MIPS Datasheet

Description:

This is a working synthesizable 32-bit instructions MIPS multicycle 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.

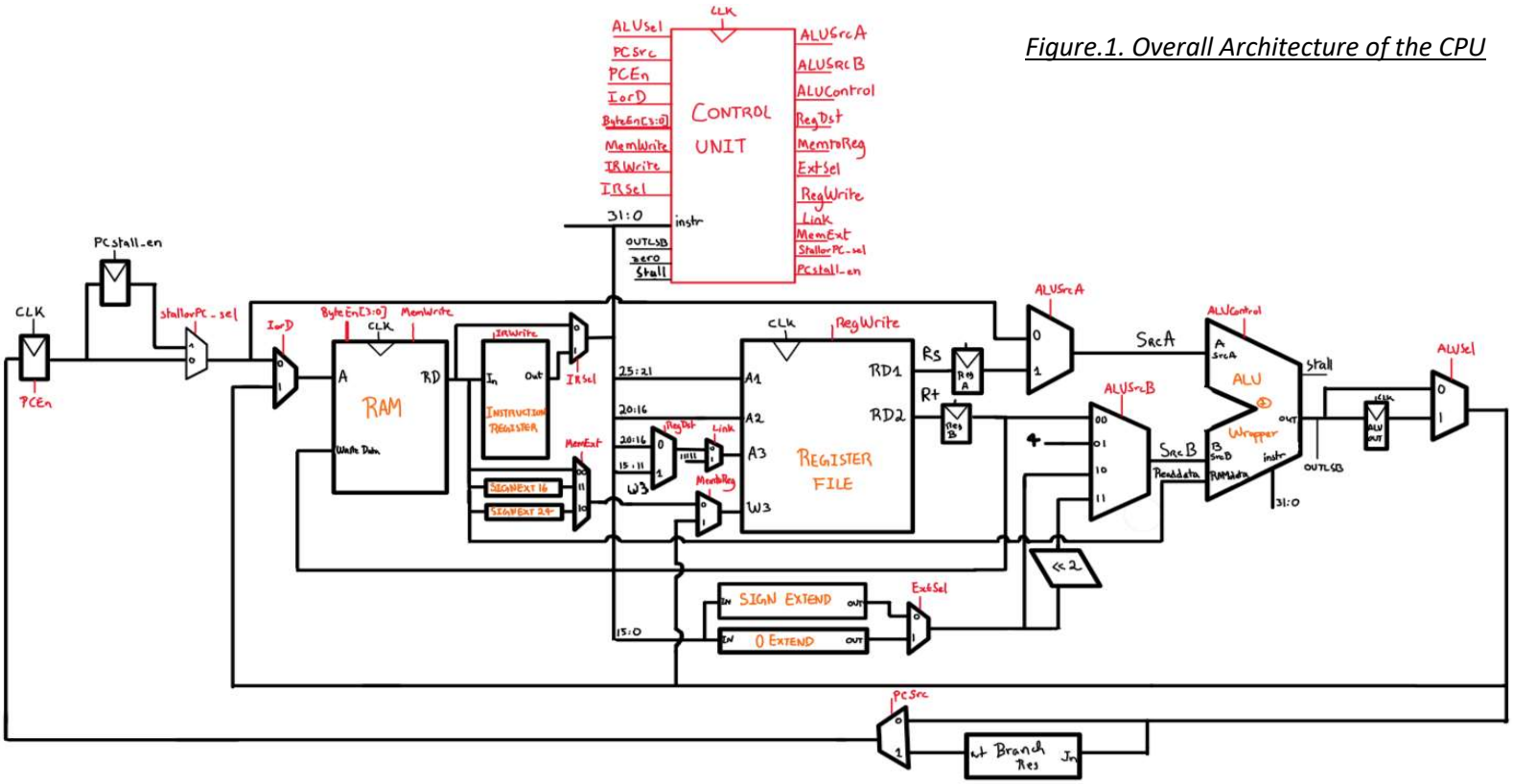


Figure.1. Overall Architecture of the CPU

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
- Focus is stressed on functionality rather than performance.

