Pipeline Hazards Resolution for a New Programmable Instruction Set RISC Processor

Hajer Najjar[1], Riad Bourguiba[2]
Tunis El Manar university, National engineering school of Tunis, LR11ES20 Laboratory of Analysis, Design and Control of Systems (LACS), 1002, Tunis, Tunisia

Jaouhar Mouine[3]
Prince Sattam Bin Abdulaziz University
Saudi Arabia

***Abstract*—The work presented in this paper is a part of a project that aims to concept and implement a hardwired programmable processor. A 32-bit RISC processor with customizable ALU (Arithmetic and Logic Unit) is designed then the pipeline technique is implemented is order to reach better performances. However the use of this technique can lead to several troubles called hazards that can affect the correct execution of the program. In this context, this paper identifies and analyzes all different hazards that can occur in this processor pipeline stages. Then detailed solutions are proposed, implemented and validated.**

***Keywords*—Processor; RISC; hardware; instruction set; pipeline; hazards; branch predictor; bypass**

I. Introduction

The use of smart digital devices in almost all fields of everyday life increases the embedded systems challenges especially those of microprocessors. In fact, they have to cope with a wide panel of applications and provide at the same time, high performances in terms of operation frequency, energy consumption and area occupation. This complicates extremely the design process and affects the cost and the time to market. On the other hand, these processors are unsuitable for applications with hard real-time constraints. Application Specific Instruction set Processors (ASIP) [1][2] were proposed as an alternative that uses an additional instruction set addressing a defined domain. Thus, compared to the general purpose processors, they provide a significant acceleration but only for a limited class of applications. For better reuse level, designers introduce the configuration concept to make processors customizable and thus fit a large panel of applications. The main proposed solution is to couple a programmable functional unit to the hardwired processor [3][4][5].

In this context, a new configuration technique is introduced. Actually, a hard-wired processor based on a 32-bits RISC architecture is designed then improved to make it able to cope with customizable instruction sets [6]. For better performances, the pipeline technique is implemented to insure better run-time acceleration. However, a processor with pipelined architecture handles many instructions at the some clock cycle which can lead to some execution troubles. Especially when there are dependencies or resources conflicts between these instructions. Such troubles are called hazards since they can randomly happen during the program execution [7][8][9].

In this paper the different hazards related to the architecture of the designed programmable processor are analyzed and classified then the proposed solutions are detailed, implemented and validated. In the second section, an overview of the programmable processor design is presented. In the third section we the data-path is explored in order to find out the different hazards that can happen and classify them. The fourth and fifth sections are respectively dedicated to the implemented solutions for control and data hazards. Then, in the next section the simulation results are illustrated and discussed and finally conclusion and outlooks are presented in the last section.

II. Architecture Overview

The proposed architecture consists of a 32-bits RISC processor based on a programmable ALU that can be customized to handle a large range of instruction sets thanks to the look up table (LUT) technology. In fact, the ALU is composed of a paged LUT where each page contains the truth table of a single operation. Being made of SRAM, these pages can be rewritten if needed to fit any specific application. As for a standard RISC processor we use register/register architecture where the ALU is connected to registers to retrieves its non-immediate operands and store its computing results. This registers are assembled into a register bank. On the other hand, the memory presents two independent interfaces for data and instructions since the processor is designed according to the Harvard architecture.

This processor implements instructions belonging to three different kinds that are namely:

- Arithmetic and logic instructions: They include arithmetic, logic, shift and comparison instructions. The operands are either immediate value or from register bank. The result is stored in the general-purpose registers.
- Instructions for memory access: The address is calculated using a base value and an offset. The address of the register containing the first data and the immediate value representing the offset are both extracted from the instruction word.
- Jump and branching instructions that can be conditional or unconditional.

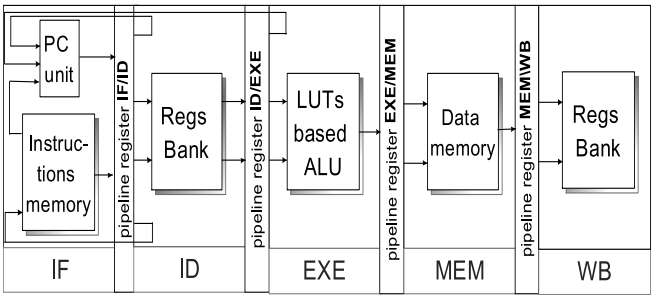

Fig. 1. Pipeline Stages of Our Programmable Processor.

The instruction word is 32-bits length and its structure depends on the data it includes. The R format is composed from the operands and result registers address besides the instruction codes and the shift amount for shift operation. It includes mainly arithmetic and logic instructions with non-immediate operands. Indeed, all instructions using an immediate value, whether for ALU operations, memory access or conditional jump, belong to the I format. Finally the J format is reserved to the unconditional jump instruction.

As shown in Fig.1, all units of the targeted architecture are organized into five balanced pipeline stages representing the five execution phases of an instruction. These stages are:

- Instruction Fetch (IF): Selection and extraction of the instruction from the instruction memory.
- Instruction Decode (ID): Decode of the instruction and extraction of the operands from the register bank in order to be sent to the ALU.
- Execution (EXE): The instruction result is computed.
- Memory access (MEM): The data load and store operations are achieved.
- Write back (WB): The instruction result is stored in the register bank.

