Four stage pipeline quaternary processor

Procesador cuaternario pipeline cuatro etapas

Tiago Gouveia¹ Wellington Oliveira²
Milton Ernesto Romero Romero^{2*} Evandro Mazina Martins¹

Recibido 04 de junio de 2018, aceptado 25 de marzo de 2019

Received: Junio 04, 2018 Accepted: March 25, 2019

ABSTRACT

The scale of integration of processors has increased in recent decades, new challenges have emerged and chip area has become an important issue. Designers have been motivated to seek new techniques and technologies, among them, the multi-value logic (MVL). The quaternary representation, domain D: {0, 1, 2, 3} reduces the number of connections due to the fact that, approximately, 70% of the circuit area is being used for interconnections and pads. This work proposes the design of a four stages pipelined quaternary processor (eCPU) with sixteen instructions and the handling of hazards utilizing hybrid (static and dynamic) techniques with the scope to demonstrate the correct functionality with respect to the design specification, based on a universal set of quaternary logic gates already proposed in the literature. The eCPU has been designed via hardware description in Quartus® environment written in VHSIC Hardware Description Language (VHDL) and simulations performed in ModelSim®, demonstrating the correct behavior with respect to the specifications. The simulations are performed by executing several programs written in the chosen quaternary assembly language with the support of a two phase's compiler written in Java to generate quaternary machine code.

Keywords: Control processing unit, multi valued-logic, quaternary compiler.

RESUMEN

La escala de integración de los procesadores se ha incrementado en las últimas décadas, han surgido nuevos desafíos y el área de chips se ha convertido en un tema importante. Esto ha motivado a los proyectistas a buscar nuevas técnicas y tecnologías, entre ellas, la lógica de múltiples valores (MVL). La representación cuaternaria, dominio D: {0, 1, 2, 3}, reduce el número de conexiones debido al hecho de que, aproximadamente, el 70% del área del circuito se utiliza para interconexiones y pads. Este trabajo propone el proyecto de un procesador cuaternario de cuatro etapas (eCPU) con dieciséis instrucciones y el manejo de conflictos de hardware, de datos y de control utilizando técnicas híbridas (estáticas y dinámicas), para demostrar la funcionalidad correcta con respecto a la especificación del proyecto, con base en un conjunto universal de compuertas lógicas cuaternarias ya propuestas en la literatura. El eCPU se ha proyectado utilizando el lenguaje de descripción de hardware VHDL en el ambiente Quartus® y simulado en ModelSim® que demuestra el comportamiento correcto con respecto a la especificación. Las simulaciones se realizan ejecutando varios programas escritos en el lenguaje de la máquina con el soporte de un compilador de dos fases escrito en Java para generar código de máquina cuaternario.

Palabras clave: Unidad de procesamiento de control, lógica de múltiples valores, compilador cuaternario.

¹ Universidade Federal de Mato Grosso do Sul. Faculdade de Engenharia. Campo Grande, Brasil.
E-mail: tiago-lucas-gouveia@hotmail.com; evandro.mazina@ufms.br

² Universidade Federal de Mato Grosso do Sul. Faculdade de Computação. Campo Grande, Brasil.
E-mail: well.oliveira.snt@gmail.com; romero@facom.ufms.br

* Corresponding Author: romero@facom.ufms.br

INTRODUCTION

The increased integration of integrated circuits brings new challenges and chip area has become an important issue [1]. An alternative to decrease chip area, of binary digital circuits design, is to increase the digital representation to the Multiple-Valued Logic (MVL) base L , with domain $D: \{0, 1, 2, \dots, L-1\}$ in order to increase the entropy for each connection. MVL was introduced first by Lukasiewicz and Post for ternary logic [2-4]. The main advantage of the MVL digital processing is the reduced chip area, due to the fact that the interconnections contribute to about 70% of the Integrated Circuit (IC) area [13]. The IC area decreases by means of fewer interconnections, less quantity of IC pins and pads. Additionally, the processing performance might be faster for the MVL circuits in comparison to the binary circuits for the same resolution [4-5]. To design an actual quaternary CPU (eCPU), a quaternary algebraic system is needed. There are many proposed MVL algebras [6-8] and a suitable algebra needs to define a universal set of operators which must permit: 1) the development of minimization tools; 2) the operators' implementation with the IC gates; 3) the low complexity learning curve to facilitate designer's work. A suitable quaternary algebra that copes with all above-mentioned characteristics has been presented in [9] and it is comprised of Extended products, Successor and Maximum operators. Therefore, the aim of this paper is to design sixteen instructions, four stages pipelined quaternary processor architecture, that allow to demonstrate correct functionality with respect to the specifications in which the design criteria is the simplicity of implementation, not taking into consideration any possible optimization, nor specific criteria to define each stage pipeline functionality.

