Implementation of Unipolar and Polar Line Encoding Techniques with MIPS in VHDL

Reginald Geoffrey L. Bayeta IV
Electronics and Communication Department
De La Salle University - Manila
Manila, Philippines
reginald_geoffrey_bayetaiv@dlsu.edu.ph

Isaiah Jassen L. Tupal
Electronics and Communication Department
De La Salle University - Manila
Manila, Philippines
isaiah_tupal@dlsu.edu.ph

***Abstract*—MIPS or Microprocessor without Interlocked Piped States is a general-purpose computer architecture that requires single cycle to execute and a single chip to implement. A useful feature to include in this architecture is line encoding as transmission of data is a ubiquitous feature of electronic devices. With this in mind, an implementation of MIPS computer with select Line Encoding techniques was developed using Vivado and VHDL. Line encoding techniques are Unipolar and polar techniques. Based on the result of the simulation, it can be said that the system accurately encodes the data from the computer. Furthermore, it was found that the execution time is directly correlated to the log of the data string to be encoded. Although successful, the system still has room for improvement such as the use of flags.**

Keywords—MIPS, Computer architecture, line coding, VHDL Computer organization

I. INTRODUCTION

MIPS or the Microprocessor without Interlocked Piped Stages is a general-purpose microprocessor architecture that is developed for single VLSI chip implementation. The objective of developing this design is high compiled code execution performance. When it was developed in the 80s, it differed from it contemporizes that followed the trend of computer architecture. [1] This architecture is also a popular model for teaching computer architecture to students due to its instruction set having a simple structure and low barrier for learning. In additional to that, this architecture demonstrates the relationship between high- and low-level languages with ease. [2] The MIPS architecture is a single cycle architecture, and its instruction memory and data memory are separated. The architecture of this model is shown in Figure 1.

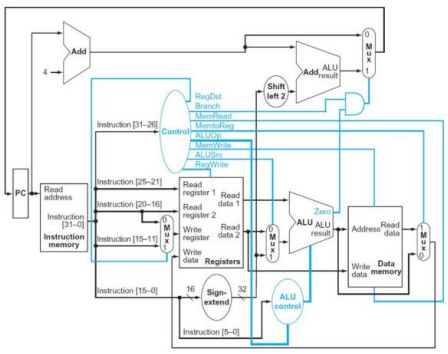

Fig. 1. MIPS architecture [3]

The idea now for this architecture is to add data transmission features for other components to communicate to the MIPS architecture. This feature is a concept in Data communication, which is any procedure that lets information flow from sender to receivers in electronic systems [4]. Currently, information being passed around in text, numbers, images, and other file formats are represented as sequences of bits encoded and sent from a transmitter then received and decoded by the receiver as shown in Figure 2.

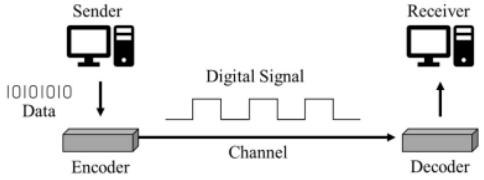

Fig. 2. Data communication

The encoded signal has the following properties: signal level, pulse rate, spectrum, error detection and correction, noise immunity, and complexity [4]. This encoding or *Line Encoding* must be selected with the data at hand in mind, and the features of the hardware such as bandwidth and self-synchronization [4]. This paper presents unipolar and polar line encoding schemes with MIPS in VHDL. The objective of the paper is to create a line encoding module that will encode a serial input with the select line coding techniques. The next is to combine the module with the MIPS architecture and to test if it can encode data from a register. Lastly, it is important that the performance of the computer be tested by measuring the clock cycles given the data string to be encoded.

A. Unipolar

In a unipolar scheme, the output signal levels are either above or below of the time axis. Signal levels can be represented by positive or zero (or negative).

1. Non-Return-To-Zero (NRZ)

   NRZ coding is the most common line coding scheme as it is easy to engineer, and it makes good use of bandwidth. The known issue of NRZ is its lack of synchronization capability and it has DC component making it undesirable for signal transmission.

   a. NRZ-Level (NRZL)

      In NRZL, voltage is constant during bit interval and no transition occurs; zero voltage at bit ‘0’ and constant positive voltage at bit ‘1’. In some implementation, negative voltage is used for bit ‘0’. This

coding scheme is widely used in digital logic systems [4].

b. NRZ-Mark (NRZM)

In NRZM, transition occurs when bit '1' (mark) is handled and bit '0' accounts for no transition. NRZM is also known as differential NRZ encoding and is primarily used in magnetic recording [4].

c. NRZ-Space (NRZS)

NRZS is the opposite of NRZM; signal transition occurs at bit '0' (space) and bit '1' indicates no transition.

2. Return to Zero (RZ)

RZ is like NRZL where '0' is represented by absence of pulse, but '1' in this coding scheme is represented by two half-bit pulse (high to low) in one period. It is used in baseband transmission and magnetic recording [4].

B. *Polar*

In polar schemes, the voltages are on both sides of the time axis; '0' and '1' can be represented by both positive and negative levels.

1. NRZ and RZ

In polar NRZ and RZ, '0' is represented by negative voltage instead of no pulse.

2. Phase encoding

Phase encoding overcomes several limitations of NRZ wherein it has no baseline wandering and no DC component. Its known drawback is its high signal rate and bandwidth requirement compared to NRZ.

a. Biphase Level

In Biphase-L, bit '1' is represented by two half-bit pulse (high to low) in one period and low-to-high for bit '0'. It is used for magnetic recording and data communications. Reversing positions assigned for each bit will make Biphase-L to Manchester encoding [4].

b. Biphase Mark

In Biphase-M, transition occurs at the start of bit inverval. Bit '1' is represented by a second transition half a bit later, while '0' bit does not have a second transition [4].

c. Biphase Space

Biphase-S is the opposite of Biphase-M; Bit '0' is represented by a second transition half a bit later, while '1' bit does not have a second transition [4].

