UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM MICROELETRÔNICA

THIAGO DE OLIVEIRA SILVA

Elastic Circuits in FPGA

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Microeletrônica.

Orientador: Prof. Dr. André Inácio Reis

Porto Alegre
2017

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Silva, Thiago de Oliveira
Elastic Circuits in FPGA / Thiago de Oliveira Silva. -- 2017.
65 f.
Orientador: André Inácio Reis.

Dissertação (Mestrado) -- Universidade Federal do Rio Grande do Sul, Instituto de Informática, Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR-RS, 2017.

1. Elastic Circuits . 2. Digital IC . 3. IC Design Methodology . 4. FPGA . 5. Asynchronous Circuits. I. Reis, Andre Inácio, orient.

ACKNOWLEDGEMENTS

This work is dedicated to all who have supported me during the graduation period, helping me get through the task of dedicating time to study and develop the experiments in parallel with a full time job.

My parents Jose and Sonia, who have always dedicated their attention and support, never letting me drop the ball with their encouraging words and advises. They are the reason of any success I may have in my lifetime, my role models.

My girlfriend Danielle, who has been by my side during this work development, always pushing me forward. I was able to complete this effort because of all her support and love.

My advisor, for the support on the things I did not know how to do and for helping me to develop the research skills needed

My friends and colleagues, who have patiently supported me by backing me up when needed.

All the professors, who have shared their knowledge with me and strengthened to my technical and academic background.

RESUMO

O avanço da microeletrônica nas últimas décadas trouxe maior densidade aos circuitos integrados, possibilitando a implementação de funções de alta complexidade em uma menor área de silício. Como efeito desta integração em larga escala, as latências dos fios passaram a representar uma maior fração do atraso de propagação de dados em um design, tornando a tarefa de “timing closure” mais desafiadora e demandando mais iterações entre etapas do design.

Por meio de uma revisão na teoria dos circuitos insensíveis a latência (Latency-Insensitive theory), este trabalho explora a metodologia de designs elásticos (Elastic Design methodology) em circuitos síncronos, com o objetivo de solucionar o impacto que a latência adicional dos fios insere no fluxo de design de circuitos integrados, sem demandar uma grande mudança de paradigma por parte dos designers.

A fim de exemplificar o processo de “elasticização”, foi implementada uma versão síncrona da arquitetura do microprocessador Neander que posteriormente foi convertida a um Circuito Elástico utilizando um protocolo insensível a latência nas transferências de dados entre os processos computacionais do design. Ambas as versões do Neander foram validadas em uma plataforma FPGA utilizando ferramentas e fluxo de design síncrono bem estabelecidos.

A comparação das características de timing e área entre os designs demonstra que a versão Elástica pode apresentar ganhos de performance para sistemas complexos ao custo de um aumento da área necessária.

Estes resultados mostram que a metodologia de designs elásticos é uma boa candidata para projetar circuitos integrados complexos sem demandar custosas iterações entre fases de design e reutilizando as já estabelecidas ferramentas de design síncrono, resultando em uma alternativa economicamente vantajosa para os designers.

Palavras-chave: Elastic Circuits. Digital IC. IC Design Methodology. FPGA. ASIC. Synchronous Circuits. Asynchronous Circuits.

ABSTRACT

The advance of microelectronics brought increased density to integrated circuits, allowing high complexity functions to be implemented in smaller silicon areas. As a side effect of this large-scale integration, the wire latencies became a higher fraction of a design’s data propagation latency, turning timing closure into a challenging task that often demand several iterations among design phases.

By reviewing the Latency-Insensitive theory, this work presents the exploration of the Elastic Design methodology in synchronous circuits, with the objective of solving the increased wire latency impact on integrated circuits design flow without requiring a big paradigm change for designers.

To exemplify the elasticization process, the educational Neander microprocessor architecture is synchronously implemented and turned into an Elastic Circuit by using a latency-insensitive protocol in the design’s computational processes data transfers. Both designs are validated in an FPGA platform, using well known synchronous design tools and flow.

The timing and area comparison between the designs demonstrates that the Elastic version can present performance advantages for more complex systems at the price of increased area.

These results show that the Elastic Design methodology is a good candidate for designing complex integrated circuits without costly iterations between design phases. This methodology also leverages the reuse of the mostly adopted synchronous design tools, resulting in a cost-effective alternative for designers.

Keywords: Elastic Circuits. Digital IC. IC Design Methodology. FPGA. ASIC. Synchronous Circuits. Asynchronous Circuits.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

AC	Accumulator
ADDR	Address
ALU	Arithmetic Logic Unit
DSM	Deep Submicrometer
DUT	Design Under Test
EB	Elastic Buffer
EB	Elastic Buffer
EDA	Electronic Design Automation
FF	Flip Flop
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HLT	Halt
IC	Integrated Circuit
IP	Intellectual Property
IR	Instruction Register
JMP	Jump
LDA	Load Accumulator
MAR	Memory Address Register
MDR	Memory Data Register
MEM	Memory
MUX	Multiplexer
PC	Program Counter
PCTRL	Patient Control Unit
RTL	Register Transfer Level
SDC	Synopsis Design Constraint
SELF	Synchronous Elastic Flow
STA	Store Accumulator

SUMMARY

1 INTRODUCTION

Over the past decades, the use of electronic components by industries like automotive, aerospace and personal systems has rapidly increased. This expansion brought a big electronics technology leap in a short time, as mapped by the Moore Law for the integrated circuits (ICs), which have to perform really complex functions in the smallest area, power consumption and time possible.

The development of IC design methodologies and tools was a key factor to enable the advance of chips capabilities, exploring the semiconductor fabrication processes enhancements. As a well stablished concept, the Synchronous IC Design has been the main methodology adopted by the semiconductor industry in its development.

However, with the process nodes shrinking, IC designers face more challenges to keep increasing their clock based designs' performance due to issues not perceived in previous technologies, like increased wire-latencies.

Aiming at solving such problems, alternative designs methodologies have been researched and proposed. The Asynchronous Design methodology claims to brake the synchronous circuits' fixed clock period dependency to compute data, what would bring more flexibility to designs and avoid data propagation time impact on functionality.

Representing a big paradigm change to the IC design community and having few Electronic Design Automation (EDA) tools support, the asynchronous circuits are not largely adopted. With the intent of reusing several aspects of the synchronous design concepts, the Latency Insensitive and Elastic Design methodologies arise as alternatives that are not so disruptive and solve the technology advance related issues without leaving behind all the synchronous knowledge developed, at the cost of increased area.

Even with the advances of synchronous based tools, the real circuit latencies uncertainty in the early stages of development is a key factor that leads to timing closure problems in current designs, resulting in cost increase due to reiterations between design steps. An example is the accurate physical synthesis delay estimation identifying timing problems and requiring rework in the Register Transfer Level (RTL) design.

A key aspect to sustain the advance of complex ICs in a cost-effective fashion is the exploration of new design methodologies, like the Elastic design. Such methodologies leverage the well stablished ground to solve the technology advance issues and avoid the drawbacks of the synchronous flow in timing closure process, without impacting the performance of synchronous designs.

This work exercises the Elastic Design methodology by converting a synchronous implementation of the Neander processor developed on top of well-known synchronous design practices and tools to demonstrate the impacts and requirements of the new methodology adoption. The Elastic Circuit is deployed in a Field Programmable Gate Array (FPGA) platform and the resulting functionality, area and performance are analyzed against the expected outcomes.

A review of the Latency insensitive and Elastic design methodology is done in section 2. The Neander architecture background is explained in section 3 to review the processor's functionality and characteristics. Section 4 brings an overview of this works' target, an Elastic version of the Neander processor evaluated in an FPGA platform. The whole design flow is explained in Section 5, detailing each design phase performed to generate the Elastic circuit, which have its results analyzed in section 6. Section 7 concludes this work with the key aspects and remarks, opening room for future work.

2 ELASTIC CIRCUITS

Given the advance of deep submicrometer technologies (DSM) with process nodes from 0.1um to tens of nanometers, complex Integrated Circuit designs have to overcome new challenges.

As the logic cells become smaller and faster, the distance between those cells increases, and that makes wire latencies more significant in the overall design timing challenge. Other factors that contribute to the wire latency problem are the increase in operating frequencies and die size, which make even smaller the time available for a signal to travel from one logic unit to another.

Although there are process related techniques that try to solve the metal lines latency impact increase, like copper metallization and low-k dielectric insulators, these workaround solutions are not sufficient to completely solve the interconnect delay problem (BOHR, 1998), (FLYNN, 1999).

Asynchronous circuits paradigm is an alternative to overcome the wire latency ratio increase effect, since it is not based on a clock cycle and execution occurs when data is ready to be consumed. However, the lack of EDA tools compatible with an asynchronous design flow and the higher complexity of such designs, when compared to synchronous circuits, prevent the asynchronous circuits adoption by the larger part of the IC design community.

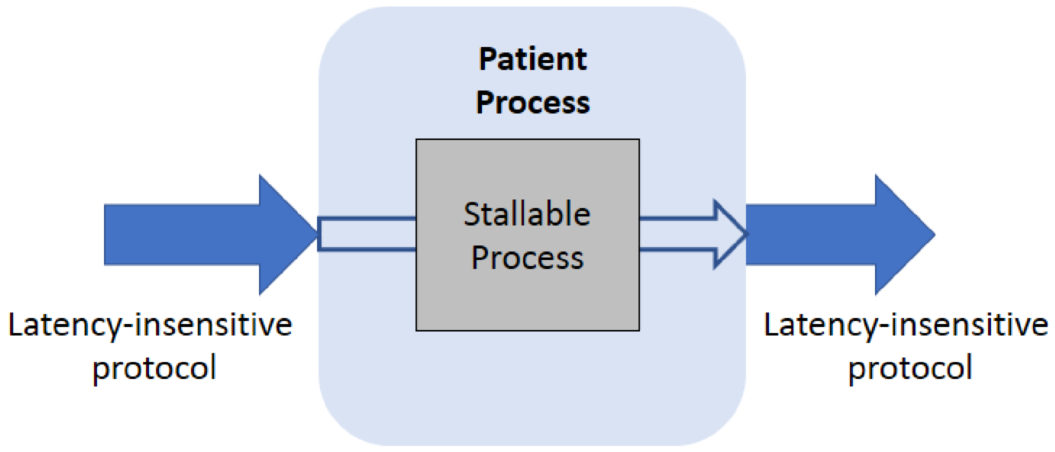
The Latency-Insensitive theory (CARLONI, 2001) was proposed as a conciliatory solution between the asynchronous world and the traditional synchronous design methodology.

The principle of separating computation from communication in a design is the basis of the Latency-Insensitive theory. The design parts responsible for data computing are designated as computational processes, while the communication channels are responsible for exchanging the processed data.

By decoupling communication from computation, Latency-Insensitive design suggests that the computational processes in a system exchange data over communication channels which implement an abstract latency-insensitive protocol (CARLONI, 1999). The latency-insensitive protocol ensures that computational processes’ data exchanges are completed despite of channel latency variations.

For a computational process to be compliant with the latency-insensitive protocol, it has to be stallable, meaning that it can maintain its current state if a stalling condition is applied. For instance, a Finite State Machine (FSM) can be stallable if its state transitions can be stalled by a stop input. The latency-insensitive version of a computational process is a Patient Process, which is obtained by making a stallable process implement a latency-insensitive protocol, as shown in the Figure 1 below.

Figure 1 – Patient Process composition

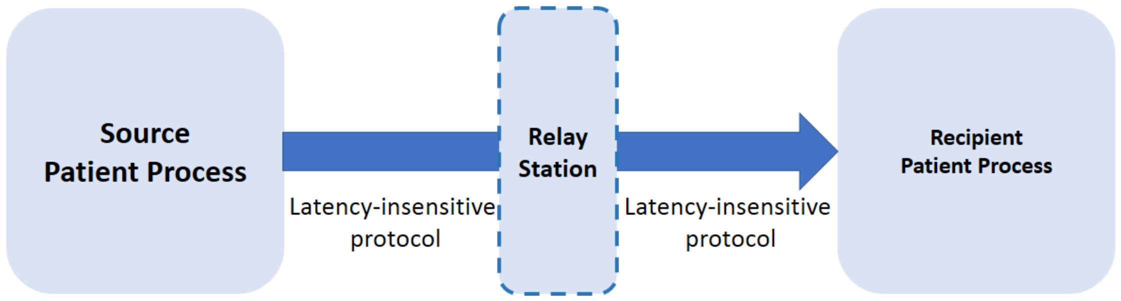


Reference: Carmona (2001).

Hence, a latency-insensitive design is composed by Patient Processes that communicate with each other through a latency-insensitive protocol. The protocol makes a data source Patient Process wait for the availability of the data recipient Patient Process.

Once the design is adapted to be latency-insensitive, a lengthy communication channel that does not meet the timing constraints of the system's clock can be segmented by the insertion of Relay Stations. These components do not have any computation purpose, only serving the purpose of partitioning a long communication channels while maintaining a functional equivalency in the data flow, meaning that the Relay Stations do not interfere in the data sequence generated by the Patient Process, as shown in the Figure 2 below.

Figure 2 – Relay Station in a latency-insensitive channel



Based on Elastic Buffers (EB), that use control signals to manage valid data transfers, the SELF (Synchronous Elastic Flow) (CORTADELLA, 2006) is a latency-insensitive protocol. Its specification provides an abstract model for elastic channels and buffers. Its main features are the efficient implementation of elastic communication channels and an automatable design methodology enablement.

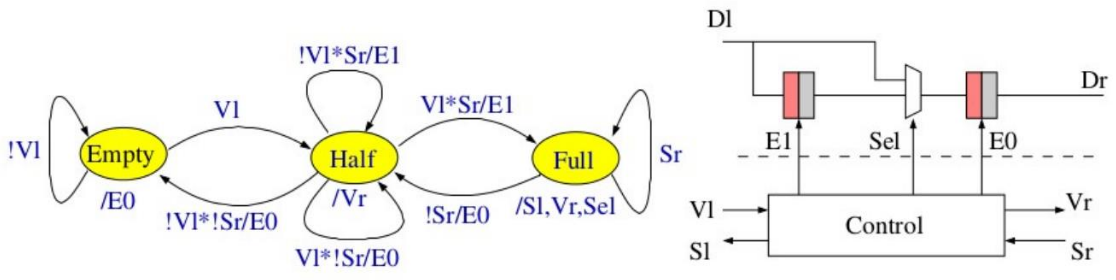
An elastic channel has three possible states: Transfer, Idle and Retry. The first state occurs when both master and slave pieces of the system are able to transmit/receive data. An Idle state happens when the sender is not providing valid data to the receiver. When the master is generating valid data but the receiver is not able to process it, the system is in Retry state.

An elastic buffer is divided in datapath and control. The datapath is where its data storage elements are present, while the control logic determines the system state based on its master/slave interfaces' control signals.

The Elastic Buffers purpose is similar to the Relay Stations, with the advantage of not inserting additional latency when replacing a synchronous component of the original design. This is achieved through optimizations in the Elastic Buffers organization. Several implementations of elastic buffers are possible. The variations include, for instance, the use of registers or latches, storage capacity, datapath and control organization optimizations and double-pumping, which ensures two data moves within the same system clock cycle.

This work uses the W2R1 elastic buffer control and datapath definition, which uses registers and a simple FSM controlling the system in its three possible states. Figure 3 below shows the control and datapath specification for the chosen Elastic Buffer organization.

Figure 3 – Specification for the W2R1 EB



Reference: Cortadella (2005).

The Latency-Insensitive Theory and the SELF maintain the simplicity of synchronous circuits by applying design methodology changes on top of the existing and well established synchronous design methodology and toolset turns these approaches more attractive to the IC design community.

2.1 Advantages and Liabilities

The main advantage of the elastic paradigm is its communication latency insensitivity. This provides robustness to the circuits, as the process and/or operating conditions variations and its associated timing variation is managed by the latency-insensitive protocol.

Since the latency-insensitive protocol makes computational processes dependent on data validity, power optimizations can be performed making the system active only upon the presence of data to be processed.

Beside the intrinsic advantages of the elastic designs, another positive characteristic is that elasticity enables performance optimizations by applying transformations like bubble insertion, variable-latency units, speculative and out-of-order executions, as mentioned in (CORTADELLA, 2010). These correct-by-construction transformations preserve the behavior of the circuit while boosting its performance.

Among all the advantages mentioned, the “elasticization process” brings some disadvantages as well. One of the main drawbacks is the additional latency that this methodology inserts in communication channels when compared to traditional synchronous circuits that have no buffers between computational processes. This added latency is mainly perceived when the communication channel bandwidth utilization is low. The optimization transformations previously mentioned also help reducing the impacts of the increased latency.

