The Journal of Supercomputing (2020) 76:7394–7415
https://doi.org/10.1007/s11227-020-03174-5

Design and implementation of an elastic processor with hyperthreading technology and virtualization for elastic server models

Parth Bir[1] · **Shylaja Vinaykumar Karatangi**[1] · **Amrita Rai**[1]

Published online: 24 January 2020

Abstract

The server models functioning in the industry are required to be more elastic in nature. They are constantly scaling-up and scaling-down on required computation power depending on different conditions. These elastic cloud platforms use accelerators like DSP's, TPU's, GPU's, FPGA's, and multi-core processors to provide exponential computing power and outsource their services. This process is not only costly and non-efficient but is also responsible for damaging the server's hardware architecture. Furthermore, these additions degrade the level of threading and symmetric parallel processing capability of the architecture. Intel uses hyperthreading technology (HTT) to split the workload between hardware and operating system to avoid additions, but that too is only possible up until a certain limit. This paper presents design methodology and implementation of an elastic-natured 32-bit RISC-pipelined processor inspired from Intel Xeon and MIPS to function as a standard integrated platform for server models. It implements concepts of hyperthreading technology (HTT) and virtualization on hardware basis. It will allow to derive multiple outputs from units on hardware basis to enhance security and performance without compromising compatibility. The designed elastic core uses a probabilistic node-based closed-queuing network model for server analysis and implementation. Hence, elastic behavior from individual core microarchitecture to server model architecture enables a generic automated scaling self-aware optimization architecture.

Keywords Elastic · DSP · TPU · GPU · FPGA · Multi-core processors · RISC · Intel Xeon · MIPS · HTT · Virtualization · CQN

✉ Parth Bir
parthbir.17@gmail.com

Shylaja Vinaykumar Karatangi
shilajachalageri@gmail.com

Amrita Rai
amrita.tu@gmail.com

[1] GL Bajaj Institute of Technology and Management, Greater Noida, India

1 Introduction

The cloud computing has enabled to muster sufficient computational for the time being. Nevertheless, the demand for computational power is still rising. The nature and type of tasks vary, and scaling accordingly is a difficult task. Today, the cloud platforms provide resources for every task: storage, simulation, outsourcing application-specific services, etc. However, at the hardware end, calculations of diverse versatile nature require specialized hardware. In addition, complications caused by leakage current due to size, power consumption, and introduction of fuzzy logic, etc., are rendering cloud systems present hardware architecture obsolete.

In order to introduce a base for diverse functionality, cloud platforms are adding digital signal processors (DSPs), field-programmable gate array (FPGA), and graphics processing unit (GPU), etc. In addition, although use of standard multicore processors increases the net performance, irregular addition leads to degradation in performance of overall server architecture. These specialized additions pose a problem as they inhibit to form multiple threads within the system. Therefore, this lack of threads or interconnectivity between the systems on platform is bound to cause failure on the operating system end too. At present, most of the cores use ARM 64-bit architecture due to its data-handling capacity, net CPI, and other characteristics. However, massively parallel cores like DSP, FPGA, and GPU, etc., i.e., cores with the ability to handle instructions equal to or more than 64, have compatibility issues when working simultaneously with standard cores.

The designed processor core is of elastic nature [1]. It is inspired from Intel's Xeon series [2, 3] and MIPS-32 [4, 5]. The term elastic here refers to its ability to scale its hardware resources as per requirement using concepts of virtual memory [6, 7] and hyperthreading technology (HTT) [8–10]. This hardware-level implementation of HTT and virtual memory enables the designed processor core to function with core of any architecture in a smooth manner. This provides salient features of parallel execution, shorter instruction set of single instruction multiple data (SIMD) nature in a RISC format. In addition, these traits enable threading together the required hardware resources as per requirement, ease in scaling process, and changes if any are required.

Any microprocessor is made up of two major units, i.e., control unit and data path unit. The design flow methodology starting from a brief introduction to HTT afterward covers the design phases from FSM's, behavioral-level design, structural unit description, simulation results, logic utilization, followed by description of the mathematical model then ending with FPGA implementation and conclusion.

The hyperthreading technology (HTT) [11, 12] forms multiple threads within a core unit. The operating system (OS) acts as an interface to form these multiple threads between hardware and software. These threads execute simultaneously to hide the latencies caused by other factors such as data access. Intel invented the HTT to ease the performance load on hardware in server models. It works by making a software copy of the core. The OS copies hardware core to generate the software core. Next, OS acts as the control unit for both hardware and software core. Therefore, HTT allows optimum utilization of resources. Intel inculcated

HTT in Xeon series [13, 14] to design processor cores specifically for server models. The Xeon series provide a series of scalable processors. The series defines different processors based on different socket, functionality, and cloud requirements. However, Intel only provides HTT for processors based on its Nehalem and Pentium microarchitectures.

The HTT tends to improves performance by providing as many independent functions in the pipeline as possible. The designed core possesses hybrid-natured pipelining with virtual memory. This allows the core to manipulate instructions independently, and execute multiple threads simultaneously. In addition, it only uses hardware resources to achieve HTT functionality and is therefore considerably faster. The required units are mapped on dedicated memory. Next, they are executed in a simultaneous manner. These virtual hardware resources are executed in parallel using direct memory access (DMA), specialized instructions and dedicated memory.

The designed ASIC core can support multiple sockets. It has diverse functionality for a variety of platforms like quantum computing, edge computing, artificial intelligence, etc. It can simultaneously communicate with accelerators like GPU, DSP, and FPGA. In addition, it poses no compatibility issues similar to Intel Xeon series and other server processors. Therefore, the designed ASIC core is more suited in terms of performance, compatibility, and future needs for server architectures [15].

2 Designed elastic processor

The elastic computing nature of the designed processor provides flexibility to the otherwise rigid nature of hardware server architecture. The multiple threads formed by HTT tend to act as control lines, which enable elastic properties for the system without actually disrupting the prior model. In addition, it enables server system to support functions of diverse nature. The elastic nature of the server architecture will allow scope for improvement and solve the current problems faced by the industry. Furthermore, it also simplifies the scaling process and addition of accelerators. In addition, scaling the present hardware without changing the entire setup is more suitable and prevents various losses caused due to current system practices.

2.1 Instruction set architecture (ISA) and control models

The designed processor functions on a hybrid ISA and control models which enable it to treat server architecture in terms of standardized nodes. This nodal characteristic is of utmost importance as described below and under mathematical model for server implementation.

2.1.1 Dynamic reconfigurable ISA (DRISA)

The proposed hybrid ISA [16] is designed with respect to hyperthreading, to keep maximum number of independent instructions in the pipeline-using concept of virtualization [17]. In order to function as a massively parallel processor, the ISA inculcate features from RISC, CISC, and VLIW architecture.