III. Pipeline Hazards

The use of pipeline datapath, although it insures a significant computing-power improvement of the processor, generates several execution troubles organized in three hazards categories [10][11][12]:

- Structural hazards: They consist on resources conflicts. In fact, they happen when two different instructions being performed by the processor need to use the some resource at the same clock cycle. For example, if the processor provides only one interface to communicate with memory, it can be requested, at the same time by the instructions in the IF and the MEM stages to access respectively, an instruction and a data. None of these hazards are present in our processor thanks to its design.
- Control hazards: While the branch decision is being calculated, wrong instructions can be introduced into the pipeline which leads to this type of hazards. The branch prediction technique is used to reduce the frequency of wrong instructions fetch.
- Data hazards: They happen when an instruction needs to use a data that is not yet available because it is still being computed by a previous instruction in the pipeline. To solve these hazards, the simplest way is to stall partially the pipeline until the data is ready. However, the frequent use of this technique can lead to an important decrease in terms of pipeline performances. For that, hardwired solutions are implemented to avoid the use of idle cycles as much as possible.

IV. Proposed Branch Prediction Solution

Branch prediction is used to prevent the control hazards. In fact, our programmable processor handles branch instructions that need some clock cycles to decide about the next fetch address. In fact for the conditional branch, the decision is calculated at the EXE stage and even for the unconditional branch the jump address can't be known before the ID stage. Meanwhile, the following instructions are injected into the pipeline. Thus, each time a jump decision is made these wrong instructions must be eliminated. The excessive occurrence of this process considerably decreases the processor run-time performances. Therefore, a branch prediction unit is inserted in the IF stage to speculate the branch decision and to load into the pipeline the most likely true instructions.

A. Static Prediction of the Branch Decision

A first branch prediction approach is the static prediction [13]. Indeed, this method uses a fixed speculation algorithm that is based on the classification of branch instructions in some with a high probability of being taken and others that are often or always not taken. An example of a static prediction consists in classifying the instructions according to their nature: conditional and unconditional. This way, the unconditional branch will form the family of instructions whose prediction is always "taken", while the conditional branch instructions will always be predicted as "not taken". This heuristic based on inaccurate criteria of instructions is not reliable enough because a large number of conditional branches are often taken specially those used for loop management. For this purpose other more precise algorithms are used such as the BTFN (Backward Taken Forward Not taken) which predicts that all the backward jumps are taken. Thus, for the instructions related to the loopback condition, the prediction is always correct as large as the loop is going on and is false only when exiting it. Although the performances enhancement, this algorithm is unsuitable for the instruction sets that rarely use iterations. Generally, the performance of the static prediction is limited because it does not consider the application proprieties and the instructions diversity.

B. Dynamic Prediction of the Branch Decision

An alternative the dynamic branch predictor can be use instead of the static one. The different methods belonging to this category are based on a real-time learning. In fact, decisions history related to each branch instruction are saved then used to better predict future decisions.

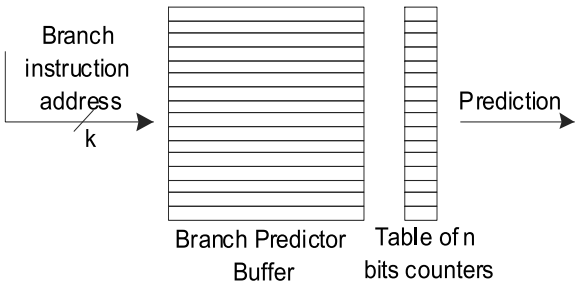

Fig. 2. One-Level Branch Predictor.

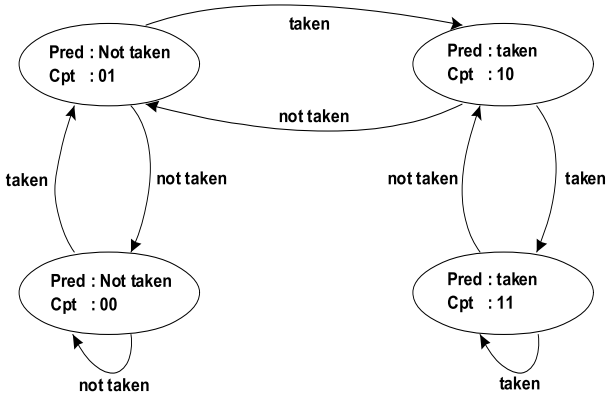

Fig. 3. FSM for the 2-Bits Saturation Counter.

The simplest dynamic predictor is based on the *one-level branch predictor* [11][14]. As shown in Fig.2, this predictor uses a cache memory called *Branch Predictor Buffer* (BPB) or *Branch history table*, which is indexed by the least significant bits of the branch instruction address. This table maps to each address an n-bit *saturation counter* that functions as follows:

- Whenever a branch is taken, the counter increments unless it is already at its maximum value.
- Whenever a branch is not taken, the counter decrements unless it is already equal to zero.

The branch decision is deduced from the value of the *saturation counter*. In fact, if its value is lower than 2^{n-1}, the branch is predicted "not taken" otherwise it is “taken”.

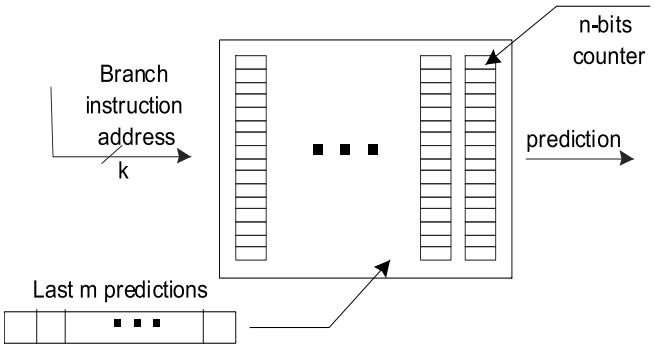

Fig. 4. Correlation-based Branch Predictor.

