Journal Name
ISSN
Copyright

A Systematic Approach for Building Processors in a Computer Design Lab Course at Universities in Developing Countries

Raed A. Alqadi and Luai M. Malhis
Department of Computer Engineering, An-Najah National University, Nablus, Palestine

Abstract: This study presents a systematic technique for building processors by using off-the-shelf components. The main objective of this methodology is to introduce computer engineering, electrical engineering and computer science students, in developing countries, to all phases of CPU design using very primitive ICs and home made tool kits. Using this technique, students enrolled in processor design lab can design and build processors for a defined instruction set with readily available components at minimal cost. The proposed methodology has been implemented in the computer engineering department at An-Najah National University and has proven to be efficient in teaching computer architecture and processor design, as well as boosting computer-engineering students' self-confidence without requiring them to use very advanced laboratory equipment. Nevertheless, the sole purpose of this technique and the objective of building this microprocessor is pure educational and is not to introduce a new methodology for building microprocessors for commercial purposes. In this study, we will present our methodology by giving an instruction set example, then describing the design and implementation steps followed. In addition, we also discuss the primitive components used to build the datapath, the controller and the software tools used in both the implementation and testing phases. We will show that the proposed methodology is very effective in providing students with the experience of microprocessor design without the need for advanced and expensive kits and devices.

Key words: Processor design lab course, computer architecture, datapath, controller, computer engineering labs, effective education

INTRODUCTION

Most universities offering degrees in computer engineering, electrical engineering and computer science require students on the average to complete at least two courses in computer organization and architecture. Also, an additional course in microprocessors and/or microcontrollers is almost always required. Universities in American and European countries usually have design labs for building processors in order to enhance the computer architecture, organization and microprocessor courses. In such labs, students design and build complete educational CPUs. However, generally, in universities of developing countries, the attention, given to computer design labs is minimal. Students of these universities lack the hands on experience in designing and implementing a complete processor. The reason is the lack of necessary hardware and software resources to aid students in all phases of processor design. In this study, we will overcome these limitations by developing a methodology and related software tools by which students can develop their own design tools and can use off-the-shelf components to implement their processors. This methodology discusses all design and implementation phases of a processor and includes: defining an instruction set, laying out a data-path and the memory interface to execute this instruction set and implementing control with appropriate timings for control signals. We believe that developing in-house hardware and software tools is an effective method in teaching computer-hardware related courses. This philosophy has been used in many universities in developed countries, for example (Hersch, 1994), (Ozcan, 1996) and (Pastor *et al*., 2004). More papers discussing methods to improve teaching processor design courses are found in. (Gang, 2003), (Gottlib and Carter, 2003), (Nicoud, 1991) and (Nicoud and Sommer, 1975).

The design techniques section contains an overview of the design philosophy Section instruction set design describes the instruction set that will be used in the design example. Our instruction set example is based on microcontroller architecture such as the Intel and Microchip microcontrollers. The datapath section focuses on the datapath layout, with emphasis on custom ALU design and implantation. Off-the-shelf EPROMS are used to implement the ALU. The controller design and

Corresponding Author: Raed A. Alqadi, Department of Computer Engineering, An-Najah National University, Nablus, Palestine

implementation are discussed next along with all control signals and their timings. An assembler for our processor is described next. The last section introduces a hardware kit that we developed to interface the processor with a Personal Computer (PC). This Kit is accompanied by software that enables downloading programs to the processor as well as tools for debugging the hardware of the processor. Final thoughts and concluding remarks are then presented.

DESIGN TECHNIQUE

Due to the current situation in Palestine, where An-najah N. University is located, we face many challenges to obtain complex ICs such as FPGAs, CPLDs, GALs, therefore, we have to use easy-to-get components such as common RAMs, EPROMs/EEPROMs and MSI ICs. Hence, what we are presenting is a design of a CPU that mainly uses EPROMs for implementing Arithmetic and Login Units (ALU) and Control Units (CU). We use in-house software programs discussed later to produce files downloadable to EPROMs to generate the required functionality. These programs are developed by the authors. Moreover, the assembler for this processor is also developed by the authors and also described in a later section.

The technique presented in this study has been applied to teaching a processor design lab course at An-Najah National University. This course is part of the curriculum to obtain a B.S. degree in computer engineering. In this course, which usually taken as a fourth year level, students are asked to design and build a CPU with given specifications.

Figure 1 illustrates the main steps in the design procedure; some of the steps have to be done manually while others are partially or fully automated. Step 1 is mainly a manual step which involves selecting the instruction set and designing an appropriate instruction set format. Step 2 is the datapath design which involves designing the ALU, registers, internal buses and memory interface. This step has been decomposed into three sub steps; step 2.1 is fully automated while steps 2.2 and 2.3 are partially automated. Step 3 is the controller design step; for simplicity it has been decomposes into three sub-steps. An iteration process of steps 3.1 and 3.2 may be needed. The last step is the testing step in which an interface circuitry and a software tool have been built to facilitate this process.

INSTRUCTION SET DESIGN

Since our objective is mainly educational, we have chosen a design example for an 8-bit processor architecture suitable for micro-controllers. Nevertheless, this methodology can be extended to the design of general purpose processors. Recent trends in computer architecture uses reduced instruction set design philosophy (RISC), therefore, we decided to use RISC as the base of our microprocessor design. RISC design is dominating because of its simplicity and efficiency in hardware design. Despite the fact that the proposed architecture is RISC architecture, it has many common features with accumulator-based like machines. Such architecture is very popular in microcontroller products such as Microchip (www.microchip.com) and Intel 80×51 (www.intel.com). The instruction format for the design example used throughout this study is shown in Fig. 2. As shown in this Fig. 2, a single instruction format is used; therefore, all instructions will follow the same format as explained in Fig. 2.

