Multi-cycle MIPS Architecture with Wishbone Interface
- ECE485 Final Project -

IVAN DARIO CASTELLANOS

***Abstract*— A 32-bit MIPS Architecture based microprocessor with Wishbone interface to memory is implemented. The microprocessor supports a predefined set of microinstructions including type R, type I and type J instructions. This set includes different arithmetic and logic functions, as well as memory access instructions, branches and jumps. The communication between the microprocessor and the main memory is implemented with a Wishbone Interface. The complete design was implemented using VHDL. Final results and simulations supporting each of the microinstructions behavior are included at the end of this report.**

I. INTRODUCTION

THE main objective of this project is to implement a 32-Bit Microprocessor that supports a defined set of the MIPS Instruction Set Architecture. The microprocessor is implemented entirely in VHDL. The functions allowed are: NOP (No operation), AND, OR, XOR, NOR, ADD, SUBTRACT and Set on Less Than. These functions can be accessed via a type R instruction that has the two operands in the Register File (RF) or via a type I instruction in which one of the operands is the data in one register in the RF and the other is contained within the instruction itself. In addition to these instructions the ISA implemented includes: Load Upper Immediate (higher 16 bits in a 32 bit word), Load Word (RF register = Memory data), Store Word (Memory position = RF data), Branch when Equal and Not Equal (BEQ, BNE), Jump, Jump Register and Jump and Link.

The microprocessor implemented is based on the Multi-cycle datapath explained in the class text book[1] and the distributed class example, the single cycle datapath in VHDL. This Multi-cycle datapath had a limited support with respect to the microinstructions desired and therefore needed modifications in order to support them. The memory size is 512 x 32bits and it's assumed to be in Princeton format, i.e. same memory for both instructions and data. To comply with the project requirements the first 256 words are used only for instructions and the lower 256 words for data, however all the memory can be addressed by the program.

Exception support was also required in the microprocessor. Given the required ISA, the exception functions capabilities are limited. Two different types of exceptions can be handled. The first one occurs when the microinstruction code is invalid meaning the instruction is incorrect. The other occurs when a type R or type I instruction results in an overflow, as can happen with addition and subtraction functions.

One final requirement of the project is to implement a Wishbone interface between the microprocessor and the memory. The Wishbone architecture permits the connection of circuit functions together in a simple, efficient and portable way. It is an open core architecture that is widely spread and utilized and hence its implementation in this project. The Wishbone architecture specifies two or more circuits that communicate through an interconnection interface called Intercon[2]. There are different types of Wishbone Interconnections and the one implemented is a simple Master/Slave point to point Intecon. The details of the implementation will be explained later in this report.

[1] J. Hennessy and D. Patterson, Computer organization and Design. Morgan Kaufmann Publishers, San Francisco, 1998.
[2] Silicore Corporation Website, "Wishbone Frequently Asked Questions" http://www.silicore.net

II. DATAPATH DESCRIPTION AND IMPLEMENTATION

As stated before, the datapath is based on the class textbook's multi-cycle datapath. The resulting datapath after modifications is shown in the next page in figure 1. The determining factor on how to modify the datapath depends completely on the Instruction Set to be supported. This is defined in the Project Requirements and is also shown in table 1. The components that were designed and modified are explained in the following sections.

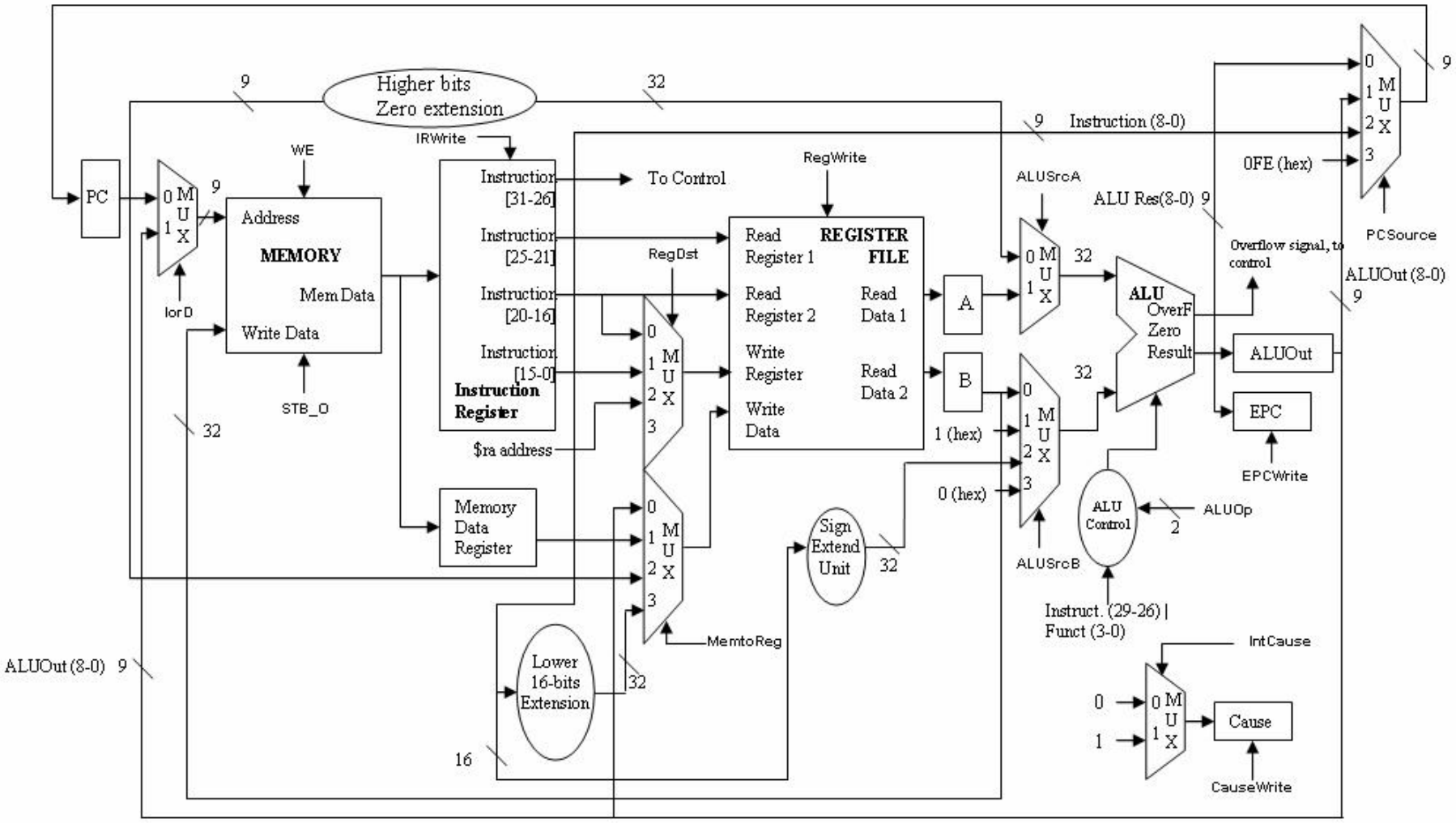
Opcode (31 : 26)	Function (5 : 0)	Instruction	Meaning
000000	000000	NOP	No Operation
000000	100100	AND	and $1, $2, $3
000000	100101	OR	or $1, $2, $4
000000	100110	XOR	xor $1, $2, $5
000000	100111	NOR	nor $1, $2, $6
000000	100000	Add	add $1, $2, $7
000000	100010	Subtract	subtract $1, $2, $8
000000	101010	Set Less Than	slt $1, $2, $9
001100	xxxxxx	AND imm.	andi $1, $2, 100
001101	xxxxxx	OR imm.	ori $1, $2, 101
001110	xxxxxx	XOR imm.	xori $1, $2, 102
001000	xxxxxx	Add imm.	addi $1, $2, 103
001001	xxxxxx	Subtract imm.	subi $1, $2, 104
001010	xxxxxx	Set Less Than imm.	slti $1, $2, 105
100011	xxxxxx	Load Word	lw $1, 100($2)
001111	xxxxxx	Load Upper Imm.	lui $1, 100
101011	xxxxxx	Store Word	sw $1, 100($2)
000100	xxxxxx	Branch Equal	beq $1, $2, 100
000101	xxxxxx	Branch Not Equal	bnq $1, $2, 100
000010	xxxxxx	Jump	j 100
000011	xxxxxx	Jump and Link	jal 100
000000	001000	Jump Register	jr rs

