Designing a Complete Pipelined Datapath to MIPS ISA: Learning in Pratice

Francisco Carlos Silva Junior
Federal University of Piaui
juninho.ufpi@hotmail.com

Ivan Saraiva Silva
Federal University of Piaui
ivan@ufpi.edu.br

Laysson Oliveira Luz
Federal University of Piaui
layssonluz@ufpi.edu.br

Ramon S. Nepomuceno
Federal University of Piaui
ramonn76@gmail.com

ABSTRACT

This paper presents a pipeline implementation of the MIPS architecture supporting all conflicts and stalls, including some ones not addressed in the literature. It also presents a performance comparison between a pipelined implementation and a multicycle one. The conflicts and stalls are carefully explained, their reasons for occurring, how they are resolved and what loss in performance they can cause in the pipeline. An architecture that supports such conflicts is proposed and implemented in VHDL. An explanation on the ISA MIPS and how it has evolved through time is given as well. The results section shows that the pipelined datapath is 3 time faster than the multi-cycle implementation and requires 50% less logic elements in an FPGA implementation.

Categories and Subject Descriptors

B.7 [**Integrated circuits**]: Types and Design Styles—*microprocessor and microcomputers*; B.2 [**Arithmetic and logic structure**]: Design Styles—*pipeline*; B.5 [**Design**]: [Data-path design]; C.1.1 [**Processor architectures**]: Single data stream architectures—*pipeline processors*

General Terms

Design, Performance

Keywords

MIPS, Pipelining, Microarchitecture

1. INTRODUCTION

The MIPS microprocessor has a RISC (Reduced Instruction Set Computer) architecture and is used both as an example of ISA (Instruction Set Architecture) and as an example of a microarchitecture in many graduation courses in the whole world. It is also used commercially in embedded systems and in video-games, like the PlayStation [7].

In [5], one of the most respected books used in courses of computer organization and architecture, three versions of microarchitectures for the implementation of the ISA MIPS are presented. In all these versions, however, the microarchitecture is conceived with a subset of the instructions in mind. Such approach, although in most times enough to transmit the knowledge needed for the discipline, isn't so when one desires to give the students the means to implement a data-path of that complexity. Particularly, the presentation of a data-path that explores the pipelined execution technique makes the understanding of its fundamental concepts and demonstrates the conflicts that may eventually lead to a loss of performance. Not-withstanding, the presentation given is not enough for the student to observe all the existent conflicts.

This paper presents an implantation of the data-path for the ISA MIPS using the pipelined execution technique, as well as every instruction listed in [3]. This implantation handles all potential conflicts with the instructions, inserting the smallest possible amount of pipeline stalls. This paper presents also a development environment which was created so as to allow students to develop applications using high-level languages, particularly C, and make simulations or executions with cycle precision, using the ModelSim tool [1] or prototyping boards with reconfigurable devices, in particular we have used the Altera DE2-115 board.

This microarchitecture was developed in VHDL, and, as is shown in [5], it uses temporal superposition of the several phases of instruction execution, which are namely five stages: Instruction Fetch (IF); Instruction Decode and Register Read (ID); ALU Execution (EX); Data Memory Access (MEM) and Write Result Back (WB). This implementation has a greater flow of instructions, allowing the execution of one instruction per cycle when the pipeline is full. It is expected, thus, to obtain a better performance when compared with a mono-cycle data-path, where an instruction is executed in a single cycle, and a multi-cycle data-path, where an instruction is executed in at least three cycles.

In this paper, the implementation was compared in terms of area and performance with a multi-cycle data-path implementation, also implemented in VHDL.

2. THE ISA MIPS

The ISA (Instruction Set Architecture) defines the instruction set supported by a micro-processor or a particular architecture. The definition normally also contains the instruction's format, the register set and its uses, addressing modes, native data types, memory architecture, interruptions and exception treatment. The ISA MIPS specifies an instruction set composed of five categories: Arithmetic; Data Transfer; Logical; Bitwise Shift; Conditional Branch and Unconditional Jump. These instructions operate over a set of 32 registers, whose specific uses are also defined by the ISA, and not all are available to the programmer.

