Efficiently Handling of Content in Pipeline Stages on Interrupts in Processors

Arvind Ramshetty[1], S.K. Satyanarayana[2]

[1]PG Research Scholar, [2]Assistant Professor
Dept. of ECE, Sreenidhi Institute of Science & Technology, Hyderabad, TS, India

***Abstract*—This paper evaluates the solutions to the problems that occur in processors when handling Interrupts. When processor enters into interrupt handling mode the Instructions which are present in the various stages of the pipeline must be flushed out and the address of the handler is loaded into PC(program counter). This flushing of the pipeline causes loss of many clock cycles which affects efficiency. This paper tackles this problem and improves the overall performance of the processor. The technique discussed in this paper can also be extended to Branches with return values in conjunction with branch predictors.**

Index Terms: Pipelined processors, Clock Cycles, Precise Interrupts, Stack Space

I. INTRODUCTION

In PipelinedProcessors,instructions are fetched from the memory before the previous instructions are executed and have written the result to the register file or some other memory location, this allows faster CPU throughput and increase in the performance. In a sense the pipelined processors operate on more than one instruction at a time in a single clock cycle by forwarding the instruction to different stages in processor.

Traditionally processors on interrupts halts the normal sequential execution of instructions, like on reset or when the memory access fails etc. Most interrupts have associated interrupt handler (software routine that executes when exceptions or interrupts occurs). Servicing takes place within handler or by branching to a specific service routine. This causes the entire pipeline to flush and fetch the new instructions and then decode and so on. This results in loss of clock cycles in fetching the instructions from memory again since memory accesses are slow which decreases the processing speed; especially in Interrupt-driven embedded (real-time)systems the penalty that have to be paid again in re-fetching and redoing the process is lethal.This can be improved by adding a new set of memorystack(Stack Space)for each stage which upon interrupts or branches moves the information contained in those stages to their corresponding memory and then once the interrupts are serviced the contents of the memory are popped to their corresponding following stages. This prevents from accessing memory again for the instruction to be fetched, decoded and so on.

II. Precise Interrupt Processors[1]

When an interrupt occurs the state of the interrupted process is saved. The state typically consists of program counter, registers and memory. If the saved process is in consistent with the sequential architecture model then the interrupt is precise. The saved state should reflect:

1. All instructions preceding the instruction indicated by the saved program counter have been executed and have modified the process correctly.
2. All instructions following the instruction indicated by the saved program counter are unexecuted and have not modified the process state.
3. If the interrupt is caused by an exception condition raised by an instruction in a program, the saved program counter points to the interrupted instruction.

From the above conditions we can infer that in precise interrupt processors the instructions before the execute stage will be flushed and instructions from execute through Write Back stage will be completed before the control is given to interrupt handler. Before handler takes control the state of the processor must be saved.

III. Design and its Implementation

The design I've implemented has the precise interrupt architecture. It is a 32bit processor consisting of 6 pipeline stages. It contains normal working registers as well as special register for working when interrupts occur. The interrupts(external) are acknowledged in the execute stage. Then this signal is propagated to the Write Back stage through Memory Access stage. Upon receiving the interrupt from Execute stage, the fetch stage is disabled to stop the further fetching of the instructions from memory and the instruction in Decode and Reg. Read and Issue stage are held unless the Interrupt from the write back stage is raised. Once the Interrupt is raised from the WB stage then the data in all three stages are moved to their corresponding stack space and since the state of the processor must also be saved the

Status Register is also moved to its own stack space. After the interrupt is raised the PC is loaded with the address of the interrupt handler. The interrupt handler has its own array of limited registers (Interrupt handle registers) to make sure that the original data in the working registers are not corrupted. Since once the interrupt handler begins to service,if there are no separate registers to handle the interrupts, the handler has to move all the data in the general working registers to a stack memory and retrieve once it has been serviced. This takes many more clock cycles. To avoid this it's better to use a limited number of special registers only for Interrupts.

