A 1 GHz MIPS Processor Architecture with Low Flush Rate and Zero Stall

Ahmad Fajar Firdaus, Iman Prayudi, Randy Widialaksono
School of Electrical Engineering and Informatics
Bandung Institute of Technology, Indonesia

Abstract – This paper presents the architecture of a high speed MIPS-ISA RISC processor with optimizations in flush and stall events. The high speed architecture is achieved by pipelining and separating the data-path of address memory calculation from ALU. Flush duration has been reduced to one clock cycle for branch miss penalty and to zero clock cycle for jump instructions. Flush event occurrence has also been significantly reduced using 2-bit dynamic branch predictor. The processor does not need to stall in case of load-use event. We included several new instructions, interrupt/exception handler unit, and interface modules between the processor and the display for supporting various applications. The memory organization and instruction hex code are also designed to be compatible with standard assemblers for fast application implementation. The design has been implemented and verified on FPGA using game software. The processor is implemented using the CMOS 0.18μm technology library that results in 1 ns critical path.

Keywords – MIPS, pipeline, flush, stall

I. INTRODUCTION

MIPS is a RISC microprocessor architecture mainly used in embedded systems. MIPS processor implementation requires a small and simple instruction set. This paper proposes techniques that will speed-up execution of the MIPS processor.

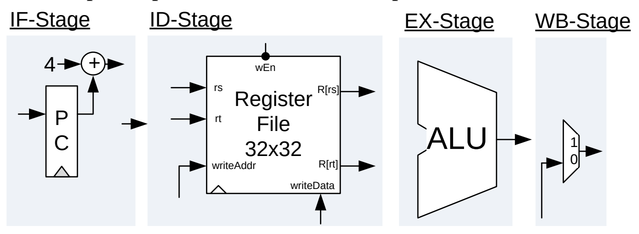

Figure 1 Illustration of 4-Stage MIPS Architecture

The initial step is to design the single cycle architecture. The design consists of four stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), and Write Back (WB) as illustrated in Figure 1. The *Program Counter* generates the Instruction Memory (IM) address to extract the 32-bit instruction in the IF stage. The fetched instruction is then decoded in the ID stage and values in the *Register File* (RF) unit are also fetched. All operations (e.g. add, or, and, sub, sll) occur in EX stage. These operations receive control signals from the ID stage. Memory Access also occur at the EX stage. Operation result is then written in the *Register File* on the WB Stage.

Due to the long propagation delay of the single cycle architecture, we modified the design by adding pipeline registers and moving the data address calculation adder from ALU. The detail explanation is discussed in Section 2.

In order to improve the speed performance in terms of total clock cycles in execution, we reduced flush occurrence and eliminated the need to stall. Flush reduction concerns only to the jump and branch instructions. After a branch instruction, fetched instructions are valid only if the branch decision is correctly predicted. We chose the 2-bit dynamic branch prediction to increase the probability of a correct prediction. If the prediction is wrong, then the falsely fetched instructions need to be flushed. The number of instructions to be flushed depends on the location of the branch decision unit. The branch decision unit, which is originally located in the ALU unit of EX stage, is moved to the ID stage. This arrangement reduces the number of flush from two clock cycles to only one clock cycle. In order to eliminate the need of flush on jump instructions, the jump address calculation is moved to the IF stage. The details of modification in pipeline architecture will be discussed in Section 3.

The proposed design has been verified using a board puzzle game called “Sokoban”. To support game implementation we added several new features, which are covered in Section 4. Finally Section 5 covers performance test results of the proposed processor design.

The following figure illustrates the design flow of this project.

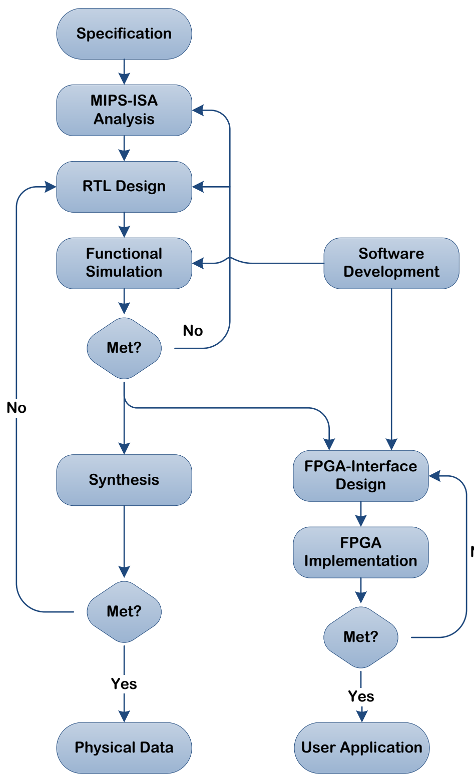

Figure 2 Design Flow

II. PROCESSOR DESIGN ARCHITECTURE

The initial design of the processor is a single cycle processor. It has small area, but has the disadvantage of a long propagation delay. This delay is the main factor that determines the speed performance of the processor. We identified the critical path of this initial design, shown as a red line in Figure 3. The critical path starts from the instruction memory through the control lines to data memory address bus.

