THE UNIVERSITY OF NEW SOUTH WALES

SYDNEY 2052 · AUSTRALIA

Session 2, 2002

COMP3211
Computer Architecture

Assignment 1

Single-cycle Processor Design

Group members:

Weng Mun Au Yong
Lih Wen Koh
Seng Lin Shee

Tutor:

Usama Malik

Table of Contents

1. Problem description

Design a single-cycle RISC machine that *adds* $(100)_{10}$ to the *maximum* of a set of integers stored in data memory (the number of integers in the set may vary).

The basic requirements are as follows:

1. Based on the MIPS instruction set, determine the minimum instruction set needed for this machine.
2. Construct a block diagram of your computer, clearly indicating which signals connect the components. (In this setting a block should be considered to be a stage of the datapath.)
3. Describe how each instruction in your instruction set is performed by the machine (including dataflow and control signals).
4. Provide an outline description of the interface and function of each block.
5. Draw a register transfer level diagram of each block identified above.
6. Determine or justify the delay of each block using the following assumptions:
   - the (inertial) delay of a gate is 0.5 ns
   - the time needed to set/clear a flip-flop is 0.5 ns; a flip-flop can be read without delay
   - within a block, wires have no delay associated with them; between blocks, assume there is a 0.2 ns transport delay
7. Construct a VHDL model of your design from steps 1 – 6
8. Assemble the instructions needed to test your design by hand and load them into instruction memory. Initialize your data memory with suitable values.
9. Simulate and verify the correctness of your design
10. Analyze the performance of your design

2 Minimum Instruction Set for Machine

In this section, we describe how we determine the minimum instruction set, based on the MIPS instruction set, needed for this machine which adds $(100)_{10}$ to the maximum of a set of integers stored in data memory.

2.1 High Level Description (Pseudocode) of Machine

To guide us in deriving the assembly code, here is a high-level description of what this machine does.

/* For a given size N, assume the array ***num*** is already initialized with
desired integer values */

```
int num[N] ;
int max = num[0] ;
```

/* Find the maximum from the array of integers */

```
for i from 0 to N do
      if ( num[i] > max) do
            max = num[i] ;
      end if ;
end for ;
```

/* Final result */

```
int result = max + 100 ;
```

2.2 Assembly (Pseudo)-Code for Machine

From the high-level description above, we derived the equivalent assembly code for this machine based on a few assumptions as stated below.

Data Memory

Assume that the data memory is already initialized with the required integer values at program startup. Here we define the format of the data in the data memory: the first integer(N) indicates the number of integer values that follow, from which the maximum will be computed.

Index	*Data In Memory*
0	Size (N)
1	num1
2	num2
…	…
N	numN
N+1	0
…	…
31	0

Figure 2-1: Data Memory of Size 32x32

Instruction Memory

Assume that the contents of all registers in the register file are initialized to zero at program startup. The program startup and linking sections are not included here. Only the operations of interest (fetching integers, finding the maximum and adding of $(100)_{10}$ to the maximum) are loaded into the instruction memory.

Using registers

R0 (0x00) as offset to base address (initially 0, incremented by 4 each time),
R1 (0x01) as the counter (initially 0, incremented by 1 each time),
R2 (0x02) as the loop count (size N),
R3 (0x03) to hold the maximum value,
R4 (0x04) to hold current integer to be compared with,

The following are the instructions (with description and the actual instruction code) we would like to load into the instruction memory to test our machine.

Index	*Instruction Description*	*32-bit Instruction Code (in Hex)*			
0	Load size N from data memory at location 0x0000 into register R2.	0x23	0x00	0x02	0x0000
1	Add immediate value 0x0004 to the contents of register R0 and store the result back to R0. *(R0 was initialized to zero at program startup. This step increments R0 by 4)*	0x08	0x00	0x00	0x0004
2	Load first integer value from data memory at location 0x0004 (given by base address 0x0000 and offset 0x0004 in register R0) into register R3. *(initialize max value to the first number)*	0x23	0x00	0x03	0x0000
3	Load integer value from data memory at location given by base address 0x0000 and offset value in register R0, into R4. *(load current num to be compared)*	0x23	0x00	0x04	0x0000
4	Branch to Instruction[6] if R4 < R3. *(conditional branch to 'end if' only if current num < max)*	0x16	0x04	0x03	0x0001 *(1 in decimal)* *(i.e. shifted by 1 word of 4 bytes)*
5	Add immediate value 0x0000 to the contents of register R4 and store result in R3. *(update max value to the current num)*	0x08	0x04	0x03	0x0000
6	Add immediate value 0x0004 to the contents of register R0 and store result back to R0. *(increment R0 by 4)*	0x08	0x00	0x00	0x0004
7	Add immediate value 0x0001 to the contents of register R1 and store result back to R1. *(increment R1 by 1)*	0x08	0x01	0x01	0x0001

8	Branch to Instruction[3] if R1 / = R2. *(conditional branch to start of 'for' loop if there are still integers to be compared to find the max value)*	0x05	0x01	0x02	0xFFF4 *(-6 in decimal)*
9	Add immediate value $(100)_{10}$ to the contents of register R3 and store the result back to R3. *(R3 now holds the maximum value from all the nums to be compared. $(100)_{10}$ then added to the maximum value. R3 now holds the final result)*	0x08	0x03	0x03	0x0064 *(100 in decimal)*
10	Add immediate value 0x0001 to the contents of register R5 and store the result back to R5. *(R5 was initialized to zero at program startup. This step changes the content of R5 to 0x0001)*	0x08	0x05	0x05	0x0001
11	Branch to Instruction[11] if R5 /= R6 *(R6 was initialized to zero at program startup. This conditional branch is always true. This step serves as an infinite loop to signify end of program)*	0x05	0x05	0x06	0xFFFF *(-1 in decimal)*
12	NoOp *(No Operation)*	0x0000			
…	…	…			
31	NoOp	0x0000			

Figure 2-2: Instruction Memory of Size 32x32

2.3 Minimum Instruction Set Required

From above, we have identified **4 instructions** from the MIPS Instruction Set which are required in this machine. The four 32-bit instructions and their formats are as follows:

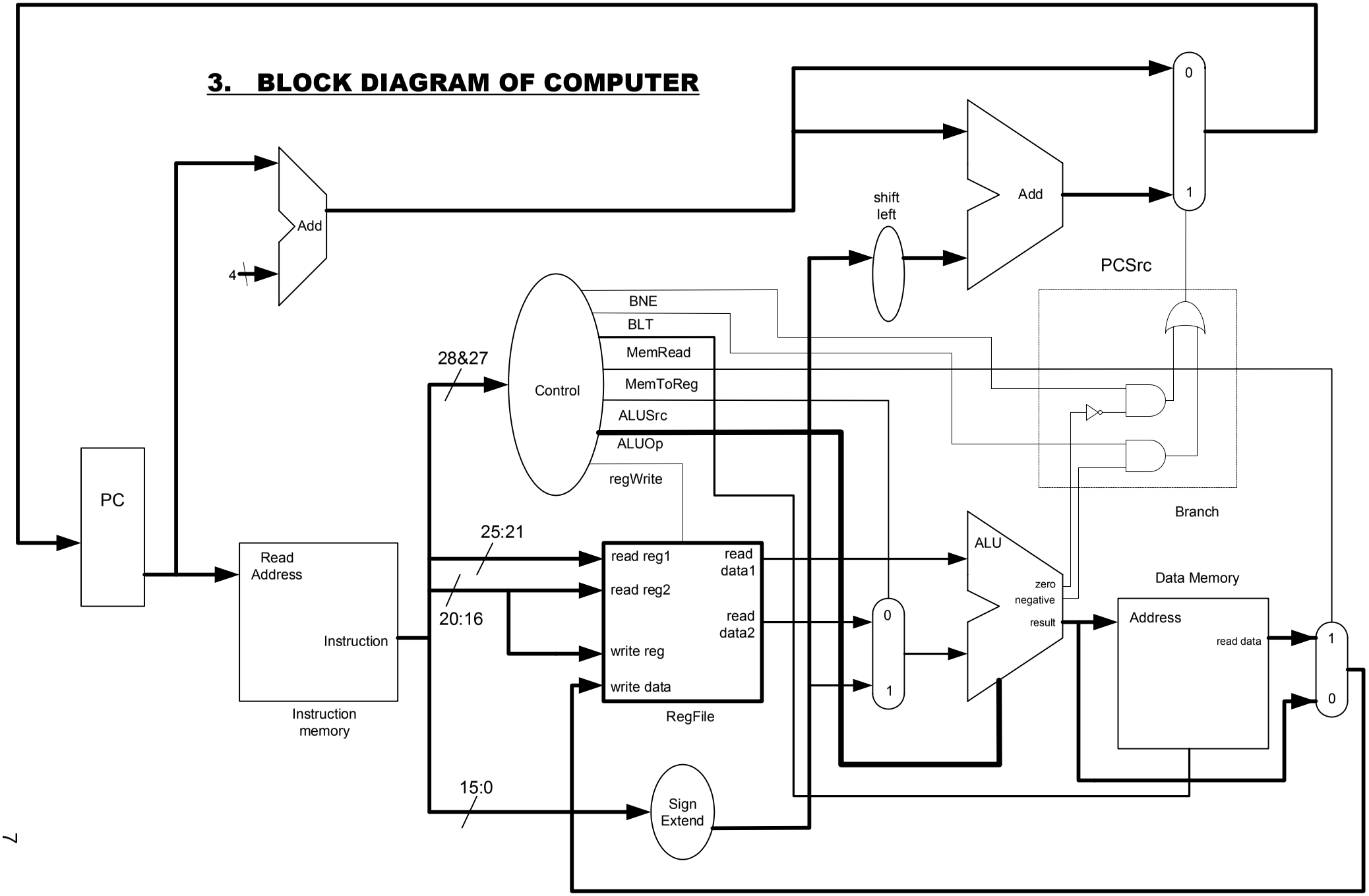
Assembly Format	*32-bit Instruction Code Format*			
addi rt, rs, imm	0x08	rs	rt	imm
lw rt, address	0x23	rs	rt	offset
bne rs, rt, label	0x05	rs	rt	offset
blt rs, rt, label	0x16	rs	rt	offset

Figure 2-3: 32-bit instructions and their respective formats for this machine

Note:

blt is listed as a pseudo-instruction in the MIPS instruction set. So the op field (bits 31 to 26) for *blt* has been chosen as 0x16 where no other MIPS instruction had taken this slot.

3. BLOCK DIAGRAM OF COMPUTER



4 Execution Path for Each Instruction

4.1 Load Word (lw)

There are 5 steps in executing a *load word* instruction : (refer to *Figure 4-1*)

Example : lw R1, offset(R2)

1. An instruction is fetched from the instruction memory and the PC is incremented.

2. A register (*R2*) value is read from the register file.

3. The ALU computes the sum of the value read from the register file and the sign-extended lower 16 bits of the instruction (*offset*).

4. The sum from the ALU is used as the address for the data memory.

5. The data from the memory unit is written into the register file: the register destination is given by bits 20-16 of the instruction (*R1*).

4.2 Add Immediate (addi)

There are 4 steps in executing an *add immediate* instruction : (refer to *Figure 4-2*)

Example : addi R1, R2, immediate

1. An instruction is fetched from the instruction memory and the PC is incremented.

2. A register (*R2*) value is read from the register file.

3. The ALU computes the sum of the value read from the register file and the sign-extended lower 16 bits of the instruction (*immediate)*.

4. The result from the ALU is written into the register file using bits 20-16 of the instruction to select the destination register (*R1*).

4.3 Branch On Not Equal (bne)

There are 4 steps in executing a *branch on not equal* instruction : (refer to *Figure 4-3*)

Example : bne R1, R2, offset

1. An instruction is fetched from the instruction memory and the PC is incremented.

2. Two registers (*R1* and *R2*) are read from the register file.

3. The ALU perform a subtract on the data values read from the register file. The value of PC + 4 is added to the sign-extended, lower 16 bits of the instruction (*offset*) shifted left by two, the result is the branch target address.

4. The Zero flag from the ALU is used to decide which adder result to store into the PC.

4.4 Branch On Less Than (blt)

There are 4 steps in executing a *branch on less than* instruction : (refer to *Figure 4-4*)

Example : blt R1, R2, offset

1. An instruction is fetched from the instruction memory and the PC is incremented.

2. Two registers (*R1* and *R2*) are read from the register file.

3. The ALU perform a subtract on the data values read from the register file. The value of PC + 4 is added to the sign-extended, lower 16 bits of the instruction (*offset*) shifted left by two, the result is the branch target address.

4. The Negative flag from the ALU is used to decide which adder result to store into the PC.

Figure 4-1 Instruction : load word (lw)

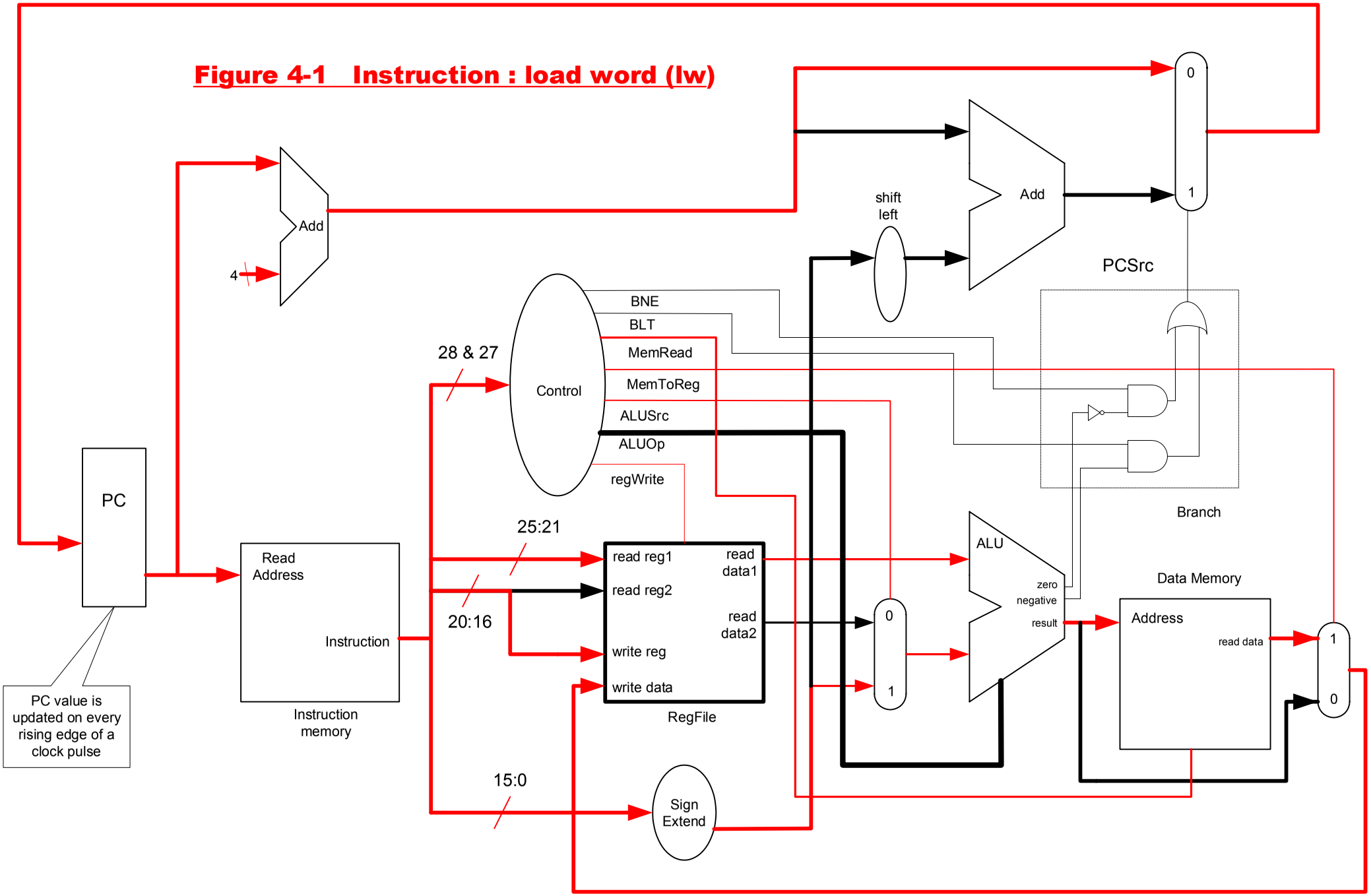


Figure 4-2 Instruction : Add Immediate (addi)

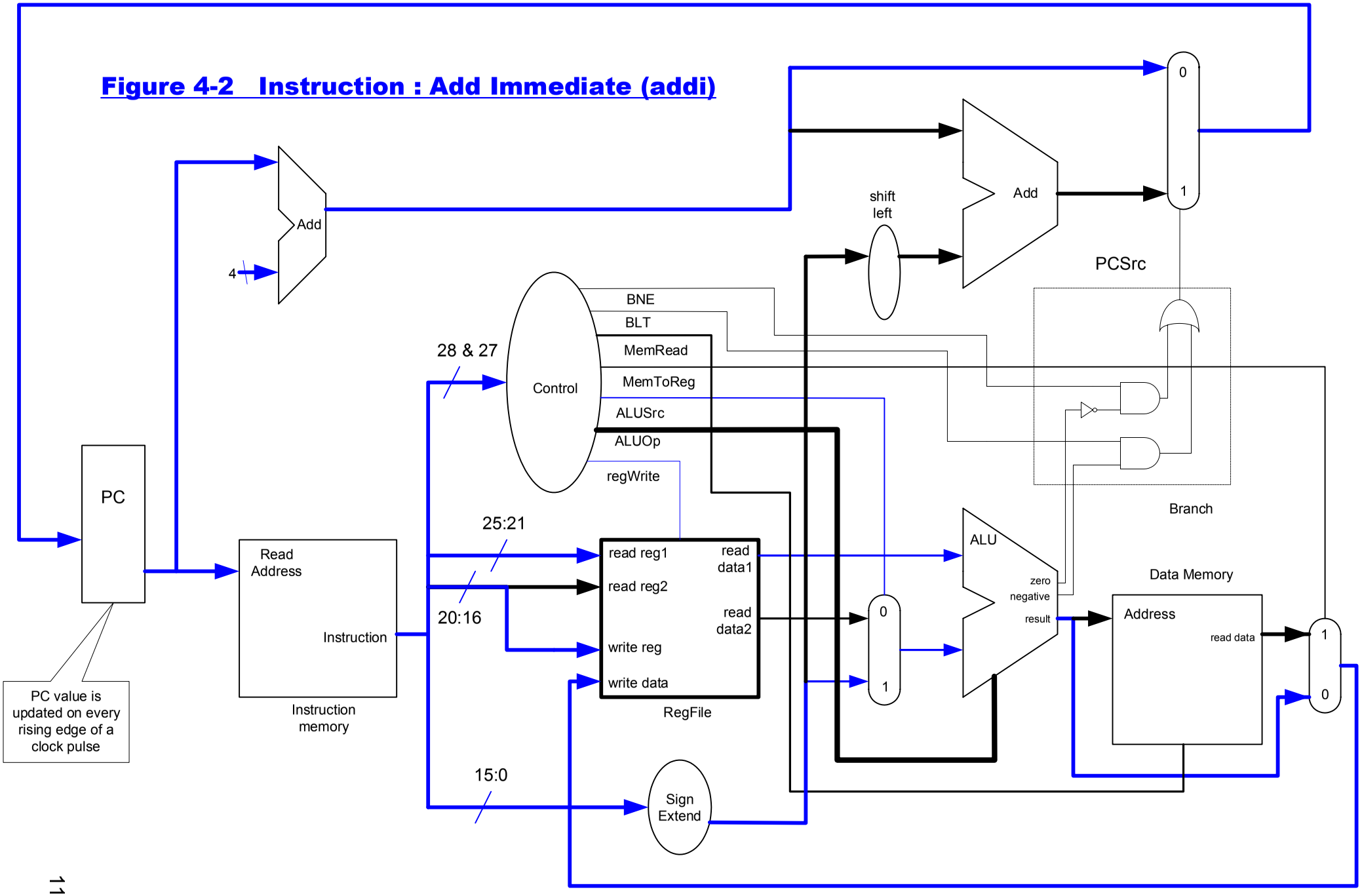


Figure 4-3 Instruction : branch not equal (bne)