The size of instruction format is 16 bits divided into the following parts:

Opcode: Operation code, size is 5 bits; hence up to 32 instructions can be supported.

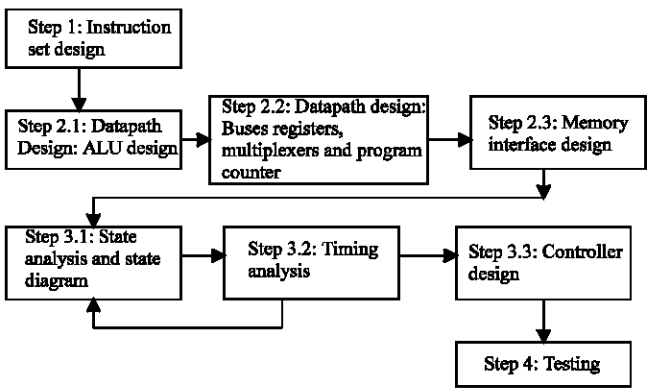

Fig. 1: Steps of processor design

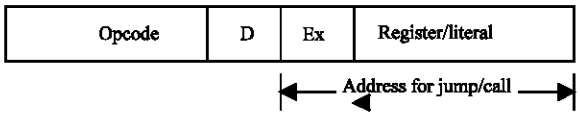

Fig. 2: Instruction format

Table 1: Instruction set

Opcode	Mnemonic	D	Operation
0	MOVLA L	0	Move Literal to A
1	MOVRA R	0	Move Register to A
2	MOVAR R	1	Move A to Register
3	MOVIRA R	0	Move indirect Register to A
4	MOVIAR R	1	Move Indirect A to Register
5	INCR R	1	Increment Register
6	DECR R	1	Decrement Register
7	ADDLA L	0	Add Literal to A
8	SUBLA L	0	Subtract A from Literal
9	ADDAR R	0,1	Add Register to A
10	SUBAR R	0,1	Subtract A from Register
11	ANDLA L	0	AND Literal with A
12	ANDAR R	0,1	AND Register with A
13	ORLA R	0	OR Literal with A
14	ORAR R	0,1	OR Register with A
15	XORLA L	0	XOR Literal with A
16	XORAR R	0,1	XOR Register with A
17	NOTR R	1	ONE'S complement of Register
18	ROLC R	1	Rotate left through carry
19	RORC R	1	Rotate right through carry
20	MVBC L	0	Move first bit in Literal to Carry
21	JMP Addr	0	Jump to Address (EX:Literal)
22	JZ Addr	0	Jump if Zero to Address
23	JNZ Addr	0	Jump if not Zero to Address
24	JC Addr	0	Jump if Carry to Address
25	JNC Addr	0	Jump if not Carry to Address
26	CALL Addr	0	Call Subroutine at Address
27	RET	0	Return
28	SETB R, B		Sets the Register bit B Bit specified by 3-bit value B=D:EX
29	CLRB R, B		Clears the Register bit specified by D:EX
30	JBS L, B		Jump if Bit Set to PC+1+Literal
31	JBC L, B		Jump if Bit cleared to PC+1+Literal

D: Destination, size is 1 bit. If D = 0 result will be stored in Accumulator A, else if D = 1 destination is the register specified in the register field.

Register/Literal: Size is 8 bits. Register part is interpreted as register address. This provides 256 data RAM locations (Registers) which is sufficient for most microcontroller applications. This field is also used to hold an 8-bit immediate value.

Ex: Extension size 2 bits. Used only in jump or call instructions to select the program counter value. This provides 10-bit jump address within the program memory. The instructions supported by our proposed microprocessor are summarized in Table 1.

DATAPATH

Figure 3 shows the datapath used in our design example; note that we have chosen the Harvard style architecture for memory interface because of its simplicity and efficiency. The main components of the data path are: The ALU, the Program Counter (PC), the Instruction Register (IR), the Accumulator (A), the Memory Data Register (MDR), the data RAM (Data registers) and the multiplexers. In the following subsections, we will describe these components. For simplicity, Fig. 3 does not show the components required to implement the CALL instruction and the bit manipulating instructions.

Registers, program counter and multiplexers: The A, MDR, IR registers are implemented with common 8-bit registers such as 74LS374 resisters, special attention has to be given to the triggering edge when the timing analysis is performed to write at the correct edge and to avoid glitches. Since IR is 16 bit registers, two 8-bit registers would be required.

In our design, we have restricted the program memory to 1K word (2 KB), such restriction is sufficient in microcontroller applications. Hence, a 10 bit program counter is required. A suitable counter must support parallel load in order to implement the conditional jump instructions. A popular counter that supports these features in addition to a clear feature (required for reset) is 74LS193. Also the counter has cascade feature, therefore, 3 ICs can be cascaded to implement the PC. The block diagram is shown in Fig. 4.

The multiplexers can be implemented using multiplexer ICs or by using Tri-state buffers such as 74LS244 ICs. The data registers (256-registers) can be implemented by using popular RAM ICs of size greater or equal to 256 such as 62xxx or 61xxx, for example a 6164 RAM IC can be used where only the first 256 bytes are used. In this case the first 8 bits of the address bus will be used and the rest of the address bus lines will be grounded. An improvement to this implementation will be to use part of these registers as Special Function Registers (SFRs) to implement the input and output ports. For example, the first 128 registers can be implemented using a RAM the upper 128 bytes can be dedicated to SFRs, in such case the SFRs will be implemented using individual registers or specialized ICs. For example an ADC can be used to add an analog input port.