In the case where a *1-bit saturation counter* is used, the predictor only takes into account the last taken decision (1 if the last time the branch was taken, 0 otherwise). This method is not very reliable because it causes errors even for branch whose decision rarely changes. In fact, if we consider a branch which follows this sequence of decisions: "taken nine successive times, not taken once". In this example, the predictor will commit two errors each time: at the beginning and at the end of the sequence. For better performance, a *2-bit saturation counter* is associated with the *branch address table* [11] [13]. In this case, the predictor operation scheme is managed by the finite state machine illustrated in Fig.3. This way, the prediction changes only if it is false twice consecutively.

In some cases, the program may contain branch instructions whose decision changes frequently and depends on decisions of previous branch instructions. For such situations, it is recommended to use more efficient prediction algorithms that take into account these variations and dependencies. An example is the *two-level branching predictors* [11][14][15]. To this category belongs the *correlation-based branch predictor*. This predictor uses a (m, n) BPB which considers the decisions of the last m stored branches in an offset register to choose a prediction among 2^m . Each one of these predictions is controlled by a saturation counter n bits. Thus, the *two-level branching prediction* consists of a 2^m column table, as shown in Fig.4. For each branch instruction, its address is used for the line selection while the sequence of m bits in the shift register allows the column choice. The value of the counter contained in the selected box allows predict whether the branch will be taken or not. As for the *one-level predictor*, this value will be updated as soon as the decision is calculated. Another *two-level prediction* scheme is the *adaptive algorithm* whose structure is illustrated in Fig.5. This algorithm saves the last m decisions related to each branch instruction and matches each m bit sequence with a *n-bit saturation counter* located in a table named "*Global pattern history*" composed of 2^m lines common to all branching instructions. Thus two branches can be referenced on the same line if they have the same decision history. The *saturation counter* allows, as for all other methods, to predict the branching decision and its value is changed according to the reliability of the prediction.

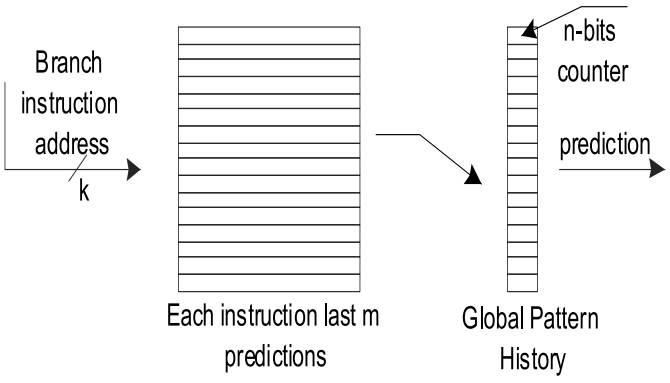

Fig. 5. Two-Level Adaptative Branch Predictor.

The test of a benchmarks range on the various predefined prediction methods shows that they have a variable reliability rate depending on the nature of the different branches [11][13]. Hence the appearance of the hybrid branch prediction which uses two or more independent prediction algorithms operating in parallel [16]. A selection vector, whose value varies according to the history of the branching decisions, allows choose for each branch instruction the most adapted predictor. Other specific predictors characterized by high levels of reliability have been implemented [17][18][19]. Among them there are "*Multiple history length use*"[20], "*de-aliased predictors*"[21], etc.

Among these different branch predictors, the selected solution in order to be implemented is the *one-level predictor* using a *2-bit saturation counter*. This choice is justified by the fact that this predictor is frequently used, it is simple to implement and it presents an acceptable error rate. These factors are sufficient for a first implementation.

C. Branch Direction Prediction

Branch prediction minimizes the impact of late decisions by injecting into the pipeline the instruction corresponding to the decision prediction immediately after the branch instruction. For this, the predictor must know the jump address from the IF stage in the case where a branch is predicted as "taken". However, this address is available only at the level of the ID stage. In order to minimize the latency generated by this address computing delay, a cache memory called "*Branch-Target Buffers*" (BTB) is used[11][14]. Its structure is described in Fig.6. Each line in this memory contains the address of a branch instruction, the jump address, and the corresponding prediction. Whenever a new instruction is present in the IF stage, if its address does not appear in the first column of the BTB, the instruction is not considered as a branch and the execution processes normally. In this case, if the decision corresponds to "taken", a jump to the address of the branch is made. After the execution of a branch instruction, if it is already registered in the BTB, then the prediction is updated according to the decision. Otherwise, the instruction as well as the jump address and the value of the prediction are saved in a line of the BTB. Sometimes the BTB can be saturated. In this case, some branches must be removed to give place to others. As for cache memories, several management algorithms are available. However, a judicious choice must be made to not alter the performance of the predictor.