The methodology utilized here to the eCPU design is performed by extending the binary logic CPU architecture which facilitates the quaternary design. The eCPU follows the Harvard architectural model, a stack segment and handles hazards with hybrid techniques. Most data hazards are managed with static techniques through the utilization of a Java compiler which translates eCPU assembly to quaternary machine code, in which RAW and WAW data hazards are handled along with dynamic technique as forwarding between Instruction Execution and Instruction Decode. The control hazards are not

fully handled, however, an interrupt mechanism is built into the eCPU for future implementation of complete control hazards. The required control signals are implemented within each pipeline stages. The description of the processor is made using the hardware description language VHDL and Quartus II development environment version 13.1 of Altera®. To validate the design, simulations are performed in Modelsim® version 10.1. Tests programs verify each instruction execution in the four stages pipeline and the whole eCPU execution is illustrated by means of calculating the ten first terms of the Fibonacci sequence in quaternary logic: {000, 001, 001, 002, 003, 011, 020, 031, 111, 202...}.

The rest of this paper is organized as follows: The second section presents the related work; The third section addresses the basic concepts; The fourth section describes the eCPU architecture; The fifth section presents the simulation utilizing the chosen technology; The sixth section discusses the limitations and the advantages of the design; The seventh section summarizes the concluding remarks and future work.

MVL RELATED WORK

There exist in the literature circuits to processing digital signals and different architectures that use MVL. In [4] the situation and the future of multiple-valued logic are analyzed and [10-12] discuss a ternary calculating machine, ternary computer, and the ternary optical computer architecture, respectively. There are two works exploiting a pipelined quaternary processor for image processing: 1) in [13] the authors claim that using the MVL logic in conventional digital systems has potential advantages of which one of the most important is the reduction of interconnections; 2) in [14] the authors discuss an NMOS pipelined image processor using quaternary logic for applications in medical image processing and robotics; Adders are a fundamental circuit for the MVL CPU design and are reviewed in [15] by proposing quaternary adders in voltage mode and in [16] demonstrating the design of High-Performance Quaternary Adders. In [17] a high-speed, low-power and compact processing element (PE) using quaternary differential logic is proposed for a multi-core single-instruction-multiple-data (SIMD) processor implemented by using the multiple-valued current mode differential logic circuitry. In [18]

the authors discuss content addressable memory in order to improve network quality of service (QoS) proposing a ternary/quaternary CAM architecture using DRAM technology processing.

Of particular interests in this work is the universal set of operators for the quaternary logic utilized for the design of MVL CPU architectures in [9], which takes advantage of the binary world knowledge, including minimization tools, synthesis methodology and implemented in a CMOS integrated circuit, demonstrating the feasibility of the hardware implementation [19].

MVL BASIC CONCEPTS

Basic concepts are related to the universal set of quaternary gates proposed in [19] that permit to design any quaternary circuit taking advantage of the binary world knowledge. To design sequential circuits a memory element is also needed, as the D flip-flop proposed later. The architectural concepts needed to design the quaternary eCPU are similar to the binary world, being necessary only the utilization of the quaternary instead of the binary algebra, therefore decreasing the learning curve of the eCPU project.

To properly perform arithmetic operations, it was necessary to define the equivalent to the bit in the binary world that is the qbit or quaternary digit $\{0, 1, 2, 3\}$, and the operations are implemented based on the MVL algebra acting on the qbit, as presented next.

MVL universal set of quaternary operators

The chosen universal set of quaternary gates consists of the following operators: Extended products $eAND_1, eAND_2, eAND_3$, Successor, and Maximum with gates representation, as shown in Figure 1, and replicated briefly next, for clarity purposes [9].

Extended MVL product term, shown in Table 1 is a binary operator, denoted by the symbol $*^i$, $eAND_i(a_1, a_2) = a_1 *^i a_2 = i$, iff $a_1 = a_2 = i$; otherwise $a_1 *^i a_2 = 0$. For instance, if $a_1 = 2, a_2 = 0$, then $a_1 *^1 a_2 = 0$; if $a_1 = 1, a_2 = 1$, then $a_1 *^1 a_2 = 1$.

