SIMULATION AND FPGA IMPLEMENTATION OF A SIMPLE COMPUTER

Dr. Kasim M. Al-Aubidy, Ra'ed F. Al-Bader, Ala'a A. Smadi
Computer Eng. Dept,
Philadelphia University,P O Box 1,
Jordan, 19392
E-mail: kma@philadelphia.edu.jo

KEYWORDS
Computer systems, Processor design, Simulators, Microprogramming.

ABSTRACT

FPGA technology offers the potential of designing high performance systems at low cost. FPGAs have been used for many computational tasks, and this paper presents the micro-operation simulation of a basic computer and its implementation on an FPGA. Also, it covers the design of an assembler for the designed computer, which can be used for educational purposes. Such implementation offers good code density, easy customization, easily developed software, small area, and high performance at low cost.

INTRODUCTION

Field programmable gate arrays (FPGAs) consist of programmable logic blocks, which can each implement a small amount of digital logic, and programmable routing which allows the logic block inputs and outputs to be connected to form larger circuits (Ciletti 2002). FPGAs have become a popular technology for creating digital systems since they can lead to a shorter time-to-market for designs than application-specific circuits (ASICs) and allow design modifications to be made after system creation (Welch and Carletta 2000, Krid and Masmoudi 2005). The primary method used for validating a design with most FPGA design flows is simulation.
A Hardware Description Language (HDL) model is more precise and concise than a logic circuit model, for system design and realization. It permits the user to create a design without having to first choose a device for implementation (Muresan 1997, Mano 1993). An HDL model is the key for the transition from System-On-a-Board to System-On-a-Chip.
This paper presents the simulation and FPGA realization of a basic computer and its assembler which can be used for educational purposes. This simulation is a set of micro-operations that represent the register transfer statements of all operations that can be implemented.

BASIC COMPUTER DESIGN AND SIMULATION

A basic computer is designed with minimum hardware components to perform the basic tasks that a computer should perform (Mano 1993). The organization of the computer is defined by its internal registers.

The internal organization of a digital system is defined by sequence of micro-operations that perform on data stored in its registers. The computer is capable of executing various micro-operations and can be instructed to perform a sequence of operations.
Instruction codes together with data are stored in memory. The computer reads each instruction from memory and places it in a control register. The control unit then interprets the binary code of the instruction and proceeds to execute it by issuing a sequence of micro-operations.
An instruction code is a group of bits that instruct the computer to perform a specific operation. It is made up of 16 bits, and divided into three parts:
- Two bits (I1,I0) to specify the addressing mode.
- Four bits binary code to specify the operation.

I_1	I_0	Op-Code	Address/Data
15	١٤	١٣ ١٠	٩ ٠

Figure 1: Instruction format.

Table 1 illustrates the addressing mode of the proposed computer, which is used to:
- Reduce the number of bits in the address field of the instruction, and
- Give user flexibility in dealing with counters, pointers …etc

Table 1: Addressing Modes.

I_1	I_0	Addressing mode
0	0	Immediate addressing *($ address)*
0	1	Direct addressing
1	1	Indirect addressing *(# address)*

The computer needs registers for manipulating data and a register for holding a memory address. Nine registers are required for the proposed computer, as shown in Table 2. Some registers (such as AC, MAR, and MBR) may receive data from several multiplexed sources.

Table 2: List of Registers for the Basic Computer.

Register	Bits No.	Function
Memory Buffer Reg. (MBR)	16	Holds memory word
Memory Address Reg. (MAR)	10	Holds address for memory
Address Reg. (AR)	10	Holds operand/instruction add.
Accumulator (AC)	8	Processor register
Counter Reg. (CR)	8	Holds count for loops
Program Counter (PC)	10	Holds address of instruction
Operation Reg. (OPR)	4	Holds code of operation
Input Reg. (INP)	8	Holds input character
Output Reg. (OUTR)	8	Holds output character