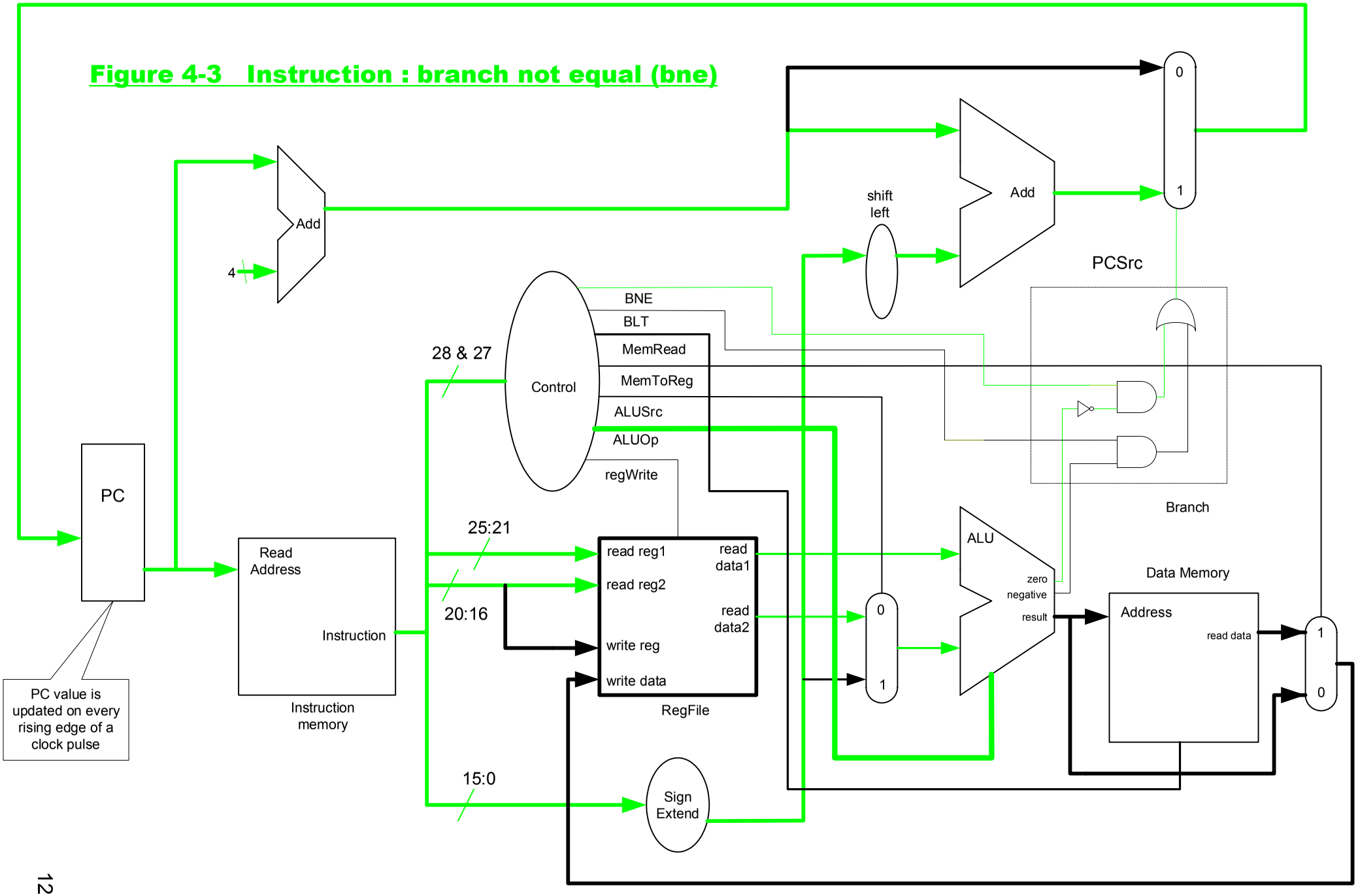
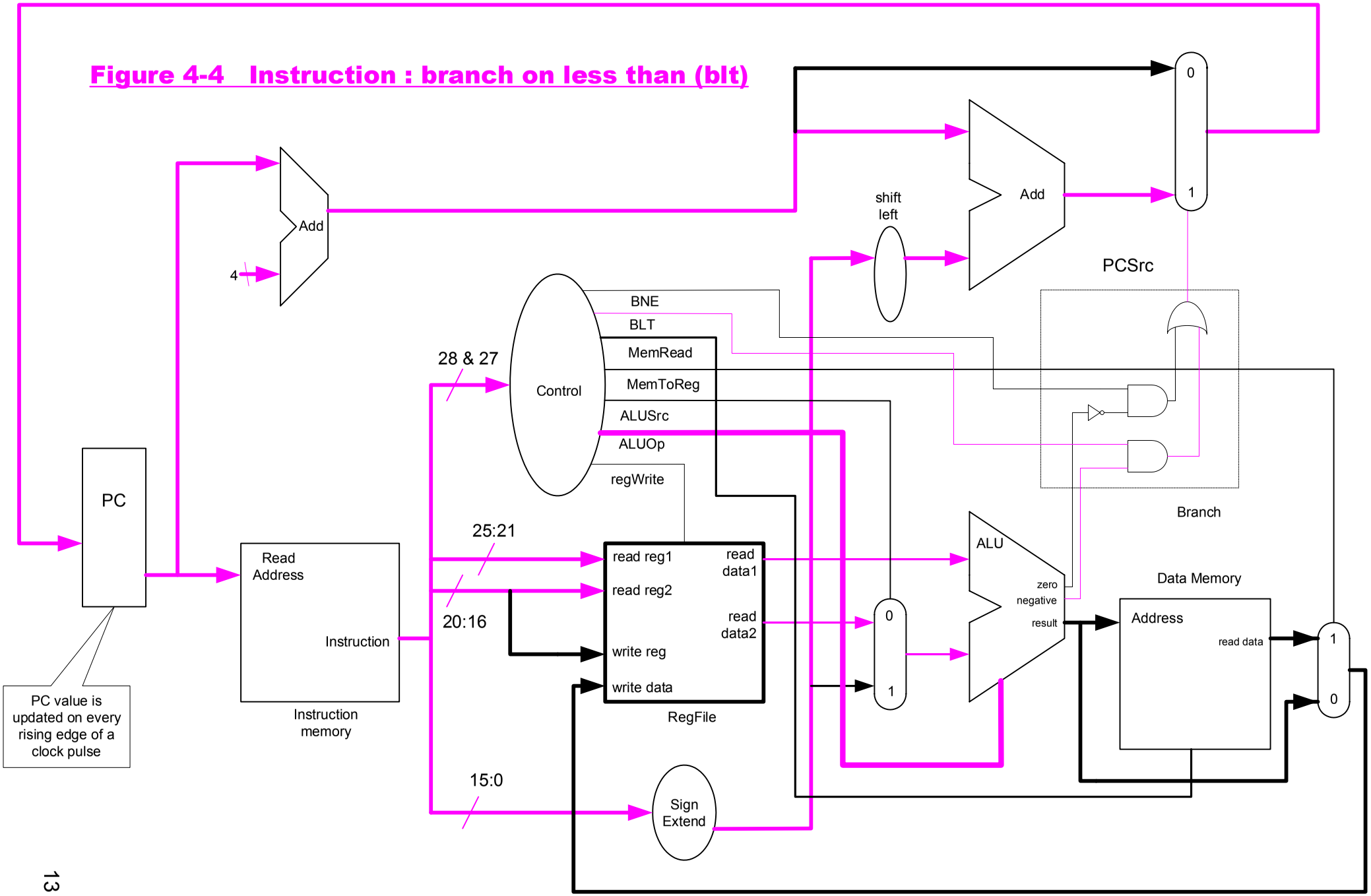


Figure 4-4 Instruction : branch on less than (blt)



5.1 2-to-1 Multiplexer for 32-bit Data

5.1.1 Block Diagram

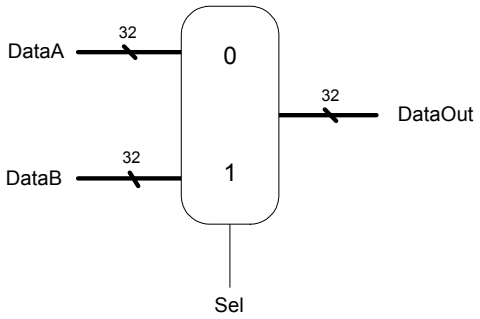


Figure 5-1: 32-bit 2-to-1 Multiplexer

5.1.2 Interface

Input ports : *DataA, DataB* -- 32-bit data values to be channeled to the output
Sel -- selection line to choose either *DataA* or *DataB*

Output port : *DataOut* -- 32-bit output

5.1.3 Function

Sel	*DataOut*
0	DataA
1	DataB

Figure 5-2: Truth Table for 2-to-1 Multiplexer

The 2-to-1 multiplexer acts as a data selector. When *Sel* is 0, *DataA* is channeled to the output port *DataOut*, and when *Sel* is 1, *DataB* is channeled to the output instead.

5.1.4 Implementation

The 2-to-1 multiplexer for 32-bit data is constructed by replicating a single bit 2-to-1 multiplexer 32 times. The implementation of a single bit 2-to-1 multiplexer is shown below.

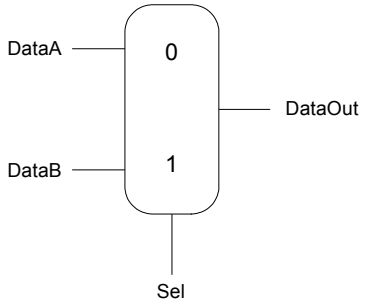


Figure 5-3: Single Bit 2-to-1 Multiplexer

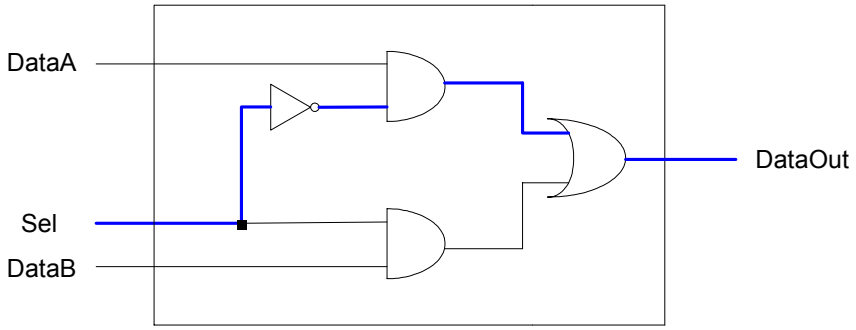


Figure 5-4: Implementation of Single Bit 2-to-1Multiplexer

Logic Equation: $\text{DataOut} = \text{DataA} \bullet \overline{\text{Sel}} + \text{DataB} \bullet \text{Sel}$

5.1.5 Delay Justification

Assuming the (inertial) delay of a gate is 0.5 ns, the critical path for a single bit 2-to-1 multiplexer is highlighted in Figure 5-4 and contributes to 3 gate delays, i.e. 3 x (0.5ns) = 1.5ns.

Replicating the single bit 2-to-1 multiplexer 32 times in a parallel fashion to build a 32-bit 2-to-1 multiplexer is still going to give a delay of **1.5 ns.**

5.1.6 VHDL Model

```
--------------------------------------------------
-- 2-to-1 32 bit Multiplexer
--
-- Function  :  Selects data from 2 different 32 bit
--              sources.
--
-- Author    :  lwkoh@cse
-- Date      :  10/09/2002
--
--------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity mux_2_1_32 is
    Port ( DataA : in std_logic_vector(31 downto 0);
           DataB : in std_logic_vector(31 downto 0);
           Sel : in std_logic;
           DataOut : out std_logic_vector(31 downto 0));
end mux_2_1_32;

architecture structural of mux_2_1_32 is

component mux_2_1_1 is
    Port ( DataA : in std_logic ;
              DataB : in std_logic ;
           Sel : in std_logic;
           DataOut : out std_logic);
end component;

begin

       bit0 : mux_2_1_1 port map(DataA(0), DataB(0), Sel, DataOut(0));
       bit1 : mux_2_1_1 port map(DataA(1), DataB(1), Sel, DataOut(1));
       bit2 : mux_2_1_1 port map(DataA(2), DataB(2), Sel, DataOut(2));
       bit3 : mux_2_1_1 port map(DataA(3), DataB(3), Sel, DataOut(3));
       bit4 : mux_2_1_1 port map(DataA(4), DataB(4), Sel, DataOut(4));
       bit5 : mux_2_1_1 port map(DataA(5), DataB(5), Sel, DataOut(5));
       bit6 : mux_2_1_1 port map(DataA(6), DataB(6), Sel, DataOut(6));
       bit7 : mux_2_1_1 port map(DataA(7), DataB(7), Sel, DataOut(7));

       bit8  : mux_2_1_1 port map(DataA(8),  DataB(8),  Sel, DataOut(8));
       bit9  : mux_2_1_1 port map(DataA(9),  DataB(9),  Sel, DataOut(9));
       bit10 : mux_2_1_1 port map(DataA(10), DataB(10), Sel, DataOut(10));
       bit11 : mux_2_1_1 port map(DataA(11), DataB(11), Sel, DataOut(11));
       bit12 : mux_2_1_1 port map(DataA(12), DataB(12), Sel, DataOut(12));
       bit13 : mux_2_1_1 port map(DataA(13), DataB(13), Sel, DataOut(13));
       bit14 : mux_2_1_1 port map(DataA(14), DataB(14), Sel, DataOut(14));
       bit15 : mux_2_1_1 port map(DataA(15), DataB(15), Sel, DataOut(15));

       bit16 : mux_2_1_1 port map(DataA(16), DataB(16), Sel, DataOut(16));
       bit17 : mux_2_1_1 port map(DataA(17), DataB(17), Sel, DataOut(17));
       bit18 : mux_2_1_1 port map(DataA(18), DataB(18), Sel, DataOut(18));
       bit19 : mux_2_1_1 port map(DataA(19), DataB(19), Sel, DataOut(19));
       bit20 : mux_2_1_1 port map(DataA(20), DataB(20), Sel, DataOut(20));
       bit21 : mux_2_1_1 port map(DataA(21), DataB(21), Sel, DataOut(21));
       bit22 : mux_2_1_1 port map(DataA(22), DataB(22), Sel, DataOut(22));
       bit23 : mux_2_1_1 port map(DataA(23), DataB(23), Sel, DataOut(23));
```

```
       bit24 : mux_2_1_1 port map(DataA(24), DataB(24), Sel, DataOut(24));
       bit25 : mux_2_1_1 port map(DataA(25), DataB(25), Sel, DataOut(25));
       bit26 : mux_2_1_1 port map(DataA(26), DataB(26), Sel, DataOut(26));
       bit27 : mux_2_1_1 port map(DataA(27), DataB(27), Sel, DataOut(27));
       bit28 : mux_2_1_1 port map(DataA(28), DataB(28), Sel, DataOut(28));
       bit29 : mux_2_1_1 port map(DataA(29), DataB(29), Sel, DataOut(29));
       bit30 : mux_2_1_1 port map(DataA(30), DataB(30), Sel, DataOut(30));
       bit31 : mux_2_1_1 port map(DataA(31), DataB(31), Sel, DataOut(31));

end structural;


--------------------------------------------------
-- Single Bit 2-to-1 Multiplexer
--
-- Function  :  As data selector.
--              When Sel = 0, DataOut = DataA
--              When Sel = 1, DataOut = DataB
--
-- Author    :  lwkoh@cse
-- Date      :  10/09/2002
--
--------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity mux_2_1_1 is
    Port ( DataA : in std_logic;
           DataB : in std_logic;
           Sel : in std_logic;    -- select line
           DataOut : out std_logic);
end mux_2_1_1;

architecture structural of mux_2_1_1 is

component INV1 is
    Port ( in1 : in std_logic;
           out1 : out std_logic);
end component;

component AND2 is
    Port ( in1 : in std_logic;
           in2 : in std_logic;
           out1 : out std_logic);
end component;

component OR2 is
    Port ( in1 : in std_logic;
           in2 : in std_logic;
           out1 : out std_logic);
end component;

signal nSel : std_logic ;  -- inversion of select line signal
signal Y    : std_logic_vector(1 downto 0);
```

```
begin

    inv_1 : INV1 port map(Sel, nSel);

    and_1 : AND2 port map(DataA, nSel, Y(0));
    and_2 : AND2 port map(DataB,  Sel, Y(1));

    or_1  : OR2  port map(Y(0), Y(1), DataOut) ;

end structural;



--------------------------------------------------
-- Primitive Gate Functions
-- Function : Provide all necessary basic gates
--            to construct higher level components.
--
-- Author   : lwkoh@cse
-- Date     : 10/09/2002
--
--------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity AND2 is
    Port ( in1 : in std_logic;
              in2 : in std_logic;
           out1 : out std_logic);
end AND2;

architecture behavioral of AND2 is
begin
    out1 <= (in1 and in2) after 500 ps  ;
end behavioral;



library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity INV1 is
    Port ( in1 : in std_logic;
           out1 : out std_logic);
end INV1;

architecture behavioral of INV1 is
begin
    out1 <= (not in1) after 500 ps;
end behavioral;
```

