An - Najah Univ. J. Res. (N. Sc.) Vol. 20, 2006

An Educational Processor: A Design Approach

معالج تعليمي، طريقة تصميم

Raed Alqadi, & Luai Malhis

Department of Computer Engineering, Faculty of Engineering, An-Najah National University, Nablus, Palestine.

E-mail: alqadi@najah.edu

Received: (5/1/2005), Accepted: (7/3/2006)

Abstract

In this paper, we present an educational processor developed at An-Najah National University. This processor has been designed to be a powerful supplement for computer architecture and organization courses offered at the university. The project is intended to develop an easily applicable methodology by which students get a valuable experience in designing and implementing complete processor with simple readily available off-the-shelf components. The proposed methodology is beneficial to computer engineering students enrolled at universities, specially in developing countries, where advanced laboratory equipments are rarely available. The design philosophy is to build a general-purpose processor using simple and wide spread integrated circuits. This methodology includes: defining an instruction set, datapath layout, ALU implementation, memory interface, controller design and implementation. For testing and evaluation purposes, a debugging tool is also built and implemented to serially connect the processor to a personal computer. In this paper, we present the methods and components used in the design and implementation phases and the tools developed to complete the design process. This methodology has shown that students enrolled in the computer architecture course get the full experience in processor design without the need for advanced laboratory equipments. The components used are cost efficient and methods proposed allow students to work at home, hence, this methodology has proven to be cost effective and yet very educational.

Key Words: Educational Processor, RISC Architecture, Computer Architecture, Instruction Set, Controller Design, Monitor Program.

ملخص

تحتوي هذه الورقة على طريقة لتصميم المعالجات لأغراض تعليمية، تعتمد على استخدام شرائح الكترونية متوفرة في الأسواق المحلية، ولا تحتاج إلى استخدام أجهزة متقدمة، غير متوفرة في البلدان النامية في معظم الأحيان. وقد طورت هذه الطريقة المقترحة، في جامعة النجاح الوطنية كوسيلة تعليمية إضافية لمساقات تصميم الحاسوب، وعمارة الحاسوب، إذ تتيح للطالب كسب المهارات الأساسية المتعلقة بتصميم المعالجات ابتداء من وضع مجموعة التعليمات، ثم تطوير وحدة الحساب والمنطق وتصميم وحدة التحكم بالمعالج. كما تم تطوير عدة برمجيات تساعد المصمم على تنفيذ المراحل المختلفة في التصميم وبنائها، ووضح البحث كيفية ربط المعالج بجهاز حاسوب شخصي لفحصه، وتحميل برامج لتنفيذها باستخدام المعالج، وتعتمد الطريقة المقترحة في الأساس على نوعية الشرائح، والمعدات المتوفرة بسعر زهيد وتقدم في الوقت نفسه وسيلة تعليمية قيمة تتيح لطلبة جامعات الدول النامية تصميم معالجات وفحصها دون الحاجة إلى توفير أجهزة متقدمة أو شرائح باهظة الثمن.

1. Introduction

Computer architecture, computer organization, and microprocessors, are typical courses taught in computer/electrical engineering and computer science departments. In these courses, a detailed study of processor instruction set and design is investigated thoroughly. In a computer architecture course, students are mainly theoretically introduced to the design and implementation of an educational processor like the DLX processor(9). Students can get a great experience in the processor design methodologies if they are asked to complete a practical processor design project. In this paper we propose a new processor that we call Educational Processor Unit (EPU), and a new design methodology that could be used for educational purposes.

The processor we propose has been designed and developed at An-Najah National University as a supplement to a computer architecture course. The architecture of this processor has been chosen to assimilate the current trends in processor design. The EPU's instruction set is designed to follow the reduced instruction set architecture (RISC)(1,9) philosophy. Next, we give an overview of the EPU and the design methodology adapted.

The instruction set for this processor is complete and general. It provides a good example on the design of general-purpose processors. The design methodology is driven by the fact that advanced IC components such as ASCIs, CPLDs, FPGAs, and sometimes GALs, in addition to advanced laboratory equipments may not be available at the universities of developing countries. Hence, we based our design methodology on using basic integrated circuit (IC) components that are readily available at affordable prices. In fact, we will rely on common components such as ROMs, PALs, RAMs, and common MSI logic circuits. The authors also believe that an efficient method for improving teaching computer-hardware related courses is to develop your own tools for designing, debugging and testing purposes. Therefore, to facilitate the EPU design, we have developed software tools such as: ALU generator, controller generator, assembler, and tools for an interface circuit that connects the EPU to a personal computer. The interface circuit facilitates data transfer between the EPU and the personal computer as well as simplifying the testing and debugging process. A monitor program has also been developed that can be loaded into the memory of the EPU. The monitor has been designed for the purposes of loading user programs into EPU's memory, getting a dump of the EPU's memory and executing a loaded program. Consequently, this methodology allows students to complete their projects at home with minimal cost and with no need for advanced laboratory equipment.

With these objectives in mind, we propose the design of a new processor that will certainly enhance the students' knowledge in this subject and yet is cost-effective. This methodology is very practical such that students are able to build complete processors at minimal cost. The experience they will go through will make them very knowledgeable in all phases of processor design. The tools they will build will provide a good example in interfacing hardware components with a personal computer. Proposing and developing new teaching methodologies to aid students in their courses is popular in worldwide-known universities and many papers with similar interests have been proposed. See for example(1-7, 10).

A detailed discussion of the processor proposed, methodology adapted, components used, and the communication software developed are presented in the following sections. Section 2 describes the instruction set chosen for this processor. Section 3 discusses the datapath including register file and other datapath components. The ALU design and its generator are also presented in this section. Section 4 contains a detailed discussion of the control unit and the timings of the control signals needed to control the datapath execution. A software tool for generating the control signals' values is also presented. Section 5 includes a discussion of interfacing the EPU serially to a PC accompanied with a monitor program used to debug the EPU and to download and execute user programs. Final thoughts and concluding remarks are presented in section 6.

2. The Processor and Its Instruction Set

The processor we propose is a 16-bit general purpose machine with features of recent processor design methodologies. Since recent trends in computer architecture tend to use RISC technology, we decided to use this trend because of its popularity in computer architecture courses as well as its simplicity in hardware design. Since the main objective of this paper is to show a practical and applicable processor deign methodology, one could argue that 8-bit machine would be sufficient. However, we decided to use 16-bit architecture to show that the method is easily applicable to design and implement a 16-bit processor. The first step in processor design is to select the instruction format(s) and the instruction set. Modern machines use uniform, orthogonal instruction set and format; hence our instruction set follows this scheme. We have decided to use three formats which are 16-bits in size as shown in Figure 1. Any instruction in the instruction set can be placed into one of three categories: *Immediate, Register or Jump.*

Immediate (register) type instruction requires performing an ALU operation on two operands one is a register (source and destination) and the other is an immediate value (a register). Jump (conditional / unconditional) is relative to current PC and an 11-bit signed value (-1024 to + 1023) is added to the PC.

Immediate	Opcode(5 bits)	RD	Literal (8-bit)
Register	Opcode(5 bits)	RD	RS
Jump	Opcode(5 bits)	**Address (11 bits)**	

Figure (1): Instruction Formats

The **16** bits of an instruction is divided into the following parts:

1. **Opcode**: Operation code, 5 bits, which means that up to 32 instructions can be defined in the instruction set.
2. **RD:** Destination register and source of first operand, 3 bits.
3. **RS:** Source register of second operand, 3 bits.
4. **Literal:** Signed immediate data, 8 bits.
5. **Address:** Jump and conditional jump, 11 bits. The address is relative to the program counter.

In our machine, eight 16-bit registers will be used. Register R7 is used as program counter and will also be visible to programmers so that it can be used to implement some instructions such as call, return, and jump to an address specified by a register. Also, one register will be used as a stack register but that is left to the programmer to choose.

Throughout this paper, we will use R6 for stack operations to implement pseudo instructions such as push, pop, call and return. The complete instruction set is shown in Table 1. Even though there are only 30 instructions, almost any program can be written using this instruction set.

Table (1): Instruction Set

No	Opcode	Instruction	Operation
0	00000	LU RD, L	Load upper byte of RD with L, old upper byte is moved to lower byte.
1	00001	LL RD, L	Load lower byte of RD with L
2	10000	LW RD, [RS]	Store RD in memory at address specified by RS
3	11000	SW [RS], RD	Load RD from memory at address specified by RS
4	11001	MOV RD, RS	Move RS to RD
5	00010	ADDL RD, L	ADD L to RD
6	00011	SUBL RD,L	SUB L from RD
7	00100	ANDL RD,L	AND RD with L
8	00101	ORL RD,L	OR RD with L
9	00110	XORL RD,L	XOR RD with L
10	00111	CMP RD, L	Compare RD with L
11	10010	ADD RD, RS	ADD RS to RD
12	10011	SUB RD,RS	SUB RS from RD
13	10100	AND RD,RS	AND RD with RS
14	10101	OR RD,RS	OR RD with RS
15	10110	XOR RD,RS	XOR RD with RS

… Continue table (1)

No	Opcode	Instruction	Operation
16	10111	CMP RD, RS	Compare RD with RS
17	11010	ROL RD	Rotate left one bit, MSB goes to Carry
18	11011	ROR RD	Rotate right one bit
19	11100	SHL RD	Shift left one bit, MSB goes to Carry
20	11101	SHR RD	Shift right one bit
21	11111	SAR RD	Shift Arithmetic right one bit
22	01000	JMP address	Jump to Address
23	01001	JZ address	Jump if Zero to Address
24	01010	JNZ address	Jump if not Zero to Address
25	01011	JC address	Jump if Carry to Address
26	01100	JNC address	Jump if not Carry to Address
27	01101	JS address	Jump if sign is set to Address.
28	01110	JNS address	Jump if not sign to Address
29	10001	SWAP RD,RS	RD gets the bytes of RS but the low byte of RS is swapped with upper byte of RS before loading into RD

To keep the design simple so that it can be easily implemented by students, some instructions will be defined as pseudo instructions. Pseudo instructions are implemented by using a combination of few instructions from the original instruction set. Table 2 shows some of these instructions. Note that original instruction set is general and sufficient to implement any program.