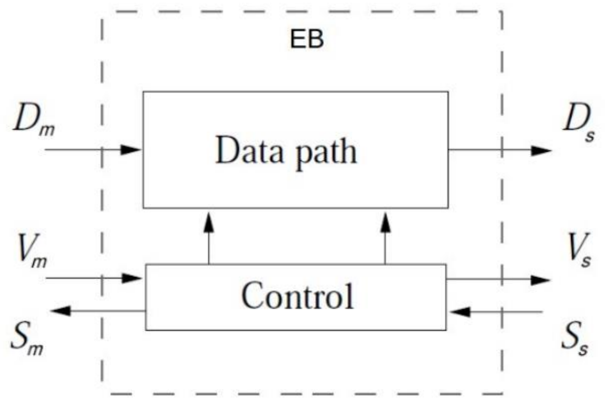
Another important overhead of the elastic paradigm is the higher area inherent to its control and storage logic (Elastic Buffers). If the original circuit's area and/or complexity isn't high enough to justify the elastic infrastructure, the elasticization process will probably not bring significant advantages.

2.2 Elasticization Process

The elasticization process targets at converting a traditional synchronous design into an elastic one. This process involves turning the original design units into latency-insensitive protocol compliant ones.

An Elastic Buffer (EB) is responsible for implementing the elastic protocol, having storage units used to keep data when the following Patient Process is busy. Different EB implementations are possible, like using registers and/or transparent latches, as explained in (CORTADELLA, 2006). Regardless of the different organizations, the basic structure of an Elastic Buffer is composed of a control and a datapath, as shown in Figure 4 below.

Figure 4 – Elastic Buffer base structure

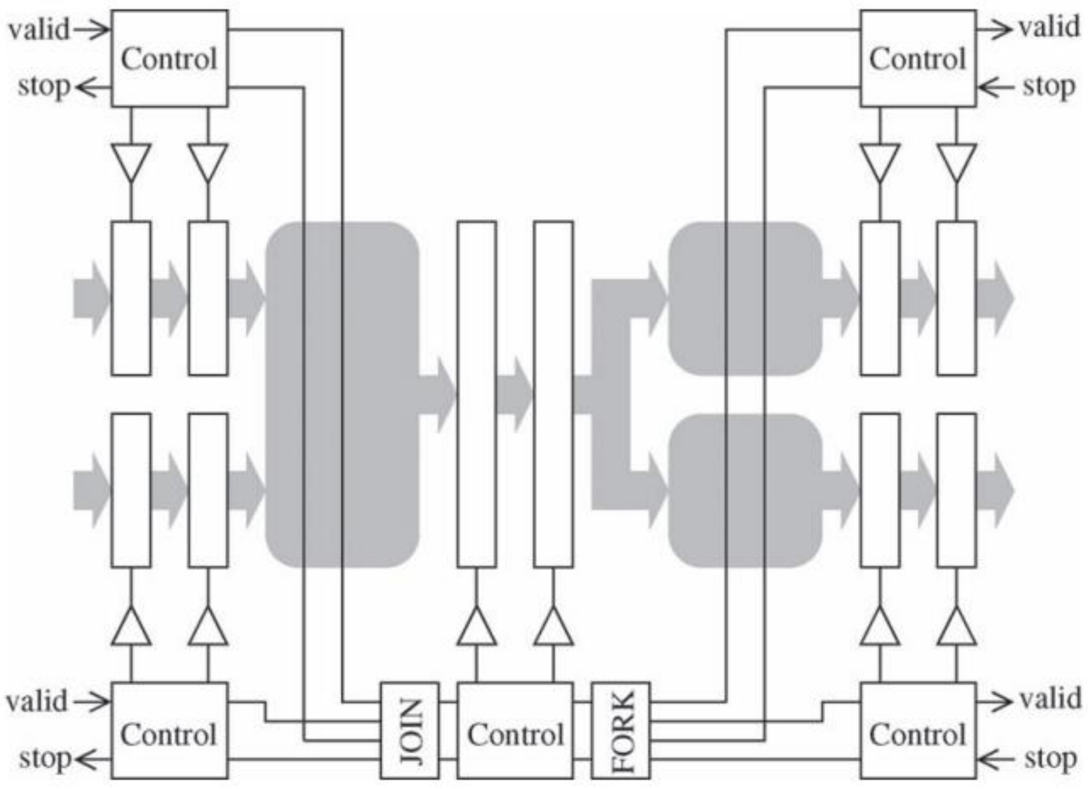


Reference: Cortadella (2005).

The elasticization process can be applied in different granularities, depending on the design characteristics, target application and area budget. For SoC designs, where the reuse of IP is a strong demand, Elastic Buffers can be inserted in between the design's black boxes, as long as these boxes are Patient Processes, as explained in (CARLONI, 2002). Finer grain elastic circuits can be obtained by replacing every register in the design by an EB, at the expense of highly increased area, presented in (JACOBSON, 2002).

Depending on the number of channels communicating to one module, the fork or join of data from different sources might be needed. This requires a special control in the involved Elastic Buffers, because the target logic availability and the data sent to/from it have to be synchronized across all involved modules/channels in the communication. An example of join/fork Elastic Buffers can be seen in Figure 5, from (CARMONA, 2009).

Figure 5 – Synchronous elastic module with multiple inputs and outputs.



Reference: Carmona (2009).

By having the ability to tolerate latency changes, the elastic systems enable performance boost by making communication timing constraints more flexible and making optimization techniques possible. This tolerance helps to reduce the increased latency of the elastic protocol when the elastic channel bandwidth capacity is not being fully used, for example. Some of the techniques, such as Recycling, Early Evaluation, Anti-token Insertion, Variable-latency units and Speculative Execution are presented in (CORTADELLA, 2010). For example, Variable Latency units introduce the concept of logic module tuned for the most common case of its computation, while adding latency to the least common case.

3 NEANDER PROCESSOR

The Neander processor is a hypothetic machine conceived for education purposes. Since it is a simple machine, it facilitates the introduction of concepts such as computer organization and architecture. In summary, the main characteristics of the Neander processor are (WEBER, 2009):

- Addressing and data width of 8 bits
- Data represented in two complement
- One accumulator of 8 bits (AC)
- One program counter of 8 bits (PC)
- One state register with two condition codes: Negative (N) and Zero (Z)

The instruction set that the Neander implements is composed by data movement (LDA, STA), arithmetic (ADD), logic (AND, OR, NOT) and branch (JMP, JN, JZ) instructions, besides the HLT and NOP instructions that do not perform data operations. Each instruction is coded in 4 bits, what defines the minimum size of the Instruction Register (IR) that will hold the instructions read from memory.

The 8 bits addressing width allows the Neander to address a maximum of 256 positions. Since the processor works with 8 bits words, this gives a total amount of 256 bytes of addressable memory. The Neander addressing scheme is direct, so every address used by a program is directly mapped to the position in the memory. The memory address for the current state of the execution is held in the Memory Address Register (MAR).

Some of the Neander instructions are followed by an extra byte that carries an operand address (ADDR). This address can represent the memory position that contains the data to be used in the current operation or be the actual memory address to which the PC has to be pointed, as shown in the Table 1 below.

Table 1 – Neander processor instruction set

Instruction code	Instruction	Description
0000	NOP	No operation.
0001	STA ADDR	Stores the data of AC into MEM(addr).
0010	LDA ADDR	Loads the data of MEM(addr) into AC.

Instruction code	Instruction	Description
0011	ADD ADDR	Sums the data of MEM(addr) with AC and stores the result in AC.
0100	OR ADDR	Logic “OR” of the data of MEM(addr) with AC and stores the result in AC.
0101	AND ADDR	Logic “AND” of the data of MEM(addr) with AC and stores the result in AC.
0110	NOT	Inverts the data of AC and stores the result in AC.
1000	JMP ADDR	Branches the execution to the addr position in memory.
1001	JN ADDR	Branches the execution to the addr position in memory if the NZ indicates a negative value.
1010	JZ ADDR	Branches the execution to the addr position in memory if the NZ indicates a zero value.
1111	HLT	Halts the processor execution.

Reference: Weber (2006).

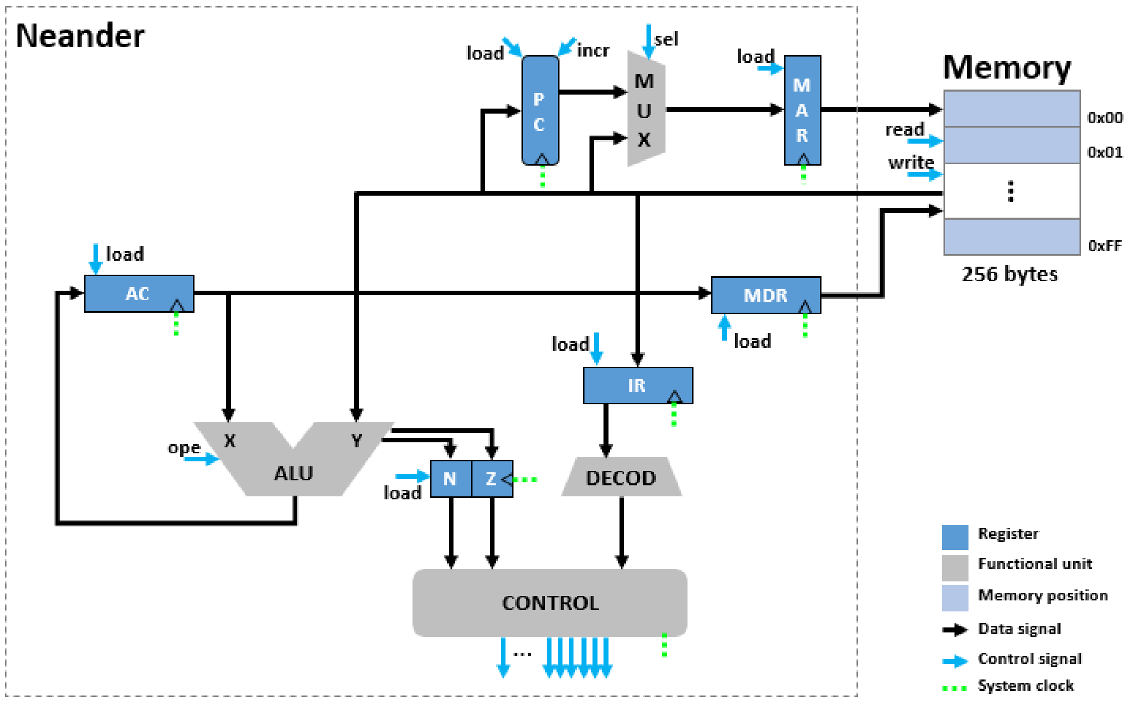
For STA operations, the Neander uses a Memory Data Register (MDR) to keep the data that will be stored in the memory address indicated by MAR. This register completes the list of registers needed by the Neander’s datapath:

- AC – Accumulator
- IR – Instruction Register
- MAR – Memory Address Register
- MDR – Memory Data Register
- NZ – Negative / Zero condition codes
- PC – Program Counter

The Control unit is responsible for controlling the data flow in the Neander datapath. The unit does this by arbitrating when each register needs to store the data in its input, through the *load_* control signals; by controlling the memory read and write accesses; and by selecting if the MAR uses the PC or the operand address (ADDR) read from memory to point to a determined memory position. All these control actions are dependent on the instruction being executed and the execution time of the instruction cycle, which will be explained below.

The Control unit is also responsible for selecting the appropriate operation to be executed by the Arithmetic Logic Unit (ALU) based on the current instruction and execution step. Along with the datapath, the control unit composes the Neander organization, as shown in the Figure 6 below.

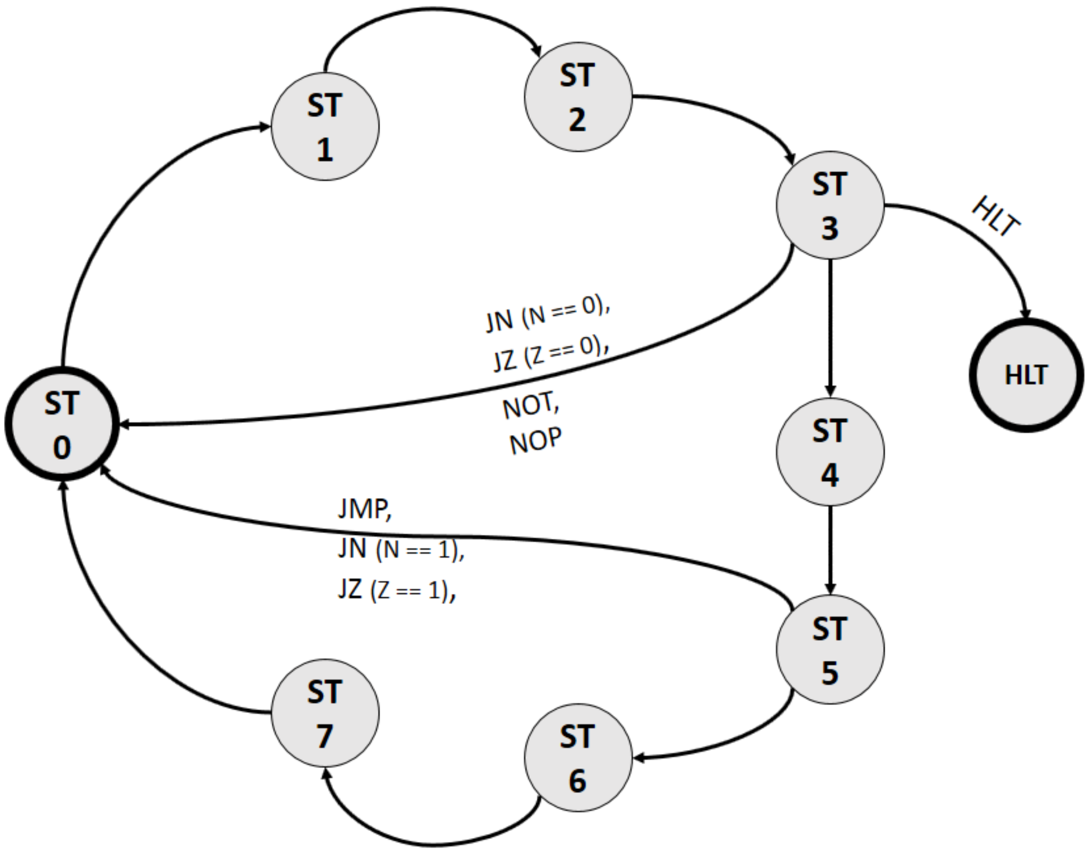
Figure 6 – Neander organization



Reference: Weber (2006).

Based on the data movements between the datapath elements defined by each Neander instruction, the Control unit can be represented by an FSM with eight states, as shown in Figure 7 below. Each state represents one execution step and the control actions performed in each state are instruction dependent, as well as the transitions. For simplicity, the conditions for each transition and the control actions of each state are omitted. The only conditions shown are the ones that make the execution flow go back to state 0 or halt the processor.

Figure 7 – Neander control unit FSM

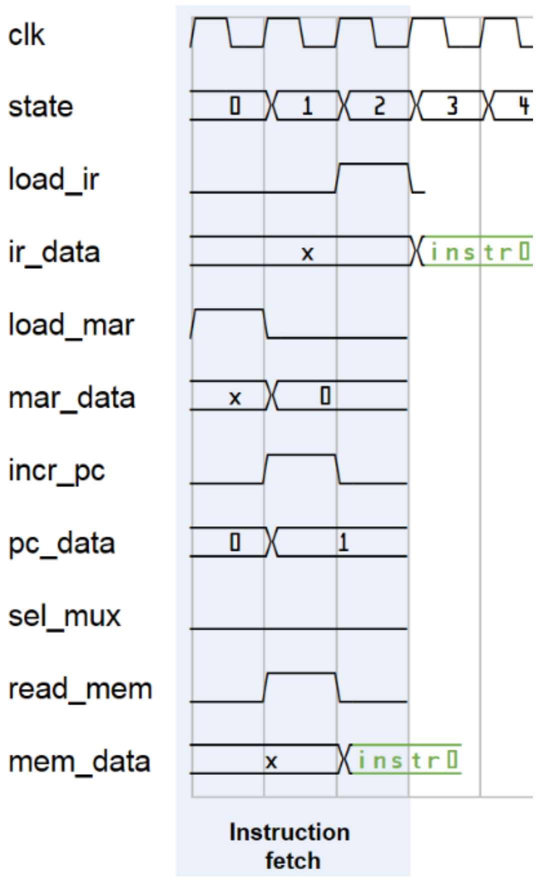


An instruction cycle is the group of data movements and control actions that compose an instruction execution steps. In the Neander architecture, this instruction cycle is divided in two phases: instruction fetch phase and execution phase.

The instruction fetch phase is the first phase of an instruction cycle and does not depend on the instruction being executed. In fact, the instruction that will be executed subsequently is fetched from memory in this phase, as shown in Figure 8 below. From state 0 to state 2, the control unit accesses the memory to fetch the instruction to be stored into IR and increments the PC to point to the instruction's operand address or the next instruction.