Table 3: Basic computer instruction formats

MEMORY REFERENCE INSTRUCTIONS				
Symbol	Code for the 6 left most bits			Description
	I1I0=00	I1I0=01	I1I0=11	
NOP	000000	010000	110000	No operation
ADD	000001	010001	110001	Add memory word to AC
SUB	000010	010010	110010	Subtract memory word from AC
AND	000011	010011	110011	AND memory word to AC
OR	000100	010100	110100	OR memory word to AC
XOR	000101	010101	110101	XOR memory word to AC
LDA	000110	010110	110110	Load memory word to AC
STA	000111	010111	110111	Store content to AC in memory
BUN	001000	011000	111000	Branch unconditionally
BSA	001001	011001	111001	Branch and save return address
DSZ	001010	011010	111010	Decrement and skip if zero
LDC	001011	011011	111011	Load CR
RET	001100	011100	111100	Return
BZ	001101	011101	111101	Branch if zero
BC	001110	011110	111110	Branch if carry
REGISTER REFERENCE INSTRUCTIONS (Hexadecimal code)				
CLA	3C01			Clear AC
CLS	3C02			Clear all status flags
CMA	3C04			Complement AC
SRA	3C08			Shift right AC
SLA	3C10			Shift left AC
INC	3C20			Increment AC
HALT	3C40			Terminate program
INPUT-OUTPUT REFERENCE INSTRUCTIONS				
INP	BC01			Input character to AC
OUT	BC02			Output character from AC
SKI	BC04			Skip on input flag
SKO	BC08			Skip on output flag
ION	BC10			Interrupt on
IOF	BC20			Interrupt off
SFI	BC40			Set input flag
SFO	BC80			Set output flag

The basic computer has eight registers, a memory unit, and a control unit. Paths must be provided to transfer information from one register to another and between memory to registers. A more efficient scheme for transferring information in a system with many registers is to use a common bus. A multiplexer can be used to design the common bus. The connection of the registers to the common bus system is shown in Fig. 2.

Computer Instructions:

The basic computer has three basic instruction code formats. A memory-reference instruction uses ten bits to specify either an address or an operand and two bits to specify the addressing mode (I_0,I_1). For immediate addressing it is 00, 01 for direct addressing, and 11 for indirect address. The register-reference instructions are recognized by the operation code 1111 and a 00 in the left most bits of the instruction. A register-reference instruction specifies an operation on the AC register. An operand from memory is not needed; therefore, the other 10 bits are used to specify the operation to be executed. Similarly, an input-output instruction does not need a reference to memory and is recognized by the operation code 1111 and 11 in the left most bits of the instruction. The remaining 10 bits are used to specify the type of the input-output operation. This technique allows having up to 35 different operations, as given in Table 3.

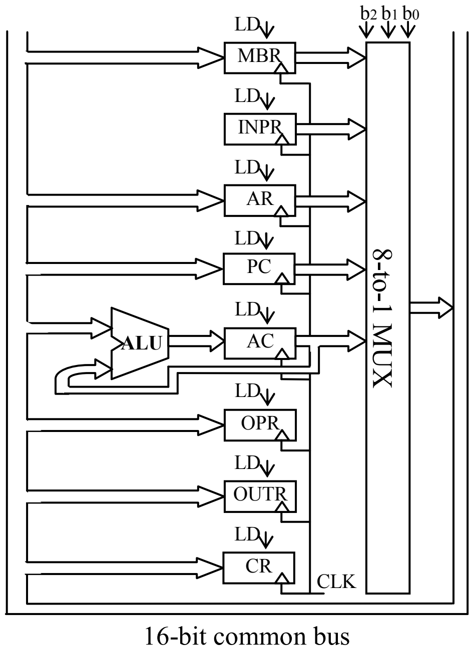


Figure 2: Basic computer registers connected to a common bus.

Timing and Control:

The timing for all registers in the basic computer is controlled by a master clock generator. The control signals are generated by the control unit and provide control inputs for the multiplexer in the common bus, control inputs in processor registers, and micro-operations for the accumulator. The control unit, Fig. 3, consists of three decoders, a sequence counter, and a number of control logic gates. The control unit gets the operation code from the OPR through a 4x16 decoder. Bits 14 & 15 of the instruction code are transferred to two flip-flops designated by the symbols $I_1 \& I_0$. Bits 0 through 9 are applied to the control logic gated directly from the MBR through the common bus. The outputs of the counter are decoded into 4 timing signals T0 through T3. The sequence counter (*SC*) can be incremented or cleared. Most of the time, the counter is incremented to provide the sequence of timing signals out of the 2x4 decoder. Once, the counter is cleared, causing the next active timing signal to T0. The decoder is used to determine the cycle to be performed.