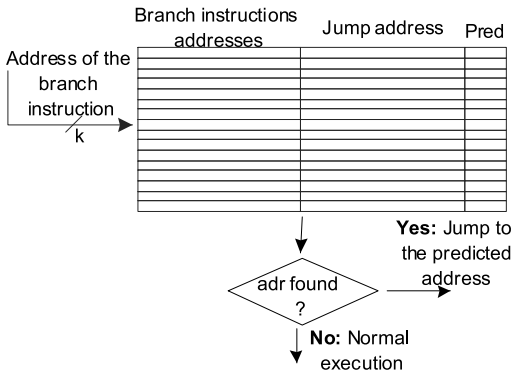

Fig. 6. Branch Target Buffer.

D. Proposed Prediction Algorithm

For the programmable processor, a prediction algorithm that uses the one-level branch predictor and the 2-bit saturation counter was selected for the decision prediction, and the "Branch-Target Buffers" for calculating the branch address. Diagram illustrated in Fig.7 details this algorithm.

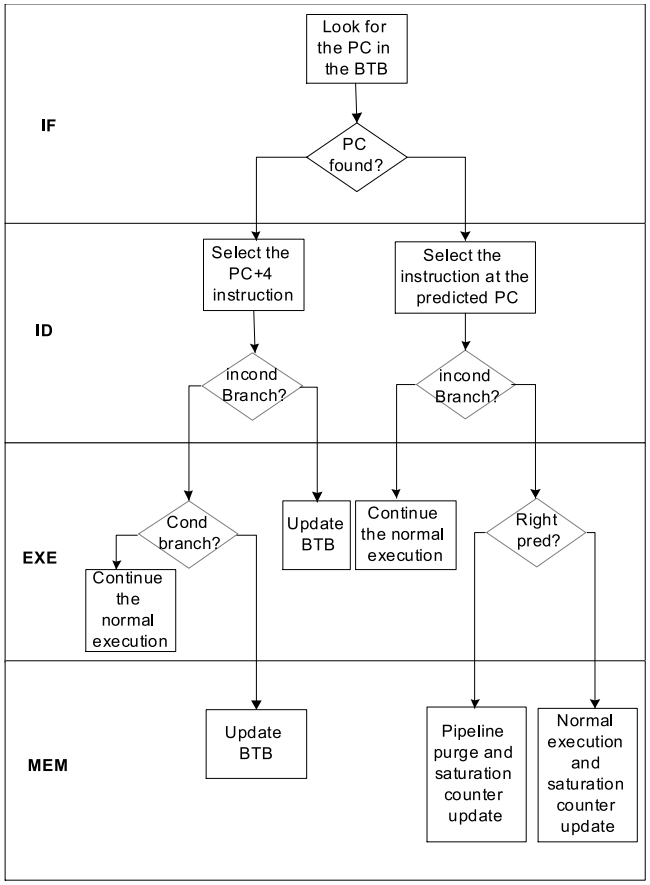

Fig. 7. Prediction Algorithm.

V. Proposed Solutions for Data Hazards

A. Data Hazards Analysis

The resolution of data hazards requires, at first, to study all data-path situations presenting this kind of hazards. For this, first the instructions involved in the happening of these hazards are split into two classes: consumer instructions and producer ones. The first category includes instructions that must use data that is not yet available in the register bank. While the second includes those responsible for calculating the missing results.

It should be noted that an instruction is considered as consumer or producer regarding the pipeline stage in which it needs to use the missing data or produces the required result. For example, for an arithmetic addition instruction, if it needs an operation which is not yet available at the register bank it is "consumer instruction at the EXE stage". However, in the case where a next instruction needs its computation result when it has not yet reached the WB stage, it is considered as "producer instruction at the EXE stage". In the rest of this study, the producer and consumer instructions are respectively named *Res* and *Reg* followed by the name of the stage of the concerned

pipeline (ResEXE, RegEXE ...). In order to identify all stages that can host consumer instructions, we ask the question: "*Does this stage use a data retrieved from the register bank* ". While for producer instructions the question is "*Does this stage produce a result to the register bank?*". Thus, the producer instructions are ResEXE and ResMEM whereas the consumer ones are RegID, RegEXE and RegMEM.

Following this classification, all data hazards that may occur in the proposed processor pipeline are identified by studying all possible scenarios from the point of view of producer instructions. In other words, at the output of each pipeline stage that may contain producer instructions this question is asked: *"which of the following instructions may need to use the result of this stage before it reaches the registers?"*

A producer instruction *i* can belong either to ResEXE class or ResMEM one depending on the pipeline stage that computes its result. This produced data can only be used after it has been saved in a register. Meanwhile, three subsequent instructions will have passed the ID stage and have no longer read access to the register bank. Each of these instructions may need to use the result of instruction *i* in one of the pipeline stages ID, EXE, or MEM. Thus, for each instruction category, nine cases of hazards can occur. All these data hazards are summarized in Fig8. The name of each of these hazards will include both consumer and producer instructions names and their relative position where the producer one is always considered as the instruction *i*. For example, if an instruction must use in the ID stage, the result of a previous one computed in the EXE stage, then, it is the hazard named "ResEXE_RegID_i+1" .

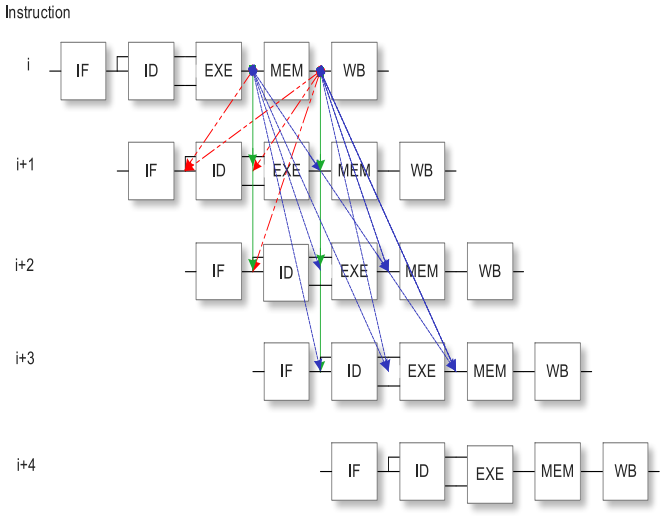

Fig. 8. Data Hazards.