Successor (a_1) is a unary operator, denoted by the symbol a_1^1 , $a_1^1 = (a_1 + 1) \bmod 4$, where in this case + stands for the arithmetic addition and mod stands for

Figure 1. MVL gates representation.

the modulo operator. The a_1^1 is the next element in the ordered cyclic domain $D: (0, 1, 2, 3)$ [10] when the Successor is applied once to (a_1). The a_1^2 is the second next element in the ordered cyclic, etc., for instance, if ($a_1 = 2$) $a_1^1 = 3$, $a_1^2 = 0$, $a_1^3 = 1$.

Maximum (a_1, a_2) is a binary operator, denoted by the symbol $+$, defined, as usual, being the larger of the two values, for instance: if $a_1 = 2, a_2 = 0$, then $a_1 + a_2 = 2$. From now on $+$ always stands for Maximum (not arithmetic addition) operator, unless stated otherwise.

By utilizing these operators any quaternary circuit can be designed based on the concept of MVL functions: $F(a_1, a_2, \dots, a_n)$, where n is the number of quaternary variables. In the chosen quaternary algebra, functions are defined as composed of subfunctions: $F_1(a_1, a_2, \dots, a_n), F_2(a_1, a_2, \dots, a_n), F_3(a_1, a_2, \dots, a_n)$, i.e., given a quaternary function each $F_i(a_1, a_2, \dots, a_n)$, $i = 1, 2, 3$ copes with the minterms equal to i and takes two possible values i or 0 . An MVL minterm is a product term ($eAND_i$) that has all the variables presented in any form of the Successor operator, analogously to the binary minterm.

The sum of extended products form (SOPE) is a sum (maximum operator) of MVL subfunctions. This process yields the function in equation (1) in the sum of extended product form (SOPE) [9] for the example shown in Table 1, analogously to the sum of product form (SOP) in binary circuits' synthesis.

Analogously to the binary base, there is another universal set of MVL operator comprised of Extended OR operator, Successor, and Minimum. Extended OR operator is a binary operator for $i = 0, 1, 2$, denoted by the symbol $+^i$, $eOR_i(a_1, a_2) = a_1 +^i a_2 = i$, iff $a_1 = a_2 = i$; otherwise $a_1 +^i a_2 = 3$. For instance, if $a_1 = 2, a_2 = 3$, then $a_1 +^1 a_2 = 3$; if $a_1 = 1, a_2 = 1$, then $a_1 +^1 a_2 = 1$.

The Minimum is a binary operator, as usual, being the minimum between the two operands.

Table 1. $eAND_1: a_1 *^1 a_2$; $eAND_2: a_1 *^2 a_2$; $eAND_3: a_1 *^3 a_2$; example: $F(a_1, a_2)$.

$a_1 *^1 a_2$					$a_1 *^2 a_2$					$a_1 *^3 a_2$					$a_1 *^3 a_2$				
a_1/a_2	0	1	2	3	a_1/a_2	0	1	2	3	a_1/a_2	0	1	2	3	a_1/a_2	0	1	2	3
0	0	0	0	0	0	0	0	0	0	ρ	0	0	0	0	0	3	0	0	0
1	0	1	0	0	1	0	1	0	0	1	0	1	0	0	1	0	1	3	0
2	0	0	0	0	2	0	0	2	0	2	0	0	0	0	2	0	3	0	0
3	0	0	0	0	3	0	0	0	0	3	0	0	0	3	3	2	0	2	0

$$\begin{aligned}
 F(a_1, a_2) &= F_1(a_1, a_2) + F_2(a_1, a_2) + F_3(a_1, a_2) \\
 F_1(a_1, a_2) &= a_1 *^1 a_2 \\
 F_2(a_1, a_2) &= a_1^3 *^2 a_2 \\
 F_3(a_1, a_2) &= a_1^3 *^3 a_2^3 + a_1^2 *^3 a_2^1 + a_1^1 *^3 a_2^2
 \end{aligned} \tag{1}$$

This is the duality by complementation, named the product of extended sums form (POES), which is a product (minimum operator) of MVL subfunctions, based on maxterms. An MVL maxterm is a sum term ($eOR_i(a_1, a_2)$), that has all variables presented in any form of the Successor operator. For a detailed explanation, please refer to [9].