Table (2): Pseudo Instructions

No	Pseudo Instruction	Sequence of Instructions
1	LI RD, 16-bit	LU RD, L; ; Load Upper 8-bits of RD LL RD, L ; Load Lower 8 bits of RD
2	CALL address	SW [R6], R7 ; Push PC on stack ADD R6, 1 ; Increment stack pointer JMP address ; Go to subroutine
3	RET	SUB R6,1 LW R7,[R6]
4	PUSH RD	SW [R6], RD ADD R6,1
5	POP RD	SUB R6,1 LW RD, [R6]

The next section discusses the datapath that implements these instruction and components used in the implementation.

3. Component Organization and Layout

Figure 2 shows the components that make up the datapath for our EPU. As mentioned above, the register file consists of 8 registers of which R7 is used to implement the program counter (PC). As shown in the figure the datapath consists of the following components: special-purpose registers (IR, MAR, FLAGS and PC (R7)), a set of general purpose registers (total of 7 R0-R6 in the register file), multiplexers, a sign-extension, SWAP, PC select (OR) and memory units. All registers are 16 bits in size. These components are discussed in details in the following subsections.

3.1 Registers, Program Counter and Multiplexers

In this subsection, we describe the main components of the datapath shown in Figure 2 which includes: IR, Register File, Multiplexer, MAR, Memory, Flags, OR, Swap, and the Sign Extension components. The IR holds the instruction fetched from memory. The MAR holds the operand address to be loaded (stored) from (into) memory. Each of these two registers is implemented by using two 74LS374 ICs[*]. The register file is a two-port register file with RS and RD fields (3 bits each) connected to the address lines to select the corresponding data register at the RS and RD output data ports. The RD field also selects the destination register.

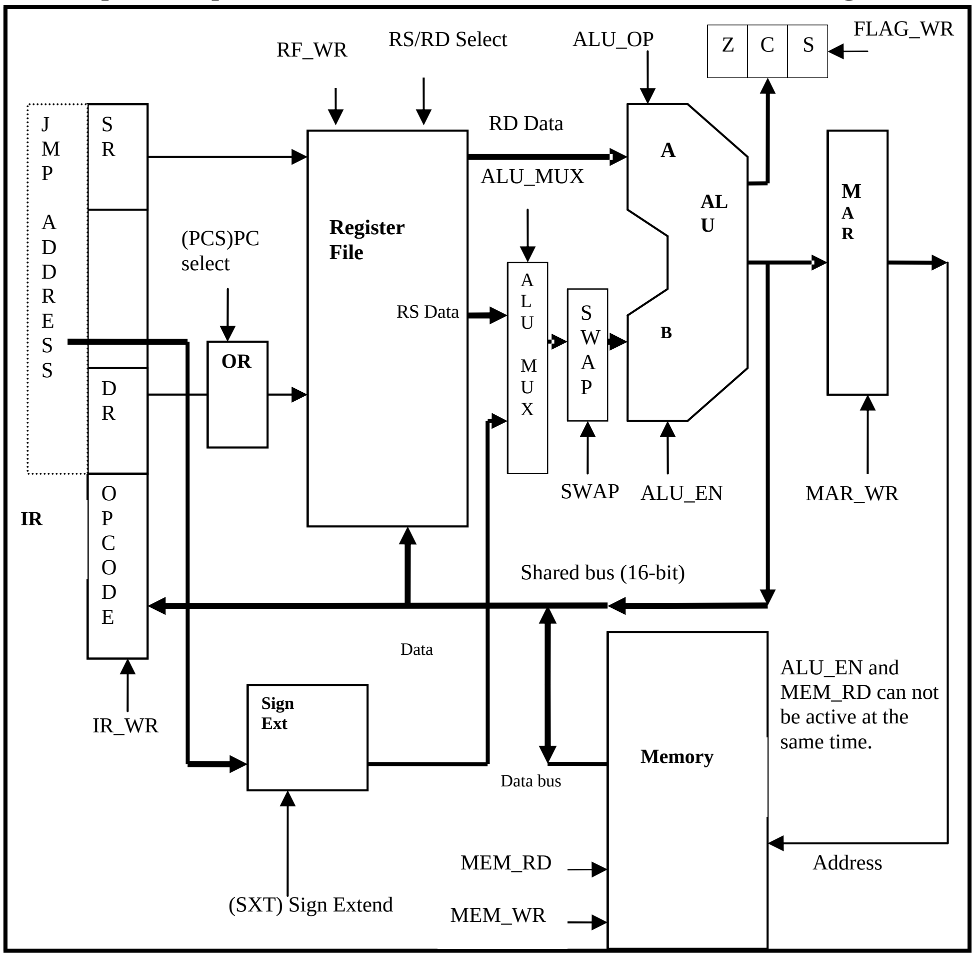

Figure (2): Datapath

[*] Even though we use LS in our description S, LS, ALS, F, and HCT technologies can be used.

The register file can be implemented by using specialized register file ICs or by using common registers and decoders. We used the second option because our objective is educational and we believe that it is important to let the students design and build every major component of the EPU. The hardware needed to implement these components is significant and is shown in Figure 3. This is because the design methodology dictates using readily available and easy to program ICs. In the design of the register file, it has been found that there is a problem in the number of Tri-State buffers needed which is 32 ICs. Therefore, to reduce the number of ICs (mainly the tri-state buffers) we have altered the register file implementation as shown in Figure 4. Such implementation uses much less hardware, but requires special attention to timing. The main difference between the register file shown in Figure 3 and the one shown in Figure 4 is that the second requires two cycles to output the RS and RD ports whereas the first one requires one cycle. Note that in this design we have used the internal Tri-state buffers of the 74LS374 registers.