TABLE I. UNIPOLAR AND POLAR LINE CODING SCHEMES

Line Coding Schemes	***Present State***	***Input bit/s***	***Output*** ($0<t<\frac{T}{2}$)	***Output*** ($\frac{T}{2}<t<T$)	***Next State***
NRZS	-	0 1	0 1	0 1	-
NRZM	0 0	0 1	0 1	0 1	0 1
	1 1	0 1	1 0	1 0	1 0
NRZS	0 0 1 1	0 1 0 1	1 0 0 1	1 0 0 1	1 0 0 1
Biphase-L	-	0 1	0 1	1 0	-
Biphase-M	0 0 1 1	0 1 0 1	1 1 0 0	1 0 0 1	1 0 0 1
Biphase-S	0 0 1 1	0 1 0 1	1 1 0 0	0 1 1 0	0 1 1 0
Unipolar RZ	-	0 1	0 1	0 0	-

II. COMPUTER ARCHITECTURE SPECIFICATION AND APPLICATION

A. *Application Area*

The MIPS architecture is made as a general-purpose computer. Hence it has a wide range of application from IoT devices to embedded systems. One thing is common for these applications: they all require some aspect of communication. It is this reason why the authors of this paper decided to add the line encoding functionality on the basic MIPS architecture. When adding this functionality, the simplicity of the implementation is also considered. The functionality of this feature is independent from any algorithm written in the instruction memory of the computer. Although, the algorithm must adapt to the feature such as waiting for the data to be fully encoded before sending another data for serial transmission.

B. *Instruction Set Architecture*

The instruction set of this computer is the same as the usual MIPS architecture as shown in figure 3.

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

Fig. 3. Instruction Set of MIPS [3]

For this architecture, the only additional instructions are the ones for line encoding. In this, we used the opcode part of the 32-bit instruction line to indicate that the computer will now start to encode the data from register 1. The *funct* part of the instruction set, on the other hand, is used to select the type of line encoding to be done on the data that is about to be transmitted. Additional instructions are shown in Table 2.

TABLE II. LINE ENCODING INSTRUCTIONS

Opcode	Funct	Line encoding
111111	000001	NRZL
	000010	NRZM
	000011	NRZS
	000100	BIPHASE L
	000101	BIPHASE M
	000110	BIPHASE S
	000111	UNIPOLAR RZ

The implementation of this system revolves around picking a special register (in this paper it is register 1) and then converting it to serial data wherein a line encoding module would then convert the serial bits into the line encoded bits. The type of encoding is dependent on the Funct part of the instruction of MIPS. This architecture was implemented in the following arrangement.

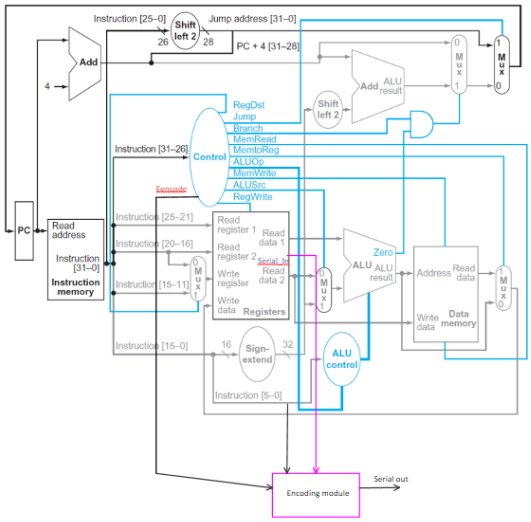

Fig. 4. Architecture of the program

Since the added feature is a simple feature, the algorithm for this will be the following:

Encode Algorithm
addi $0 $1 b
encode S

Fig. 5. Encode assembly.

N is the encoding while S is the string to be encoded.

C. Computer Performance Testing

MIPS is a single cycle computer which makes means that the algorithm part will not take much of the execution time, Since the encoding module supports 32 bits, it will take 32 clock cycles for the encoding to be done assuming that the MSB of the register is also 1.

First and foremost, the line encoding will be verified to see if it outputs the correct waveform. Then the next test would be to see the number of clock cycles needed given the data about to be encoded.

III. DESIGN OPERATION, TEST DATA AND STATISTICAL ANALYSIS OF RESULTS

A. Verification of Encoding

For the verification of line encoding, the input value is hex 0x8D or decimal 141. Results is shown in Figures 5-11.

