International Journal of Applied Engineering Research
ISSN 0973-4562 Volume 10, Number 10 (2015) pp. 24883-24892

http://www.ripublication.com

Hazards Analysis In A Processor Using Bluespec System Verilog

[1]Siva Kumar Nagulapati and [2]Dr. Jayanta Biswas

[1]M.Tech. VLSI Design, SASTRA University, Thanjavur, Tamil Nadu, India
[2]Department of ICT, SASTRA University, Thanjavur, Tamil Nadu, India
[1]nagulapatisivakumarreddy@gmail.com, [2]jayantab2002@gmail.com

Abstract

This paper implements a prototype (sequential) processor and a simple pipeline processor with Bluespec System Verilog (BSV). Hazards are analysed in the processors, designed with atomic rules and methods, using Bluespec System Verilog Hardware Description Language. The pipeline processor operation methodologies and performance is compared with a prototype sequential processor in terms of hazards, processor cycles consumed for each instruction. The functionality of the designed processors is verified with GTK cycle accurate simulator. Altera design flow is adopted to verify the static timing which provides a flexible platform for fast exploration of micro-architectures for multithreaded and multicore CPUs. The concepts of concurrency, scheduling, parallelism and pipeline hazards optimization are used for better performance, flexibility and productivity.

Keywords—Bluespec, BSV, pipeline, mnemonic, opcode, hurdles, sequential, instruction, hazards

1. Introduction

Instruction pipelines, also called as RISC pipelines, are used in designing central processing units (CPUs) to allow overlapping execution of multiple instructions with the same circuitry. These uses RISC design philosophy. Pipeline technique speeds up the execution by fetching the next instruction while other instructions are being decoded and executed. The circuitry is partitioned into multiple stages with each stage processing an instruction at a time and total throughput increases by a factor equal to number of pipeline stages.

Fig. 1 shows the pipeline concept of executing three instructions – ADD, SUB and AND in a three stage pipeline containing fetch, decode and execute blocks. Add

instruction is fetched in cycle 1. In cycle 2, subtraction instruction is fetched and addition instruction is decoded. Likewise, in cycle 3, immediate logical AND instruction is fetched, subtraction instruction is decoded and addition instruction is executed. Hazards associated with the design should be monitored throughout the execution time for optimization and performance enhancement of the processor. RISC processors have reduced number of instruction classes which provide simple operations and the instructions can be executed in a single cycle.

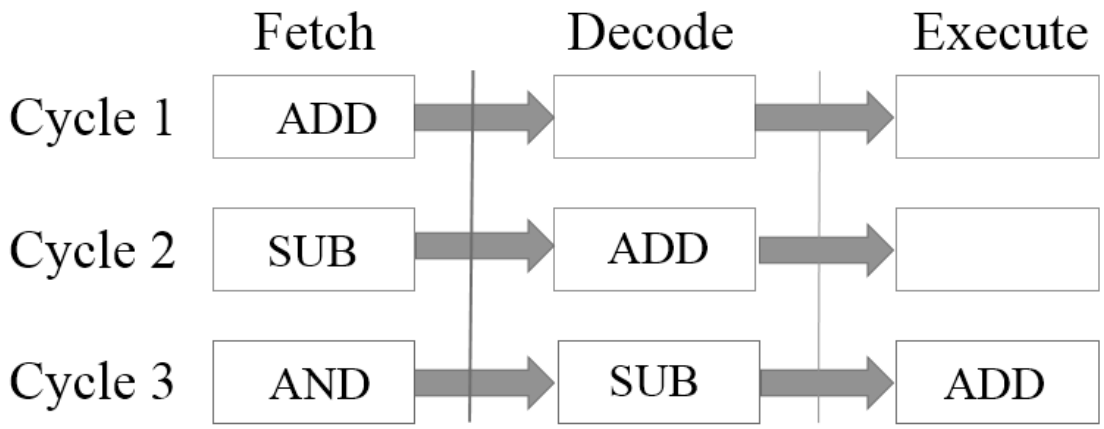

Fig. 1: Pipeline Instruction Flow

1.1 Paper Organization

Section 2 provides a brief introduction to Bluespec tool and its HDL features. Section 3 presents the concepts on pipeline technique and hazards associated with the prototype and pipeline processors. In Section 4, results are viewed concluded in Section 5.

