# ARM’S-BIG-BROTHER Spec Sheet

## 1 Intro

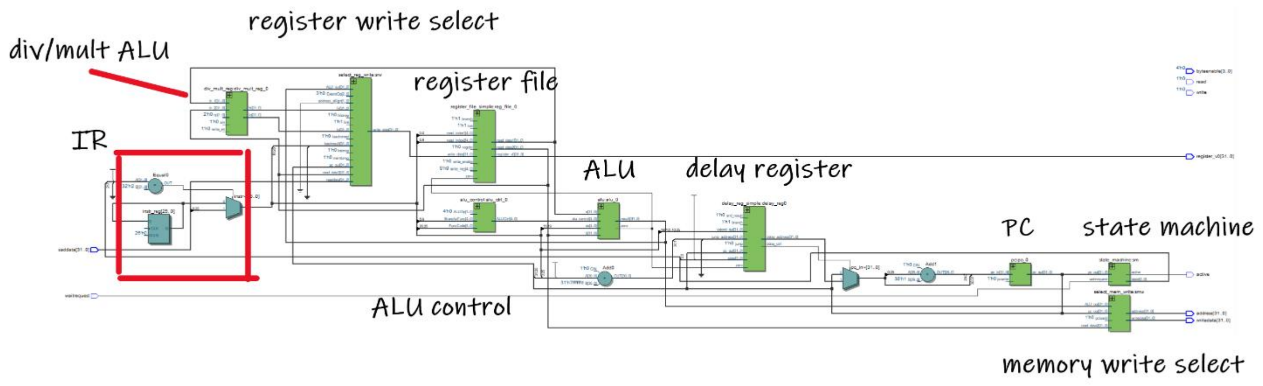
The ABB chip is a 32-bit based AVALON bus interface compatible CPU, based on a subset of the MIPS ISA architecture.

## 2 CPU architecture overview

ABB has the main basic sections of any CPU: State Machine, PC, IR, ALU, Register File, and Control Unit. On top of this, we have added an ALU Control Unit, Delay Register, and a div/mult ALU, register write select, and a memory write select.

Memory is not a part of the CPU itself, instead, the CPU uses a standard Avalon Bus interface to communicate with external memory.

Excluding the control unit, clock input, and reset input, the CPU diagram has the following form:



1) State machine: A clocked component. Stores 5 states used to synchronise the events of the CPU. HALT, FETCH, DECODE, EXEC1, EXEC2. The machine begins in the halt state when reset moves into the FETCH state, then cycles through FETCH -> DECODE -> EXEC1 -> EXEC2 until the program terminates by setting the PC to 0, and returns to HALT. The introduction of this component to manage the CPU lends it-self well to future work pipe-lining the CPU.

2) Program Counter: A clocked component. Stores the next instruction address and increases by 4 at every fetch cycle unless written to by the delay register.

3) Instruction Register: A clocked component. Stores the current instruction and is updated every fetch cycle.

4) Register File: A clocked component. Contains 31 32-bits registers that support read from any specific register and write into any register and a constant 0 register. The value stored in register 2/v0 is considered the program output and is available as the register_v0 output.

5) ALU: A combinatorial component. An independent module that supports unsigned arithmetic operations (addition, subtraction), bitwise operations (and, or, xor, arithmetic and logical shifts), and signed and unsigned comparison operations (less than, not equal, comparisons with 0) which are used to support branch instructions.

6) ALU Control: A combinatorial component. It takes ALUOp (for I-type instructions), FuncCode (for R-type instructions) and BranchFunc (for branch instructions) as inputs from the control unit and produce a specific ALU control signal to feed the ALU.

7) div/mult ALU: A combinatorial component. It is independent of the main ALU. Supports multiplication and division and stores data into hi and lo registers. The reason for keeping it separate was because in the future if we wanted to improve this CPU, it would be useful to have these time-consuming instructions running in parallel and not form a bottleneck in the execution.

8) Branch register: A clocked component. Stores information about where branch or jump instructions will jump to, and holds that information until the following instruction has been fetched. This is required as MIPS uses a branch delay slot, where all jumps and branches execute the instructions immediately after them before jumping.

9) Control unit: A combinatorial component. This has 2 main purposes; to ensure that the right data flows to the correct locations and ensure that data is stored in registers or memory correctly at the right time. Most of the outputs are single-bit signals controlling data flow, basically controlling multiplexers. Some of the controls delegate control to other components such as the ALU Control or the div/mult ALU for more specific controls. The component is not clocked but gets state inputs which affect the control signals it produces.