The term elastic computing [18–20] refers to the ability of the system to scale its resources according to the requirement. Present cloud systems like Amazon Ec2 and Microsoft Azure tend to manually allot a larger or smaller computing platform or resources to the user using operating system whenever there is a need for scaling. In addition, if the workload increases up to a certain limit, the system fails to function properly and will indefinitely crash. This process is not only inefficient but also causes loss or contamination of data in most cases. Hence current process requires relies entirely on inefficient software practices. It leads to less efficient results in terms of performance, data security, cost, workload, power, and man-power management.

Each individual instruction is made of multiple smaller instructions. Next, either each instruction can be processed at once, or it can be broken into independent individual parts. These smaller independent sub instructions can then be executed in different pipeline stages. However, each smaller or subinstruction is of same length, and similarly, each longer or parent instruction is of the same length. This kind of symmetrical nature for instructions, allows configuring the server architecture as per ones' requirement. Hence, designed ISA functions as a dynamic reconfigurable ISA.

The dynamic single instruction multiple data (SIMD) nature helps support the process for virtualization. In addition, being of the same length, different parts can function on different configurations. This also eases the process of scaling computational power as per requirement. The designed ISA is as follows:

Category	Instruction: symbol	Comments
Arithmetic	add: add	Requires min. three operands
	subtract: sub	Requires min. three operands
	add immediate: addi	Used to add constants
Data transfer	load word: lw	Word travels from memory to register
	Store word: sw	Word travels from register to memory
	Load byte: lb	Byte addresses/data from memory to register
	Store byte: sb	Byte addresses/data from register to memory
	Load upper immediate: lui	Loads required constant in upper bits
Conditional branch	Branch on equal: beq	Test for being equal, PC-relative branch test
	Branch on not equal: bne	Not-equal test, PC-relative test
	Set on less than: slt	For beq and bne
	Set less than immediate: slti	Compares constants
Unconditional jump	Jump: j	Jump to target address
	Jump register: jr	For switching to procedure return
	Jump and link: jal	For procedure call

2.1.2 Finite state machine (FSM)

For each defined state of the FSM, the server architecture requires different control signals for different units. In addition, considering HTT and DRISA, the FSM

model should permit possibility for extension. In order to allow an expanding cloud natured architecture, the FSM is designed in accordance with the EralngC model. This gives designed FSM a uniform model, which can be continuously monitored and reconfigured if needed.

Prime parameters for a server like sockets, sockets per server, cores, cores per server, and number of servers, etc., are manipulated easily via the designed FSM. The designed FSM supports virtualization and HTT. It can serve as an integral building block in the server architecture, without restraint. In addition, it provides flexibility depending on the need (Figs. 1, 2, 3, and 4).