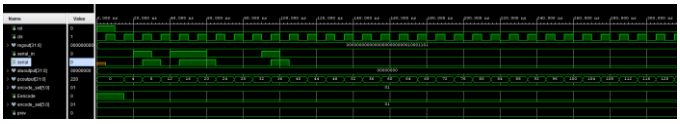

Fig. 5. Simulation of NRZL

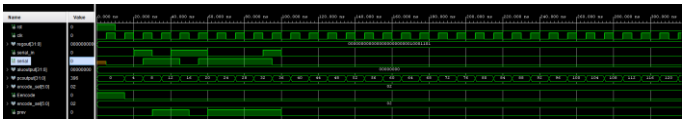

Fig. 6. Simulation of NRZM

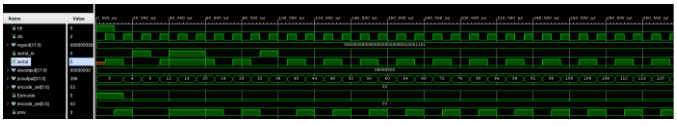

Fig. 7. Simulation of NRZS

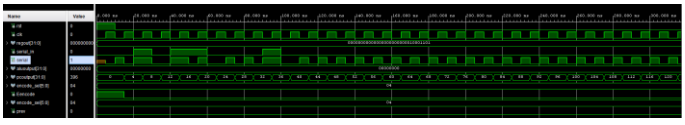

Fig. 8. Simulation of Biphase L

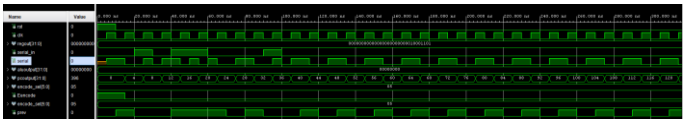

Fig. 9. Simulation of Biphase M

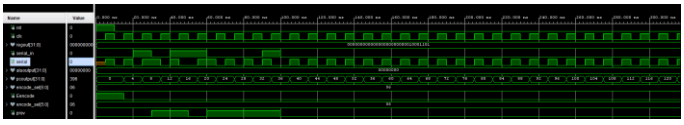

Fig. 10. Simulation of Biphase S

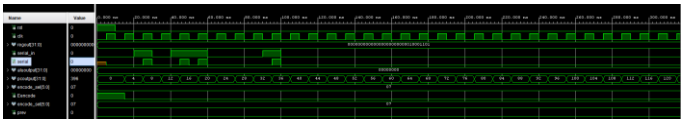

Fig. 11. Simulation of Unipolar RZ

B. Clock cycles of data string

The next test is to see how the line encoding would perform given the data string about to be encoded. Since it is known through inspection that the MSB of the data string

affects the execution time of the encoding, power of 2 is used for this testing.

TABLE III. TOTAL CLOCK CYCLES GIVEN THE DATA STRING

Data String to be encoded	Clock Cycles
1	3
2	4
4	5
8	6
16	7
32	8
64	9
128	10
256	11
512	12
1024	13
2048	14
4096	15
8192	16
16384	17
32768	18
65536	19
131072	20
262144	21
524288	22
1048576	23
2097152	24
4194304	25
8388608	26
16777216	27
33554432	28
67108864	29
134217728	30
268435456	31
536870912	32
1073741824	33
2147483648	34
4294967296	35

When the data string is inputted into the logarithm of base 2, the relationship of the clock cycle and the data string becomes linear as shown in the figure below.

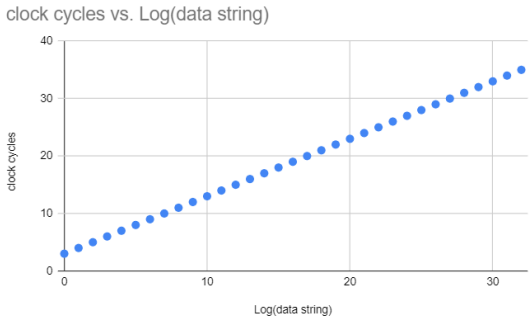

Fig 12. Data string and clock cycles

To test this relationship, linear regression was performed to the log of the data string and clock cycles. It was found that the $R^2 = 1$ while the slope of the line is 1 and the b = 3. This means that the relationship of the execution time of the program and the data string is a perfect linear fit with the following equation:

$$\text{Clock cycles} = \log(\text{datastring}) + 3 \quad \text{[Equation 1]}$$

IV. CONCLUSION AND FUTURE IMPROVEMENT

Based on the results of the verification of line encoding, it can be said that the MIPS with line encoding functionality can accurately represent data strings from the register 1 through select line coding protocols and formats. Furthermore, it was found that the performance of the computer is directly proportional to the logarithm of the data string. This is due to how the MSB dictates the number of clock cycles. Although this project is a success, there are still improvements that can be done to the implementation. First is to use a flag to notify the receiver and the computer on whether or not the encoding process is still executing. Furthermore, one can use read data 1 or read data 2 instead of a single register for better flexibility.

ACKNOWLEDGMENT

We would like to thank Engr. Dino Dominic F. Ligutan and Dr. Cesar A. Llorente for their teachings and guidance in our Computer Architecture and Organization Laboratory (LBYCPD3) and Lecture (CSYSARC) courses.

REFERENCES

[1] Hennessy, J., Jouppi, N., Przybylski, S., Rowen, C., Gross, T., Baskett, F., & Gill, J. (1982). MIPS. ACM SIGMICRO Newsletter, 13(4), 17–22. doi:10.1145/1014194.800930

[2] Pearson, Murray & Mcgregor, Tony & Holmes, Geoffrey. (1998). Teaching computer systems to majors: a MIPS based solution. 5. 10.1145/1275182.1275187.

[3] D. A. Patterson and J. L. Hennessy, Computer Organization and Design: the Hardware/Software Interface. Erscheinungsort nicht ermittelbar: Morgan Kaufmann Publishers In, 2013

[4] Forouzan, B. A. (2013). Data communications and networking, fifth edition. New York: McGraw-Hill.

