers  HI=16'b0000000000000000;  LO=16'b0000000000000000;  SR=16'b0000000000000000;  end  always@(op\_code)  begin  if (op\_code>=0 && op\_code<=7) //R TYPE  begin  rs\_value=regfile[rs\_index];  rt\_value=regfile[rt\_index];  end  else if (op\_code>=8 && op\_code<=11) //I TYPE  begin  rs\_value=regfile[rs\_index];  end  else  begin //J TYPE IS USED WITH PC  rs\_value=0;  rt\_value=0;    end  end  //Implementing SR  always @(\*)  begin  if(rs\_value==rt\_value)  SR[3]=1; //zero flag  else if(rs\_value<rt\_value)  SR[2]=1; //negative flag  else if(rs\_value[15]==1 && rt\_value[15]==1)  SR[0]=1; //overflow flag  else if(rs\_value[15]==0 && rt\_value[15]==0)  SR[0]=1; //overflow flag  else if(rs\_value[16:0] || rt\_value[16:0])  SR[1]=1; //carry flag  end      always @(\*)  begin  //assigning values of low and high to output  low\_out = lo\_out;  high\_out = hi\_out;  //assigning value of rd,high and lo to register  regfile[rd\_index] = rd\_value;  HI = hi\_out;  LO = lo\_out;  end  ALU call(op\_code,rd\_value,rs\_value,rt\_value,shamt,constant,hi\_in,lo\_in,hi\_out,lo\_out);  endmodule  **CONTROL UNIT**  module control\_unit(clk,rst,opcode,rd\_reg,rs\_reg,rt\_reg,shamt,constant,address,rd\_data);  input clk,rst;  reg [15:0] instr;  reg [7:0] PC;  output [3:0] opcode;  output [2:0] rd\_reg,rs\_reg,rt\_reg,shamt;  output [5:0] constant;  output [8:0] address;  output [15:0] rd\_data;  //ROM MEMORY  reg [15:0] ROM [0:15];  initial  begin  //(R TYPE INSTRUCTION)  ROM[0] = 16'b 0000\_111\_001\_000\_000; // Opcode = 0 so rd = rs + rt  ROM[1] = 16'b 0001\_111\_001\_000\_001; // Opcode = 1 so rd = rs << shamt  ROM[2] = 16'b 0010\_111\_001\_001\_001; // Opcode = 2 so rd = rs >> shamt  ROM[3] = 16'b 0011\_111\_001\_001\_000; // Opcode = 3 so rd = [hi,lo] = rs x rt  ROM[4] = 16'b 0100\_111\_001\_001\_000; // Opcode = 4 so move from lo to rd  ROM[5] = 16'b 0101\_111\_001\_001\_000; // Opcode = 5 so move from hi to rd rd = rs + consatnt  ROM[6] = 16'b 0110\_111\_001\_001\_000; // Opcode = 6 so rd = rs | rt  ROM[7] = 16'b 0111\_111\_001\_001\_000; // Opcode = 7 so rd = rs & rt  //(I TYPE INSTRUCTION)  ROM[8] = 16'b 1000\_111\_001\_001000; // Opcode = 8 so rd = rs + consatnt  ROM[9] = 16'b 1001\_111\_001\_001000; // Opcode = 9 so rd = constant  ROM[10] = 16'b 1010\_111\_001\_001000; // Opcode = 10 so rd = memory(rs + constant)  ROM[11] = 16'b 1011\_111\_001\_001000; // Opcode = 11 so memory(rs + constant) = rd  //(J TYPE INSTRUCTION)  ROM[12] = 16'b 1100\_000000010\_000; // Opcode = 12 so PC = PC + address  ROM[13] = 16'b 1101\_000000010\_000; // Opcode = 13 so PC = PC + address  ROM[14] = 16'b 1110\_000000010\_000; // Opcode = 14 so PC = PC + address  end  // Making Clock  always @ (posedge clk,posedge rst)  begin  if(rst==1)  begin  PC<=0;  end  /\*else if(opcode>=12 && opcode<=14)  begin  PC <= PC + address;  end\*/  else  begin  PC <= PC + 1;  end  end  //Assigning instruction to instr reg using ROM[PC]  always@(\*)  begin  instr = ROM[PC];  end  decode call(instr,opcode,rd\_reg,rs\_reg,rt\_reg,constant,shamt,address);  data\_path uut(clk,opcode,rd\_offset,rs\_offset,rt\_offset,shamt,constant,rd\_data);  endmodule  **DECODER**  module decode(instr,opcode,rd\_index,rs\_index,rt\_index,constant,shamt,address);  //inputs  input [15:0] instr;  //outputs  