Table 1. Instruction Set requirements[3].

A. 32-Bit ALU

The 32-Bit ALU must support the different logic and arithmetic functions required. It was based in the class homework were a 32 bit ALU was designed. Only a slight modification was needed in order for it to support the NOR function. All other functions were supported using the single bit ALUs designed and the Most Significant Bit ALU. The inputs of the ALU are: both 32 bit operands (A and B) and a 3 bit control line, "operation" (Refer to table 2). The design uses the 9 gates Full Adder to implement all logic functions without additional gates. Only an inverter is needed to obtain the NOR result. The VHDL file of the bit_ALU, msb_ALU and the 32-Bit ALU is included in Appendix B.

[3] J. Stine, "ECE485 Final Project, Multi-cycle MIPS Architecture with Wishbone Interface", Illinois Institute of Technology, 2003



Multi-cycle Datapath. All control signals are in bold and in this letter type.

Figure 1. Multi-cycle Datapath. All control signals are in bold and in this **letter type** (write enable signals for the PC are explained in figure 2).

B. ALU Control

The initial ALU control had to be modified as well since there is support for additional functions. Specifically the microprocessor accepts now Type I instructions which vary the format in which the textbook's ALU control receives the operation to be performed. In this case the function field of the micro-instruction (Instruction [5-0]) doesn't determine the type of operation to be performed. This information is contained instead in the Opcode (Instruction [31-26]). The ALU Control receives also a two bit input from the main control (ALUOp) which forces the ALU to do Subtract (for BEQ, BNE and exception support), Add (Memory location operations, PC update), or the operation required by the type R or type I instruction. Another possible approach was to change the operation codes of the type R instructions in order to have fewer inputs into the ALU control (just opcode field) but this approach was avoided since it would restrict the microprocessor to be used with the MIPs instructions.

Not all bits of the Opcode or the function are necessary for the ALU control to determine which operation to perform. Table 1 reveals that the ALUControl might be implemented with only the Opcode [3-0] (Instruction[29-26]) and the Function [3-0] bits, i.e. 8 bits instead of the complete 12 (Opcode [5-0], Function [5-0]). Table 2 shows the behavior of the ALUControl.

ALUOp (From control)	Opcode[3-0] (Instruction [29-26])	Function[3-0] (Instruction [3-0])	To ALU (Operation)	Operation Performed	
00	xxxx	xxxx	010	Add	
x1	xxxx	xxxx	110	Subtract	
1x	0000	0000	010	Add	Type R
1x	0000	0010	110	Subtract	
1x	0000	0100	000	AND	
1x	0000	0101	001	OR	
1x	0000	0110	100	XOR	
1x	0000	0111	101	NOR	
1x	0000	1010	111	slt	
1x	1000	xxxx	010	Addi	Type I
1x	1001	xxxx	110	Subtracti	
1x	1010	xxxx	111	slti	
1x	1100	xxxx	000	ANDi	
1x	1101	xxxx	001	ORi	
1x	1110	xxxx	100	XORi	

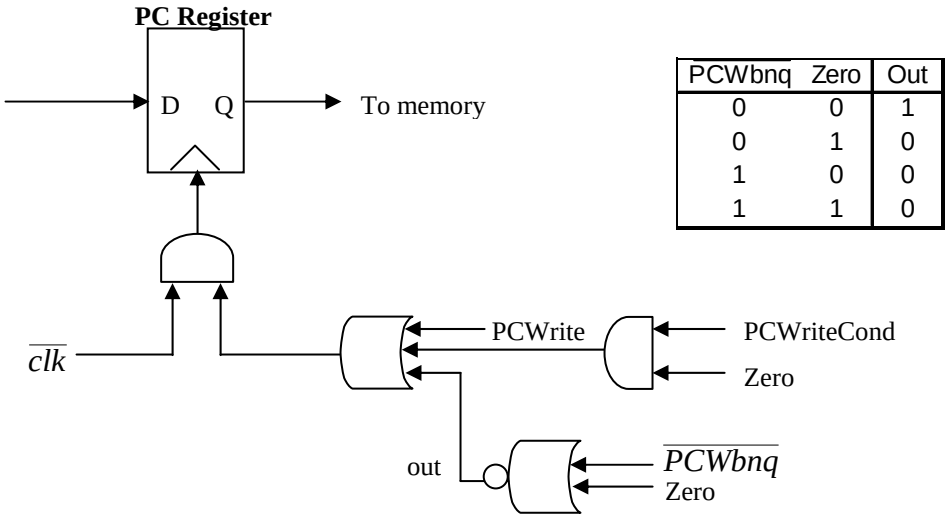
Table 2. ALUControl unit inputs and 3 bit Output.

Espresso was used to derive the logic functions that implement the ALUControl. These functions can be seen in the VHDL file of this unit, alucontrol.vhd, included in Appendix B.