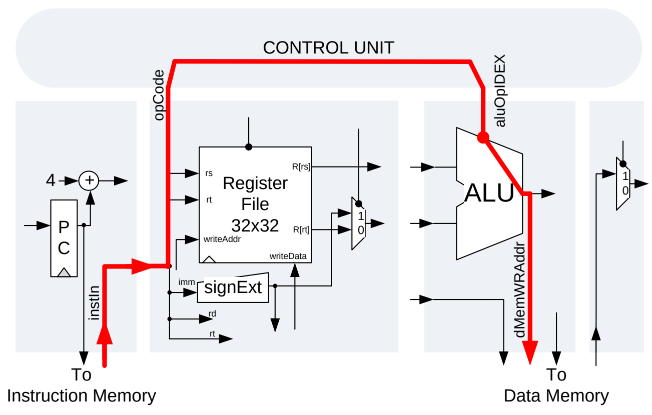

Figure 3 Single Cycle Processor

In order to improve the speed performance in terms of propagation delay, three main steps were done. The first is to add pipeline registers between each stage.

A. Four Stage Pipeline Architecture

Using the pipeline architecture, the critical path is significantly reduced, as shown in Figure 4. The new critical path is from ALU control line to data memory address bus which is significantly shorter than the previous critical path in Figure 3.

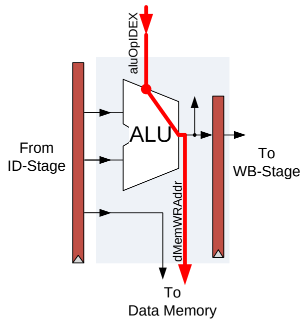

Figure 4 Critical Path in Pipeline Architecture

As consequence of the pipeline architecture, three types of hazards are: structural hazard, data hazard, and control hazard. Structural hazards happen when the hardware is unable to execute two different instructions at the same time. An example case is illustrated in the following figure.

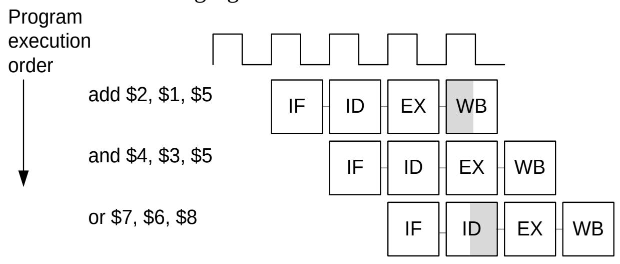

Figure 5 Structural Hazard

In this case, the hardware *Register File* needs to be accessed at the same clock cycle by *Write Back (WB)* stage and *Instruction Decode (ID)* stage. To solve this problem, we separated the time for read access and write access. The write access in *Write Back (WB)* stage is done on the first half of the clock cycle, and the read access of the *Instruction Decode (ID)* is done on the second half of the clock cycle.

The *data hazard* happens when an instruction depends on the result of a previous instruction but the result is not yet available. The result data is yet to be written back to the register file in *Write Back (WB)* stage. The required result data is already valid after *Execute (EX)* stage; hence we forwarded the result data from *Execute (EX)* stage for the next instruction. The following figure shows the data-path of the forwarding unit.

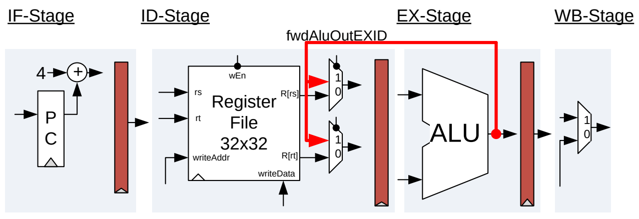

Figure 6 Forwarding Unit Datapath

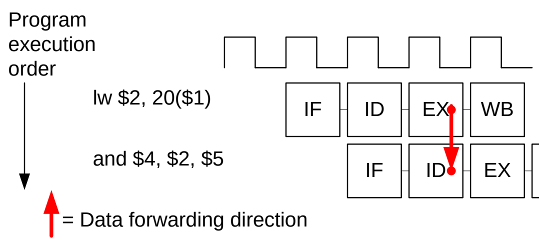

Figure 7 Data Forwarding Scenario

The *control hazard* occurs when the processor needs to make a decision based on the results of an instruction that have not yet finished executing. This hazard occurs on branch instructions, thus we need a mechanism to predict which instruction will be fetched on the next clock cycle. The chosen mechanism for our processor will be explained in detail in Section 3.