B. Solutions Discussions

In order to solve data hazards, the simplest way is to delay the execution of the consumer instruction until the result of the producer one is available in the register bank. However, the frequent use of this solution degrades considerably the processor performances. Therefore, architectural solutions are proposed to limit as much as possible, pipeline stalling.

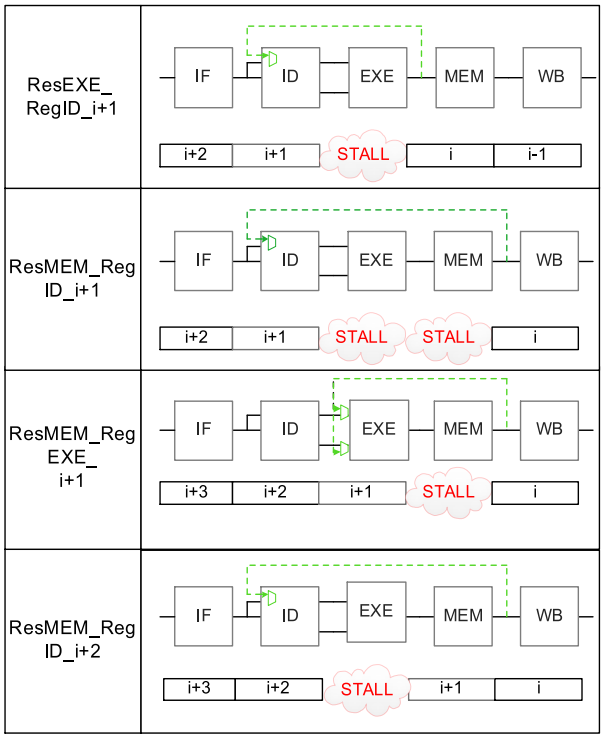

Fig. 9. Proposed Solutions form the First Category of Data Hazards.

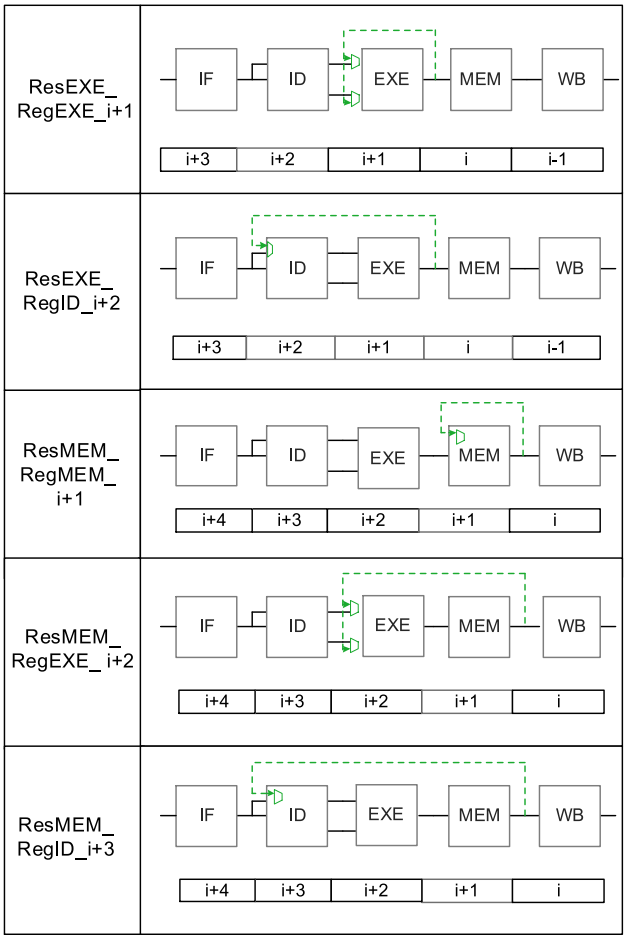

Fig. 10. Proposed Solutions for the Second Category of Data Hazards.

Since there is no common solution to all situations, hazards are classified according to the availability and the position of the result at the clock cycle it is claimed. Three categories are identified:

- The result is not yet calculated.
- The result is ready and has not left the stage where it has been calculated.
- The result is ready and left the stage where it has been calculated but has not yet reached the register bank.

1) First category solutions: The requested data is not yet ready. Therefore, the use of idle cycles is imperative until the result computation ends. Then, a bypass is used to bring back this result to the pipeline stage that requires it. All situations belonging to this category and their proposed solutions are summarized in Fig.9.

2) Second category solutions: In this case, the result is available in the stage where il was calculated. A bypass is simply added to send it back. The different solutions of this hazards category are illustrated in Fig.10.