APPENDIX

TOP MODULE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_signed.all;
use IEEE.numeric_std.all;

entity MIPS_VHDL is
port (
 clk,reset: in std_logic;
 pc_out, alu_result: out std_logic_vector(31 downto 0)
);
end MIPS_VHDL;

architecture Behavioral of MIPS_VHDL is
 signal pc_current: std_logic_vector(31 downto 0); -- 15
 signal pc_next,pc2: std_logic_vector(31 downto 0); -- 15
 signal instr: std_logic_vector(31 downto 0); -- 15
 signal alu_op: std_logic_vector(1 downto 0); -- 1
 signal
jump,branch,mem_read,mem_write,alu_src,reg_write:
std_logic; --
 signal reg_dst,mem_to_reg: std_logic;
 signal reg_write_dest: std_logic_vector(4 downto 0); -- 2
 signal reg_write_data: std_logic_vector(31 downto 0); --
15
 signal reg_read_addr_1: std_logic_vector(4 downto 0); --
2
 signal reg_read_data_1: std_logic_vector(31 downto 0); --
15
 signal reg_read_addr_2: std_logic_vector(4 downto 0); --
2
 signal reg_read_data_2: std_logic_vector(31 downto 0); --
15
 signal sign_ext_im,read_data2,zero_ext_im,imm_ext:
std_logic_vector(31 downto 0); --15
 signal JRControl: std_logic;
 signal ALU_Control: std_logic_vector(3 downto 0);--2
 signal ALU_out: std_logic_vector(31 downto 0); -- 15
 signal zero_flag: std_logic;
 signal im_shift_1, PC_j, PC_beq,
PC_4beq,PC_4beqj,PC_jr: std_logic_vector(31 downto 0);
--15
 signal beq_control: std_logic;
 signal jump_shift_1: std_logic_vector(31 downto 0); --
14?
 signal mem_read_data: std_logic_vector(31 downto 0); --
15
 signal no_sign_ext: std_logic_vector(31 downto 0); -- 15
 signal sign_or_zero: std_logic; --
 signal tmp1: std_logic_vector(15 downto 0); -- 8
 signal pc_jump: std_logic_vector(31 downto 0);
signal pc_branch: std_logic_vector(31 downto 0);
begin
-- PC of the MIPS Processor in VHDL
process(clk,reset, pc_current)
begin
 if(reset='1') then
  pc_current <= x"00000000";
 elsif(rising_edge(clk)) then
  pc_current <= pc_next;
 end if;
end process;
-- PC + 4
  pc2 <= pc_current + x"00000004";
--  pc_next <= pc2; -- di to kasama
-- instruction memory of the MIPS Processor in VHDL
Instruction_Memory: entity
work.Instruction_Memory_VHDL
        port map
       (
        pc=> pc_current,
        instruction => instr
        );
-- jump shift left 1
-- jump_shift_1 <= instr(29 downto 0) & '0'; --13 to 0
    jump_shift_1 <= pc_current(31 downto 28) & instr(25
downto 0) & "00";
-- control unit of the MIPS Processor in VHDL
control: entity work.control_unit_VHDL
   port map
   (reset => reset,
    opcode => instr(31 downto 26),
    alu_op => alu_op,
    jump => jump,
```

```
   branch => branch,
   mem_read => mem_read,
   mem_write => mem_write,
   alu_src => alu_src,
   reg_write => reg_write,
   reg_dst => reg_dst,
   mem_to_reg => mem_to_reg,
   sign_or_zero => sign_or_zero
   );

-- multiplexer regdest, if 0 i instructions, 1 if R-instructions
  reg_write_dest <= instr(15 downto 11) when  reg_dst= '1' else -- 25 to 21
       instr(20 downto 16) when  reg_dst= '0' else -- 6,4 | 01 -- 20 to 16
       instr(25 downto 21);                    -- 9,7 | 00        -- 25 to 21
-- multiplexer regdest -- original
--  reg_write_dest <= "11111" when  reg_dst= '1' else
--       instr(15 downto 11) when  reg_dst= '0' else -- 6,4 | 01
--       instr(30 downto 26);                    -- 9,7 | 00

-- register file instantiation of the MIPS Processor in VHDL
 reg_read_addr_1 <= instr(25 downto 21); -- 12, 10
 reg_read_addr_2 <= instr(20 downto 16);   -- 9, 7

register_file: entity work.register_file_VHDL
 port map
 (
 clk => clk,
 rst => reset,
 reg_write_en => reg_write,
 reg_write_dest => reg_write_dest,
 reg_write_data => reg_write_data,
 reg_read_addr_1 => reg_read_addr_1,
 reg_read_data_1 => reg_read_data_1,
 reg_read_addr_2 => reg_read_addr_2,
 reg_read_data_2 => reg_read_data_2
 );

-- sign extend
 tmp1 <= (others => instr(15));
 sign_ext_im <=  tmp1 & instr(15 downto 0);
 zero_ext_im <= "0000000000000000"& instr(15 downto 0);  -- 9 zeroes
 imm_ext <= sign_ext_im when sign_or_zero='1' else zero_ext_im;

-- new
--pc_jump <= pc_current(31 downto 28) & instr(25 downto 0) & "00";
--pc_branch <= std_logic_vector(TO_UNSIGNED(to_integer(unsigned(pc_current)) + to_integer(unsigned("00000000000000" & instr(15 downto 0) & "00") + 4), pc_branch'length));

-- JR control unit of the MIPS Processor in VHDL
 JRControl <= '1' when ((alu_op="00") and (instr(5 downto 0)="001000")) else '0'; -- orig 1000 -- 001000, 5 downto 0

-- ALU control unit of the MIPS Processor in VHDL
ALUControl: entity work.ALU_Control_VHDL port map
  (
   ALUOp => alu_op,
   ALU_Funct => instr(5 downto 0),
   ALU_Control => ALU_Control
   );

-- multiplexer alu_src
 read_data2 <= imm_ext when alu_src='1' else reg_read_data_2; -- 1
-- ALU unit of the MIPS Processor in VHDL
alu: entity work.ALU_VHDL port map
  (
   a => reg_read_data_1,
   b => read_data2,
   alu_control => ALU_Control,
   alu_result => ALU_out,
   zero => zero_flag
   );
```

```
-- immediate shift 1

-- im_shift_1 <= imm_ext(30 downto 0) & '0';

    im_shift_1 <= imm_ext(29 downto 0) & "00";

 no_sign_ext <= (not im_shift_1) + x"00000001"; --0001

-- PC beq add

-- PC_beq <= (pc2 - no_sign_ext) when im_shift_1(31) =
'1' else (pc2 +im_shift_1); -- 31 is 15