B. Data Memory Address Adder Separation

We separated the data memory address calculation from ALU to shorten the delay. By this separation we obtained a 2 ns speed improvement. The new critical path is from ID/EX pipeline register via adder to data memory address bus as shown in Figure 8.

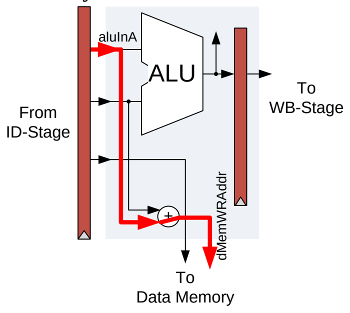

Figure 8 Data Memory Address Adder Critical Path

Because of the new critical path, we replaced the adder with the Kogge-Stone adder [3], which is a parallel prefix type adder and currently regarded as one of the fastest adders for VLSI [4].

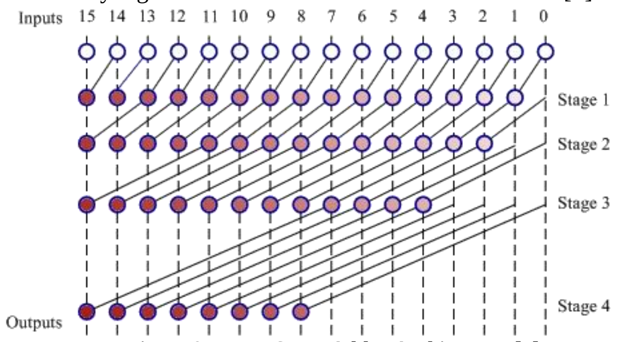

Figure 9 Kogge-Stone Adder Architecture [5]

III. PIPELINE ARCHITECTURE OPTIMIZATION

Flush and stall decreases processor CPI performance. Both of these events can be minimized by optimizing either in software or hardware. In software, the order of instructions are arranged to avoid flush/stall triggering event, which is not concerned in this paper. By optimizing in hardware we can reduce the number of flush and stall regardless the order of instructions is fetched.

A. Flush Reduction

Flush means cancelling a fetched instruction without keeping the current *Program Counter* value. We have reduced the duration of flush required when jump or branch instruction is executed. As a result, the jump instruction no longer needs a flush. In case of an incorrect branch prediction, the flush penalty is reduced from two clock cycles into only one. These improvements are achieved by adding dynamic branch predictor and modifying the architecture.

1. Jump instruction

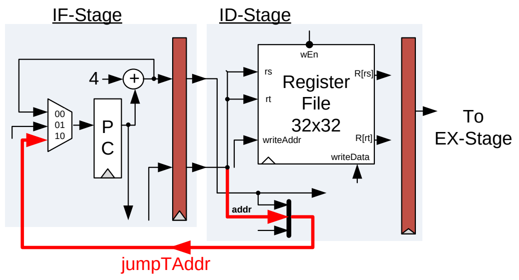

Figure 10 Jump Address Calculation in ID Stage

In the above architecture, jump address calculation is done in the ID stage, while the jump target address has to be valid one clock cycle before the ID stage. Address *Program Counter*+4 is always fetched when the jump is calculated in ID stage, which is most likely the wrong instruction. Therefore the processor needs to flush one clock cycle.

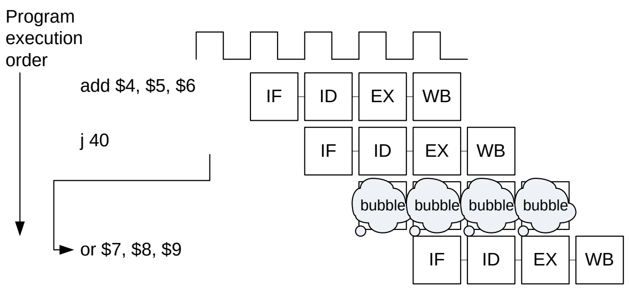

Figure 11: Execution of jump instruction before improvement

Optimization is done by moving the calculation of jump address from the ID stage to the IF stage, as illustrated in below figure.