To define the MVL delay latch: D-latch, there are many alternatives, but in the same way as in the binary methodology, in the MVL D-latch independently of the current state Q , the next state Q^* is set to the input data D after a delay. The State equation Q^* for the quaternary D-latch is shown in equation (2). For the interested reader, definitions of types of quaternary latches are found in [9, 20]. With the two sets of universal operators along with the memory element, it is possible to design the proposed eCPU as presented next.

$$Q^* = D *^1 D + D *^2 D + D *^3 D \tag{2}$$

The design criteria for the eCPU architecture to prove concepts are based on the simplicity of the implementation. The four stages pipeline eCPU executes sixteen instructions chosen to demonstrate the feasibility of the architecture, letting write any program in the eCPU assembly language working in the domain of natural numbers for arithmetic operations. The eCPU memory consists of Harvard architecture, a stack segment, and the Register File. In the Harvard memory, the instruction and data modules are addressed by qbytes, defined as a set of four qbits, and each unit has a capacity of two hundred fifty-six words. The Register File contains a bank of sixteen registers for general use.

Table 2 shows the three types of implemented instructions its syntax and semantics: 1) type R: ADD, SUBT, MULT, DIV, PUSH, POP, GT, LS, MOV; 2) type I: ADDI (immediate addition), LOAD, STORE, NOP; and 3) type J: JUMP. The instruction's format has eight qbits size. Table 3 shows the OpCode and encoding, where tt, ss, dd are the source registers, destination, and CCCC immediate value, respectively.

The instruction set type R presents the first two qbits as OpCode, the next four qbits as source registers address and the last two, the destination register. Type R has one or three registers as operands. The

Table 2. Instructions types.

Instruction	Type	Syntax	Semantics	Instruction	Type	Syntax	Semantics
ADD	R	ndd \$d, \$t, \$s	$\$d = \$t + \$s$	PUSH	R	push \$s	Memory[stack pointer]
SUBT	R	subt \$d, \$t, \$s	$\$d = \$t - \$s$	POP	R	pop \$d	$\$d = \text{Memory}[\text{stack pointer}]$
MULT	R	mult \$d, \$t, \$s	$\$d = \$t * \$s$	ADDI	I	addi \$d, C	$\$d = \$d + \$s$
DIV	R	div \$d, \$t, \$s	$\$d = \$t / \$s$	LOAD	I	load \$d, C	$\$d = \text{Memory}[C]$
GT	R	gt \$d, \$t, \$s	$\$d = (\$t < \$s)$	STORE	I	store \$s, C	Memory[C] = \$s
LS	R	ls \$d, \$t, \$s	$\$d = (\$t > \$s)$	NOP	I	nop	No Operation
MOV	R	mov \$d, \$s	$\$d = \t	JUMP	J	jump \$s, C	If (\$c = 0) then PC = 0

Table 3. Instructions encoding.

Instruction	OpCode	Encoding (2) (2) (2) (2)				Instruction	OpCode	Encoding (2) (2) (2) (2)		
ADD	10	OP	tt	ss	dd	PUSH	31	OP	ss	0000
SUBT	11	OP	tt	ss	dd	POP	32	OP	dd	0000
MULT	12	OP	tt	ss	dd	ADDI	20	OP	dd	CCCC
DIV	21	OP	tt	ss	dd	LOAD	23	OP	dd	CCCC
GT	22	OP	tt	ss	dd	STORE	33	OP	ss	CCCC
LS	30	OP	tt	ss	dd	NOP	00	OP	00	0000
MOV	13	OP	00	ss	dd	JUMP	01	OP	ss	CCCC

instruction set type I presents the first two qbits as OpCode, the next two qbits as the destination or source register address and the last four an immediate value (immediate) or an address (LOAD and STORE). Type I has varying numbers of operands, from zero (NOP) to two, using the immediate value.

In the instruction set, type J is the conditional branch which presents the first two qbits as OpCode, the next two qbits is an address to check the condition and the last qbits is the address to jump.

The eCPU modules are detailed as follows:

- 1) Instruction Fetch Unit: in Figure 2 three functionalities are implemented in this module: a) the usual Instruction Fetch handled via the PC block that chooses between the next instruction address (PC+1) or the jump instruction address (PCin); b) the managing of the stack segment via the Stack Pointer Block which inputs to the unified two segments memory, one for the instructions and one for the stack; and c) the managing of the interrupts via the Interrupt Unit block. Table 4 defines its input and output signals. Note that

the unified memory (Instruction and Stack) is a two-port memory placed in this module for ease of implementation. PC points to the address to the next instruction in the instruction memory to fetch the instruction. With the PCSrc signals (source PC) and PCin (PC input), both generated in the Instruction Execution Unit, the processor determines whether follows its normal cycle,

Figure 2. Instruction fetch.