INSTRUCTION CYCLES

A program residing in the memory unit of the computer consists of a sequence of instructions. Each instruction cycle is subdivided into a sequence of sub-cycles. The value of three flip-flops is entered into a decoder to determine the cycle to be served, as illustrated in Table 4. In this computer each instruction cycle is divided into the following five sub-cycles:

Table 4: Cycles Combination.

Flip-flops			Cycle
G	**F**	**R**	
0	0	0	Fetch & Decode
0	0	1	Indirect
0	1	0	Direct
0	1	1	Execution
1	0	0	Interrupt

Fetch and Decode Cycle

Initially, the program counter is loaded with the address of the first instruction in the program. The sequence counter is cleared to 0, providing a decoded timing signal T0. After each clock pulse, sequence counter is incremented by one, so that the timing signals go through a sequence T0, T1, T2 and T3. The micro-operations for the fetch and decode cycle can be specified by the following statements:

$$\boxed{\begin{array}{l}
C_0T_0 : PC \rightarrow MAR \\
C_0T_1 : M(MAR) \rightarrow MBR,\ PC+1 \rightarrow PC \\
C_0T_2 : MBR < I_0, I_1 > \rightarrow (I_0, I_1), \\
\qquad MBR < OP\_Code > \rightarrow OPR, \\
\qquad MBR < Address > \rightarrow AR \\
\overline{I_0}C_0T_3 :\ 1 \rightarrow R,\ 1 \rightarrow F, 0 \rightarrow G \quad (\text{ExecuteCycle}) \\
\overline{I_1}I_0C_0T_3 : O \rightarrow R,\ 1 \rightarrow F, 0 \rightarrow G \quad (\text{DirectCycle}) \\
I_1I_0C_0T_3 : 1 \rightarrow R,\ 0 \rightarrow F, 0 \rightarrow G \quad (\text{IndirectCycle})
\end{array}}$$

Indirect Cycle

At this cycle, the effective address of the operand is to be read from memory. The AR holds the address of memory word which contains the effective address of the operand.

$$\boxed{\begin{array}{l}
C_1T_0 : AR \rightarrow MAR \\
C_1T_1 : M(MAR) \rightarrow MBR \\
C_1T_2 : MBR < Address > \rightarrow AR \\
C_1T_3 : 0 \rightarrow R,\ 1 \rightarrow F, 0 \rightarrow G \quad (\text{Direct Cycle})
\end{array}}$$

Direct Cycle

The effective address of the operand may be read during two time pulses. Therefore, to disable the delay of waiting for T3, the sequence counter may be cleared at time T2. Thus, the next time pulse will be T0 of the execution cycle and not T3 of the indirect cycle.

$$\boxed{\begin{array}{l}
C_2T_0 : AR \rightarrow MAR \\
C_2T_1 : M(MAR) \rightarrow MBR, \\
C_2T_2 : 1 \rightarrow R,\ 1 \rightarrow F,\ 0 \rightarrow G, \\
\qquad 0 \rightarrow CC \quad (\text{Execution Cycle})
\end{array}}$$

Execute cycle

At this cycle, the fetched instruction (register-reference, memory reference, or input-output-reference) is executed. The type of the instruction is decided according to the following table:

Table 5: Combinations of Instruction Types

q15	I0	I1	Instruction Type
0	X	X	MRI
1	0	0	RRI
1	0	1	IOI

Interrupt Cycle

The interrupt cycle is initiated after the execute cycle if the interrupt flip-flop (*INF*) is equal to 1. The flip-flop is set to 1 manually using a switch, and it may be set during the execution of the program by the instruction ION. The flip-flop is reset to 0 whenever the interrupt is served or by the instruction *IOF*.