From the datapath shown in Fig. 3, the control signals can be identified. These signals are illustrated in the Figure and divided into: ALU Control, ALU output

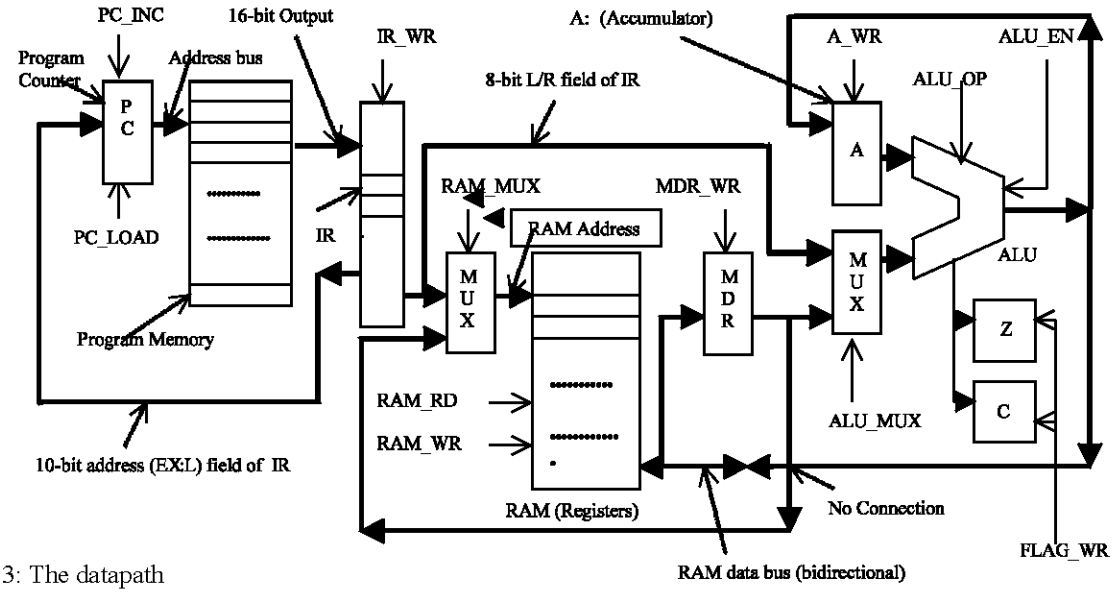

Fig. 3: The datapath

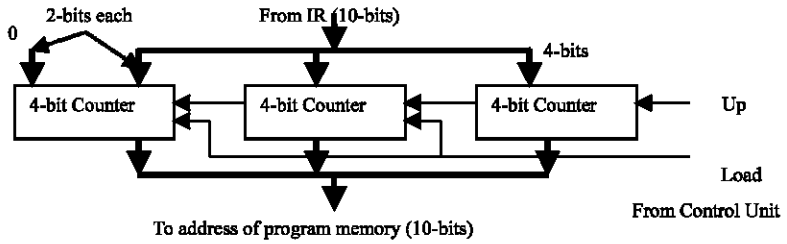

Fig. 4: Program counter register

enable, ALU Mux selection, RAM address Mux selection, PC load and PC increment signals; and write signals for the registers A, MDR, IR and the flags. These signals will be generated by the control unit described in section 5.

ALU design: Of course it is possible to build the ALU by using components such as 74181, but we believe it is better to let the student design custom ALUs from components such as GALs, PALs, or EPROMS. Here, we will present our technique for building 8-bit ALUs 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 5 shows an 8-bit ALU built by using two 27512 EPROMS. If smaller EPROMs are available, the code can be easily modified to use the smaller ICs, but more stages will be used in this case. Note that the address bus is used for inputs and data bus is used for outputs.

The ALU must perform a set of operations required for executing instructions in the instruction set. The following operations are sufficient for the instructions defined in our instruction set. Let A and B be the inputs and F be the output.

1. F = NOT A. One's complement of A.
2. F = A AND B.
3. F = A OR B.
4. F = A XOR B.
5. F = A+B.
6. F = A-B.
7. F = A+1.
8. F = A-1.
9. F = A.
10. F = B.
11. F = A ROL 1 through carry.
12. F = A ROR 1 through carry

The following algorithm is used to generate the program files for the two 27512 EPROMS. This algorithm is designed for a 2-stage ALU but can easily be modified for more ALUs with more stages.