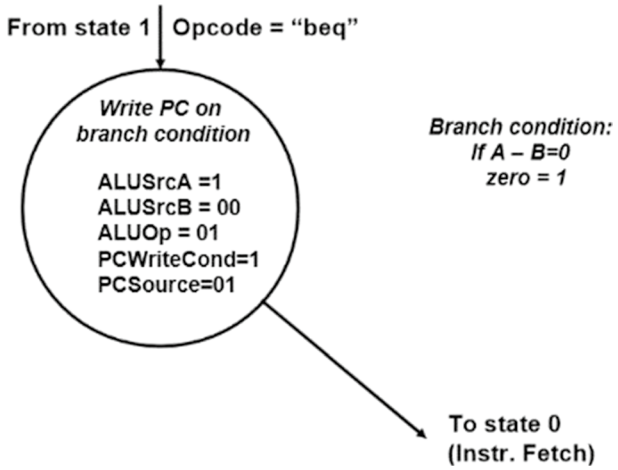


Fig. 1 FSM branch instruction

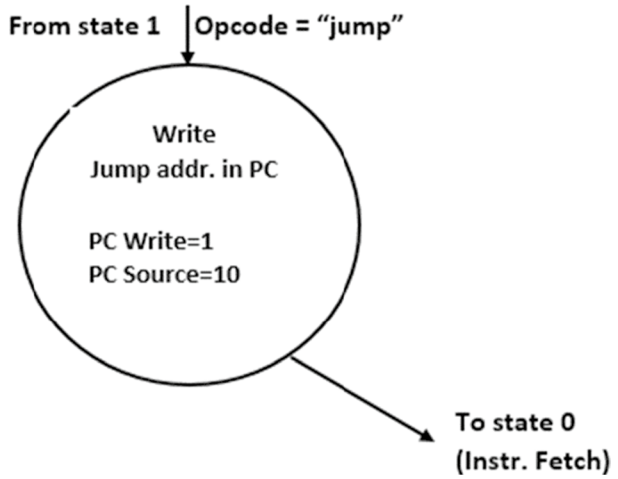


Fig. 2 FSM jump instruction

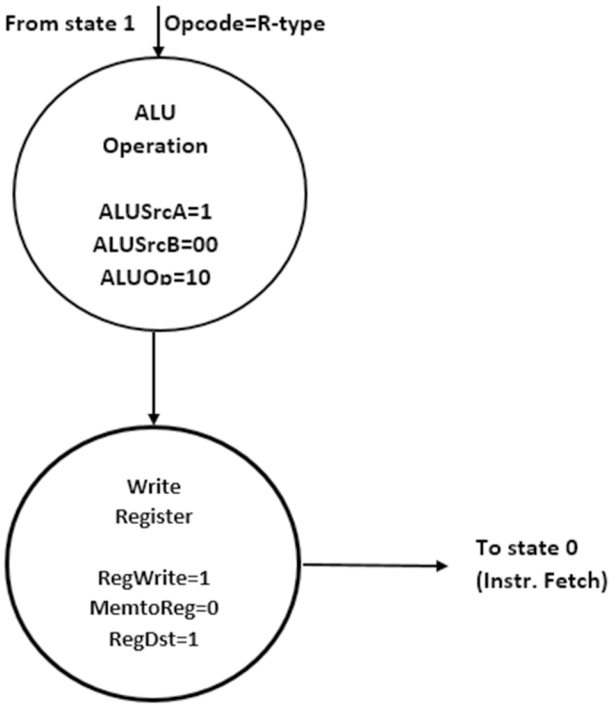


Fig. 3 FSM R-instruction

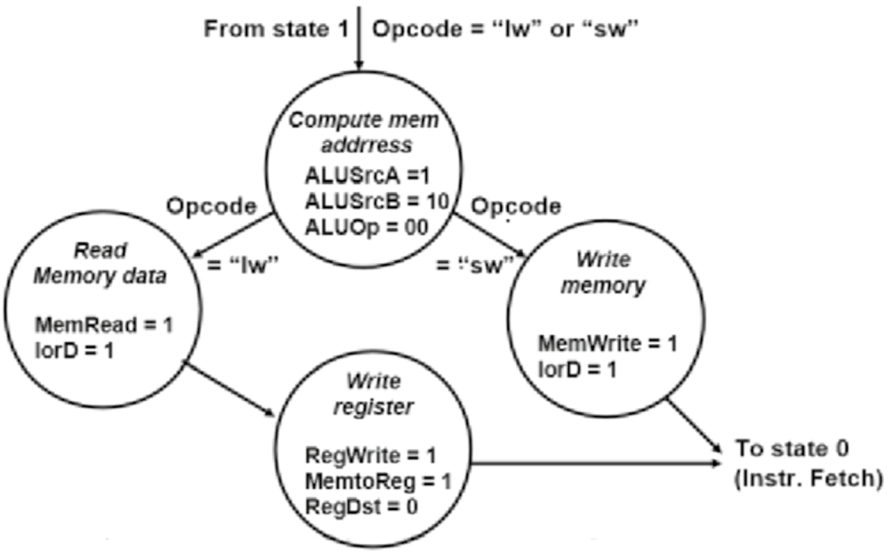


Fig. 4 FSM memory access

2.1.3 Data path unit

The designed data path is a massive parallel threaded multi-core platform. It is designed by the FSM state graphs for each instruction and additional control signals. In addition, the virtual memory allows the data path to function as a network. This ability to control each unit as a network in a server architecture enables improved control over the entire server (Figs. 5).

2.1.4 Control unit

Initially, finite state machine graphs for each system state are implemented. This process helps generate required signals needed to control the systems current state. The same process is repeated for each state. These generated signals are termed as control signals. The unit responsible for generating control signals with respect to state of the system is termed as the control unit.

The designed core uses a virtual memory unit and data path unit to form a reversible logic process. Next, this process helps extract and learn data characteristics from the system. Next, system is trained using the extracted features. This circuit-level deep learning paradigm enables the designed core to function in new environments without any constraints.

In this manner, the control unit updates by itself. For a server architecture, this refers to hardware automation. The said process allows the hardware to scale itself as per the requirement. In addition, over time, the system gets more precise and faster (Fig. 6).

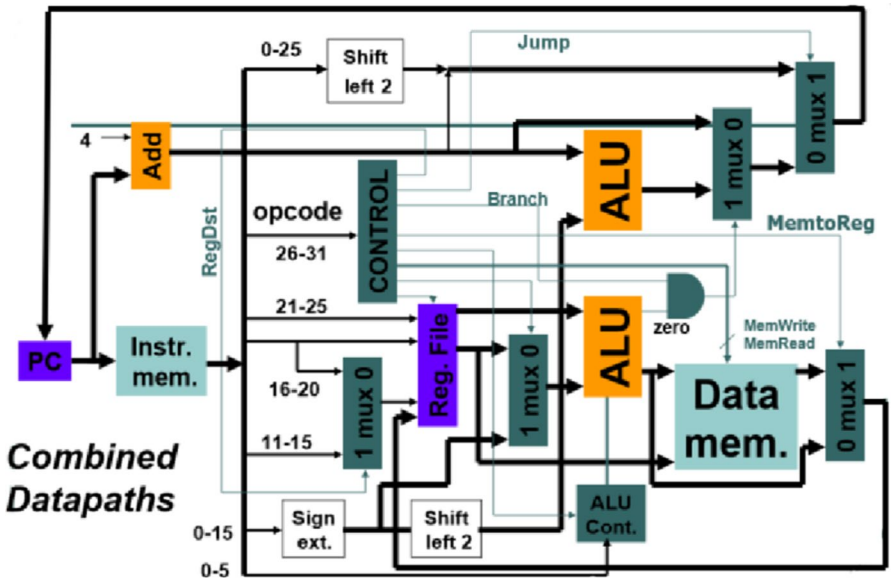


Fig. 5 Integrated data path

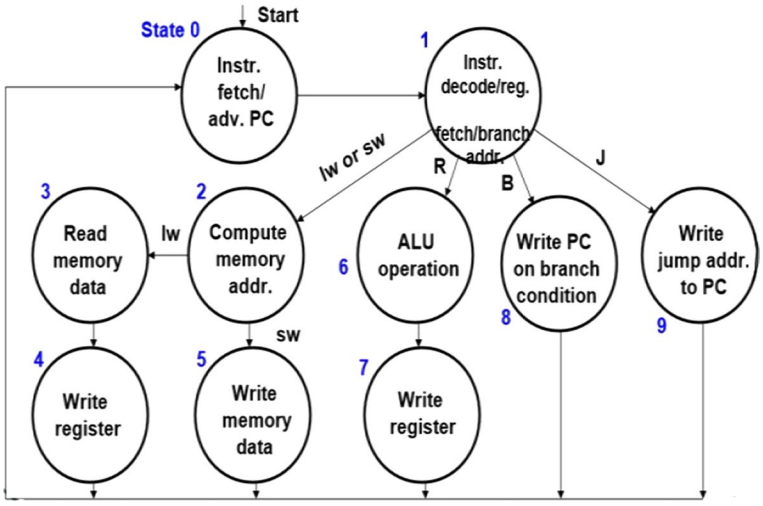


Fig. 6 FSM control unit

2.2 Integrated processor module

Server processors integrate a large number of cores to enhance performance. Each core has the same architecture. This generic nature of cores reduces the functional abilities of the processor. However, with diverse multiple units, the performance of the processor increases exponentially. The integrated processor module contains multiple processing units. It is to be noted that the implementation of the integrated processor module is based on the principles of HTT and virtualization. These principles enabled the hardware implementation of the elastic core both individually and at the server level. The primary factor to enable the adaptive nature of the elastic core is done by basing its functionality on probability as per ErlangC model. In addition to probabilistic nature, including CQN theory addresses the task of routing the data flow at a microarchitecture level. While directing the data flow at the microarchitecture level, probability is used as a range confinement constraint. This allows successful implementation of the HTT at hardware level. In addition, virtualization process acts as a fail-safe feature. In a case of workload over burst, virtualization allows expansion of present hardware to meet requirements. Furthermore, implementing this behavioral model in different units of the elastic processor, allows a homogenous and uniform architectural layout. As for the core architecture, the designed processor contains following major units:

1. ALU
2. Data memory
3. Register file
4. Instruction memory
5. Program counter

2.2.1 Arithmetic logic unit (ALU)

The arithmetic logic unit (ALU) is used for general calculations and instruction manipulation. Under normal conditions, only the original ALU remains functional. If better performance were required, the entire ALU unit is mapped on virtual memory unit. Next, both the original and virtual ALU are executed simultaneously. Under these high-performance conditions, the virtualized ALU unit serves as the master unit and the rest as slave units.