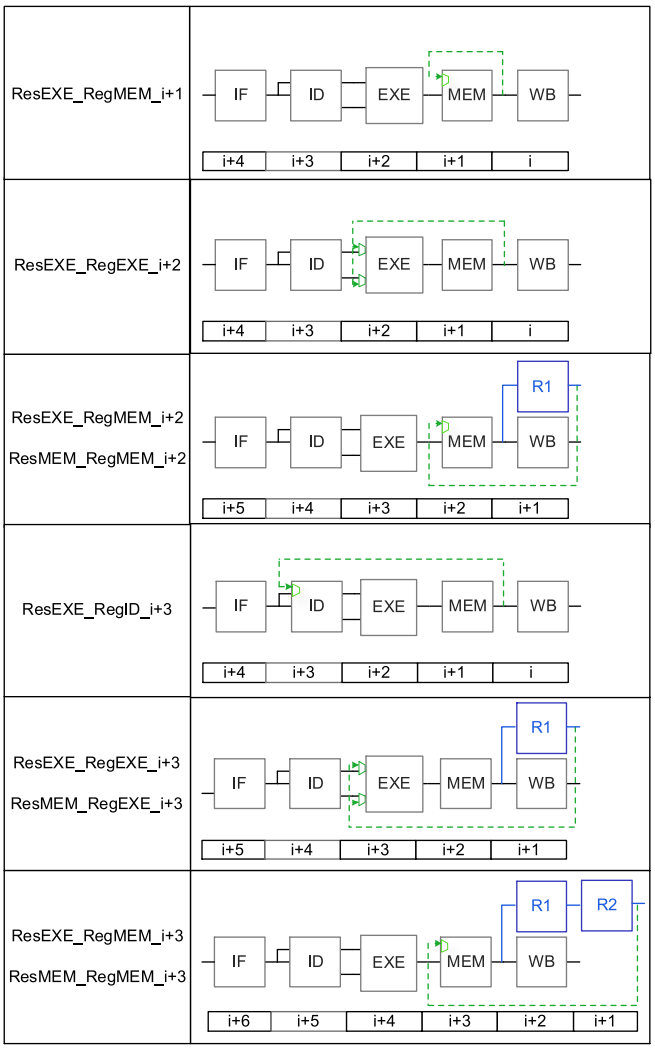

Fig. 11. Proposed Solutions for the Thired Category of Data Hazards.

3) Third category solutions: These hazards occur when the producer instruction result has left its initial pipeline stage but has not yet attained the register bank. Thus, it remains unreachable by the consumer instruction. To solve these hazards, the result must first be located in the pipeline. If the consumer instruction reaches the stage where it need to use the producer instruction result before the latter reaches the WB stages, then, the use of bypass is enough. Otherwise, if the result is already stored in the register bank then it becomes inaccessible because only the instruction in the ID stage has a reading access to the register bank. For such situations, additional registers must be provided to save the result until the consumer instruction requests it. Then, it is brought back via bypass. Fig.11 presents solutions for each instruction belonging to this category.

C. The Implementation of the Proposed Solution

A global solution, built from the specific ones proposed for each hazard, is implemented. In fact, bypasses are inserted into the pipeline architecture and multiplexers are used to select the suitable data.

Besides these architectural modifications, the control of stall cycles as well as multiplexers added to the global architecture has to be insured. However, this management of these parameters depends on several factors such as instructions and hazard types. For this, the control of each pipeline stage is analyzed separately. In the absence of hazards, the data is directly extracted from the previous stage. Otherwise, the requested data is either available on one of the bypass paths, or it is still being processed. For this last situation one or more waiting cycles must be inserted. Thus the analysis concerns only stages which can host consumer instructions that are ID, EXE and MEM.

D. Data Hazards Control

1) ID stage control: There are six hazards where a consumer instruction is present in the ID stage. Three of them belong to the first category therefore they require one or two waiting cycles before reading the data from one of the bypass. The other three are from the second and the third categories, so the data is read immediately from one of the ID stages inputs. Thus depending on the hazard, one of three following solutions must be selected:

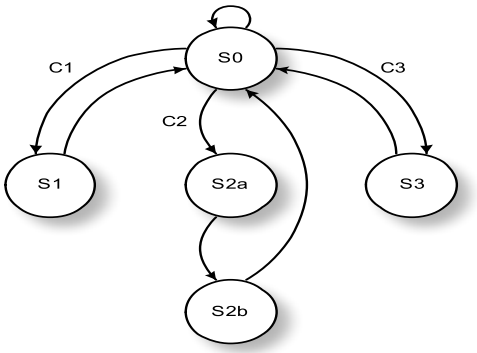

Fig. 12. ID Stage Control.

- Insert 2 stall then read the data.
- Insert 1 stall then read the data.
- Read the date immediately.

These solutions are managed using the FSM presented by the Fig.12. It is composed by five states:

- S_0: Default status, it processes the immediate data reading situations and corresponds to the first waiting cycle if needed.
- S_1: Used by the hazard ResEXE_RegID_i+1 needing only one waiting cycle.
- S_{2a}: It corresponds to the second waiting cycle for the ResMEM_RegID_i+1 hazard needing two stall cycle.
- S_{2b}: Used by the ResMEM_RegID_i+1 hazard.
- S_3: Used by the hazard ResMEM_RegID_i+2 needing only one waiting cycle.

Transitions from S_0 to the other states are controlled by the following conditions:

- **C_1:** stage(i)=ID and TypeInst(i)=RegID and RS(i)=RD(i-1) and TypeInst(i-1)=ResEXE.
- **C_2:** stage(i)=ID and TypeInst(i)=RegID and RS(i)=RD(i-1) and TypeInst(i -1)=ResMEM.
- **C_3:** stage(i)=ID and TypeInst(i)=RegID and RS(i)=RD(i-2) and TypeInst(i -2)=ResMEM.