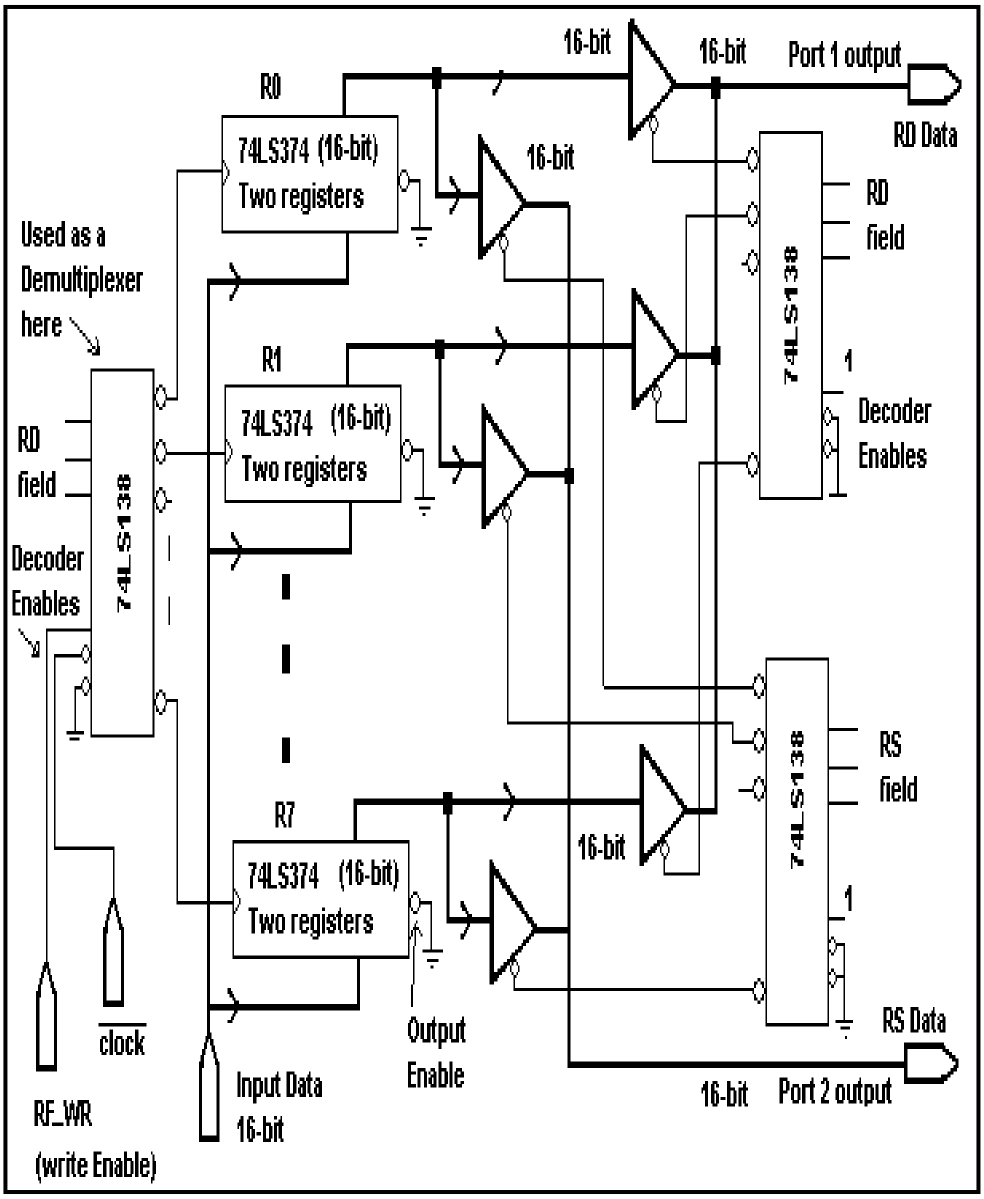

Figure (3): Possible Implementation of the Register File

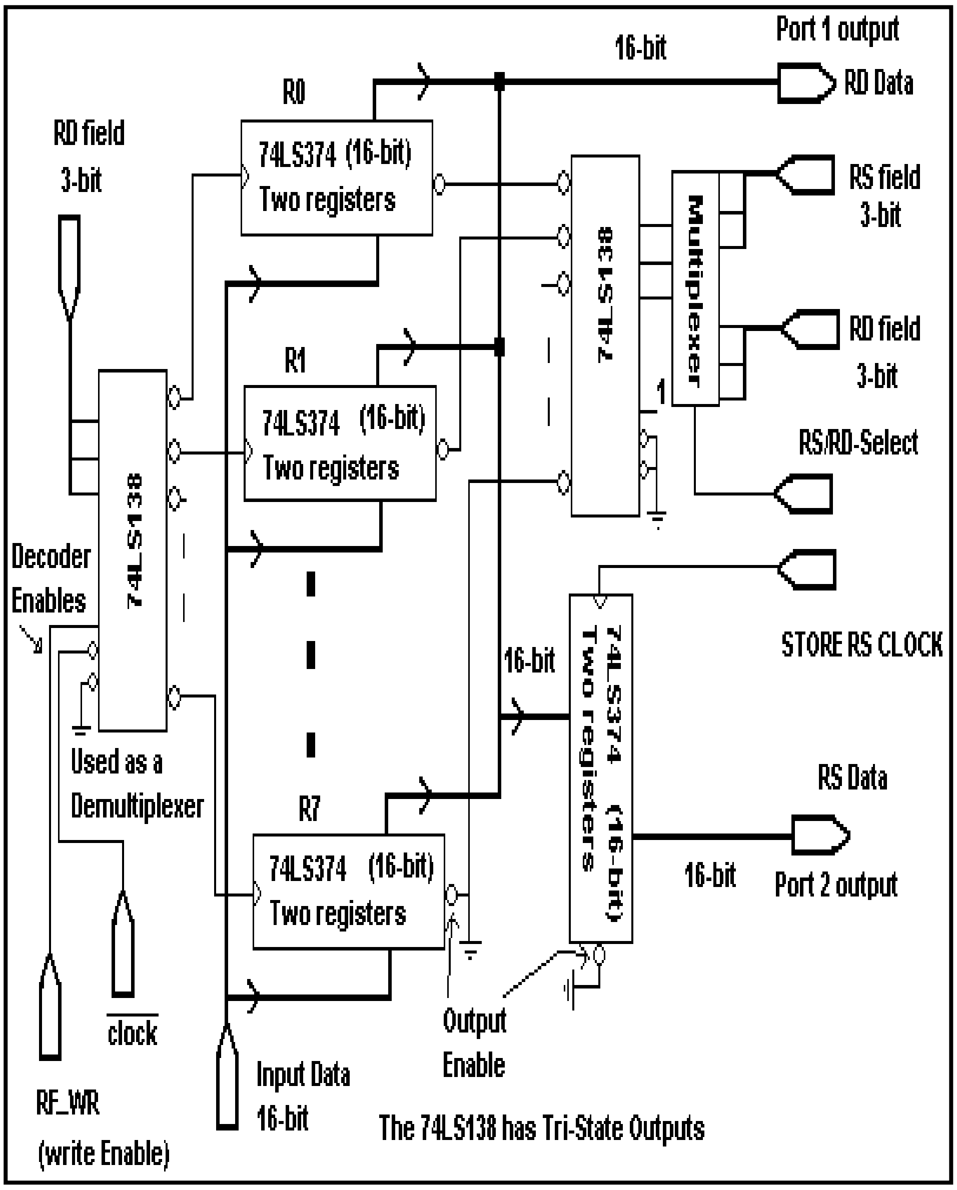

Figure (4): Register File Used in Our Implementation

The PC Select (OR) Unit: The OR component consists of three OR gates to make the output 7 to select the PC (R7) or pass the RD field, as it will be made clear later. This will be required to increment the PC, and also in a jump instruction. When the PC select signal is asserted(1) the output will be 7(PC), when the PCS signal is 0 the output is the RD field. Figure 5(a) illustrates the OR component.

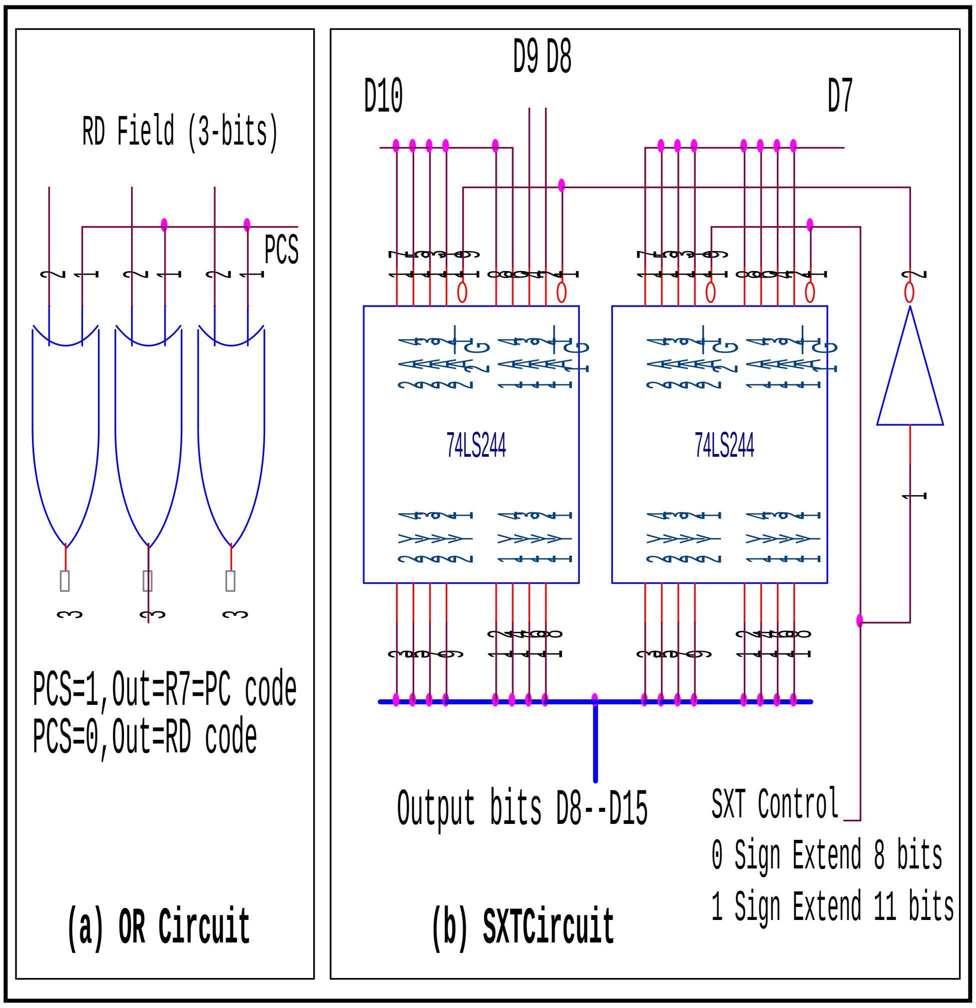

Figure (5): OR and SXT Circuit

Sign Extend Unit: The circuit of this unit is shown in Figure 5(b). It has a sign extend component that will be used to perform either sign extend 8-bit literal value to 16-bits or sign extend 11-bit literal value to 16 bits. We need to sign extending 8-bit literal value to 16-bit in instructions where one of the operands is literal. We need to sign extending 11-bit literal value to 16 bits for jump instructions.

The ALU MUX: This multiplexer is needed to select either the output of the SXT unit or the source register data (RS) for the second ALU operand. A single bit control signal (ALU_MUX) is needed for this purpose. This multiplexer is also implemented by using tri-state buffers (74LS244) ICs.

The Swap Circuit: This circuit either passes the output of the ALU MUX as it is or swaps the lower byte with the upper byte. The swapping is used to implement the LU and the SWAP instructions. Only in these two instructions the SWAP signal is asserted, while for all other instructions the swap circuit simply passes its inputs. The swap circuit is implemented by using two multiplexers. Each multiplexer has two sets of inputs where each set takes 8 bits (either the lower 8 bits or the upper 8 bits) see Figure 6 for details.