After the instruction is fetched from memory, the Neander processor goes to the next phase of the instruction cycle, the execution phase. Depending on the instruction to be executed, this phase's actions can take from 1 (i.e.: NOP) to 5 (i.e.: STA) execution steps.

Figure 8 – Neander instruction fetch phase



As an example, Neander can execute a simple program that sums the values of three subsequent memory positions (WEBER, 2009). The program's instruction section goes from positions 0 to 127 of the memory and the data section goes from position 128 to 255.

The program will sum the data in the positions 128, 129 and 130, and finally will store the result in the position 131. The initial state of the memory is shown in the Figure 9 below. The memory data, addresses and the instructions operands are represented in hexadecimal base.

Excluding the instruction fetch phases, the program execution is represented in the Figure 10, along with the memory state at the end of the program execution. Note that the control unit has issued one *load_ir* pulse for each instruction executed.

Figure 9 – Sum program

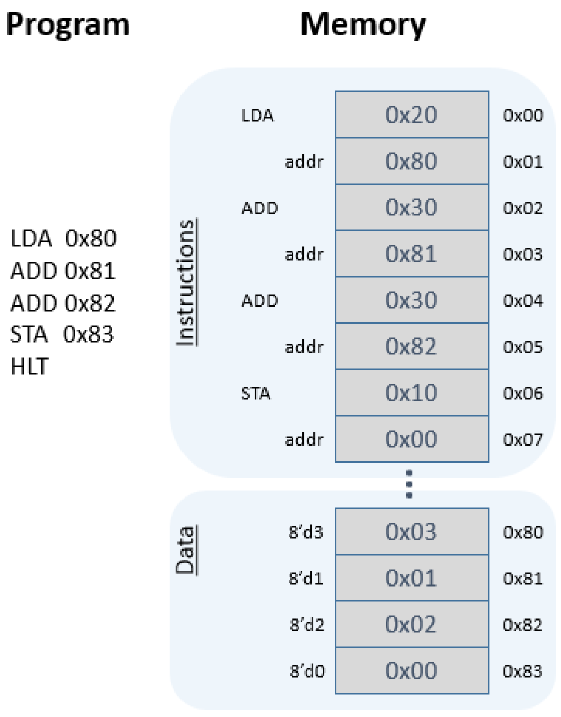
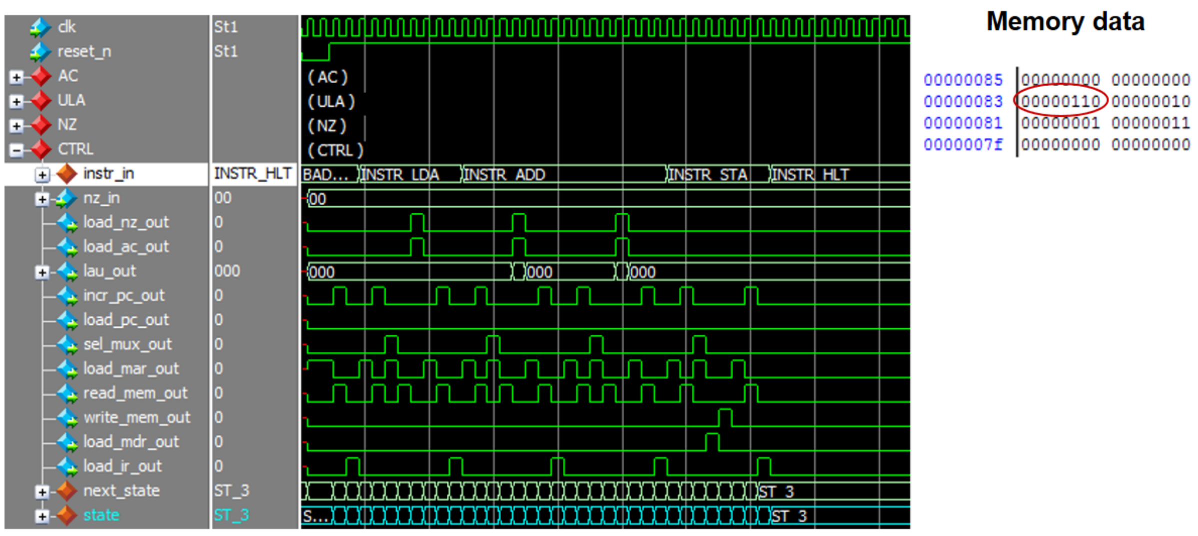


Figure 10 – Sum program execution


```
clk          St1
reset_n      St1
AC                    (AC)
ULA                   (ULA)
NZ                    (NZ)
CTRL                  (CTRL)
instr_in     INSTR_HLT  BAD... INSTR_LDA  INSTR_ADD  INSTR_STA  INSTR_HLT
nz_in        00         00
load_nz_out  0
load_ac_out  0
lau_out      000        000  000  000
incr_pc_out  0
load_pc_out  0
sel_mux_out  0
load_mar_out 0
read_mem_out 0
write_mem_out 0
load_mdr_out 0
load_ir_out  0
next_state   ST_3       ST_3
state        ST_3       S...  ST_3
```

Memory data

```
00000085 |00000000 00000000
00000083 |00000110 00000010
00000081 |00000001 00000011
0000007f |00000000 00000000
```

The LDA instruction moves the data from MEM(0x80) to the AC. This instruction's execution phase takes until state 7 and requires two memory read accesses: one for reading the operand's address and another for reading the operand's data in the position indicated by the operand's address (in this case, position 0x80).

The ADD instructions make the ALU sum the data currently in the AC with the operand’s data indicated in the instruction (in this case, position 0x81 and, subsequently, position 0x82). This instruction also requires the same two memory read accesses as the LDA instruction.

Finally, the STA instruction moves the data from AC to MEM(0x83). This instruction requires one memory read access, for reading the operand address, and one memory write access, for storing the data. After the STA instruction, the Neander processor is halted by the HLT instruction.

By using the base concepts of computer organization and architecture, the Neander processor serves as a great platform for organizational concepts exploration, such as the application of the latency-insensitive design methodology on top of the regular synchronous design.

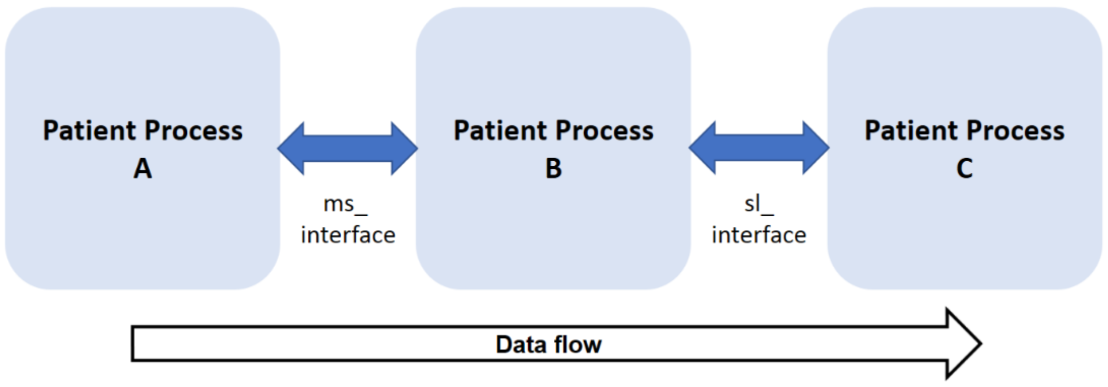
4 ELASTIC NEANDER IN FPGA

The main purpose of this work is to exercise the elastic design methodology (SELF), converting an existing system into an elastic one, which is capable of dealing with latency variations without having its functionality compromised. Both the original and the converted systems are evaluated in a FPGA platform to demonstrate that the proposed methodology does not interfere with the original system’s functionality and to enable the performance and area evaluation after the design and synthesis flows.

The elastic design methodology enables circuit designs to be more robust to process and environment variations by implementing an abstract latency-insensitive protocol between all computational processes in the system. This characteristic makes the data flow in the design not fully dependent on one clock cycle period, since a Patient Process that is receiving data only starts computation upon the presence of valid data on its input channel, leaving the stall caused by the protocol.

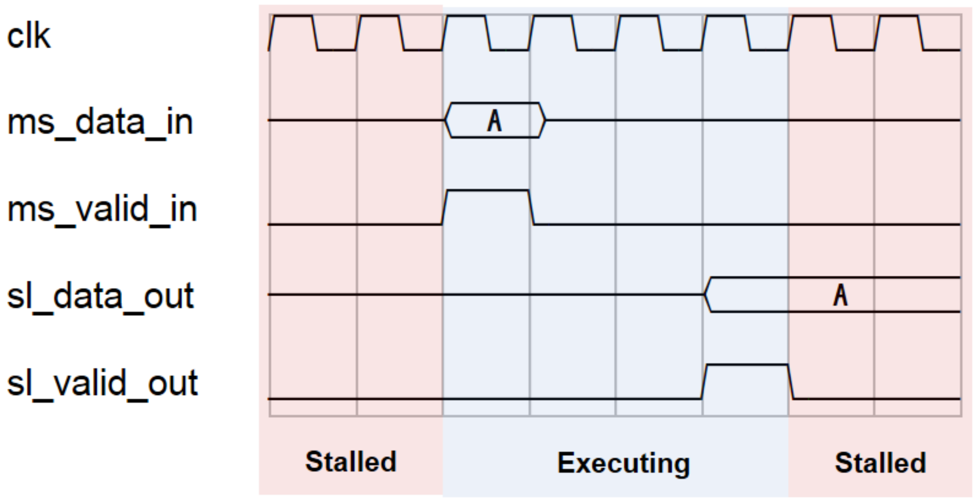
Each Patient Process in the system has at least two elastic inferfaces: the *ms_* interface and the *sl_* interface. The *ms_* signals compose the interface with the process generating the data (process A in Figure 11) that will be consumed by the current process (process B in Figure 11). The *sl_* signals interface the current process (process B in Figure 11) with the next process in the data flow (process C in Figure 11), which consumes data from the current process.

Figure 11 – Patient Process interfaces



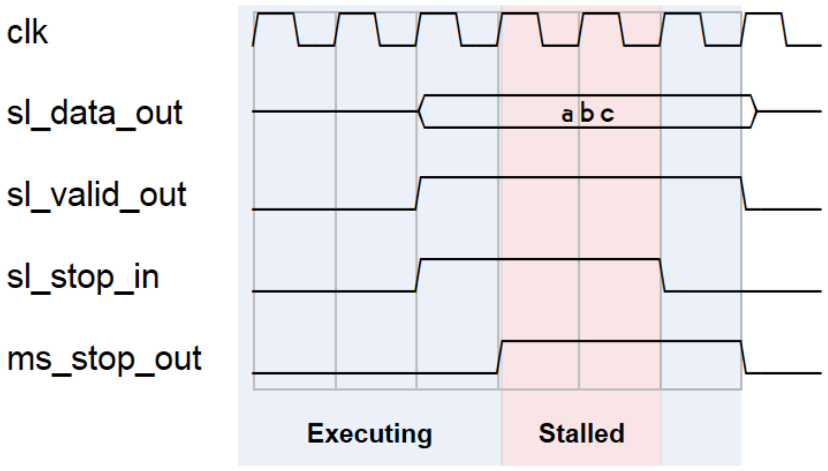
The Figure 12 below shows a Patient Process in its stalled period (shaded in red), when there is no valid data to be consumed. The blue shaded area shows its executing period, from valid input data arrival to a valid data output.

Figure 12 – Patient Process B stall and execution periods



The SELF protocol also defines that a process is stalled when the consuming process (i.e.: process B in Figure 11) is not able to receive new data, signaling this condition to the source process (i.e.: process A in Figure 11) through the stop signal, as shown in Figure 13 below. This event is called backpressure, and is another key characteristic that enables the latency-insensitivity in elastic systems.

Figure 13 – Patient Process A stall and execution periods



By implementing the characteristics presented above, the application of the elastic design methodology in an existing system breaks the dependence on a rigid amount of clock cycles between the system's processes. For instance, a data transfer can take N amount of clock cycles in a determined process technology and N*2 in another process technology, and the elastic system will still work correctly, being latency-equivalent to the original synchronous circuit.

Having simple organization and architecture, the Neander processor is a good candidate for testing the elastic design methodology to convert a strict synchronous system (CARLONI, 2001) into an Elastic one. The components in its organization enable the application of Elastic concepts like conversion of Stallable Processes into Patient Processes and the insertion of EBs in a timing critical communication channel, to segment the delay and support timing-closure.

The FPGA platform is used in this work as an evaluation environment tool to prove that, after the elasticization process, the Neander design can still normally perform its functions in a real device. A simple demonstration is built in the Terasic DE-0 Cyclone III evaluation board, to observe the functional behavior of the Neander processor across synchronous and elastic designs.

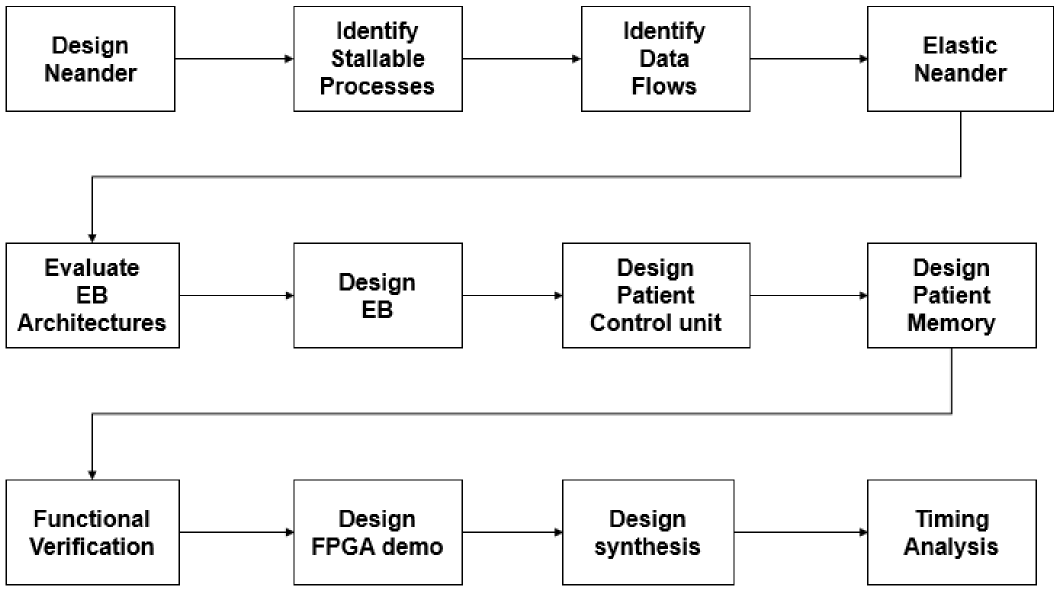
Other important aspect of the use of the FPGA platform in this work is to benchmark how these devices perform when implementing new architectural design approaches such as the elastic design methodology.

The full flow of this project is represented in Figure 14 below, which shows an overview of the necessary steps for the completion of the Elastic Neander, starting from a synchronous Neander implementation until its demonstration on the FPGA development board:

- **Design Neander** – Implementation of a synchronous Neander design.
- **Identify Stallable Processes** – Identification of Neander Computational Processes that can be stalled.
- **Identify Data Flows** – Identification of data movement paths in the Neander design.
- **Elastic Neander** – Overall Elastic Neander organization.
- **Evaluate EB organizations** – Evaluation and definition of an adequate Elastic Buffer organization.
- **Design EB** – Implementation of the chosen EB organization.
- **Design Patient Control Unit** – Adapt Neander control unit to be a Patient Process.
- **Design Patient Memory** – Adapt Memory module to be a Patient Process.
- **Functional Verification** – Verification of the design's functionality in simulation environment.
- **Design FPGA demo** – Integration of the demonstration required modules.

- **Design Synthesis** – Synthesis of the design to the target FPGA device.
- **Timing Analysis** – Timing closure analysis of the synthesized circuit.

Figure 14 – Elastic Neander Project phases



5 ELASTIC NEANDER PROJECT

This section describes the sequence of steps performed to design, verify and validate the Elastic Neander design, starting from the Synchronous Neander implementation. After presenting the design steps, a comparison between the original Neander and the Elastic Neander circuits is done.

The design units are coded in RTL level Verilog, using the Altera Quartus 13.1 suite as the Logic and Physical synthesis tools, and the Mentor Graphics ModelSim Altera 10.1d as the simulation tool.