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

entity OR2 is
    Port ( in1 : in std_logic;
              in2 : in std_logic;
           out1 : out std_logic);
end OR2;

architecture behavioral of OR2 is
begin
    out1 <= (in1 or in2) after 500 ps ;
end behavioral;
```

5.1.7 Simulation Waveforms

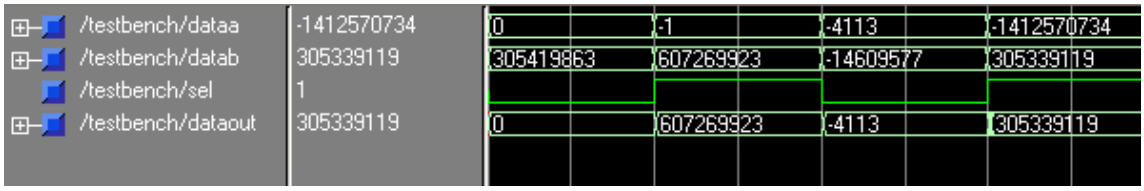


Figure 5-5: Simulation Waveform for 32-bit 2-to-1 Multiplexer

5.2 Left Shifter

5.2.1 Block Diagram

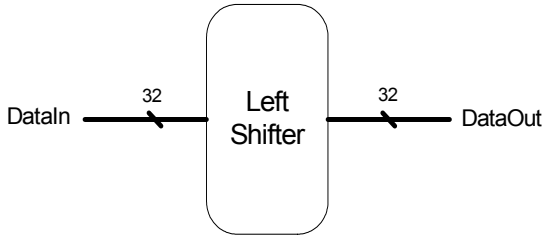


Figure 5-6: 32-bit Left Shifter

5.2.2 Interface

Input port : *DataIn* -- 32-bit input data to be left shifted 2 positions to the left, assuming the leftmost bit is the most significant bit.

Output port : *DataOut* -- 32-bit left-shifter output.

5.2.3 Function

This block shifts the 32-bit input data 2 positions to the left, i.e. towards the most significant bits, and replaces the 2 least significant bits by 00, essentially multiplying *DataIn* by $(4)_{10}$.

For example, *DataIn* of 32-bit data 0xFFFFFFFF will give *DataOut* of 0xFFFFFFFC.

5.2.4 Implementation

The input data bits from position 0 to 29 are directed to the output data bits from position 2 to 31 respectively. The remaining output data bits, bit 0 and 1 are constantly fed by logic '0'.

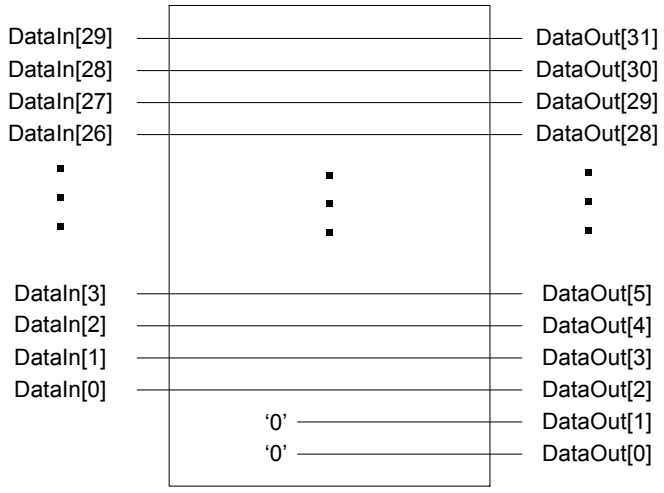


Figure 5-7: Implementation of Left Shifter

5.2.5 Delay Justification

As there are no 'physical' gates involved in the implementation of this left shifter, it is assumed that there is **0 ns** delay in this block.

5.2.6 VHDL Model

```
----------------------------------------------------
-- 32-bit Left Shifter
--
-- Function  :  Shifts a 32 bit data to the left
--              by 2 bit positions.
--
-- Author    :  lwkoh@cse
-- Date      :  10/09/2002
--
----------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity shifter_32 is
    Port ( DataIn : in std_logic_vector(31 downto 0);
           DataOut : out std_logic_vector(31 downto 0));
end shifter_32;

architecture Behavioral of shifter_32 is

begin

    DataOut(31 downto 2) <= DataIn(29 downto 0);
       DataOut(1 downto 0)  <= "00";

end Behavioral;
```

5.2.7 Simulation Waveforms

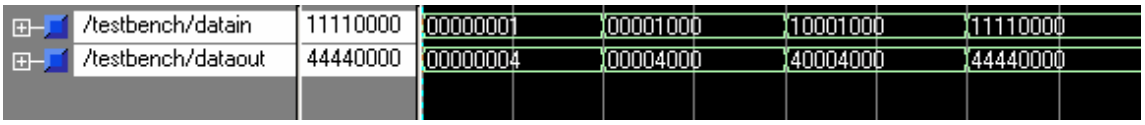


Figure 5-8: Simulation Waveform for 32-bit Left Shifter

5.3 Sign Extender

5.3.1 Block Diagram

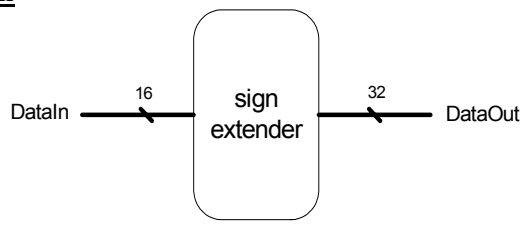


Figure 5-9: 32-bit Sign Extender

5.3.2 Interface

Input port : *DataIn* -- 16 bit input data to be sign-extended

Output port : *DataOut* -- 32 bit output data which has been sign-extended

5.3.3 Function

This block sign-extends a 16-bit input data to a 32-bit output data by padding the most significant 16 bits of the output data by either '1' or '0' depending on the most significant bit of the input data.

For example,

DataIn of 0x0023 will be sign-extended to give *DataOut* of 0x00000023.
DataIn of 0xFA46 will be sign-extended to give *DataOut* of 0xFFFFFA46.

5.3.4 Implementation

The input data bits from position 0 to 15 are directed to the output data bits from position 0 to 15 respectively. The remaining output data bits, from position 16 to 31 are fed by input data bit of position 15.

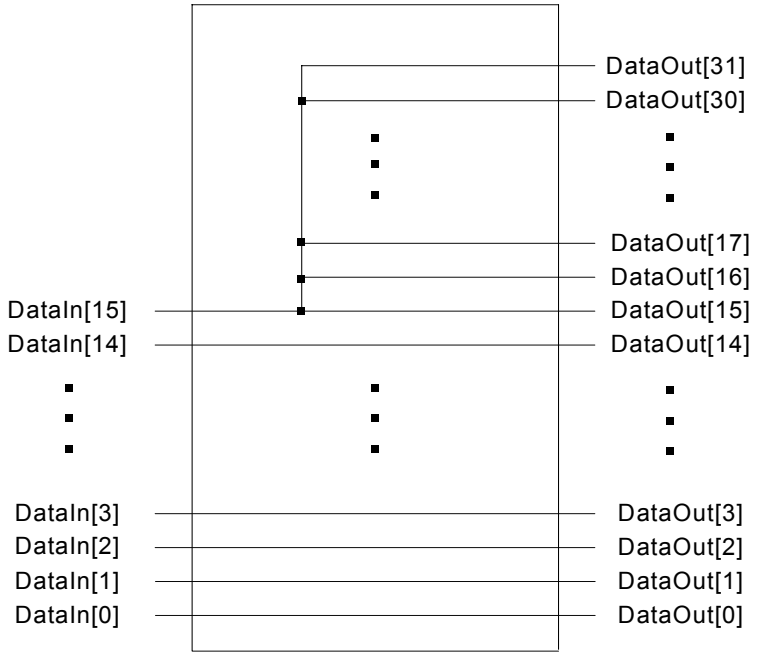


Figure5-10: Implementation of 32-bit Sign Extender

5.3.5 Delay Justification

As there are no 'physical' gates involved in the implementation of this sign extender, it is assumed that there is **0 ns** delay in this block.

5.3.6 VHDL Model

```
--------------------------------------------------
-- 16-to-32 bit Sign Extender
--
-- Function  :  Sign extends 16 bit numbers to 32
--              bit ones.
--
-- Author    :  lwkoh@cse
-- Date      :  10/09/2002
--
--------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity signExtend_16_32 is
    Port ( DataIn : in std_logic_vector(15 downto 0);
           DataOut : out std_logic_vector(31 downto 0));
end signExtend_16_32;
```

```
architecture Behavioral of signExtend_16_32 is

begin

    DataOut(15 downto 0) <= DataIn(15 downto 0) ;
       DataOut(31) <= DataIn(15);
       DataOut(30) <= DataIn(15);
       DataOut(29) <= DataIn(15);
       DataOut(28) <= DataIn(15);
       DataOut(27) <= DataIn(15);
       DataOut(26) <= DataIn(15);
       DataOut(25) <= DataIn(15);
       DataOut(24) <= DataIn(15);
       DataOut(23) <= DataIn(15);
       DataOut(22) <= DataIn(15);
       DataOut(21) <= DataIn(15);
       DataOut(20) <= DataIn(15);
       DataOut(19) <= DataIn(15);
       DataOut(18) <= DataIn(15);
       DataOut(17) <= DataIn(15);
       DataOut(16) <= DataIn(15);

end Behavioral;
```

5.3.7 Simulation Waveforms

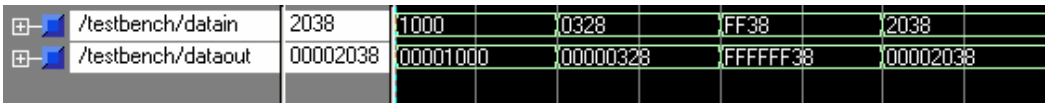


Figure 5-11: Simulation Waveform for 32-bit Sign Extender

5.4 Arithmetic Logic Unit (ALU)

5.4.1 Block Diagram

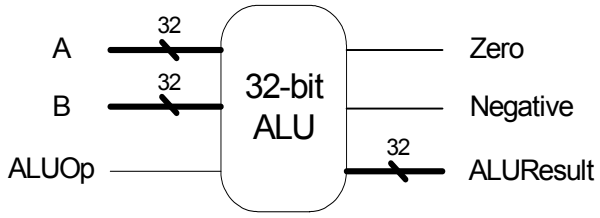


Figure 5-12: 32-bit Arithmetic Logic Unit (ALU)

5.4.2 Interface

Input ports : *A* -- 32-bit data A
B -- 32-bit data B
ALUOp -- When *ALUOp* = 0, an addition is performed.
When *ALUOp* = 1, a subtraction is performed.

Output ports : *Zero* -- Zero flag, asserted when *ALUResult* is zero.
Negative -- Negative flag, asserted when *ALUResult* is negative.
ALUResult -- 32-bit result from either addition/subtraction.

5.4.3 Function

The arithmetic logic unit block has 2 functions, addition or subtraction, selected by the *ALUOp* signal.

5.4.4 Implementation

The 32-bit ALU is constructed using 32-bit adder, XOR gates, OR gates and NOT gate as its components, interconnected in the manner shown in Figure 5-13.

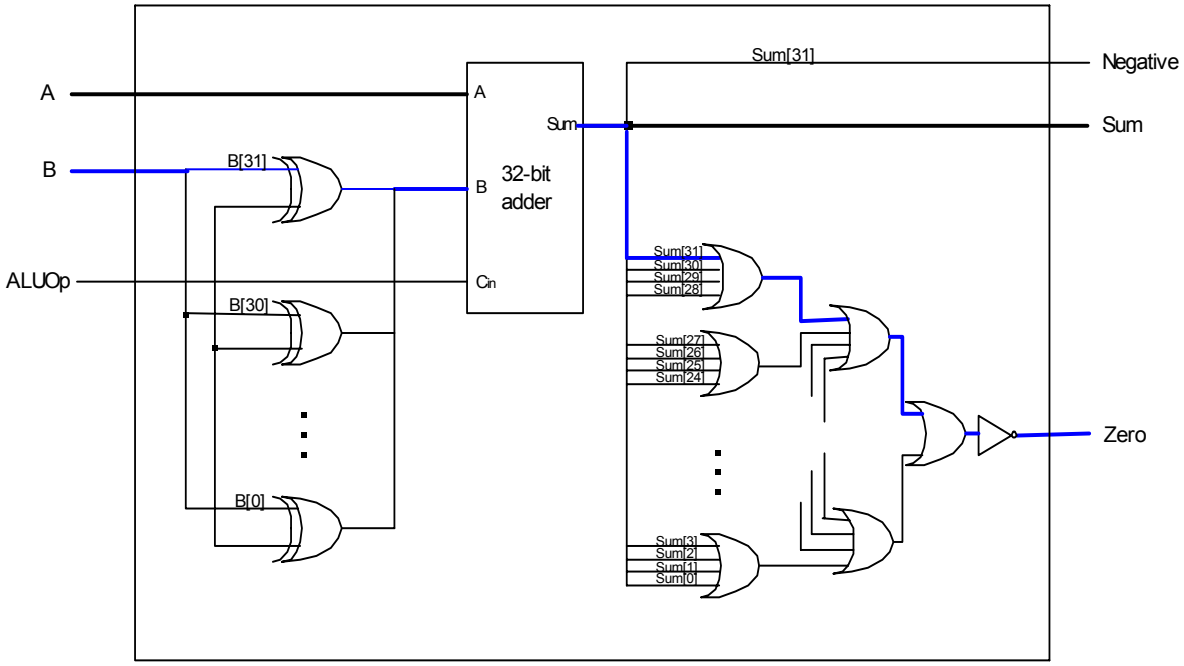


Figure 5-13: Implementation of 32-bit ALU

32-bit adder

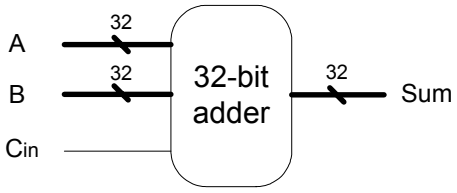


Figure 5-14: Block Diagram of a 32-bit Adder

The 32-bit adder uses 16-bit adder as its building block, interconnected in the manner as shown in Figure 5-15.

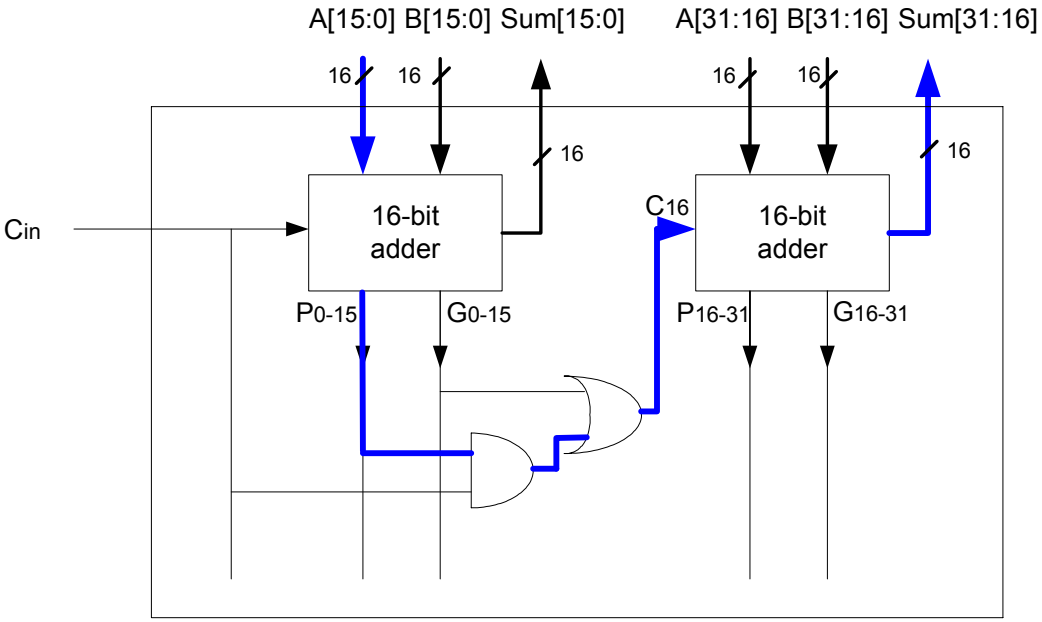


Figure 5-15: Implementation of a 32-bit Adder

16-bit Carry Lookahead Adder

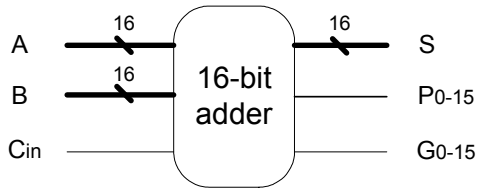


Figure 5-16: Block Diagram of a 16-bit adder

Carry Lookahead configuration is used here instead of ripple carry configuration to improve performance by reducing the delay in the carry path.

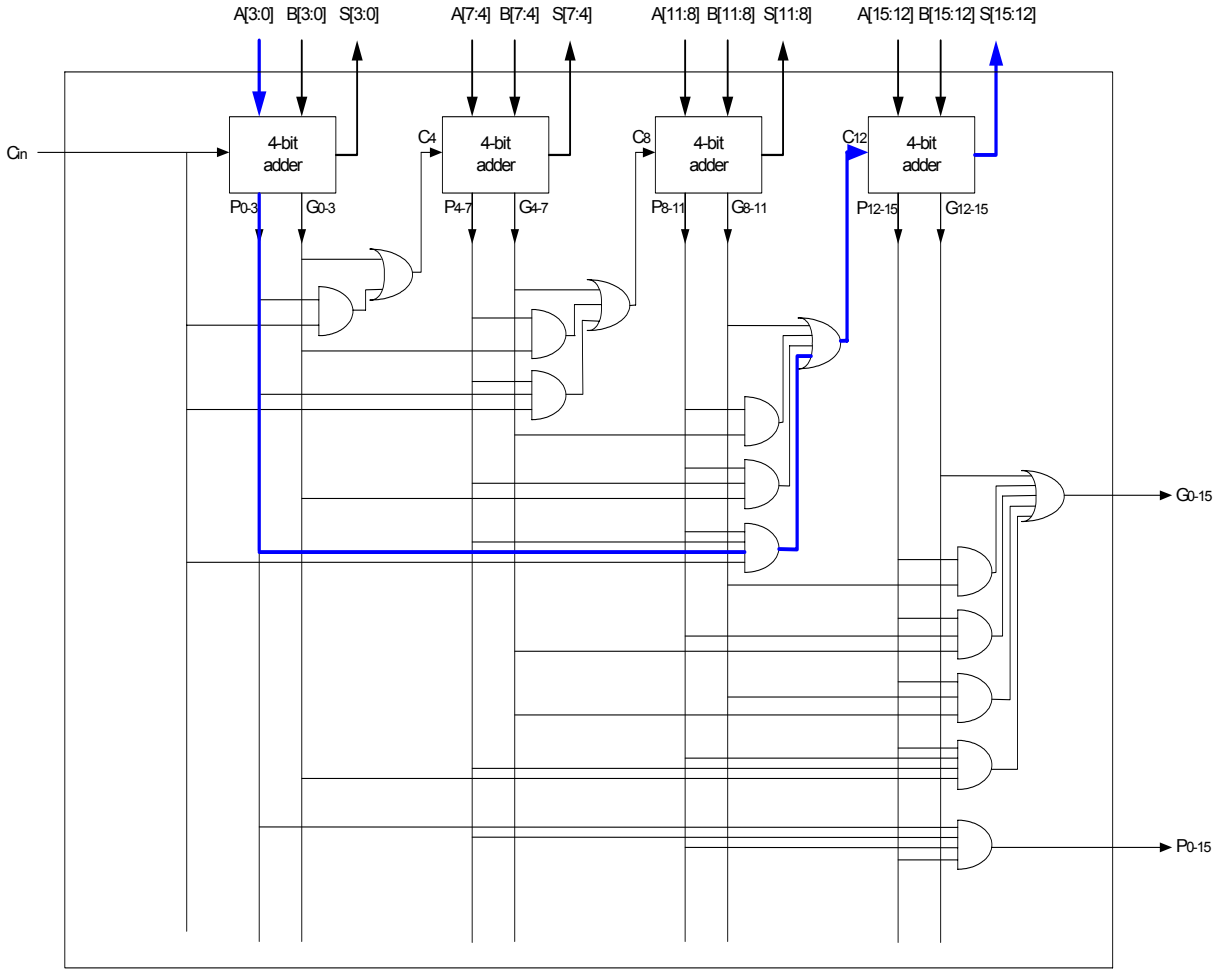


Figure 5-17: Implementation of a 16-bit Carry Lookahead Adder

4-bit Carry Lookahead Adder

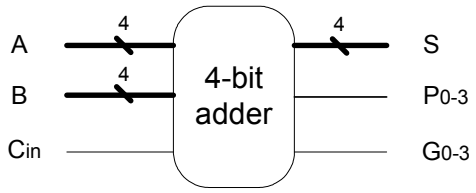


Figure 5-18: Block Diagram of a 4-bit Carry Lookahead Adder

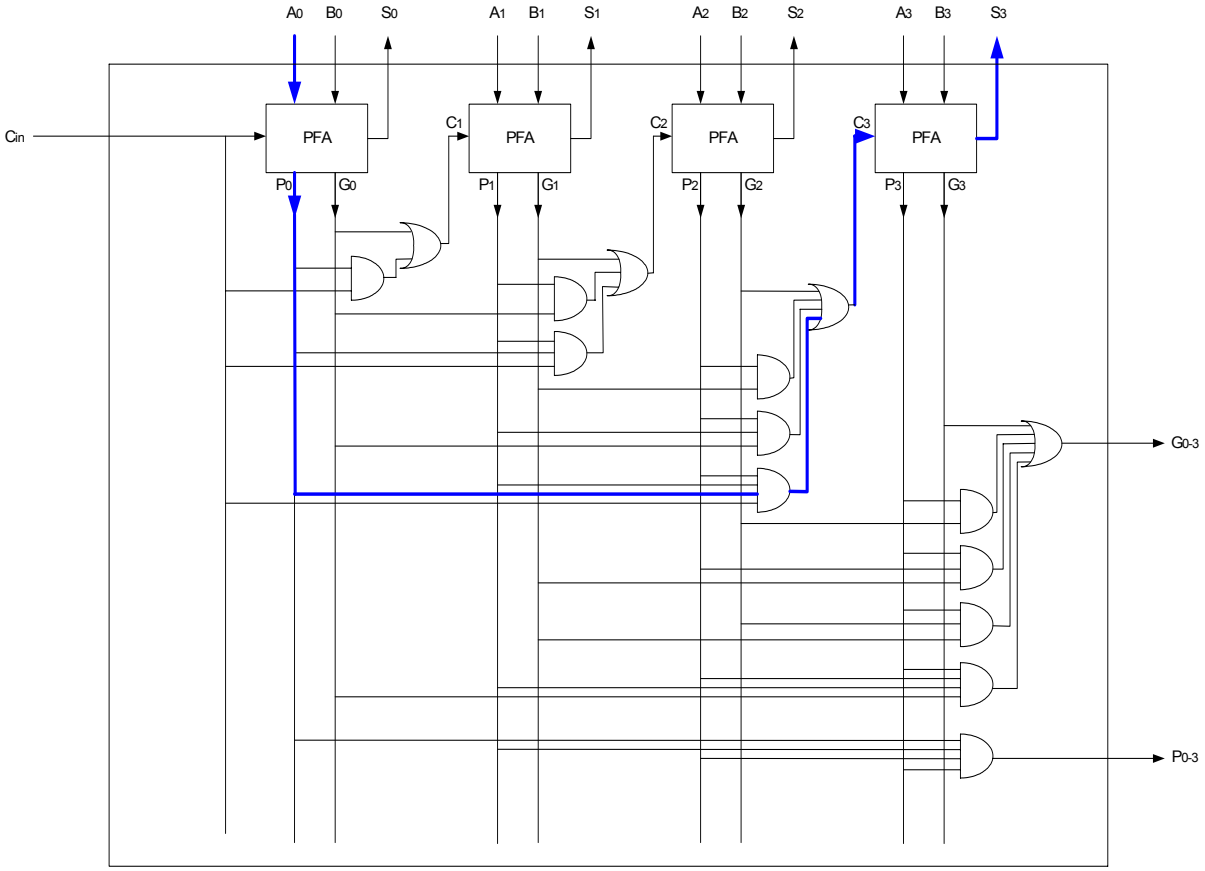


Figure 5-19: Implementation of a 4-bit Carry Lookahead Adder

Partial Full Adder

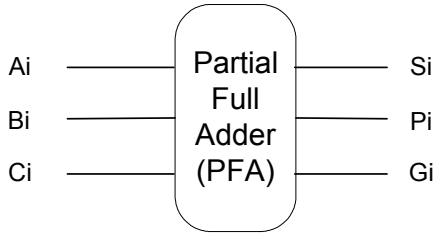


Figure 5-20: Block Diagram of a Partial Full Adder (PFA)

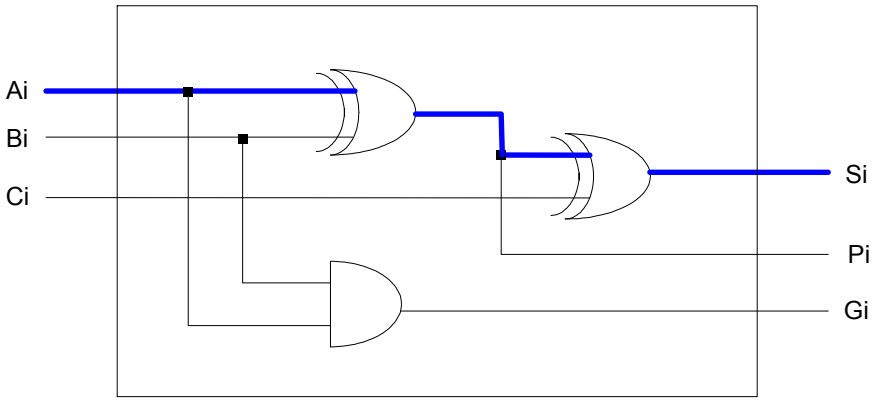


Figure 5-21: Implementation of a Partial Full Adder

Pi: propagate function, Gi: generate function

5.4.5 Delay Justification

The critical path in a partial full adder, as highlighted in Figure 5-21, has 2 gate delays, giving 2 x (0.5ns) = 1.0 ns.

Going up one level to 4-bit adder, referring to Figure 5-19, the critical path contributes to 2 x (1.0ns) (partial full adders) + 2 x (0.5ns) (gates in the propagate path) = 3.0 ns.

At the 16-bit adder level, referring to Figure 5-17, the critical path contributes to 2 x (3.0ns) (4-bit adders) + 2 x (0.5ns) (gates in the propagate path) = 7.0 ns.

At the 32-bit adder level, referring to Figure 5-15, the critical path contributes to 2 x (7.0ns) (16-bit adders) + 2 x (0.5ns) (gates in the propagate path) = 15.0 ns.

At the 32-bit ALU level, referring to Figure 5-13, the critical path contributes to 1 x (0.5ns) (XOR gate) + (15.0ns) (32-bit adder) + 4 x (0.5ns) (OR and NOT gates) = **17.5 ns**.

5.4.6 VHDL Model