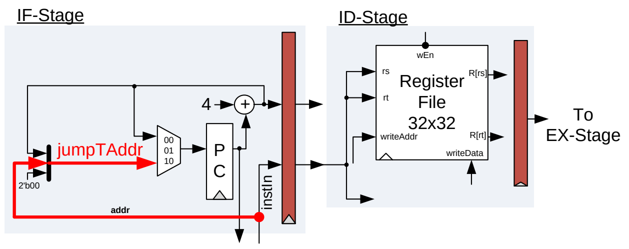

Figure 12 Jump Address Calculation in IF Stage

With this, the need of flush on jump instructions is eliminated.

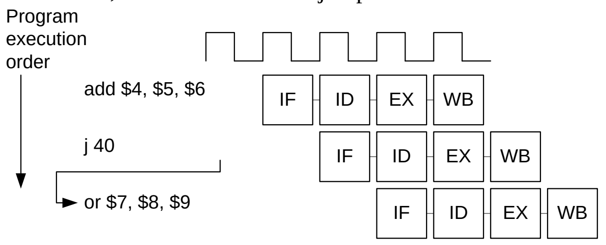

Figure 13: Execution of jump instruction without flush

2. Branch instruction

There are two mechanism of branch prediction in pipeline architectures: static and dynamic. Static branch prediction means the processor always have the same prediction (whether *taken or not taken)* on any clock cycle. This mechanism have very significant disadvantage. Assume the processor always predicts *not-taken*, while a program contains 100 branch instructions that are always taken. The processor will always miss-predict the branch, and do 100 flushes. This means the processor will lose 100 clock cycles for flushing the miss-predicted instructions.

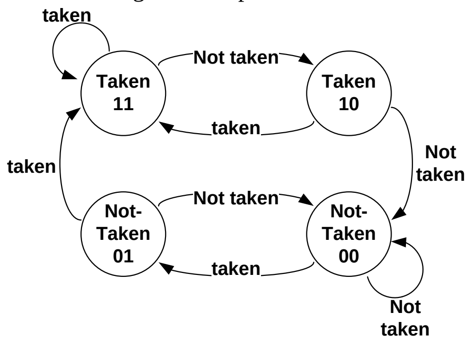

Figure 14 2-bit Dynamic Branch Prediction

The proposed processor design use 2-bit dynamic branch prediction [1]. This mechanism uses a state machine that predicts considering the history of branch instructions, as shown in the following figure.

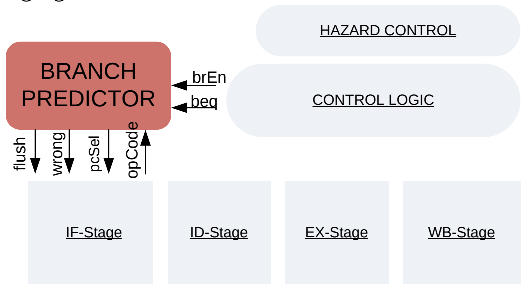

Figure 15 Branch Predictor Module

Assume the initial state is not-taken (00), then for the case above, the processor will only miss predict two times. It will only flush two times, thus it completes 98 clock cycles faster than the static mechanism.

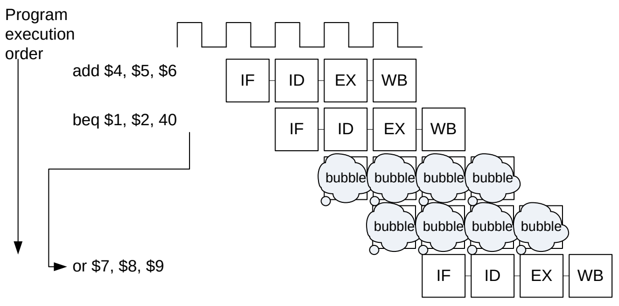

Figure 16 Branch Prediction Miss

Branch instructions, such as branch on equal (*beq*), compare two register values, *rs* register and *rt* register. For *beq* if the two registers contain the same value, the *Program Counter* will jump to the branch offset. The comparison of the two values occurs in the ALU. Because the ALU unit is located in the EX stage, the decision on whether to take the branch or not occurs on the third clock cycle. When the processor miss-predict the comparison result, then the two previously fetched instructions needs to be flushed [1]. This event is described in Figure 16, with *always not-taken* prediction mechanism in which the next instruction fetched is always *ProgramCounter+4* address.

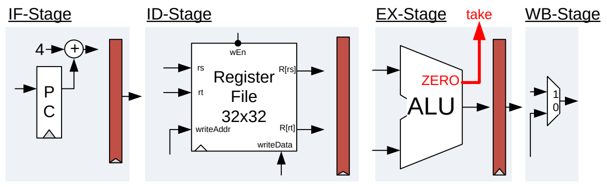

Figure 17 Branch Decision in EX Stage

The following figure illustrates the architectural change in which a comparator is used in ID stage to decide whether the branch is to be taken.