Table 4. Instruction fetch signals.

Signal	Type	Description	Signal	Type	Description
address_Out	out	Address of the current instruction	postFlush	out	It flushes the stack
clk	in	Clock	program_size		Number of instructions
context_switch	in	Enable context switching	rst	in	Reset
instruction	out	Instruction to be executed	stack_in	in	Data to be pushed in the stack
PCIn / jump_address	in	The address specified in the jump instruction	stack_op / op	in	The operation to be performed
PCOut / Address_in	out/in	Pointer to the next instruction	Stack_Out	out	Data output indicated by the stack pointer
PCSr / int	in	It indicates whether or not occurs jump	StackOverflow	out	Stack overflow exception
PointerOut / stack_pointer		Pointer to the stack segment			

that is PC+1, or if a jump instruction occurs. The Stack Pointer Block handles the POP and PUSH instructions execution in the stack that is the basic mechanism to cope with the control hazards.

- 2) Instruction Decode: in Figure 3, two functionalities are implemented in this module: a) the usual decoding of the instruction and data fetch from the register file REG FILE block; b) the managing of the forwarding implementation via FORWARDING CONTROL block. Table 5 defines its input and output signals.
- 3) Instruction Execution: in Figure 4 three functionalities are implemented in this module: a) the execution of the instructions via the ALU; b) the managing of the forwarding implementation via the MUX blocks; and c) managing of the interrupt implementation via the INTERRUPT TRIGGER block.

Table 6 defines its input and output signals. Within this stage, the ALU (arithmetic and logic unit) is placed. In order to implement quaternary logic operations qbit and qbit vector types are defined

Figure 3. Instruction decode.

Table 5. Instruction decode signals.

Signal	Type	Description	Signal	Type	Description
Address_in / Address_out	in / out	The address of the current instruction	MemWrite	out	Enable writing to memory the coded instruction
ALUOp	out	The operation to be executed	postFlush	in	It flushes the stack
ArithmeticOp	out	Enables ALU operation	RegData1	out	Source operand one
Branch	out	Check whether the jump is taken or not	RegData2	out	Source operand two
clk	in	Clock	RegDest	out	Destination register of the encoded instruction
forwarding_data	out	Data to be forwarded to the memory	RegLastCallOut	out	Controls last register retrieve
forwarding_data_enable	out	Sets the forwarding operation from registers to memory	RegWrite	out	Enables writing the destination register of the encoded instruction
ForwardingEnable	out	Enables forwarding to registers	reset	in	Reset
ForwardingOp	out	Sets the forwarding operation to registers	stack_op	out	The operation to be performed
Instruction / (3-2) register1 / (5-4) register2 / (7-6) opcode / register1 / register2	in	Instruction to be decoded and decodification	WData	in	Data to be written in the register WRegister
JumpAddress	out	The address specified in the jump instruction	WEnable	in	Enable writing to WRegister register
MemAddress	out	Memory address to be accessed by the coded instruction	WRegister	in	The register to be written

Figure 4. Instruction execution.

which correspond to the quaternary digit and quaternary digit vector types. The ALU executes operations between them. In the R2 register input, the first possible choice named MemAddress comes from the quaternary MUX with two inputs (the other two are not used). The control signal in this MUX, named (3) in the datapath, refers to MemAddress [3] qbit of the register. Due to the fact that the ALU executes operations with 8 qbits registers, the extend 3 block extends from 4 qbits to 8 qbits filling with the value 3 the additional 4 qbits and the extend 0 block extends from 4 qbits to 8 qbits filling with the value 0 the additional 4 qbits.

Figure 5 shows the interrupt module which is placed at this stage and it is responsible for identifying arithmetic (i.e. overflow) and stack exceptions and their management. For the context switch, a routine is implemented that records the first ten (0.9) registers. Table 7 defines its input and output signals.

- 1) Write Back: in Figure 6 two functionalities are implemented in this module: 1) to save

Table 6. Instruction execution signals.