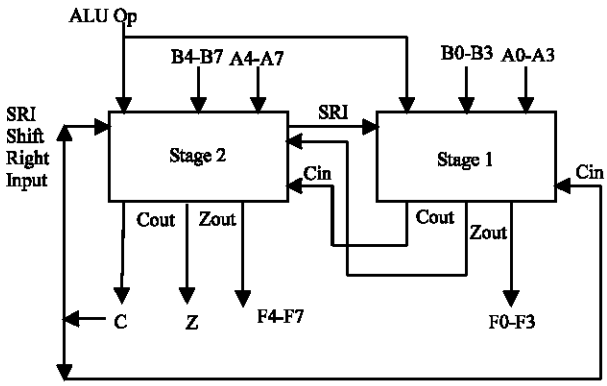

Fig. 5: Block diagram of the two-stage ALU

```
Algoritm ALuGenStage ( int stage) // Pseudocode C code
    Assign the bits for A, B, Cin, Zin and SRI(bit for rotate operation)
    Open Binaryfile
    Address = inputs = [b0--b13] // bits 0 to 13
    FOR address =0 to Address = 2^(Number of inputs)
        ; extract the operands from the address bits and shift them; appropriately
        A = [b0--b3]            // bits 0--3
        B= [b4--b7]>>4          // bits --7
        Cin =[b8]>>5;
        SRI = [b9] >>4;         // Put bit in fifth position
        OP = [b10-13]>>10
        Zin =b[14] >>9;         //Put in fifth position
        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 DEC:   if (stage == 1) F = A-1 ; else F = A- Cin ;break ;
            case AND:   F = A  and B;          if (stage == 2) F = F | SRI;
            case OR:    F = A | B;
            case XOR:   F =  A ^ B;
            case NOT:   F = ~A;
            case PASSA: F = A;    if (stage == 2) F = F | SRI;
            case PASSB: F = B;    if (stage == 2) F = F | SRI;
            case ROL:   F = A << 1;
                        F = F |Cin;  // First bit is carry from previous stage
            case ROR:   F =  (A|SRI) >> 1;
                        if (stage == 1) {Cin = A and 0x0001;} // Cin = A0
                        Cout = Cin <<4; // Cout = Cin
                        F = F|Cout;
                        // In the above, for stage 1, Cout = A0,
                        // For Stage 2 Cout = Cin which is Cout of first stage
                        // which is A0, so A0 goes to carry flag
        End Switch
        Zout = F   0x0F;
        if (Zout ==0) Zout = 0x20;else Zout = 0;
        if (stage == 2) Zout = Zout  and  Zin;
        F = F  and  0x01F; F = F|Zout;
        Store the output F byte to file.
    End FOR
```

Table 2: Control signals to control the datapath

Control signal	Abbreviation	Description
Program Counter Control	PC_INC, PC_LOAD	Control signals for the program counter
Instruction Register	IR_WR	Enable writing to instruction Register
ALU MUX select	ALU_MUX	ALU Multiplexer control line
RAM (Register file) read	RAM_RD	Enable reading from register file
MDR Write	MDR_WR	Enables writing on the MDR register
ALU enable	ALU_EN	Enables the ALU output
RAM MUX select	RAM_MUX	RAM Multiplexer Control Line
ALU operation	ALU_OP	The Specified ALU Operation
A Write enable	A_WR	Enables writing on the Working register
RAM (Register file)write	RAM_WR	Enables writing on the register file
Flags Write	*FLAG_WR*	Write the carry and Zero D-Flip Flops

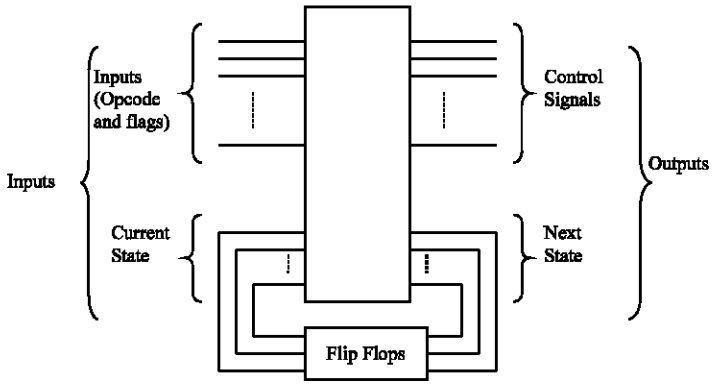

Fig. 6: Finite state machine

THE CONTROL UNIT

This part is the most difficult and time consuming part to design and implement. Traditionally this part is implemented as hardwired control or microcode control. Because of the simplicity of the hardwired design and for education purposes we choose to implement the control unit using the hardwired control methodology. This unit is abstracted as a Finite State machine shown in Fig. 6. The control unit is responsible for asserting and un-asserting all control signals necessary for the datapath to function properly. In our approach, we based our design on using the most primitive (i.e., off-the-shelf) components that are available to the students at affordable prices. The control signals of our datapath components are described in more details in Table 2.

First we must categories the instructions in the instruction set into groups such that all related instructions are placed in one group (Table 3). All instructions in a given group require the same set of states in the state diagram to be 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 and control signal values are determined on state bases rather than instruction bases. For simplicity, the bit operation instructions and the call and return instructions implementation will not be described in this paper and usually they are left for the students for extra credit.

A state diagram that executes the group of instructions in C3 is shown in Fig. 7. Note that it is possible to execute some instructions in fewer states and also it is possible to reduce the number of states, but the states shown simplify the design of the control unit. The operations in each state and the asserted signals are shown in Table 4.

Although there are many possible implementations, we have chosen the states in Fig. 7 to facilitate the control unit implementation. In fact, this technique will still work for different state diagrams that implements the instruction set. A simple implementation will use all input signal including the current state, Z, C, D (direction) and the opcode as inputs to the controller the output is the control signal values. It is also possible to simplify the control unit further by breaking it into two units as shown in Fig. 8. Unit 1 generates the control signal independent of the states and unit 2 generates signals that are dependent on the state. Since our instruction has been carefully chosen to simplify the design in general, the ALU_OP, ALU_MUX signals can be generated by a separate logic unit (ROM, PLA, or PAL). Also, notice that the states are sequential except when exiting state 2,

hence, the opcode is actually needed to determine the state following state 1. Thus, only three bits will be needed as input to the second unit because there are 7 possible branches after S1.