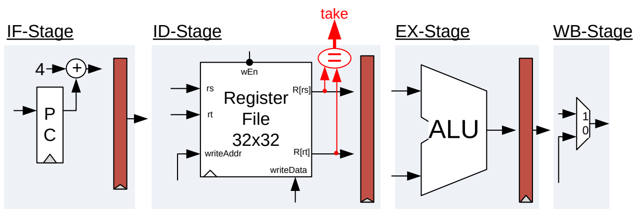

Figure 18 Branch Decision in ID Stage

Because of this architectural change, the correct next instruction can be fetched one stage earlier. Thus the number flush needed is reduced to one clock cycle.

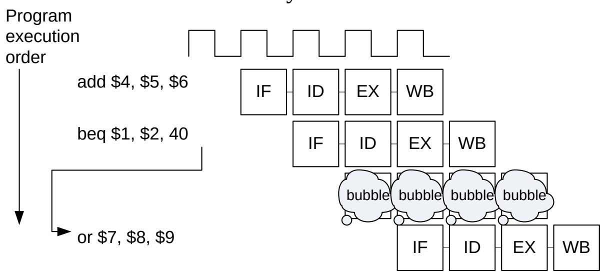

Figure 19 Reduced Flush on Branch

B. Elimination of stall

Stall means cancelling a fetched instruction while the *Program Counter* is kept unchanged. Load-use event occurs when a *load word* instruction is followed by an arithmetic instruction (e.g. *add, sub, and, or,* etc) which needs the data from *load word* execution. In case a load-use event occurs, a stall is needed because the valid data from *load word* is provided on the WB stage. On the EX stage of this *load word* instruction, the data to be fetched is already needed by the next instruction. One solution is to do a stall and forward the data from WB stage to the ID stage [1].

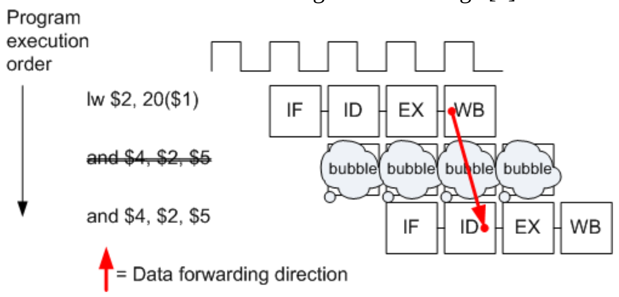

Figure 20 Stall on Load-Use case

Stall will add an extra clock cycle to execute a load-use, and this will affect the processor performance.

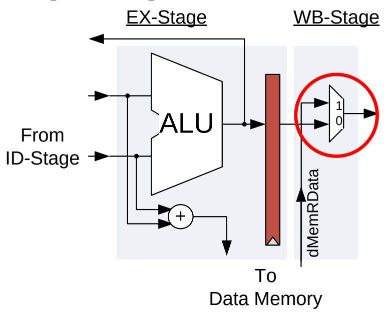

Figure 21 EX-WB Stage, Before Movement

Architecture optimization is done by moving the multiplexer on the WB stage to the EX stage. This multiplexer chooses which data enters the *Register File*; the ALU Output or the Data Memory Output. This movement will eliminate the need of stall because the valid data is already provided on the EX stage, hence forwarding can be done from the EX stage to the ID stage.

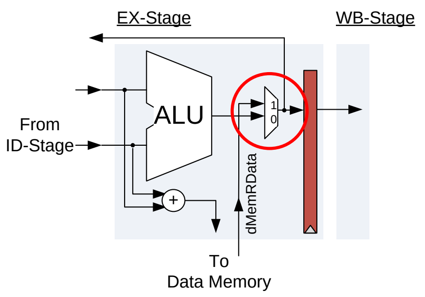

Figure 22 EX-WB Stage, After Movement

Figure 7 shows the scenario of load-use event execution.

IV. FEATURES

We implemented a board puzzle game called “Sokoban” to verify the functionality of the processor. The game is displayed on VGA output and players interact on the FPGA board. The game uses all of the supported instructions and the interrupt handling routine.

A. Assembler Compatibility

The processor is designed to directly run hex code assembled by the MARS Assembler [4]. This is achieved by organizing the memory to fulfill MIPS specification.