This basic computer serves the interrupt by saving the next sequential instruction in memory address 0, and then it starts execution from address 1 in the memory. The micro-operations required for this instruction are:

$$\boxed{\begin{array}{l}
C_4T_0 : PC \rightarrow MBR < Data >, 0 \rightarrow INF \\
C_4T_1 : 0 \rightarrow PC, 0 \rightarrow MAR \\
C_4T_2 : MBR < Data > \rightarrow M(MAR), PC+1 \rightarrow PC \\
C_4T_3 : 0 \rightarrow R, 0 \rightarrow F, 0 \rightarrow G \quad (\text{Fetch Cycle})
\end{array}}$$

Table 6: Micro-operations of all instructions.

Instr.	**Micro-operations.**
Memory Reference Instructions (MRIs)	
NOP	No micro-operation
ADD	$q_0C_3T_0 : AC + MBR < Data > \rightarrow AC$
SUB	$q_1C_3T_0 : AC - MBR < Data > \rightarrow AC$
AND	$q_2C_3T_0 : AC \bigcap MBR < Data > \rightarrow AC$
OR	$q_3C_3T_0 : AC \bigcup MBR < Data > \rightarrow AC$
XOR	$q_4C_3T_0 : AC \oplus MBR < Data > \rightarrow AC$
LDA	$q_5C_3T_0 : MBR < Data > \rightarrow AC$
STA	$q_6C_3T_0 : AR \rightarrow MAR, AC \rightarrow MBR < Data >$ $q_6C_3T_1 : MBR < Data > \rightarrow M(MAR)$
BUN	$q_7C_3T_0 : AR \rightarrow PC$
BSA	$q_8C_3T_0 : PC \rightarrow MBR < Address >, AR \rightarrow MAR$ $q_8C_3T_1 : MBR < Address > \rightarrow M(MAR),$ $AR + 1 \rightarrow AR$ $q_8C_3T_2 : AR \rightarrow PC$
DSZ	$q_9C_3T_0 : MBR < Data > -1 \rightarrow MBR < Data >$ $q_9C_3T_1 : MBR < Data > \rightarrow M(MAR),$ $MBR < Data > \rightarrow AC$ $q_9C_3T_2 : IF(AC = 0) Then PC + 1 \rightarrow PC$
LDC	$q_{10}C_3T_0 : MBR < Data > \rightarrow CR$
BZ	$q_{11}C_3T_0 :$ if $ZF = 1$ then $PC + 1 \rightarrow PC$ else $MBR < Add > \rightarrow PC$
BC	$q_{12}C_3T_0 :$ if $CF = 1$ then $PC + 1 \rightarrow PC$ else $MBR < Add > \rightarrow PC$
Register Reference Instructions (RRIs)	
CLA	$rB_0T_0 : 0 \rightarrow AC$
CLS	$rB_1T_0 : 0 \rightarrow$ All Status Flags
CMA	$rB_2T_0 : \overline{AC} \rightarrow AC$
SRA	$rB_3T_0 : SHR(AC) \rightarrow AC, 0 \rightarrow AC(7)$
SLA	$rB_4T_0 : SHL(AC) \rightarrow AC, 0 \rightarrow AC(0)$
INC	$rB_5T_0 : AC + 1 \rightarrow AC$
HALT	$rB_6T_0 : 0 \rightarrow StartIndex$
Input/Output Reference Instructions (IORIs)	
INP	$pB_0T_0 : INPR \rightarrow AC, 0 \rightarrow FGI$
OUT	$pB_1T_0 : AC \rightarrow OUTR, 0 \rightarrow FGO$
SKI	$pB_2T_0 : IF(FGI = 1)$ then $PC + 1 \rightarrow PC$
SKO	$pB_3T_0 : IF(FGO = 1)$ then $PC + 1 \rightarrow PC$
ION	$pB_4T_0 : 1 \rightarrow INF$
IOF	$pB_5T_0 : 0 \rightarrow INF$
SFI	$pB_6T_0 : 1 \rightarrow FGI$
SFO	$pB_7T_0 : 1 \rightarrow FGO$

Note that:
- $q_{15}\overline{I_1}\,\overline{I_0}C_3 = r$ (Common to all RRIs).
- $MBR(n) = B_n$ [MBR (0-9) that specifies the operation].
- $q_{15}I_1\overline{I_0}C_3 = p$ (Common to all input-output instructions).
- $MBR(n) = B_n$ [MBR (0-9) that specifies the operation].