5.1 Neander Design

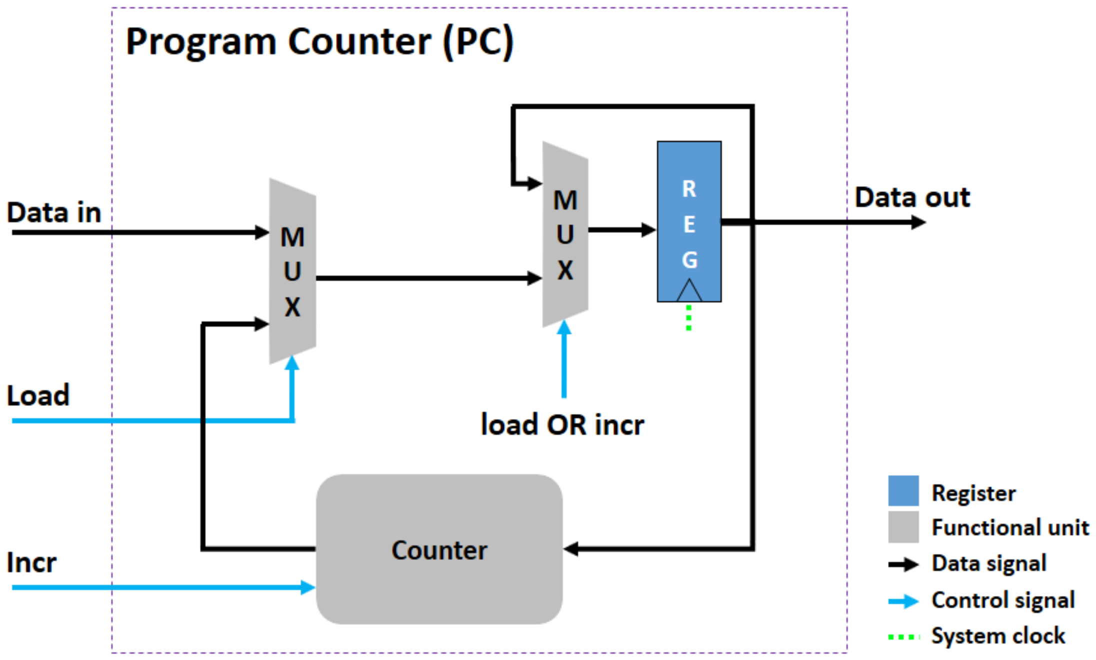
The Neander design is based on the specification from (WEBER, 2009), which is briefly explained in section 3. This design is a strict synchronous system, meaning that the data transfers between the Computational Processes in the system are purely dependent on the system clock period, being specified as a multiple of the clock cycles.

This design is composed by registers as the storage elements of the datapath, a control unit and combinational elements:

- 5 registers (AC, IR, MAR, MDR, and NZ)
- 1 special register with integrated counter (PC)
- 1 ALU
- 1 Control unit
- 1 Multiplexer for selecting MAR input

The PC is implemented as a regular register with the addition of an integrated counter, as shown in the Figure 15 below. The PC has two control inputs: Load and Incr, used to load the output register either with the data in its input or with the counter value, respectively.

Figure 15 – Neander Program Counter



The ALU is a purely combinational logic module that is responsible for executing the following operations with its X and Y inputs of 1 byte each and for generating the N and Z condition codes, stored in the NZ register:

- ADD – Adds the X and Y inputs
- AND – Logic AND between X and Y bits
- NOT – Inverts all X bits
- OR – Logic OR between X and Y bits
- Y – Outputs the Y input

The Control unit is the responsible for controlling the data flow between the Neander datapath registers, selecting the ALU operation and operating the Memory. The control signals depend both on the execution state and the instruction, as specified in the Table 2 and Table 3 below, where *sel* is the selection for the MUX in the MAR input, *ld* is a load signal to the registers, *incr* is the PC increment sgnal and *rd/wr* are the memory control signals.

Table 2 – Control signals during STA, LDA, ADD, OR, AND and NOT

Execution State	STA	LDA	ADD	OR	AND	NOT
st0	sel=0, ld MAR	sel=0, ld MAR	sel=0, ld MAR	sel=0, ld MAR	sel=0, ld MAR	sel=0, ld MAR
st1	rd, incr PC	rd, incr PC	rd, incr PC	rd, incr PC	rd, incr PC	rd, incr PC
st2	ld IR	ld IR	ld IR	ld IR	ld IR	ld IR
st3	sel=0, ld MAR	sel=0, ld MAR	sel=0, ld MAR	sel=0, ld MAR	sel=0, ld MAR	UAL(NOT), ld AC, ld NZ, goto st0
st4	rd, incr PC	rd, incr PC	rd, incr PC	rd, incr PC	rd, incr PC	
st5	sel=1, ld MAR	sel=1, ld MAR	sel=1, ld MAR	sel=1, ld MAR	sel=1, ld MAR	
st6	ld MDR	rd	rd	rd	rd	
st7	wr, goto st0	UAL(Y), ld AC, ld NZ, goto st0	UAL(ADD), ld AC, ld NZ, goto st0	UAL(OR), ld AC, ld NZ, goto st0	UAL(AND), ld AC, ld NZ, goto st0	

Reference: Weber (2006).

Table 3 – Control signals during JMP, JN, JZ, NOP and HLT

Exec State	JMP	JN, N=1	JN, N=0	JZ, Z=1	JZ, Z=0	NOP	HLT
st0	sel=0, ld MAR	sel=0, ld MAR	sel=0, ld MAR	sel=0, ld MAR	sel=0, ld MAR	sel=0, ld MAR	sel=0, ld MAR
st1	rd, incr PC	rd, incr PC	rd, incr PC	rd, incr PC	rd, incr PC	rd, incr PC	rd, incr PC

Exec State	JMP	JN, N=1	JN, N=0	JZ, Z=1	JZ, Z=0	NOP	HLT
st2	ld IR	ld IR	ld IR	ld IR	ld IR	ld IR	ld IR
st3	sel=0, ld MAR	sel=0, ld MAR	incr PC, goto st0	sel=0, ld MAR	incr PC, goto st0	goto st0	halt
st4	rd	rd		rd			
st5	ld PC, goto st0	ld PC, goto st0		ld PC, goto st0			
st6							
st7							

Reference: Weber (2006).

5.2 Stallable Processes

The first step towards converting the Synchronous Neander into the Elastic Neander is to identify the stallable Computational Processes in the processor's organization, to enable the implementation of a latency-insensitive protocol.

The Control unit, being an FSM, is stallable upon the inclusion of extra control signals to prevent its next state logic from moving to a next execution state if the register that will receive data in is not ready to do so, for example. This leads to the definition that the Neander registers should also be able to be stalled, achieving a finer granularity Elastic circuit (CORTADELLA, 2006).

Being clock driven storage elements, the registers in the Neander organization are also Stallable Processes, since they can have its clock input gated and dependent on a control signal to store new data (CARLONI, 2001).

As the main storage element in the Neander system, the memory should also be possible to stall, otherwise the Control unit would lose data read from memory. Since the 256 bytes memory is a passive storage element activated by *rd* and *wr* signals managed by the Control unit and synchronously driven by the system clock, it is also a Stallable Process.

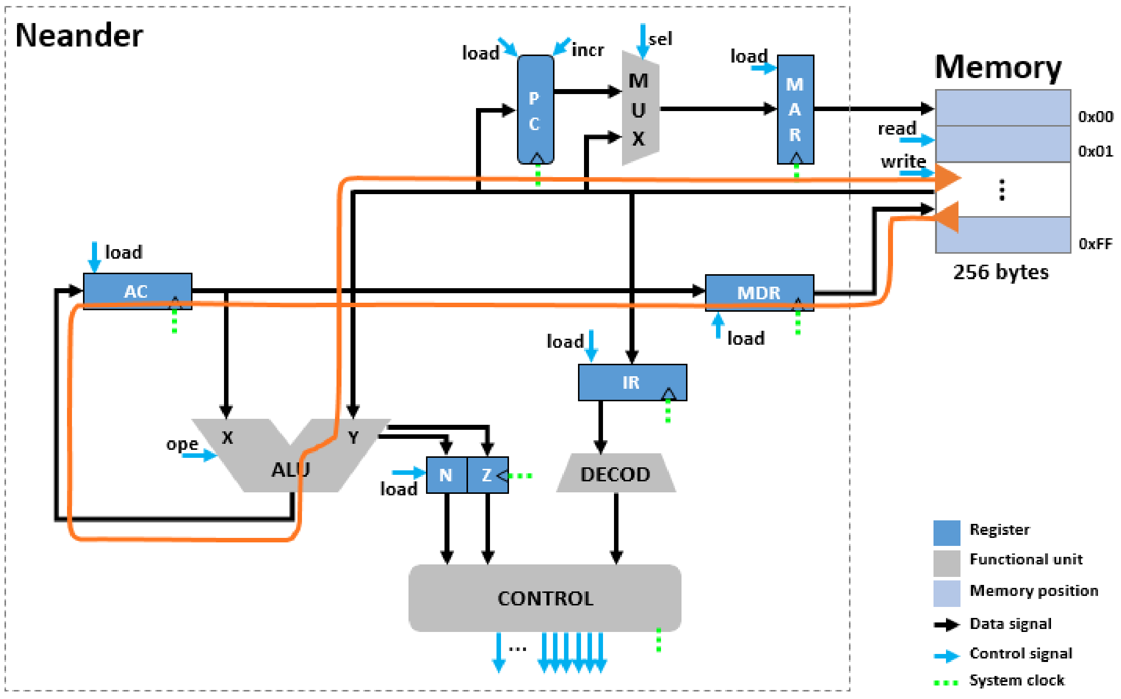
Some parts of the Neander organization are purely combinational pieces: the MUX in MAR's input; and the ALU. Initially, this work will consider the delay added by those parts of the system as negligible, but there are techniques to better map this type of Computational Processes' delay, as presented in section 6.

5.3 Data flows

To identify the elastic protocol interfaces needed and how they relate with each other, the Neander data flows had to be analyzed. The data flows are the possible paths through which data can travel in the Neander organization. In this particular design the data flow is variable, since a different destination is possible for each data read from memory, depending on the instruction and execution state.

A data flow identifies a backpressure path. In the Figure 16 below, the backpressure emitted by the memory data interface in a write event is highlighted. The arrows in the beginning and end of the data flow indicate the backpressure direction. Hence, the parts of the design that are affected by this backpressure path are: MDR -> AC -> MEM.

Figure 16 – Memory data write backpressure



Since AC's inability to receive new data does not necessarily mean that the IR, for instance, cannot receive new data from the memory, this backpressure path is actually dependent on the execution state. Hence, the Control unit is the responsible for receiving all the elastic protocol signals and managing which backpressure signal demands a stall of the processor’s execution at each execution state.

5.4 Elastic Neander

Having identified the Stallable Processes and the possible data flows that the Control unit has to consider to stall the processor execution, the Elastic Neander interconnections between the system’s Patient Processes can be defined as in the Figure 17 below.

The Neander registers are replaced by Elastic Buffers, which are storage elements capable of supporting latency variations by implementing the elastic protocol.

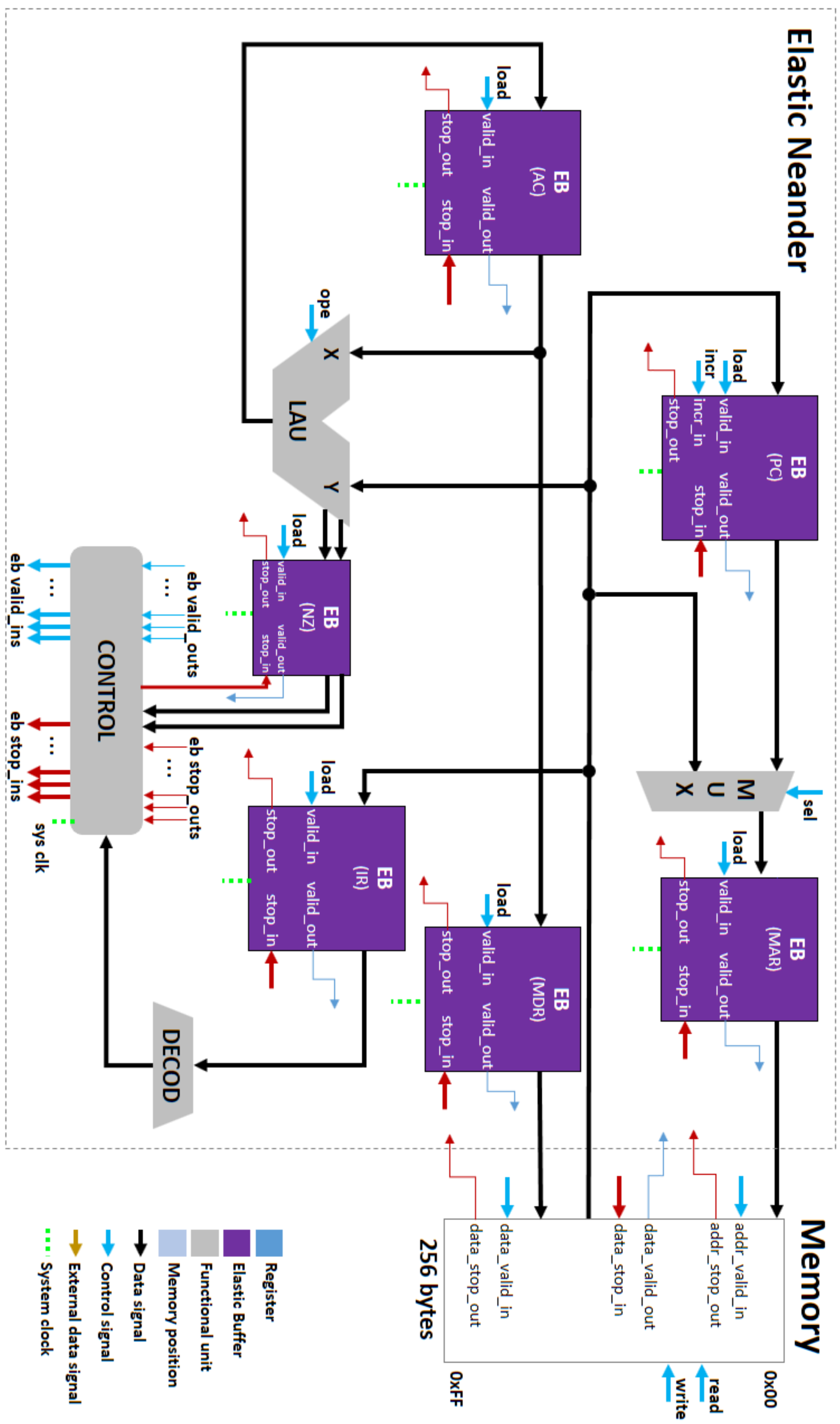
The memory of the Elastic Neander also implements the elastic protocol in three interfaces, as listed below.

- **addr_** – Address interface, used when Neander reads from or writes data into memory. The memory is the Slave of this interface.
- **data_in_** – Data input interface, used when Neander writes data into memory. The memory is the Slave of this interface.
- **data_out** – Data output interface, used when Neander reads data from memory. The memory is the Master of this interface.

Since the data flows depend on the Neander execution state, the Control unit implements several elastic interfaces, with all the registers (EBs) and with the memory. All the elastic protocol control inputs to the EBs and Memory are generated by the Control unit.

By receiving all the EBs and Memory *stop_out* signals, the Control unit has visibility of all possible backpressures in the system and is able to stall the execution if needed.

Figure 17 – Elastic Neander



Moreover, the Control unit receives all the *valid_out* signals from the other Patient Processes in the system. This allows the unit to continue the execution flow upon the presence of valid data from the process that generates data that will be consumed in the next execution state, for instance when the IR outputs data that will be used by the own Control unit in the execution phase.

In the Neander design, the data read from the memory is consumed by several processes, what could characterize a fork in the elastic protocol (CORTADELLA, 2006). However, this type of control is applicable when all the consuming processes use the data from the source process at the same time, what is not the case in the Neander processor. Hence, the control is the responsible for identifying which process consumes data from memory at each execution state and operate the elastic protocol accordingly.

5.5 Elastic Buffer organizations

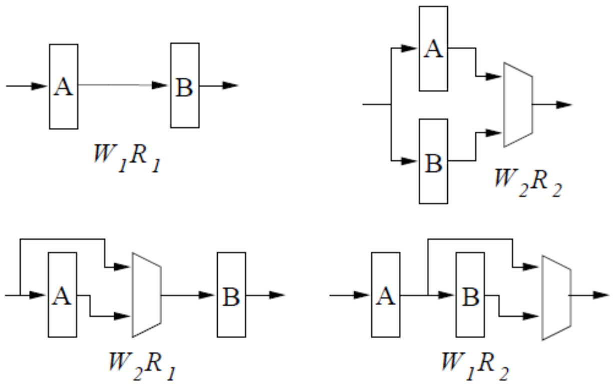
An elastic buffer's base organization is divided in a datapath – in which the data in the elastic channels is stored and output from the buffer – and a control part – which is responsible for controlling the data flow in the datapath storage elements and also implementing the master and the slave elastic interfaces of the buffer, as shown in Figure 4, in section 2.