2. Bluespec System Verilog

The Hardware Description Language (HDL) for Bluespec tool is BSV. It is a strongly typed hardware synthesis language which uses the Term Rewriting System (TRS) to compute a series of atomic actions. BSV is an object-oriented hardware description language. BSV borrowed its ideas from two sources - System Verilog and Haskell for its structural abstractions. Concepts of modules and module hierarchy, separation of interfaces from modules, syntax for user-defined types, syntax for blocks and loops are chosen from System Verilog. For more advanced types, parameterization and static elaboration, Haskell is used, which is a pure functional programming language and has powerful features than those in languages like C#, C++ and Java.

BSV's behavioral model is popular among formal specification languages for complex concurrent systems as the motivations behind this model are extraction of parallelism and correctness. Every program or code has its own package and package name as its file name with.*bsv* extension. A module inside the package represents the

design. A program can contain multiple packages or multiple modules in a single package. It is not advisable to include multiple modules in a single package because when the concept of complex processing is accessed, it may contain more than expected functional units or modules and rises the confusion of module descriptions at the interfaces acting between modules and its sub-modules.A module is treated as fundamental unit of the design and is used for synthesizable architectural modeling, exploration and validation at a system level.

The Fig. 2 describes the construction of a BSV design. The two stages of compilation, type checking and code generation, are done by a single compile command. Compiler completes type checking and static elaboration and compiles the design into a TRS. TRS is converted into Verilog or C program. Intermediate.*bi/.bo/.ba/.h* files are not directly viewed by the user. BSV compiler maps the design into parallel clocked synchronous hardware and includes dynamic scheduler which allows multiple rules to fire in a single clock cycle. Bluespec Development Workstation (BDW) is a graphical design environment for creating, building, analysing and simulating BSV designs. The source files are named with extension *'.bsv'* and the project file is named after *Top* file name with extension '.*bspec*' while loading in the workstation.

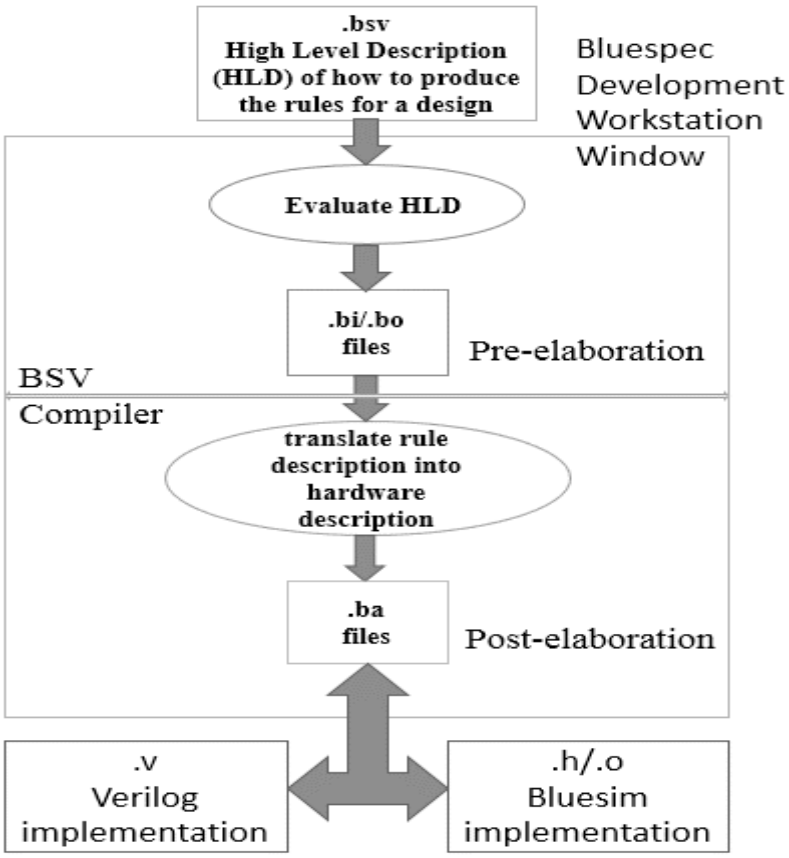

Fig. 2: BSV Compilation Flow

3. PROCESSOR DESIGNING

The above Fig. 3 shows a five stage pipeline architecture. Instructions are fetched in IF stage from the instruction memory. These instructions are decoded in the ID stage. The opcode operation is executed in the EX stage. Data memory is used by load/store instruction in MA stage. The results obtained from either execute stage or memory access stage are written back to the destination address by the WB stage. There are two forward paths from the execution stage, one to the memory access stage and the other directly to the write-back stage. The shaded regions are the pipeline registers. These registers cross the forward going paths and stores the previous stage data, making available for next stage.