C. Branch if Not Equal (BNQ) microinstruction support

The example datapath in the project definition[4] did not support this function and therefore a modification was needed. The previous datapath supported the BEQ instruction when the control sets the PCWriteCond bit high. This allows the PC register to be updated if the ALU zero signal is asserted, indicating that the subtraction is zero and both numbers are equal. In order to support BNQ the function shown in figure 2 was implemented.

[4] J. Stine, "ECE485 Final Project, Multi-cycle MIPS Architecture with Wishbone Interface", pg. 6. Illinois Institute of Technology, 2003



$\overline{PCWbnq}$	Zero	Out
0	0	1
0	1	0
1	0	0
1	1	0

Figure 2. Support for BNQ microinstruction.

$\overline{PCWbnq}$ is always set high by the control. Only when the BNQ instruction is performed the main control unit sets this line to zero. In this way the output of the NOR gate is only high when the 'zero' signal from the ALU is equal to 0, indicating that both values subtracted are not equal; permitting the branch to take place on the falling edge of the clock.

D. LUI Instruction Support

The LUI instruction is supported by incrementing the size of the Register File 'Write Data' multiplexor. This multiplexor was initially a 2 to 1, 32 bit multiplexor. It is changed for a 4 to 1 multiplexor. Input number 3 of the mux is determined by the Instruction[15-0] field (see figure 1). The LUI operation loads the immediate value from this field in the upper 16 bits of a desired register. Therefore a "Low Zero Extend" unit is implemented to place zeros in the lower 16 bits of the 32 bit word, and place the 16 bits from the instruction in the upper 16 bits of the word. The selection signal for the mux is therefore 2 bits wide, MemtoReg. The Main Control Unit performs the instruction fetch (IF) and then continues to the next state by setting this signal to '11' and asserting the RF write enable bit.

E. Jump and Link, Jump Register microinstruction support

To support the Jump Register instruction the ALU B Mux is hardwired to a value of 0 in its third input (see figure 1). This input was previously set by the "shift left 2" unit but its no longer needed since the word format is defined to be of 32 bits instead of 4 8-bit words; avoiding the necessity to shift the address left by 2. During the Execution phase, EX, the Main control sets the select input of Mux B to 3. The PCSource Mux select bit is set to 0 as well, the ALU operation is add and the PCWrite is enabled. The result is that the input A of the ALU is added to zero and the result ($rs content) is fed to the PC.

The Jump and Link instruction requires further modifications. The RF Write Register Mux is incremented to a 4-1 32bit mux and has input number 2 hardwired to the value of $ra (assumed to be 31dec, or 11111b). The RF Write Data mux is fed in its input 2 the output of the program counter (which during the Instruction Decode phase, ID, is PC+1). The main control therefore sets the RF write enable bit, sets the Write Register mux select input to 2 and sets the Write Data mux select input to 2. This actions, during ID phase, write the value of PC+1 to register $ra and then continuing to the Execution phase in the same way as the jump instruction already supported, allowing the PC update and performing the jump. See figure 1.

F. MEMORY AND WISHBONE INTERFACE

The project definition required a 32bit x 256 word memory for instructions and 32bit x 256 word memory for data. A single memory was therefore implemented for simplicity, with the upper 256 words designated for instructions and the lower 256 for data. Nevertheless, all the memory may be addressed by the program. Since the memory is fixed to 512 words it only requires 9 bit for its address input. The PCSource mux, Memory Address mux and the PC register were therefore downsized to 9 bits as well, instead of the unnecessary 32 bits.

The interface between the microprocessor and the memory is implemented with the Wishbone Interface. As explained before, there are different possible Interconnections available but the one chosen for simplicity is the Master/Slave point to point Intercon.

The Wishbone interface that was implemented was based on the Wishbone library for VHDL[5]. This library specifies the Wishbone interface as seen in figure 3 and figure 4 for a point to point connection Intercon.

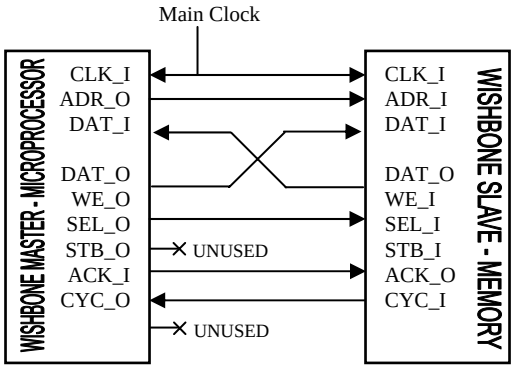


Figure 3. Wishbone Point to point Intercon.

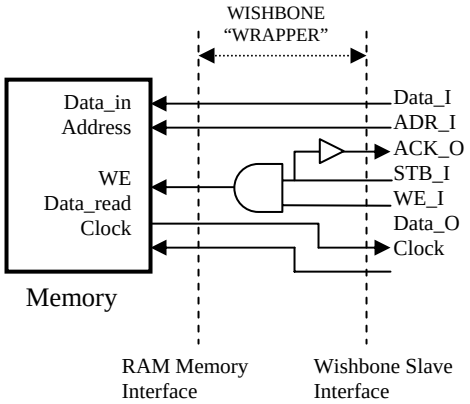


Figure 4. Wishbone Memory wrapper (to connect to Wishbone interface).

Figure 3 shows two important signals that the main control unit must handle. These are the Strobe Output (STB_O) and the Acknowledge (ACK). The Strobe Output indicates a valid data transfer cycle. The Slave must assert the Acknowledge signal in response, indicating a successful communication over the bus. The main control is designed therefore to wait until the ACK_I signal is received indicating a successful memory access operation. If the ACK_I signal is not asserted then the main control was defined to wait until this signal arrives.

Figure 4 shows the "Wishbone Wrapper" proposed in [3]. This circuit will allow the memory to be compatible with a Wishbone Architecture interface as desired. The Acknowledge signal is always asserted when an incoming Strobe signal is detected. This might imply that the main control "waiting" state might not be necessary but nevertheless it is implemented in order to maintain a true Wishbone compatibility. The WE bit is also ANDed with the Strobe bit to allow memory access only when the STB bit is asserted.

The memory VHDL File (mem512.vhd) is included in Appendix B as well as the "wrapper" (memwb.vhd).

G. Main Control Unit

[5] Silicore Corporation, "WISHBONE Public domain library for VHDL" technical reference manual, Minnesota, 2001.

The Main Control unit is the microprocessor element that required the most design effort and time. The project definition required the Main Control unit to be implemented using a ROM memory instead of a logic gate and register FSM. The ROM data width is equal to all control outputs required to control the datapath. The total number of control signals necessary is 22. Each word in the memory will represent a state denoting each of the phases of the instruction execution, IF, ID, EX, MEM and WB. It is important to remember that not all phases are used by all instructions; some can require all 5 stages while others will be ready in 2. The state words are stored in the ROM in a way that facilitates their addressing by some control logic according to the instruction to be performed. It is important to note as well that the main control designed has the Wishbone control interface built in and therefore a separate unit is not necessary.