```
--------------------------------------------------
-- 32 bit Arithmetic Logic Unit (ALU)
--
-- Function  :  Perform addition or subtraction
--              depending on the operation selection
--              signal (ALUOp)
--
-- Author    :  lwkoh@cse
-- Date      :  10/09/2002
--
--------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity alu_32 is
    Port (
             A : in std_logic_vector(31 downto 0);
             B : in std_logic_vector(31 downto 0);
            ALUOp : in std_logic;
            Zero  : out std_logic;
            Negative : out std_logic;
            ALUResult : out std_logic_vector(31 downto 0));
end alu_32;

architecture structural of alu_32 is

component adder_32 is
    Port ( A : in std_logic_vector(31 downto 0);
           B : in std_logic_vector(31 downto 0);
           Cin : in std_logic;
           Sum : out std_logic_vector(31 downto 0));
end component;

component INV1 is
    Port ( in1 : in std_logic;
           out1 : out std_logic);
end component;

component XOR2 is
    Port ( in1 : in std_logic;
            in2 : in std_logic;
           out1 : out std_logic);
end component;

component OR2 is
    Port ( in1 : in std_logic;
            in2 : in std_logic;
           out1 : out std_logic);
end component;
```

```
component OR4 is
    Port ( in1 : in std_logic;
           in2 : in std_logic;
           in3 : in std_logic;
           in4 : in std_logic;
          out1 : out std_logic);
end component;

signal Y : std_logic_vector(31 downto 0);
signal Z : std_logic_vector(10 downto 0);
signal Sum: std_logic_vector(31 downto 0);
begin

       adder:   adder_32 port map(A, Y, ALUOp, Sum);

       xor_0:   XOR2 port map(B(0),  ALUOp, Y(0));
       xor_1:   XOR2 port map(B(1),  ALUOp, Y(1));
       xor_2:   XOR2 port map(B(2),  ALUOp, Y(2));
       xor_3:   XOR2 port map(B(3),  ALUOp, Y(3));
       xor_4:   XOR2 port map(B(4),  ALUOp, Y(4));
       xor_5:   XOR2 port map(B(5),  ALUOp, Y(5));
       xor_6:   XOR2 port map(B(6),  ALUOp, Y(6));
       xor_7:   XOR2 port map(B(7),  ALUOp, Y(7));
       xor_8:   XOR2 port map(B(8),  ALUOp, Y(8));
       xor_9:   XOR2 port map(B(9),  ALUOp, Y(9));
       xor_10:  XOR2 port map(B(10),  ALUOp, Y(10));
       xor_11:  XOR2 port map(B(11),  ALUOp, Y(11));
       xor_12:  XOR2 port map(B(12),  ALUOp, Y(12));
       xor_13:  XOR2 port map(B(13),  ALUOp, Y(13));
       xor_14:  XOR2 port map(B(14),  ALUOp, Y(14));
       xor_15:  XOR2 port map(B(15),  ALUOp, Y(15));
       xor_16:  XOR2 port map(B(16),  ALUOp, Y(16));
       xor_17:  XOR2 port map(B(17),  ALUOp, Y(17));
       xor_18:  XOR2 port map(B(18),  ALUOp, Y(18));
       xor_19:  XOR2 port map(B(19),  ALUOp, Y(19));
       xor_20:  XOR2 port map(B(20),  ALUOp, Y(20));
       xor_21:  XOR2 port map(B(21),  ALUOp, Y(21));
       xor_22:  XOR2 port map(B(22),  ALUOp, Y(22));
       xor_23:  XOR2 port map(B(23),  ALUOp, Y(23));
       xor_24:  XOR2 port map(B(24),  ALUOp, Y(24));
       xor_25:  XOR2 port map(B(25),  ALUOp, Y(25));
       xor_26:  XOR2 port map(B(26),  ALUOp, Y(26));
       xor_27:  XOR2 port map(B(27),  ALUOp, Y(27));
       xor_28:  XOR2 port map(B(28),  ALUOp, Y(28));
       xor_29:  XOR2 port map(B(29),  ALUOp, Y(29));
       xor_30:  XOR2 port map(B(30),  ALUOp, Y(30));
       xor_31:  XOR2 port map(B(31),  ALUOp, Y(31));

       or_1  :  OR4  port map(Sum(31), Sum(30), Sum(29), Sum(28), Z(0));
       or_2  :  OR4  port map(Sum(27), Sum(26), Sum(25), Sum(24), Z(1));
       or_3  :  OR4  port map(Sum(23), Sum(22), Sum(21), Sum(20), Z(2));
       or_4  :  OR4  port map(Sum(19), Sum(18), Sum(17), Sum(16), Z(3));
       or_5  :  OR4  port map(Sum(15), Sum(14), Sum(13), Sum(12), Z(4));
       or_6  :  OR4  port map(Sum(11), Sum(10), Sum(9), Sum(8), Z(5));
       or_7  :  OR4  port map(Sum(7), Sum(6), Sum(5), Sum(4), Z(6));
       or_8  :  OR4  port map(Sum(3), Sum(2), Sum(1), Sum(0), Z(7));
```

```
       or_9  :  OR4  port map(Z(0), Z(1), Z(2), Z(3), Z(8));
       or_10 :  OR4  port map(Z(4), Z(5), Z(6), Z(7), Z(9));

       or_11 :  OR2  port map(Z(8), Z(9), Z(10));

    inv_1 :  INV1 port map(Z(10), Zero);

    Negative  <= Sum(31);
    ALUResult <= Sum ;

end structural;



--------------------------------------------------
-- 32-bit adder
--
-- Function  :  Adds two 32 bit numbers.
--              Outputs a 32 bit number.
--
-- Author    :  lwkoh@cse
-- Date      :  10/09/2002
--
--------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity adder_32 is
    Port ( A : in std_logic_vector(31 downto 0);
           B : in std_logic_vector(31 downto 0);
           Cin : in std_logic;
           Sum : out std_logic_vector(31 downto 0));
end adder_32;

architecture structural of adder_32 is

component AND2 is
    Port ( in1 : in std_logic;
           in2 : in std_logic;
          out1 : out std_logic);
end component;

component AND3 is
    Port ( in1 : in std_logic;
           in2 : in std_logic;
           in3 : in std_logic;
          out1 : out std_logic);
end component;

component OR2 is
    Port ( in1 : in std_logic;
           in2 : in std_logic;
          out1 : out std_logic);
end component;
```

```
component OR3 is
    Port ( in1 : in std_logic;
           in2 : in std_logic;
           in3 : in std_logic;
          out1 : out std_logic);
end component;

component adder_16 is
    Port ( A : in std_logic_vector(15 downto 0);
           B : in std_logic_vector(15 downto 0);
           Cin : in std_logic;
           S : out std_logic_vector(15 downto 0);
           P_15_0 : out std_logic;
           G_15_0 : out std_logic);
end component;

-- internal propagate and generate functions
signal prop, gen : std_logic_vector(1 downto 0);

-- misc internal signals
signal c_1, c_2, c_3 : std_logic ;

begin

    adder16_0 : adder_16 port map(A(15 downto 0),  B(15 downto 0), Cin,
                                  Sum(15 downto 0),  prop(0), gen(0));
    adder16_1 : adder_16 port map(A(31 downto 16), B(31 downto 16), c_2,
                                  Sum(31 downto 16), prop(1), gen(1));

    and_1 : AND2 port map(prop(0), Cin, c_1);
    or_1  : OR2  port map(gen(0) , c_1, c_2);

end structural;


----------------------------------------------------
-- 16-bit adder
--
-- Function  :  Adds two 16 bit numbers.
--              Outputs a 16 bit number.
--
-- Author    :  lwkoh@cse
-- Date      :  10/09/2002
--
----------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity adder_16 is
    Port ( A : in std_logic_vector(15 downto 0);
           B : in std_logic_vector(15 downto 0);
           Cin : in std_logic;
           S : out std_logic_vector(15 downto 0);
           P_15_0 : out std_logic;
           G_15_0 : out std_logic);
end adder_16;
```

```
architecture structural of adder_16 is

component AND2 is
    Port ( in1 : in std_logic;
           in2 : in std_logic;
           out1 : out std_logic);
end component;

component AND3 is
    Port ( in1 : in std_logic;
           in2 : in std_logic;
           in3 : in std_logic;
           out1 : out std_logic);
end component;

component AND4 is
    Port ( in1 : in std_logic;
           in2 : in std_logic;
           in3 : in std_logic;
           in4 : in std_logic;
           out1 : out std_logic);
end component;

component OR2 is
    Port ( in1 : in std_logic;
           in2 : in std_logic;
           out1 : out std_logic);
end component;

component OR3 is
    Port ( in1 : in std_logic;
           in2 : in std_logic;
           in3 : in std_logic;
           out1 : out std_logic);
end component;

component OR4 is
    Port ( in1 : in std_logic;
           in2 : in std_logic;
           in3 : in std_logic;
           in4 : in std_logic;
           out1 : out std_logic);
end component;

component adder_4 is
    Port ( A : in std_logic_vector(3 downto 0);
           B : in std_logic_vector(3 downto 0);
           Cin : in std_logic;
           S : out std_logic_vector(3 downto 0);
           P_3_0 : out std_logic;
           G_3_0 : out std_logic);
end component;

-- internal propagate and generate functions
signal prop, gen : std_logic_vector(3 downto 0);
-- carry in for each alu_4 block
signal C : std_logic_vector(3 downto 0);
```

```
-- miscellaneous internal signals
signal C1_1, C2_1, C2_2, C3_1, C3_2, C3_3, G_1, G_2, G_3 : std_logic ;

begin

    adder4_0 : adder_4 port map(A(3 downto 0),   B(3 downto 0),   Cin,
                                S(3 downto 0),   prop(0), gen(0));
    adder4_1 : adder_4 port map(A(7 downto 4),   B(7 downto 4),   C(1),
                                S(7 downto 4),   prop(1), gen(1));
    adder4_2 : adder_4 port map(A(11 downto 8),  B(11 downto 8),  C(2),
                                S(11 downto 8),  prop(2), gen(2));
    adder4_3 : adder_4 port map(A(15 downto 12), B(15 downto 12), C(3),
                                S(15 downto 12), prop(3), gen(3));


       and_1 : AND2 port map(prop(0), Cin, C1_1);
       or_1  : OR2  port map(gen(0),  C1_1, C(1));

       and_2 : AND2 port map(prop(1), gen(0),  C2_1);
       and_3 : AND3 port map(prop(1), prop(0), Cin, C2_2);
       or_2  : OR3  port map(gen(1),  C2_1, C2_2, C(2));

       and_4 : AND2 port map(prop(2), gen(1),  C3_1);
       and_5 : AND3 port map(prop(2), prop(1), gen(0),  C3_2);
       and_6 : AND4 port map(prop(2), prop(1), prop(0), Cin, C3_3);
       or_3  : OR4  port map(gen(2),  C3_1, C3_2, C3_3, C(3));

       and_7 : AND4 port map(prop(3), prop(2), prop(1),
                             prop(0), P_15_0);

       and_8 : AND2 port map(prop(3), gen(2),  G_1);
       and_9 : AND3 port map(prop(3), prop(2), gen(1),  G_2);
       and_10: AND4 port map(prop(3), prop(2), prop(1), gen(0), G_3);
       or_4  : OR4  port map(gen(3),  G_1, G_2, G_3, G_15_0);

end structural;