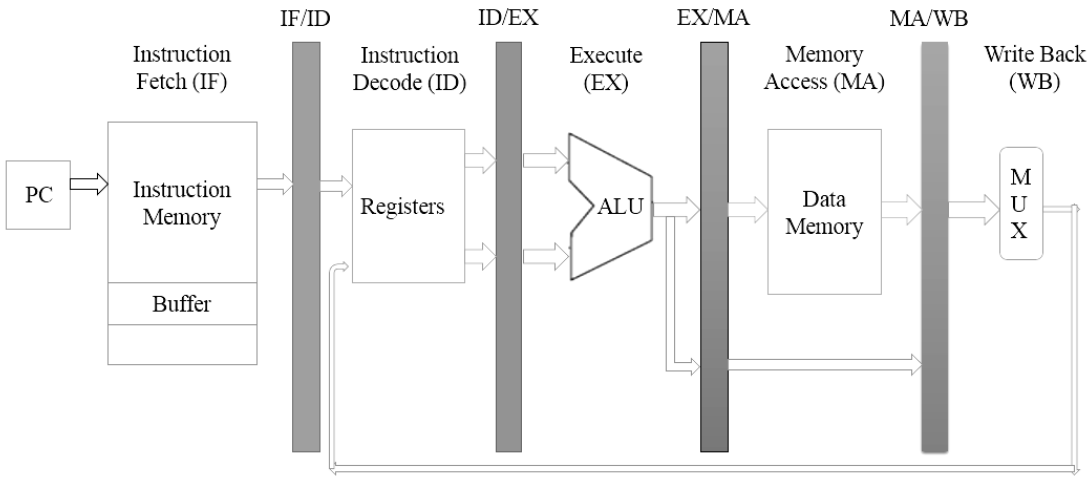

Fig. 3: Pipeline structure

3.1 Instruction Format

A 24-bit instruction is used as shown in Fig. 4. Bits from 0 to 4 are dedicated for mnemonics of instructions. Bit 5 is used as write acknowledgement for destination register from bit 6 to 10, whether to write to a register or to be used in other data transfer and control operations. Bits 11 and 17 are similar acknowledgement bits for sources 1 and 2 respectively. Bit 23 is used as acknowledgement signal for accessing the memory block. Based on the opcode at the beginning of the format, the destination and source bits play different roles for better performance and optimization. The acknowledgement bits are used to load the data from the memory or to use direct data. If both source1 acknowledgement and source2 acknowledgement bits are zero, address is loaded from source1 bits and source2 bits. If either of source's acknowledgement is 1, the other source is loaded with direct data from the instruction.

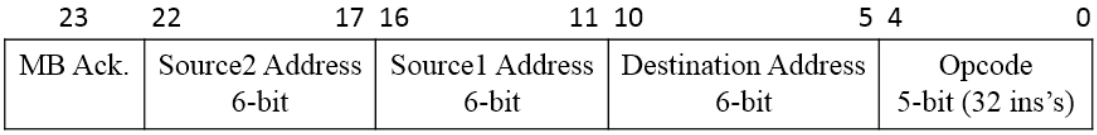

Fig. 4: Instruction format

Table 1 shows the set of mnemonics and their respective opcodes used in this work. These include arithmetic, logical, load-store, branching and machine control instructions. All these are commonly utilized instructions in a RISC processor architecture.