In addition, after workload analysis of the server, the meta-data sets are formed due by data memory and the hypervisor interface traffic intensity. Based on traffic intensity analysis of the meta-data sets, additional and specialized ALU units are framed. In order for workload optimization, a set of ALU units are mapped using virtualization process: linking of the different ALU units to act as an integrated MAC and initial processing unit for server traffic. This method of initial optimization in the designed elastic core grants elasticity to the virtualization process itself. It is vital to automate this link between HTT and virtualization to enable a self-aware scaling process in the server model architecture. Hence, designed ALU unit is of utmost importance in workload assignment and optimization of the servers' architectural data flow model (Fig. 7).

2.2.2 Data memory

The data memory unit serves as the major resource to support the hardware implementation of HTT, virtualization, and threading of individual units to form hybrid functional specific units. It is the data memory over which the virtualized units are mapped. The limit of dynamic memory elements per server is established as per

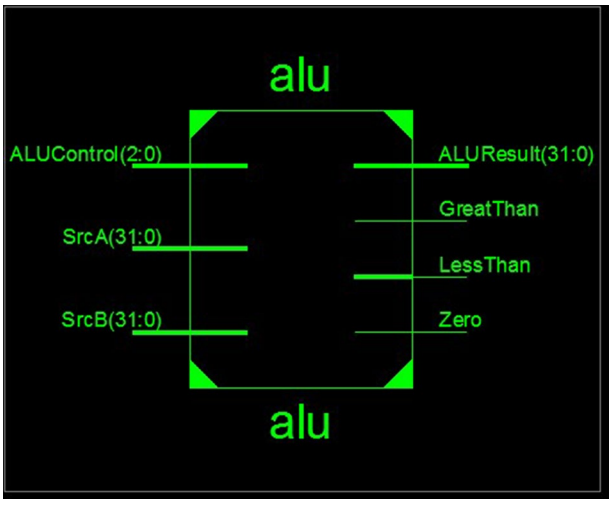


Fig. 7 ALU

Eqs. (5) and (6). The limit for RAM per core is given by Eq. (6), while Eq. (5) gives the number of VM's per server.

In addition, the interconnected interface between the hypervisor and the data memory is one of the factors for individual core RAM requirement. However, this interface is also exploited to introduce meta-data sets depicting the type and size of data processing instances over server. These meta-data sets are responsible for prediction of the unit to be mapped over data memory. Similar to branch prediction, pre-unit mapping over data memory using HTT and virtualization helps in reduction in the net number of operations executed. Over a period of time, due to this process of pre-unit mapping, server performance drastically increases. In addition to operation reduction, it also reduces workload, enhances per unit output, and optimizes server performance. Also, current server models or outdated models will show a significant boost in performance due to optimization of the workload. Hence, this simple process can act as a cost saving and updating factor for outdated server models. But, more importantly, in order to support this interface and implement, this learning model, HTT, and virtualization are vital.

This interface-based traffic intensity calculation, coupled with HTT and virtualization process, helps in formation of a generic automated self-aware, hardware-based optimization engine. Hence, instead of requiring human input, the server system can monitor, assess, and correct itself using the principles of the designed elastic processor (Fig. 8).

2.2.3 Register file

The register file forms a pack of registers. Instead of having individual register access points, it provides simultaneous access to all the registers. This allows uninterrupted access to every register. Hence, this helps in reducing the delay caused by data access. The register file is divided into two halves. The first half is for use by core only, whereas the second half is used for holding some external memory access and to support virtualization. In order to support virtualization, it helps increase the frequency of pre-unit mapping process. Register file is faster in operations compared to data memory. Therefore, it is used for dynamic monitoring of the hypervisor interface during earlier stage of core addition to the server architecture interface (Fig. 9).

2.2.4 Program counter (PC)

The Program Counter (PC) is responsible for pointing to the next instruction to be executed. A virtual PC is also created via help of register file. Together, both program counters help in better memory and resource management. In addition, either both can help work on different segments of an instruction (As indicated in DRISA) or independently on different instructions. This process allows faster execution of multiple independent instructions present in the pipeline. In addition, PC is responsible for supporting the CQN theory built between master cores and slave cores to increase net coverage and decrease unreachability, respectively, in the server architecture (Fig. 10).

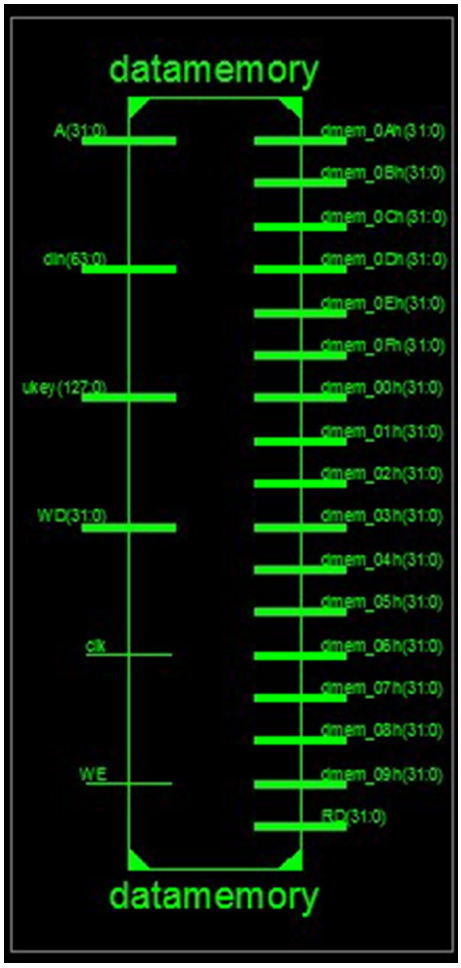


Fig. 8 Data memory

2.3 Mathematical model for server-level implementation

Due to the variety of hardware, namely CPU, GPU, TPU, etc., used in server construction, it is a complex task to match the hardware functionality, performance, and other characteristic factors as per the specific requirements of the server. Hence, the designed systems mathematical model allows treating each individual entity as a node. Henceforth, if each individual entity is treated as a node differing only on the basis of input and output, respectively, server system design process is simplified. As for the present server models, they can be further optimized for maximum output as per vendor and user requirements. In order to adequately determine the number and configuration of nodes, workload requirements are translated to capacity measures in terms of CPU, memory, and number of servers required for paramount performance. The provided mathematical model covers the mathematical proof required for implementing,