Figure 6 shows the state diagram of the main control unit. Table 3 shows as well all states implemented and their outputs. A clear understanding of how the states are sequenced can be obtained from table 4 and table 5 which show the contents of the ROM. As stated before, it is up to the control logic to sequence the states correctly, according to the instruction to be performed.

Table 4 shows the actual contents of the control ROM and table 5 shows this value in hexadecimal format, in the same way the memory is initialized with table 4 data via a .mif file. The ROM is a 22 bit x 40 memory. Depending on the operation to be performed, the control logic selects the appropriate address for the ROM to start. The sequencing is then handled by a counter which increments the address by one until the operation is finished. The Instruction Fetch phase is performed at the end of each instruction, allowing the control logic to know which operation is to be performed next and where to address its beginning. The only situations that can disturb the counter linear sequencing are: Illegal Operation exception, Overflow exception and waiting for the Acknowledge response from memory (Wishbone). These possibilities should also be addressed by the control logic. Figure 5 shows the Main Control Unit implemented.

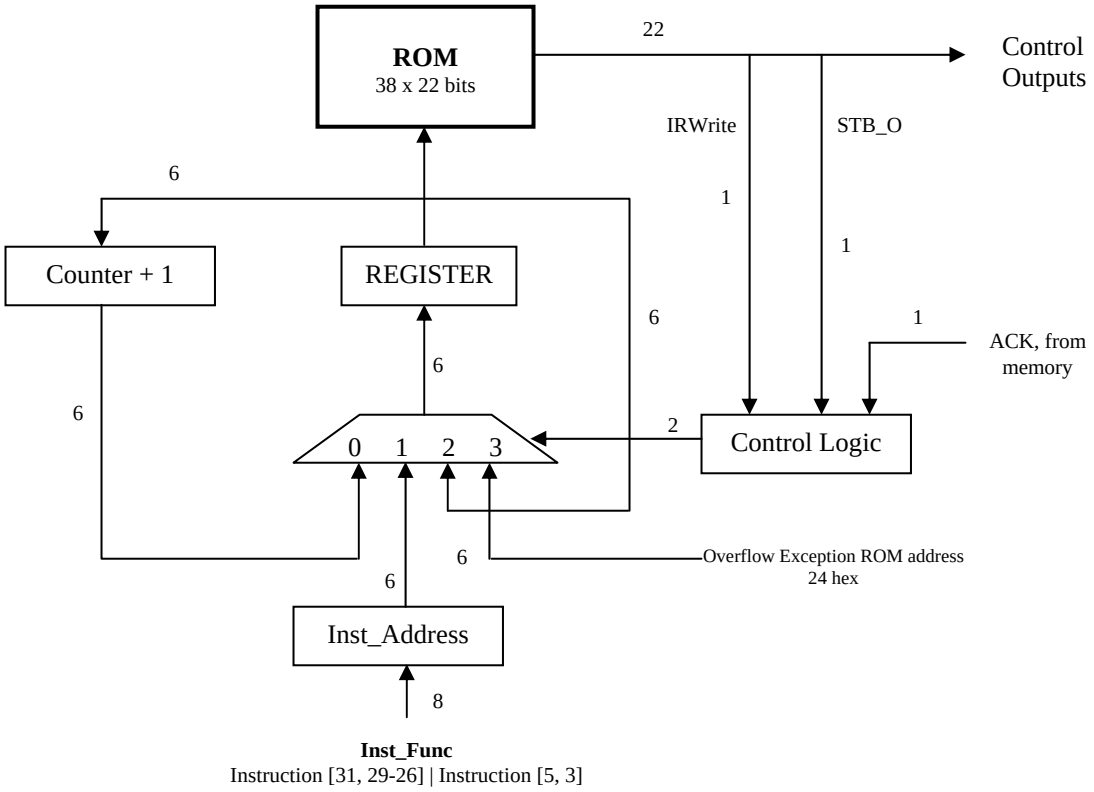


Figure 5. Main Control Unit.