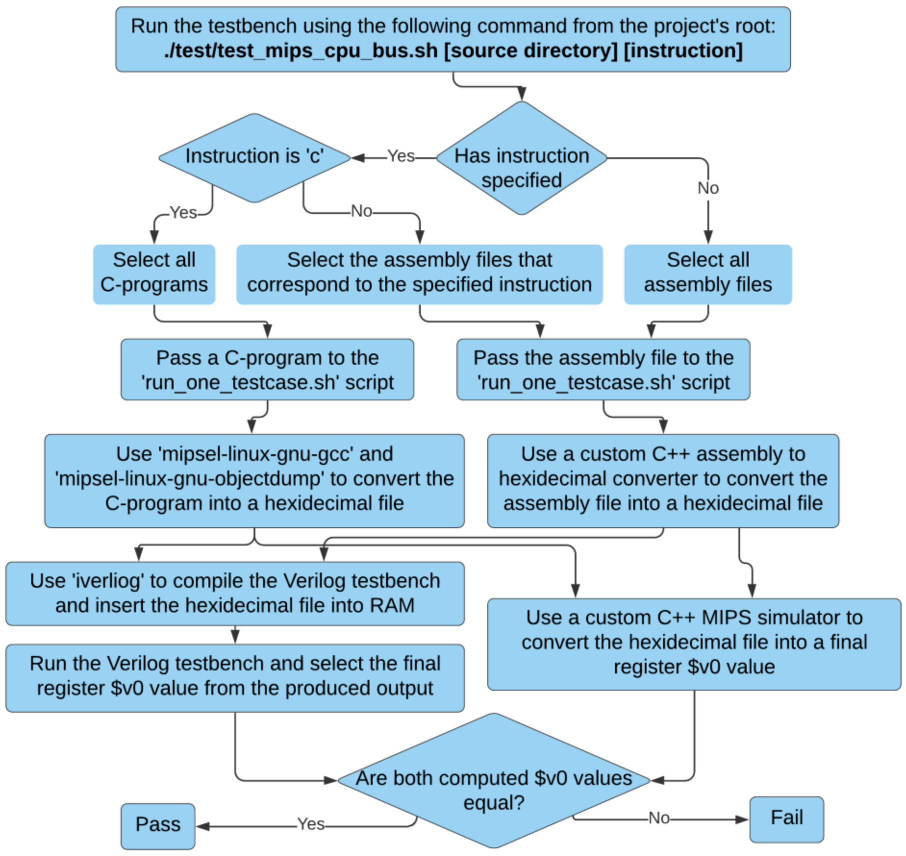
19) Register write select: A combinatorial component. This component ensures that the correct data is passed into the register write data input, and acts sort of like a glorified multiplexer taking data from many different parts of the CPU and selecting the correct one to pass.

20) Memory write select: A combinatorial component. This component is similar to the register write select but selects what data to pass to the memory write data input.

Sign extensions appeared in many parts in our CPU, including both signed and unsigned extensions. In particular, the immediate values from I-type instructions are only 16-bit, while in the ALU and the majority of other components, the inputs are 32-bit wide. Therefore, sign extensions are used most of the time to extend these 16-bit values into 32-bit wide.

## 3 TESTING APPROACH AND TESTBENCH

It was decided to split our group into two subgroups: three people to work on the CPU and two people to work on the testbench. This split seemed desirable as it would reduce the possibility of identical logical errors appearing in both the CPU and the testbench. If a person were to implement the CPU and then implement the testbench for it afterwards, that person might assume that something is true when implementing the CPU and then assume the same thing when implementing the testbench. The result would be that the testbench passes, even though the assumption was wrong in the first place.


Run the testbench using the following command from the project's root:
**./test/test_mips_cpu_bus.sh [source directory] [instruction]**
Has instruction specified
Yes
Instruction is 'c'
No
Yes
No
Select all C-programs
Select the assembly files that correspond to the specified instruction
Select all assembly files
Pass a C-program to the 'run_one_testcase.sh' script
Pass the assembly file to the 'run_one_testcase.sh' script
Use 'mipsel-linux-gnu-gcc' and 'mipsel-linux-gnu-objectdump' to convert the C-program into a hexidecimal file
Use a custom C++ assembly to hexidecimal converter to convert the assembly file into a hexidecimal file
Use 'iverliog' to compile the Verilog testbench and insert the hexidecimal file into RAM
Run the Verilog testbench and select the final register $v0 value from the produced output
Use a custom C++ MIPS simulator to convert the hexidecimal file into a final register $v0 value
Are both computed $v0 values equal?
Yes
Pass
No
Fail

A flowchart that abstractly describes the testbench can be seen on the left. The command in bold is used to run the testbench where “[source directory]” is a required and “[instruction]” is an optional string argument. Rectangular blocks represent processes and rhombuses represent conditions. Strings inside single quotes represent custom bash scripts or Linux tools/ programs.