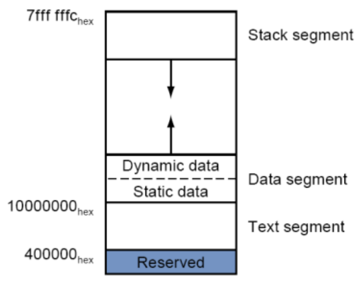

Figure 23: Memory Organization [1]

Normal execution instructions are located inside the text segment. This segment has base address of 00400000_{hex}. Data memory has base address of 10010000_{hex}. The exception/interrupt segment has base address of 80000180_{hex}.

B. Exception/Interrupt Handler

The processor has both exception and interrupt handling routines. Exception is defined as when normal flow of instructions is disrupted by the processor itself, such as an overflow exception. Overflow is detected in the EX stage, and then the overflow instruction and the two next fetched instructions are flushed.

Interrupt is defined as when normal flow of instructions is disrupted by an external source. In our specific game application, 4 external interrupts are needed to represent each of the pushbuttons. Each external interrupt is assigned to a unique *cause register* value. Interrupts and exceptions are handled by the co-processor. The co-processor has registers that contain information needed to describe and handle interrupt/exception. The *Program Counter* will point to address 80000180_{hex} every time an exception/interrupt occurs. The co-processor registers record the type of exception/interrupt and the *Program Counter* address when an exception occurred [1].

C. Extended Instruction Set

Necessary additional instructions are added in order to support our application. The instructions supported are: *Add, Add Unsigned, Subtract, And, Or, Shift Left Logical, Shift Right Logical, Set Less Than, Add Immediate, Add Immediate Unsigned, And Immediate, Or Immediate, Load Upper Immediate, Load Word, Store Word, Branch On Equal,* and *Jump.* Instructions for co-processor communication are also added: *Move from Co-processor, Move to Co-processor,* and *Exception Return.* Refer to Appendix B for register transfer level description of the instructions.

D. Game Application

The software is hand-written MIPS assembly code and hex code is obtained using the MARS (MIPS Assembler and Runtime Simulator) assembler[7].

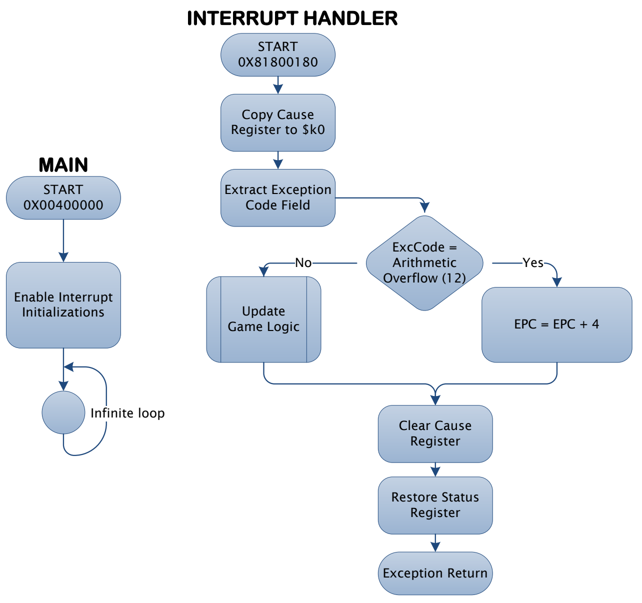

Figure 24 Application Flowchart

In the application, the text segment contains initializing registers and enabling interrupt registers. The game logic is located in the interrupt segment. Each interrupt has unique cause register values. It is then checked by the software and the game logic updates the data memory as necessary. The *Update Game Logic* routine is explained detail in Appendix C. Video display shows the representation of the data memory, which is refreshed 60 times per second. Initial display of the first level of the game is shown in Figure 25.

E. VGA Display

The game display is shown on a VGA monitor. We have created the VGA-Processor interface (shown in Appendix D) so that the game can be played interactively. VGA module reads the data memory of the processor then renders the display. In order for the processor and the video module to operate at the same, the data memory is made dual-port; a read-write port for the processor and a read-only port for the display.

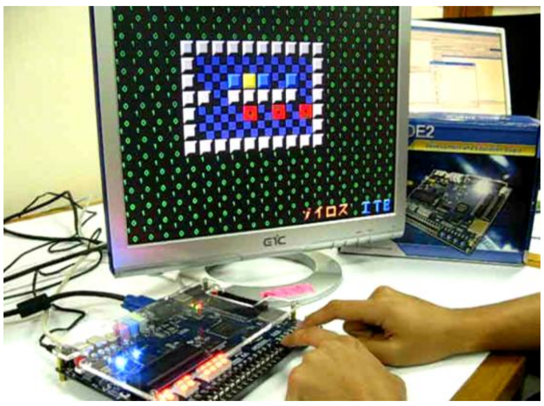

Figure 25 Display of Game Application

V. PERFORMANCE TEST

A. Physical Comparison

Architectures *B_SC, PP_A, PP_B, PP_C, PP_D* and *ZS* are synthesized using 0.18µm standard cell CMOS technology. These designs are made for testing and benchmarking purposes. The following table shows features of the architectures.