--------------------------------------------------
-- 4-bit adder
--
-- Function  :  Adds two 4 bit numbers.
--              Outputs a 4 bit number.
--
-- Author    :  lwkoh@cse
-- Date      :  10/09/2002
--
--------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity adder_4 is
    Port ( A : in std_logic_vector(3 downto 0);
           B : in std_logic_vector(3 downto 0);
           Cin : in std_logic;
           S : out std_logic_vector(3 downto 0);
```

```
           P_3_0 : out std_logic;
           G_3_0 : out std_logic);
end adder_4;

architecture structural of adder_4 is

component AND2 is
    Port ( in1 : in std_logic;
           in2 : in std_logic;
           out1 : out std_logic);
end component;

component AND3 is
    Port ( in1 : in std_logic;
            in2 : in std_logic;
            in3 : in std_logic;
           out1 : out std_logic);
end component;

component AND4 is
    Port ( in1 : in std_logic;
            in2 : in std_logic;
           in3 : in std_logic;
            in4 : in std_logic;
           out1 : out std_logic);
end component;

component OR2 is
    Port ( in1 : in std_logic;
            in2 : in std_logic;
           out1 : out std_logic);
end component;

component OR3 is
    Port ( in1 : in std_logic;
            in2 : in std_logic;
           in3 : in std_logic;
           out1 : out std_logic);
end component;


component OR4 is
    Port ( in1 : in std_logic;
           in2 : in std_logic;
           in3 : in std_logic;
           in4 : in std_logic;
           out1 : out std_logic);
end component;


component PFA is  -- partial full adder
    Port (  A : in std_logic;
            B : in std_logic;
            C : in std_logic;
            S : out std_logic;    -- sum
            P : out std_logic;    -- propagate function
            G : out std_logic);   -- generate function
end component ;
```

```
-- internal propagate and generate functions
signal prop, gen : std_logic_vector(3 downto 0) ;
signal C : std_logic_vector(3 downto 0);  -- carry in for each bit


-- miscellaneous internal signals
signal C1_1, C2_1, C2_2, C3_1, C3_2, C3_3, G_1, G_2, G_3 : std_logic ;

begin

       PFA_0 : PFA port map(A(0), B(0), Cin,  S(0), prop(0), gen(0));
       PFA_1 : PFA port map(A(1), B(1), C(1), S(1), prop(1), gen(1));
       PFA_2 : PFA port map(A(2), B(2), C(2), S(2), prop(2), gen(2));
       PFA_3 : PFA port map(A(3), B(3), C(3), S(3), prop(3), gen(3));

       and_1 : AND2 port map(prop(0), Cin, C1_1);
       or_1  : OR2  port map(gen(0), C1_1, C(1));

       and_2 : AND2 port map(prop(1), gen(0), C2_1);
       and_3 : AND3 port map(prop(1), prop(0), Cin, C2_2);
       or_2  : OR3  port map(gen(1),  C2_1, C2_2, C(2));

       and_4 : AND2 port map(prop(2), gen(1),  C3_1);
       and_5 : AND3 port map(prop(2), prop(1), gen(0),  C3_2);
       and_6 : AND4 port map(prop(2), prop(1), prop(0), Cin, C3_3);
       or_3  : OR4  port map(gen(2),  C3_1, C3_2, C3_3, C(3));

       and_7 : AND4 port map(prop(3), prop(2), prop(1), prop(0), P_3_0);

       and_8 : AND2 port map(prop(3), gen(2),  G_1);
       and_9 : AND3 port map(prop(3), prop(2), gen(1), G_2);
       and_10: AND4 port map(prop(3), prop(2), prop(1), gen(0), G_3);
       or_4  : OR4  port map(gen(3),  G_1, G_2, G_3, G_3_0);


end structural;



-----------------------------------------------------
-- Primitive Gate Functions
--
-- Function  :  Provide all necessary basic gates
--              to construct higher level components.
--
-- Author    :  lwkoh@cse
-- Date      :  10/09/2002
--
-----------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity INV1 is
    Port ( in1 : in std_logic;
           out1 : out std_logic);
end INV1;
```

```
architecture behavioral of INV1 is
begin
    out1 <= (not in1) after 500 ps;
end behavioral;



library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity AND2 is
    Port ( in1 : in std_logic;
              in2 : in std_logic;
           out1 : out std_logic);
end AND2;

architecture behavioral of AND2 is
begin
    out1 <= (in1 and in2) after 500 ps  ;
end behavioral;



library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity AND3 is
    Port ( in1 : in std_logic;
              in2 : in std_logic;
                    in3 : in std_logic;
           out1 : out std_logic);
end AND3;

architecture behavioral of AND3 is
begin
    out1 <= (in1 and in2 and in3) after 500 ps ;
end behavioral;


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity AND4 is
    Port ( in1 : in std_logic;
              in2 : in std_logic;
                    in3 : in std_logic;
                    in4 : in std_logic;
           out1 : out std_logic);
end AND4;

architecture behavioral of AND4 is
begin
    out1 <= (in1 and in2 and in3 and in4) after 500 ps ;
end behavioral;
```

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

entity OR2 is
    Port ( in1 : in std_logic;
              in2 : in std_logic;
           out1 : out std_logic);
end OR2;

architecture behavioral of OR2 is
begin
    out1 <= (in1 or in2) after 500 ps ;
end behavioral;


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity OR3 is
    Port ( in1 : in std_logic;
              in2 : in std_logic;
                    in3 : in std_logic;
           out1 : out std_logic);
end OR3;

architecture behavioral of OR3 is
begin
    out1 <= (in1 or in2 or in3) after 500 ps ;
end behavioral;


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity OR4 is
    Port ( in1 : in std_logic;
              in2 : in std_logic;
                    in3 : in std_logic;
                    in4 : in std_logic;
           out1 : out std_logic);
end OR4;

architecture behavioral of OR4 is
begin
    out1 <= (in1 or in2 or in3 or in4) after 500 ps ;
end behavioral;




library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity XOR2 is
    Port ( in1 : in std_logic;
              in2 : in std_logic;
           out1 : out std_logic);
end XOR2;
```

```
architecture behavioral of XOR2 is
begin
    out1 <= (in1 xor in2) after 500 ps ;
end behavioral;



library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity PFA is  -- partial full adder
    Port (  A : in std_logic;
            B : in std_logic;
            C : in std_logic;
            S : out std_logic;    -- sum
            P : out std_logic;    -- propagate function
            G : out std_logic);   -- generate function
end PFA ;

architecture structural of PFA is

component AND2 is
    Port ( in1 : in std_logic;
            in2 : in std_logic;
           out1 : out std_logic);
end component;

component XOR2 is
    Port ( in1 : in std_logic;
            in2 : in std_logic;
           out1 : out std_logic);
end component;

signal propagate : std_logic ;

begin

        P <= propagate ;
        xor_1 : XOR2 port map(A, B, propagate);
       xor_2 : XOR2 port map(C, propagate, S);
       and_1 : AND2 port map(A, B, G);

end structural;
```

5.4.7 Simulation Waveforms

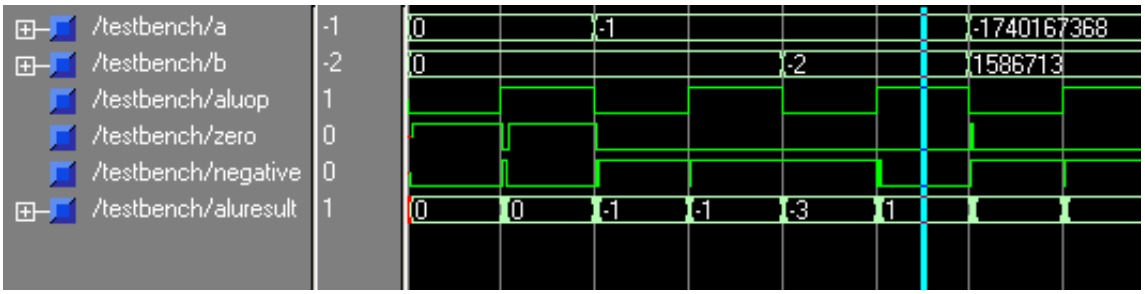


Figure 5-22: Simulation Waveform for 32-bit ALU

5.5 Program Counter (32 bit register)

5.5.1 Block Diagram

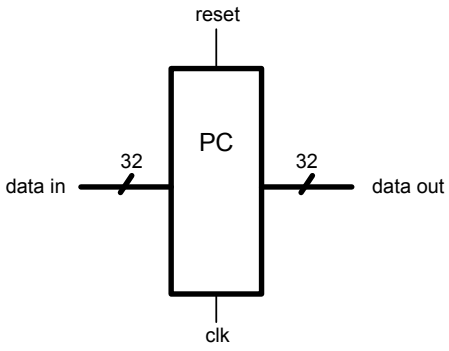


Figure 5-23: Program Counter

5.5.2 Interface

Input ports : data in - 32 bit input address / data
clk - clock input
reset - asynchronous reset (preload with address 0x00000000)

Output port : data out – 32 bit data from input port "data in" after a rising edge at "clk" input.

5.5.3 Function

data out ← data in (after rising edge of "clk")

This component is actually a 32 bit register that holds on to the value of the previous input until the next rising edge of the clock.

This component is used as both a PC as well as registers in the *register file*.

5.5.4 Implementation

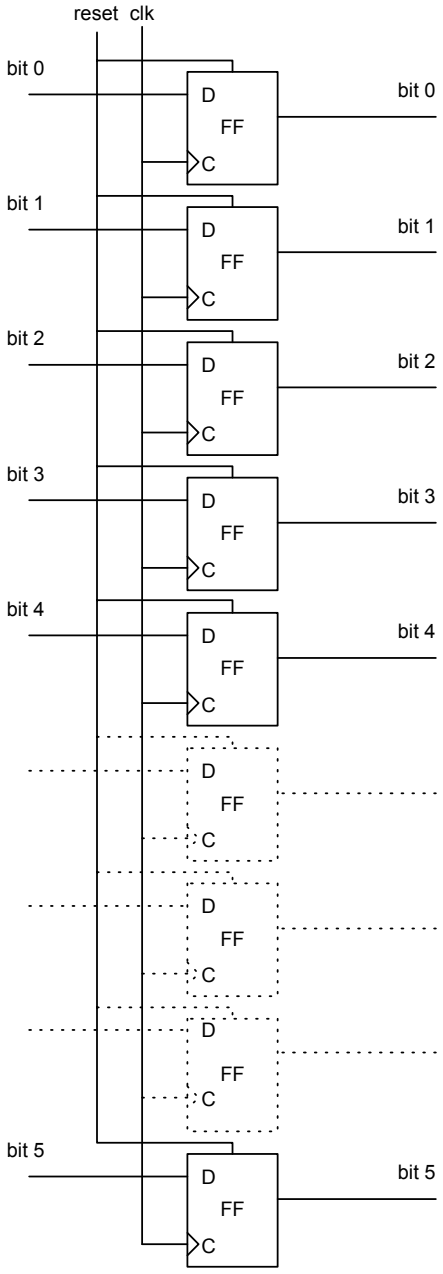


Figure 5-24: Implementation of the Program Counter (32-bit register)

5.5.5 Delay Justification

The flip-flop delays are assumed to be 0.5ns.

The 32 bit register (PC) contains 32 flip-flops in parallel. Thus the critical path includes only one flip-flop delay.

Delay of the 32 bit register(PC) = **0.5ns**

5.5.6 VHDL Model

```
----------------------------------------------------
-- Program Counter (32-bit register)
--
-- Function  :  Stores the given 32 bit value after
--              a clock pulse and holds it until the
--              clock cycle.
--
-- Author    :  senglin@cse
-- Date      :  10/09/2002
--
----------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity pc is
    Port ( addressIn : in std_logic_vector(31 downto 0);
           addressOut : out std_logic_vector(31 downto 0);
           clk : in std_logic;
                  reset : in std_logic);  -- asynchronous reset
end pc;

architecture Behavioral of pc is

begin

     process(reset, clk) is
     begin
        if (reset='1') then
             addressOut <= X"00000000" after 500 ps;
          elsif rising_edge(clk) then
               addressOut <= addressIn after 500 ps;
          end if;
     end process;
end Behavioral;
```

5.5.7 Simulation Waveforms

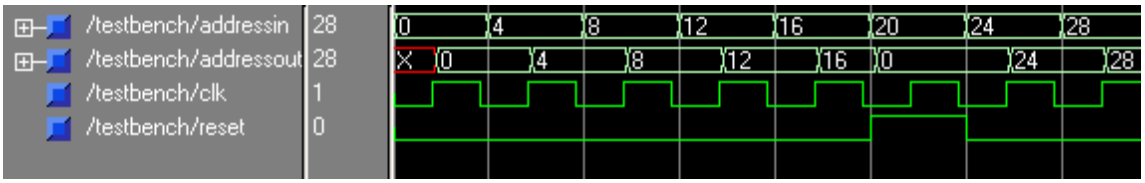


Figure 5-25: Simulation Waveform for Program Counter (PC)

5.6 Instruction Memory (32 x 32 bits)

5.6.1 Block Diagram

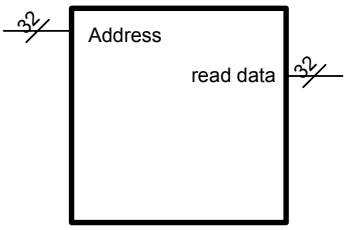


Figure 5-26: Instruction Memory

5.6.2 Interface

Input port : Address – 32 bit input address to determine the location of the data to be read.

Output port : read data – 32 bit data from the register at address of "Address".

5.6.3 Function

Output ← 32 bit data from Address

- Stores the required instruction to perform calculations
- The instruction is stored in instruction memory when started.

5.6.4 Implementation

The instruction memory is constructed using multiple levels of abstraction. It consists mainly of memory banks and a decoder. The following diagrams are used to illustrate the implementation of the instruction memory.

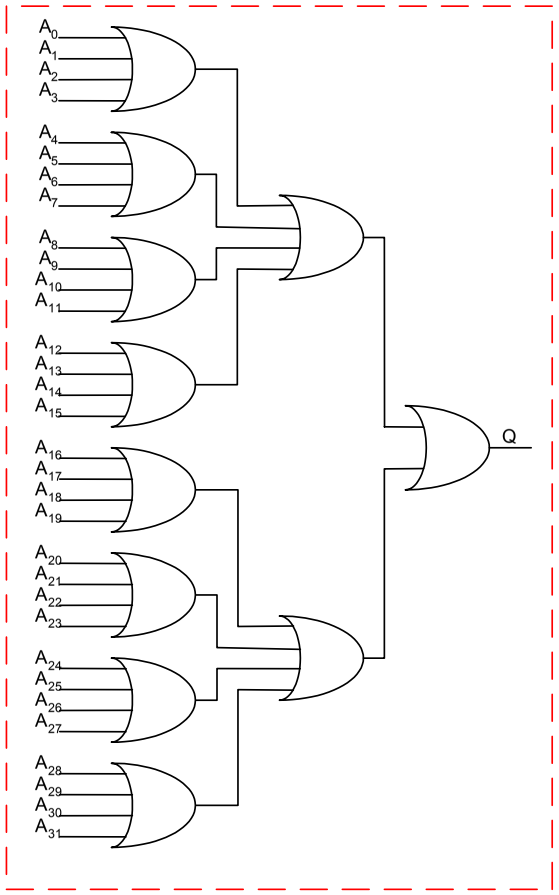
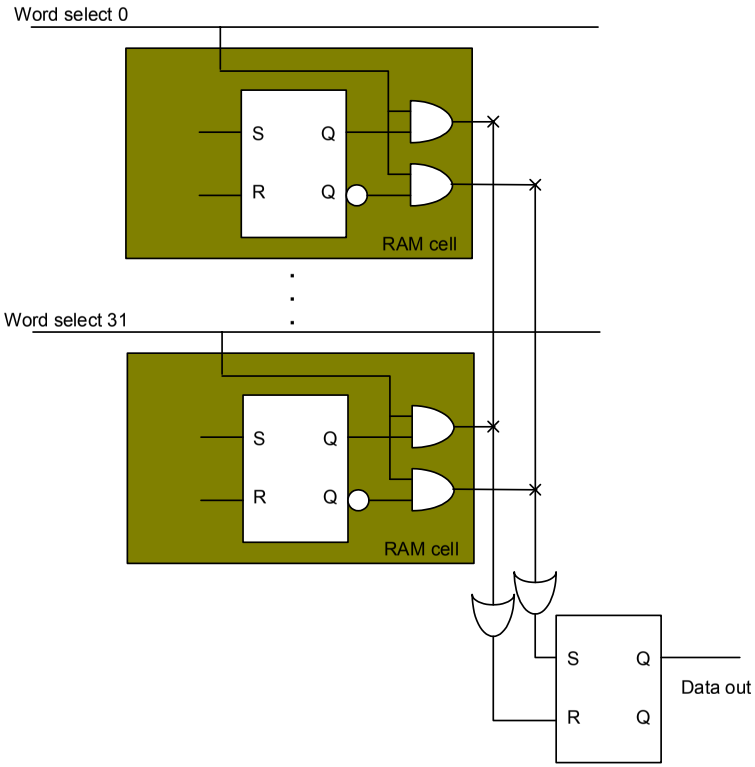


Figure 5-27: Implementation of the memory bank

GATE LEVEL IMPLEMENTATION OF A 32 INPUT OR-GATE

IMPLEMENTATION OF AN INSTRUCTION MEMORY (32 x 32 bits)



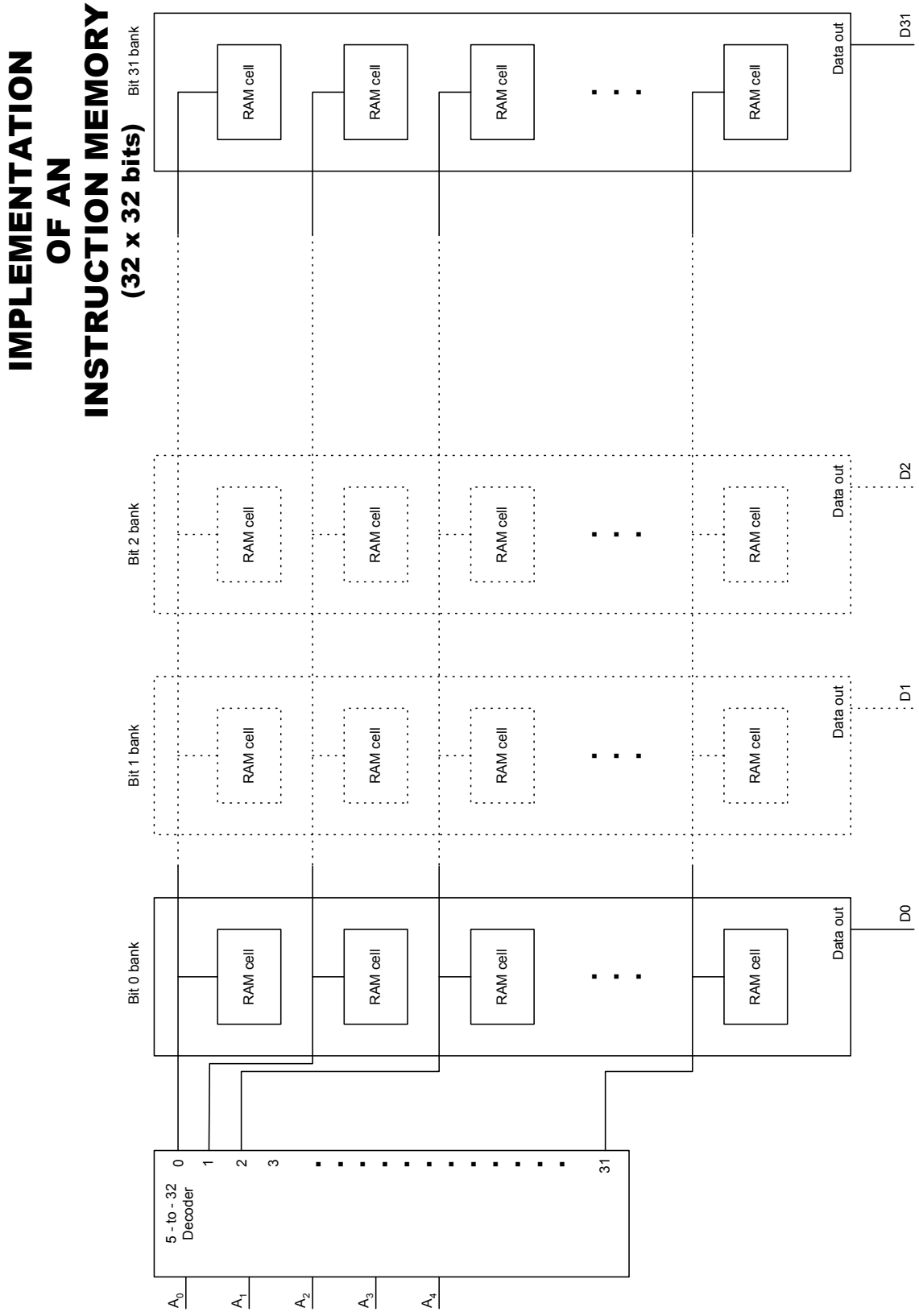


Figure 5-28: Implementation of the Instruction Memory

5.6.5 Delay Justification

The inertial gate and flip-flop delays are assumed to be 0.5ns.