As described in (CORTADELLA, 2006), the minimum depth of an elastic buffer is the sum of the forward latency and the backward latency in clock cycles. In this work, targeting the minimal latency possible added by the elastic buffers, both forward and backward latencies are equal to 1. This definition implies in the minimum buffer depth of 2 storage positions.

An elastic buffer can have different datapath organizations, what requires different control logic. Considering that the depth of the elastic buffers used in this work is equal to 2, the datapath organizations shown in Figure 18 were analyzed, where *W* stands for the number of write ports, while *R* stands for the number of read ports.

Since the forward and backward latencies are equal to 1, the *W1R1* elastic buffer cannot be implemented using regular flip flops in a same frequency clock edge, what would result in a latency of 2. Hence, this implementation requires the use of techniques to enable the propagation latency to be equal to 1, such as making each flop active to different clock edges, using double frequency clock on the right-hand flop or replacing the flops with transparent latches of different polarity.

Figure 18 – Elastic buffer datapath organizations



Reference: Cortadella (2005).

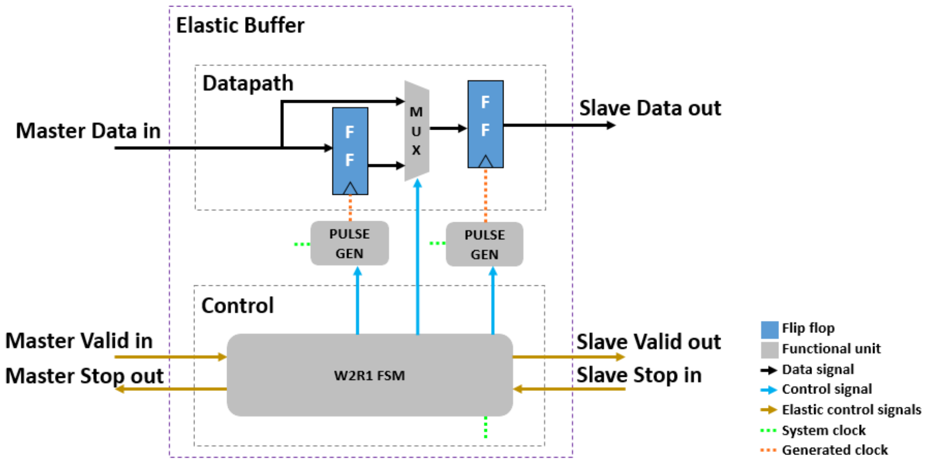
The options *W2R2* and *W1R2* do not have to overcome this *W1R1* characteristic, but make both flops active at all times, what leads to higher toggle rates. Considering the best balance between implementation simplicity and performance and also EB power consumption, this work makes use of the *W2R1* option. This organization leaves the flop next to the master elastic interface as a backup to the slave elastic interface flop, only activating the left-hand side flop when there is backpressure coming from the slave interface.

5.6 Elastic Buffer design

Having defined the W2R1 organization as the elastic buffer datapath, the correspondent elastic buffer control module is implemented. The datapath consists in 2 registers and a MUX to select the input of the slave side register depending on the state of the EB's control logic (EMPTY, HALF or FULL). The control coordinates the storage elements in the datapath and implements the master and slave elastic interfaces of the elastic buffer, as shown in Figure 19 below.

The *W2R1* control specification defines enable signals to the registers in the datapath. These enable signals could be used directly in the registers' clock enable port, if available. However, to enable portability between different tools and components libraries, the Elastic Buffer used in the Elastic Neander is implemented with pulse generators, that bypass the system clock in the presence of an enable input.

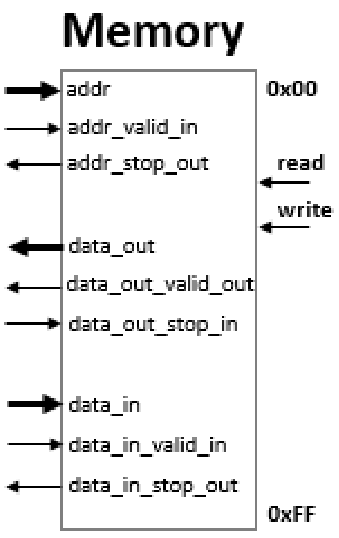
Figure 19 – Elastic Buffer design



5.7 Patient Memory

As well as the registers are replaced by equivalent elastic elements, the main storage piece of the Neander organization is also turned into a Patient Process, the Patient Memory, that is composed of three elastic interfaces, as shown in Figure 20 below: ADDR, DATA_IN and DATA_OUT.

Figure 20 – Patient memory elastic interfaces



The ADDR elastic interface is used by the Neander on both read and write requests to memory. Hence, this interface shall accept valid addresses if the memory is available to perform a read or a write operation. The ADDR is a slave elastic interface since it receives data (address) from a master, in this case, MAR.

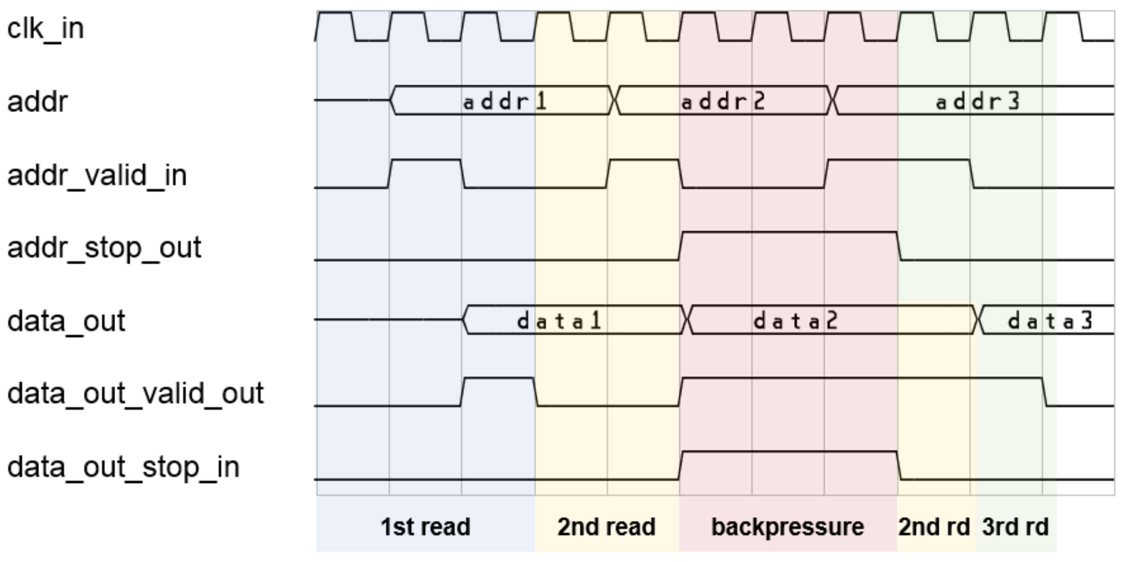
During memory write operations, the DATA_IN interface is used after a valid address has been provided to the memory. The data input is a slave interface that can generate a backpressure to the Neander organization while the memory is not able to receive a data write.

In read operations, after issuing a valid data in the ADDR interface, the Control Unit uses the DATA_OUT interface to get the data from memory. The latter is a master interface, since it outputs data to be consumed by another elastic interface. Hence, the DATA_OUT can receive backpressure from the other elements in the Neander organization.

To implement the elastic interfaces and be a Patient Process, the memory needs to be stallable. Therefore, the DATA_OUT interface must keep its state upon the presence of backpressure coming from the Neander elements.

The backpressure received in the DATA_OUT interface is propagated to the ADDR interface, which becomes unable to receive new valid addresses since the previous address read is still pending by the Neander, as shown in the Figure 21 below.

Figure 21 – Patient memory elastic interfaces



The 1st read is concluded normally, with the ADDR interface receiving a valid data and this generating a valid data on the DATA_OUT elastic interface. When the data from the 2nd valid read address is read to be output, a backpressure is present on the DATA_OUT interface, what causes the memory to keep the output data until it can be consumed by the Neander datapath. This backpressure is propagated to the ADDR interface, which becomes unable to receive addr3 as a valid data. After the backpressure is off, the 2nd read is concluded and the 3rd read is performed, with the acceptance of addr3 as a valid data, followed by the output of data3.

By implementing the elastic protocol, the Patient Memory can have variable latency and maintain the system latency-equivalent behavior. This characteristic is a key advantage when changing the core memory block in technology process exchanges. Hence, an IP with 1-cycle latency can be replaced by a 2-cycle memory Intellectual Property (IP) block without redesign on other parts of the system.

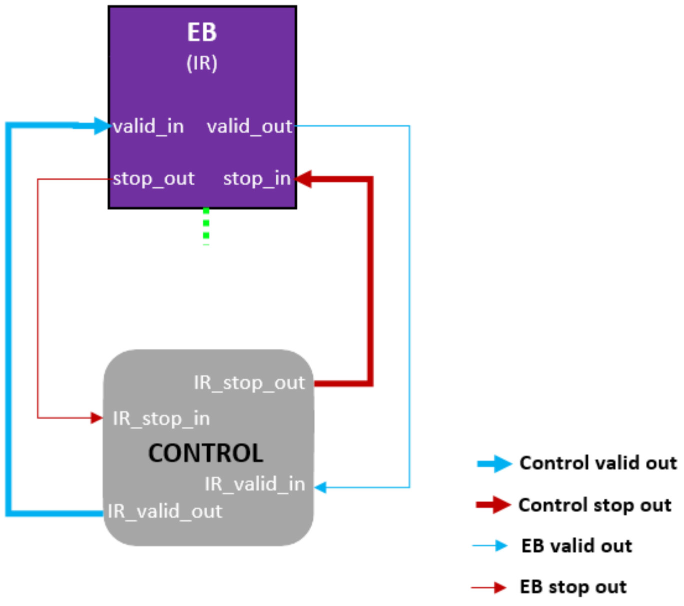
5.8 Patient Control Unit

As the central control unit of the microprocessor design, the Neander Control unit is the responsible for coordinating the data flow among the datapath registers, ensuring that the right data is moved to the right units in the right time to complete an instruction cycle.

Since the data related units implement the elastic protocol in the Elastic Neander, the Control unit needs to be modified to be compatible with the elastic behavior of the data. With the Elastic Buffers replacing the synchronous registers and with the memory implementing the elastic protocol, the Elastic Neander control unit implements an elastic interface with each of the datapath components: AC, IR, MAR, MDR, NZ, PC and MEM. In the Control perspective, the input elastic signals are correspondent to the EB’s outputs and the Control’s output elastic signals correspond to the EB’s inputs, as exemplified in Figure 22 below.

To make the Elastic Neander Control Unit a Patient Process, the FSM has to maintain the current state in case of a stall caused by the elastic protocol. Hence, the transitions between states of the Patient Control Unit (PCTRL) have to be sensitive to the datapath elastic signals relevant to the instruction being executed.

Figure 22 – IR<->Control elastic interfaces

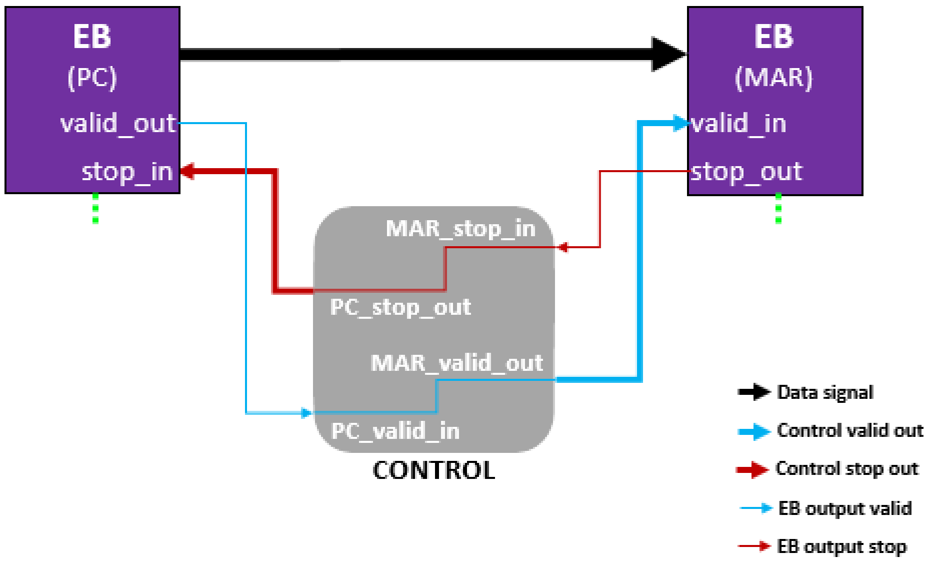


In summary, the Patient Control Unit's FSM was defined based on the following general rules for each state and instruction, considering the previous and the current datapath elements' state:

1- Registers' load input is replaced by the correspondent EB's valid input;
2- Elastic signals from a previous datapath element are connected to the current datapath element;
3- Transitions to a next state wait until the current datapath element is ready to receive data (current state's EB stop = 0);
4- Transitions to a next state wait until the previous datapath element data is ready (previous state's EB valid = 1);

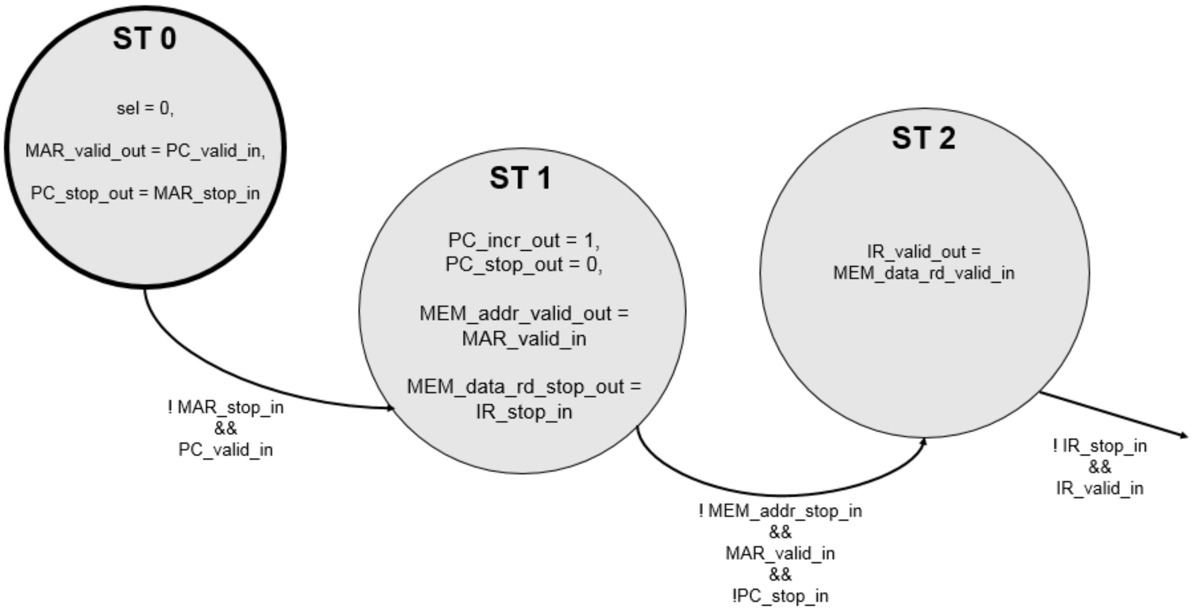
As an example, Figure 23 shows the dataflow in ST0, when MAR receives a valid data as the address of the next position to read from memory. Since the data that the address register stores in ST0 comes from PC, which has been loaded in a previous instruction cycle, the PCTRL connects the elastic control signals of the involved EBs. To keep the program counter's data output unchanged until MAR consumes it, PC's stop is kept asserted until ST0.

Figure 23 – Elastic Neander dataflow in ST0



Since the instruction fetch phase is the same for all the instructions of the Neander architecture, the Figure 24 below shows the transitions of the FSM from ST0 to ST2 in any instruction cycle.

Figure 24 – Instruction fetch phase FSM


ST 0
sel = 0,
MAR_valid_out = PC_valid_in,
PC_stop_out = MAR_stop_in
ST 1
PC_incr_out = 1,
PC_stop_out = 0,
MEM_addr_valid_out = MAR_valid_in
MEM_data_rd_stop_out = IR_stop_in
ST 2
IR_valid_out = MEM_data_rd_valid_in
! MAR_stop_in && PC_valid_in
! MEM_addr_stop_in && MAR_valid_in && !PC_stop_in
! IR_stop_in && IR_valid_in

During ST0, the MAR EB receives a valid data provided by the PC EB (MAR_valid_out = PC_valid_in / sel = 0). Since the data being output by PC is ready to be consumed by MAR, the stop signal of the PC slave interface is deasserted. The transition to from ST0 to ST1 is sensitive to the MAR's stop output and the PC's valid output. By considering these two signals as transition condition, the Control Unit ensures that in ST1 all the data needed to that state's computation is ready.