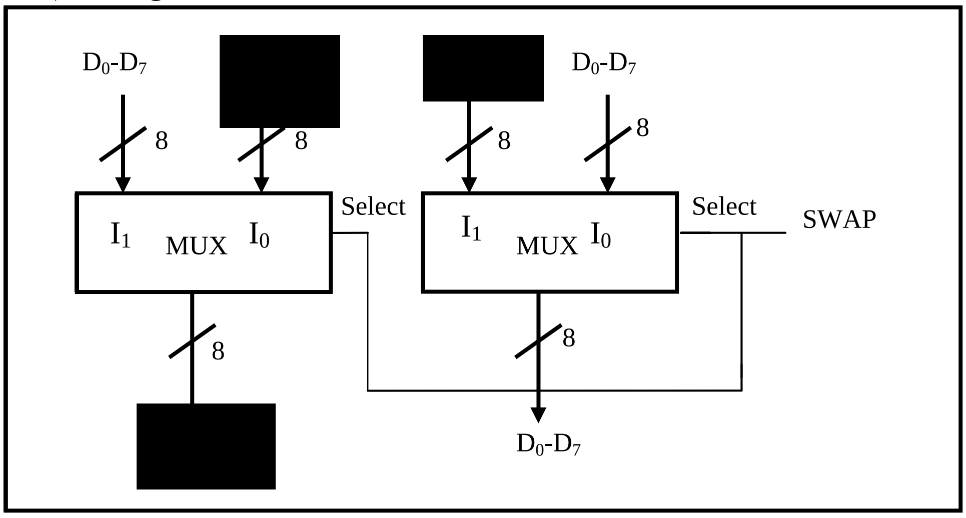

Figure (6): The Swap Circuit

3.2. The Arithmetic and Logic Unit

In this section, we present a technique for building custom ALUs by using EPROMs/EEPROMs, even though it is possible to build the ALU by using components such as 74181 or similar ICs. We believe it is better to let the students design ALUs from components such as GALs, PALs, or EPROMs. Here, we will describe building 16-bit ALU by using our software tool that generates EPROM programming files. By using this method, we can readily build custom ALUs from popular EPROMs such as 2764, 27128, 27256, and 27512.

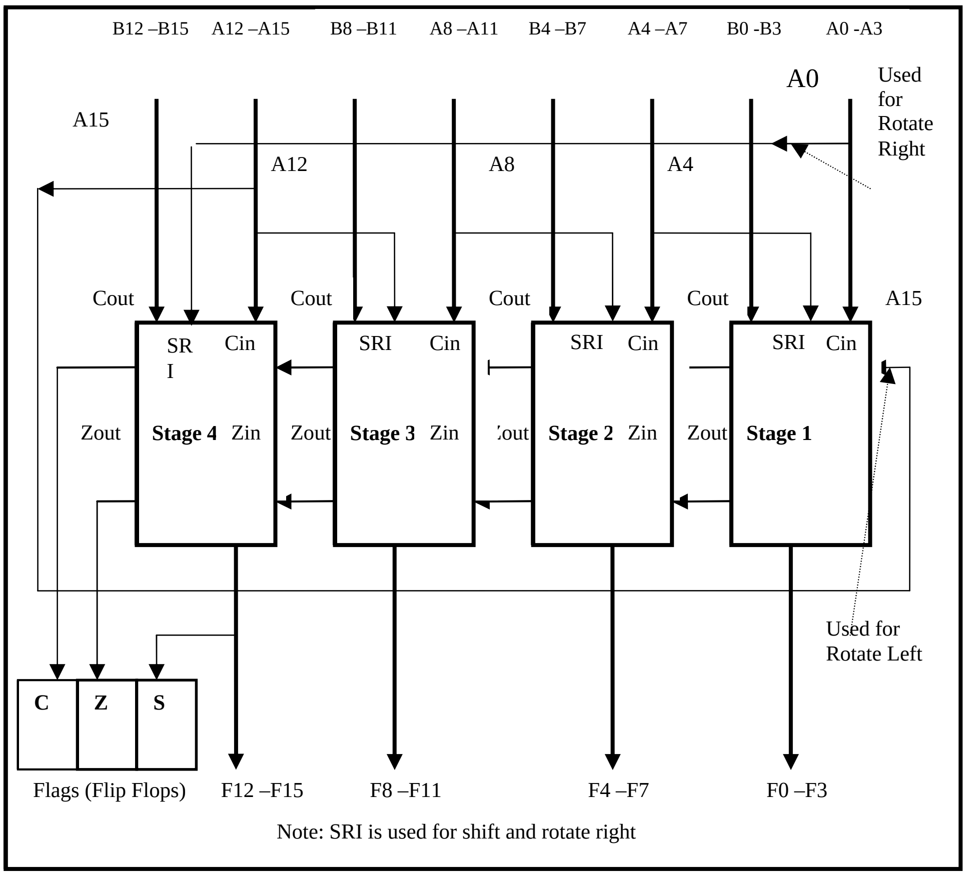

Figure (7): Block Diagram of Four Stages ALU

Figure 7, shows a 16-bit ALU built using four stages where each stage takes 4 bits of a16-bit operand. Each stage corresponds to one EPROM IC. Note that the address bus is used for inputs and data bus is used for the outputs. The set of ALU operations required must satisfy the requirement of the instruction set. The following set of operation is generally sufficient for a wide range of instruction sets, recall that A is connected to RD (destination register) and B is connected to the swap unit, which passes through or swaps either an RS or sign-extended literal.

1. F=A AND B.
2. F=A OR B.
3. F=A XOR B.
4. F=A+B ; ADD
5. F=A-B ; SUB
6. F=A+1. ; INC
7. F=A. ; PASSA
8. F=B ; PASSB
9. F=A ROL 1
10. F=A ROR 1
11. F = A SHL 1
12. F = A SAR 1
13. F = A SHR 1
14. F = [A15:A8]:[B7:B0] ; needed for LL instruction
15. F = [B15 :B7]:[A7:A0]; needed for LU instruction

The following algorithm is used to generate the program files for the four 27512 EPROMs. This algorithm is designed for a 4-stage ALU but can easily be modified for ALUs with more stages.

Algoritm ALUGenStage (int Stage)

```
{   Assign the bits for A, B, Cin, Zin and SRI to the Address input.
    Address = inputs = A,B, Zin, SRI
    Open Binaryfile
    FOR address =0 to Address = $2^{(No.\ of\ Inputs)}$ {
        // extract the operands from the address bits, and shift them appropriately
        A =  [AD3:AD0]          ; bits 0—3 of address
        B =  [AD7:AD4]>>4       ; bits 4—7 of address
        Cin=[AD8]>>8;
        SRI = [AD9] >>4;  // Put bit in fifth position
        OP = [AD13:AD10]>>10;
        Zin =AD[14] >>14;
        Let F be the output of 5 bits where the fifth bit is the Carry (Cout)
        switch(OP){
        case ADD:  if (stage == 1) F = A + B;  else F = A + B +Cin;
        case SUB:  if (stage == 1) F = A -B;   else F = A - B - Cin;
        case INC:  if (stage == 1)F = A +1 ;   else F = A + Cin;
        case AND:  F = A & B;
        case OR:   F = A | B;
        case XOR:  F = A ^ B ;
        case PASSA:    F =A
        case PASSB:    F = B;
        case ROL:  F = A << 1;
```

```
                F = F| Cin ; // First bit is carry from previous stage
case SHL:   F = A >> 1;
                if (stage != 1) F = F | Cin ;
case ROR: // Note SRI in the fifth position
                F =  (A|SRI) >> 1;
case SLR: // Note SRI in the fifth position
                F =  (A|SRI) >> 1;
                if(stage ==4) F = F& 0x07 ; // clear bit15
case SAR: // Note SRI in the fifth position
                if(stage ==4) SRI=(A & 0x8) << 1;
                //SRI =A4=bit15 of the 16-bit input
                F =  (A|SRI) >> 1;
Case LL: // In this case B is immediate data and A is destination
                if (stage ==1) || (stage ==2) F = B
                else F=A//result will be [F15:F10]=[A15:A8]:[B7:B0]
Case LU: if (stage ==1)|| (stage ==2) F= A
             else F = B  // [F15:F10] = [B15:B8]:[A7:A0]
             //This works in conjunction with the SWAP circuit
}
Zout = F & 0x0F; //   the 4 outputs excluding the carry zero
if (stage ==1) Zin =1
if (Zout ==0 && Zin=1) Zout = 0x20 ; //6th bit // set Zout
else Zout = 0;
```

```
        F = F|Zout;
        //Result in F=F4:F0 is the result of operation, F5 is Cout and F6is the Zout
        //Store the output F byte to file.
    }
```

4. The Control Unit

Because of the simplicity of the hardwired design and for educational purposes, we choose to implement the control by unit using the hardwired control methodology. This unit is abstracted as a finite state machine. In this abstraction, the control unit transits through a finite set of states and is responsible for asserting and un-asserting all control signals necessary for the datapath to function properly. The control unit simply remembers the current state of the system and then based on both, the current state and the set of input parameters (instruction opcode), the next state is determined. In addition, necessary control signals for the current state are also asserted. This process is repeated for all instructions in the instructions set.

4.1 EPU Control

The control signal needed to control the various activities of the datapath components are shown in Figure 2. These control signals are specific to the type of components chosen for our datapath design. Control signal names and values may change if the datapath components change. Nevertheless, we based our design on using the most primitive off-the-shelf components that are available at affordable prices. The control signals of our datapath components are described in more details in Table 3. Note that IR_WR and RD/RS-Select use the same control signal. This fact will be illustrated in the timing analysis described later. Also notice that the ALU_EN and the RAM_RD are complements of each other. The ALU_EN is connected to the Output Enable (OE) of the EPROMs that implement the ALU.