Table 1: Mnemonics & Opcodes

*	Mnemonic	Opcode	*	Mnemonic	Opcode	*	Mnemonic	Opcode
01	ADD,ADI	00001	12	INC	01100	23	RSTR	10111
02	ADDC,ADIC	00010	13	DEC	01101	24	RLDR	11000
03	SUB,SBI	00011	14	CMP	01110	25	WSTR	11001
04	SUBC, SBIC	00100	15	SET	01111	26	WLDR	11010
05	MUL, MUI	00101	16	CLR	10000	27	CJMP	11011
06	DIV	00110	17	ROR	10001	28	UJMP	11100
07	AND, ANI	00111	18	ROL	10010	29	CALL	11101
08	ORR, ORI	01000	19	SHR	10011	30	RET	11110
09	EOR, ERI	01001	20	SHL	10100	31	HLT	11111
10	NOT	01010	21	EXC	10101	00	NOP	00000
11	BIC,BCI	01011	22	MOV, MVI	10110			

3.2 Hazards Analysis in Prototype Processor Design

A sequential prototype design resembling pipeline stages is designed without using pipeline registers. Data transfer instructions also use the memory effectively. This design follows the same procedure as a processor, fetching instruction from its memory, decoding the instruction and loading required data, executing the operation regarding the instruction and finally storing the result.

Sequentially, four instructions - ins1, ins2, ins3 and ins4, are loaded and executed in the design. These do not use memory access stage. First instruction is loaded and passed to decode stage. While the ins1 is being decoded, ins2 is fetched. Later, ins1 operation is executed based upon the opcode and in the same cycle ins2 is decoded and ins3 is fetched. In the fourth cycle, ins1 result is recorded in this stage, ins2 operation is executed, ins3 is decoded and final instruction, ins4 is fetched. During 4^{th} execution cycle, as ins4 is fetched, the design considers ins4 destination address as ins1's destination address and ins1's result is stored in the ins4's destination address. This leaves behind the ins1's destination memory as empty.

Successive execution of instructions are delayed if a control transfer instruction like jump or call instruction is executing in present cycle and consumes more cycles, introducing *delay slots*. When control hazards are active, it chokes the pipeline and causes the instructions next to branch or jump instruction to stall one clock cycle. While shifting the execution control to branch target, the present data like program counter value, flags status etc. are saved in a stack pointer and are retrieved back to their respective positions after executing branch target data. Executing proceeds with next instruction in the flow.

Structural hazards arise if functional units are not pipelined properly and if there is not enough resources. These hazards effect the design primarily due to

asynchronized timing requirements between the stages of architecture. All the instructions doesn't execute in same number of processing cycles. Some instructions consume one or two more processing cycles when compared to simple instructions. These issues add up weight on data dependencies, resulting in data loss, missing the instruction execution order, etc.

Constraints like these are suppressed by ordering the instructions in fetching stage such that the next instruction is fetched after a certain delay. Thus the structure becomes a single cycle design which executes one instruction at a time and fetches the next instruction after fully executing the previous instruction. These dynamic hazards can also be avoided by introducing pipeline registers and reservation stations.

Data hazards are based on instruction execution order where later instructions' inputs are using outputs of present executing instruction. In such cases, pipeline stalls. It is like a gap in the pipeline at that cycle time. Data forwarding method can be used to reduce the stalls. This can also be avoided by reordering the instructions order in such a way that independent instructions are placed between dependent instructions.

3.3 Pipeline Processor Design

The pipeline registers are initialized as shown in Fig. 5. These registers are not of same size as each stage produces more data bits than its previous stage. These registers are denoted by both the neighbouring stages. Register between IF and ID is noted as IF/ID or IFDE, likewise for the others too. Backward passing paths do not cross these registers.

```
//---------- 4 pipeline registers---
   Reg#(Bit#(25))  ifde  <- mkReg(0);
   Reg#(Bit#(121)) idex  <- mkReg(0);
   Reg#(Bit#(153)) exma  <- mkReg(0);
   Reg#(Bit#(249)) mawb  <- mkReg(0);
```

Fig. 5: Pipeline registers initialization

After the instruction fetch, pipeline register *ifde* is loaded with instruction and is masked. In the decode stage, the instruction is decoded and loads the data from the memory. Some instructions act upon direct data where there is no need to load data from the memory. This is specified using acknowledgement signals, whether an instruction loads data bits from the memory or acts on direct data.

Scheduling plays a vital role in architecture modelling. Signal passing lines, control signals, data forwarding paths are to be synchronized to one another. This synchronization precisely reduces the slack in setup and hold timings for the module. Proper predicates are defined to the rule bodies to fire them at necessary cycle. Bluespec runs with atomic clock actions and the rules are fired with conditions and

count values if necessary. A process counter runs in every cycle and is used to fire the required rules.