In ST1, the Control Unit deasserts PC's stop input, since the incremented PC output will be used in a next instruction cycle time. Also in ST1, the MEM address interface receives a valid data, the address of the data to be read from memory. As the IR will receive the data read from memory in ST2, the FSM connects the IR stop output with the MEM data read interface stop input, to avoid reading a data that the IR will not be able to receive. The transition from ST1 to ST2 is conditioned to the state of MEM's address stop output signal, which indicates the hability of the memory to receive a new address and read/write data from/to it. Also, the MAR's valid output and PC's stop input are considered to guarantee that the FSM leaves ST1 with all data needed to the next states ready.

The ST2 is the state where the IR is loaded with the instruction read from the memory, so the IR valid input is connected to the MEM data valid output. Hence, the FSM will only transition from ST2 to ST3 when IR has received and propagated a valid data to its output. The presence of a valid data in IR's output is crucial, since starting from ST3 the FSM is in instruction execution phase.

Starting from ST3, the FSM executes the instruction fetch from memory in the previous phase, from ST0 to ST2. Since the execution phase is dependent on the instruction, from Figure *25* to Figure *33* below shows each FSM's state outputs and transition conditions considering the current instruction.

Figure 25 – FSM outputs and transitions from ST3 - STA, LDA, ADD, OR, AND, JMP, JN (n=1), JZ (z=1)

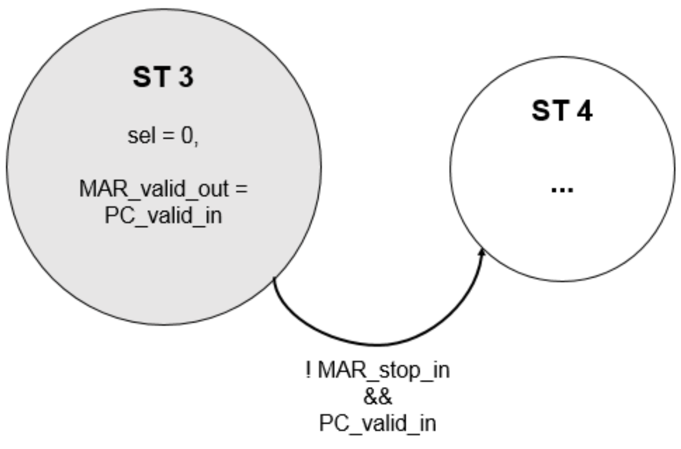


Figure 26 – FSM outputs and transitions from ST3 - JN (n=0), JZ (z=0)

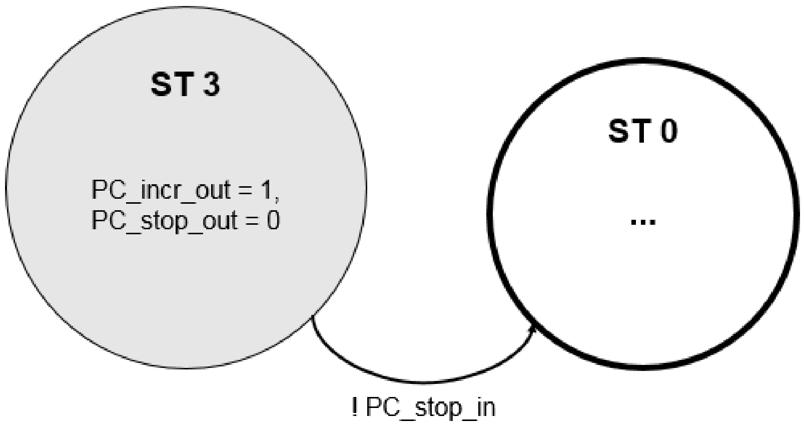


Figure 27 – FSM outputs and transitions from ST3 - NOT

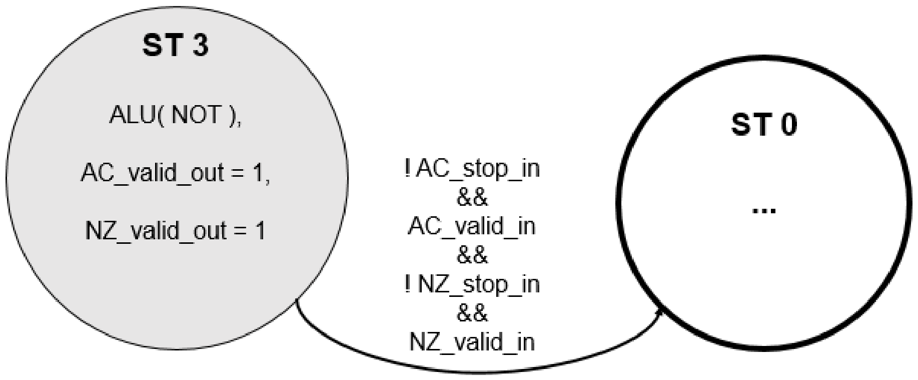


Figure 28 – FSM outputs and transitions from ST4 - STA, LDA, ADD, OR, AND

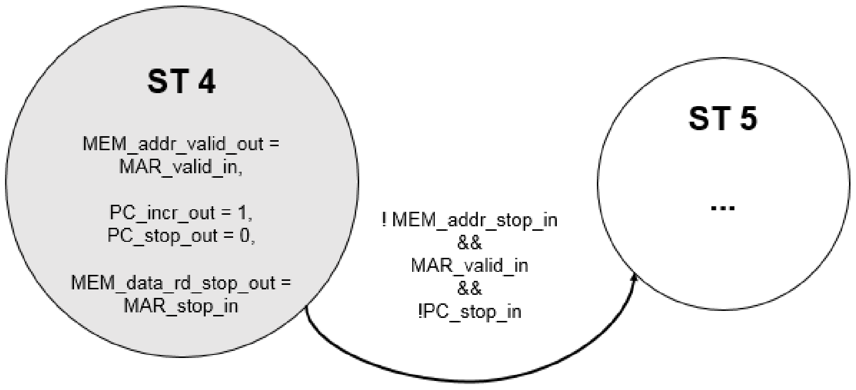


Figure 29 – FSM outputs and transitions from ST4 - JMP, JN (n=1), JZ (z=1)

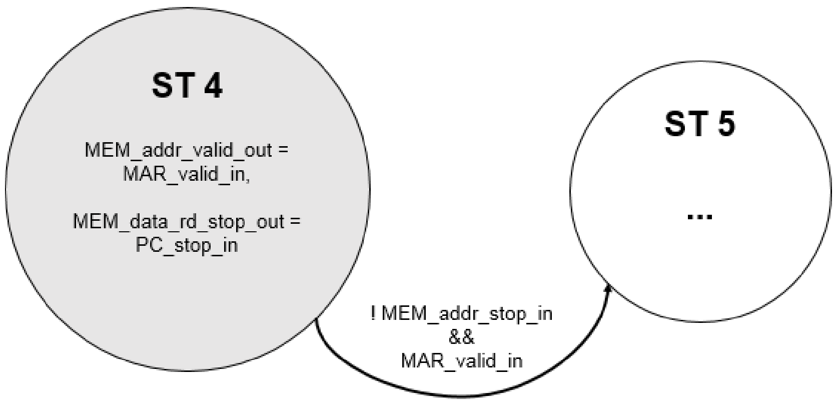

ST 4
MEM_addr_valid_out = MAR_valid_in,
MEM_data_rd_stop_out = PC_stop_in
ST 5
...
! MEM_addr_stop_in && MAR_valid_in

Figure 30 – FSM outputs and transitions from ST5 - STA, LDA, ADD, OR, AND

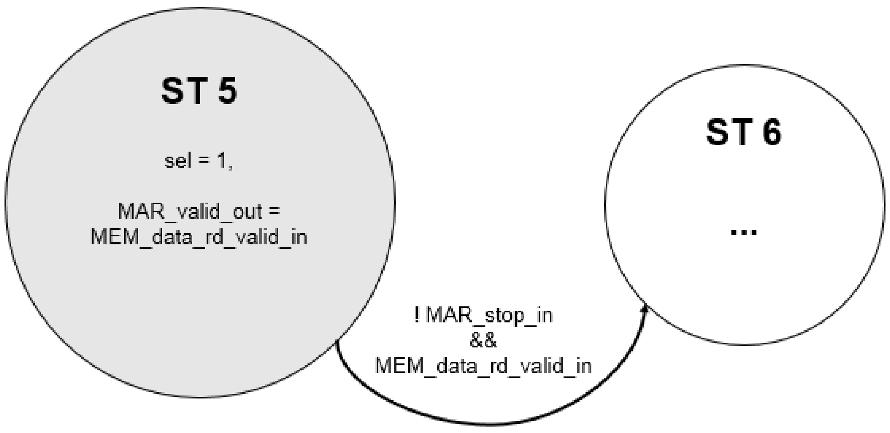

ST 5
sel = 1,
MAR_valid_out = MEM_data_rd_valid_in
ST 6
...
! MAR_stop_in && MEM_data_rd_valid_in

Figure 31 – FSM outputs and transitions from ST5 - JMP, JN (n=1), JZ (z=1)

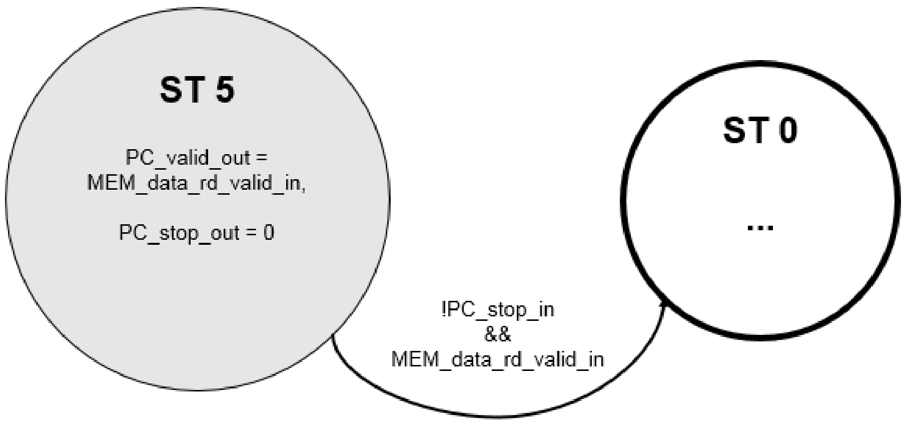

ST 5
PC_valid_out = MEM_data_rd_valid_in,
PC_stop_out = 0
ST 0
...
!PC_stop_in && MEM_data_rd_valid_in

Figure 32 – FSM outputs and transitions from ST6 – STA

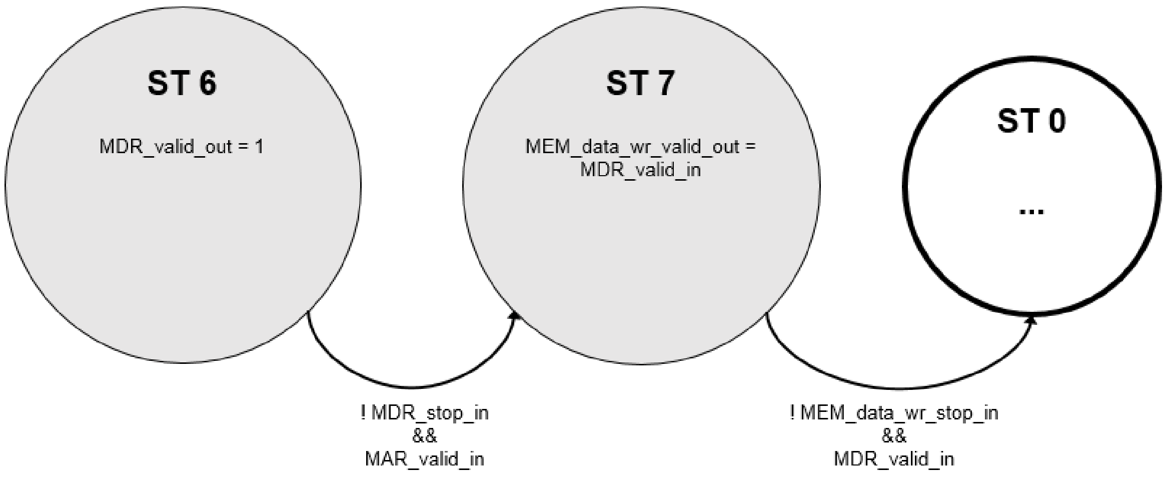
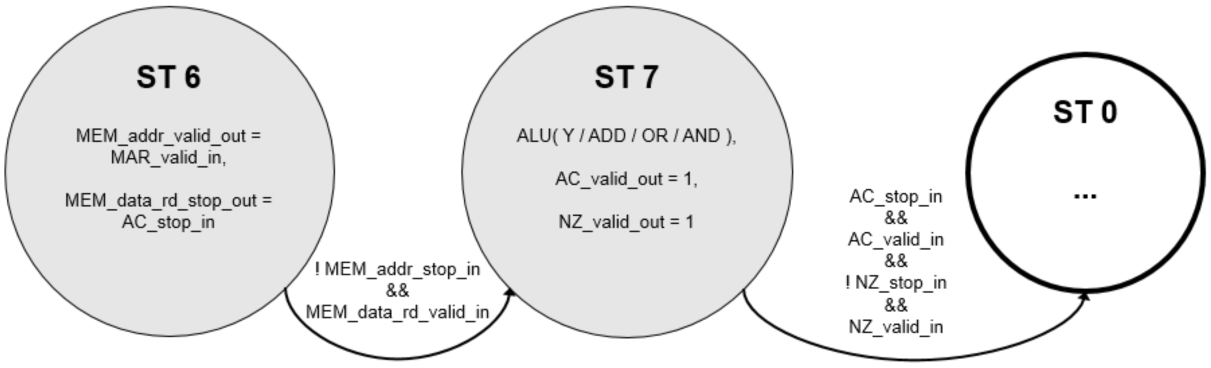

ST 6
MDR_valid_out = 1
ST 7
MEM_data_wr_valid_out = MDR_valid_in
ST 0
...
! MDR_stop_in && MAR_valid_in
! MEM_data_wr_stop_in && MDR_valid_in

Figure 33 – FSM outputs and transitions from ST6 - LDA, ADD, OR, AND


ST 6
MEM_addr_valid_out = MAR_valid_in,
MEM_data_rd_stop_out = AC_stop_in
ST 7
ALU(Y / ADD / OR / AND),
AC_valid_out = 1,
NZ_valid_out = 1
ST 0
...
! MEM_addr_stop_in && MEM_data_rd_valid_in
AC_stop_in && AC_valid_in && ! NZ_stop_in && NZ_valid_in

In the new organization, the FSM can stay in a state for several cycles until the elastic channels involved in the dataflow indicate that all involved datapath elements are ready for the next state. Since the PC value should be incremented only once per Control increment command, the PC_incr_out signal is high during only one clock cycle where PC is able to receive data. Similarly, since the AC input is connected to its output after the ALU operation, the AC_load_out signal is high during only one clock cycle where AC is able to receive data. Otherwise, the PCTRL would corrupt the execution flow by pointing to a wrong memory address or performing multiple operations in the same data instead of only one.

The resulting FSM is robust to the insertion of EBs in any of the Elastic Neander's elastic channels, giving that the elastic signals on the last storage stage in the channel are connected to the PCTRL as the elastic signals for a given channel. As an example, an EB could be inserted between MAR and the MEM and the FSM would still behave correctly if the EB output elastic control signals were connected to the PCTRL instead of MAR’s output elastic control signals.

States ST2, ST6 and ST7 take at least 2 clock cycles, given that the FSM waits until that the data being input in the current datapath element are output by this same element and can be used in the next state. This restriction applies to these states because there is a strong dependence by the FSM on the current states data. For instance, in ST3 the PCTRL uses the IR output data to perform the execution phase of the instruction cycle, hence the instruction needs to be valid in the Control’s input.

5.9 Functional Verification

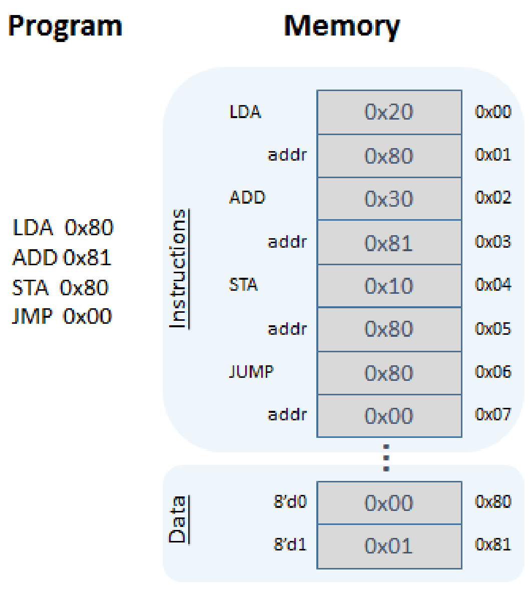
The functional behavior of the Synchronous and Elastic Neander designs was tested using the ModelSim Altera v10.1d simulation tool. A simple test environment – comprised of a Verilog testbench, a simulation tool compilation script and a memory initialization file – was built.