DESIGN	Data Memory Address Calculation	Branch Prediction Method	Flush Reduction/ Stall Elimination	Supported Instructions	Exception/ Interrupt Handler
B_SC*	ALU	Always Not Taken	None	10	None
PP_A**					
PP_B**	Adder				
PP_C**	Kogge-Stone Adder				
PP_D**		2-bit Dynamic	Yes		
ZS**				20	Yes

Table 1 Architecture Features (Appendix A)
*** : Single Cycle** **** : Pipeline**

The results of synthesis are shown in the following table.

DESIGN	AREA (um²)	DATA ARRIVAL TIME (ns)	AREA (XOR Unit Area)	DATA ARRIVAL TIME (XOR Unit Delay)
B_SC	164680.0867	3.29	6188.37499	31.83871
PP_A	185300.4404	2.53	6963.24999	24.48387
PP_B	185666.3444	1.08	6976.99999	10.45161
PP_C	192259.2693	0.97	7224.74999	9.38710
PP_D	194867.1669	0.97	7322.74999	9.38710
ZS	226201.8553	1.00	8500.24999	9.67742

Table 2 Physical Comparison of Architectures

B. Software Benchmarking

The performance improvement is benchmarked using various programs. Source code of programs used to benchmark is listed in Appendix C.

From the benchmark results, PP_D has the best performance. Therefore it is chosen to be developed further for user application, which results as ZS design.

1. Flush Reduction

The impact of architectural improvement is emphasized; therefore only PP_C & PP_D architectures are tested. PP_C architecture does not have any flush reduction mechanism, while PP_D does. Therefore we compared them to analyze improvements.

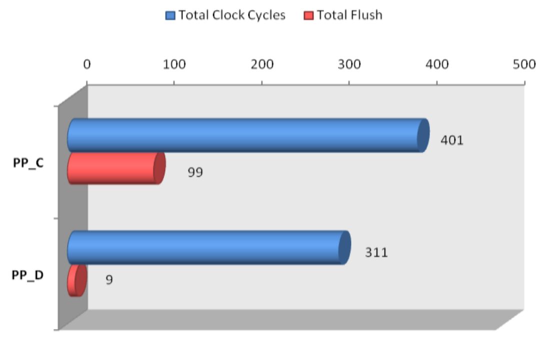

Figure 26 Branch-Flush Reduction Chart (Appendix C. Listing 1)

The program executed for the test in Figure 26 is a nested loop program. PP_C which has two clock cycles flush and static branch prediction mechanism completes the program in 401 clock cycles with 99 flush events. PP_D which has improvements embedded completed the same program in 311 clock cycles with only 9 flush events. Because the program runs 10x10=100 loops, it can be seen that PP_C always mispredicted the branches while PP_D only mispredicted when it exits the inner loop.

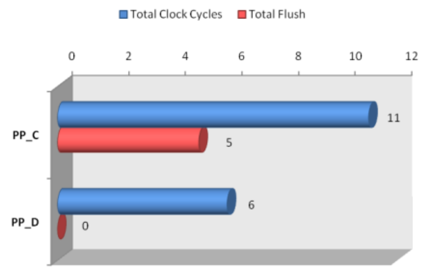

Figure 27 Jump-Flush Reduction Chart (Appendix C. Listing 2)

The program executed for the above test purely consists of five jump instructions. PP_C has to flush each time a jump instruction is met because the jump address calculation is in the second stage of the pipeline. PP_D doesn't need to flush on jump instructions, therefore it can complete the same program faster than PP_C.

2. Stall Elimination

The program executed for stall analysis below consists of loading a value from data memory, immediately increment the value then storing it back to the data memory for 20 values.

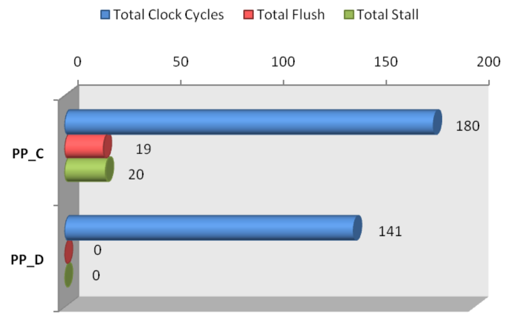

Figure 28 Load-Use Benchmark Chart (Appendix C. Listing 3)

PP_C needs to flush for 19 clock cycles and stall for 20 clock cycles to complete the program. PP_D does not need to flush nor stall at all, therefore it can complete 19+20=39 clocks faster than without the architecture modification.

