International Journal of Applied Engineering Research, ISSN 0973-4562 Vol. 7 No.11 (2012)
© Research India Publications; http://www.ripublication.com/ijaer.htm

Vhdl Implementation of A Mips-32 Pipeline Processor

[1]Kirat Pal Singh, [2]Shivani Parmar
[1,2]Assistant Professor
[1,2]Electronics and Communication Engineering Department
[1]SSET, Surya World Institutions of Academic Excellence, Bapror, Rajpura, Punjab, India
[2] Sachdeva Engineering College for Girls, Gharuan, Punjab, India
Email: [1]kiratpal.singh@suryaworld.edu.in, [2]shivaniparmar03@gmail.com

***Abstract* -** This paper presents the design and implement a basic five stage pipelined MIPS-32 CPU. Particular attention will be paid to the reduction of clock cycles for lower instruction latency as well as taking advantage of high-speed components in an attempt to reach a clock speed of at least 100 MHz. The final results allowed the CPU to be run at over 200 MHz with a very reasonable chip area of around 900,000 nm2.

Keywords- MIPS Processor, Datapath, ALU, register file, pipeline

I. INTRODUCTION

The intent of this paper is to outline the processes taken in designing, implementing and simulating a five stage pipelined MIPS-32 processor.

A five stage pipeline was chosen because it represents a standard view of the division of the CPU workload.

Basic background on the CPU to be designed is provided. A breakdown of the important functional units, along with the reasoning behind the design decisions behind each one follows. Simulation and synthesis results are included as an indication of the success of this exercise.

II. BACKGROUND

A MIPS-32 compatible Central Processing Unit (CPU) was designed, tested, and synthesized as shown in figure 1. The processor had the following attributes:

- 5 stage pipeline
- Hazard Detection and correction
- Data Forwarding to reduce stall cycles

In order to allow the simulation of the CPU program data files were created and read into the instruction memory of the CPU. A small amount of memory for both data and instructions was also included to prove the concept and functionality of the CPU while also maintaining focus on the optimization of control and data path units of the main CPU design. The processor designed was a traditional five stage pipeline design. The stages were Instruction Fetch, Instruction Decode, Execute, Memory Access, and Write Back.

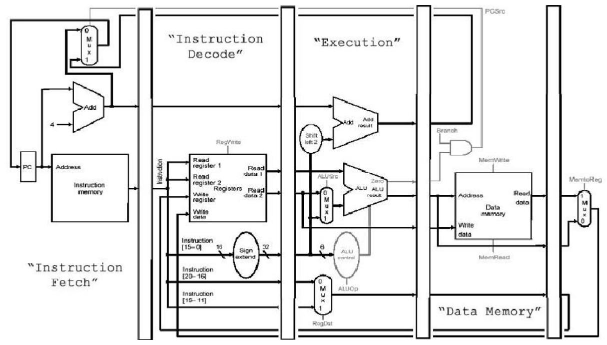

Figure 1. MIPS pipeline Processor [1]

The Instruction Fetch stage is where a program counter will pull the next instruction from the correct location in program memory. In addition the program counter was updated with either the next instruction location sequentially, or the instruction location as determined by a branch.

The Instruction Decode stage is where the control unit determines what values the control lines must be set to depending on the instruction. In addition, hazard detection is implemented in this stage, and all necessary values are fetched from the register banks.

The Execute stage is where the instruction is actually sent to the ALU and executed. If necessary, branch locations are calculated in this stage as well. Additionally, this is the stage where the forwarding unit will determine whether the output of the ALU or the memory unit should be forwarded to the ALU's inputs.

The Memory Access stage is where, if necessary, system memory is accessed for data. Also, if a write to data memory is required by the instruction it is done in this stage. In order to avoid additional complications it is assumed that a single read or write is accomplished within a single CPU clock cycle.

Finally, the Write Back stage is where any calculated values are written back to their proper registers. The write back to the register bank occurs during the first half of the cycle in order to avoid structural and data hazards if this was not the case.

The CPU included a hazard detection unit to determine when a stall cycle must be added. Due to data forwarding, this will only happen when a value is used immediately after being loaded from memory, or when a branch occurs. The hazard detection unit presents the Program Counter from updating with its next calculated value, clears out the Instruction Fetch registers, and forwards a No-op through the rest of the pipeline. A diagram of the hazard detection unit and its influence on the CPU as a whole is shown in figure 2.