Featured Elements:

Datapath:

LU + wrapper

- Execute arithmetic operations and deals with R-type instructions.

egister File

- Holds onto data, reducing data access time by avoiding the RAM

ıstruction Register

- Holds onto instructions

ign 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.
- **Link:** Enables multiplexer which bypasses the option of Register Destination for the Return Address Register, $ra. Useful for instruction such as JAL and Branch/link.
- **MemExt:** Enables multiplexer to select between data outputted from RAM, data extended by 16 bits when half loads are called or data extend by 24 bits when a byte is loaded.
- **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 that selects between output of the ALU or the Branch register when branch instructions are called.
- **PCEn:** Enables register to hold onto the next instruction.
- **PCorStall_en**: Enables path with stalled PC or regular path
- **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:

- The team decided to implement a **multicycle processor**.
  - It is more tedious as it has extra internal registers, uses an ALU, needs PC updates, and memory organization but will lead to a faster processor. Its control path is faster and more easily pipelined if more speed is desired.
- Implement **a state machine decoder**
  - Decodes each instruction individually apart from R-type which are approached in one category.
- Implement a **full ALU** instead of using specific Verilog arithmetic operations for each instruction
  - A **wrapper** processing specifically R-type instructions was also added around the main ALU to keep better track of the different type of operations.
- Have an **assembler** to convert assembly language to binary code
- When errors were encountered during testing, minimized changes to the original circuit was favored.
  - Thus, resulting in **addition of multiplexers** to fix paths that encountered errors and find solutions to repair instructions.

Clever Features:

- Implement state machine in decoder
  - The path for each implemented instruction is straightforward. If the client was to have an additional professional look at the processor, having a standard format of fetch, decode, exec1/2/3 also eases their understanding and further contribution.
  - If the client encounters any bugs or want to include a new instruction the decoder facilitates future debugging processes and the addition new 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.
- ALU + wrapper
  - All arithmetic operations are basically performed by the ALU and its wrapper. Bugs revolving arithmetic operations are more rapidly pin pointed.
- Script and c++ file which generates testcases
  - The file intakes a string and generates 6 testcases with proper naming and reference which are automatically place in wanted directories. This allows a more thorough testing on a wide range of random cases.

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

Observable outputs to consider the testbench successful

- RD1 output data read from register A1, RD2 output data read from register A2
- A1 and A2 must be ranged between 0 and 31
- Data inputted in W3 must be written in the register Destination A3
- The Testbench incorporated a couple of specific cases to verify its global functionality. If the register file performed as asserted at each clock the test was deemed correct.

2) ALU and its wrapper Testbench approach:

Component tasks

- Combinatorially complete an operation depending on the opcode

Note

- Multiplier and divider have their own testbench due to complexity
  - Manually verify multiple different specific test cases to verify its overall functionality.

Observable outputs to consider the testbench successful

- To test the ALU, 3 testcases were written per ALU arithmetic operations.
- Two edge cases and a random test case.
- All the test cases are manually written and if the ALU outputs as the wanted asserted value than the testcase is considered a pass.

3) Testing of the CPU as a whole

Main approach

As mentioned above, a script and C++ file was written to generate multiple specific and random testcases.

- The c++ file cins a string and determines whether the instruction is signed or unsigned
  - It will adjust the limits used in testcases and variables inputted in function depending on the sign.
  - 7 sets of arguments are then generated :
    - 2 sets for the lower limit of the 32-bit signed/unsigned values, 3 randomly generated sets between the upper and lower limit, and, 2 sets for the upper limit.
  - A function takes the input instruction and arguments and writes the skeleton for the test-cases and produces the result in the same scope. These are coutted to stdout.
- The script file compiles and echoes the input parameter into the executable and goes through the output file and separates the test-case skeletons and results into their own files with the correct name and directories.
- Instructions that could not conform to the script's structure were manually written by team members.

- The testing is broken down in 5 directories:
  - **0- instructions assembly** : contains all the testcases in assembly language.
  - **1- binary**: contains the testcases in hex binary which are going to RAM and outputted from the assembler.
  - **2- simulator**: has the output executable file from compiling all the files together.
  - **3- output:** has .stdout file which is the raw output from the executable cpu but also contains .out file which is the extracted output lines from .stdout that are going to be compared to the reference file
  - **4- reference**: has the expected output for the testcases.

The .out file is compared to the reference file to verify the functionality of the CPU regarding each instruction such as correct read/writing in RAM. Having a testing approach broken down in 5 directories facilitates testing debugging.