In the schedule analysis window, scheduling information is viewed which include warnings, conflicts between rules, rule relations and method calls for a module. Conflicts between two selected rules is studied and optimized using *rule relations* option. *Rule Order* tab in the schedule analysis provides the details about the rules in the module and lists the available rules and methods in the left pane and the information regarding selected rule or method in the right pane.

3.4 Data Forwarding

Table 2 shows the analysis of dependent instructions, with and without data forwarding concept in the processor. The instruction consumes more processing cycles till the availability of data in the destination address of previous instruction. This degrades the performance of the processor and increases the timing constraints.

Table 2: Analysis of stalls

	No. of processor cycles	No. of stalls
Without Data Forwarding	24	11
With Data Forwarding	15	2

The saving of ALU result prior to write-back stage helps in data forwarding data to the successive dependent instruction. At the ALU operating stage, the result of an executed instruction is stored in a register tagged with destination address. The next dependent instruction in the queue is stalled for one processor cycle after the IF stage of the instruction. During ID stage, the instruction is decoded and the previously stored result tagged with the destination address is loaded in the register. This procedure of data forwarding avoids stalls for true data dependencies between the instructions and increases the performance of the processor. Data forwarding reduces the delay incurred due to stalls to a minimum count, by making data available to the next instruction at its decode stage.

4. RESULTS

The initial sequential design is simulated for basic arithmetic and logical instructions. Due to structural hazards, it is hard to execute load-store instructions in between instruction flow because of dependencies and timing mismatch between successive stages of execution. The data forwarding concept cannot be implemented in the design unless the hazards associated with the design are resolved. Fig. 6 shows the simulation result of a sequential processor. From the simulation wave, we can conclude that this design is suffering from structural hazards which stalls the pipeline. The present executing instruction's result is stored in the future fetched instruction's destination address and this procedure follows through subsequent fetched instructions. *'mdat32'* is the memory address to store the final result after write-back

stage for the first instruction which executes addition operation. During fourth cycle of storing the result, fourth instruction is fetched and its output memory address is considered as present first instruction's write-back address. Processor stores in the fourth instruction's write-back address in *'mdat35'*.

The waveform in Fig. 7 shows the pipeline design results. Pipeline registers reduce data and structural hazards allowing timing synchronization between different stages of pipeline. Hence only one instruction resides in a pipeline stage for a current processor cycle. Resources conflicts are supressed by defining resources each stage individually. Even though number of processor cycles increased, hazards are optimized and timing synchronization issues are resolved for better performance and productivity. The data shown in the figure is in hexa-decimal format for our convenience of study. Data forwarding principles optimized the number of delay slots during the execution of dependent instructions. This analysis is shown in table 2, where without data forwarding idea, 24 processor cycles are consumed.