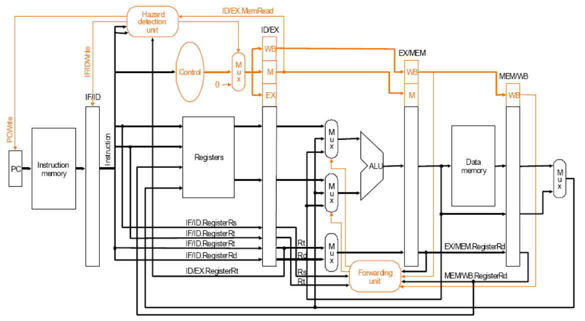

Figure 2. Hazard Detection Highlighted [1]

Data forwarding is required to eliminate the majority of the stall cycles. Without a forwarding unit, any time a value is used immediately after being calculated a stall cycle must be added. In addition, any time a value is fetched from memory, two stall cycles are introduced. This is shown in figure 3.

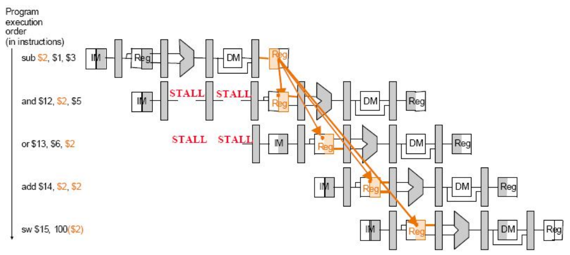

Figure 3. Data Forwarding [1]

With a forwarding unit, these stall cycles can be alleviated. See Figure 4.

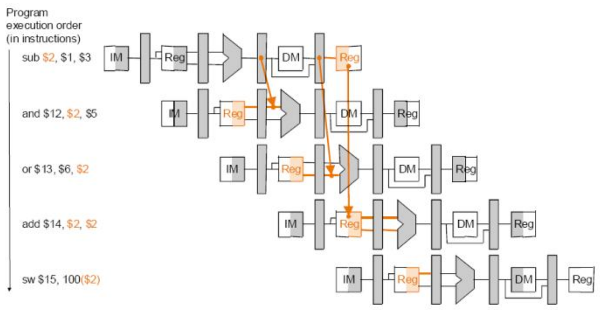

Figure 4. Stall Cycles Removed

The forwarding unit monitors the output of the ALU and system memory and determines whether these values are going to be needed as ALU inputs. If the recently calculate value is needed elsewhere in the data path before it is written to the register bank it will sent to the appropriate ALU input. A diagram of the forwarding unit and its affect on the CPU is shown in figure 5.

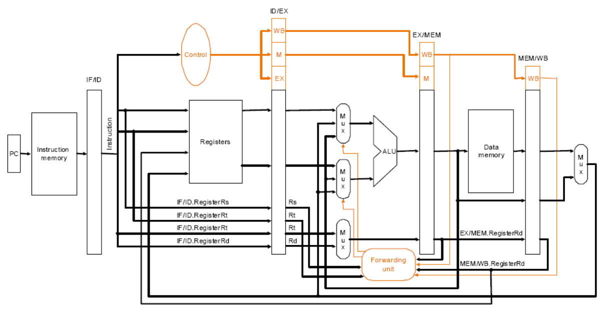

Figure 5. Forwarding Unit

III. IMPLEMENTATION

The overall CPU block is responsible for tying all of the stages together as well as providing the access to the outside world that the test bench uses to load instruction memory and monitor the register bank for test verification. Because the individual stages were made responsible for buffering their own individual outputs, it was not necessary for the CPU to contain any “glue” logic, it was simply necessary to correctly connect the different stages together. The designers and authors of the CPU itself and the individual stages can be seen in Table 1 The CPU is composed of the five different stages: Instruction Fetch, Instruction Decode, Execution, Data Memory, and the Writeback stage.

The instruction fetch stage has multiple responsibilities in that it must properly update the CPU's program counter in the normal case as well as the branch instruction case. The instruction fetch stage is also responsible for reading the instruction memory and sending the current instruction to the next stage in the pipeline, or a stall if a branch has been detected in order to avoid incorrect execution. The instruction fetch stage is composed of three components: instruction memory, program counter, and the instruction address adder. The instruction memory also takes inputs from the outside world that allow the loading of instruction memory for later execution.