The selection of the assembly files/ C-programs is accomplished using regex inside bash loops. The selected files are then passed sequentially into the next testbench stages. The testbench will print multiple lines of additional information for each test case to stderr to assist debugging processes. This includes a comment next to a "Fail" output, specifying at which stage the test case failed (For simplicity, the flowchart only contains the final "Fail" condition).

The backbone of the testbench is formed by two C++ programs: One that converts MIPS assembly into hexadecimal and one that simulates a MIPS CPU by producing a final register $v0 value from a hexadecimal file. These C++ programs are controlled by several bash scripts that connect the entire testbench.

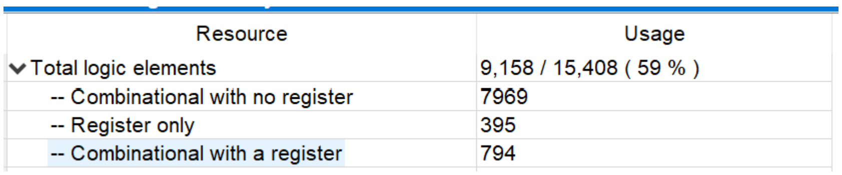
This testing approach took a while to set up but yields multiple benefits when compared to computing test cases and results manually. Firstly, new test cases can easily be added: A new assembly file must be inserted inside 'test/assembly/'. Its name needs to start with a lowercase MIPS instruction name terminated by a '_' character. No further steps are necessary. Secondly, the person adding the new assembly file does not need to know anything about MIPS instruction decoding as this will be done automatically by the testbench. Thirdly, the person adding the new test case does not need to compute the final expected (reference) register $v0 value manually as the testbench does this. Calculating this value by hand can become very tedious as the test case becomes more complicated. These factors make adding new test cases easy and fast. This made it possible to test CPUs using as many test cases as necessary for every instruction, making sure that all potential edge cases are considered.

The testbench relies mainly on the final value in register $v0. Hence, test cases must modify the value inside $v0. Preferably, $v0 is modified continuously by the test case so that its final value is dependent on all instructions tested by the test case.

## 4 AREA AND TIMING SUMMARY

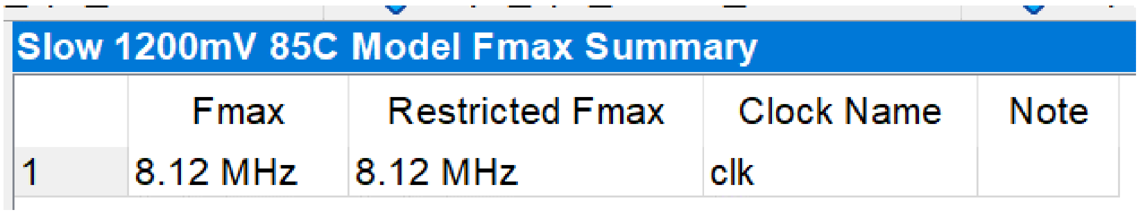
Using the compilation, and timing tools in Quartus with the Cyclone IV E Auto FPGA setting, our solution was synthesised and analysed to obtain the area and maximum clock rate possible for our CPU.

Our design was evaluated and emulated with the EP4CE15F23C6 device consuming the following resources when synthesised.



| Resource | Usage |
|---|---|
| Total logic elements | 9,158 / 15,408 ( 59 % ) |
| -- Combinational with no register | 7969 |
| -- Register only | 395 |
| -- Combinational with a register | 794 |