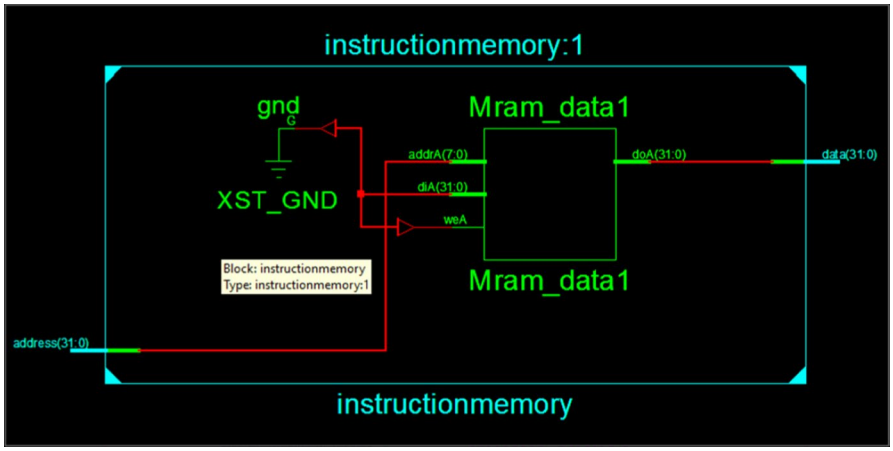
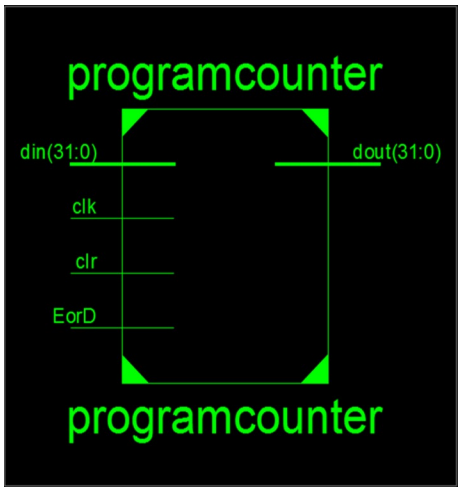


Fig. 9 Instruction memory

Fig. 10 Program counter



maintaining the server and as well as measures its current state and performance. Unlike the current server models in use, a hybrid version of the ErlangC model [21] based on ErlangC equation as shown in Eq. (1) is used for implementation. Equation (1) predicts the probability of a request delayed based on traffic intensity (A) and number of cores (N). Calculation of P_{w} enables the adaptive nature of the designed elastic core as is made evident from further analysis.

$$P_{\mathrm{W}} = \frac{\frac{A^N N}{N!N-A}}{\left(\sum_{i=0}^{N-1} \frac{A^i}{i!} + \frac{A^N N}{N!N-A}\right)} \tag{1}$$

where P_w probability of a request delayed, A traffic intensity, N number of cores.

For implementing or scaling the server model [22], the most important factors that come into play are the core and memory requirements.

2.3.1 Core requirements

As per ErlangC model server analysis, it is imperative to control server characteristics at the core level in order to make a meaningful impact at the higher order. The designed elastic core takes into account the following results to enable a core-level adaptive and self-aware environment for server implementation.

$$\text{Cores} = \text{VMs/HT/CO} \tag{2}$$

Equation (2) is responsible for determining the number of processing cores required for server implementation. It aims at bringing the net core requirement to as minimum as possible by splitting the assigned workload. The workload approximation on the basis of ErlangC model is split on the basis of number of virtual machines (VM), hyperthreading (HT) coefficient, and core oversubscription (CO) coefficient. The factors taken into account are of prime importance as they will be responsible for workload sharing during operation.

$$\text{Sockets} = \text{Cores/Cores\_per\_socket} \tag{3}$$

Next, Eq. (3) describes the net number of core sockets or CPU socket requirement which is proportional to the number of cores. Depending on the number of cores and cores per socket, socket requirement is calculated. Also, cores per socket is a variable factor due to hardware in habitation.

$$\text{Servers} = \text{Sockets/Sockets\_per\_server} \tag{4}$$

where HT hyperthreading coefficient, CO CPU oversubscription coefficient.

Lastly, Eq. (4) depicts the number of servers required. Factors affecting server implementation include number of sockets and sockets per server. Also, similar to sockets servers are indirectly dependent upon the number of cores.

These relationships establish the fact that sockets are servers that are dependent on the number of cores present. Henceforth, the designed system directly targets the core implementation and its influencing factors.

2.3.2 Memory requirements

Determining memory requirements for server operations requires checking effect of every influencing factor. Since only virtual components are present, only factors affecting VM's are taken into account.

$$\text{VMs\_per\_server} = \text{VMs}/\text{Servers} \tag{5}$$

$$\text{RAM} = \text{RAM\_per\_VM} *(\text{VMs\_per\_server}/\text{MO} + \text{OS\_RAM}) \tag{6}$$

where VM virtual machine, MO RAM oversubscription coefficient.

Therefore, initially Eq. (5) helps in calculating VM's per server. Number of VM's per server is simply depicted by the ratio of the number of VM's to the number of servers. Next, Eq. (6) helps in determining the net RAM requirement. Factors influencing RAM requirement include RAM requirement per VM, VM's per server, RAM oversubscription coefficient (MO) and OS_RAM, i.e., the RAM required for hypervisor and the operating system (OS). In addition, RAM requirement for hypervisor varies depending on server model. It depends on whether hypervisor is using system memory or not. It is recommended not to use system memory as hypervisors RAM as it creates another variable factor.

2.3.3 Core-specific nodal model

The mathematical model of the processing unit and its respective memory establish that server performance is solemnly based on ground processing nodes. Therefore, it is established that control at higher order, i.e., at server level and mere core addition is not efficient. This model of computation of current servers is expensive and bound to fail in the long term.

Hence, the designed elastic core addresses the server architecture at the basic or core level. The elastic nature of the designed processor is based on a probabilistic model. Hence, the designed core is adaptable to the changing needs of the server architecture.

$$\text{ASA} = \frac{P_{\text{w}} * \text{Average Processing Time}}{\text{Number of Cores} - \text{TI}} \tag{7}$$

$$\text{Immediate Response} = \left(1 - P_{\text{w}}\right) * 100 \tag{8}$$

$$\text{Occupancy} = \frac{\text{TI(Erlangs)}}{\text{Number of Cores}} * 100 \tag{9}$$

For current server model core-level analysis, Eq. (7) determines the average speed of answer (ASA). Next, ASA is calculated on the basis of average processing time, number of cores, traffic intensity (TI) in terms of number of requests or erlangs and probability that a request delays (P_{w}). In addition, P_{w} helps determine other governing factors namely the immediate response time in Eq. (8). Next, Eq. (9) determines the occupancy in terms of TI, erlangs, and current number of working cores. Occupancy gives us a measure of server performance at core level.

$$\text{Number of Required} = \frac{\text{Current Working Cores}}{1 - (\text{Shrinkage}/100)} \tag{10}$$

where ASA average speed of answer, TI traffic intensity in terms of requests or erlangs, *Shrinkage* server performance error rate based on past performance.