The unit responsible for maintaining the program counter itself consisted of a 32-bit register for the address and an update line that would allow the address to update or not. This update line was necessary because for some hazards it is necessary to stall a cycle so it is required to ensure the same instruction will be executed on the next cycle.

The instruction memory unit was designed to model a small amount of cache and therefore was made to be accessed within a single CPU cycle. The instruction memory was sized at 1k bits and could therefore at maximum contain 32 separate instructions. In a real system this would be much larger to accommodate much larger instructions or would be attached to a much larger memory hierarchy. The instruction memory handled the reading or writing of a value into instruction memory within a single CPU cycle.

The final piece of the instruction fetch stage was the instruction memory address adder. This piece of purely combinational logic was responsible for adding 4 to address that was currently being read in the instruction memory. Whether or not this result was actually used to update the program counter was controlled by the hazard detection unit in the instruction decode stage.

The Decode Stage is the stage of the CPU's pipeline where the fetched instruction is decoded, and values are fetched from the register bank. It is responsible for mapping the different sections of the instruction into their proper representations (based on R or I type instructions). The Decode stage consists of the Control unit, the Hazard Detection Unit, the Sign Extender, and the Register bank, and is responsible for connecting all of these components together. It splits the instruction into its various parts and feeds them to the corresponding components. Regisers Rs and Rt are fed to the register bank, the immediate section is fed to the sign extender, and the ALU opcode and function codes are sent to the control unit. The outputs of these corresponding components are then clocked and stored for the next stage.

The Control unit takes the given Opcode, as well as the function code from the instruction, and translates it to the individual instruction control lines needed by the three remaining stages. This is accomplished via a large case statement

The hazard detection unit monitors output from the execute stage to determine hazard conditions. Hazards occur when we read a value that was just written from memory, as the value won't be available for forwarding until the end of the memory stage, and when we branch. The hazard detection unit will introduce a stall cycle by replacing the control lines with 0s, and disabling the program counter from updating. When a branch is detected the hazard detection unit will allow the PC to write, but will feed it the branch address instead of the next counted value.

The sign extender is responsible for two functions. It takes the immediate value and sign extends it if the current instruction is a signed operation. It also has a shifted output for branches.

One of the primary pieces of data storage in the CPU is the register bank contained within the instruction decode stage. This bank of registers is directly reference from the MIPS instructions and is designed to allow rapid access to data and avoid the use of much slower data memory when possible. The register bank contained in the CPU consisted of the MIPS standard 32 registers with register 0 being defined as always zero. The registers are defined as being written in the first half of the cycle and read in the second half. This is done to avoid structural hazards when one instruction is attempting to write to the register bank while another is reading it. Setting the register bank to this configuration also avoids a data hazard because a value that was just written can be read out in the same cycle.

The execute stage is responsible for taking the data and actually performing the specified operation on it. The execute stage consists of an ALU, a Determine Branch unit, and a Forwarding Unit. The execute stage connects these components together so that the ALU will process the data properly, given inputs chosen by the forwarding unit, and will notify the decode stage if a branch is indeed to be taken.

The alu is responsible for performing the actual calculations specified by the instruction. It takes two 32 bit inputs and some control signals, and gives a single 32 bit output along with some information about the output – whether it is zero or negative. This was accomplished by a large case statement dependent on the input control signals.

The determine Branch object is responsible for looking at the output of the alu, and the type of instruction it is decoding, and determining whether the system is to branch or not. For example, if the determine Branch unit sees a BEQ instruction; it will be looking at the 'is Zero' output of the ALU to determine branch success. The output of this unit is fed back to the decode stage's hazard detection unit.

The forwarding unit is responsible for choosing what input is to be fed into the ALU. It takes the input from the decode stage, the value that the alu has fed to the Memory stage, and the value that the Alu has fed to the write back stage, as well as the register numbers corresponding to all of these, and determines if any conflicts exist. It will choose which of these values must be sent to the ALU. For example, if one instruction uses a value that was calculated in the previous instruction, the forwarding unit would ignore the basic input value, and instead forward the output of the memory stage to the input of the alu instead.