The ALU_OP is easily figured from the instruction, for example ADDAR indicates that the operation is ADD. Also the ALU_MUX should select Literal (L) for if the operand is Literal and MDR if the operand is Register (R). Decoding instructions and determining next state after state 1 (decode state) is determined based on the opcode, the destination field (D), Zero flag, (Z) and the Carry Flag (C). The result of the decode stage is the next state and the selection of second ALU operand either Register or Literal (Table 5).

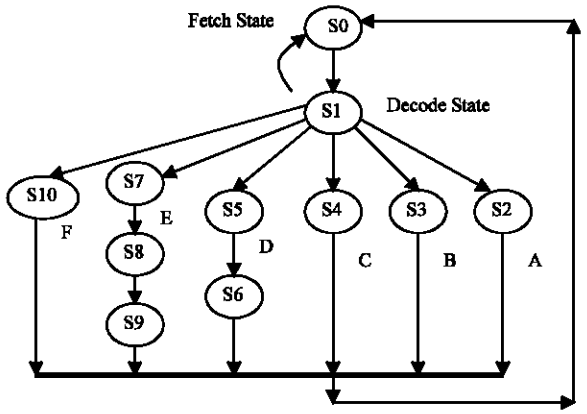

Fig. 7: The state transition diagram

Table 3: Categorizing instructions into groups

Group A OP A,L OP A,R with d = A	Group B OP R OP A,R With d = R	Group C (Jump)	Group D (Load A indirect}	Group E (Store A indirect to R)	Group F MOVBC
MOVLA	MOVAR	JMP	MOVIRA	MOVIAR	MOVBC
ADDLA	ADDAR	JZ			
SUBLA	SUBAR	JNZ			
ANDLA	ANDAR	JC			
ORLA	ORAR	JNC			
XORLA	XORAR				
MOVRA	ROLC				
ADDAR	RORC				
SUBAR	INCR				
ANDAR	DECR				
ORAR	NOTR				
XORAR					

Timing consideration: Before implementing the control, the correct timing diagram for control signals asserted for a group of instructions must be determined. This process is necessary to ensure that there are no glitches and the data is written at the correct triggering edges 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 the RAM which contains the register file is level triggered (active low signal). Also, since the ALU bus and the RAM bus are shared, the ALU output and the RAM read signal cannot be enabled together. The states S8 and S10 have been added to guarantee glitch free design.

To guarantee these timing requirements, two approaches are possible: in the first one, all register clock signals (write) are generated by the controller, this is an easy approach and but it requires extra states. In the second approach, the state register is triggered by clock negative edge and the data is written to registers A, MDR…etc at the clock's positive edge. The register clock signals (A_WR_CLK, MDR_WR_CLK, IR_WR_CLK, FLAG_WR_CLK) are generated by *anding* the corresponding write enable signals with the clock. Note that this will generate a glitch free write clocks, however, the opposite will not. In this study, we have used the second approach for generating the clock signals for A, MDR, IR and the flags. However, we used the first approach in states 8, 9 and 10.

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 as shown in Fig. 9. Similar timing diagrams must be determined for other groups but are not discussed here and can be deduced from Fig. 9.

Table 4: Control Signals Asserted in Each State

State	Group	Operation	Control Signals (other signals are in their initial value, Some signals are active Low)
S0	ALL	IR? RAM[PC]	IR_WE, ALU_EN, RAM_MUX = L
S1	ALL	MDR?RAM[L]	RAM_RD, PC_INC, RAM_MUX = L
S2	A	A?ALU	ALU_EN, RAM_MUX = L, A_WR, FLAG_WR
S3	B	RAM[L]?ALU	ALU_EN, RAM_MUX = L, RAM_WR, FLAG_WR
S4	C	PC?IR[0--9]	PC_LOAD
S5	D	MDR?RAM[MDR]	RAM_RD, RAM_MUX=MDR
S6	D	A?MDR	ALU_EN, RAM_MUX=MDR, A_WR
S7	E	Stabilize address	RAM_RD, RAM_MUX=MDR
S8	E	RAM[MDR]?A	ALU_EN, RAM_MUX=MDR, RAM_W
S9	E	Avoid Glitches	ALU_EN, RAM_MUX=MDR
S10	F	C?IR[0]	ALU_EN, RAM_MUX = L, FLAG_WR