Table (3): Control Signal Names and Functions

Control signal	**Abbreviation**	**Description**
Register File Write	RF_WR	Write to Destination Register
Instruction Register Write	IR_WR Also RS/RD Select	Write Instruction to Instruction Register (Selects RD/RS in register file)
MAR Register Write	MAR_WR	Write to MAR Register
MEM Read	MEM_RD	Enables reading from Memory
MEM Write	MEM_WR	Enables writing to Memory
ALU enable	ALU_EN	Enables the ALU Tri-state buffer output
ALU MUX Selection	ALU_MUX	Operand select Control Line
ALU operation	ALU_OP	(4-bits)The Specified ALU Operation
PC Select	PCS	Selects PC(R7) or RD implemented by ORing PCS with RD (RD = 111=PC)
Sign Extend	STX	Either Sign Extends 8-bit literal or 11-bit address to 16-bit value
Flags Write	FLAG_WR	Write the carry, Sign and Zero D-Flip Flops

The first step in designing control is to categorize the instructions in the instruction set into groups such that all related instructions are placed in one group. All instructions in a given group transit through the same set of states in the state diagram when executed. Also, the same control signals are asserted and un-asserted for all instructions in the group. This makes the design much easier to manage. In addition, control signal values are determined by state bases rather than by instruction bases. Each instruction in the instruction set is placed into one of seven categories as shown in Table 4. The categorizing is based on the type of operation (ALU, Memory Load/Store, or Jump) and on whether it affects the FLAGS, or if branch taken or not.

Table (4): Categorizing the Instruction Set

Group A *OP RD, RS* *OP RD,L* *(flags affected)*	**Group B** *OP RD, RS* *OP RD,L* *(flags not affected)*	**Group C** *OP RD, RS* *OP RD,L* *(Only Flags affected)*	**Group D** *Taken jumps* *(Conditional And un-conditional)*	**Groups E And F** *Memory Load and Store*	**Group G** *Untaken jumps* *(Conditional)*
ADDL RD, L	**Group B1**	CMP RD,L	JMP address	**Group E**	JZ address
SUBL RD,L	LL RD, L	CMP RD,RS	JZ address	LW RD,[RS]	JNZ address
ANDL RD,L	MOV RD,RS		JNZ address		JC address
ORL RD,L			JC address		JNC address
XORL RD,L	**Group B2**		JNC address		JS address
ADD RD, RS	LU RD, L		JS address	**Group F**	JNS address
SUB RD,RS	SWAP RD,RS		JNS address	SW [RS], RD	
AND RD,RS					
OR RD,RS					
XOR RD,RS					
ROL RD					
ROR RD					
SHL RD					
SAR RD					
SLR RD					

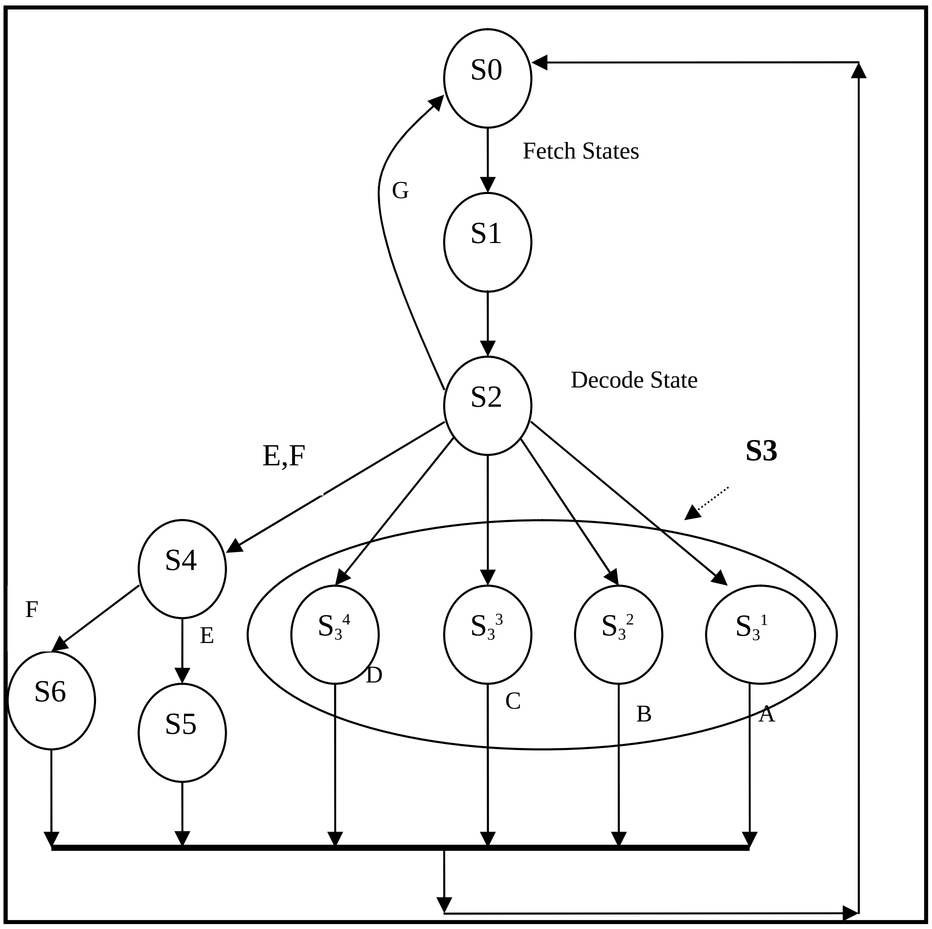

Figure (7): The State Diagram

The state diagram that comprises the states of the finite state machine that executes all instructions as categorized in Table 4 is shown in Figure 7. The operations performed in each state and which control signals are asserted is shown in Table 5. In order to clarify the operations in each state, S3 has been shown as 4 sub-states, but it is actually one state. Although there are many possible implementations, we have chosen the above states to facilitate the control unit implementation. In fact, the technique will still work for different state diagrams.

Table (5): The Operations Performed in Each State and Control Signals Asserted

State	**Group**	**Operation**	**Control Signals (other signals are in their initial value, some signals are active Low)**
S0	ALL	MAR←PC	ALU_OP = PASS RD, ALU_EN, MAR_WR, PCS(PC SELECT)
S1	ALL	IR←MEM[PC]	MEM_RD, IR_WR, ALU_EN = DISABLED
S2	ALL	PC←PC+1	ALU_OP = INC A, ALU_EN, RF_WR, PCS (PC SELECT)
S_3^1	A	RD← RD op (RS or L)	ALU_OP = F(opcode) ALU_EN, RF_WR, FLAG_WR
S_3^2	B	RD← RD op (RS or L)	ALU_OP = F(opcode) ALU_EN, RF_WR
S_3^3	C	RD op (RC or L)	ALU_OP = F(opcode)= subtract, ALU_EN, FLAG_WR
S_3^4	D	PC←PC+Address(STX)	ALU_OP =F(opcode)=ADD, ALU_EN, RF_WR, PCS (PC SELECT)
S4	E,F	MAR←SR	ALU_OP =PASS RS, ALU_EN, MAR_WR
S5	E	RD← MEM[RS]	ALU_EN=DISABLED, MEM_RD, RF_WR
S6	F	MEM[RS]← RD	ALU_OP= PASS RD,ALU_EN, MEM_WR

To simplify the control unit, we need to analyze the control signals carefully. There are three signals that are functions of the opcode only and not of the current state. These signals are: ALU_MUX, SWAP, and SXT. The ALU_MUX signal is selected as the value of the most significant bit in the opcode (see Table 1 for opcode assignment). The SWAP and SXT signals are generated along with the other control signals as state-dependent signals. Generating these two signals along with the state dependent signals add no cost to the required hardware. Since, (see figure 8), we have 14 bits (7 state-dependent control signals, 4-bit ALU operation and 3-bit next state value) we need two EPROM ICs. Thus, the two signals SWAP and SXT can be included at no cost. Tables 6 describes how SXT and SWAP are determined along with an *instruction group*. Based on current state and the instruction group other control signals are generated as shown in Table 7. Note that the ALU_MUX signals is taken directly form the opcode (see Figure 8).

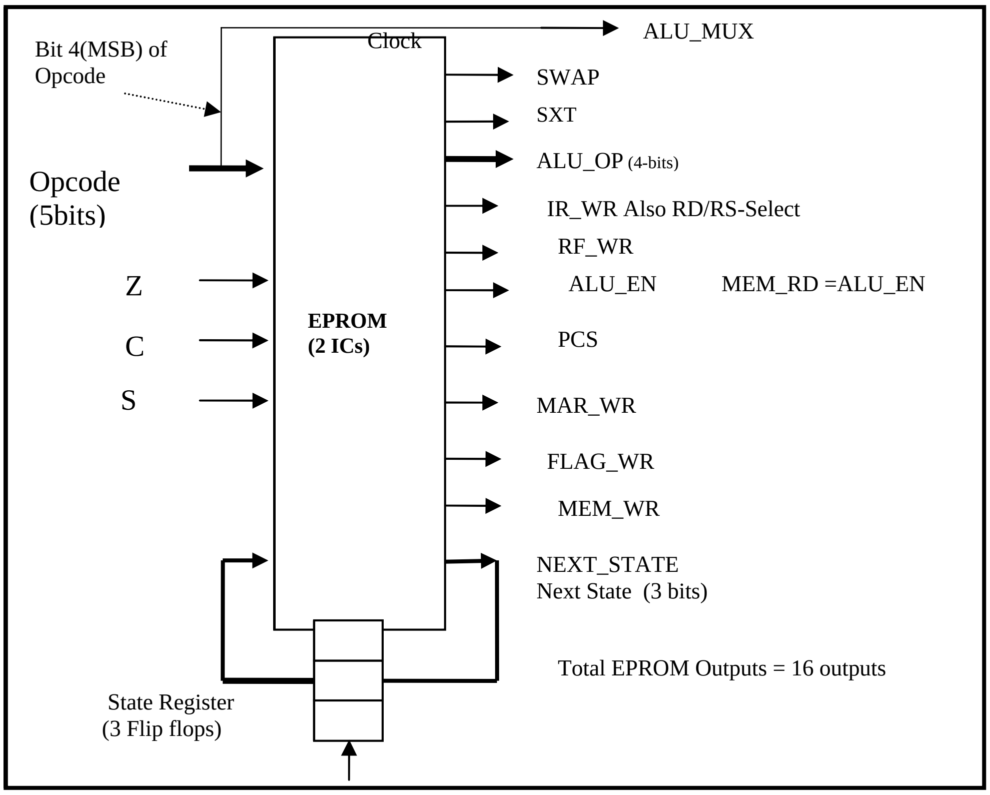

Figure (8): The Control Unit

Table (6): State Independent Control Signals and Group code.