Complete Computer Description

The instruction cycle flowchart is shown in Fig. 4. Figure 5 shows the schematic diagram of the basic computer. It is compound of one 16-bit register, three 10-bit registers, four 8-bit registers, one 4-bit register, eight D-Flip-flops, one 1024x16-bit RAM, one ALU, one 8-bit inverter, two 2-to-1 Multiplexer, one 4-to-1 Multiplexer, one 4-to-16 decoder, one 3-to-8 decoder, one 2-to-4 decoder, and one 2-bit sequence counter.

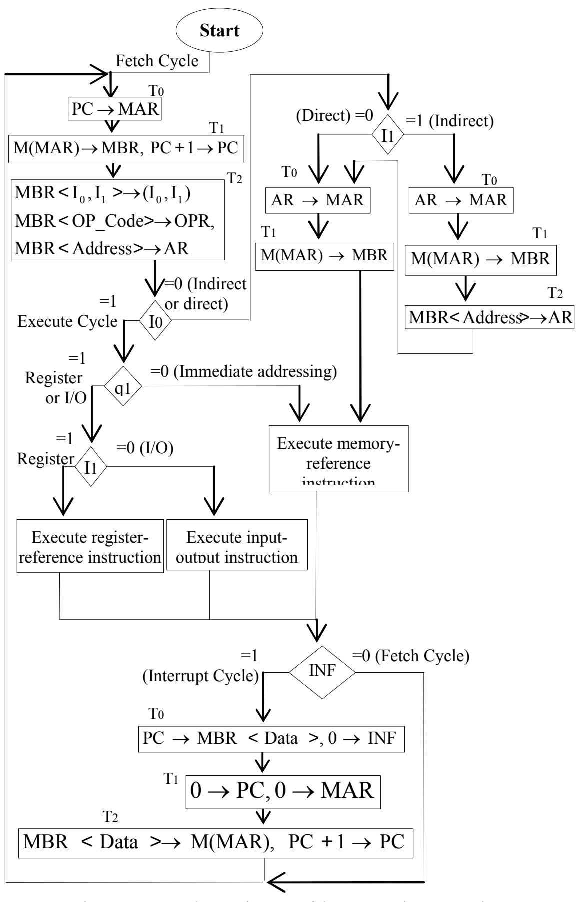


Figure 4: Flowchart of instruction cycle.

ASSEMBLER DESIGN

A two pass assembler has been designed and implemented to write assembly programs and use the output of the assembler to run these programs on the basic computer.

Input and Output Files

Figure 6 shows the files used as input and those generated as output by the assembler. These are;
- Source File (input): It is a text file containing the source program to be assembled. It has a *".asm"* extension. It consists of two segments, the code segment followed by the data segment. The data segment starts with "data:".
- Binary Code File (output): It contains the assembled statements represented in binary form. This file is stored in the block memory, and it has a *".dat"* extension.