PC_beq <= (pc2 - no_sign_ext) when im_shift_1(31) = '1'
else (pc2 + im_shift_1);

-- beq control

--   beq_control <= branch and zero_flag;

beq_control <= branch and zero_flag;

-- PC_beq

   PC_4beq <= PC_beq when beq_control='1' else pc2;

-- PC_j

-- PC_j <= pc2(31) & jump_shift_1; -- 31 is 15

    PC_j <= jump_shift_1;

--   PC_j <= pc_current(31 downto 28) & instr(25 downto
0) & "00";

-- PC_4beqj

-- PC_4beqj <= PC_j when jump = '1' else  PC_4beq;

-- PC_jr

-- PC_jr <=
std_logic_vector(TO_UNSIGNED(to_integer(unsigned(pc
_current)) + to_integer(unsigned("00000000000000" &
instr(15 downto 0) & "00") + 4), PC_Jr'length));

-- PC_jr <= reg_read_data_1;

-- PC_next

--pc_next <= PC_j when jump = '1' else

--           PC_jr when (branch = '1') else

--           pc_current+x"00000004";

-- pc_next <= PC_jr when (JRControl = '1') else
PC_4beqj;



--    pc_next<=pc_jump when jump = '1' else

--           pc_branch when (branch = '1' and zero_flag = '1'
and instr(31 downto 26) = "000100")

--                or (branch = '1' and zero_flag = '0' and instr(31
downto 26) = "000101") else

--           pc_current+x"00000004";



--pc_next

pc_next <= PC_j when jump = '1' else PC_4beq;
```

```
-- data memory of the MIPS Processor in VHDL

data_memory: entity work.Data_Memory_VHDL port
map

 (

 clk => clk,

 mem_access_addr => ALU_out,

 mem_write_data => reg_read_data_2,

 mem_write_en => mem_write,

 mem_read => mem_read,

 mem_read_data => mem_read_data

 );

-- write back of the MIPS Processor in VHDL

 reg_write_data <= ALU_out when (mem_to_reg = '0')
else -- 1 pc2

      mem_read_data when (mem_to_reg = '1') else
ALU_out; -- 0

-- output

 pc_out <= pc_current; -- pc_current original

 alu_result <= ALU_out;



end Behavioral;
```

INSTRUCTION MEMORY

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

USE IEEE.numeric_std.all;

-- VHDL code for the Instruction Memory of the MIPS
Processor

entity Instruction_Memory_VHDL is

port (

 pc: in std_logic_vector(31 downto 0);

 instruction: out  std_logic_vector(31 downto 0)

);

end Instruction_Memory_VHDL;



architecture Behavioral of Instruction_Memory_VHDL is

signal rom_addr: std_logic_vector(4 downto 0);

 type ROM_type is array (0 to 31) of std_logic_vector(31
downto 0);

 constant rom_data: ROM_type:=(

-- FOR REGINALD : 11811269 : SUM=29 : 4to6=112 :
```

```
-- FOR ISAIAH   : 11847115 : SUM=28 : 4to6=471 :

-- 29: 0000000000011101

-- 28: 0000000000011100

-- 112: 0000000001110000

-- 471: 0000000111010111


-- FOR REGINALD, comment out if ISAIAH

-- "001000"&"00000"&"10000"&"0000000000011101",
-- addi $s0, $zero, a (ID#)

-- "001000"&"00000"&"10001"&"0000000001110000",
-- addi $s1, $zero, b (4th-6th digit of ID#)

-- FOR ISAIAH, comment out if REGINALD

 "001000"&"00000"&"10000"&"0000000000011100",
-- addi $s0, $zero, a (ID#)

 "001000"&"00000"&"10001"&"0000000111010111",
-- addi $s1, $zero, b (4th-6th digit of ID#)


 "000100"&"10000"&"10001"&"0000000000000110",
-- LOOP: beq $s0, $s1, EXIT_LOOP


"000000"&"10001"&"10000"&"01000"&"00000"&"101010", -- if $s1 < $s0 | b < a, effective skip next instruction

 "000100"&"00000"&"01000"&"0000000000000010",
-- if b > a : go to ELSE


"000000"&"10000"&"10001"&"10000"&"00000"&"100010", -- if b < a : a -= b

 "000100"&"00000"&"00000"&"1111111111111011",
-- BACK TO MAIN LOOP


"000000"&"10001"&"10000"&"10001"&"00000"&"100010", -- if b > a : b -= a

 "000100"&"00000"&"00000"&"1111111111111001",
-- BACK TO MAIN LOOP

 "101011"&"00010"&"10000"&"0000000000000000",
-- Store the final value of a (GCD) to function return value register

  others=>(others=>'0'));

begin


  rom_addr <= pc(6 downto 2); -- 6 downto 2 -- originally 4 - 1

  instruction <= rom_data(to_integer(unsigned(rom_addr))) when pc < x"00000040" else x"00000000";


end Behavioral;
```

CONTROL UNIT

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

-- VHDL code for Control Unit of the MIPS Processor

entity control_unit_VHDL is

port (

  opcode: in std_logic_vector(5 downto 0);

  reset: in std_logic;

  alu_op: out std_logic_vector(1 downto 0);


jump,branch,mem_read,mem_write,alu_src,reg_write,reg_dst,mem_to_reg, sign_or_zero: out std_logic