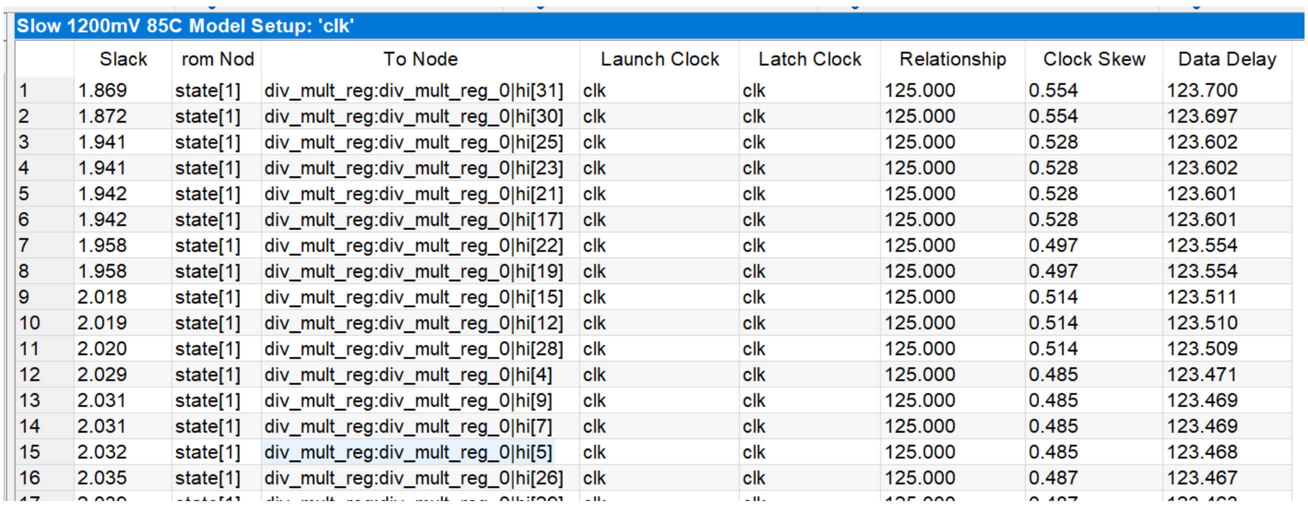
Simulating under realistic conditions of 1200mV at 85°C our design was measured to have a clock rate:



Slow 1200mV 85C Model Fmax Summary

| | Fmax | Restricted Fmax | Clock Name | Note |
|---|---|---|---|---|
| 1 | 8.12 MHz | 8.12 MHz | clk | |

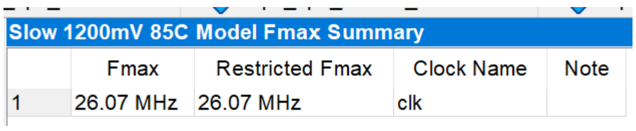
We also observe that our critical path is due to the div/mult alu:



**Slow 1200mV 85C Model Setup: 'clk'**

| | Slack | rom Nod | To Node | Launch Clock | Latch Clock | Relationship | Clock Skew | Data Delay |
|---|---|---|---|---|---|---|---|---|
| 1 | 1.869 | state[1] | div_mult_reg:div_mult_reg_0\|hi[31] | clk | clk | 125.000 | 0.554 | 123.700 |
| 2 | 1.872 | state[1] | div_mult_reg:div_mult_reg_0\|hi[30] | clk | clk | 125.000 | 0.554 | 123.697 |
| 3 | 1.941 | state[1] | div_mult_reg:div_mult_reg_0\|hi[25] | clk | clk | 125.000 | 0.528 | 123.602 |
| 4 | 1.941 | state[1] | div_mult_reg:div_mult_reg_0\|hi[23] | clk | clk | 125.000 | 0.528 | 123.602 |
| 5 | 1.942 | state[1] | div_mult_reg:div_mult_reg_0\|hi[21] | clk | clk | 125.000 | 0.528 | 123.601 |
| 6 | 1.942 | state[1] | div_mult_reg:div_mult_reg_0\|hi[17] | clk | clk | 125.000 | 0.528 | 123.601 |
| 7 | 1.958 | state[1] | div_mult_reg:div_mult_reg_0\|hi[22] | clk | clk | 125.000 | 0.497 | 123.554 |
| 8 | 1.958 | state[1] | div_mult_reg:div_mult_reg_0\|hi[19] | clk | clk | 125.000 | 0.497 | 123.554 |
| 9 | 2.018 | state[1] | div_mult_reg:div_mult_reg_0\|hi[15] | clk | clk | 125.000 | 0.514 | 123.511 |
| 10 | 2.019 | state[1] | div_mult_reg:div_mult_reg_0\|hi[12] | clk | clk | 125.000 | 0.514 | 123.510 |
| 11 | 2.020 | state[1] | div_mult_reg:div_mult_reg_0\|hi[28] | clk | clk | 125.000 | 0.514 | 123.509 |
| 12 | 2.029 | state[1] | div_mult_reg:div_mult_reg_0\|hi[4] | clk | clk | 125.000 | 0.485 | 123.471 |
| 13 | 2.031 | state[1] | div_mult_reg:div_mult_reg_0\|hi[9] | clk | clk | 125.000 | 0.485 | 123.469 |
| 14 | 2.031 | state[1] | div_mult_reg:div_mult_reg_0\|hi[7] | clk | clk | 125.000 | 0.485 | 123.469 |
| 15 | 2.032 | state[1] | div_mult_reg:div_mult_reg_0\|hi[5] | clk | clk | 125.000 | 0.485 | 123.468 |
| 16 | 2.035 | state[1] | div_mult_reg:div_mult_reg_0\|hi[26] | clk | clk | 125.000 | 0.487 | 123.467 |