The definition of the ISA MIPS has evolved with time from the MIPS I ISA to the MIPS V ISA [2]. In order to implement the microarchitecture presented in this paper, the definitions available in [3] were used. These instructions have three formats, those being R, I and J. The R format specifies instructions that execute operations among registers and supports three register addresses. The I format specifies instructions that use immediate operands, including Arithmetics, such as addi (Add immediate); Logical, such as ori (Or Immediate); Conditional Branch, such as beq (Branch on Equal) or Data Transfer, such as lw (Load Word). The J format specifies instructions of the Unconditional Jump kind, and, in this implementation, includes solely the instructions jal (Jump and Link) and jump. This complete table with all instructions can be seen in several sources, such as [3], or [5].

3. THE PROPOSED ARCHITECTURE

In the implementation and pipeline execution of the MIPS architecture there are some undocumented conflicts. These conflicts concern the execution of the BEQ (Branch on Equal) or BNE (Branch not Equal) or the JR (Jump Register) and another previous instruction that modifies one of the register to be compared in the BEQ or BNE or the register where the branching address is stored in the JR. In this paper, BEQ refers to BEQ and BNE, because these instructions do the same thing, changing only the comparison, BEQ compares if is equal and BNE compares if is not equal.

These conflicts exist due to how the register bank is written on only in the fourth cycle, after the instruction is fetched in the memory and a later instruction needs the data which is about to be written. The later will read the register bank in the second cycle, which means it will read the wrong data, for that data will only be updated in the fifth execution cycle. In case this conflict is not handled, this will result in failure, since the data will be incorrect by the end of the execution. Table 1 shows what those conflicts are.

In order to handle these conflicts, a forwarding unit was used, which is an additional hardware that sends the updated data to the instruction that needs it. A hazard detection unit was also added; it verifies the need to insert a bubble (which then stops, in the next cycle, the PC from being updated and also stops the pipeline register IF/ID from being written and writes a nop[1] in register ID/EX's control signals).

[1]An instruction that does no operation to change state [5].

3.1 HANDLING BEQ AND JR CONFLICTS

There are three types of conflicts involving BEQ and JR:

- **No Stalls**: The BEQ or JR instruction is in the IF/ID pipeline register; The instruction will be executed during the next cycle; The data that will be used by the instruction was generated in previous cycles (previous instruction), but has not been written in the bank register yet.
- **One Stall**: The BEQ or JR instruction is in the IF/ID pipeline register; The instruction will be executed during the next cycle; The data that will be used by the instruction (BEQ or JR) will be generated by a previous instruction in the next clock cycle.
- **Two Stalls**: The BEQ or JR instruction is in the IF/ID pipeline register; The instruction will be executed during the next cycle; The data that will be used by the instruction will be generated two clock cycles after (case where the LOAD is the instruction preceding BEQ or JR).

The pipeline execution of BEQ and JR instructions are very similar: (i) They are both jumping instructions; (ii) Their address calculation is performed in the decoding pipeline stage; (iii) Regarding other instruction, they have the same data dependencies. The only difference is that BEQ must to verify a branch condition which must be true in order to the jump to be taken. So to handle conflicts involving such instructions will do the same verification.

In the first case presented above, as the data is already available, the conflict can be resolved only using the forwarding unit. It will detect this conflict by checking if one of the source instruction's register RS and/or RT (stored in the IF/ID pipeline register) is the same target registers used by the instruction currently stored in EX/MEM and/or MEM/WB pipeline registers. It is also necessary to verify if the instruction currently stored in EX/MEM pipeline register is not a LOAD instruction. In the case of data dependency with a LOAD instruction the data present in the EX/MEM pipeline register is an address. After such as verification, a forwarding[2] must to be performed taking the data from the EX/MEM or MEM/WB pipeline register, thus solving the data conflict.

In the second case, the date has not yet been generated, an stall is necessary. The data needed to the comparison of BEQ instruction or to the address calculation of the JR instruction will be generated in the next clock cycle. The hazard detection unit verifies the need for a stall by comparing the source register in the IF/ID pipeline register with the target register in the ID/EX pipeline register. If The instruction in the D/EX pipeline register is a LOAD instruction a stall cycle is inserted but the conflict is not yet handled.