No.	Inputs				Outputs		Group
	Opcode	**S**	**Z**	**C**	**SWAP**	**SXT**	
1	Group A	-	-	-	0	0	A
2	Group B1	-	-	-	0	0	B
3	Group B2				1	0	B
4	Group C	-	-	-	0	0	C
5	Group E	-	-	-	0	0	E
6	Group F	-	-	-	0	0	F
7	JMP	-	-	-	0	1	D
8	JS	0	-	-	0	1	G
9	JS	1	-	-	0	1	D
10	JNS	1	-	-	0	1	G
11	JNS	0	-	-	0	1	D
12	JZ	-	0	-	0	1	G
13	JZ	-	1	-	0	1	D
14	JNZ	-	1	-	0	1	G
15	JNZ	-	0	-	0	1	D
16	JC	-	-	0	0	1	G
17	JC	-	-	1	0	1	D
18	JNC		-	1	0	1	G
19	JNC		-	0	0	1	D

F (opcode) = 0 if operation requires select Literal, F(opcode) = 1 if operation requires RS

Table (7): Generating Next State and State-dependent Signals

Inputs		Outputs	
Present State	**OPCODE (Group)**	**Next State**	**Other Control Signals**
S0	-	S1	S0 Signals as in Table 5, SWAP & SXT as in Table 6
S1	-	S2	S1 Signals as in Table 5, SWAP & SXT as in Table 6
S2	A,B,C,D	S3	S2 Signals as in Table 5, SWAP & SXT as in Table 6
S2	E,F	S4	S2 Signals as in Table 5, SWAP & SXT as in Table 6
S2	G	S0	S2 Signals as in Table 5, SWAP & SXT as in Table 6
S3	A	S0	S_3^1 Signals as in Table 5, SWAP & SXT as in Table 6
S3	B	S0	S_3^2 Signals as in Table 5, SWAP & SXT as in Table 6
S3	C	S0	S_3^3 Signals as in Table 5, SWAP & SXT as in Table 6
S3	D	S0	S_3^4 Signals as in Table 5, SWAP & SXT as in Table 6
S4	E	S5	S4 Signals as in Table 5, SWAP & SXT as in Table 6
S4	F	S6	S4 Signals as in Table 5, SWAP & SXT as in Table 6
S5	-	S0	S5 Signals as in Table 5, SWAP & SXT as in Table 6
S6	-	S0	S6 Signals as in Table 5, SWAP & SXT as in Table 6

4.2 Generating the Control Hardware of EPU

Based on the above explanation, we will present an algorithm for generating a controller implemented from EPROMs /EEPROMs. Although the algorithm presented is specific to the state diagram shown in Figure 7, it can be easily modified for any state diagram. In our implementation we need two ROM ICs (*ROM1* and *ROM2*).

Algorithm Generate Controller

```
Note: is concatenate operation
Open ROM1_file and ROM2_file
Address = inputs = [OPCODE, Z,C, S, and PRESENT _ STATE = STATE _ REGISTER]
Let O1 =[SWAP,SXT]; // 2 bits
Let O2 = [ALU_OP,IR_WR,RF_WR]; // 6 -bits
Let O3=[ALU_EN,PCS, MAR_WR, FLAG_WR, MEM_WR]; // 5bits and
Let GenerateTable5 (Si,GROUP) be a function that sets O2 and O3 as shown in the corresponding table row in Table 5,where Si is a state //
For address = 0 to 2^(No of Inputs)
    Extract OPCODE, PRESENT_STATE from address
    Set GROUP = Table6_Group (opcode,Z,C,S);// Group output as a function of opcode in Table6
    Set O1=[SWAP, SXT] = Table6_SWAP_SXT(opcode); // SWAP and SXT can as function of opcode,Z,C,S in Table 6
    [O2:O3] = GenerateTable5(PRESENT_STATE, GROUP)
    IF (PRESENT_STATE) NOT in (S0, S1, …S6) // invalid state
        ROM1 = 0, ROM2 =0 // That is, send to initial state 0
    ELSE {
        IF (PRESENT_STATE == S0     )          NEXT_STATE = S1
        ELSE IF (PRESENT_STATE == S1)        NEXT_STATE= S2
        ELSE IF( PRESENT_STATE in (S3,S5,S6)      NEXT_STATE = S0
        ELSE IF (PRESENT _ STATE= S4 && GROUP = E) NEXT _
         STATE = S5
         ELSE IF (PRESENT_STATE= S4 && GROUP= F)   NEXT _
        STATE = S6
        ELSE IF (PRESENT_STATE== S2){
                    SWITCH (GROUP){
                    CASES A, B, C, D: NEXT_STATE = S3
                    CASES E, F: NEXT_STATE = S4
                    CASE G: NEXT_STATE = S0
             }// End Switch
    }// End ELSE_IF
    ROM1 = O1:O2 ;// byte to be written to ROM1 file
    ROM2 = O3:NEXT_STATE ,// byte to be written to ROM2 file
}// end ELSE
Write ROM1 to ROM1_File and ROM2 to ROM2_File

End for
```

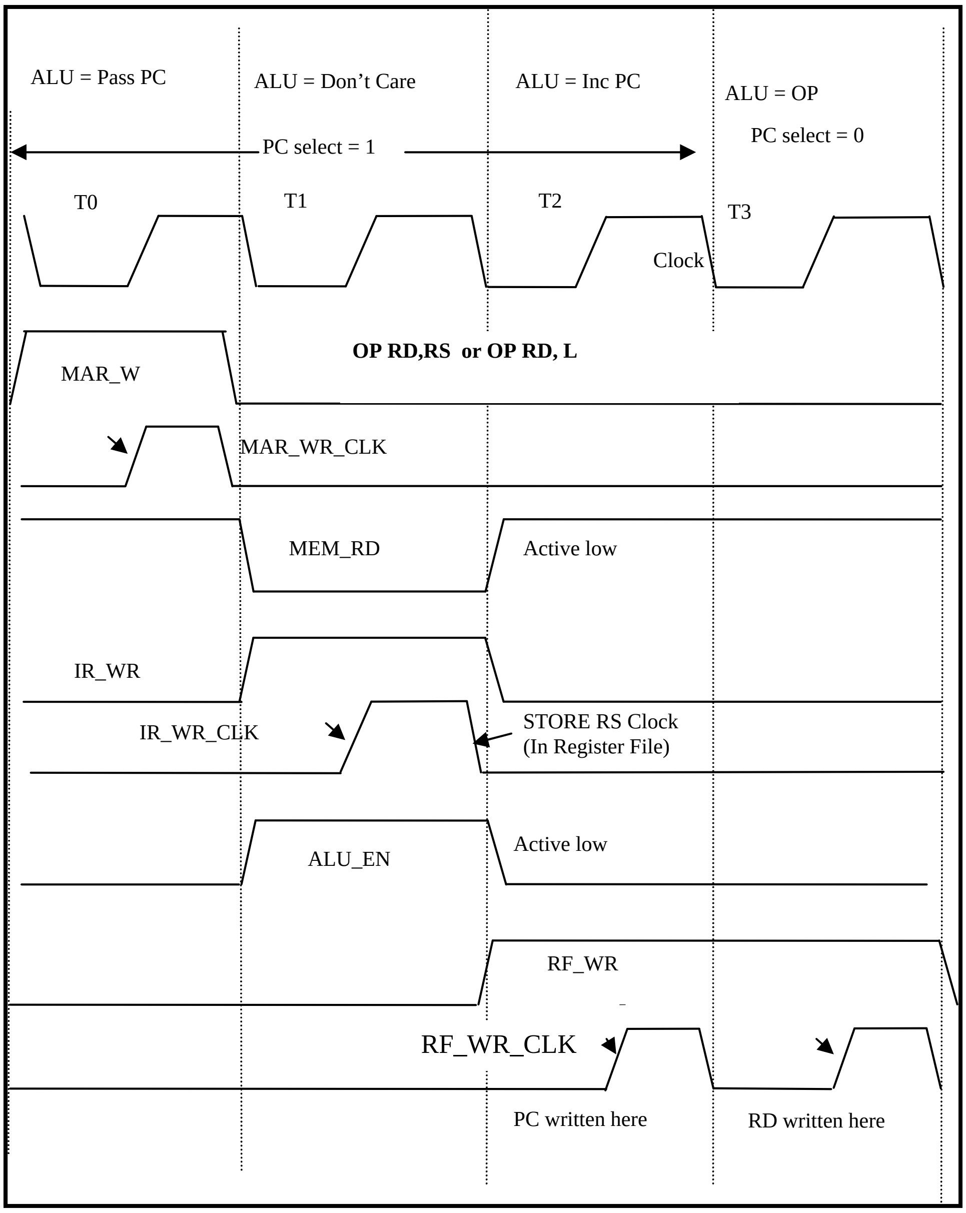

Figure (9): Timing Diagram for Group A

4.3 EPU Timing Requirements

Before implementing the control, the timing diagram for each group of instructions shown in Table 4 must be drawn. This process is necessary to ensure that there are no glitches and the data is written at the correct triggering edge to guarantee the setup and hold times for the registers and memory. Also, attention should be given to edge- triggered and level-triggered components, for example memory read/write signals are level-triggered active low signals. Also, since the ALU bus and the RAM bus are shared the ALU output and the RAM output are shared, therefore they cannot be enabled together.