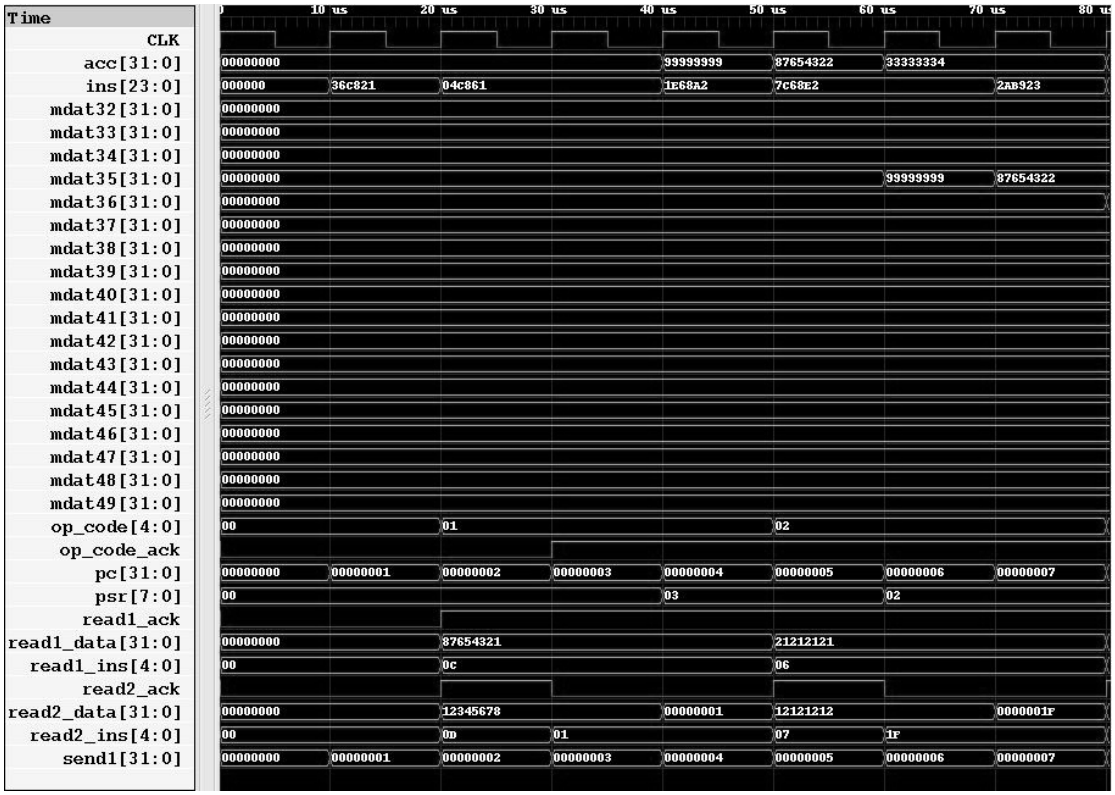

Fig. 6: Prototype (Sequential) Processor Waveform

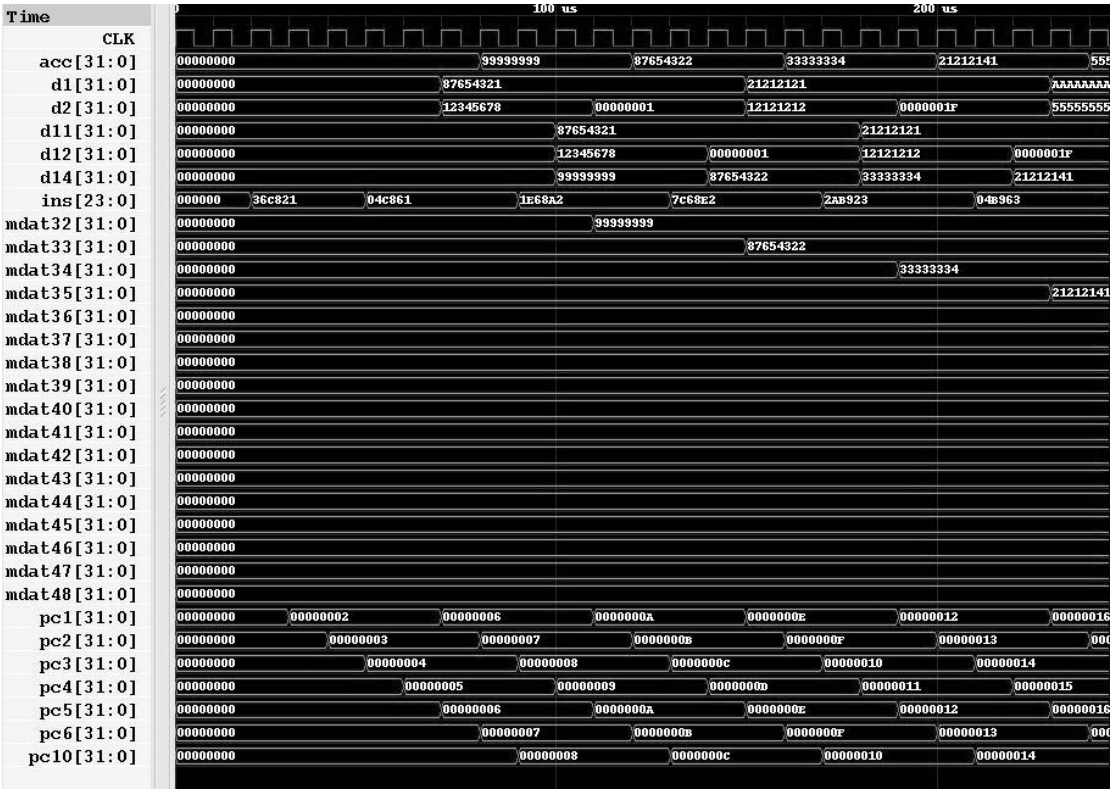

Fig. 7: Pipeline Processor Waveform

The Verilog version of the design is compiled in Altera Quartus II version 9.1 web edition, to meet the timing requirements. The design is loaded in a new project directory and compiled for multi-corner timing analysis. Slack value for both setup and hold times are verified. In the Altera Time-Quest Timing Analyzer, timing netlist is created and clock values are updated. *Fmax* for the prototype processor is obtained at 18 MHz and for pipeline processor the value of *Fmax* is 168 MHz. This variation is due to resources overlapping in the sequential design along with data hurdles and timing synchronization issues.

5. CONCLUSION

A Prototype Sequential processor is designed without pipeline registers initialization. The verilog version of the design program is obtained from Bluespec and static timing analysis is performed using Altera Quartus II tool. Detailed analysis is performed on the Pipeline design with optimization protocols for maximum throughput. Scheduling analysis is throughly worked out for timing synchronization between the pipeline stages. Timing analysis is performed using the same Altera tool and is determained to work at 168 MHz for Stratix-III platform.

References

[1] Yuan-Chu Yu and Yuan-Tse Yu, (2013), Design of a High Efficiency Reconfigurable Pipeline Processor on Next Generation Portable Device, IEEE, in the proceedings of Digital Signal Processing and Signal Processing Education Meeting (DSP/SPE), pp: 42 - 47.

[2] Rishiyur S. Nikhil, (2007), Composable Guarded Atomic Actions: A Bridging Model for SoC Design, IEEE, in the proceedings of 7th International Conference on Application of Concurrency to System Design (ACSD), pp: 23 - 28.

[3] J. Robert Heath and Sreenivas Durbha, (2001), Methodology For Synthesis, Testing, And Verification of Pipeline Architecture Pocessors From Behavioral Level - Only HDL Code And A Case Study Example, IEEE, in the proceedings of SoutheastCon, pp: 143 - 149.