Area and Timing Summary for the "Cyclone IV E 'Auto" variant in Quartus

Area

Fitter Status: Successful – Sun Dec 20 15:01:31 2020
Quartus Prime Version: 16.0.0 Build 211 04/27/2016 SJ Lite Edition

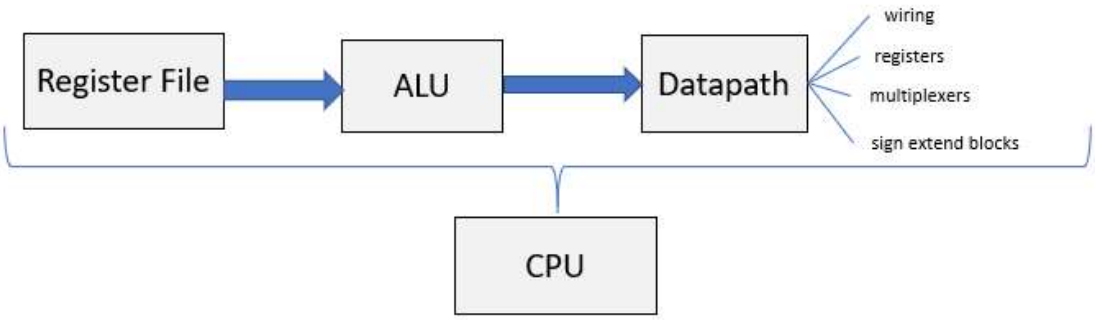
Revision Name:	mips
Top-level Entity Name:	mips_cpu_bus
Family:	Cyclone IV E
Device:	EP4CE10F17C6
Timing Models:	Final
Total logic elements:	4,113/10,320 (40%)
Total combinational functions:	4,106/10,320 (40%)
Dedicated logic registers:	1,225/10,320 (12%)
Total Registers:	1225
Total pins:	138/180 (77%)
Total virtual pins:	0
Total memory bits:	0/423,936 (0%)
Embedded Multiplier 9-bit elements:	0/46 (0%)
Total PLLs:	0/2 (0%)

Timing

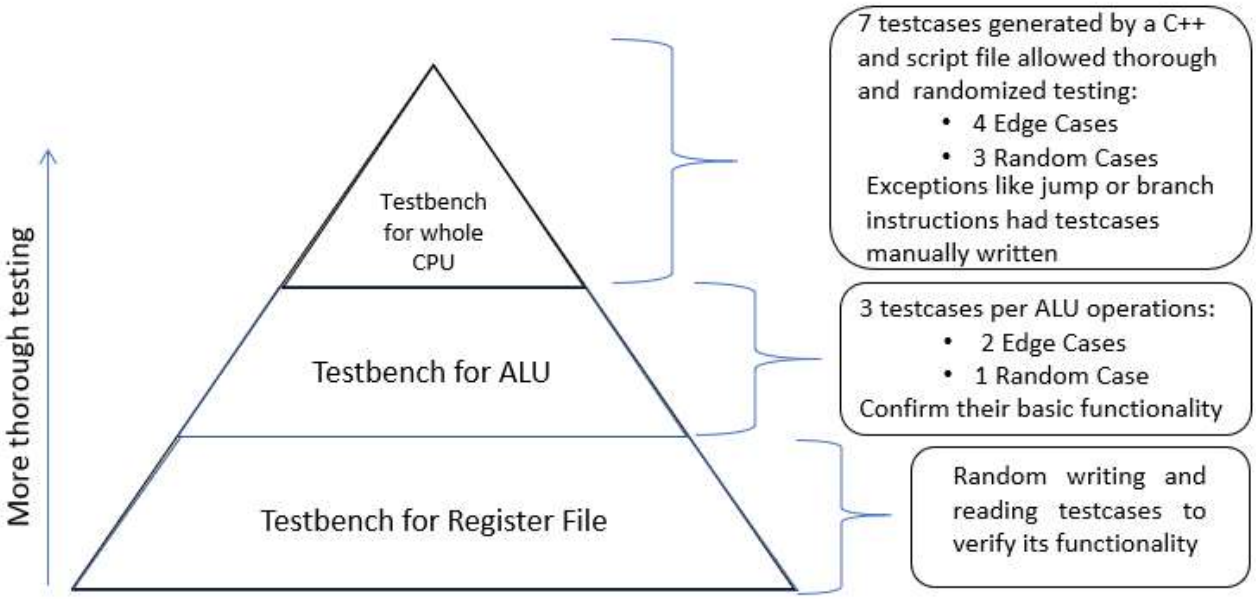
Fmax:	37.6MHz
Restricted Fmax:	37.6Mhz
Clock Name:	clk

TESTING PROCESS DIAGRAM

Implementation process of the Datapath



Testing Intensity



Testing Flow