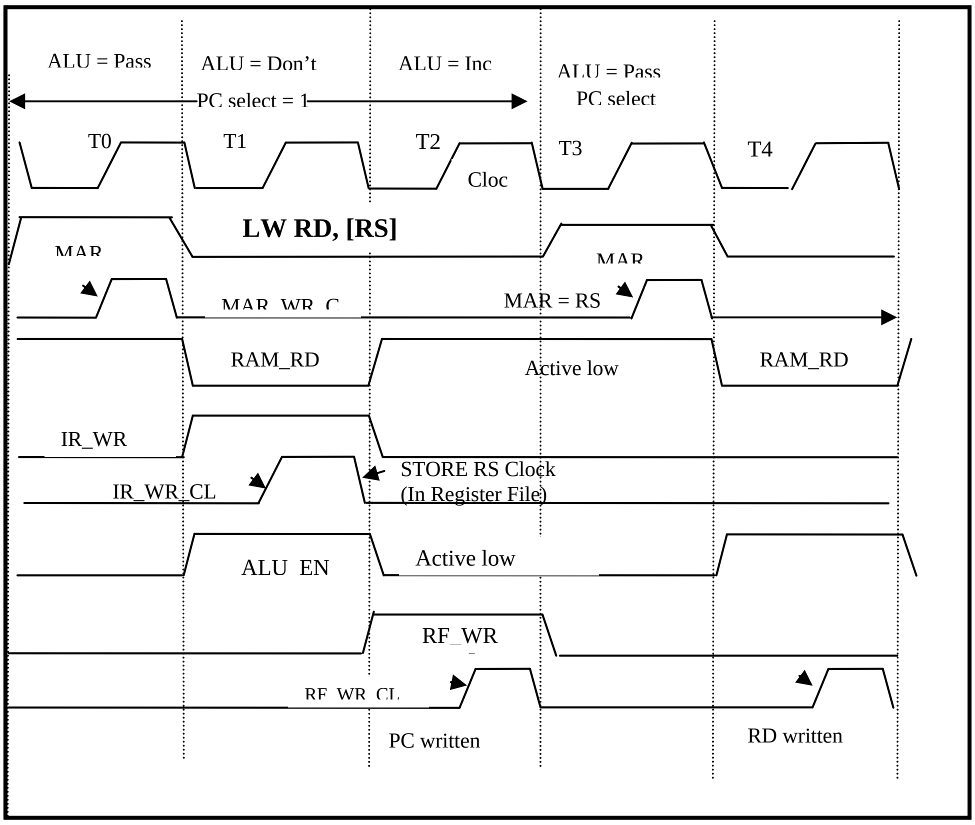

Figure (10): Timing Diagram for Load Operation

To guarantee timing requirements for correct EPU operations, two approaches are possible. In the first approach, all clock signals are generated by the controller, this is an easy approach, but it requires extra states. In the second approach, the state register is triggered by negative edge and the register clock signals (IR_WR, MAR_WR and FLAG_WR) are generated by ANDing the corresponding write signal with the clock. Note that this will generate a glitch free write signal, however the opposite will not. This timing step may result in modifying the state design step and an iterative process is necessary to refine the design. To illustrate the timing diagram, we will present the timing diagram for Group A, Figure 9, (ALU operations) and Group E, Figure 10, (Load Operation). Other group timings have been omitted for space considerations

5. Monitor Program and Interfacing to PC

In this section, we will describe a monitor program (simple operating system) used to download and execute user programs. The monitor program is stored in an EPROM while user programs are downloaded into RAM. The monitor program allows the EPU to be interfaced to a PC through serial port, and hence, can communicate with an interface program written by the authors. The user can download programs to the EPU RAM and execute the downloaded programs. The programs must first be written in assembly using the EPU instruction set and then assembled by the assembler developed by the authors. The assembler generates specially formatted binary files which the monitor downloads to RAM. The binary file is composed of frames (records) of the following format:

SOF FT AH-AL COUNT HB-LB HB-LB …… HB-LB CSH-CSL

Where:

SOF: Start of frame, one byte, we use ':'.

FT: Frame Type; 00:Regular Frame, FF: Last Frame, one byte

AH-AL:	High and Low bytes of address at which the instructions in the record start.
COUNT:	Number of Instruction words in the frame usually 16. This is a one byte.
HB-LB :	High byte and Low byte of an instruction.
CSH-CSL:	High and Low bytes of Checksum.

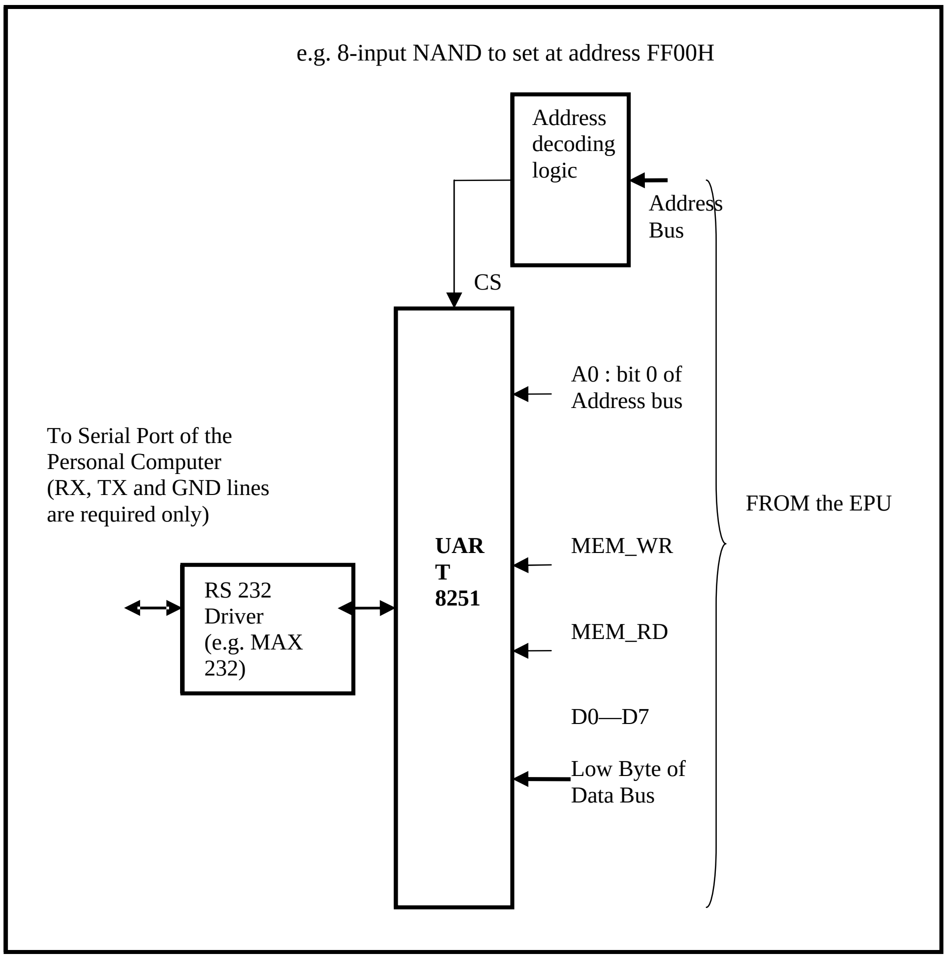
e.g. 8-input NAND to set at address FF00H
Address decoding logic
Address Bus
CS
A0 : bit 0 of Address bus
To Serial Port of the Personal Computer (RX, TX and GND lines are required only)
FROM the EPU
UAR T 8251
MEM_WR
RS 232 Driver (e.g. MAX 232)
MEM_RD
D0—D7
Low Byte of Data Bus

Figure (11): Interfacing the EPU to a Personal Computer

Before describing the monitor program, we need to describe the required hardware in order to interface the EPU with the personal computer. Since we are using the serial port, then a UART will be required which can be implemented either by hardware or by software. Here, we will describe the interface that uses the hardware UART 8251[8]. Implementing the UART using software is fairly simple and we recommend using it if a hardwired UART is not available. Figure 11 shows the interface circuit. The UART code will reside at addresses FF00H and FF01H of the EPU memory as shown in Figure 12.