Finally, on the basis of results from Eqs. (7), (8), and (9), respectively, we can determine the number of cores required. In addition, Eq. (10) introduces the shrinkage factor to calculate the response. Shrinkage factor is a measure of server failure rate as per past records. This method of adding a probabilistic factor by the designed elastic core gives the sense of self-adaptability to the server model.

2.4 Input output methodology

The designed elastic cores generic and adaptable nature allows its addition to any server architecture model. Similar to addition of DSP's, GPU's, FPGA's, and TPU's, etc., in any architecture, designed elastic core is added as an accelerator unit. For base-level implementation, depending on specifications, an accurate optimized architecture is devised using the mentioned mathematical model. Using the prescribed mathematical model, specifications from microarchitecture to server-level or superscalar architecture can be successfully defined in terms of computational capacity. Next, its adaptable nature is utilized to plug in the service-oriented architecture (SoAeu) of current server models. Furthermore, as an accelerator, it can act as both as assessment and acceleration unit to enhance and optimize server specification and performance based on acquired data. Using the current cloud-based distributed or parallel computational, existing output methodology can be used for driving required output.

3 Simulations and physical design

Since, the designed elastic cores' functionality is unique. The test benches and simulation procedures are designed in a manner so as to allow individual core functioning. Next, the simulation procedures for large-scale testing are done by multiple core implementations using parallel and distributed computational models. For integrated testing stage, FPGA implementation is limited to core functioning and characteristic functional verification. Future work for the elastic core includes server-level implementation and verification in hybrid environments consisting of both commercial server processors and the designed elastic processor.

3.1 Logic synthesis

The logic synthesis results are of prime importance as they determine the subsequent results for physical design and fabrication costs. For the designed core, logic synthesis is done using Xilinx ISE Design Suite 14.7 inbuilt logic simulator. Simulations indicated the number of slice registers, look up-tables (LUT), LUT_FF pairs, bonded IOB's, and BUF along BUFG CTRL's used (Figs. 11 and 12).

pb_Processor Project Status			
Project File:	new_rp_32.xise	**Parser Errors:**	No Errors
Module Name:	pb_Processor	**Implementation State:**	Synthesized
Target Device:	xc7a100t-3csg324	**•Errors:**	No Errors
Product Version:	ISE 14.1	**•Warnings:**	58 Warnings (24 new)
Design Goal:	Balanced	**•Routing Results:**	
Design Strategy:	Xilinx Default (unlocked)	**•Timing Constraints:**	
Environment:	System Settings	**•Final Timing Score:**	

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	2254	126800	1%
Number of Slice LUTs	1958	63400	3%
Number of fully used LUT-FF pairs	75	4137	1%
Number of bonded IOBs	259	210	123%
Number of BUFG/BUFGCTRLs	1	32	3%

Fig. 11 Logic utilization

```
================================================================================
Timing constraint: Default OFFSET OUT AFTER for Clock 'clk'
  Total number of paths / destination ports: 64 / 64
--------------------------------------------------------------------------------
Offset:              0.650ns (Levels of Logic = 1)
  Source:            RF/regfile_1_31 (FF)
  Destination:       Reg1<31> (PAD)
  Source Clock:      clk rising

  Data Path: RF/regfile_1_31 to Reg1<31>
                               Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDE:C->Q             3   0.361   0.289  RF/regfile_1_31 (RF/regfile_1_31)
     OBUF:I->O                0.000          Reg1_31_OBUF (Reg1<31>)
    ----------------------------------------
    Total                     0.650ns (0.361ns logic, 0.289ns route)
                                      (55.5% logic, 44.5% route)

================================================================================
```

Fig. 12 Net delay

3.2 Clock results

The clock cycle results and cycles per instruction (CPI) for the designed processor core were determined using the ISIM simulator in Xilinx ISE Design Suite 14.1. Figure 13 indicates the reset state for the designed processor core. As indicated in the figure, during the reset state, every unit is assigned a particular value. These values define a particular state for the system. In addition, the clock timing helps establish the initiation time and other aspects like CPI, clock frequency, etc. Moreover, it establishes the abilities of the system in case of failure and particular fail-safe features in place for recovery of data. Furthermore, Fig. 14 shows clock results and other characteristics of the processor when tested using a designed test bench and under normal conditions once all the start-up programs have been successfully executed. Conclusively, both Figs. 13 and 14 show that the designed FSM is working properly and no unit is in undefined state. Most importantly, read_reg_1 and read_reg_2 along with alu_ina and alu_inb signals in Fig. 14 verify the node-level adaptive characteristics. Since, the core is under adaptive mode, workload sharing or splitting among different units allows optimum utilization of the resources. Furthermore, if the required resources are not present, additional resources, namely alu_inb, are mapped using the HTT and virtualization for maximum output.

The designed integrated elastic processor (shown in Fig. 15) satisfies the performance benchmark and hurdles presented. In addition, its physical design parameters including gate count, area used, clock gating, etc., are much easier to fabricate. Compared to three-dimensional multilayers which are the root cause of lack of efficiency

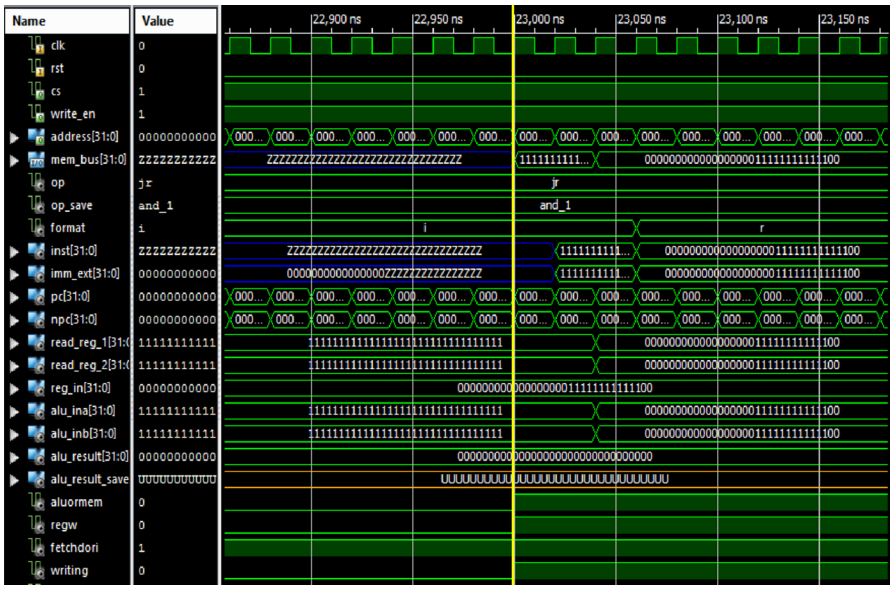


Fig. 13 Clock enabled and registers reset

in physical design, the designed cores' manufacture process and its reconfigurable nature allows complex operation even in a simpler architecture.