output reg [3:0] opcode;  output reg [2:0] rd\_index,rs\_index,rt\_index,shamt;  output reg [5:0] constant;  output reg [8:0] address;  //dividing each instruction according to its type  always@(instr)  begin  opcode=instr[15:12]; // assigning first 4 bits to opcode  if (opcode>=0 && opcode<=7) // if opcode ranges from 0 to 7 then it is r type  begin  opcode=instr[15:12];  rd\_index = instr[11:9];  rs\_index = instr[8:6];  rt\_index = instr[5:3];  shamt = instr[2:0];  end  else if(opcode>=8 && opcode<=11 ) // if opcode ranges from 8 to 11 then it is i type  begin  opcode = instr[15:12];  rd\_index = instr[11:9];  rs\_index = instr[8:6];  constant = instr[5:0];  end  else // otherwise it is j type  begin  opcode=instr[15:12];  address = instr[11:3];  end  end  endmodule  **PROCESSOR**  module processor(clk,reset,PC,opcode,rd\_index,rs\_index,rt\_index,shamt,constant,address,rd\_value,rs\_value,rt\_value,low\_out,high\_out,SR);  //inputs  input clk,reset;  reg [7:0] pc;  //outputs  output [7:0] PC;  output [3:0] opcode;  output [2:0] rd\_index, rs\_index, rt\_index, shamt;  output [5:0] constant;  output [8:0] address;  output [15:0] rd\_value,rs\_value,rt\_value,low\_out,high\_out,SR;  // Making Clock  always @ (posedge clk,posedge reset)  begin  if(reset==1)  begin  pc<=0;  end  else if (pc>14)  begin  pc <= 0;  end  else if(opcode>=12 && opcode<=14)  begin  pc <= pc + address;  end  else  begin  pc <= pc + 1;  end    end  assign PC = pc;  //using wires to transfer controls from CU to Datapath  wire [3:0] opcode;  wire [2:0] rd\_index,rs\_index,rt\_index,shamt;  wire [5:0] constant;  wire [8:0] address;  //calling CU and Datapath  control\_unit cal1(PC,opcode,rd\_index,rs\_index,rt\_index,shamt,constant,address);  data\_path cal2(opcode,rd\_index,rs\_index,rt\_index,shamt,constant,rd\_value,rs\_value,rt\_value,low\_out,high\_out,SR);  endmodule  **TEST BENCH**  module tb\_processor;  // Inputs  reg clk;  reg reset;  // Outputs  wire [7:0] PC;  wire [3:0] opcode;  wire [2:0] rd\_index;  wire [2:0] rs\_index;  wire [2:0] rt\_index;  wire [2:0] shamt;  wire [5:0] constant;  wire [8:0] address;  wire [15:0] rd\_value;  wire [15:0] rs\_value;  wire [15:0] rt\_value;  wire [15:0] low\_out;  wire [15:0] high\_out;  wire [15:0] SR;  // Instantiate the Unit Under Test (UUT)  processor uut (  .clk(clk),  .reset(reset),  .PC(PC),  .opcode(opcode),  .rd\_index(rd\_index),  .rs\_index(rs\_index),  .rt\_index(rt\_index),  .shamt(shamt),  .constant(constant),  .address(address),  .rd\_value(rd\_value),  .rs\_value(rs\_value),  .rt\_value(rt\_value),  .low\_out(low\_out),  .high\_out(high\_out),  .SR(SR)  );  initial begin  // Initialize Inputs  clk = 0;  reset = 0;  // Wait 100 ns for global reset to finish  #100;    // Add stimulus here  reset = 1;  #100;  reset = 0;  #100;  end    always  begin  clk=!clk;  #100;  end    endmodule | | | | |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  |  | |
|  | | **CONCLUSION** | | |  | |
|  | |  |  |  |  | |
|  | **The custom 16-bit processor project successfully implements a simplified yet functional processor architecture. By integrating various components and designing a robust instruction set, the project demonstrates the practical application of Verilog HDL in computer system design.** | | | | |  |