2) Control of the EXE stage: Among the six hazards including consumer instruction in the EXE stage, only one belongs to the first category and requires a waiting cycle. In the remaining cases, the missing data is available in one of the bypass paths. Thus the control of the EXE stage is done either by immediate reading of the data from the different inputs, or by inserting one stall cycle before. Since there is not a single common solution in all situations, the FSM illustrated in Fig.13 is used. It has only two states:

- S_0: Default status, it processes the immediate data reading situations and corresponds to the first waiting cycle if needed.
- S1: State that processes the ResMEM_RegEX_i+1 hazard requiring only one idle cycle.

The transition condition is:

C : stage(i)=EXE **and** TypeInst(i)=RegEXE **and**

((NbOpInstr(i)= 2 **and** (RS1(i)=RD(i-1) **or** RS2(i)=RD(i-1))

or (NbOpInstr(i)= 1 **and** (RS1(i)=RD(i-1))) **and**

TypeInst(i-1)=ResMEM.

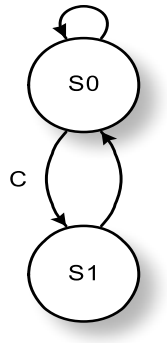

Fig. 13. EXE Stage Control.

3) Control of the MEM stage: The remaining six hazards present consumer instruction at the MEM stage. No waiting cycles are needed for all these cases. Thus the control of this stage consists only on the correct programming of the input multiplexer to choose the suitable data.

VI. Implementation of the Proposed Solutions

The programmable processor as well as the proposed hazards solutions are described in VHDL language. The implementation and validation step were performed using Mentor Graphics Questasim software tool. For this work, we focus only on the validation of techniques we used to solve the different types of pipeline hazards. We choose to study an example of each type.

A. Branch Detection

To validate the branch predictor unit, a code sequence including branch instructions is used. Simulation results are illustrated in Fig.14. *Pred* is the 2-bits saturation counter used by the branch predictor. Its initial value is "01". Thus, the first branch instruction is predicted as "not taken" and no jump is made. When this instruction reaches the EXE stage, the branch is "taken". Then, the instruction at the jump address is injected into the pipeline, *Pred* is incremented and the wrong instructions are neutralized. If an branch instruction is fetched again, then, it is predicted as "taken".

B. Data Hazards Management

This following pseudo-code sequence is used to simulate data hazards:

```
addi  R2, R0, 100
lw    R3 0(R2)
addu R4, R2, R3
```

S_OP1 and *S_OP2* are the controllers of multiplexers at the input of the EXE stage. 0 selects the ID stage data, while 1, 2 and 3 correspond respectively to the MEM and WB stages and the additional register data. Analyzing the simulation of fig.15, it should be noted that the controllers value depend on the required data position. Sometimes, the use of a stall cycle is imperative as for the last instruction.

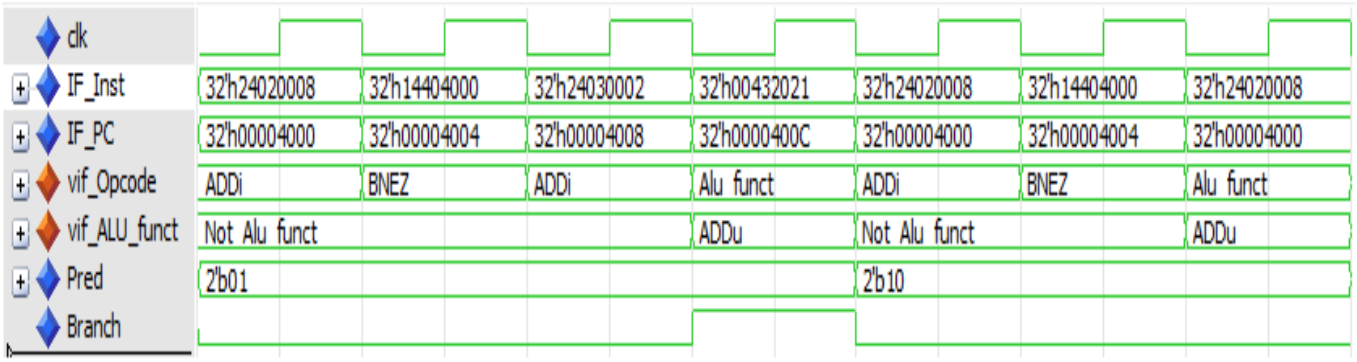

Fig. 14. Branch Detection Simulator.