 );

end control_unit_VHDL;


architecture Behavioral of control_unit_VHDL is


begin

process(reset,opcode)

begin

 if(reset = '1') then

   reg_dst <= '0';

   mem_to_reg <= '0';

   alu_op <= "00";

   jump <= '0';

   branch <= '0';

   mem_read <= '0';

   mem_write <= '0';

   alu_src <= '0';

   reg_write <= '0';

   sign_or_zero <= '1';

 else

 case opcode is

  when "000000" => -- R-TYPE

    reg_dst <= '1';

    mem_to_reg <= '0';

    alu_op <= "10";

    jump <= '0';

    branch <= '0';

    mem_read <= '0';

    mem_write <= '0';
```

```
 alu_src <= '0';
 reg_write <= '1';
 sign_or_zero <= '1';

when "100011" => -- LW
 reg_dst <= '0';
 mem_to_reg <= '1'; -- 1
 alu_op <= "00";
 jump <= '0';
 branch <= '0';
 mem_read <= '1';
 mem_write <= '0';
 alu_src <= '1';
 reg_write <= '1';
 sign_or_zero <= '1';

when "101011" => -- SW
 reg_dst <= '0'; --X
 mem_to_reg <= '0'; -- X
 alu_op <= "00";
 jump <= '0';
 branch <= '0';
 mem_read <= '0';
 mem_write <= '1';
 alu_src <= '1';
 reg_write <= '0';
 sign_or_zero <= '1';
when "000100" => -- BEQ
 reg_dst <= '0';
 mem_to_reg <= '0';
 alu_op <= "01";
 jump <= '0';
 branch <= '1';
 mem_read <= '0';
 mem_write <= '0';
 alu_src <= '0';
 reg_write <= '0';
 sign_or_zero <= '1';
when "001011" => -- SLIU
 reg_dst <= '0';
 mem_to_reg <= '0';
 alu_op <= "11";
 jump <= '0';
 branch <= '0';
 mem_read <= '0';
 mem_write <= '0';
 alu_src <= '1';
 reg_write <= '1';
 sign_or_zero <= '0';
when "000010" => -- J
 reg_dst <= '0';
 mem_to_reg <= '0';
 alu_op <= "00";
 jump <= '1';
 branch <= '0';
 mem_read <= '0';
 mem_write <= '0';
 alu_src <= '0';
 reg_write <= '0';
 sign_or_zero <= '1';
when "000011" => -- JAL
 reg_dst <= '1';
 mem_to_reg <= '1';
 alu_op <= "00";
 jump <= '1';
 branch <= '0';
 mem_read <= '0';
 mem_write <= '0';
 alu_src <= '0';
 reg_write <= '1';
 sign_or_zero <= '1';
when "001000" => -- ADDI
 reg_dst <= '0';
 mem_to_reg <= '0';
 alu_op <= "00";
 jump <= '0';
 branch <= '0';
 mem_read <= '0';
 mem_write <= '0';
 alu_src <= '1';
```

```
  reg_write <= '1';
  sign_or_zero <= '1';
 when others =>
   reg_dst <= '1';
   mem_to_reg <= '0';
   alu_op <= "00";
   jump <= '0';
   branch <= '0';
   mem_read <= '0';
   mem_write <= '0';
   alu_src <= '0';
   reg_write <= '1';
   sign_or_zero <= '1';
 end case;
 end if;
end process;


end Behavioral;
```

REGISTER_FILE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.all;
-- VHDL code for the register file of the MIPS Processor
entity register_file_VHDL is
port (
 clk,rst: in std_logic;
 reg_write_en: in std_logic;
 reg_write_dest: in std_logic_vector(4 downto 0);
 reg_write_data: in std_logic_vector(31 downto 0);
 reg_read_addr_1: in std_logic_vector(4 downto 0);
 reg_read_data_1: out std_logic_vector(31 downto 0);
 reg_read_addr_2: in std_logic_vector(4 downto 0);
 reg_read_data_2: out std_logic_vector(31 downto 0)
);
end register_file_VHDL;


architecture Behavioral of register_file_VHDL is
type reg_type is array (0 to 31) of std_logic_vector (31
downto 0);
signal reg_array: reg_type;
begin
 process(clk,rst)
 begin
 if(rst='1') then
   reg_array(0) <= x"00000000";
   reg_array(1) <= x"00000001";
   reg_array(2) <= x"00000002";
   reg_array(3) <= x"00000003";
   reg_array(4) <= x"00000004";
   reg_array(5) <= x"00000005";
   reg_array(6) <= x"00000006";
   reg_array(7) <= x"00000007";
   reg_array(8) <= x"00000008";        -- t0
   reg_array(9) <= x"00000009";
   reg_array(10) <= x"0000000A";
   reg_array(11) <= x"0000000B";
   reg_array(12) <= x"0000000C";
   reg_array(13) <= x"0000000D";
   reg_array(14) <= x"0000000E";
   reg_array(15) <= x"0000000F";
   -- $s0-$s7
   reg_array(16) <= x"00000010";       -- s0 -- running
counter
   reg_array(17) <= x"00000011";       -- s1 -- 9 --
maximum of loop
   reg_array(18) <= x"00000012";       -- s2 -- result
   reg_array(19) <= x"00000013";       -- s3 -- 1 comparator
   reg_array(20) <= x"00000014";       -- s4
   reg_array(21) <= x"00000015";       -- s5
   reg_array(22) <= x"00000016";       -- s6
   reg_array(23) <= x"00000017";       -- s7
   reg_array(24) <= x"00000018";
   reg_array(25) <= x"00000019";
   reg_array(26) <= x"0000001A";
   reg_array(27) <= x"0000001B";
   reg_array(28) <= x"0000001C";
   reg_array(29) <= x"0000001D";
   reg_array(30) <= x"0000001E";
   reg_array(31) <= x"0000001F";
```

```
 elsif(rising_edge(clk)) then
   if(reg_write_en='1') then
    reg_array(to_integer(unsigned(reg_write_dest))) <=
reg_write_data;
   end if;
 end if;
 end process;

 reg_read_data_1 <= x"00000000" when reg_read_addr_1
= "00000" else
reg_array(to_integer(unsigned(reg_read_addr_1)));
 reg_read_data_2 <= x"00000000" when reg_read_addr_2
= "00000" else
reg_array(to_integer(unsigned(reg_read_addr_2)));

end Behavioral;
```