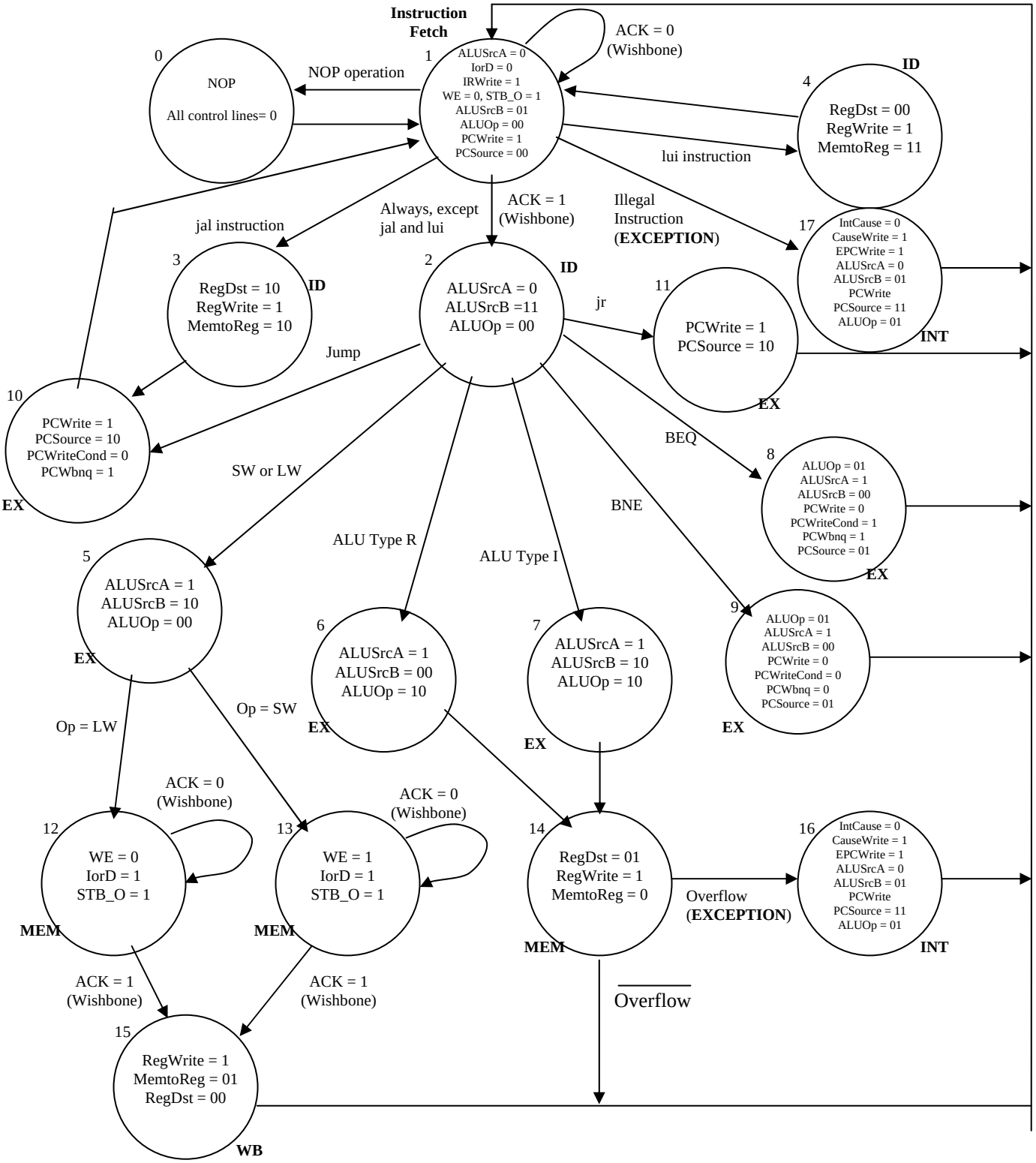


Figure 6. Main Control State diagram. **IF, ID, EX, MEM and WB labels are included only to show which step # the main control is executing.** For example the MEM label does not necessarily mean that there is a memory access, only that it is the fourth step in the execution.

STATE #			STB_O	CauseWrite	IntCause	EPCWrite	RegDst	RegWrite	ALUSrcA	ALUSrcB	WE	MemtoReg	IorD	IRWrite	PCWrite	PCWriteCond	PCWbnq	ALUOp	PCSource
0	No Operation	NOP	0	0	0	0	00	0	0	00	0	00	0	0	0	0	1	00	00
1	Always (IR = Memory[PC], PC = PC+4)	IF	1	0	0	0	00	0	0	01	0	00	0	1	1	0	1	00	00
2	Always, except (jal & lui) (A = Reg[IR[25-21]], B	ID	0	0	0	0	00	0	0	10	0	00	0	0	0	0	1	00	00
3	jal instruction ($ra = PC + 1)	ID	0	0	0	0	10	1	0	00	0	10	0	0	0	0	1	00	00
4	lui instruction ($rt = constant) ENDS	ID	0	0	0	0	00	1	0	00	0	11	0	0	0	0	1	00	00
5	Mem reference(ALUOut = PC + se(IR[15-0]))	EX	0	0	0	0	00	0	1	10	0	00	0	0	0	0	1	00	00
6	ALU-R instruction (ALUOut = A op B)	EX	0	0	0	0	00	0	1	00	0	00	0	0	0	0	1	10	00
7	ALU-Imm instruction (ALUOut = A op se(IR[15-0]	EX	0	0	0	0	00	0	1	10	0	00	0	0	0	0	1	10	00
8	BEQ (if (A == B) PC = ALUOut) ENDS	EX	0	0	0	0	00	0	1	00	0	00	0	0	0	1	1	01	01
9	BNE (if (A != B) PC = ALUOut) ENDS	EX	0	0	0	0	00	0	1	00	0	00	0	0	0	0	0	01	01
10	Jump and jal (PC = PC[31-28] || IR[25-0]) ENDS	EX	0	0	0	0	00	0	0	00	0	00	0	0	1	0	1	00	10
11	jr (PC = $rs) ENDS	EX	0	0	0	0	00	0	1	11	0	00	0	0	1	0	1	00	00
12	Mem reference/Load	MEM	1	0	0	0	00	0	1	10	0	00	1	0	0	0	1	00	00
13	Mem reference/Store ENDS	MEM	1	0	0	0	00	0	1	10	1	00	1	0	0	0	1	00	00
14	ALU Operation Type R ENDS	MEM	0	0	0	0	01	1	1	00	0	00	0	0	0	0	1	00	00
15	ALU Operation Type Imm ENDS	MEM	0	0	0	0	00	1	1	10	0	00	0	0	0	0	1	00	00
16	Load (Reg[IR[20-16]] = MDR) ENDS	WB	0	0	0	0	00	1	1	10	0	01	1	0	0	0	1	00	00
17	Overflow Interrupt	INT	0	1	1	1	00	0	0	01	0	00	0	0	1	0	0	01	11
18	OtherOp Interrupt	INT	0	1	0	1	00	0	0	01	0	00	0	0	1	0	0	01	11

Table 3. All possible states implemented that support the ISA required. Their sequencing depends on the way they are stored in the ROM and the Control logic. IF, ID, EX, MEM and WB labels are included to show which step the main control is executing.