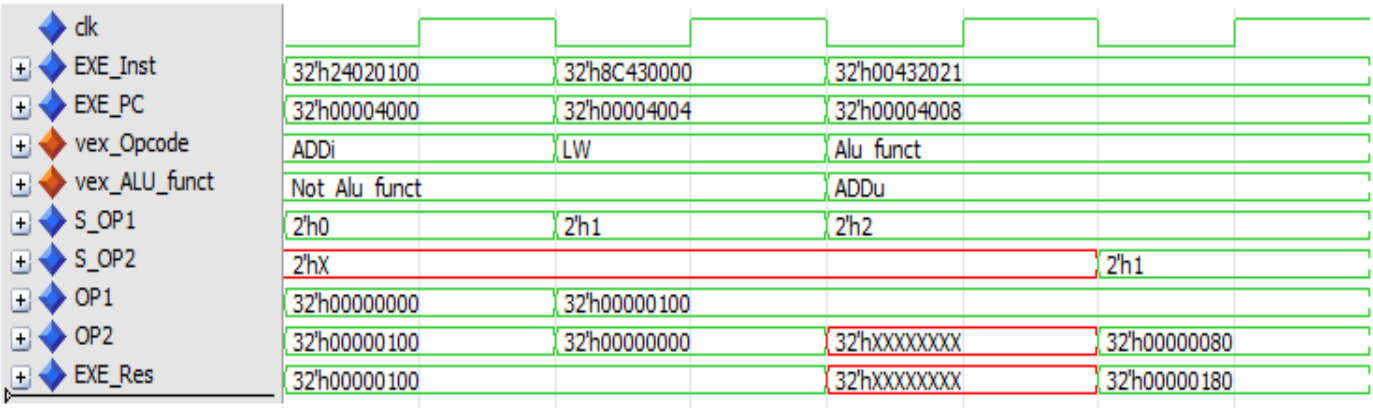

Fig. 15. Data Hazards Control.

VII. Conclusion

The integration of the pipeline technique in the processor architectures is essential to ensure high performances level able to cope with nowadays technology challenges. However, a wrong control of the pipeline stages can disturb instructions execution and alter processor performances. Therefore a precise analysis of hazards related to each pipelined processor architecture, is necessary to be able to implement the suitable control system.

In this paper, a detailed study of the different kinds of hazards that can occur in the pipelined architecture of our programmable processor is presented. Mainly, data and control hazards are identified. Then, solutions for each category are proposed. A prediction algorithm that uses the one-level branch predictor with a 2-bit saturation counter associated to a "Branch-Target Buffers" is used to solve control hazards while for data hazards, bypasses are implemented and controlled using FSMs. These different solutions are implemented and validated using Mentor Graphics Questasim software tool.

In future work, the suitable compiler for this programmable processor will be designed then its silicon implementation will be performed.

References

[1] P. Ienne, R. Leupers, "Automated processor configuration and instruction extension" in Customizable embedded processors, design technologies and applications, 1st ed. Elsevier Science, 2006, ch6, pp.117-142.

[2] M.K. Jain, M. Balakrishnan, A. Kumar, "ASIP design methodologies: survey and issues", inproceedings of Fourteenth International Conference on VLSI Design, Bangalore, India, 2001.

[3] C. Zhang, "Design of Coarse-Grained Reconfigurable Architecture for Digital Signal Processing", PhD thesis, Department of Electrical and Information Technology, Lund University, Sweden, 2009.

[4] N. Vassiliadis, N. Kavvadias, G. Theodoridis, S. Nikolaidis, "A RISC architecture extended by an efficient tightly coupled reconfigurable unit", in International journal of electronics, vol. 93, no. 6, pp. 421-438, 2006.

[5] M.O. Abdulfattah, "Architectural synthesis of a coarse grained run-time-reconfigurable accelerator for DSP applications", PhD thesis, Technische Universität, Darmstadt, Germany, 2006.

[6] H.Najjar, R.Bourguiba, J.Mouine "A new programmable ALU architecture for hard-core processor", International IEEE Multi-Conference on Systems, Signals and Devices, 2016.

[7] D. M. Harris, and S. L. Harris, "Microarchitecture" in Digital Design and Computer architecture, 1st ed. Morgan Kaufmann, 2012, ch. 7, sec. 5, pp. 701-720.

[8] D. A. Patterson, and J. L. Hennessy, "Enhancing performance with pipelining" in Computer Organization and Design, 3ed ed. Morgan Kaufmann, 2007, ch. 6, sec. 2, pp.384-399.

[9] D. A. Patterson, and J. L. Hennessy, "Pipelining" in Computer Architecture and Quantitative Approach, 2nd ed. Morgan Kaufmann, 2012, ch. 3, sec.3, pp. 139-146.

[10] D.A. Patterson and J.L. Hennessy. Computer Organization and Design: The Hardware/Software Interface. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2011.

[11] John L. Hennessy and David A. Patterson. Computer Architecture (2nd Ed.) : A Quantitative Approach. Morgan Kaufmann Publishers Inc., 1996.

[12] D.M. Harris and S.L. Harris. Digital Design and Computer Architecture. Engineering professional collection. Morgan Kaufmann, 2013.

[13] L Gwennap. New algorithm improves branch prediction. MicroDesign Ressources, 9(4), 1995.

[14] B. Lee. Dynamic branch prediction. http ://web.engr.oregonstate.edu/ benl/Projects/branch pred/. Accessed : 2018-01-25.

[15] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. pages 51-61,1991.

[16] S McFarling. Combining branch predictors. Technical report, 1993.

[17] Andre Seznec and Pierre Michaud. A case for (partially) tagged geometric history length branch prediction. J. Instruction-Level Parallelism, 8, 2006.

[18] Andre Seznec. Analysis of the o-geometric history length branch predictor. SIGARCH Comput. Archit. News, 33(2):394-405, May 2005.

[19] Daniel A. Jimenez and Calvin Lin. Dynamic branch prediction with perceptrons. pages 197,2001.

[20] Andre Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. SIGARCH Comput. Archit. News, 30(2) :295-306, May 2002.

[21] Pierre Michaud, Andre Seznec, and Richard Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. SIGARCH Comput. Archit. News, 25(2) :292-303, May 1997.