MIPS Datasheet

Description:

This is a working synthesizable 32-bit instructions MIPS processor designed for general use for clients to integrate in a variety of products. It is implemented with a memory-mapped bus interface and can perform 48 instructions.

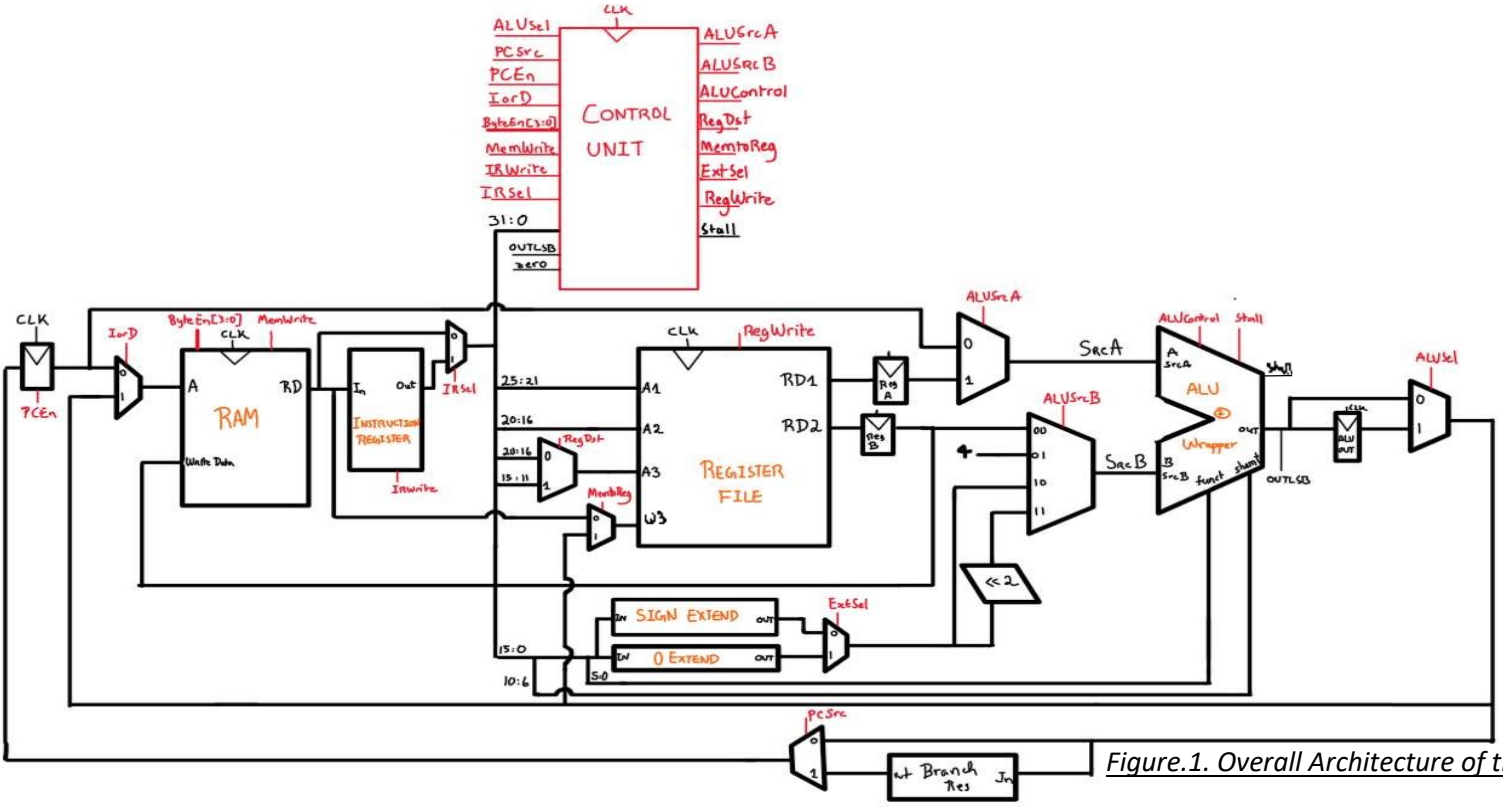

Figure.1. Overall Architecture of the CPU

Specifications:

- CPU interface: ***bus*** [requires instructions and data to be fetched over the same interface, allows memory to have variable latency]
- Avalon supported RAM
- Must successfully execute the 48 wanted instruction
- All signals are synchronous to clock
- Includes a reset behavior
  - Must be set high for at least 1 cycle
- Includes halt behavior
  - Active signal high
- Must have two special memory locations
  - 0x00000000 : Attempt to execute address 0 causes CPU to halt
  - 0xBFC00000: where execution should start after reset

Featured Elements:

Datapath:

1) **ALU + wrapper**
   - Execute arithmetic operations and deals with R-type instructions.
2) **Register File**
   - Holds onto data, reducing data access time by avoiding the RAM
3) **Instruction Register**
   - Holds onto instructions
4) **Sign Extend**
   - When an I-type instruction is called, the 16-bit signed offset stored in the immediate field needs to be signed extended in order to know if it is a negative or positive offset.
5) **ALUSrcA Multiplexer**
   - Selects between Program Counter [PC] instruction and output RD1 of register file.
6) **ALUSrcB Multiplexer**
   - Selects between the output RD2 of the register file, the value 4, the immediate offset or the immediate offset shifted twice to the left.
7) **Branch Register**
   - Holds onto branched address due to delay slot.