Signal	Type	Description	Signal	Type	Description
Address_in / Address_out	in/out	Entry instruction address	MemAddress	in	Current memory address
ALUOp	in	The operation to be executed	MemAddressIn	in	Memory address used
ALUResult	out	The result from the ALU	MemToReg	out	Indicates whether the instruction is a LOAD
ALUZ	out	The signal indicating the occurrence of the jump to start flush	MemWriteIn / MemWrite	in	Enables write to memory
ArithmeticOp	in	ALU operation	R1/MemWrite Data	out	Data to be stored in memory
BranchIn	in	Indicates whether the instruction enables JUMP	R1_in	control	Set multiplexer
exception_interrupt	out	Launch arithmetic exception	RegData	in	Source data from the last executed instruction
Forward_data	in	Data for forwarding to memory	RegData1	in	Source operand one
Forward_data_enable	in	Enable forwarding to memory	RegData2	in	Source operand two
ForwardEnable	in	Enable forwarding registers	RegDest	in	Instruction result of the destination register
ForwardOp	in	Sets the forwarding operation in registers	RegWriteIn / RegWriteOut	in	Enable writing to register RegDest
JumpAddress	out	Next instruction address in the event of a jump	Stack_op_in / Stack_op_out	in	Sets the operation

Figure 5. Interrupt module.

Table 7. Interrupt module signals.

Signal	Type	Description
address	in	Set exception address source
Arithmetic_interruption	in	Arithmetic exceptions
clk	in	Clock
clk_out	out	Controls clk signal
context_change	out	Context switch
reset	in	Reset
reset_out	out	Controls reset signal
stack_overflow	in	Set stack exceptions

the data into the destination register in the file register; 2) to direct the data to Read or Write into the data memory. Table 8 defines its input and output signals. Note carefully, that in the implementation, Write Back has direct connections to the instruction fetch module running on the same clock cycle due to ease of implementation.

Table 8. Instruction write back signals.

Signal	Type	Description	Signal	Type	Description
Address_In/ Address_Out	in/out	Entry instruction address	MemWriteData	in	Data to be stored in memory
ALUResult / (3-0) Address	in	ALU Result	RegData	out	Data to be written to RegDestIn
clk	in	Clock	RegDestIn / RegDestOut	in/out	Instruction result of the destination register
Exception_interruption_in	in	Arithmetic exception entry	RegWriteIn / RegWriteOut	in	Enables write to register RegDestIn
Exception_interruption_out	out	Arithmetic exception output	RegWriteIn / RegWriteOut	in/out	To RegWrite
MemRead	in	Enables read to memory	rst	in	Reset
MemToReg	in	Indicates whether the instruction is a load	Stack_in_data	in	Data to be stored in memory
MemWrite	in	Enables write to memory	stack_op_in / stack_op_out	in/out	Sets the operation performed by the system

The Hazards of the processor are resolved as follows:

- 1) Structural Hazards: to cope with memory contention the Harvard architecture along with the file register are implemented. In the Harvard architecture, the instruction memory (placed in IF block) and data memory (placed in WB block) are separate units. The access to the file register permits read and write in the same cycle by taking advantage of the different phases of the clock.
- 2) Data Hazards: to cope with RAW (Read After Write), WAW (Write After Write) data Hazards, an static technique provided by the compiler along with a forwarding mechanism (dynamic technique) are implemented. The forwarding mechanism takes place between ID and IE stages.
- 3) Control Hazards: to cope with changes in the PC from the PC+1 normal execution, a signal called flush was implemented to implement the flush operation in the pipeline for the sake of simplicity. If this signal is set to logic level three, then all units will be reset (flush), as well as the backup registers. The fact that this signal is generated at the execution stage ensures that only the correct instructions are executed, that is, the units are reset before the Jump instruction and all previous instructions to the Jump instruction are fully executed. There are not a complete treatment of control hazards, in this eCPU only the mechanism to handle these hazards are implemented to

Figure 6. Write back.

demonstrate correct functionality and feasibility of the implementation. Even, if there is not any interrupt vector neither interrupt handlers, this mechanism will allow to implement those operations which will be considered for future work. Figure 7 shows the eCPU datapath with the four-stage pipeline comprised of IF, ID, IE, WB functional units.

SIMULATION RESULTS

The simulations are performed in Modelsim® as follows: 1) To check each instruction, programs are simulated to verify the Instruction Fetch, Decode, Execution and Write Back units; 2) To check the whole project each instruction, in the instruction set, is simulated; and 3) To check a whole program, the Fibonacci sequence is simulated to verify the microprocessor as a whole, in which the first ten terms of the Fibonacci sequence are calculated. Table 9 shows the quaternary assembly program and the machine code generated by the compiler. The comments (i.e. - RAW, - WAR) demonstrate some data hazards that happen in the code.