[4] Y. Li and W. Chu, (1996), Aizup - A Pipelined Processor Design and Implementation on XILINX FPGA Chip, IEEE, in the proceedings of IEEE Symposium on FPGAs for Custome Computing Machines, pp: 98 - 106.

[5] John L. Hennessy and David A. Patterson, Computer Architecture-A Quantitative Approach, 4th ed., vol. 2. Elsevier, 2007.

[6] William Stallings, Computer Organization And Architecture - Designing For Performance, 8th ed., Pearson, 2010.

[7] Monica S. Lam and Robert P. Wilson, (1992), Limits of Control Flow on Parallelism, IEEE, in the proceedings of 19th Annual International Symposium on Computer Architecture, pp: 46 - 57.

[8] Daniel L. Rosenband and Arvind, (2004), Modular Scheduling of Guarded Atomic Actions, IEEE, in the proceedings of 41st Design Automation Conference, pp: 55 - 60.

[9] Arvind, Rishiyur S. Nikhil, Daniel L. Rosenband and Nirav Dave, (2004), High-level Synthesis: An Essential Ingredient for Designing Complex ASICs, IEEE, ACM International Conference on Computer Aided Design (ICCAD), pp: 775 - 782.

[10] Daniel L. Rosenband, (2004), The Ephemeral History Register: Flexible Scheduling for Rule-Based Designs, IEEE, in the proceedings of 2nd ACM International Conference on Formal Methods and Models for Co-Design (MEMOCODE), pp: 189 - 198.

[11] Arvind and Xiaowei Shen, (1999), Using Term Rewriting Systems To Design And Verify Processors, IEEE, Micro, pp: 36 - 46.

[12] Nirav Dave, (2004), Designing a Reorder Buffer in Bluespec, IEEE, in the proceedings of 2nd ACM International Conference on Formal Methods and Models for Co-Design (MEMOCODE), pp: 93 - 102.

[13] Tomoyuki Nakabayashi, Takahiro Sasaki, Kazuhiko Ohno and Toshio Kondo, (2011), Design and Evaluation of Variable Stages Pipeline Processor Chip, IEEE, in the proceedings of 16th Asia and South Pacific Design Automation Conference (ASP-DAC), pp: 95 - 96.