L: Literal, IR[0] = L[0] = first bit in literal. IR[0--9] = address

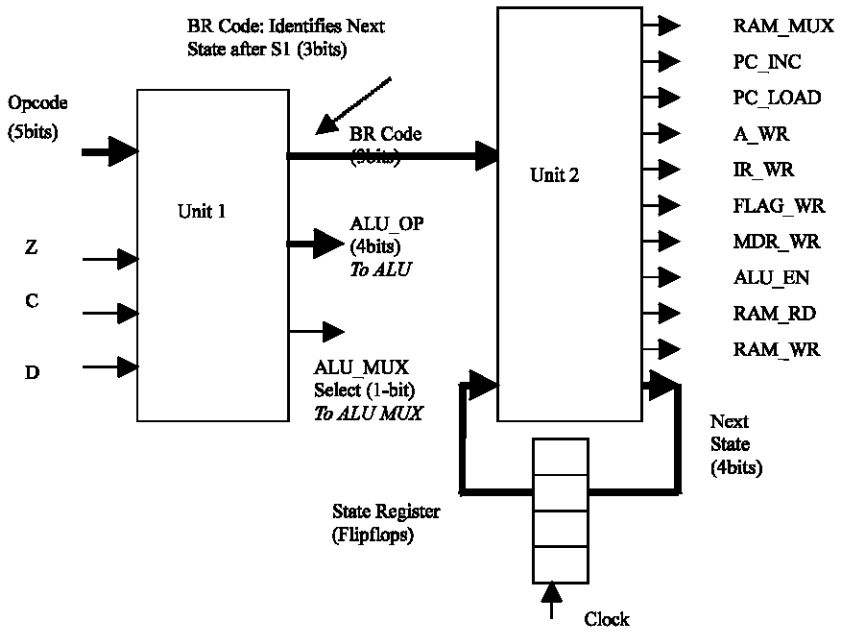

Fig. 8: Block diagram of the control unit

Table 5: Control signals

Inputs					Outputs		
Type	Opcode	D	Z	C	ALU_OP	Next State	ALU_MUX
1	Op A, L (in Group A)	A	-	-	Figured From opcode	BR_S2	Select L
2	Op A, R (in Group A)	A	-	-	Figured From opcode	BR_S2	Select MDR
3	Op A, R (in Group B)	R	-	-	Figured From opcode	BR_S3	Select MDR
4	OP R (in Group B)	R	-	-	Figured From opcode	BR_S3	Select MDR
5	MOVIRA (Group D)	A	-	-	PASS MDR	BR_S5	Select MDR
6	MOVIAR (Group E)	R	-	-	PASS A	BR_S7	Select MDR
7	JMP	-	-	-	-	BR_S4	-
8	JZ		0		-	BR_S0	-
9	JZ		1		-	BR_S4	-
10	JNZ		1		-	BR_S0	-
11	JNZ		0		-	BR_S4	-
12	JC		-	0	-	BR_S0	-
13	JC		-	1	-	BR_S4	-
14	JNC		-	1	-	BR_S0	-
15	JNC		-	0	-	BR_S4	-
16	MOVBC R (Group F)	A	-	-	PASS MDR	BR_S10	Select MDR

Note: D = A means destination is A, hence D = 0 and D = R means destination = Register (D = 1)

Generating the control hardware: Based on the above explanation, we will present an algorithm for generating a controller implemented from EROMs/EEPROMs. Although the algorithms presented are written for the state diagram presented above, they are easily modified for any state diagram.

Algorithm Generate Unit 1
Address = inputs
For address = 0 to $2^{(\text{Number of inputs})}$
Extact opcode, d, C and Z from address
Opcode = address[0—4], d = address[5], C= address[6], Z= address[7]
Match opcode, d, Z, C with Type in Table 5
Generate BR_CODE, ALU_OP, ALU_MUX as in Table row matching Type.
Output = ALU_OP:BR_CODE:ALU_MUX, where : is concatenate operation
Write output to File
End For

For Unit 2, two 8-bit ROMs will be needed Unit2_ROM1, Unit2_Rom2
Algorithm Generate Unit 2

```
Note: is concatenate operation
Address = inputs=[BR_CODE:PRESENT_STATE=
STATE_REGISTER]
O1 =[RAM_MUX:PC_INC:PC_LOAD:A_WR:IR_ WR:
FLAG_WR:MDR_W:ALU_EN]
O2=[RAM_RD:RAM_WR]
Let Generate_Table4_Outputs(Si) be a function that sets
O1 and O2 as shown in the corresponding table row in
Table 4
For address = 0 to 2^(Number of inputs)
        Extract BR_CODE, PRESENT_STATE from
        address
        BR_CODE = address[0—2],
        PRESENT_STATE=address[3-6]
        IF (PRESENT_STATE) NOT in (S0, S1 …S10) //
        invalid state
                Unit2_ROM1 = 0, Unit2_ROM2 =0 // that
                is send to initial state 0
        ELSE {
                [O1:O2] = Generate_Table 4_Outputs
                (PRESENT_ STATE)
                IF (PRESENT_STATE == S0)
                        NEXT_STATE = S1
                ELSE IF( PRESENT_STATE in
                (S2,S3,S4,S6,S9,S10))
                        NEXT_STATE = S0
                ELSE IF (PRESENT_STAT in ( S5, S7, S8))
                        NEXT_STATE =
PRESENT_STATE +1
                ELSE IF (PRESENT_STATE== S1){
                        //NEXT_STATE determined by
                BR_CODE
                        SWITCH (BR_CODE){
                    CASE BR_S2 :NEXT_STATE = S2
                    CASE BR_S3 :NEXT_STATE = S3
                    CASE BR_S4 :NEXT_STATE = S4
                    CASE BR_S5 :NEXT_STATE = S5
                    CASE BR_S7 :NEXT_STATE = S7
                    CASE BR_S10 :NEXT_STATE = S10
                }// end switch
        }
        Unit2_ROM1 = O1
        Unit2_ROM2 = O2:NEXT_STATE:0:0// the zeros
are just pad
    }
    Write Unit_ROM1 to ROM1_File
    Write Unit2_ROM2 to ROM2_File
End For
```