			STB_O	CauseWrite	IntCause	EPCWrite	RegDst	RegWrite	ALUSrcA	ALUSrcB	WE	MemtoReg	IorD	IRWrite	PCWrite	PCWriteCond	PCWbnq	ALUOp	PCSource
NOP	NO OPERATION	NOP	0	0	0	0	00	0	0	00	0	00	0	0	0	0	1	00	00
	INSTRUCTION FETCH	IF	1	0	0	0	00	0	0	01	0	00	0	1	1	0	1	00	00
LOAD	Always, except (jal & lui)	ID	0	0	0	0	00	0	0	10	0	00	0	0	0	0	1	00	00
	Mem reference(ALUOut = A + se(IR[15-0]))	EX	0	0	0	0	00	0	1	10	0	00	0	0	0	0	1	00	00
	Mem reference/Load	MEM	1	0	0	0	00	0	1	10	0	00	1	0	0	0	1	00	00
	Load ENDS	WB	0	0	0	0	00	1	1	10	0	01	1	0	0	0	1	00	00
	INSTRUCTION FETCH	IF	1	0	0	0	00	0	0	01	0	00	0	1	1	0	1	00	00
ALU-R	Always, except (jal & lui)	ID	0	0	0	0	00	0	0	10	0	00	0	0	0	0	1	00	00
	ALU-R instruction (ALUOut = A op B)	EX	0	0	0	0	00	0	1	00	0	00	0	0	0	0	1	10	00
	ALU Operation Type R ENDS	MEM	0	0	0	0	01	1	1	00	0	00	0	0	0	0	1	00	00
	INSTRUCTION FETCH	IF	1	0	0	0	00	0	0	01	0	00	0	1	1	0	1	00	00
ALU-Imm	Always, except (jal & lui)	ID	0	0	0	0	00	0	0	10	0	00	0	0	0	0	1	00	00
	ALU-Imm instruction (ALUOut = A op se(IR[15-0]	EX	0	0	0	0	00	0	1	10	0	00	0	0	0	0	1	10	00
	ALU Operation Type Imm ENDS	MEM	0	0	0	0	00	1	1	10	0	00	0	0	0	0	1	00	00
	INSTRUCTION FETCH	IF	1	0	0	0	00	0	0	01	0	00	0	1	1	0	1	00	00
STORE	Always, except (jal & lui)	ID	0	0	0	0	00	0	0	10	0	00	0	0	0	0	1	00	00
	Mem reference(ALUOut = A + se(IR[15-0]))	EX	0	0	0	0	00	0	1	10	0	00	0	0	0	0	1	00	00
	Mem reference/Store ENDS	MEM	1	0	0	0	00	0	1	10	1	00	1	0	0	0	1	00	00
	INSTRUCTION FETCH	IF	1	0	0	0	00	0	0	01	0	00	0	1	1	0	1	00	00
LUI	lui instruction ($rt = constant) ENDS	ID	0	0	0	0	00	1	0	00	0	11	0	0	0	0	1	00	00
	INSTRUCTION FETCH	IF	1	0	0	0	00	0	0	01	0	00	0	1	1	0	1	00	00
BEQ	Always, except (jal & lui)	ID	0	0	0	0	00	0	0	10	0	00	0	0	0	0	1	00	00
	BEQ (if (A == B) PC = ALUOut) ENDS	EX	0	0	0	0	00	0	1	00	0	00	0	0	0	1	1	01	01
	INSTRUCTION FETCH	IF	1	0	0	0	00	0	0	01	0	00	0	1	1	0	1	00	00
BNE	Always, except (jal & lui)	ID	0	0	0	0	00	0	0	10	0	00	0	0	0	0	1	00	00
	BNE (if (A != B) PC = ALUOut) ENDS	EX	0	0	0	0	00	0	1	00	0	00	0	0	0	0	0	01	01
	INSTRUCTION FETCH	IF	1	0	0	0	00	0	0	01	0	00	0	1	1	0	1	00	00
JUMP	Always, except (jal & lui)	ID	0	0	0	0	00	0	0	10	0	00	0	0	0	0	1	00	00
	Jump and jal (PC = PC[31-28] || IR[25-0]) ENDS	EX	0	0	0	0	00	0	0	00	0	00	0	0	1	0	1	00	10
	INSTRUCTION FETCH	IF	1	0	0	0	00	0	0	01	0	00	0	1	1	0	1	00	00
JAL	jal instruction ($ra = PC + 1)	ID	0	0	0	0	10	1	0	00	0	10	0	0	0	0	1	00	00
	Jump and jal (PC = PC[31-28] || IR[25-0]) ENDS	EX	0	0	0	0	00	0	0	00	0	00	0	0	1	0	1	00	10
	INSTRUCTION FETCH	IF	1	0	0	0	00	0	0	01	0	00	0	1	1	0	1	00	00
JR	Always, except (jal & lui)	ID	0	0	0	0	00	0	0	10	0	00	0	0	0	0	1	00	00
	jr (PC = $rs) ENDS	EX	0	0	0	0	00	0	1	11	0	00	0	0	1	0	1	00	00
	INSTRUCTION FETCH	IF	1	0	0	0	00	0	0	01	0	00	0	1	1	0	1	00	00
Z-INT	Overflow Interrupt	INT	0	1	1	1	00	0	0	01	0	00	0	0	1	0	0	01	11
	INSTRUCTION FETCH	IF	1	0	0	0	00	0	0	01	0	00	0	1	1	0	1	00	00
Oth-INT	OtherOp Interrupt	INT	0	1	0	1	00	0	0	01	0	00	0	0	1	0	0	01	11
	INSTRUCTION FETCH	IF	1	0	0	0	00	0	0	01	0	00	0	1	1	0	1	00	00

Table 4. Actual content of the control ROM. Each instruction has a linear sequence. The control logic addresses the ROM to begin at the desired operation (position) and a counter increments by one the address until the operation finishes. The control logic selects the new address start point according to the next instruction.

For memory initialization

0	:	00 00 10	;
1	:	20 10 D0	;
2	:	00 20 10	;
3	:	00 60 10	;
4	:	20 61 10	;
5	:	00 E3 10	;
6	:	20 10 D0	;
7	:	00 20 10	;
8	:	00 40 18	;
9	:	01 C0 10	;
A	:	20 10 D0	;
B	:	00 20 10	;
C	:	00 60 18	;
D	:	00 E0 10	;
E	:	20 10 D0	;
F	:	00 20 10	;
10	:	00 60 10	;
11	:	20 69 10	;
12	:	20 10 D0	;
13	:	00 86 10	;
14	:	20 10 D0	;
15	:	00 20 10	;
16	:	00 40 35	;
17	:	20 10 D0	;
18	:	00 20 10	;
19	:	00 40 05	;
1A	:	20 10 D0	;
1B	:	00 20 10	;
1C	:	00 00 52	;
1D	:	20 10 D0	;
1E	:	02 84 10	;
1F	:	00 00 52	;
20	:	20 10 D0	;
21	:	00 20 10	;
22	:	00 70 50	;
23	:	20 10 D0	;
24	:	1C 10 47	;
25	:	20 10 D0	;
26	:	14 10 47	;
27	:	20 10 D0	;

Table 5. Actual ROM Contents.

There are two main components in the Main Control Unit. One of them is the Control Logic unit (figure 5). This unit drives the select input of the multiplexor and therefore selects between 4 different sources to feed the address to the ROM. The first input comes from the counter which, as explained earlier, just receives the actual address and adds 1. This counter is implemented using half adders as shown in figure 7.

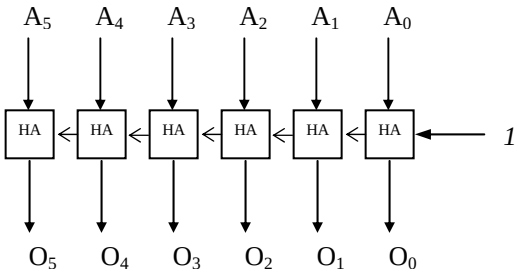


Figure 7. Counter module implementation.

The second input to the multiplexor, input 1, comes from the Inst_address module. This module receives the instruction to be performed and outputs the correct ROM address at which the operation begins. As in the ALU control module not all Opcode and function bits are necessary (12 bits) to determine the operation, only 8. These are: Instruction [31, 29-26] and the Instruction [5, 3]. The inst_address module is completely combinational. Espresso was used to determine the logic functions for each of the 6 address outputs depending on the inst|func code. The VHDL code implemented uses this equations and is also included in Appendix B.

Input 2 of the mux comes from the register that latches the actual address. The purpose of this input is to maintain the current address if required by the control logic, i.e. in the Wishbone interface when the memory does not reply with ACK assertion when the Strobe Output is set high. In this case the control logic forces the address to remain unchanged until the Acknowledge sign comes and the memory access operation can be performed.