In the third case, the instruction preceding BEQ or JR is a LOAD instruction. Ins this case two stalls are necessary. The first one is inserted following the verification described in the previous case. The second stall cycle is inserted when

[2]Anticipate the data

Table 1: Major Conflicts

	Type I or R (except LW, BEQ and BNE)	LW
`Type R or I (except for LW, BEQ and BNE)`	Conflict happens when at least one of the operands is equal to the target register of the instruction which precedes it. It is solved with forwarding, and no bubble.	Conflict happens when the target register of the LW is equal to one of the operands of the next instruction and if the LW precedes this instruction. It is solved with the insertion of a bubble and forwarding in the next cycle.
`BEQ`	Conflict happens when the type R or I instruction (except for LW and branch) precedes the BEQ and if the target register is equal to one of the operands of the BEQ. It is solved with the insertion of a bubble and with forwarding in the next cycle.	Conflict happens when the LW preceder the BEQ and the target register of the LW is equal to one of the operands of the BEQ. It is solved with the insertion of two bubbles and with forwarding in cycle following the insertion of the second bubble.
`JAL`	Conflict happens when the target register of the types R or I instruction (except LW and jumps) is equal to the operand register of the JAL and if the type R instruction precedes the JAL. It is solved with the insertion of a bubble and with forwarding in the following cycle.	Conflict happens when the LW precedes the JAL and if its target register is equal to JAL's operand register. It is solved with the insertion of two bubbles and with forwarding in the next cycle following the insertion of the second bubble.

the Hazard unit verifies that the instruction in the MEM pipeline stage is a LOAD instruction and the source register stored in the IF/ID pipeline register is the same target register stored in the EX/MEM pipeline register. Ins this case the data in the EX/MEM pipeline register is an address. After the second stall cycle, the data will be forwarded, thus solving the data conflict.

4. THE DATA PATH

The proposed data path implementation can be seen in figure 1. This is a simplified figure, i.e., it hides some parts that would make the figure too complex and are not really necessary to understanding the work. For example, how was handled the conflicts involving BEQ and JR. To handle such conflicts is necessary adequately select the inputs of mux 3, mux 4 and mux 5, additionally the forwarding unit and the hazard unit must to work properly.

As it was seen in the previous section, there are three type of conflicts involving BEQ and JR. When the first case happens, the forwarding unit treats the conflicts sending signals to mux 3 and mux 4 to select the correct inputs, that can be: (i) Data from ALU (EX/MEM pipeline register); (ii) Data to bank register (MEM/WB pipeline register) or; (iii) Data from bank register. Thus, the data pointed by RS and RT instruction fields will be corrects.

When the second or third case happens the hazard unit is responsible to insert a pipeline stall. The Hazard unit sends a signal to mux 5. The mux 5 has as inputs control signals for a nop operation and control signals for the decoded instruction. If the Hazard unit detects a conflict, it will select the nop operation and thus the pipeline stall is inserted. Therefore, conflicts involving BEQ and JR can be executed and handled.

5. RESULTS

In order to includes in the work's goals a performance and area comparison, a description of a multi-cycle MIPS microprocessor was developed. The performance analyses was based in simulation results and the area analyses was based in synthesis results. The simulation was performed using the ModelSim [1] tool and the synthesis was performed using the Quartus II [1] tool. The benchmark application used in the simulation was a matrix multiplication. It was coded in C language and compiled with the GNU Cross-COMPILE [4]. Among others features the GNU Cross-compiler needs configuration to informs how the memory space must to be used. It generates a .elf file [8], so a ELF2MIF tool was developed to obtain a .mif file [6] that was compiled with the VHDL description.

$$C = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Although it is a simple benchmark application, the assembly generated by the GNU Cross-COMPILE has 257 instructions. To the performance analysis goal stated in the paper this simple application is sufficient enough. The pipeline execution took 1038 cycles, 52 of those being nop operations inserted due conditional branching instruction. Whenever a conditional branch is taken a stall is inserted. and 48 other nops because of data conflicts, thus having 100 cycles of pipeline stalls. This performance for a pipelined execution is satisfactory, as only about 10% of the total cycles were bubbles by the end of the execution.

However, these bubbles don't affect performance so much when compared to the multi-cycle execution, which took 3171 cycles, over three times the number of cycles of the pipelined execution. With that, it can be shown the superiority in performance of the pipeline, reducing to less than $\frac{1}{3}$ the amount of cycles needed for this execution of a matrix multiplication.