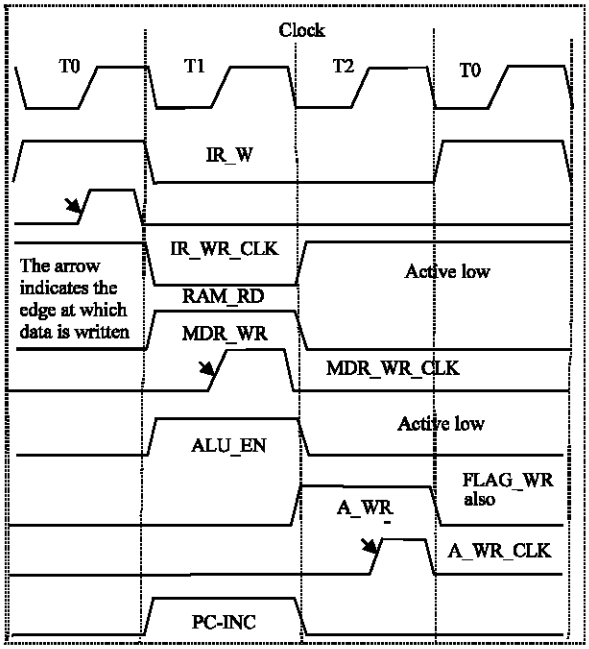

Fig. 9: Timing diagram for group a instructions

PROGRAMMING THE PROCESSOR

Since we have chosen Harvard style memory architecture as shown in Fig. 3, the program has to be stored in the program memory that can be implemented using either RAM or EPROM/EEPROM. However, in order to use RAM, an interface circuitry would be necessary and will be described shortly.

If an EPROM/EEPROM is to be used, the students can write the program in machine or in assembly language. However, writing in machine language requires editing the EPROM hex/binary files; which is acceptable because of the simplicity of the instruction set. Of course, writing in assembly language requires that an Assembler be available. Actually-a note worth mentioning here-is that writing the assembler can also be assigned to the students which would be a good educational project.

In order to facilitate programming our CPU, we have developed an Assembler that generates files with binary and hex format that can be downloaded to either EPROM or RAM.

The assembler and disassembler: Our assembler is a two-pass assembling process. In the first pass the assembler finds the labels and stores them in a symbol table and also checks for syntax errors. In the second pass, the assembler translates the assembly code into machine code.

In order to simplify witting the assembler, the following rules has been used: (1) one instruction per line, (2) labels start with a special character $ (3) anything that follows a semi-column is a comment.

Since the program memory is 16-bit wide, the assembler produces two briary files; one for the low byte and one for the high byte. The files can be downloaded to EPROMs by using any universal programmer. However, if a RAM is used, the files will be down loaded by the interface circuit. To facilitate debugging, disassembler has also been developed. The following code shows an example of an assembly program that adds the values in registers 20 through 30 and stores the result in A.

Sample Program:

```
DEFINE    Count     1          ; register 1 is a counter
DEFINE    Sum       2          ; sum register
DEFINE    IndxReg   3          ; register 3 used as index
;; This is a comment
          MOVLA     10         ; A= 10
          MOVAR     Count      ; Count = 10
          MOVLA     20         ; stat addres of
                                 registers to be added
          MOVAR     IndxReg    ; IndexReg = 20
          MOVLA     0          ; A =0
          MOVAR     Sum        ; Sum =0
$L1       MOVIRA    IndexReg   ; A =
                                 Register[IndexReg]
          ADDAR     Sum        ; Sum  = Sum + A
          DECR      Count
          JNZ       $L1
          MOVRA     Sum        ; A = Sum
$forever  Jmp       $forever
```

Assembler Algorithm

Pass1 : Initialize AddressCount =0
tok = Get_Next_Token()
If tok == DEFINE get following two tokens
- First is an Id and second is value
- Store (Id, value) in symbol table
- Otherwise print error and exit

if tok == label (starts with $)
- Store (label, AddrCount)
- Get next token (should be instruction)

if tok== Instruction
- Get following tokes (1 or two) depending on instruction
- otherwise error.

// All syntax errors should be caught in Pass1

Pass2: // Initialize AddressCount =0
If tok == DEFINE skip
if tok == label (starts with $)
- Get following token (should be instruction)

if tok== Instruction but not jump or conditional jump or call
- Get following tokens (operands: 1 or 2) depending on instruction.

if the operand is not numbers get their values from symbol table
if tok == jump or conditional jump or code
- get the following token which should be a label
- get the label value from symbol table

Substitute the values for opcode, operand, or label in a 16-bit value and write first byte to file1 and second to file2

INTERFACE CIRCUIT

Although building this circuit is not a necessity for building and testing the processor, its implementation significantly simplifies programming, testing and debugging the processor. There are two main purposes for the interface circuit, the first is to program the CPU by downloading programs in the program memory if a RAM is used and the second is to test and debug the whole CPU. Of course using a RAM for the program memory will simplify programming the CPU, because it takes away the burden of erasing and re-programming EPROMs that had to be removed and reinstalled in the circuit for reprogramming.

Program download circuit: Our interface circuit is a microcontroller-based circuit which interfaces serially to a PC. We used PIC16F877 (Microchip, 2005) for implementing this circuit and developed a C++ based PC program as interface program. The interface program downloads a binary/or hex file to the upper or lower bank of the 16-bit program memory. Figure 10 shows the block diagram of the circuit that is used to connect the PC to the CPU program memory.

Another possibility is to first use EPROM to test the processor and then a small Monitor program can be written and stored in EPROM. This monitor program, in addition to adding UART as an SFR to the processor can be used to download programs to RAM. Such a technique can be used without using the PIC controller. However, using a PIC controller requires almost the same amount of hardware.