Component delays :

32 input OR-Gate	= 0.5(or-gate) + 0.5(or-gate) + 0.5(or-gate) = 1.5ns
1 memory bank	= 0.5(and-gate) + 1.5(32 input or-gate) + 0.5(flip flop) + = 2.5ns
5 to 32 Decoder	= 0.5(not-gate) + 0.5(and-gate) + 0.5(and-gate) = 1.5ns
Data memory	= 1.5(5-to-32 decoder) + 2.5(memory bank) = **4.0ns**

5.6.6 VHDL Model

```
----------------------------------------------------
-- Instruction Memory (32 x 32 bits)
--
-- Function  :  Stores the required instructions to
--              execute the defined task.
--
-- Author    :  senglin@cse
-- Date      :  10/09/2002
--
----------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity instructionmem is
    Port ( addressIn : in std_logic_vector(31 downto 0);
           dataOut : out std_logic_vector(31 downto 0));
end instructionmem;

architecture Behavioral of instructionmem is
  type mem_array is array(0 to 31) of std_logic_vector(31 downto 0);
begin
  mem_process: process(addressIn)
    variable data_mem: mem_array := (
      -- values here are the instructions in HEXADECIMAL format
          0 => X"8C020000",
          1 => X"20000004",
          2 => X"8C030000",
          3 => X"8C040000",
          4 => X"58830001",
          5 => X"20830000",
          6 => X"20000004",
          7 => X"20210001",
          8 => X"1422FFFA",
          9 => X"20630064",
          10=> X"20A50001",
          11=> X"14A6FFFF",
          others => X"00000000");
      variable addr: integer;
  begin
    addr:=conv_integer(addressIn(6 downto 2));
    dataOut <= data_mem(addr) after 4 ns;
  end process;
end Behavioral;
```

5.6.7 Simulation Waveforms

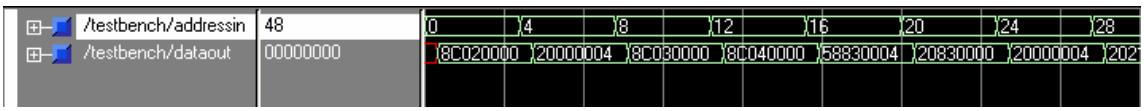


Figure 5-29: Simulation Waveform for the Instruction Memory

5.7 Data Memory (32 x 32 bits)

5.7.1 Block Diagram

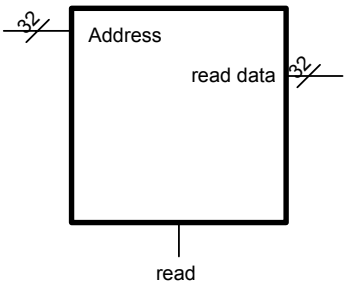


Figure 5-30: Data Memory of Size 32x32 bits

5.7.2 Interface

This component has

- 1 input of 32 bits wide for the location in memory
- 1 output of 32 bits wide for output of data stored in memory given the address.
- 1 read indicator line which activates reading of data memory.

5.7.3 Function

Read	Output
0	"00000000"
1	32 Bit data from address

Figure 5-31: Function Table for the Data Memory

- Stores the required data to perform calculations
- Memory is initialised with predefined values when started.
- "Read" should be asserted to read data from memory, otherwise "00000000" is returned.

5.7.4 Implementation

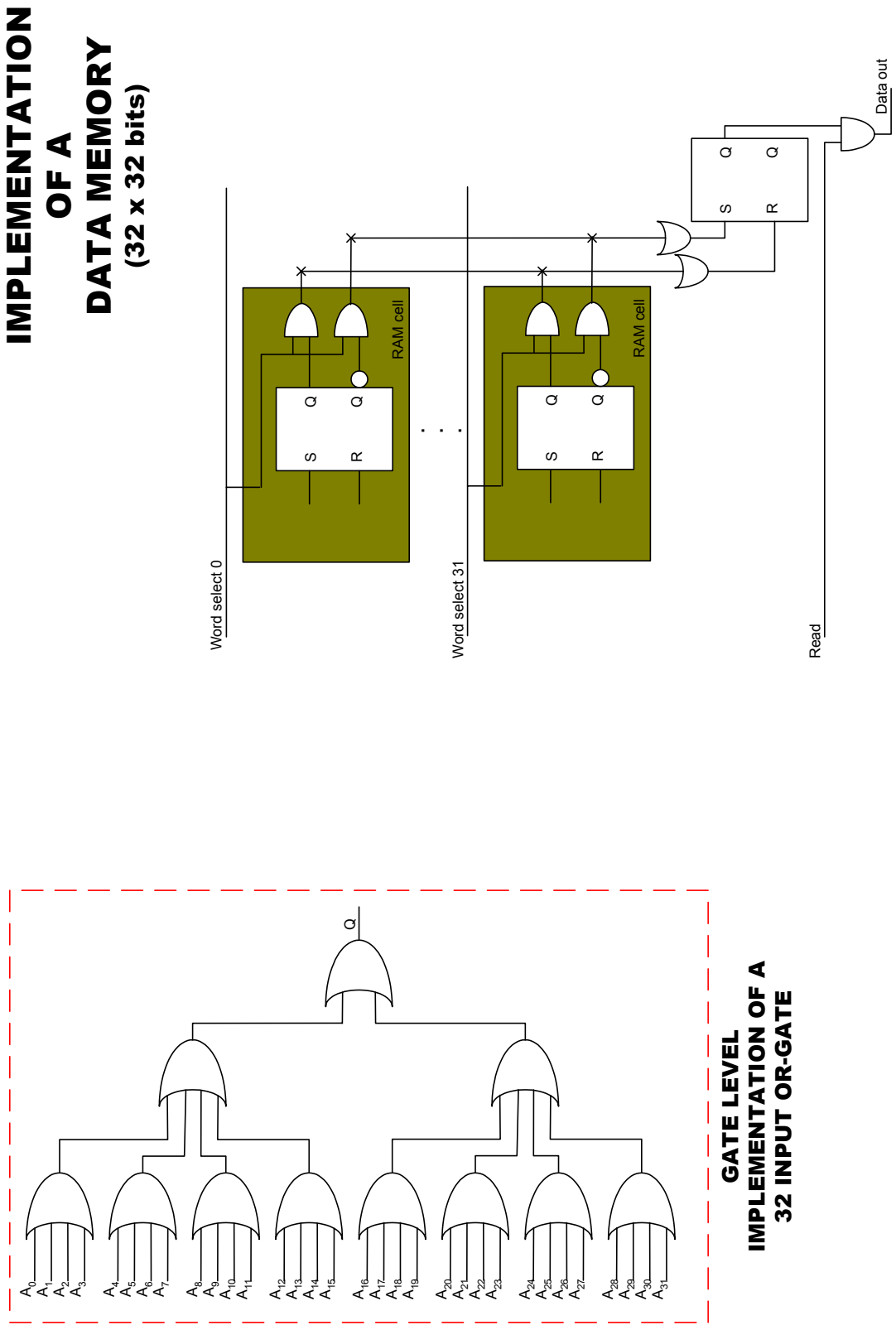


Figure 5-32 : Implementation of a Data Memory Block

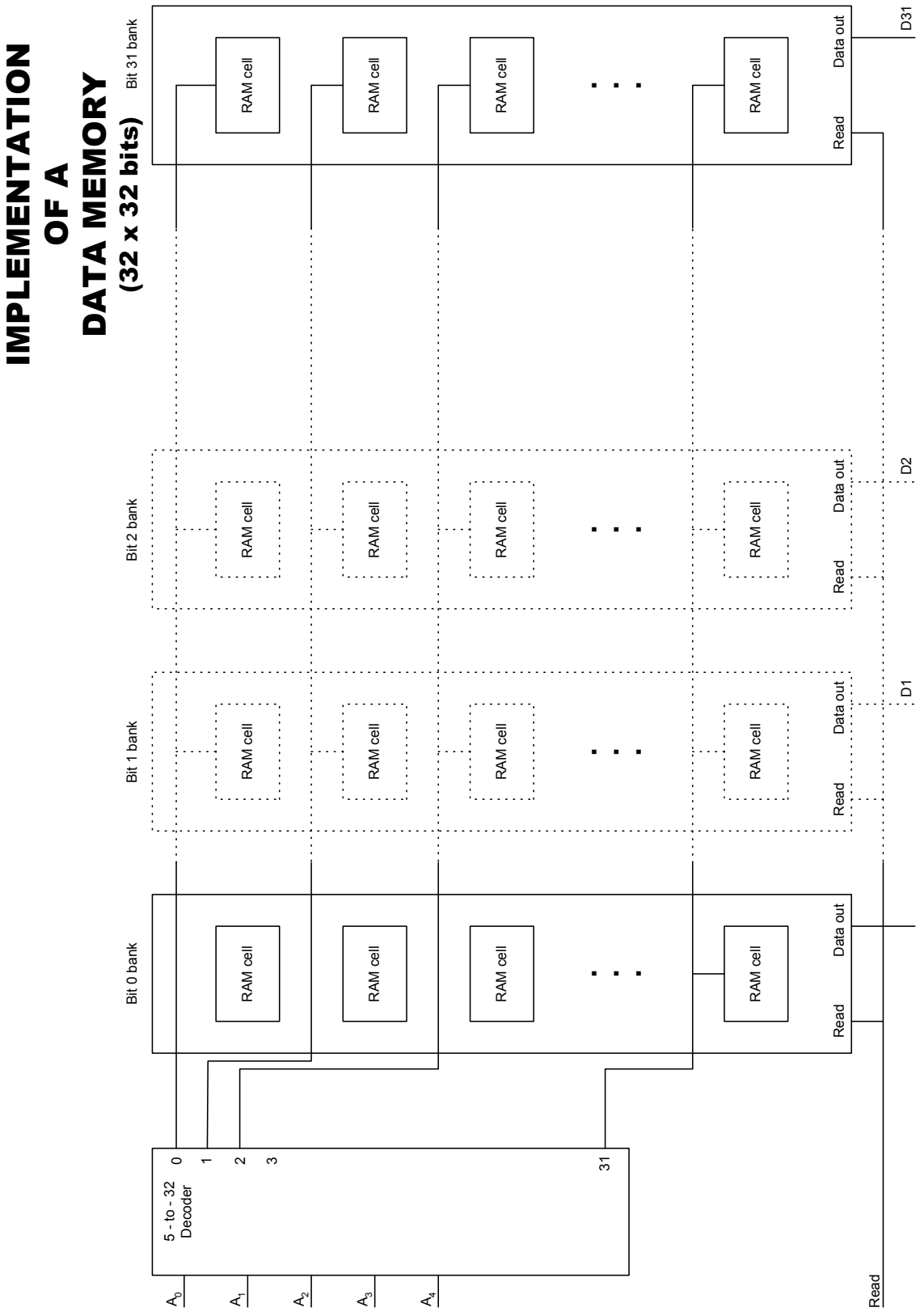


Figure 5-33 : Implementation of Data Memory

5.7.5 Delay Justification

The inertial gate and flip-flop delays are assumed to be 0.5ns.

Component delays :

32 input OR-Gate	= 0.5(or-gate) + 0.5(or-gate) + 0.5(or-gate) = 1.5ns
1 memory bank	= 0.5(and-gate) + 1.5(32 input or-gate) + 0.5(flip flop) + 0.5(and-gate) = 3.0ns
5 to 32 Decoder	= 0.5(not-gate) + 0.5(and-gate) + 0.5(and-gate) = 1.5ns
Data memory	= 1.5(5-to-32 decoder) + 3.0(memory bank) = **4.5ns**

5.7.6 VHDL Model

```
--------------------------------------------------
-- Data Memory (32 x 32 bits)
--
-- Function  :  Stores the predefined numbers to
--              operate on.
--              This memory is initialised at startup
--              and is not writable.
--
-- Author    :  senglin@cse
-- Date      :  10/09/2002
--
--------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity datamem is
    Port ( read_NotWrite : in std_logic;
              addressIn     : in std_logic_vector(31 downto 0);
           dataOut       : out std_logic_vector(31 downto 0));
end datamem;

architecture Behavioral of datamem is
  type mem_array is array(0 to 31) of std_logic_vector(31 downto 0);
begin
  data_process: process(addressIn, read_NotWrite)
    variable data_mem: mem_array := (
      -- change the values here to modify data in memory
          0 => X"00000005",
          1 => X"00000001",
          2 => X"00000003",
          3 => X"00000005",
          4 => X"00000004",
          5 => X"00000002",
        others => X"00000000");
      variable addr: integer;
  begin
    addr:=conv_integer(addressIn(6 downto 2));
    if read_NotWrite = '1' then
         dataOut <= data_mem(addr) after 4500 ps;
    else
        dataOut <= X"00000000" after 4500 ps;
    end if;
  end process;
end Behavioral;
```

5.7.7 Simulation Waveforms

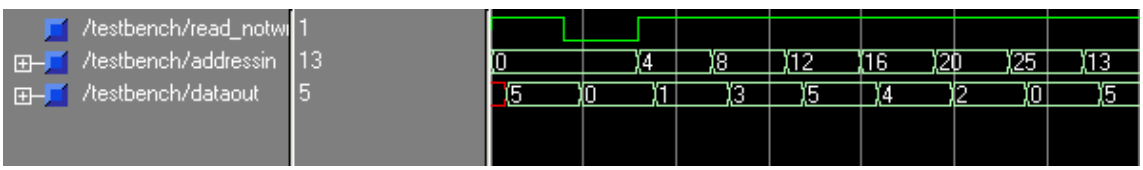


Figure 5-34: Simulation Waveform for the Data Memory

5.8 Register File (32 x 32)

5.8.1 Block Diagram

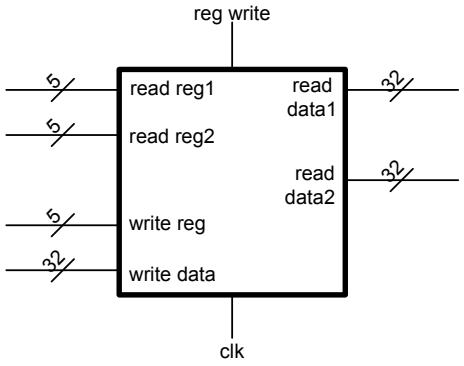


Figure 5-35: Register File

5.8.2 Interface

Input port : readreg1 -- address of register in register file to be read (read port 1)
readreg2 -- address of register in register file to be read (read port 2)
writereg -- address of register to be written (write port)
writedata -- input data to be written to register at address "writereg"
clk -- the clock input to govern the write update to register

Output port : readdata1 -- data in register at address "readreg1"
readdata2 -- data in register at address "readreg2"

5.8.3 Function

This component provides fast acess memory for computation purposes.

The functions are as follows :
Output at "readdata1" ← data at address "readreg1"
Output at "readdata2" ← data at address "readreg2"

When regwrite is asserted, data at "writedata" will be written to register address "writereg" at the next arriving clock edge.