The last input, 3, is hardwired to the ROM address of the overflow exception. In the event of an overflow the control logic selects the third input of the mux to feed the ROM and therefore the microprocessor enters the Exception state. The Control logic is a combinational element and behaves as shown in table 6.

STB_O	ACK	IRWrite	Overflow	OUTPUT (mux select)	
0	0	0	0	00	Counter
0	0	0	1	11	Overflow Exc.
0	0	1	0	xx	
0	0	1	1	xx	
0	1	0	0	00	Counter
0	1	0	1	00	Counter
0	1	1	0	xx	
0	1	1	1	xx	
1	0	0	0	10	HOLD/ No response
1	0	0	1	10	from memory (mem access)
1	0	1	0	10	HOLD/ No response
1	0	1	1	10	from memory (IF)
1	1	0	0	00	Counter
1	1	0	1	00	Counter
1	1	1	0	01	Instr. Fetch (IF)
1	1	1	1	01	Instr. Fetch (IF)

Table 6. Control logic Truth table.

The IRWrite bit cannot be high without the STB_O bit being high as well since the state that sets the IRWrite (Instruction Fetch) also sets the STB_O as seen in table 4; therefore the outputs in these cases can be assumed as don't cares. The control logic uses the IRWrite control bit from the ROM output to determine when the microprocessor is in the IF state since this is the only one that asserts it. When IRWrite is high the Control Logic should permit the Inst_Address module to feed the ROM address in order to begin the new instruction at the correct ROM address. Finally, when the overflow bit is high the Control Logic directs the ROM to the correct Overflow Exception state (address = 24 hex, see table 4 and 5) to serve the exception.

H. Design Considerations

Each one of the components of the microprocessor was tested independently and extensively to ensure that they behave as expected. Detecting errors early in the design flow will avoid the necessity of more time consuming and complicated modifications later when the system is assembled. The first major unit assembled together was the datapath. It is very important that the datapath works correctly, before designing the control unit since any changes in the datapath will require modifications of the control, incrementing the design effort and time required. The memory and Wishbone interface were also tested extensively before attempting the design of the Main Control Unit.

Different problems rose as the system size incremented when connecting new modules. One of the biggest challenges was the implementation of the memory module. Different configurations were attempted in order for the memory to satisfy the desired specifications. For example, a problem rises when the address and the Data_out are defined as registered since the address is sampled in the rising edge of the clock and the respective data will therefore appear only until the next rising edge. This behavior will require each memory access state to consume 2 clock cycles instead of one, incrementing the complexity of the main control and degrading the overall performance of the microprocessor.

In order for the memory to work as desired the address and Data_out inputs were selected as unregistered and the Data_in as registered. This means that a memory write event is synchronous while reading is asynchronous; the data is ready when the address changes, after a response delay. The same implementation was necessary for the Main Control ROM. This behavior is problematic since the data output of the ROM memory can vary until it stabilizes and therefore glitches can occur in the control signals which is highly undesirable. For example the PC register might be loaded by a glitch when it is not required to do so. The problem was solved by gating the offending control signals with the clock (or clock_bar). An example of this can be seen in figure 2, where the PCWrite control signal is gated.

III. RESULTS

The microprocessor was tested extensively. Since the microprocessor contains many different components, the results of the verifying simulations of each module are not included due to their extension. In the following section only the results that reflect how the complete system behaves are included. All the microinstructions implemented were tested extensively as well. Trivial cases as well as random input vectors were tested for the different logic and arithmetic operations. **However, giving the extension of the different microinstructions available, not all of these simulations are included in this report** since it would be impractical. Instead, a few input vectors for each instruction is shown and marked with detail in the output waveforms, indicating the instruction fetch phase, where the result is present, etc.

All microinstructions and exception conditions behaved as expected, indicating a correct implementation of the 32-bit microprocessor. All arithmetic and logic functions responded correctly to the different input vectors applied and the different instructions (jumps, branches, memory access and exception conditions) responded correctly as well.

In order to test the microprocessor the different microinstructions should be stored in the RAM as a normal program. All different operations, memory access, jumps, branches, arithmetic and logic operations are tested in this way. Exception conditions were also tested and included in the result waveforms.

Appendix A includes all waveforms preceded with the respective memory initialization for each test. Branches, jumps, exception conditions and memory access operations are not included in this section of the report. It is easier to visualize their behavior in the waveforms and the .mif files included in appendix A. The results of the different arithmetic and logic functions are shown in table 7 and 8:

Logic Operation	1st Operand	2nd Operand	Result Obtained	Comments
AND	00000000	FFFFFFF	00000000	correct
	F3245201	4ABC3A0C	42241200	correct
	003273CF	403BF20A	00000000	correct
ANDi	42241200	7FB0	00001200	correct
	00000000	D231	00000000	correct
	0032720A	382C	00003008	correct
OR	00000000	FFFFFFF	FFFFFFF	correct
	023428AB	00025426	02742EEB	correct
	00003012	26011240	26013252	correct
ORi	FFFFFFF	2360	FFFFFFF	correct
	02742EEB	642A	02746EEB	correct
	26013252	7048	2601725A	correct
XOR	FFFFFFF	02742EEB	FD8BD114	correct
	003273CF	403BF20A	400981C5	correct
	00003012	26011240	26012252	correct
XORi	FD8BD114	4012	FD8B9106	correct
	400981C5	002F	400981EA	correct
	26012252	2252	26010000	correct
NOR	023428AB	00025426	FD8BD114	correct
	00F03204	A0C67030	5F098DCB	correct
	00001123	00DC4500	FF23AADC	correct

Table 7. Logic functions included in Appendix A. **NOTE:** not all input test vectors used are included in the waveforms due to the length and complexity increase in the waveforms. **Table 7 and 8 only show the values extracted and verified from the waveforms**.

Arithmetic Operation	1st Operand		2nd Operand		Result obtained		Comments
	HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL	
ADD	7FFFFFFF	2147483647	00000000	0	7FFFFFFF	2147483647	correct
	804421C	-21430185	042AB021	69906465	846ED1DD	-22073112099	correct
	7261A000	1919000576	0004380A	276490	7265D80A	1919277066	correct
ADDi	846ED1DD	-22073112099	A201	-24063	846E73DE	-2073136162	correct
	7265D80A	1919277066	2645	9797	7265FE4F	1919286863	correct
	7FFFFFFF	2147483647	0001	1	80000000	OVERFLOW	correct
SUB	0042F507	4388103	0038B041	3715137	A44C6	672966	correct
	00000000	0	7B042001	2063867905	84FBDFFF	-2063867905	correct
	A0021E31	-1610473939	7FF26410	2146591760	200FBA21	OVERFLOW	correct
SUBi	0042880B	4360203	9304	-27900	0042F507	4388103	correct
	A0026041	-1610457023	4210	16912	A0021E31	-1610473939	correct
	00003004	12292	3004	12292	00000000	0	correct
SLT	00000001	1	00002402	9218	00000001	1	correct
	00003CD8	15576	00002402	9218	00000000	0	correct
	00003CD8	15576	00003CD8	15576	00000000	0	correct
SLTi	00000001	1	0002	2	00000001	1	correct
	00002402	9218	1AB3	6835	00000000	0	correct
	00003CD8	15576	3CFF	15615	00000001	1	correct

Table 8. Arithmetic functions included in Appendix A. **NOTE:** as in table 7, not all input test vectors used are included.