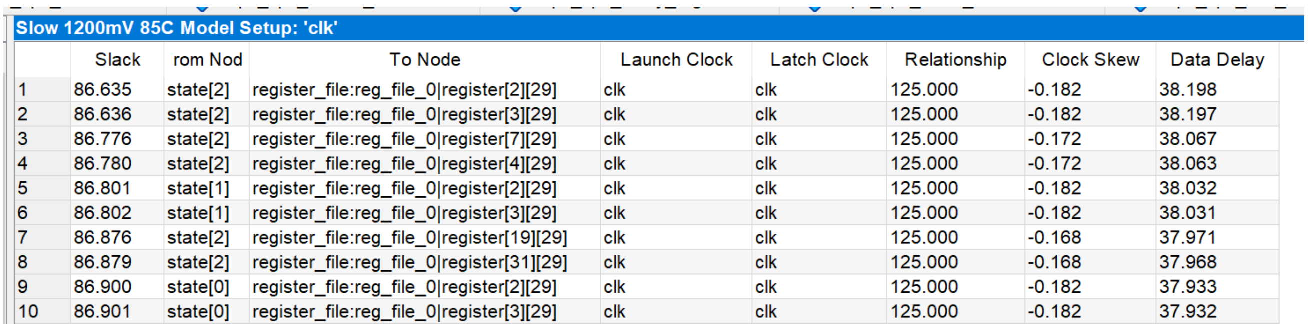
Specifically writing to the hi register, which contains the result of the upper result of multiplication, and the modulo result of a division. This could be improved by having the div/mult ALU running in parallel to the main CPU execution. This means that the hi and lo registers would not have valid results until a few cycles later, however, most instructions could run faster. In order to have an approximation of what this could look like, we re-ran the simulation without the div/mult ALU and managed a clock speed of:



**Slow 1200mV 85C Model Fmax Summary**

| | Fmax | Restricted Fmax | Clock Name | Note |
|---|---|---|---|---|
| 1 | 26.07 MHz | 26.07 MHz | clk | |

With a critical path having to do with writing to the register files:



**Slow 1200mV 85C Model Setup: 'clk'**

| | Slack | rom Nod | To Node | Launch Clock | Latch Clock | Relationship | Clock Skew | Data Delay |
|---|---|---|---|---|---|---|---|---|
| 1 | 86.635 | state[2] | register_file:reg_file_0\|register[2][29] | clk | clk | 125.000 | -0.182 | 38.198 |
| 2 | 86.636 | state[2] | register_file:reg_file_0\|register[3][29] | clk | clk | 125.000 | -0.182 | 38.197 |
| 3 | 86.776 | state[2] | register_file:reg_file_0\|register[7][29] | clk | clk | 125.000 | -0.172 | 38.067 |
| 4 | 86.780 | state[2] | register_file:reg_file_0\|register[4][29] | clk | clk | 125.000 | -0.172 | 38.063 |
| 5 | 86.801 | state[1] | register_file:reg_file_0\|register[2][29] | clk | clk | 125.000 | -0.182 | 38.032 |
| 6 | 86.802 | state[1] | register_file:reg_file_0\|register[3][29] | clk | clk | 125.000 | -0.182 | 38.031 |
| 7 | 86.876 | state[2] | register_file:reg_file_0\|register[19][29] | clk | clk | 125.000 | -0.168 | 37.971 |
| 8 | 86.879 | state[2] | register_file:reg_file_0\|register[31][29] | clk | clk | 125.000 | -0.168 | 37.968 |
| 9 | 86.900 | state[0] | register_file:reg_file_0\|register[2][29] | clk | clk | 125.000 | -0.182 | 37.933 |
| 10 | 86.901 | state[0] | register_file:reg_file_0\|register[3][29] | clk | clk | 125.000 | -0.182 | 37.932 |

Parallelising the div/mult ALU results in an over 3-time speed up.

## 5 Future Improvement

Using the insight from our testing speed is clearly a factor which needs to be improved on for the ABB chip. Two paths moving forward have already been identified for future improvement. The first is to pipeline. Given our current state machine setup, our CPU has many of the components required for pipelining, small additions to handle the hazards involved with the process would have to be added, but the overall structure lends itself to pipelining.

The second is to parallelise certain components namely the multiplier and divider. This would be a bigger challenge and would require the introduction of more hardware to ensure that all instructions have the correct data at the correct points in time. One potential way to approach this would be to add very basic out of order execution to the CPU.