Figure 8 shows in the registers (left column) /u2/fr/tmp_rf {0}, {1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}, {9}, from top to bottom, the ten first terms of the Fibonacci sequence: {0, 1, 1, 2, 3, 5, 8, 13, 21, 34} in decimal that corresponds to {000, 001, 001, 002, 003, 011, 020, 031, 111, 202} in quaternary logic (last column).

In order to check the interrupt mechanism, the results are popped from the stack generating an exception in the stack (two many pops) and triggering the context switch and pushed the results previously saved in the registers.

To illustrate final hardware results, some Quartus statistics are provided as shown in Figure 9.

SIMULATION DISCUSSION

The simulation results show that the quaternary processor is executing instructions in accordance with the design specifications. The Jump instructions are the most critical since generates control Hazards. The method for solving this problem used in this project causes a delay in the pipeline, but it is defined

Figure 7. Datapath.

Table 9. Fibonacci sequence machine code.

Instruction		Machine Code	Instruction		Machine Code
ADDI \$8,7		20200013	JUMP \$4, FIBO	-WAR	01100111
ADDI \$4,0		20100000	END:		
ADDI \$0,1		20000001	POP \$9		32000021
ADDI \$1,1		20010001	POP \$8		32000020
PUSH \$0		31000000	POP \$7		32000013
PUSH \$1		31000100	POP \$6		32000012
FIBO:			POP \$5		32000011
ADD \$2, \$1, \$0		10010002	POP \$4		32000010
PUSH \$2	- RAW	31000200	POP \$3		32000003
MOV \$0, \$1		13000100	POP \$2		32000002
MOV \$1, \$2	- WAR	13000201	POP \$1		32000001
ADDI \$8, -1		20203333	POP \$0		32000000
JUMP \$8, END		01200130

Figure 8. Fibonacci results.

Figure 9. Quartus statistics results.

this way to simplify implementation, as the eCPU optimization in this proposal is not considered. To properly perform arithmetic operations, it is necessary to implement a package called qbit.vhd file. In this package is defined the qbit, the qbit vector types and the way the operations are implemented. The method of implementation of the arithmetic

operations is simplified because the objective is to deal with aspects of MVL eCPU architecture and not with the implementation of the quaternary arithmetic operations which are left for future work. The main difficulty of this work for the design and simulation of the current quaternary eCPU is the lack of an environment that synthesizes the VHDL

in quaternary logic, therefore, to demonstrate the feasibility of the hardware, the actual implementation of the quaternary ports should be used. Note that, conceptually, the binary architecture is similar to the MVL eCPU, with the difference that by increasing the base of representation, the instructions, data, and control signals are represented in a more compact way (quaternary qbits), which simplifies the design, because it reduces the number of signals involved in the implementation and reduces chip area as the interconnections are also decreased.

CONCLUSION AND FUTURE WORK

In this work, a MVL sixteen instructions set four stages pipeline eCPU Harvard architecture working properly with respect to specification has been shown. The architectural design criterion is being assumed to be the simplicity of implementation, as the main purpose is to demonstrate correct functionality and implementation feasibility. The correct simulated behavior demonstrates the correctness with respect to the specification and the quaternary hardware feasibility must be complemented with an actual implementation utilizing the quaternary gates. To facilitate user programming a two phase's MVL compiler written in Java has been coded. The MVL architecture resembles, clearly, the binary CPU implementation decreasing the learning curve to the design of MVL CPUs. Future work is related to the implementation of the architecture developed here in terms of the actual hardware implementation of the chosen quaternary gates, by developing some synthesis tools; the complete control hazards treatment; and to develop quaternary circuits' minimization tools for the chosen MVL algebra and circuits to perform quaternary arithmetic.

ACKNOWLEDGMENT

The authors would like to thank the Technology Center of Electronics and Informatics of Mato Grosso do Sul (CTEI-MS) for the facilities and the Conselho Nacional de Pesquisa (CNPq) for the financial support.