TIMING CONSIDERATIONS

In order to obtain the clock frequency at which the processor can operate the critical path of the datapath must be analyzed. There are two possible paths that might determine the clock frequency, these are shown in figure 8. Although path #2 seems shorter it must be considered that the memory access might not be so fast and therefore a timing figure for this path is required. In both cases however the delay of the control signals must be added since the Main Control Unit also has its own delay and will therefore affect the different multiplexors and the ALU in the datapath. The worst case delay for the ALU must be considered in the calculation. Since the adder and subtracting operations are performed by a ripple carry adder then the worst case delay will occur when the carry propagates throughout the entire 32 bit carry chain. The test vector utilized to generate the complete carry propagation is the addition of 7FFF_FFFFhex + 0000_0001hex. This is the delay that must be considered for the ALU.

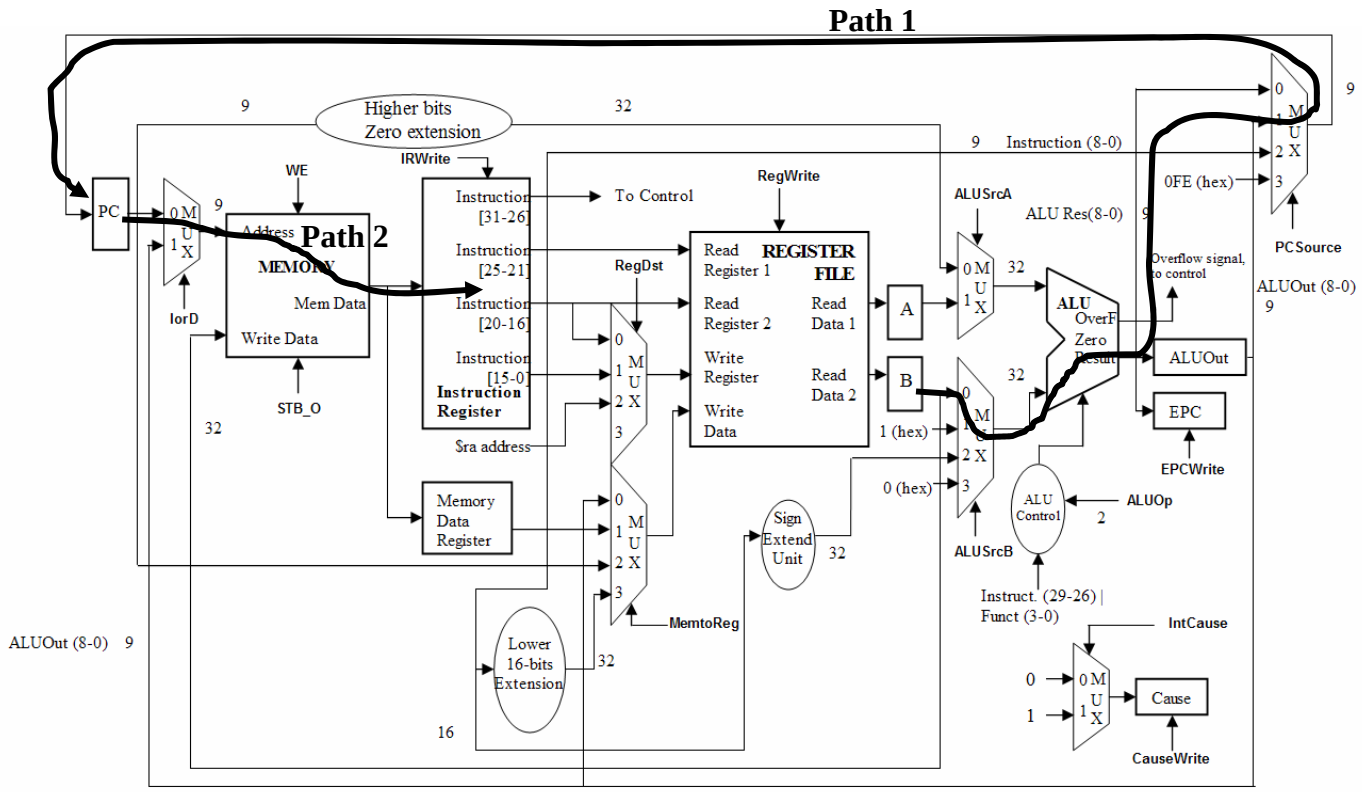


Figure 8. Microprocessor Datapath used to obtain maximum operating frequency. Two possible critical paths are shown. (From figure 1)

IV. CONCLUSION

The design and implementation of this microprocessor required substantial design effort in order to comply with the project specifications. All the test vectors and microinstructions were performed correctly as shown in the Waveforms. Also, a brief timing analysis was done in order to determine a rough estimation of the maximum operating frequency of the device.

Different aspects of the design were particularly challenging, like the memory implementation with the Wishbone Interface and the Main Control unit with a ROM based approach. The different issues and conditions encountered were addressed successfully and the microprocessor implemented behaves as expected. An important aspect that must be noticed is the effect of implementing the Main Control Unit with a ROM and sequencer logic instead of the common registered based FSM. Debugging was easier in some cases when the problem was caused by an inappropriate control signal. In this case all that was needed was to modify the initialization file of the ROM, sometimes by just changing a bit, instead of redesigning the complete FSM. This approach proved to be advantageous for this reason; analogous to what is practiced in real life when the Flash ROM BIOS of a computer is updated for example.

As a final remark the project provided a very good perspective of the different conditions and issues that might arise when designing a simple architecture that would otherwise be difficult to get without being involved directly with its design.

REFERENCES

[1] J. Hennessy and D. Patterson, Computer organization and Design. Morgan Kaufmann Publishers, San Francisco, 1998.
[2] J. Stine, “ECE485 Final Project definition , Multi-cycle MIPS Architecture with Wishbone Interface”, Illinois Institute of Technology, 2003.
[3] Silicore Corporation, “WISHBONE Public domain library for VHDL” technical reference manual, Minnesota, 2001.
[4] Silicore Corporation Website, “Wishbone Frequently Asked Questions”. http://www.silicore.net