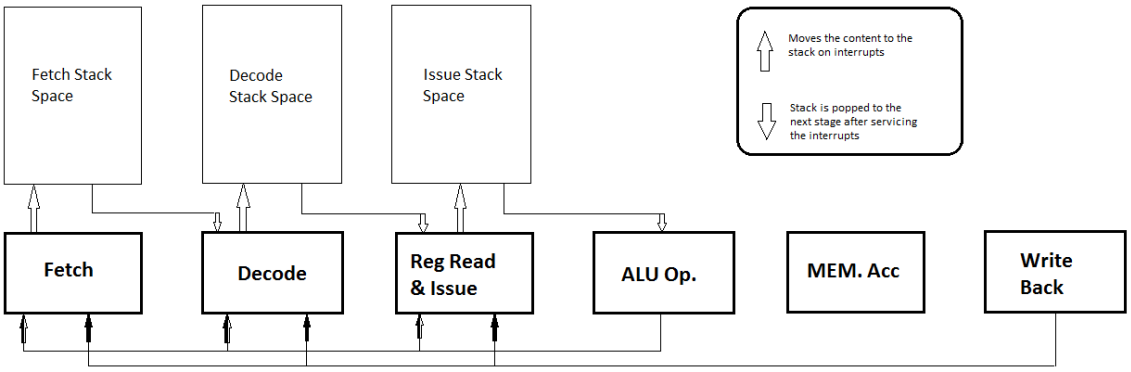


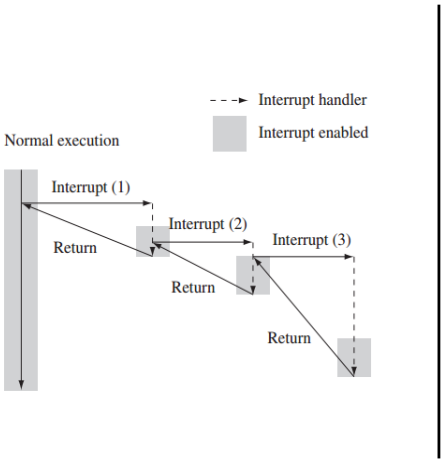
Figure 1 General Overview of how Intermediate Data is handled

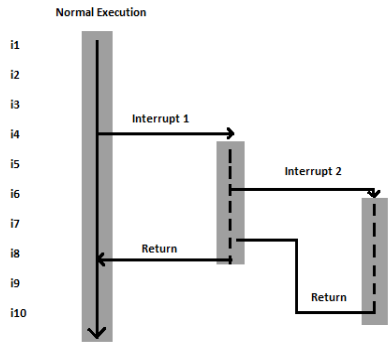
The above figure shows briefly how the content in stages are pushed and popped from their respective stack space before and after interrupts are serviced.

Up Arrow indicates that the data is pushed onto the stack and after servicing, the stacks are popped to their next adjacent stages which are indicated by down arrows.

IV. Nested Interrupts

System architects must design a system that handles the multiple interrupt sources without increasing the latency of the interrupt i.e. the time it takes to respond to an interrupt, else the system performance is degraded. Figure 2 shows how traditional processors handles multiple interrupts, in contrast figure 3 shows how a system with separate interrupt stack space are handled.





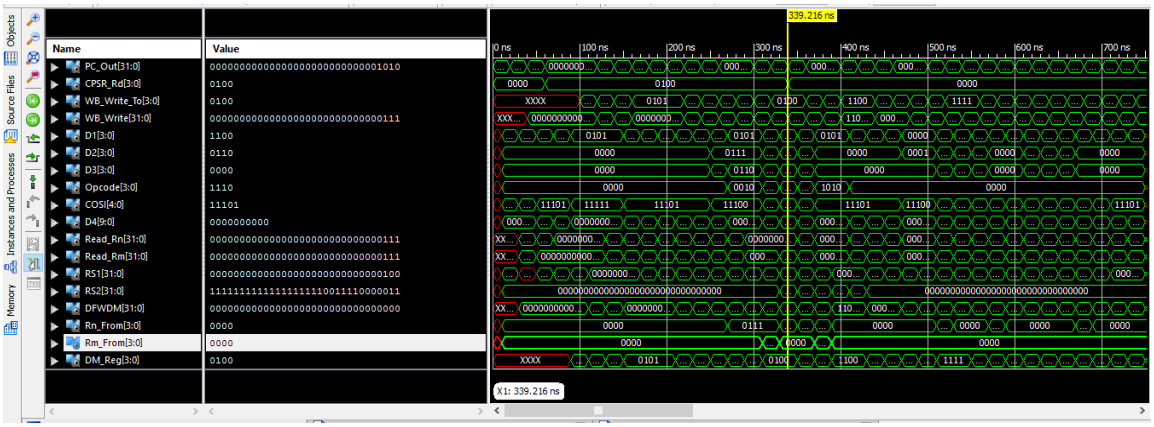
[2]