5.8.4 Implementation

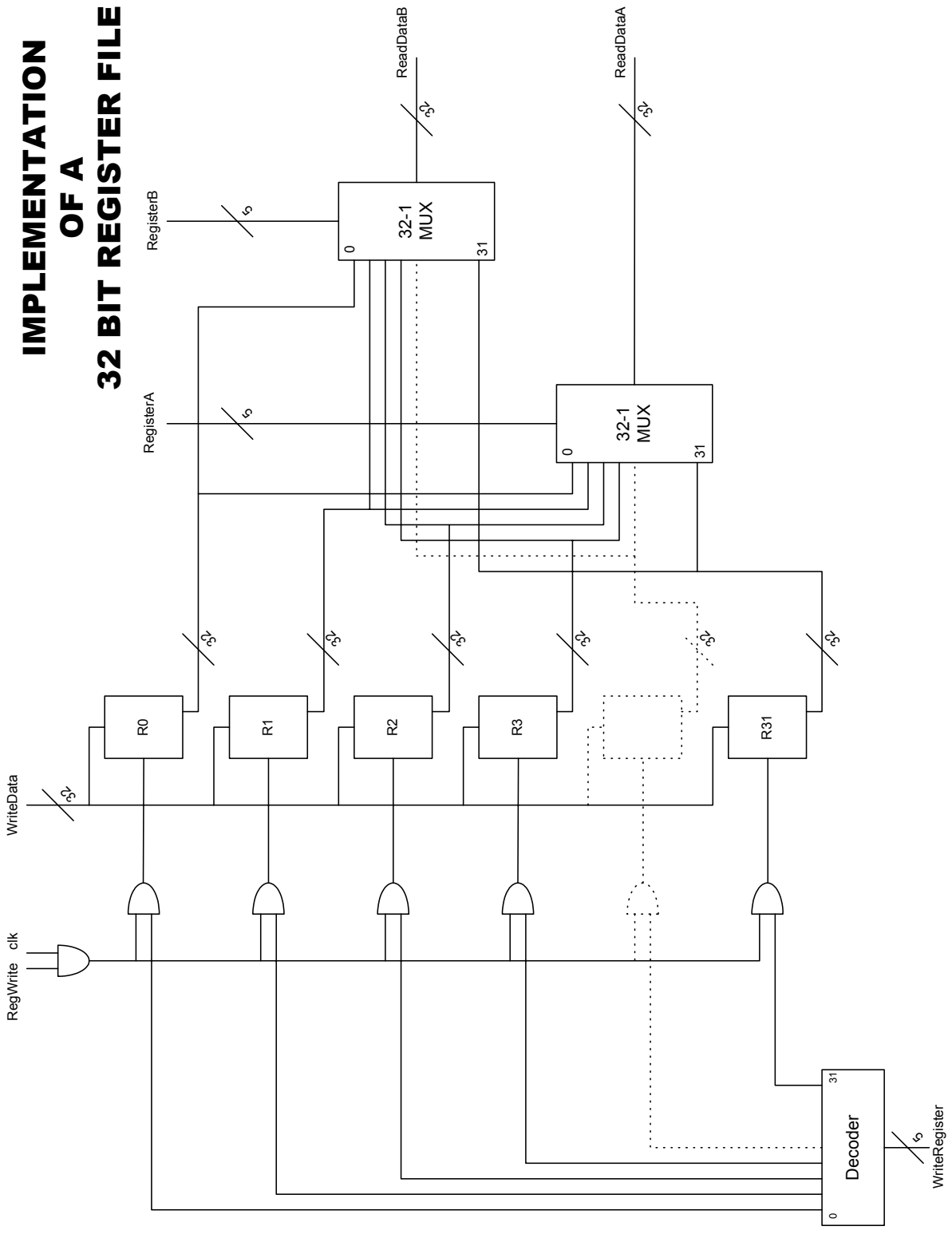


Figure 5-36 : Implementation of a 32 bit Register File

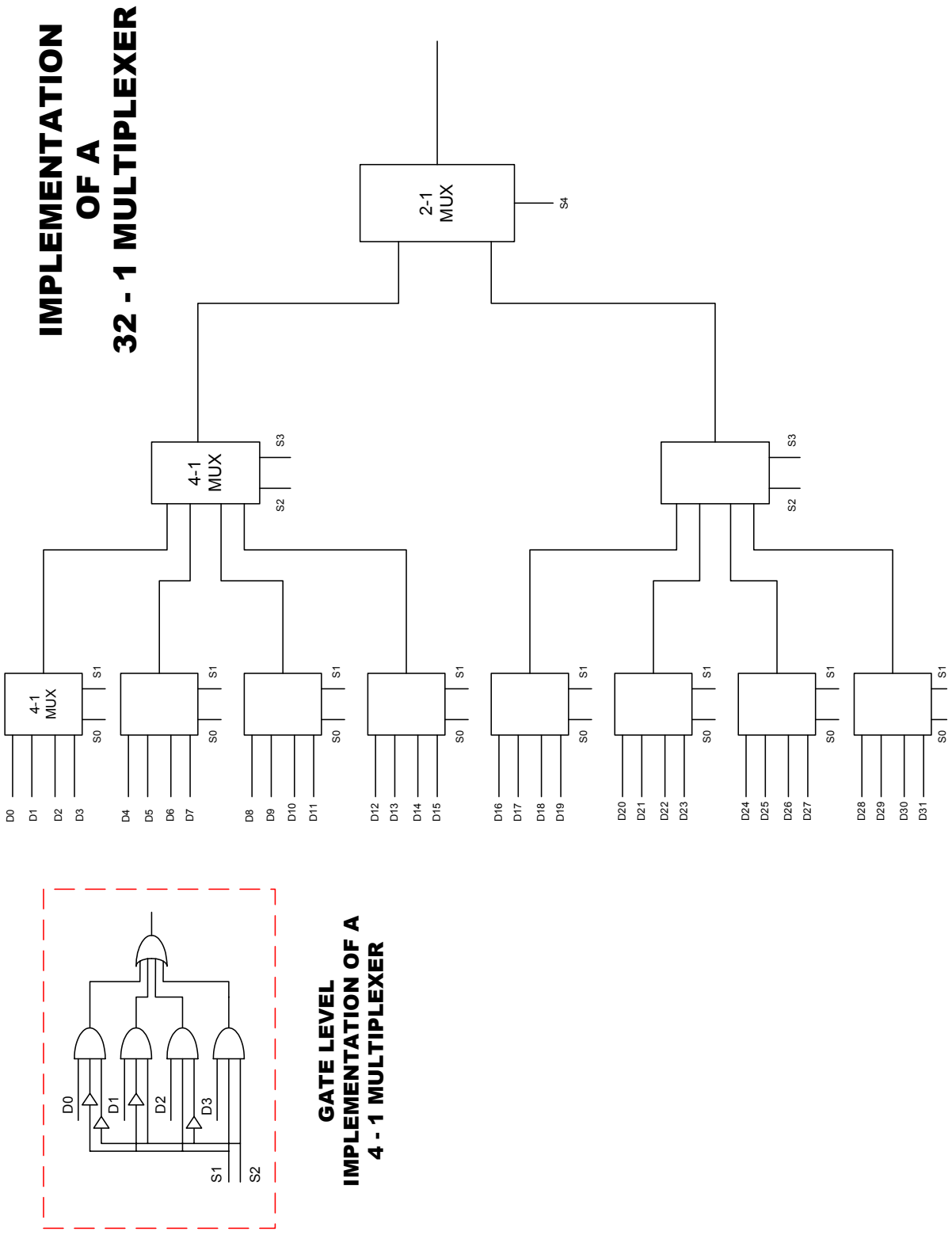


Figure 5-37 : Implementation of a 32 to 1 Multiplexer

GATE LEVEL IMPLEMENTATION OF A 5 - 32 DECODER

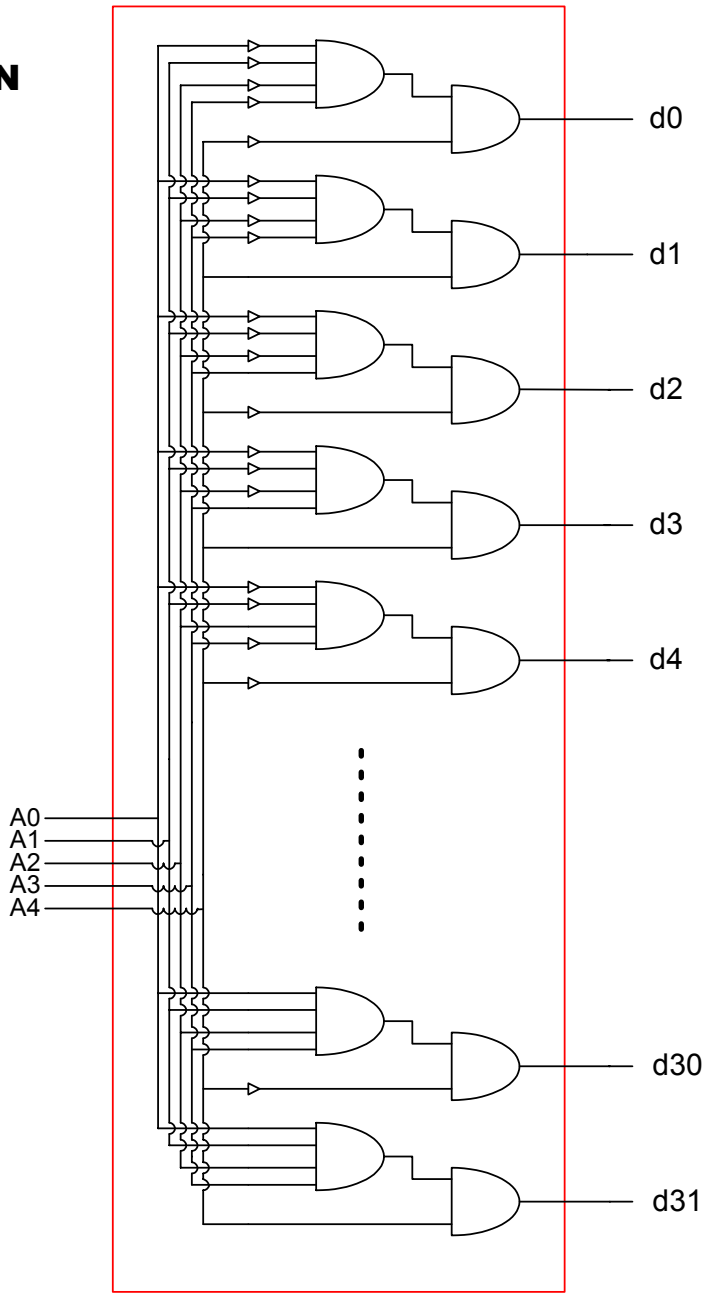


Figure 5-38 : Implementation of a 5-to-32 Decoder

5.8.5 Delay Justification

The inertial gate and flip-flop delays are assumed to be 0.5ns.

Component delays :

5 to 32 Decoder	= 0.5(not-gate) + 0.5(and-gate) + 0.5(and-gate) = 1.5ns
4 to 1 Multiplexer	= 0.5(not-gate) + 0.5(and-gate) + 0.5(or-gate) = 1.5ns
32 to 1 Multiplexer	= 2 x 1.5(4 to 1 Multiplexer) + 1.5(2 to 1 Multiplexer) = 4.5ns
32 bit Register	= 0.5ns (D-Flip Flop)
32 bit Register file	= 1.5(5 to 32 Decoder) + 0.5(not-gate) + 0.5(32 bit Register) + 4.5(32 to 1 Multiplexer) = **7ns**

5.8.6 VHDL Model

```
-----------------------------------------------------
-- Register File (32 x 32)
--
-- Function  :  Provides fast memory to store temporary
--              variables during calculation operations.
--
-- Author    :  senglin@cse
-- Date      :  10/09/2002
--
-----------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity regfile is
    Port ( RegisterA : in std_logic_vector(4 downto 0);
           RegisterB : in std_logic_vector(4 downto 0);
           WriteRegister : in std_logic_vector(4 downto 0);
           WriteData : in std_logic_vector(31 downto 0);
           ReadDataA : out std_logic_vector(31 downto 0);
           ReadDataB : out std_logic_vector(31 downto 0);
           RegWrite : in std_logic;
                  clk      : in std_logic);
end regfile;
```

```
architecture Behavioral of regfile is

  type mem_array is array(0 to 63) of std_logic_vector(31 downto 0);

begin
  mem_process: process(RegisterA, RegisterB, WriteRegister, WriteData,
                       RegWrite, clk)
      variable data_mem: mem_array := (others => X"00000000");
       variable addrA: integer;
       variable addrB: integer;
       variable addrW: integer;

  begin
      addrA:=conv_integer(RegisterA(4 downto 0));
       addrB:=conv_integer(RegisterB(4 downto 0));
       ReadDataA <= data_mem(addrA) after 7 ns;
       ReadDataB <= data_mem(addrB) after 7 ns;

       if (RegWrite = '1'  and rising_edge(clk)) then
           addrW:=conv_integer(WriteRegister(4 downto 0));
            data_mem(addrW) := WriteData;
       end if;

  end process;

end Behavioral;
```

5.8.7 Simulation Waveforms

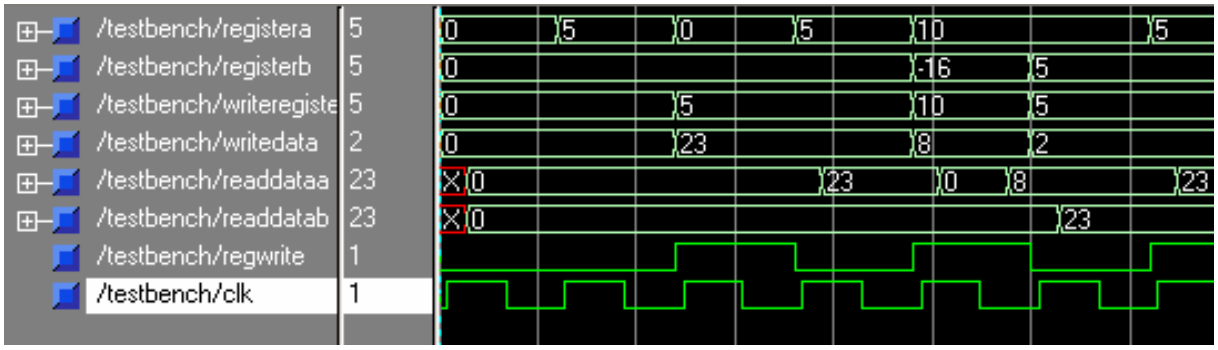


Figure 5-39: Simulation Waveform for the Register File

5.9 Control Logic

5.9.1 Block Diagram

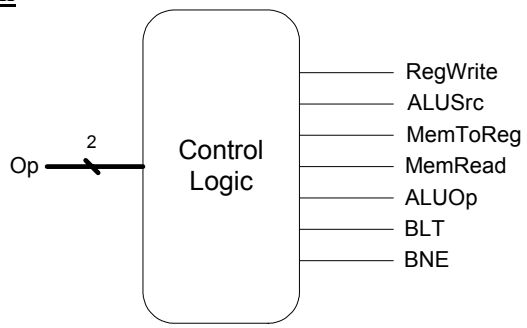


Figure 5-40 : Control Logic Block

5.9.2 Interface

Input port : *Op* -- 2-bit input signal used to decode the different instructions where control signals take values based on the instruction being executed.

Output ports : *RegWrite* -- Only when *RegWrite* is '1', a register specified will have its content updated with the provided write data.

ALUSrc -- When *ALUSrc* is '0', content of a register from the read port 2 of the register file will be loaded into the second input of the arithmetic logic unit (ALU).

When *ALUSrc* is '1', a sign-extended immediate value will be loaded instead.

MemToReg -- When *MemToReg* is '1', data from the data memory is provided as the *WriteData* for the register file.

When *MemToReg* is '0', the result from the ALU is provided as the *WriteData* instead.

MemRead -- Only when *MemRead* is '1', data from the data memory, specified by a provided memory address, will be output from the data memory.

ALUOp -- When *ALUOp* is '0', an addition is performed in the ALU.

When *ALUOp* is '1', a subtraction is performed in the ALU.

BLT -- *BLT* takes on value '1' only when a conditional branch on less than is executed.

BNE -- *BNE* takes on value '1' only when a conditional branch on not equal is executed.

5.9.3 Function

This block implements the control logic of the machine, to govern the actions of the datapath for each instruction. The following truth table shows the respective control signals for each of the 4 instructions used in this machine. Each of these control signals, with the functions described as above, is implemented using primitive gates.

	Opcode (Hex)	Opcode (Binary)	regDest	ALUSrc	memToReg	regWrite	memRead	ALUOp	BLT	BNE
lw	0x23	100011	0	1	1	1	1	0	0	0
addi	0x08	001000	0	1	0	1	0	0	0	0
blt	0x16	010110	X	0	X	0	0	1	1	0
bne	0x05	000101	X	0	X	0	0	1	0	1

Figure 5-41: Truth Table for the Seven Control Signals in This Machine

5.9.4 Implementation

Based on the truth table presented earlier, logic equations are derived for each of the seven control signals. *Instruction*[28] and *Instruction*[27] from the 32-bit instructions are enough to decode the 4 different instructions used here. Renaming these to Op[1] and Op[0] respectively, we obtain the following:

$$\text{RegWrite} = \text{ALUSrc} = \overline{\text{Op[1]}} \bullet \overline{\text{Op[0]}} + \overline{\text{Op[1]}} \bullet \text{Op[0]}$$

$$\text{MemToReg} = \text{MemRead} = \overline{\text{Op[1]}} \bullet \text{Op[0]}$$

$$\text{ALUOp} = \text{Op[1]} \bullet \overline{\text{Op[0]}} + \text{Op[1]} \bullet \text{Op[0]}$$

$$\text{BLT} = \text{Op[1]} \bullet \text{Op[0]}$$

$$\text{BNE} = \overline{\text{Op[1]}} \bullet \text{Op[0]}$$

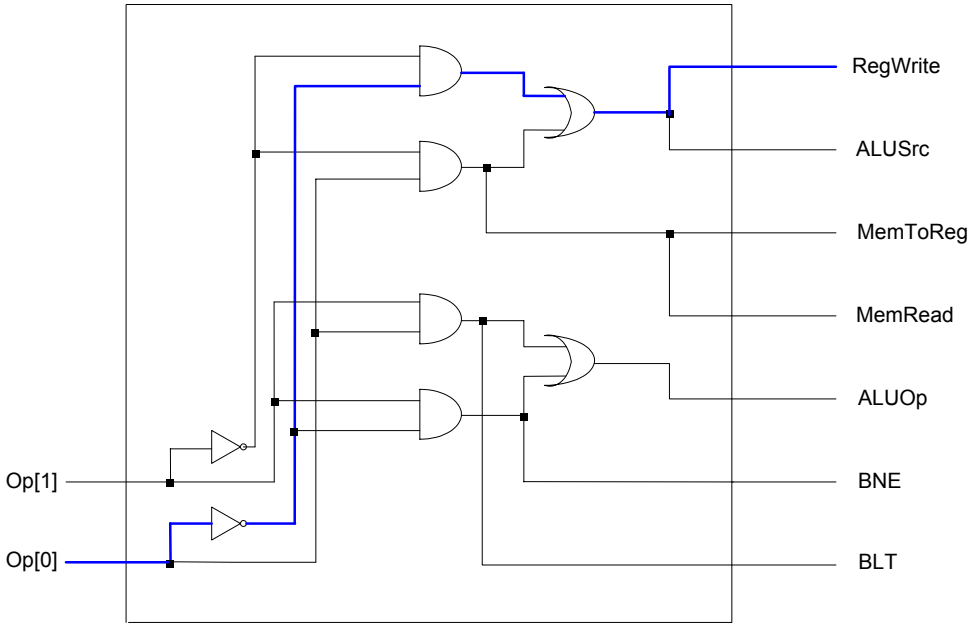


Figure 5-42 : Implementation of the Control Logic Block

5.9.5 Delay Justification

Assuming the (inertial) delay of a gate is 0.5 ns, one of the critical paths for the control logic block is highlighted in the figure above and contributes to 3 gate delays, i.e. 3 x (0.5ns) = **1.5ns**.