Due to this ease in manufacturing process, the designed core can be taken as a reference and built upon using other accelerator technologies to devise hybrid cores which are application specific in nature. The designed hybrid core can encompass design complexities from each accelerator, bringing down the cost factor by reducing the net number of cores used and addition of other specialized accelerators.

3.3 FPGA implementation and functional verification

The designed processor core is implemented on Nexys-4 DDR Artix-7 FPGA. Initially, the reset stage (Shown in Fig. 16) is implemented. Next, testing of designed processor using a simple calculation is performed using the ALU (Shown in Fig. 17). The FPGA design of the processor successfully verifies the said functions of the designed processor core.

From simulations under different conditions with ALU, register read/write and memory read/write as the constraint factors, following results were obtained:

- Clock cycle time: 6 ns
- Branch instruction: 5 ns
- Jump instruction: 2 ns
- R-type instruction: 6 ns
- Delay: 0.650 ns

In addition, considering the industry standards, comparing the designed processor core to a standard MIPS processor and a RISC-V ISA processor, with respect to use in servers, following results are obtained:

Property	MIPS processor core	Designed processor core
Power consumption	Medium	Very low
Instruction set	RISC (less complex)	Hybrid (more complex)
IPC (instructions per cycle)	1 MIPS	> 2 MIPS

Similarly, when comparing designed processor cores ISA to that of RISC-V and x86, we obtained the following results:

Property	RISC-V ISA	x86 ISA	Designed processor core ISA
Type	RISC	CISC	RISC but large in length
Nature	SIMD	MIMD	MIMD
Ideal use	Parallel execution	Serial execution	Parallel execution
Applications	IOT, parallel computing	Personal computers	Servers, IOT, supercomputing systems, etc.

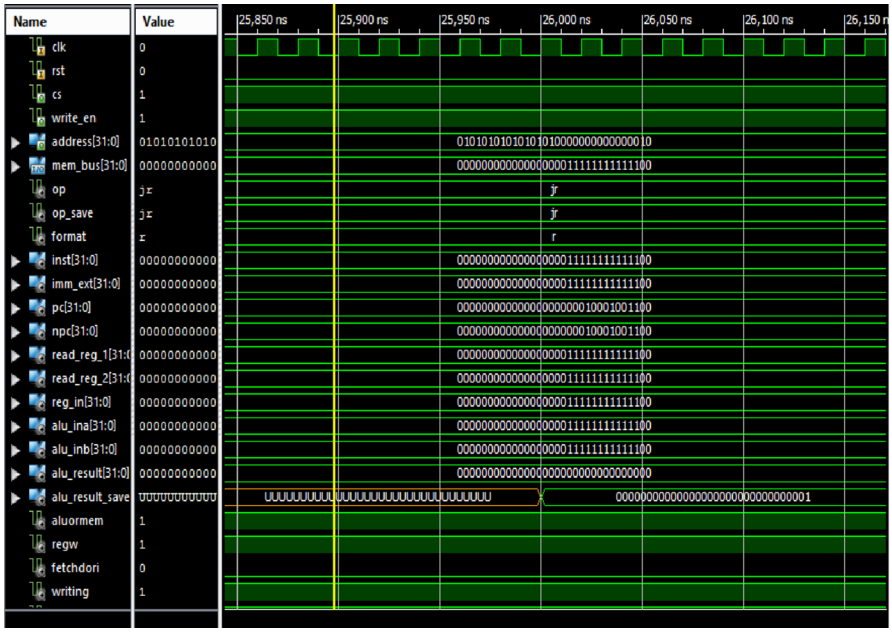


Fig. 14 Set states

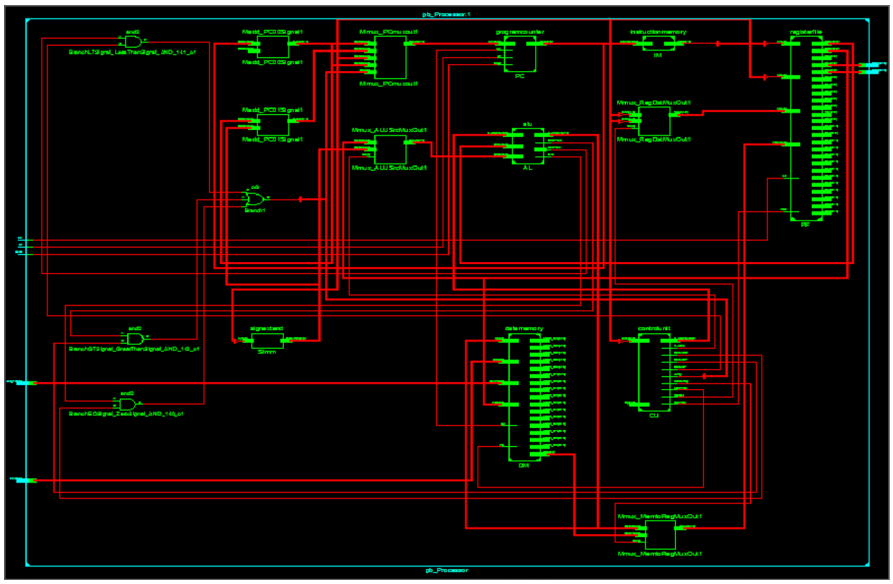


Fig. 15 RTL schematic for integrated elastic processor

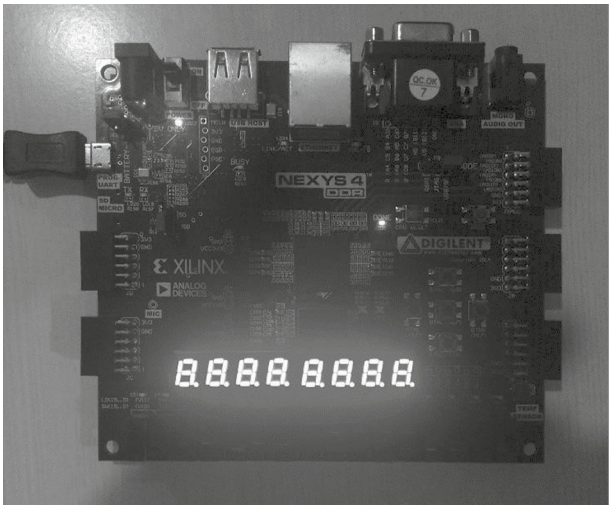


Fig. 16 Reset stage

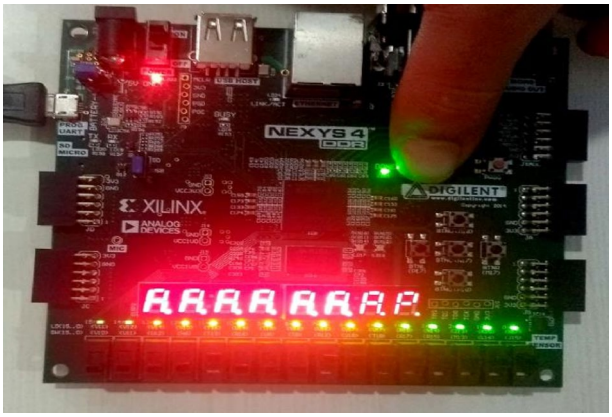


Fig. 17 Use of ALU and memory units

Execution speed	> 1 MIPS	> 1 MIPS	2 MIPS
Bits	32, 64, 128	16, 32, 64	16, 32,64
Type	Load-store	Register-memory	Load-store
General purpose	16, 32	16, 32, 64	32
Floating point	32	16	32
Security	Not secure (due to open source nature)	Very secure	Very secure (due to state of the art nature)
Compatibility	Not compatible with other ISA's	Not compatible with other ISA's	Compatible with any new environment due to its ability of virtual memory and HTT