ALU_CONTROL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- VHDL code for ALU Control Unit of the MIPS
Processor
entity ALU_Control_VHDL is
port(
  ALU_Control: out std_logic_vector(3 downto 0); -- 2
  ALUOp : in std_logic_vector(1 downto 0);
  ALU_Funct : in std_logic_vector(5 downto 0) -- 2
);
end ALU_Control_VHDL;

architecture Behavioral of ALU_Control_VHDL is

signal tmp: std_logic_vector(3 downto 0); -- ALU_FUNC
holder

begin

tmp <= ALU_Funct(3 downto 0);

process(ALUOp,ALU_Funct, tmp)
begin
case ALUOp is
when "00" =>
 ALU_Control <= "0010";
when "01" =>
 ALU_Control <= "0110";
when "10" =>
 case tmp is
   when "0000" =>
      ALU_control <= "0010";
   when "0010" =>
      ALU_control <= "0110";
   when "0100" =>
      ALU_control <= "0000";
   when "0101" =>
      ALU_control <= "0001";
   when "1010" =>
      ALU_control <= "0111";
   when others =>
       ALU_control <= "0000"; -- if it does not work,
replace with XXXX
 end case;
when "11" =>
   ALU_control <= "0111";
when others => ALU_Control <= "0000";
end case;
end process;
end Behavioral;
```

ALU

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_signed.all;
-- VHDL code for ALU of the MIPS Processor
entity ALU_VHDL is
port(
 a,b : in std_logic_vector(31 downto 0);        -- src1, src2 |
15
 alu_control : in std_logic_vector(3 downto 0); -- function
select | 2
 alu_result: out std_logic_vector(31 downto 0); -- ALU
Output Result | 15
 zero: out std_logic                            -- Zero Flag
 );
end ALU_VHDL;
```

```
architecture Behavioral of ALU_VHDL is
signal result: std_logic_vector(31 downto 0);
begin
process(alu_control,a,b)
begin
 case alu_control is
 when "0000" =>
  result <= a and b; -- and
 when "0001" =>
  result <= a or b; -- or
 when "0010" =>
  result <= a + b; -- add
 when "0110" =>
  result <= a - b; -- sub
 when "0111" => -- set less than
  if (a<b) then
   result <= x"00000001";
 else
   result <= x"00000000";
 end if;
 when "1100" => -- nor
  result <= a nor b;
 when others => result <= x"00000000"; -- when else
 end case;
end process;
  zero <= '1' when result=x"00000000" else '0';
  alu_result <= result;
end Behavioral;
```

DATA_MEMORY

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.all;
-- VHDL code for the data Memory of the MIPS Processor
entity Data_Memory_VHDL is
port (
 clk: in std_logic;
 mem_access_addr: in std_logic_Vector(31 downto 0);
 mem_write_data: in std_logic_Vector(31 downto 0);
 mem_write_en,mem_read:in std_logic;
 mem_read_data: out std_logic_Vector(31 downto 0)
);
end Data_Memory_VHDL;

architecture Behavioral of Data_Memory_VHDL is
signal i: integer;
signal ram_addr: std_logic_vector(7 downto 0);
type data_mem is array (0 to 255 ) of std_logic_vector (31
downto 0);
signal RAM: data_mem:=(
--    0=>x"00000001",
--    1=>x"00000001",
--    2=>x"00000008",
--    3=>x"00000001",
--    4=>x"00000001",
--    5=>x"00000002",
--    6=>x"00000006",
--    7=>x"00000009",
    others=>(others=>'0'));
begin

 ram_addr <= mem_access_addr(7 downto 0);
 process(clk)
 begin
  if(rising_edge(clk)) then
  if (mem_write_en='1') then
  ram(to_integer(unsigned(ram_addr))) <=
mem_write_data;
  end if;
  end if;
 end process;
   mem_read_data <=
ram(to_integer(unsigned(ram_addr))) when
(mem_read='1') else x"00000000";