5.9.6 VHDL Model

```
--------------------------------------------------
-- Control Unit
--
-- Function  :  Selects the proper control lines
--              for the data path.
--
-- Author    :  lwkoh@cse
-- Date      :  10/09/2002
--
--------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity control is
    Port (  Op       : in std_logic_vector(1 downto 0);
            regWrite : out std_logic;
            ALUSrc   : out std_logic;
            MemToReg : out std_logic;
            MemRead  : out std_logic;
            ALUOp    : out std_logic;
            BLT      : out std_logic;
            BNE      : out std_logic);
end control;

architecture structural of control is

component INV1 is
    Port ( in1 : in std_logic;
           out1 : out std_logic);
end component;

component AND2 is
    Port ( in1 : in std_logic;
            in2 : in std_logic;
           out1 : out std_logic);
end component;

component OR2 is
    Port ( in1 : in std_logic;
            in2 : in std_logic;
           out1 : out std_logic);
end component;

signal nOp : std_logic_vector(1 downto 0) ;
signal minterm0, minterm1, minterm2, minterm3 : std_logic;
signal sig01, sig23 : std_logic ;

begin

     inv_1   : INV1 port map(Op(1), nOp(1));
inv_2   : INV1 port map(Op(0), nOp(0));
```

```
     nA_nB : AND2 port map(nOp(1), nOp(0), minterm0);
     nA_B  : AND2 port map(nOp(1),  Op(0), minterm1);
     A_nB  : AND2 port map( Op(1), nOp(0), minterm2);
     A_B   : AND2 port map( Op(1),  Op(0), minterm3);

     control1 : OR2 port map(minterm0, minterm1, sig01);
     control2 : OR2 port map(minterm2, minterm3, sig23);

     regWrite <= sig01 ;
     ALUSrc   <= sig01 ;
     ALUOp    <= sig23 ;

     MemToReg <= minterm1 ;
     MemRead  <= minterm1 ;

     BLT      <= minterm3 ;
     BNE      <= minterm2 ;

end structural;


-------------------------------------------------------
-- Primitive Gate Functions
-- Function : Provide all necessary basic gates
--            to construct higher level components.
--
-- Author   : lwkoh@cse
-- Date     : 10/09/2002
--
-------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity INV1 is
    Port ( in1 : in std_logic;
           out1 : out std_logic);
end INV1;

architecture behavioral of INV1 is
begin
    out1 <= (not in1) after 500 ps;
end behavioral;



library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity AND2 is
    Port ( in1 : in std_logic;
              in2 : in std_logic;
           out1 : out std_logic);
end AND2;

architecture behavioral of AND2 is
begin
    out1 <= (in1 and in2) after 500 ps  ;
end behavioral;
```

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

entity OR2 is
    Port ( in1 : in std_logic;
              in2 : in std_logic;
           out1 : out std_logic);
end OR2;

architecture behavioral of OR2 is
begin
    out1 <= (in1 or in2) after 500 ps ;
end behavioral;
```

5.9.7 Simulation Waveforms

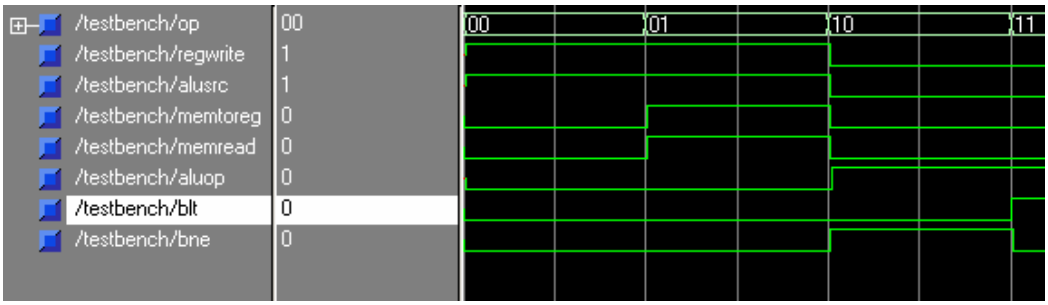


Figure 5-43: Simulation Waveform for the Control Logic Block

6 VHDL Simulation and Verification for this Machine

6.1 VHDL Model

```
--------------------------------------------------
-- The Single-Cycle RISC Microprocessor
--
-- Function  :  Adds 100 to the maximum if a set of
--              integers stored in data memory.
--
-- Author    :  lwkoh@cse
-- Date      :  10/09/2002
--
--------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity processor is
    Port (
         CLK : in std_logic;
           RESET : in std_logic;

     -- the following output signals are for observation only
           PCAddress : out std_logic_vector(31 downto 0);
           Branch    : out std_logic;
           Instruction : out std_logic_vector(31 downto 0);
           regWrite : out std_logic;
           ALUSrc   : out std_logic;
           MemToReg : out std_logic;
           MemRead  : out std_logic;
           ALUOp    : out std_logic;
           BLT      : out std_logic;
           BNE      : out std_logic;
           ReadReg1 : out std_logic_vector(4 downto 0);
           ReadReg2 : out std_logic_vector(4 downto 0);
           WriteReg : out std_logic_vector(4 downto 0);
           WriteData : out std_logic_vector(31 downto 0);
           ReadData1 : out std_logic_vector(31 downto 0);
           ReadData2 : out std_logic_vector(31 downto 0);
           Immediate : out std_logic_vector(31 downto 0);
           ALUData1  : out std_logic_vector(31 downto 0);
           ALUData2  : out std_logic_vector(31 downto 0);
           ALUResult : out std_logic_vector(31 downto 0);
           Zero      : out std_logic;
           Negative  : out std_logic;
           MemAddress : out std_logic_vector(31 downto 0);
           MemData    : out std_logic_vector(31 downto 0)
    );
end processor;
```

```
architecture mixed of processor is

  component INV1 is
    Port ( in1 : in std_logic ;
           out1 : out std_logic
      );
  end component ;

  component AND2 is
    Port ( in1 : in std_logic ;
           in2 : in std_logic ;
           out1 : out std_logic
      );
  end component ;

  component OR2 is
    Port ( in1 : in std_logic ;
           in2 : in std_logic ;
           out1 : out std_logic
      );
  end component ;

  component pc is
    Port ( addressIn : in std_logic_vector(31 downto 0);
           addressOut : out std_logic_vector(31 downto 0);
           clk : in std_logic;
           reset : in std_logic);
  end component;

  component alu_32 is
    Port (
          A : in std_logic_vector(31 downto 0);
           B : in std_logic_vector(31 downto 0);
           ALUOp : in std_logic;
           Zero  : out std_logic;
           Negative : out std_logic;
           ALUResult : out std_logic_vector(31 downto 0));
  end component;

  component control is
    Port ( Op       : in std_logic_vector(1 downto 0);
          regWrite : out std_logic;
           ALUSrc   : out std_logic;
           MemToReg : out std_logic;
           MemRead  : out std_logic;
           ALUOp    : out std_logic;
           BLT      : out std_logic;
           BNE      : out std_logic);
  end component;

  component datamem is
    Port ( read_NotWrite : in std_logic;
           addressIn     : in std_logic_vector(31 downto 0);
           dataOut       : out std_logic_vector(31 downto 0));
  end component;
```

```
component instructionmem is
  Port ( addressIn : in std_logic_vector(31 downto 0);
         dataOut : out std_logic_vector(31 downto 0));
end component;

component mux_2_1_32 is
  Port ( DataA : in std_logic_vector(31 downto 0);
         DataB : in std_logic_vector(31 downto 0);
         Sel : in std_logic;
         DataOut : out std_logic_vector(31 downto 0));
end component;

component regfile is
  Port ( RegisterA : in std_logic_vector(4 downto 0);
         RegisterB : in std_logic_vector(4 downto 0);
         WriteRegister : in std_logic_vector(4 downto 0);
         WriteData : in std_logic_vector(31 downto 0);
         ReadDataA : out std_logic_vector(31 downto 0);
         ReadDataB : out std_logic_vector(31 downto 0);
         RegWrite : in std_logic;
         clk      : in std_logic);
end component;

component shifter_32 is
  Port ( DataIn : in std_logic_vector(31 downto 0);
         DataOut : out std_logic_vector(31 downto 0));
end component;

component signExtend_16_32 is
  Port ( DataIn : in std_logic_vector(15 downto 0);
         DataOut : out std_logic_vector(31 downto 0));
end component;


-- internal signals
signal PCAddressIn, PCAddressOut, InstructionOut,
       writeDataBuff, readData1Buff, readData2Buff,
       ImmediateBuff, ALUSrcB, ALUResultBuff, MemDataBuff,
       PC_Add4_Address, PC_Branch_Address, ShiftedOffset :
          std_logic_vector(31 downto 0);

signal regWriteSig, ALUSrcSig, MemToRegSig, MemReadSig,
          ALUOpSig, BLTSig, BNESig,ZeroSig, NegativeSig,
          PCSrc, PCSrc1, PCSrc2, nZeroSig : std_logic ;

signal zeroes, extra1, extra2, extra3, extra4 : std_logic ;

signal four : std_logic_vector(31 downto 0);
```

```
begin

     zeroes <= '0' ;
     four <= X"00000004" ;

     program_counter :
             pc port map (PCAddressIn, PCAddressOut,
                          CLK, RESET);

     instruction_memory :
             instructionmem port map (PCAddressOut,
                                      InstructionOut);

     control_logic :
             control port map (InstructionOut(28 downto 27),
                               regWriteSig, ALUSrcSig,
                               MemToRegSig, MemReadSig,
                               ALUOpSig, BLTSig, BNESig);

    register_file :
             regfile port map (InstructionOut(25 downto 21),
                               InstructionOut(20 downto 16),
                               InstructionOut(20 downto 16),
                               writeDataBuff,
                               readData1Buff, readData2Buff,
                               regWriteSig, CLK);

    sign_extender :
      signExtend_16_32 port map (InstructionOut(15 downto 0),
                                 ImmediateBuff);

     ALU : alu_32 port map (readData1Buff, ALUSrcB, ALUOpSig,
                            ZeroSig, NegativeSig, ALUResultBuff);

     data_memory : datamem port map (MemReadSig,
                                     ALUResultBuff,
                                     MemDataBuff);

     PC_add4  : alu_32 port map (PCAddressOut,
                                 four, zeroes,
                                 extra1, extra2,
                                 PC_Add4_Address);

     PC_branch : alu_32 port map (PC_Add4_Address,
                                  ShiftedOffset, zeroes,
                                  extra3, extra4,
                                  PC_Branch_Address);

      left_shifter : shifter_32 port map (ImmediateBuff,
                                          ShiftedOffset);

      ALU_mux : mux_2_1_32 port map (readData2Buff,
                                     ImmediateBuff,
                                     ALUSrcSig, ALUSrcB);
```

```
        mem_mux : mux_2_1_32 port map (ALUResultBuff,
                                       MemDataBuff,
                                       MemReadSig,
                                       writeDataBuff);

        PC_mux  : mux_2_1_32 port map (PC_Add4_Address,
                                       PC_Branch_Address,
                                       PCSrc, PCAddressIn);

       branch_1 : AND2 port map(NegativeSig, BLTSig, PCSrc1);
        branch_2 : INV1 port map(ZeroSig, nZeroSig);
        branch_3 : AND2 port map(nZeroSig, BNESig, PCSrc2);
        branch_4 : OR2  port map(PCSrc1, PCSrc2, PCSrc);

        PCAddress   <= PCAddressOut ;
        Branch      <= PCSrc ;
       Instruction <= InstructionOut ;
       RegWrite    <= regWriteSig ;
        ALUSrc      <= ALUSrcSig ;
        MemToReg    <= MemToRegSig ;
        MemRead     <= MemReadSig ;
        ALUOp       <= ALUOpSig ;
        BLT         <= BLTSig ;
        BNE         <= BNESig ;
        Zero        <= ZeroSig ;
        Negative    <= NegativeSig ;
        Immediate   <= ImmediateBuff ;
        ALUData1    <= readData1Buff ;
        ALUData2    <= ALUSrcB ;
        ALUResult   <= ALUResultBuff ;
        ReadReg1    <= InstructionOut(25 downto 21);
        ReadReg2    <= InstructionOut(20 downto 16);
        WriteReg    <= InstructionOut(20 downto 16);
       ReadData1   <= readData1Buff ;
        ReadData2   <= readData2Buff ;
        WriteData   <= writeDataBuff ;
        MemAddress  <= ALUResultBuff ;
        MemData     <= MemDataBuff ;

end mixed;
```

6.2 VHDL Simulation

Strategy for simulation:

- Inputs to the developed machine are the "clk" and "reset" lines which are connected to the program counter. "clk" is also connected to the register file to control the write update of the registers.
- To obtain a snapshot of data lines in the CPU, the specific data lines of interest are connected to output lines (probing lines).
- Simulation is allowed to run for more than 4000ns using the property dialog box available from MicroSim XE.

Initiliased values in data memory:

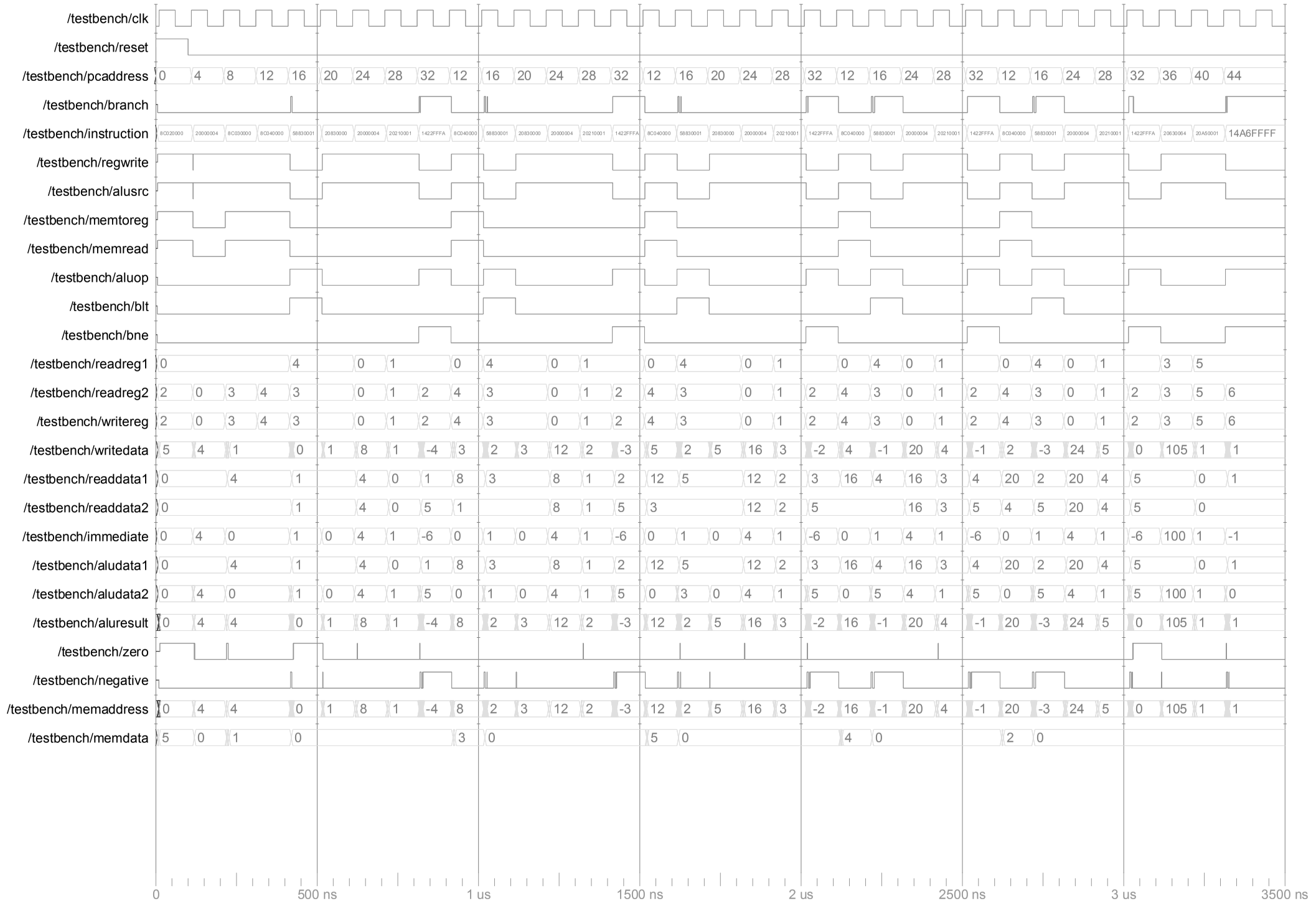
#number of elements to process	0x05	Address 0x0000
#value 1	0x01	Address 0x0004
#value 3	0x03	Address 0x0008
#value 5	0x05	Address 0x000C
#value 4	0x04	Address 0x0010
#value 2	0x02	Address 0x0014

Data output from memory can be seen from the waveform on the ***/testbench/memdata*** lines.

From the values stored in memory as stated aboved, based on the specification of the assignment, the number 100 should be added to 5 (largest integer in memory).

Verification:
The waveforms on the following page illustrates the process of finding the largest number and adding the number 100 to it.

As can be seen from the waveforms, when the program counter (***/testbench/pcaddress***) is 36, the immediate value (***testbench/immediate***) of 100 is added to the number 5 (***stored in register R3***) and the sum (105) is stored (***testbench/writedata***) back into the same register.



Entity:testbench Architecture:testbench_arch Date: Fri Sep 13 03:22:02 AUS Eastern Daylight Time 2002 Row: 1 Page: 1

7 Performance Analysis

Instruction Class	*Functional Units Used By The Instruction Class*				
Add Immediate	Instruction fetch	Register access	ALU	Register Access	Register Access
Load word	Instruction fetch	Register access	ALU	MemoryAccess	
Branch if not equal	Instruction fetch	Register access	ALU	Branch block	
Branch if less than	Instruction fetch	Register access	ALU	Branch block	

Figure 7-1: Functional Units Used Sorted By Instruction Class

Based on the critical execution paths in Section 4, we compute the required execution time for each instruction class.

Note:

- With 0.2 ns delay between 2 blocks
- Delay times are used as calculated from previous sections

ADD IMMEDIATE :

– Program counter	=	0.5 ns
		0.2 ns
– Instruction fetch from instruction memory	=	4.0 ns
		0.2 ns
– Read from register file	=	7.0 ns
		0.2 ns
– ALU Operation	=	17.5 ns
		0.2 ns
– Multiplexer	=	1.5 ns
		0.2 ns +
	=	**31.5 ns**

LOAD WORD :

– Program counter	=	0.5 ns
		0.2 ns
– Instruction fetch from instruction memory	=	4.0 ns
		0.2 ns
– Read index from register file	=	7.0 ns
		0.2 ns
– Addtion through ALU	=	17.5 ns
		0.2 ns
– Read from data	=	4.5 ns
		0.2 ns
– Multiplexer	=	1.5 ns
		0.2 ns +
	=	**36.2 ns**

(to reach input of “write data” at RegWrite)

BRANCH IF NOT EQUAL :

– Program counter	=	0.5 ns
		0.2 ns
– ALU Operation, PC + 4	=	17.5 ns
		0.2 ns
– ALU Operation, adding offset to get new value for PC	=	17.5 ns
		0.2 ns
– Multiplexer	=	1.5 ns
		0.2 ns +
	=	**37.8 ns**

BRANCH IF LESS THAN :

– Program counter	=	0.5 ns
		0.2 ns
– ALU Operation, PC + 4	=	17.5 ns
		0.2 ns
– ALU Operation, adding offset to get new value for PC	=	17.5 ns
		0.2 ns
– Multiplexer	=	1.5 ns
		0.2 ns +
	=	**37.8 ns**

As our CPU is a single cycle RISC machine ,

CPI = 1

The clock cylce for a machine with a single clock for all instructions will be **37.8 ns** (the longest instruction time)

Thus, we decided to allow a clock speed of **20 Mhz** (**50 ns** clock cycle time)