3. Bubble Sort

The following chart shows results of bubble sort program execution. Stall and flush does not exist in the single cycle architecture. PP_A, PP_B, and PP_C have static branch prediction and no modifications for flush and stall reduction. PP_D has 2-bit dynamic branch prediction with final improvements for flush and stall reduction.

Pipeline architectures need more clock cycles to execute the program than single cycle architectures, but they have significantly faster clock frequency. Thus the execution time is calculated and shown in Figure 30. PP_D finished running the program faster than PP_A, PP_B, and PP_C by the reduced number of flush and stall.

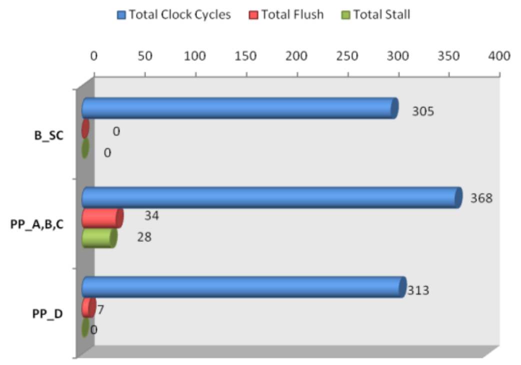

Figure 29 Bubble Sort Benchmarking (Appendix C. Listing 4)

Minimum clock period for all architectures are obtained from synthesis (critical path in ns) with 0.18µm CMOS technology. The equation used to calculate the execution time of the bubble sort program is as follows:

execution time = total clock cycle x clock period (ns)

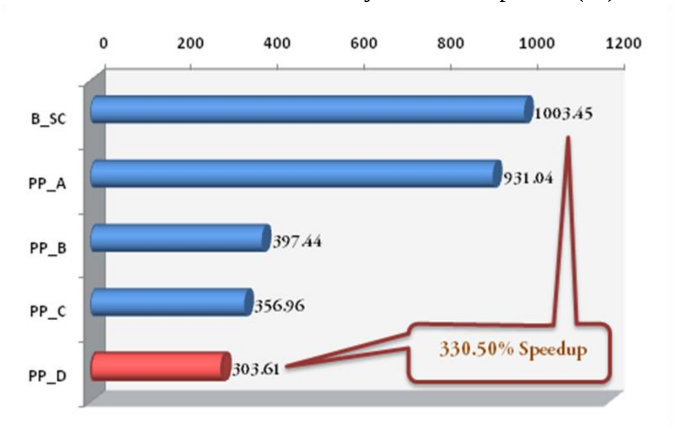

Figure 30 Bubble Sort Execution Time (Appendix C. Listing 4)

PP_D architecture has the best performance, which is 330.50% improvement of faster execution speed compared to the first single cycle design. The simulation waveform of the bubble sort program is shown in Appendix C. Figure 1.

VI. CONCLUSION

This paper has introduced MIPS pipeline architecture with low flush event occurrence and no stall on load-use event. Performance results derived from benchmark programs indicate speed-up compared to without architectural improvements. The final design has exception/interrupt handler and assembler compatibility to support various applications. Synthesis results using 0.18µmCMOS standard cell technology results in maximum core clock frequency of 1 GHz.

VII. REFERENCES

[1] Patterson, David A. Hennessy, John L. *Computer Organization and Design-The Hardware/Software Interface 3rd Edition*. San Francisco: Morgan Kaufmann Publishers, 2005.

[2] Hamblen, J O, T S Hall, and M D Furman. *Rapid Prototyping of Digital System SOPC Edition*. Springer, 2008.

[3] F. Gurkaynak, Y. Leblebici, L. Chaouat, P. McGuinness. *Higher Radix Kogge Stone Parallel Prefix Adder Architectures.* IEEE International Symposium on Circuits and Systems. May 2000.

[4] P.M. Kogge and H.S. Stone, *A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations*, IEEE Trans. on Computers, Vol. C-22, No 9, August 1973.

[5] Knowles, Simon. *A Family of Adders*. IEEE.

[6] Beaumont-Smith, Andrew. Lim, Cheng-Chew. *Parallel Prefix Adder Design*. University of Adelaide.

[7] Vollmar, K. Sanderson, P. *MARS: An Education-Oriented MIPS Assembly Language Simulator*. Missouri State University, USA.