end Behavioral;
```

MipsLine.vhdl

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
```

```
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity mips_line is
    Port ( rst : in STD_LOGIC;
           clk : in STD_LOGIC;
           regout: out STD_LOGIC_VECTOR (31 downto 0);
-- $1 for output
           serial: out STD_LOGIC;
           aluoutput : out STD_LOGIC_VECTOR (31 downto 0);
           pcoutput : out STD_LOGIC_VECTOR (31 downto 0)
           );
end mips_line;

architecture Behavioral of mips_line is

component mips_top is
    Port ( rst : in STD_LOGIC;
           clk : in STD_LOGIC;
           regout: out STD_LOGIC_VECTOR (31 downto 0);
-- $1 for output
           aluoutput : out STD_LOGIC_VECTOR (31 downto 0);
           encode_sel: out STD_LOGIC_VECTOR (5 downto 0);
           Eencode: out STD_LOGIC;
           pcoutput : out STD_LOGIC_VECTOR (31 downto 0));
end component;

component nrzl_encode is
Port (  clk: in  STD_LOGIC;
        data : in  STD_LOGIC;
        nrzl: out  STD_LOGIC);
end component;

component nrzm_encode is
Port ( clk: in  STD_LOGIC;
data : in  STD_LOGIC;
nrzm: out  STD_LOGIC;
prev: in STD_LOGIC);
end component;

component nrzs_encode is
Port ( clk: in  STD_LOGIC;
data : in  STD_LOGIC;
nrzs: out  STD_LOGIC;
prev: in STD_LOGIC);
end component;

component birz_encode is
Port ( clk: in  STD_LOGIC;
data : in STD_LOGIC;
birz: out SIGNED(1 downto 0));
end component;

component unirz_encode is
Port ( clk: in  STD_LOGIC;
data : in  STD_LOGIC;
unirz: out  STD_LOGIC);
end component;

component biphasel_encode is
Port ( clk: in  STD_LOGIC;
data : in  STD_LOGIC;
biphasel: out  STD_LOGIC);
end component;

component biphasem_encode is
Port ( clk: in  STD_LOGIC;
prev: in STD_LOGIC;
data : in  STD_LOGIC;
biphasem: out  STD_LOGIC);
```

```
end component;

component biphases_encode is
Port ( clk: in  STD_LOGIC;
prev: in STD_LOGIC;
data : in  STD_LOGIC;
biphases: out  STD_LOGIC);
end component;

component bami_encode is
Port ( clk: in  STD_LOGIC;
prev: in STD_LOGIC;
data : in  STD_LOGIC;
bami: out SIGNED(1 downto 0));
end component;

component pseudoternary_encode is
Port ( clk: in  STD_LOGIC;
prev: in STD_LOGIC;
data : in  STD_LOGIC;
pseudoternary: out SIGNED(1 downto 0));
end component;

signal count : integer range 0 to 31;
signal encode_sel: std_logic_vector(5 downto 0);
signal Eencode: std_logic;
signal serial_in:std_logic;

-- prev signal
signal prev : std_logic := '0';
signal prev_temp : std_logic := '0';
signal prev_entype : std_logic;

-- signal pass to actual serial out
signal serial_nrzl:std_logic; -- a -- no prev -- a = 1
signal serial_nrzm:std_logic; -- b -- with prev
signal serial_nrzs:std_logic; -- c -- with prev
signal serial_biphasel:std_logic; -- d -- no prev
signal serial_biphasem:std_logic; -- e -- with prev
signal serial_biphases:std_logic; -- f -- with prev
signal serial_unirz:std_logic; -- g -- no prev
signal serial_birz:std_logic; -- with negative
signal serial_bami:std_logic; -- -with negative
signal serial_pseudoternary:std_logic; -- with negative

function get_serial(
   encode_type : in STD_LOGIC_VECTOR(5 downto 0);
   a,b,c,d,e,f,g : in STD_LOGIC)
   return STD_LOGIC is
   variable output : STD_LOGIC;
      begin
         case encode_type is
            when "000001" =>
               output := a;
            when "000010" =>
               output := b;
            when "000011" =>
               output := c;
            when "000100" =>
               output := d;
            when "000101" =>
               output := e;
            when "000110" =>
               output := f;
            when "000111" =>
               output := g;
            when others =>
               output := '0';
         end case;
   return output;
end get_serial;

function get_prev(encode_type : in
STD_LOGIC_VECTOR(5 downto 0); serial_in : in
STD_LOGIC; previous : in STD_LOGIC) --serial_in
dapat
   return STD_LOGIC is
   variable temp : STD_LOGIC;
      begin
         case encode_type is
            when "000010" =>
```

```
                report("i got here");
                if(serial_in = '1') then
                    temp := not(previous);
                else
                    temp := previous;
                end if;
            when "000011" =>
                if(serial_in = '0') then
                    temp := not(previous);
                else
                    temp := previous;
                end if;
            when "000101" =>
                if(serial_in = '0') then
                    temp := not(previous);
                else
                    temp := previous;
                end if;
            when "000110" =>
                if(serial_in = '1') then
                    temp := not(previous);
                else
                    temp := previous;
                end if;
            when others =>
                temp := previous;
        end case;
    return temp;
end get_prev;

signal reg_sig : STD_LOGIC_VECTOR (31 downto 0);
begin
    mips : mips_top
        Port map(
          rst => rst,
          clk => clk,
          regout => reg_sig, -- $1 for output
          aluoutput => aluoutput,
          encode_sel => encode_sel,
          Eencode => Eencode,
          pcoutput => pcoutput);

nrzl: nrzl_encode
    Port map(clk => clk,
        data => serial_in,
        nrzl => serial_nrzl);

nrzm: nrzm_encode
    Port map( clk => clk,
        data => serial_in,
        nrzm => serial_nrzm,
        prev => prev);

nrzs: nrzs_encode
    Port map( clk => clk,
        data => serial_in,
        nrzs => serial_nrzs,
        prev => prev);

biphasel: biphasel_encode
    Port map( clk => clk,
        data => serial_in,
        biphasel => serial_biphasel);

biphasem: biphasem_encode
    Port map( clk => clk,
        prev => prev,
        data => serial_in,
        biphasem => serial_biphasem);

biphases: biphases_encode
    Port map( clk => clk,
        prev => prev,
        data => serial_in,
        biphases => serial_biphases);

unirz: unirz_encode
    Port map(clk=>clk,
        data => serial_in,
        unirz => serial_unirz);
```

```
    -- result of serial out

    serial <= get_serial(encode_sel, serial_nrzl, serial_nrzm,
serial_nrzs,

    serial_biphasel, serial_biphasem, serial_biphases,
serial_unirz);



    process(rst,clk,count,Eencode)

    begin

        if(rst = '1') then

            serial_in <= '0';

--            serial <= '0';

        end if;

        if falling_edge(clk) then

            if(Eencode = '1') then

                count <= 0;

                serial_in <= '0' ;

            elsif (count < 32) then

                serial_in <= reg_sig(count);

                count <= count + 1;

            end if;


            -- get previous serial


            prev <= get_prev(encode_sel, serial_in, prev);

        end if;

    end process;


    regout <= reg_sig;

end Behavioral;
```