Debugging and testing interface circuit: This circuit is needed in order to debug, test and examine the results of programs. The interface circuit and a corresponding PC

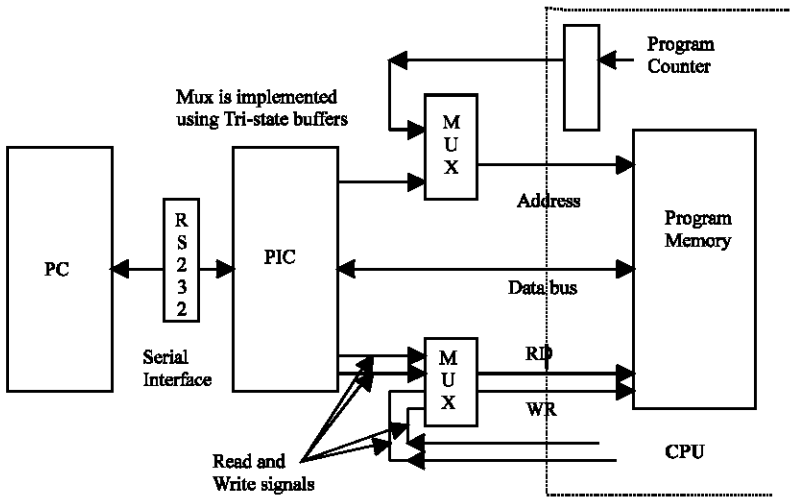

Fig. 10: Connecting the PC to Processor Memory

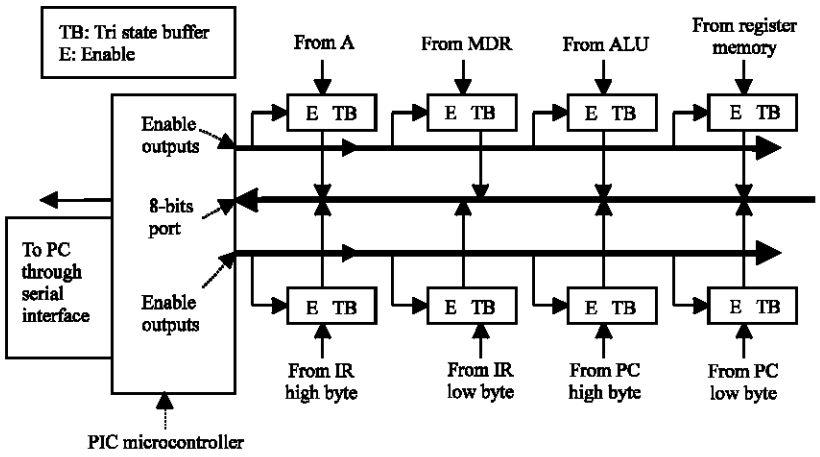

Fig. 11: Debugging interface circuit

program can read the registers in the data path as well as the register memory to test correct execution by the processor. A block diagram of the interface circuit is shown in Fig. 11. Note that the register memory address interface is not shown in the Fig. 11 the interface is similar to the one shown in Fig. 10. The microcontroller circuit along with the PC program can examine the contents of the CPU registers when run with very slow frequency or under single step execution mode.

CONCLUSIONS

In this study, we have presented a methodology by which a processor design lab course can be easily conducted at universities with limited resources (i.e., universities in devolving counties). This method is based on using easily obtainable (off-the-shelf) components as the basic building block, then, on developing programs to aid in designing these components as well as loading necessary data into these components. An assembler that translates assembly instruction into the required binary format was also presented. In addition, a hardware tool that can be used to interface the implemented processor with a PC is also given. This tool is used for debugging purposes and also used to load application programs into processor memory. This methodology has shown that it is very applicable and enhances students' knowledge on critical issues pertaining to processor design. Moreover, students also get a very good hands-on-experience in hardware design, especially CPU design. This method has been used for the past three years to conduct a computer design lab course in the computer engineering department at An-Najah national university, in Palestine. During which students have shown great interest in studying this

course. We encourage and recommend integrating this course in the curriculum of all computer engineering departments in developing countries and adapting this methodology to conduct the course. Future work can focus on the implementation of call, return, set, clear and other related jump instructions.

REFERENCES

Gang, Q., 2003. Introducing the concept of design reuse into undergraduate digital design curriculum. International Conference on Microelectronics Systems Education (MSE'03), 1-2 June 2003, Anaheim, California.

Gottlib, D.B. and N.P. Carter, 2003. Microprocessor interfacing laboratory. International Conference on Microelectronics Systems Education (MSE'03) 1-2 June 2003, Anaheim, California.

Hersch, R.D., 1994. Integrated Theory and Practice in Microprocessor System Design Course. Proc. Euromicro, North Holland, Amsterdam, pp: 227-232.

Nicoud, J.D., 1991. Dedicated Tools For Microprocessor Education. IEEE Micro January/February, 11: 62-68.

Nicoud, J.D. and R. Sommer, 1975. Modular logic elements, microprocessors and peripherals improve efficiency of teaching and development. Proc. Compcon, Spring, IEEE Computer Society Press, Los Alamitos, Calif, pp: 127-130.

Ozcan, M.B., 1996. Integration of software tools in software engineering education. 9th Conference on Software Engineering Education (CSEE) 21-24 April 1996, Daytona, FL.

Pastor, J.S., I.G. Lopez, F. Gomez-Arribas and J. Martinez, 2004. A Remote laboratory for debugging FPGA-based microprocessor prototypes. IEEE International Conference on Advanced Learning Technologies (ICALT'02) August 2004, pp: 86-90.

January 25, 2007 by Naseem