Figure 2- without Stack space handling of interrupts Figure 3- with Stack space handling of interrupts

As can be seen from figure 3 i1, i2 so on are instructions. Let the interrupt occur when instruction i4 is being fetched in a processor in figure 1, since the pipeline is flushed the handler has to return to that instruction(i4) after interrupt has been serviced. Whereas in processor in figure 3 has to return to the instruction i8; which address has been loaded in link register when interrupt occurred. Since instruction i4 is already fetched and moved to its stack space it will be moved to

the next stage i.e. decode stage once interrupt is serviced. Similarly data in decode stack space and Reg. Issue stack space are popped into Reg. Issue and Execute stage respectively.

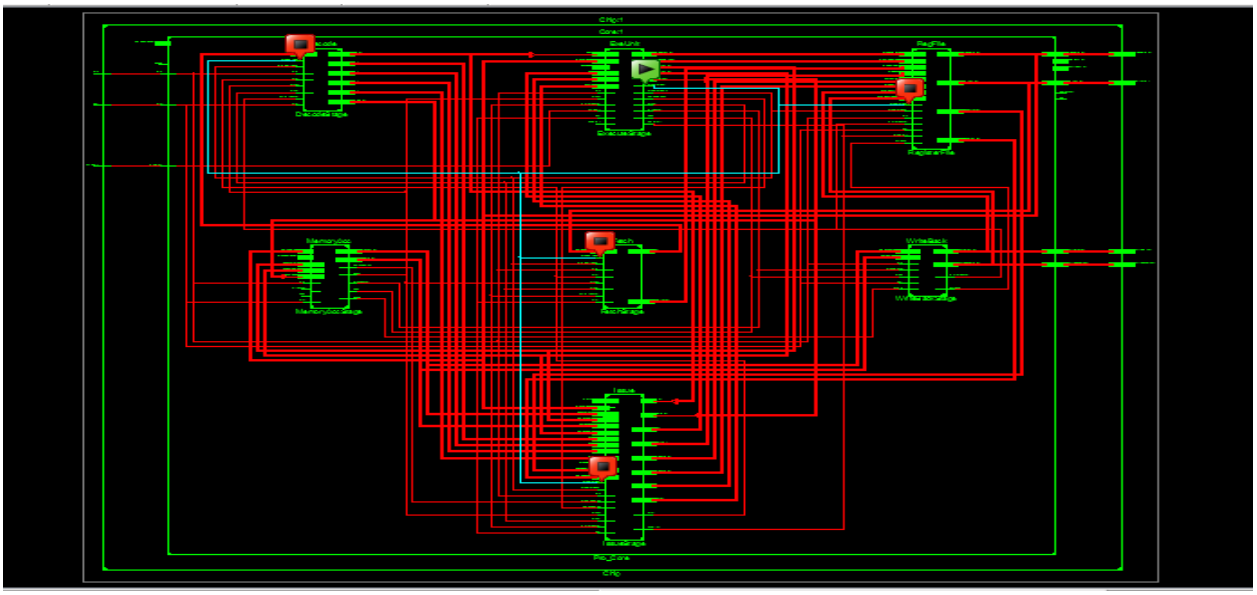
V. Results




```
HDL Synthesis Report

Macro Statistics
# RAMs                                              : 2
 256x32-bit single-port RAM                         : 1
 256x32-bit single-port Read Only RAM               : 1
# Multipliers                                        : 4
 32x32-bit multiplier                               : 4
# Adders/Subtractors                                 : 8
 32-bit adder                                       : 8
# Registers                                          : 111
 1-bit register                                     : 66
 10-bit register                                    : 1
 2-bit register                                     : 1
 32-bit register                                    : 26
 4-bit register                                     : 14
 5-bit register                                     : 2
 8-bit register                                     : 1
# Latches                                            : 107
 1-bit latch                                        : 107
# Comparators                                        : 6
 4-bit comparator equal                             : 6
# Multiplexers                                       : 1007
 1-bit 2-to-1 multiplexer                           : 812
 1-bit 6-to-1 multiplexer                           : 7
 2-bit 2-to-1 multiplexer                           : 27
 32-bit 15-to-1 multiplexer                         : 1
 32-bit 16-to-1 multiplexer                         : 2
 32-bit 2-to-1 multiplexer                          : 102
 32-bit 3-to-1 multiplexer                          : 2
 32-bit 6-to-1 multiplexer                          : 1
 4-bit 12-to-1 multiplexer                          : 1
 4-bit 2-to-1 multiplexer                           : 51
 4-bit 6-to-1 multiplexer                           : 1
# Logic shifters                                     : 2
 32-bit shifter logical left                        : 1
 32-bit shifter logical right                       : 1
```