As seen by the results above, the designed processor core not only satisfies the set benchmark, but also is an improvement over it, with minimal cost factor, increasing its efficiency manifold. It serves as an excellent addition due to its elastic nature and characteristics of virtual memory and hardware implementation of HTT.

4 Conclusion

The designed processor is most suited for server models considering the fact that server models used in the industry are based on machine and deep learning. The software-level architecture is continuously evolving but, when used with old computing resources, decelerates the performance. The proposed model is an improvement at the microarchitecture and superscalar architecture level that allows exponential increase in overall computing power. In addition, it optimizes all the influencing factors and resources. The server processor cores require flexibility and scalable performance. The implementation of hyperthreading technology (HTT) and virtualization has solved current problems of the industry. The generic automated scaling self-aware optimization nature of designed elastic processor can exponentially improve server model electronics.

For future work, the designed processor core requires to go through power analysis, physical design analysis, and server-level implementation using multiple FPGA's and other suitable environments. In addition, its performance under connection with a dedicated data acquisition system needs to be tested for large-scale server model implementation.

References

1. Wu H, Liu F, Lee RB (2016) Cloud server benchmark suite for evaluating new hardware architectures. IEEE Comput Archit Lett 16(1):14–17
2. Rekachinsky AI, Chulkevich RA, Kostenetskiy PS (2018) Modeling parallel processing of databases on the central processor Intel Xeon Phi KNL. In: 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). IEEE, pp 1605–1610
3. Gepner P, Gamayunov V, Litke W, Sauge L, Mazauric C (2014) Evaluation of Intel Xeon E5-2600v2 based cluster for technical computing workloads. In: 2014 International Conference on High Performance Computing and Simulation (HPCS). IEEE, pp 919–926
4. Omran SS, Abdul-abbas AK (2018) Design and implementation of 32-bits MIPS processor to perform QRD based on FPGA. In: 2018 International Conference on Engineering Technology and Their Applications (IICETA). IEEE, pp 36–41
5. MIPS Technologies, Inc. (2003) MIPS32™ architecture for programmers volume II: the MIPS32™ instruction set
6. Mo Mniruzzaman ABM, Nafi KW, Hossain SA, (2014) Virtual memory streaming technique for virtual machines (VMs) for rapid scaling and high performance in cloud environment. In: 2014 International Conference on Informatics, Electronics and Vision (ICIEV). IEEE, pp 1–6
7. Lee M, Park S, Song Y, Eom YI (2019) Introspection of virtual machine memory resource in the virtualized systems. In: 2019 IEEE International Conference on Big Data and Smart Computing (BigComp). IEEE, pp 1–4
8. Xu S, Gregg D (2015) Exploiting hyper-loop parallelism in vectorization to improve memory performance on CUDA GPGPU. In: 2015 IEEE Trustcom/BigDataSE/ISPA. IEEE, Vol. 3, pp 53–60

9. Gallin G, Tisserand A (2017) Hyper-threaded multiplier for HECC. In: 2017 51st Asilomar Conference on Signals, Systems, and Computers. IEEE, pp 447–451
10. Qun NH, Khalib ZI, Warip MN, Elobaid ME, Mostafijur R, Zahri NA, Saad P (2016) Hyper-threading technology: not a good choice for speeding up CPU-bound code. In: 2016 3rd International Conference on Electronic Design (ICED), 2016 Aug 11. IEEE, pp 578–581
11. Anastopoulos N, Koziris N (2008) Facilitating efficient synchronization of asymmetric threads on hyper-threaded processors. In: 2008 IEEE International Symposium on Parallel and Distributed Processing. IEEE, pp 1–8
12. Saini S, Jin H, Hood R, Barker D, Mehrotra P, Biswas R (2011) The impact of hyper-threading on processor resource utilization in production applications. In: 2011 18th International Conference on High Performance Computing. IEEE, pp 1–10
13. https://www.intel.in/content/www/in/en/products/processors/xeon.html
14. Moldovanova OV, Kurnosov MG, Mel'nikov A (2018) Energy efficiency and performance of autovectorized loops on Intel Xeon processors. In: 2018 3rd Russian-Pacific Conference on Computer Technology and Applications (RPC). IEEE, pp 1–6
15. LIBSVM A (2004) Library for support vector machines, Chih-Chung Chang and Chih-Jen Lin. Computer Science Department, National Taiwan University. http://www.csie.ntu.edu.tw/cjlin/libsvm
16. Nemirovsky D, Markovic N, Unsal O, Valero M, Cristal A (2015) Reimagining heterogeneous computing: a functional instruction set architecture (F-ISA) computing model. In: IEEE Micro
17. Oh C, Ro WW (2014) Hyper threading-aware virtual machine migration. In: 2014 International Conference on Electronics, Information and Communications (ICEIC). IEEE, pp 1–2
18. Zhu J, Li X, Ruiz R, Xu X, Zhang Y (2016) Scheduling stochastic multi-stage jobs on elastic computing services in hybrid clouds. In: 2016 IEEE International Conference on Web Services (ICWS). IEEE, pp 678–681
19. Prukkantragorn P, Tientanopajai K (2016) Price efficiency in high performance computing on amazon elastic compute cloud provider in compute optimize packages. In: 2016 International Computer Science and Engineering Conference (ICSEC). IEEE, pp 1–6
20. Shah NK (2014) Data warehouse systems in the environment of cloud computing—a comparative study of elastic cloud computing and organizational systems. In: 2014 International Conference on Computing for Sustainable Global Development (INDIACom). IEEE, pp 118–123
21. Chechina N, MacKenzie K, Thompson S, Trinder P, Boudeville O, Fördős V, Hoch C, Ghaffari A, Hernandez MM (2017) Evaluating scalable distributed Erlang for scalability and reliability. IEEE Trans Parallel Distrib Syst 28(8):2244–2257
22. "Amazon EC2," http://aws.amazon.com/ec2/