Other :

: Singular registers

: Multiplexers

RAM

Control Unit:

Inputs:

- **Instruction [31:0]**
- **OUTLSB:** least significant bit coming from the output of ALU. Is used when Set On Less intructions are called.
- **Stall:** delays the execution by one cycle for multiplying and divide.

Outputs:

- **ALUSrcA/ALUSrcB**: enables multiplexer to choose SrcA/SrcB.
- **ALU_Control:** Tells ALU which arithmetic operation to execute as well as which R-type instruction to execute.
- **ALUSel:** skip ALUOut Register or use register.
- **ByteEn [3:0]:** Enables specific byte lanes: each bit corresponds to a byte in RD1/RD2 and W3. When writing in RAM, the number of byte indicate the length of the word being written. Uncalled bytes are ignored. Similar process for reading.
- **ExtSel:** Select the immediate offset sign extended or zero extended.
- **IorD:** Enables multiplexer which selects between next instruction or data from the output of the ALU.
- **IRWrite:** Enables writing in Instruction Register
- **IRSel:** Enables multiplexer which selects between output of RAM or output of the instruction register block.
- **MemtoReg:** Enables multiplexer that selects between data from RAM or outputted from the ALU as an input to.
- **MemWrite:** Enables Writing in RAM.
- **PCSrc:** Enables multiplexer which selects between output of ALU or output of Branch register when branch instructions are called.
- **PCEn:** Enables register to hold onto the next instruction.
- **RegDst:** Enables multiplexer to select either [20:16] as input or [15:11]. Depending if it's an R-type or I-type instruction the Destination Register is specified in different parts of the instruction.
- **RegWrite**: enables writing in Registers

Design decisions taken when implementing the CPU:

- Model: Multicycle processor
  - More tedious as it has extra internal registers, uses an ALU, need PC updates, memory organization but leads to a faster processor as control path is faster and more easily pipelined if more speed is wanted.
- Implement a state machine decoder
- Implement a full ALU instead of using specific Verilog arithmetic operations for each instructions
- Not having an ALU decoder in Control Unit to manage R-Type Instructions, but implement a wrapper around the main operations ALU to process the function instructions.
- Haven an assembler to convert assembly language to binary code

Clever Features:

- Implemented state machine in decoder
  - Facilitate understanding of paths for each instruction
  - facilitate future debugging process for failing instructions.
- Sequential multiplier/divider and stall
  - Since it is sequential, the operations will be done faster and the hardware itself will take less area when manufacturing it.
  - Does take multiple cycles but can work on high frequency clock.

Approach taken to testing CPU:

First, the team began by testing the major blocks of the Datapath individually. This approach was taken to minimize the probability of failure when testing the whole CPU. It is making sure that each block does what it is supposed to do.

1) Register File Testbench approach:

Components tasks (what it needs to do)

- Need to hold temporarily onto data in wanted register and output when asked from wanted register

Inputs

- A1 (rs) corresponds to bits [25:21] of instruction.
- A2 (rt) corresponds to bits [20:16] of instruction.
- Destination register either obtained from bits [20:16] or [15:11].
- Data written to destination register obtained either from output of the ALU or the RAM.

Observable outputs to consider the testbench successful

- RD1 output data read in register A1, RD2 output data read in register A2
- A1 and A2 must be ranged between 0 and 31
- Data inputted in W3 must be written in the register Destination A3
- Used a couple of specific cases to verify its functionality

2) ALU and its wrapper Testbench approach:

Component tasks

- Combinatorially complete an operation depending on the opcode

Inputs

- SrcA and SrcB (32 bits) [Refer to *Featured elements* above if any confusion on what they represent]
- Funct [5:0] : Specific to R-type instructions, will tell ALU what operations is expected from the instruction
- ALU_Control (4 bits): Assigned to specific function codes and will trigger the operations inside the ALU for wanted output
- Shamt [10:6]: Field for shift instructions

Note

- Multiplier and divider have their own testbench due to complexity
  - Manually verify multiple different specific test cases to verify its overall functionality but more thorough testing will be done in the general ALU testbench

Observable outputs to consider the testbench successful

-

3) Testing of the CPU as a whole

Main approach

- Have 7 testcases per instruction
  - 4 edge test cases [maximum and minimum]
  - 3 test cases for randomized values