The Neander testbench consists in a stimulus generation entity that exercises the Neander Design Under Test (DUT) by generating clock and reset signals.

Since the Neander execution relies on the data stored in the Memory, the memory initialization file contains the image that represents the test program that the DUT will execute.

The Counter Program shown in Figure 34 was defined to test the Neander designs functionalities. This program reads the memory position 128 (0x80), adds the increment defined in memory position 129 (0x81) and stores the result back in position 128. The JMP instruction makes the program restart its execution after completing the addition cycle. Hence, the Counter Program executes in a loop until there is clock being fed to the DUT.

Figure 34 – Counter Program



Since the memory design target is an Altera Cyclone III FPGA, which has M9K type embedded memory blocks, the simulation libraries for such device components needed to be loaded in the simulation.

The Neander design's memory is implemented using a single port M9K Altera memory block, which has 2 clock cycles of delay between receiving an address and outputting the corresponding data.

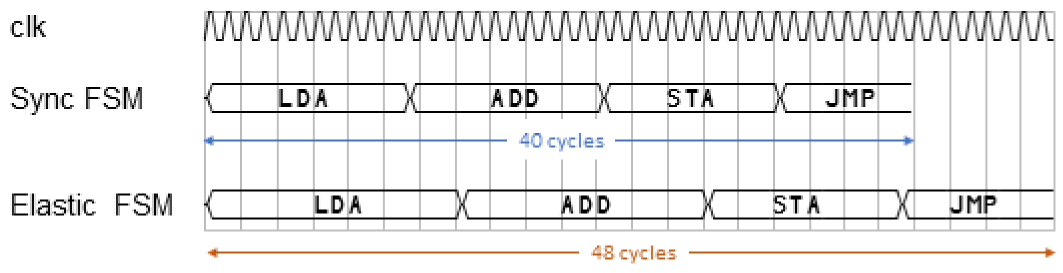
Given the target memory 2 cycles read latency characteristic, the Synchronous Neander design had to be slightly modified to accommodate this behavior. Hence, the FSM needs to wait 2 cycles on memory reading states before continuing execution.

Since the Elastic FSM is designed to tolerate variable latencies, no changes were needed regardless of the memory delay.

After running the simulation, the correctness of the program execution was performed by analyzing the waveform signals, which show that the Neander designs' FSMs behavior is correct according the definition on section 3. Also, the Modelsim memory contents inspection tool shows that the value in the memory position 128 was incremented, as expected.

The latency equivalence principle can be observed when comparing the Synchronous and the Elastic Neander waveforms, as shown in Figure *35* below. Hence, despite the added delay of the Elastic protocol, both Neander designs have the same behavior, confirming that the Elasticization process did not modify the system functionality.

Figure 35 – Latency equivalent behavior of Neander designs



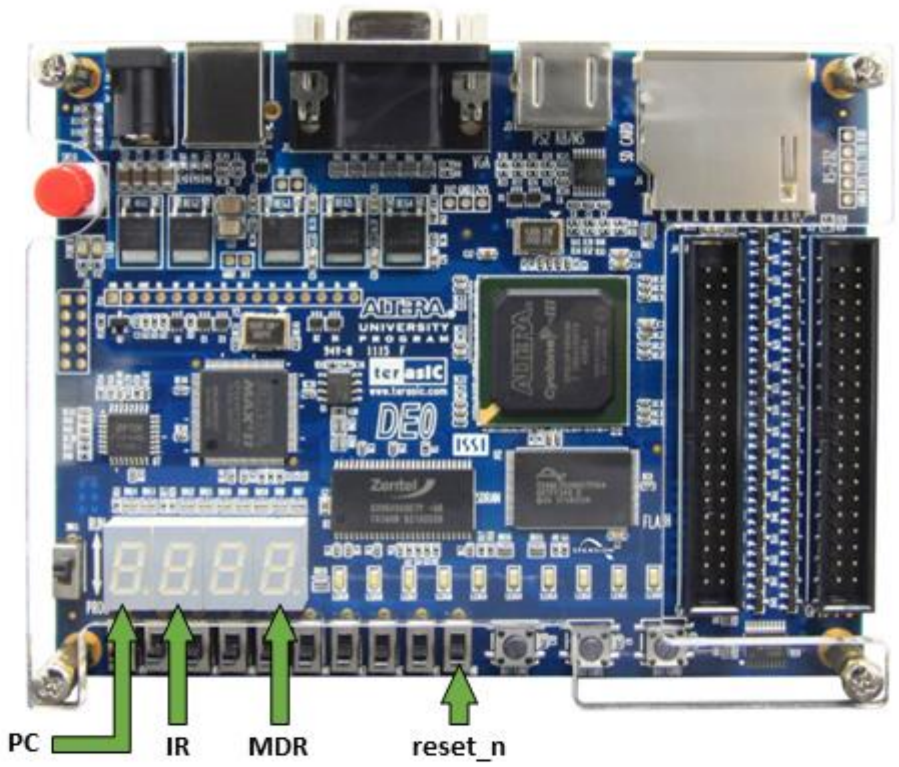
5.10 FPGA demo

The demonstration platform used to validate the designs' behavior, performance and area characteristics is the Terasic DE-0 Cyclone III evaluation board. This platform has the input and output devices necessary to make the design run on a real device. The board features used in the demo are:

- 50 MHz differential clock
- Slide switches
- Seven segment displays
- LEDs

The demonstration purpose is to show the Neander execution flow on the FPGA. Therefore, the seven segment displays were used to present key Neander's organization components data during the Counter Program execution, as shown in the Figure *36* below.

Figure 36 – Neander designs demonstration platform



The PC value displayed is incremented as the Neander goes through the counter loop, while the IR value displays the current instruction being executed. The MDR data shows the current counter value being stored in 128 memory position. All the register values are displayed in hexadecimal base. A simple binary to seven segment converter was implemented to drive the DE0 displays properly.

To make the demonstration observable, a clock divider was implemented to generate a 2Hz clock based on the board's 50MHz clock. This slow clock drives the Neander components, making its results be displayed for at least 0.5 seconds.

5.11 Design Synthesis

Having verified the Neander designs' functionality and defined the demo components, the Quartus II v13.1 tool was used to synthesize the whole design, mapping its modules to the DE0 board components.

The Quartus II Pin Assignment tool was used to map the designs' components inputs and outputs to the Cyclone III FPGA's pins, following the definition for each board's component on the DE0 user manual (TERASIC, 2011).

The synthesis flow configurations used were the tool’s defaults. The defaults include optimizations to the design, such as FSM states auto encoding, register packing and automatic gated clock conversion, which allows the tool to map clock gating logic – part of the Elastic design – to clock enable inputs of the FPGA registers.

A main synthesis directive used is the Timing-Driven Synthesis, which makes the Quartus II tool take design timing constraints into account to generate the most optimized version of the design. This configuration depends on a Synopsys Design Constraint (SDC) file with the proper commands to identify the design’s timing requirements, as shown in Table 4 below.

Table 4 – SDC commands used in Neander designs synthesis

Design Constraint	Description
create_clock	Specify the clock used by the system, in terms of frequency and design’s port assigned as clock input.
derive_pll_clocks	Automatically constrain PLL and other generated clocks.
derive_clock_uncertainty	Automatically calculate clock uncertainty to jitter and other effects.
set_input_delay	Constrain the input I/O path based on the board characteristics.
set_false_path -to [all_outputs]	Ignore the timing of clock to output paths due to low frequency characteristic of the seven segment displays.

The SDC file defined is used by Quartus II on the Analysis & Synthesis to perform a timing driven netlist generation; on the Fitter (Place & Route) to enforce placement and routing that meet the timing constraints; and lastly on the TimeQuest Timing Analysis step, to verify the generated post-synthesis design timing behavior.

5.12 Timing Analysis

The designs' timing analysis was performed using the TimeQuest Timing Analyzer tool of Quartus II. This tool analyzes the timing characteristics of the post-synthesis design using the same SDC as the prior synthesis steps, which defines the system's clock period as 7,5ns.

Using three different operating condition models the TimeQuest performs a multi-corner timing analysis. The operating conditions differ from each other in terms of voltage, process, and temperature, aiding the tool to determine the timing behavior of the design under such conditions.

For the Synchronous and Elastic Neander analysis, the TimeQuest report's main results evaluated were the Setup and Hold slacks and maximum achievable frequency (Fmax) on each operating condition, as shown in Table 5 below.

Table 5 – Neander designs timing results evaluated

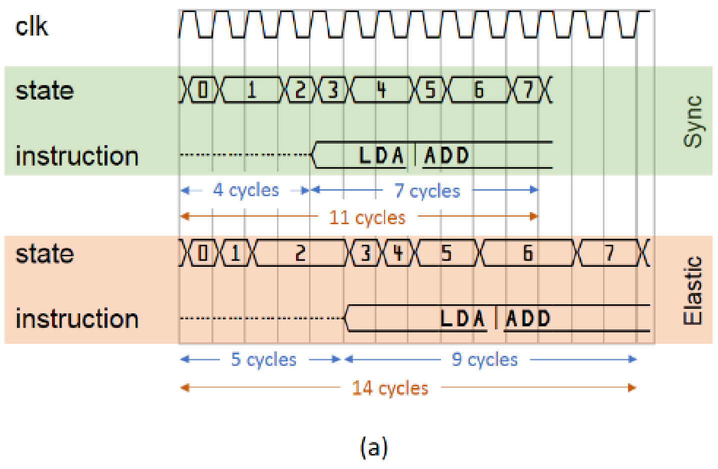
Evaluated result	Operating Condition Model	Synchronous Neander	Elastic Neander
Worst setup slack	Slow 1200mV 85C	1,924 ns	1,048
	Slow 1200mV 0C	2,148 ns	1,657
	Fast 1200mV 0C	2,749 ns	2,506
Worst hold slack	Slow 1200mV 85C	0,280 ns	0,320
	Slow 1200mV 0C	0,281 ns	0,309
	Fast 1200mV 0C	0,134 ns	0,153
Fmax	Slow 1200mV 85C	179,34 MHz	154,99 MHz
	Slow 1200mV 0C	186,85 MHz	171,14 MHz
	Fast 1200mV 0C	210,48 MHz	200,24 MHz

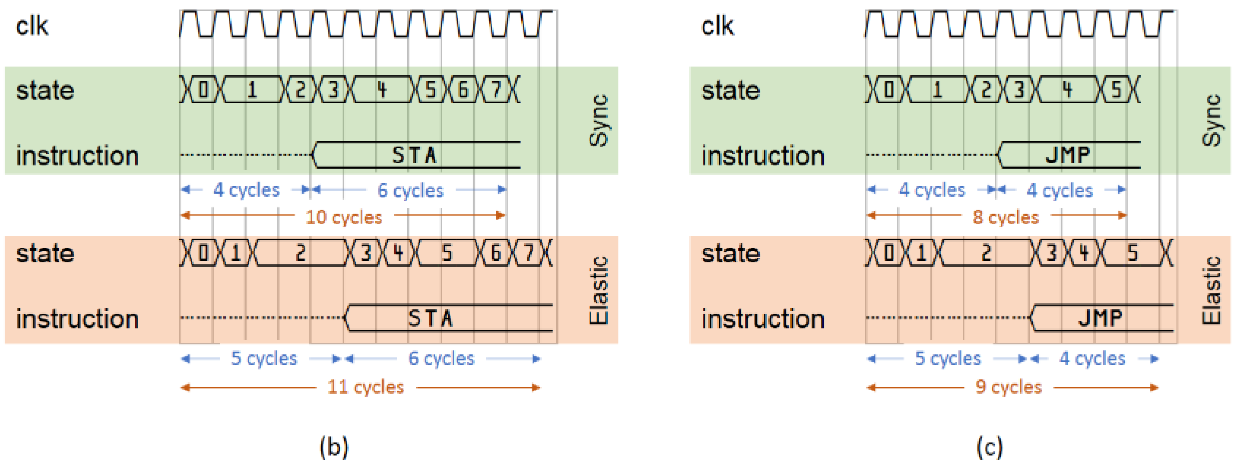
6 RESULTS ANALYSIS

Both Synchronous and Elastic Neander designs have the same functionality, as demonstrated in section 5.9 with the designs functional verification step of the design flow. However, due to the organization differences, each system differs from each other in several aspects, like data computation latency, circuit timing and area.

By making the Elastic Neander's FSM a patient process, sensitive to the elastic protocol, this version of the system has added latency when compared to the synchronous version. As an example, Figure 37 below shows a comparison between Elastic and Synchronous Neander execution time for the LDA or ADD (a), STA (b) and JMP (c) instructions.

Figure 37 - Synchronous vs Elastic instruction execution time





The Elastic FSM takes more clock cycles to execute the Neander architecture instructions when compared to the Synchronous system, as exemplified in the Table 6 below. This added latency can be even higher if the elastic channels get stalled.

Table 6 – Clock cycles per instruction execution

	Execution time per instruction (clock cycles)			
Design	STA	LDA	ADD	JMP
Synchronous	10	11	11	8
Elastic	11	14	14	9

Since a Counter Program can be implemented as a LDA, ADD, STA and JMP instructions sequence loop, the total run time of this program is determined by the sum of each instruction execution time. Hence, the Synchronous organization takes 40 clock cycles to increment the counter by 1, while the Elastic organization takes 48 clock cycles to do the same. Therefore, the added delay of the Elastic organization is 8 clock cycles per unitary increment of the Counter Program.

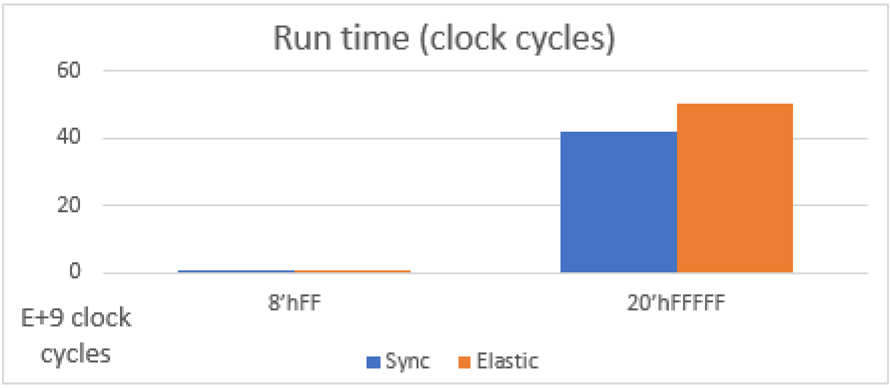
When counting from 0 to 255, the Elastic Neander takes 12.240 clock cycles to complete execution, while the Synchronous Neander takes 10.200 clock cycles. By analyzing the 2.040 clock cycles difference between the organizations, it is noted that on higher run times, the Elastic organization added latency is more evident.

To enable to analysis of the latency difference on higher run times, an auxiliary 20 bits counter was added in the Neander test bench. This extra counter measures the amount of clock cycles that the 8 bits Neander takes to count until 2^20 (20’hFFFFF). The comparison with the counter until 255 (8’hFF) can be seen in the Table 7 and Chart 1 below.

Table 7 – 8 bits counter vs 20 bits counter run time

Stop condition	Design	Clock cycles	Difference
8’hFF	Synchronous	10200	2040
	Elastic	12240	
20’hFFFFF	Synchronous	41943000	8388600
	Elastic	50331600	

Chart 1 – Run time difference increase with higher run times



After going through the synthesis process described in section 5.11, where both Synchronous and Elastic 8bits Neander were subject to the same timing constraints and synthesis tool configuration, the area and timing results are as shown in Table 8 and Table 9 below.

The Area results are specified by the amount of FPGA logic and FFs (Flip Flops) used to implement each design. Due to optimizations done by the synthesis tool, like automatic FSM codification, the resulting number of FFs is not the same as the one originally specified in the design.

Table 8 - 8bits Synchronous and Elastic Neander Area results

	Area	
Design	FPGA %	FFs
Synchronous	0,82	42
Elastic	2	75

As expected, the Elastic Neander area is higher than the Synchronous Neander's. This is explained by the additional storage elements and control logic of the Elastic Buffers that have replaced the simple registers of the Synchronous design.

The Timing results are composed of the worst Setup and Hold slacks of each circuit version, what impacts the maximum achievable frequency (*Fmax*) determined by the tool to each design.

Table 9 - 8bits Synchronous and Elastic Neander Timing results

	Timing		
Design	Setup slack	Hold slack	Fmax (MHz)
Synchronous	1,92 ns	0,28 ns	179,34
Elastic	1,04 ns	0,32 ns	154,99

Since the 8 bits Neander is a simple design, the additional logic of the Elastic version did not represent a performance advantage over the Synchronous system. This is due to the increased placing and routing complexity of the Elastic design, what makes the synthesis tool achieve worse timing results.