REFERENCES

- [1] M. Taylor. "Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse". Design Automation Conference (DAC), 49th ACM/EDAC/IEEE, pp. 1131-1136. 2012.
- [2] J. Lukasiewicz. "On three valued-logic". Eds. L. Borkowski (Select Works, North-Holland, Amsterdam), pp. 169-171. 1920.
- [3] E.L. Post. "Introduction to a general theory of elementary propositions". American Journal of Mathematics, pp. 163-185. 1920.
- [4] S. Hurst. "Multiple-valued logic its status and its future". Computers, IEEE Transactions on, pp. 1160-1179. 1984.
- [5] K. Smith. "A multiple-valued logic: a tutorial and appreciation". Computer, pp. 17-27. 1988.
- [6] D. Kozen. "On Kleene Algebras and Closed Semirings". Lecture Notes in Computer Science. Vol. 452, pp. 26-47. July, 1990.
- [7] H.H. Loomis Jr. and R.H. Wyman Jr. "On Complete Sets of Logic Primitives". IEEE Transactions on Electronic Computers, EC. Vol. 14, Issue 2, pp. 173-174. April, 1965.
- [8] P. Kerntopf, M.A. Perkowski and M.H.A. Khan. "On Universality of General Reversible Multiple-Valued Logic Gates". In Proceedings of the 34th International Multiple-Valued Logic Symposium, pp. 68-73. May, 2004.
- [9] M.E.R. Romero, E.M. Martins and R.R. Santos. "Multiple-Valued Logic Algebra for the Synthesis of Digital Circuits". 39th International Symposium on Multiple-Valued Logic, United States, Issue 5010410, pp. 262-267. 2009.
- [10] M. Glusker, D.M. Hogan and P. Vass. "The ternary calculating machine of Thomas Fowler". IEEE Annals of the History of Computing. Vol. 27, Issue 3. July, 2005. ISSN 1058-6180. DOI: 10.1109/MAHC.2005.49.
- [11] World Heritage Encyclopedia. "Ternary Computer". Publisher: World Heritage Encyclopedia, article Id: WHEBN0003025266, Collection: Classes of Computers, Russian Inventions, Soviet Inventions, Ternary Computers.
- [12] J. Yi, H. Huacan and L. Yangtian. "Ternary Optical Computer Architecture", PhysicaScripta. Vol. 2005, Issue T118, p. 98. 2005.
- [13] M. Kameyama, T. Hanyu, M. Esashi and T. Higuchi. "An NMOS pipelined image processor using quaternary logic". Solid-State Circuits Conference. Digest of Technical Papers. 1985 IEEE International.

- Vol. XXVIII, pp. 86-87. 1985. DOI: 10.1109/ISSCC.1985.1156822
- [14] M. Kameyama, T. Hanyu and T. Higuchi. "Design and implementation of quaternary NMOS integrated circuits for pipelined image processing". IEEE Journal of Solid-State Circuits. Vol. 22, Issue 1, pp. 20-27. February, 1987. ISSN 0018-9200. DOI: 10.1109/JSSC.1987.1052666
 - [15] N.W. Umredkar, M.A. Gaikwad and D.R. Dandekar. "Review of Quaternary Adders in Voltage Mode Multi-Valued Logic". International Journal of Computer Applications (0975 8887), Recent Trends in Engineering Technology. 2013.
 - [16] K.S. Vasundara Patel and K.S. Gurumurthy. "Design of High-Performance Quaternary Adders". 2011 41st IEEE International Symposium on Multiple-Valued Logic, May, pp. 22-26. 2011. ISSN 0195-623X. DOI: 10.1109/ISMVL.2011.65
 - [17] H. Shirahama, A. Mochizuki, T. Hanyu, M. Nakajima and K. Arimoto. "Design of a Processing Element Based on Quaternary Differential Logic for a Multi-Core SIMD Processor". Proceedings of the 37th International Symposium on Multiple-Valued Logic (ISMVL 2007), IEEE Computer Society. 2007.
 - [18] S. Hanzawa, T. Sakata, K. Kajigaya, R. Takemura, T. Sekiguchi and T. Kawahara. "A ternary/quaternary CAM architecture with an NPU-side IP-address compression scheme and a dynamic re-configurable CODEC scheme for large-scale flow-table lookup". IEEE International Conference on Communications, 2005. ICC 2005. 2005. Vol. 2, pp. 1048-1052. May, 2005. ISSN1550-3607. DOI: 10.1109/ICC.2005.1494508
 - [19] M.E.R. Romero, E.M. Martins, R.R. dos Santos and M. Gonzalez. "Universal set of CMOS gates for the synthesis of multiple-valued logic digital circuits". Circuits and Systems I: Regular Papers, IEEE Transactions on, pp. 736-749. 2014.