One should note that the pipeline's performance can still get better, since 48 of those bubbles were generated because of

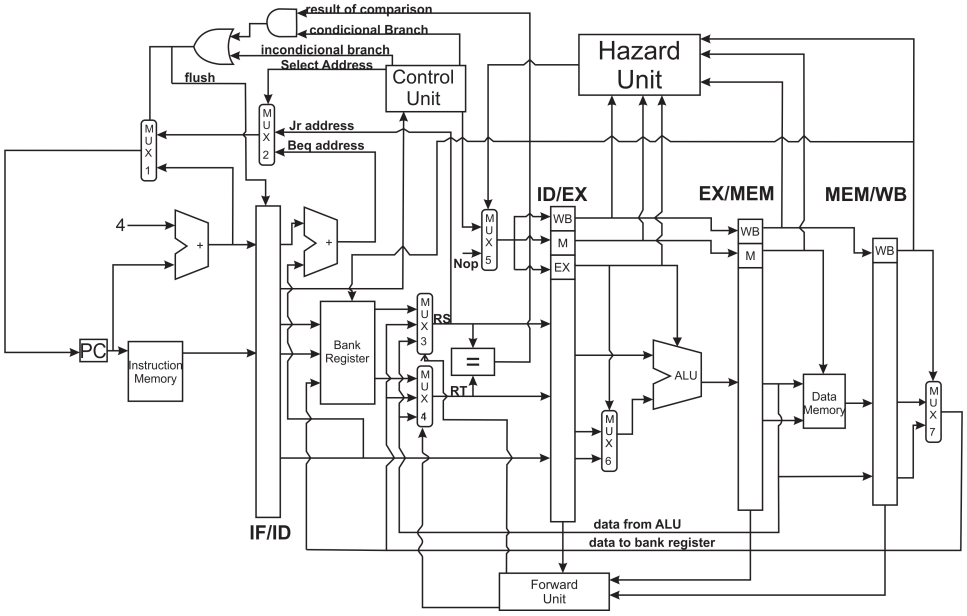

Figure 1: Data Path

```
1  int main(){
2      register int a=0,b=0,i=0;
3      for(i=0; i < 2; i++){
4          a++;
5          b++;
6      }
7  }
```

Figure 2: C code

```
18: 08000009  j 24 <main+0x24>
1c: 00000000  sll zero,zero,0x0
20: 26100001  addiu s0,s0,1
24: 2a020002  slti  v0,s0,2
28: 1440fffd  bnez  v0,20 <main+0x20>
```

Figure 3: Assembly MIPS

the way the Cross-COMPILE generates the assembly. On *for* loops in the C code, when compiled and generated the MIPS assembly, Cross-COMPILE always generates instructions SLTI preceding a BNEZ, a pseudo-instruction equivalent to the BNE, the difference being it uses *zero* one of its operands, thus creating a bubble for every iteration of that loop. This can be seen in figures 2 and 3. In order to better the performance, the Cross-Compile could put an independent instruction between the SLTI and the BEQ, thus avoiding the bubble.

In order to compare in area, it was considered the number of logical elements for each microarchitecture implementation. For the pipeline implementation, 3223 logical elements were used, which corresponds to 47% of the number of logical elements used in multi-cycle implementation, which was 6806. This area result is probably due to implementation coding style. The multi-cycle implementation uses custom megafunctions from ALTERA (lpm_mult, lpm_divide, lpm_csshift). Despite some additional hardware that the pipeline requires to handle conflicts, the number of logical components was smaller than the one on the multi-cycle. That shows that not only is it efficient, the pipeline implementation also needs less space on a chip.

6. REFERENCES

[1] Altera. quartus ii handbook version 13.1 - volume 3. `http://www.altera.com/literature/hb/qts/qts_qii5v3.pdf`.

[2] Mips technologies inc. mips® architecture for programmers volume i-a: Introduction to the mips32 architecture. Sept. 9, 2013.

[3] Wikipedia mips architecture. `en.wikipedia.org/wiki/MIPS_architecture`. Accessed: 19/03/2014.

[4] I. Cpmware. Building a gnu gcc cross compile. `www.cpmware.com/Docs/BuildingGcc.pdf`. Accessed: 31/07/2014.

[5] D. A. Patterson and J. L. Henessy. *Computer organization and design the hardware/software interface*. Morgan Kaufmann, Oxford, USA, 5th edition, 2014.

[6] Quartus. Memory inicialization file. `http://quartushelp.altera.com/13.0/mergedProjects/reference/glossary/def_mif.htm`. Accessed: 31/07/2014.

[7] A. S. R. J. Sagar Bhavsa, Akhil Rao. A 16-bit mips based instruction set architecture for risc processor. *International Journal of Scientific and Research Publications (IJSRP)*, Volume 3, 2013.

[8] E. Youngdale. The elf object file format: Introduction. *Linux Journal*, 1995.