The Memory stage is responsible for taking the output of the alu and committing it to the proper memory location if the instruction is a store. The memory stage contains one component: the data_memory object. It connects the data memory to a register bank for the write back stage to read, and also forwards on information about the current write back register. This register's number and calculated value are fed back to the forwarding unit in the execute stage to allow it to determine which value to pass to the ALU.

The data_mem object is a simulation of actual memory. It is a 1k block of cache that acts as data storage. This memory is responsible for storing both words and bytes, so it must implement optional sign extension for bytes. It must handle both read and write operations as requested.

The writeback stage is responsible for writing the calculated value back to the proper register. It has input control lines that tell it whether this instruction writes back or not, and whether it writes back ALU output or Data memory output. It then chooses one of these outputs and feeds it to the register bank based on these control lines.

IV. SIMULATION RESULTS

For simulation, a number of instructions were fed into the CPU and the outputs of registers 0 through 5 were monitored. The instructions that were tested included register based and immediate adds, subtracts (both signed and unsigned), multiplication (signed and unsigned), reading and writing data memory, and a loop that would force the CPU to jump back to the start of instruction memory and execute those same instructions again. The different adds were important because each exercised different parts of the CPU including the data forwarding unit, multiple registers and different functions within the ALU itself. The multiply instruction was also significant in that it proved that the instruction itself worked but also that the MFHI and MFLO registers within the ALU could be read and written to properly for the storing and reading of the 64 bit resultant. The jump instruction was very important also in that it exercised the branch detection unit, hazard detection unit as well as the ability of the instruction fetch stage to be able to jump to an address and continue execution with only the input of a single stall cycle. The simulation results can be seen in figures 6, 7 and 8.

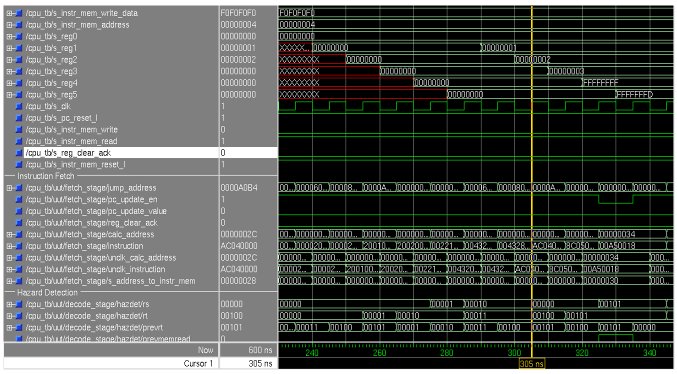

Figure 6. Simulation Waveform

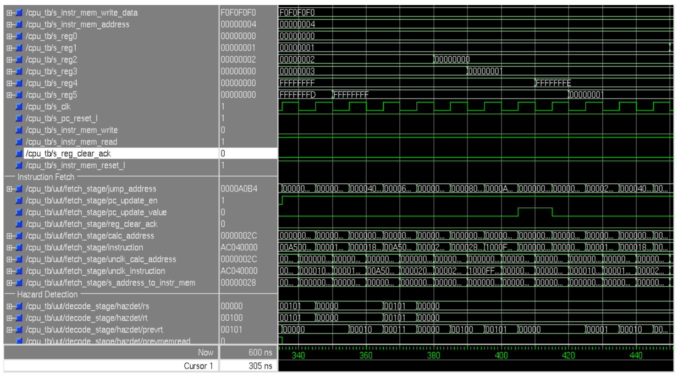

Figure 7. Simulation Waveform

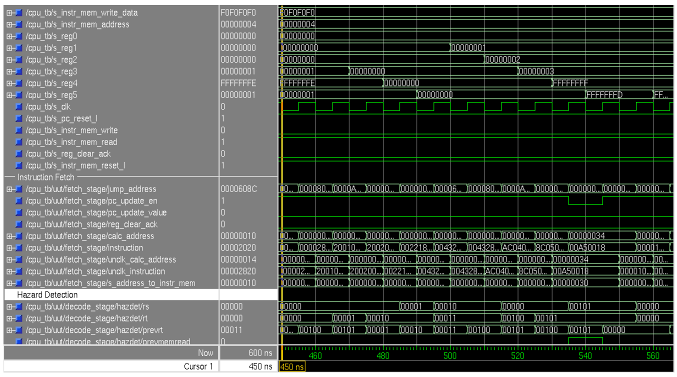

Figure 8. Simulation Waveform

This was then synthesized using Design Compiler. A clock speed of 200 MHz was achieved, along with an area of 896546.44 nm2. See Table 1.