Given the *Fmax* of each circuit, the time to count until 20’hFFFFF can be calculated as the multiplication of the clock period by the amount of clock cycles taken to complete the task. Hence, the time taken to run (*Trun*) in ns is shown in Table 10 below.

Table 10 - 8bits Synchronous and Elastic Neander *Trun*(ns) comparison

Stop condition	Design	*Trun* (ns)	Difference (ns)
20’hFFFFF	Synchronous	233.874,15	90.866,73
	Elastic	324.740,88	

In accordance to the performance difference of the two designs, the Elastic Neander presents a higher run time when compared to the Synchronous counterpart. This *Trun* disadvantage reinforces that turning simple synchronous circuits into elastic ones does not necessarily means performance improvements. Hence, the area overhead imposed by the elasticization process becomes does not bring significant advantages in such cases.

However, the elastic behavior of a circuit can be an advantage when critical computation paths start to increase the clock to clock transfers required times, lowering the *Fmax* achievable by a circuit. To demonstrate this property, the Neander datapath was slightly modified to support higher data widths. By making the Memory word size, the AC, the MDR and the ALU data widths configurable, it was possible to create 64 bits and 128 bits versions of the Neander.

The 64 bits Synchronous Neander synthesis process determines that the design's critical paths are the ones going through the ALU, resulting in a *Fmax* of 124,44 MHz – a drop of around 55 MHz compared to the 8bits version. The 64 bits Elastic Neander had a reported *Fmax* of 115.02 MHz – a drop of around 40 MHz. These results are expected, due to the increased logic and arithmetic complexity of 64bits operations when compared to 8bits operations.

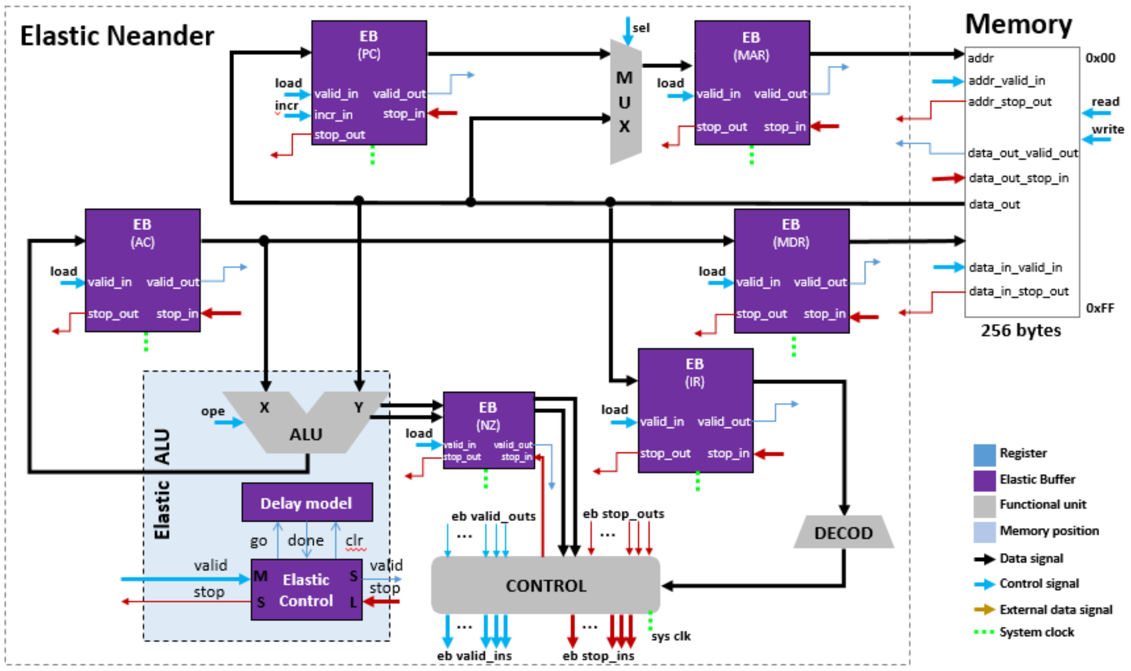
Since the Elastic Neander is tolerant to variable latencies on the computational processes composing the system (CORTADELLA, 2006), the ALU operations in the Elastic version of the design can take more than one clock cycle without impacting the processor's execution flow.

Therefore, the ALU needs to implement the elastic protocol to make its latency accountable by the other patient processes in the system. This is done by mapping the ALU delay behavior with a Delay Model, which influences the variable latency unit – in this case the ALU – elastic control. This control implements the elastic protocol as a result of the Delay Model signals and the elastic signals themselves. The Figure 38 below shows the Elastic ALU in the Elastic Neander organization. The ALU elastic interface follows the same principle of the other elastic interfaces on the design, communicating with the Control Unit.

Based on the *Fmax* drop caused by the ALU combinational logic chains on the 64 bits Neander, the ALU delay was set to 2 clock cycles instead of one. This is done by implementing a Delay Model accordingly, which will set its *done* flag after 2 clock cycles, resulting in the assertion of the elastic valid output. The design's functionality was re-verified and confirmed to be correct, despite the additional 2,1K clock cycles to count until 20'hFFFFF due to the increased delay of the Elastic ALU.

To make the synthesis tool aware of the delay behavior of the Elastic ALU, the timing constraints map this multi-cycle data transfer with a *set_multicycle_path* command. This command sets the tool to consider all transfers going through the ALU to have a 2 clock cycles delay.

Figure 38 - Elastic Neander with Elastic ALU



Having an extra clock cycle to perform the ALU dependent data transfers, the synthesis tool has more flexibility to place and route the design, leading to better performance results, as shown in Table 11 below.

Table 11 - 64bits Neander *Fmax*

Design (64 bits)	ALU delay (clock cycles)	*Fmax* (MHz)
Synchronous	1	124,44
Elastic	1	115,02
Elastic	2	153,92

With an Elastic ALU of 2 cycles delay, the 64 bits Elastic design's *Fmax* is around 23% higher than the Synchronous design's *Fmax.* This operating frequency difference results in similar run times between the two designs, reducing the Synchronous design run time advantage, as shown in Table 12 below.

Table 12 - 64bits Synchronous and Elastic Neander *Trun*(ns) comparison

Stop condition	Design	*Trun* (ns)	Difference (ns)
20’hFFFFF	Synchronous	337.053,92	3.569,38
	Elastic	340.623,30	

This experiment was repeated with 128 bits data width Neander designs, resulting in the achieving of a lower *Trun* by the Elastic design over the Synchronous, as shown in Table 13 below.

Table 13 - 128bits Synchronous and Elastic Neander comparison

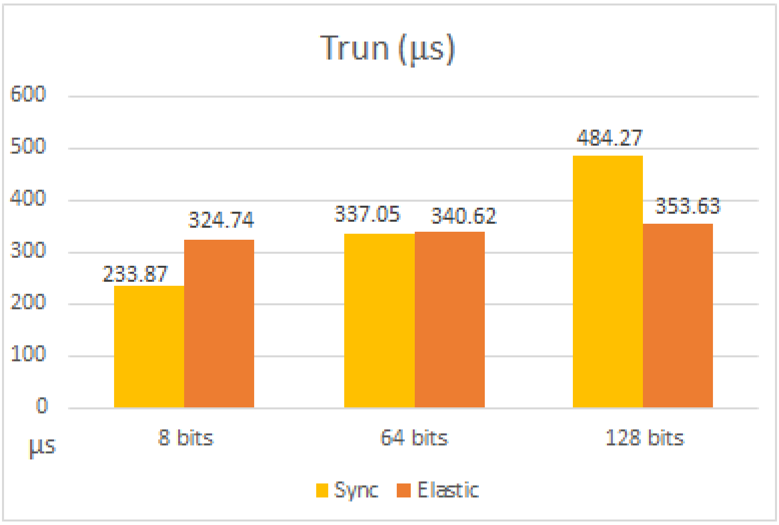
Stop condition	Design (128 bits)	ALU delay (clock cycles)	Fmax (MHz)	Trun (ns)
20’hFFFFF	Synchronous	1	86,61	484,274.22
	Elastic	2	148,26	353,627.00

By analyzing the results of Synchronous and the Elastic Neander designs over the three data width variants, it is clear that as the circuit gets denser and the logic more complex, the elastic behavior brings more performance advantages at the price of increased area, as shown in Chart 2 (a) and (b) below.

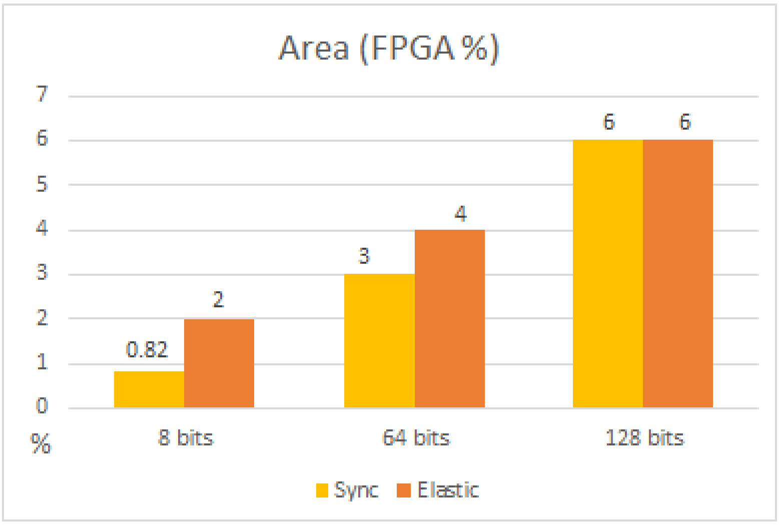
Chart 2 (b) shows that with a 128 bits data width, the FPGA percentage allocation is the same for both designs, despite the use of 282 FFs by the Synchronous design and 317 by the Elastic design. Hence, the combinational units are the main contributors for the use of Logic Elements by the synthesis tool.

The Table 14 below presents the full results for all design variants among Synchronous, Elastic, data widths and ALU delays discussed in this section. A 128 bits Elastic Neander variant with ALU delay of 3 clock cycles is part of the results, demonstrating an even higher performance advantage for Elastic system at the price of one additional FF.

Chart 2 – Results overview



(a)



(b)

Table 14 – Full Synchronous and Elastic Neander designs results

Data width	**Design**	**ALU delay** (clock cycles)	**FPGA %**	**FFs**	**F Max** (MHz)	**Stop condition**	**Clock cycles**	**T run** (ns)
8 bits	Sync	1	0,82	42	179,34	20'hFFFFF	41.942.990	233.874,15
	Elastic	1	2	75	154,99	20'hFFFFF	50.331.589	324.740,88
64 bits	Sync	1	3	154	124,44	20'hFFFFF	41.942.990	337.053,92
	Elastic	1	4	188	115,02	20'hFFFFF	50.331.589	437.589,89
		2	4	189	153,92	20'hFFFFF	52.428.739	340.623,30

Data width	**Design**	**ALU delay** (clock cycles)	**FPGA %**	**FFs**	**F Max** (MHz)	**Stop condition**	**Clock cycles**	**T run** (ns)
128 bits	Sync	1	6	282	86,61	20'hFFFFF	41.942.990	484.274,22
	Elastic	2	6	317	148,26	20'hFFFFF	52.428.739	353.627,00
		3	6	318	161,21	20'hFFFFF	54.525.889	338.228,95

7 CONCLUSIONS AND FUTURE WORK

As an alternative towards solving the challenges imposed by the increased wire-latency ratio in recent integrated circuit production technologies, the Elastic Circuits paradigm enables complex designs to become more robust to such impacts and even offers performance advantages over traditional synchronous systems, at the cost of area overhead. However, this paradigm maintains the designers' ability to use synchronous EDA tools and flow instead of imposing drastic infrastructure changes like the asynchronous methodology does.

A review of the state of the art literature on Latency Insensitive and Elastic systems highlights the principles of this type of design methodology and brings the key factors that must be taken into account when planning to apply the Elastic Design flow into a Synchronous Design. One of those factors is the ability of the original system's Computational Processes to be stalled, a major requirement to turn them into Patient Processes that are able to retain its state while other processes are not able to receive new valid data.

The simple multi-cycle 8 bits Neander processor architecture was explored as an experimental platform to exercise the principles of the Elastic design, since it has a well-known behavior, easily implemented and verified. A synchronous version of Neander organization was developed to enable the application of the elasticization process and evaluation of such procedure's effects in terms of timing and area.

By using the same synchronous design tools, an Elastic version of the Neander was implemented as a variant of the original organization. The Elastic Neander functional verification proves that both designs are latency equivalent, meaning that both versions have the same functionality despite the latencies added by the elasticization process.

The synthesis process of both designs demonstrates that, as expected, the Elastic system's area is higher than the Synchronous version, due to the replace of single storage elements by Elastic Buffers, which are dual storage elements capable of retaining data in the presence of backpressures.

Being a simple design, the 8 bits Neander does not have critical computation paths in the technology used, what results in timing slack loss by the Elastic system due to the extra logic added to control the elastic protocol.

However, on increased complexity designs like 64 and 128 bits Neander variations, the few combinational elements of the design start to suffer timing closure problems. In such expanded width datapaths, the elastic behavior enables functionally correct transformations that make the Elastic design’s timing constraints more flexible, resulting in overall performance advantages of around 30% over the Synchronous system.

The design flow is closed with the deployment of both designs in a real FPGA evaluation platform, making use of the board components to validate that the Elastic transformations preserve the behavior of the original Synchronous Neander.

Besides the area and timing aspects evaluated, the Elastic Circuits tend to offer power advantages over the Synchronous systems due to the fact that the elastic computation units and its channels are sensitive to the presence of valid data, enabling power saving when the system is idle. This analysis is open to be explored in future works.

With demonstrated timing advantages over the traditional Synchronous designs, the Elastic Circuits paradigm also offers good flexibility in its components, which could lead to even higher performance gain. Simple modifications to the presented design, like different Elastic Buffer organizations, and bigger architectural changes like Recycling, Early Evaluation and Speculative Execution are possibilities that can be explored to tune the Elastic Circuits results, as future work that can be developed in even higher complexity systems.

REFERENCES

[1] Bohr M. T. Silicon trends and limits for advanced microprocessors. **Commun. ACM**, v. 41, n. 3, p. 80–87, Mar. 1998.

[2] Flynn M. J.; Hung P.; Rudd K. W. Deep-submicron microprocessor design issues. **IEEE Micro**, v. 19, p. 11–13, July 1999.

[3] Carloni L.; McMillan K.L.; Sangiovanni-Vincentelli A.L. Theory of latency-insensitive design. **IEEE Transactions on Computer-Aided Design**, v. 20, n. 9, p. 1059–1076, September 2001.

[4] Carloni L.; McMillan K.L.; Sangiovanni-Vincentelli A.L. Latency insensitive protocols. **Proceedings of the 11th International Conference on Computer-Aided Verification**, New York: Springer-Verlag, 1999.

[5] Cortadella J.; Kishinevsky M.; Grundmann B. SELF: Specification and design of a synchronous elastic architecture for DSM systems. **TAU-2006: International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems**, 2006. Available: http://www.cs.upc.edu/~jordicf/gavina/BIB/files/self_tau06.pdf. Accessed on: 20 Jun. 2016.

[6] Cortadella J.; Galceran-Oms M.; Kishinevsky M. Elastic systems. **Proc. 8th ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign (MEMOCODE 2010)**, p. 149-158, July 2010. Available: http://www.cs.upc.edu/~jordicf/gavina/BIB/files/memocode2010.pdf. Accessed on: 20 Jun. 2016.

[7] Carloni L.; Sangiovanni-Vincentelli A. Coping with latency in SoC design. **IEEE Micro, Special Issue on Systems on Chip**, v. 22, n. 5, p. 12, October 2002.

[8] Jacobson H. M.; Kudva P. N.; Bose P.; Cook P. W.; Schuster S. E.; Mercer E. G.; Myers C. J. Synchronous interlocked pipelines. **Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems**, pp. 3–12, Apr. 2002.

[9] Carmona J.; Cortadella J.; Kishinevsky M.; Taubin A. Elastic circuits. **IEEE Transactions on Computer-Aided Design**, v. 28 n.10, p. 1437-1455, October 2009.

[10] Weber R. **Fundamentos de Arquitetura de Computadores**. 4th ed. Porto Alegre: BOOKMAN, 2012.

[11] Terasic. DE0 User Manual. March 2011. Available: http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=165&No=364&PartNo=4. Accessed on: 28 Oct. 2016.

[12] Cortadella J.; Kishinevsky M.; Grundmann B. **Synthesis of Synchronous Elastic Architectures**. Design Automation Conference, 2006 43rd ACM/IEEE.