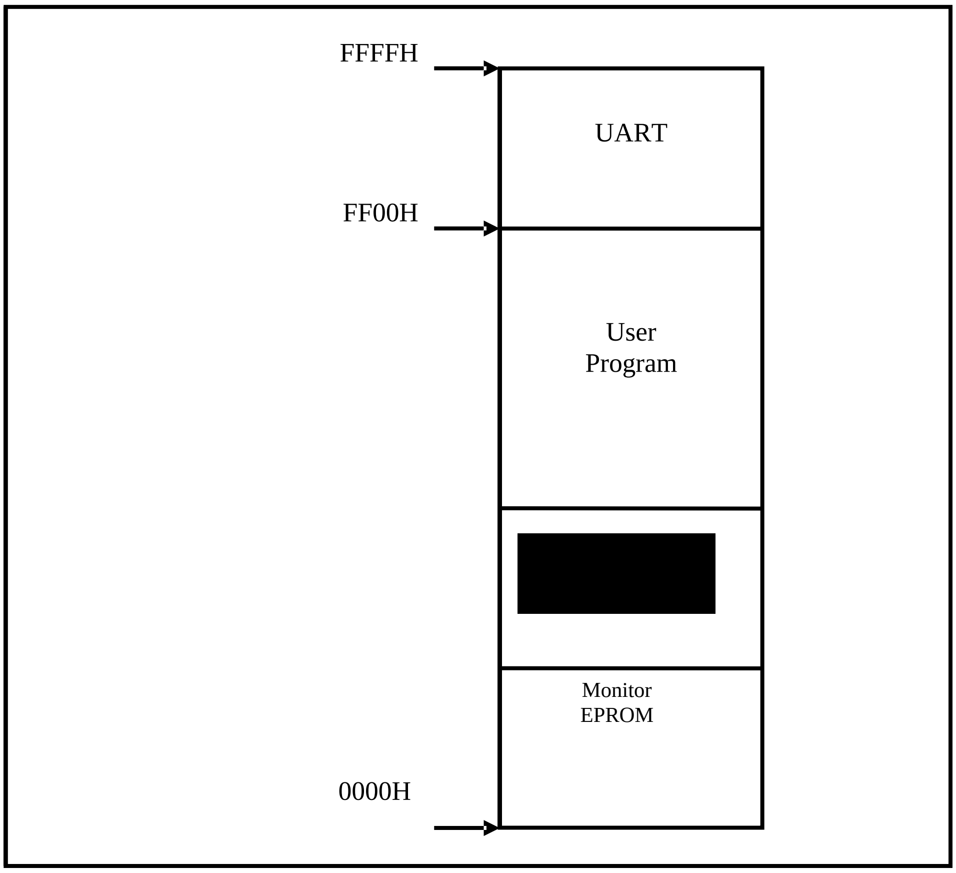

Figure (12): EPU Memory Divisions

Monitor Algorithm:

The monitor starts at address 0000H as shown in Figure 12 and will be executed when the system is reset. We will describe the monitor algorithm by using pseudo code; however some subroutines are presented in detailed assembly while some details are eliminated for simplicity of presentation. The monitor executes the following commands **'P'** to download program to the EPU, '**E**' to execute a loaded program and '**M**' to display a memory block. The pseudo code of the monitor algorithm is given below. First the main program is described followed by the subroutines.

Main Monitor Program

```
Initialize stack pointer: // Li R6, Stack-Start


Main-Loop       //Pseudo Code
    R5 = Call Read-Serial;        // Read command sent from PC
                                  ; // This is a blocking read;
    Switch (R5) {
        Case P:     Send Ack-Byte;            // Sends a 'R' for ready
                    Call Receive-Download;  // Receive user program.
        Case M:     Call Display-Memory;   / Display a block of memory
        Case E:     Jmp Execute-Program;  // Execute User Program
        Default: // Otherwise ignore
    End Switch
    Jmp  Main-Loop;    // loop forever
End Main
```

Receive-Download: //Pseudo Code

// Receives the user program file composed of frames with format described above.

Next-Frame:

```
    R5= Read-Serial          // Get Start of Frame
    If (R5 != SOF) Send-Error and return; // Not start of frame
    character so
    R5= Read-Serial          // Get Frame Type
    If (R5 == FF) return;    // if Last Frame (FF) then stop.
    R3 = Read-Word           // Get Start Address of instructions in R3
    R2 = Read-Serial;        // Get Number of Instructions in R2
    R1=0;                    // Initialize Computed Check Sum =0h
    While (R2 != 0){         // Read all Instruction words, R2 is Counter
         R5 = Read-Word      // Read Instruction
        MOV [R3], R5         // Store Instruction at address in R3
        R3 = R3+1;           // Increment Instruction address pointer
        R2= R2 -1            // Decrement Instruction Count by 1
        R1 = R1 + R5;        // Add read word to computed checksum,
    }// End While
    R5 =Read-Word;           // Get Checksum
    If (R5 != R1) Send Error;    // if received and computed checksums are
                                 // not equal  send error.
     Jmp Next-Frame;             // get Next frame
```

End Receive-Download

Display-Memory //Pseudo Code

This subroutine displays a block of memory specified by the starting and end address.

Expects Starting address and end address

```
    R2 = Read-Word          // Get block Start address and store in R2
    R3 = Read-Word          // Get End address in R3
    While (R2 != R3){       // While address not equal to End address
        LW R5,[R2]          // Read word from memory in R5
        Send-Word           // Send the word in R5 to serial port
        R2 = R2 +1          // Increment address
    }
End Display-Memory
```

Read-Serial: //Assembly code

```
// Receives a byte from UART
      LI R4, UART         // UART status register address in FF00
WAIT: MOV R5, [R4];       // Read Status Register
      ANDL R5, 01         // Is receiver empty?
      JZ WAIT             // If empty wait.
      ADDL R4, 1;         // Address of data register
      MOV R5, [R4];       // Read data register, received byte
      Li R4, 00FF         // Mask off upper byte.
      ANDL R5, R4         // Received byte in lower byte of R5
      RETURN;             // Result in lower byte of R5, high byte is zero
End Read-Serial
```

Read-Word: //Assembly code

Reads a word from the UART and stores result in R5

```
      CALL Read-Serial ;    //High byte is read and stored in Low byte of R5
      MOV R0, R5            //High byte is now in low byte R0
      SWAP R0, R0;          // Now in High byte of R0
      CALL Read-Serial;     //Low byte is read and is stored in Low
                            byte of R5
      OR R5,R0              // Word is in R5
      RETURN                // Result is in R5
```

End Read-Word

Send-Serial ///Assembly code

// Sends the lower byte in R5 serially by using the UART

```
      Li R4, UART           // Address of UART Status register
N_RDY: MOV R0,[R4];         // Read Status register
      ANDL R0, 02 ;         // Is transmitter empty
      JNZ N_RDY             //If not empty wait
      ADDL R4, 1;           // Address of data register
      MOV [R4], R5          // Send Byte
      RETURN
```

End Send-Serial

Send-Word: ///Assembly code

// Sends the word in R5 serially by using the UART

```
      SWAP R5,R5            // To send the high byte first
      CALL Send-Serial      //Send High Byte
      SWAP R5,R5            //To Send Low Byte
      CALL Send-Serial      // Send Low byte
      RETURN
```

End Send-Word

Execute-Program: //Pseudo Code

Executes the user program, it expects the starting address of the program

```
    R5 = Read-Word;    //Read the starting address of the program in R5
    MOV PC ,R5
    RETURN             // Jump to starting address of user program
End Execute-Program
```

6. Conclusion

In this paper we described an educational processor along with a methodology to develop it. The main objective behind this processor is to demonstrate that general- purpose processor can be implemented using simple off-the-shelf components. In this project, we discussed all aspects of processor design staring with instruction set definition which is based on RISC philosophy. Then, we discussed the design and implementation phase which is driven by the type of components available. Third, we showed how control signals are specified and the importance of their timings. Finally, we discussed a tool which serially interfaces the EPU with a PC. This tool includes a monitor program that resides in the EPU memory and is used to transfer data/programs between the EPU and the PC and to execute loaded programs.

The methodology presented can be easily adopted by computer engineering (science) departments at universities in developing countries to supplement a course in computer architecture or in processor design. This methodology is used by students enrolled in the computer architecture course at An-Najah N. University to build a complete and general-purpose processor. The students enjoyed and benefit greatly from the experience they acquired from this project at a minimal cost, and with no special hardware resources. They are able to define, design, implement, and debug a complete processor. Finally, we recommend adopting this methodology by universities in developing countries.

References

(1) M. Buehler, U. G. Baitinger ,"HDL-Based Development of a 32-Bit Pipelined RISC Processor for Educational Purposes," *9th Mediterranean Electrotechnical Conference* (Melecon'98), Tel-Aviv Israel, may (1998), 138-142.

(2) N. L. V. Calazans, F. G. Moraes, C. A. Marcon, "Teaching Computer Organization and Architecture With Hands-ob Experience," *32nd ASEE/IEEE Frontiers in Education Conference*, Boston MA, Nov. 6-9,(2002), T2F15-T2F20.

(3) H. Diab, I. Demaskieh, "A Computer Aided Teaching Package of Microprocessor Systems Education," *IEEE Transaction on Education*, May (1991), **34(2)**, 179-183,.

(4) J. Djordjevic, A. Milenkovic, N. Grbanovic, and M. Bojovic, "An Educational Environment for Teaching a Course in Computer Architecture and Organization", *IEEE TCCA Newsletter*, July (1999), 5-7.

(5) J. Gray,"Hands-on Computer Architecture–Teaching Processor and Integrated Systems Design with FPGAs", *Workshop on Computer Architecture Education*, June (2000), Vancouver BC.

(6) R. D. Hersch, "Integrated Theory and Practice in Microprocessor System Design Course", *Proc. Euromicro*, North Holland, Amsterdam (1984), 227-232.

(7) Intel,"MCS 51 Microcontroller Family User Manual," *www.intel.com/design/mcs51/manuals/272383.htm*

(8) V. Milutonovic ,"Surviving the Design of a 200 MHz RICS Microprocessor: Lessons Learned," IEEE Computer Society Press, (1997).

(9) Patterson & Hennessy, Computer Architecture: A Quantitative Approach, Morgan Kaufmann, (1990).

(10) B, J. Rodriguez, "A minimal TTL processor for architecture exploration", Proceedings of the 1994 *ACM Symposium on Applied Computing*, Phoenix Arizona, (1994), 338-340.