Table 1. Area and speed

Module	Area(nm^2)	Speed (ns)
CPU (Top Level)	896546.44	4.69
Instruction Fetch Stage	158685.64	3.7
Instruction Decode Stage	188066.97	2.1
Execute Stage	217061625	2.55
Memory Stage	183921875	3.23
WriteBack Stage	835.48	1.65
Program Counter	3760.84	1.38
Instruction Memory	147963.09	2.17
Control	1869.31	2.3
Sign Extender	976.03	0.51
Register Bank	150129.42	2.25
Hazard Detection Unit	677.43	2.3
ALU	196370.38	2.04
Forwarding Unit	11011.88	2.38
Data Memory	176022.13	2.32

V. CONCLUSION

MIPS processor is widely used RISC processor in industry and research area. In this paper, we have successfully designed and synthesized a basic model of pipelined MIPS processor. The design has been modeled in VHDL and functional verification policies adopted for it. The simulation results show that maximum frequency of pipeline processor is increased from 100MHz to 200MHz.

VI. FUTURE WORK

This paper presents a comparative performance analysis and finding longer path delay at different pipeline stages using different technologies device. Our future work includes changing the processor architecture to make it capable of handling multiple threads and supporting network security application more effectively.

VII. REFERENCES

[1] Hennessy, John L. and Patterson, David A. Computer Organization & Design. 1998

[2] Hennessy, John L. and Patterson, David A. Computer Architecture: A Quantitative Approach. 2003

[3] M. Shabaan "Course Notes" http://www.ce.rit.edu/~meseec/eecc550-winter2004

[4] M. Shabaan "Course Notes" http://www.ce.rit.edu/~meseec/eecc551-spring2005

[5] Anon. "MIPS Architecture" http://www.cs.wisc.edu/~smoler/x86text/lect.notes/MIPS.html

[6] Kane, Gerry MIPS RISC Architecture 2001

[7] Anon. "MIPS Reference" http://edge.mcs.drexel.edu/GICL/people/sevy/architecture/MIPSRef(SPIM).html

[8] Anon. "Basic CPU Design" http://webster.cs.ucr.edu/AoA/Windows/HTML/CPU Architecturea3.html

[9] University of Calgary "Formal Verification in Intel CPU design" http://www.cpsc.ucalgary.ca/Dept/seminars.php?id=310&category=10

[10] University of Temple "How to Design a CPU" http://www.math.temple.edu/doc/howto/en/html/CPU-Design-HOWTO-4.html

[11] Hema Kapadia, Luca Benini, and Giovanni De Micheli, "Reducing Switching Activity on Datapath Buses with Control-Signal Gating" IEEE Journal Of Solid-State Circuits, Vol. 34, No. 3, March 1999

[12] Shofiqul Islam, Debanjan Chattopadhyay, Manoja Kumar Das, V Neelima, and Rahul Sarkar, "Design of High-Speed-Pipelined Execution Unit of 32-bit RISC Processor" IEEE 1-4244-0370-7 June.2006

[13] XiangYunZhu, Ding YueHua, "Instruction Decoder Module Design of 32-bit RISC CPU Based on MIPS"Second International Conference on Genetic and Evoltionary Computing,WGEC pp.347-351 Sept. 2008

[14] Rupali S. Balpande, Rashmi S. Keote, "Design of FPGA based Instruction Fetch & Decode Module of 32-bit RISC (MIPS) Processor", International Conference on Communication Systems and Network Technologies,2011.

[15] Mamun Bin Ibne Reaz, Md. Shabiul Islam, Mohd. S. Sulaiman, "A Single Clock Cycle MIPS RISC Processor Design using VHDL", IEEE International Conference on Semiconductor Electronics, pp.199-203 Dec. 2003

[16] MIPS Technologies, MIPS32™ Architecture for Programmers Volume I: Introduction to the MIPS32™ Architecture, rev. 2.0, 2003.

[17] Diary Rawoof Sulaiman, "Using Clock gating Technique for Energy Reduction in Portable Computers" Proceedings of the International Conference on Computer and Communication Engineering pp.839 – 842, May 2008