For the results I've used a non-nested interrupt which interrupts processor at random times. Below is the fully synthesized processor.



VI. Conclusion

This paper presents the fully synthesizable processor which takes less clock cycles when interrupt occurs by not flushing the pipeline but instead saving the intermediate data on to special stack. As the number of stages increases this technique allows us to save multiple clock cycle which leads to increase in system performance. The techniques discussed can also be extended to the concept of branches with return types in conjunction with branch predictors in superscalar processors where in a single clock cycle multiple instructions are loaded and on branches and interrupts many- many clock cycles can be saved increasing the overall performance of the system.

REFERENCES

[1] James E. Smith and Andrew R. Pleszkun: "**Implementing Precise Interrupts in Pipelined Processor**" IEEE Transaction on Computers Vol..37 No. 5 May 1998.

[2] Andrew N.Sloss Dominic Symes and Chris Wright: "**ARM System Developer's Guide Designing and optimizing system software**". 2004 by Elsevier Inc.

[3] Thomas Scholz and Michael Schgfers: "**An Improved Dynamic Register Array Concept for High-Performance RISC Processors**". Dept. of Design of Integrated Circuits Technical University of Braunschweig, Germany.

[4] Wenjiang Li; Song Zhang; Xiong Jiang; Yaohui Zhang: "**A 5-stage pipelined embedded processor with optimized handling exception**". Computer Science and Network Technology (ICCSNT), 2011 International Conference on. Year: 2011, Volume: 4 Pages: 2773 - 2777, DOI: 10.1109/ICCSNT.2011.6182539

[5] Torng, H.C.; Day, M.: "**Interrupt handling for out-of-order execution processors**". Computers, IEEE Transactions on. Year: 1993, Volume: 42, Issue: 1 Pages: 122 - 127, DOI: 10.1109/12.192223.

[6] Aamer Jaleel and Jacob, B.: "**In-line Interrupt Handling and Lockup Free TLBs**". Computers, IEEE Transactions on Year: 2006, Volume: 55, Issue: 5 Pages: 559 - 574, DOI: 10.1109/TC.2006.77.

[7] J. Robert Jump and Sudhir R. Ahuja: "**Effective Pipelining of Digital Systems**".

[8] Steve Furber: "**System On Chip Architecture**".

[9] Rakesh M.R: "**RISC Processor Design in VLSI Technology Using the Pipeline Technique**".

[10] M. Schifers, U. Golze, E. Cochlovius: "**Verilog HDL Models of a Large RISC Processor**".In Proceedings of the 4th EUROCHIP Workshop, Toledo, 1993, pp. 242-246.

[11] Ozer, E.; Sathaye, S.W.; Menezes, K.N.; Banerjia, S.; Jennings, M.D.; Conte, T.M.: "**A fast interrupt handling scheme for VLIW processors**". Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on