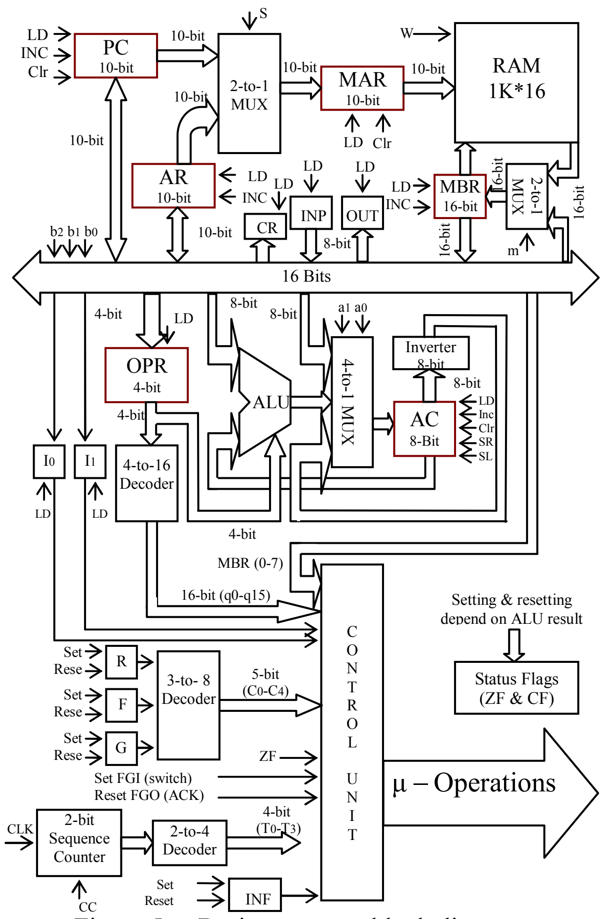


Figure 5: Basic computer block diagram.

- Hex Code File (output): It contains the assembled statements in hexadecimal form. This file is stored in the external memory, and it has a *".mem"* extension.
- Listing file (output): It consists of the source file statements, the assembled code, and the Branch Vector Table *BVT*. This file has a *".lst"* extension.

The Basic Computer Assembly Language character set consists of the following subset of the standard ASCII character set:

- Lower-case letters (a to z).
- Digits 0 through 9.
- Blanks (ASCII 32).

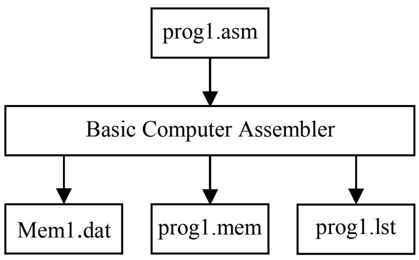


Figure 6: Assembler input and output files.

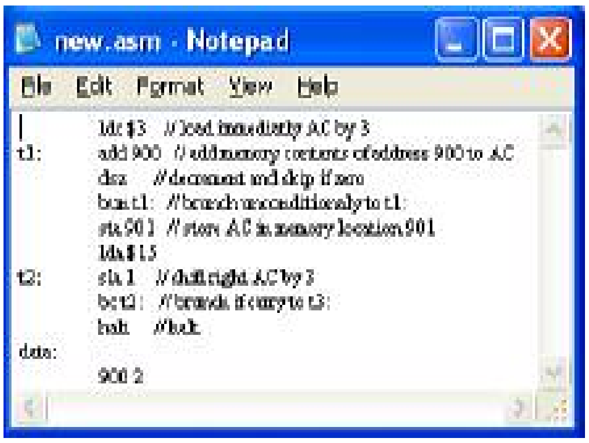
Assembler Instructions

The Basic Computer Assembly Language consists of lines of text where each line contains only one instruction and an optional comment. An instruction takes one or zero operands. Table 7 shows all instructions and their appropriate operands and addressing modes.

Table 7: Possible Types for Each Assembly Instructions.

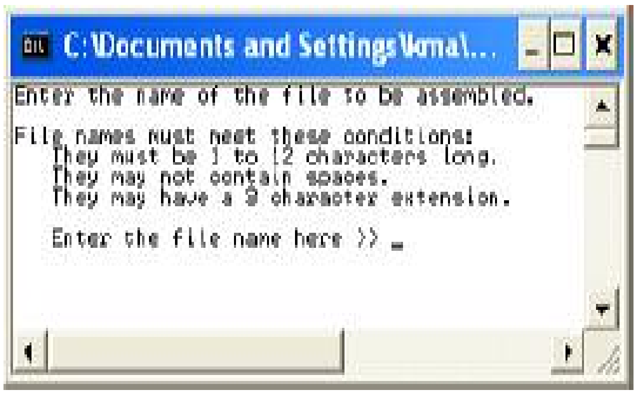
Instr.	Type of operand	Function
nop	-	No operation
add	Immediate data/Add.	Add operand to AC
sub	Immediate data/Add.	Subtract operand from AC
and	Immediate data/Add.	AND operand to AC
or	Immediate data/Add.	OR operand to AC
xor	Immediate data/Add.	XOR operand to AC
lda	Immediate data/Add.	Load operand to AC
sta	Address	Store AC content in memory
bun	Lable	Branch unconditionally
bsa	Lable	Branch and save return add.
dsz	-	Decrement and skip if zero
ldc	Immediate data/Add.	Load CR
ret	-	Return
bz	Lable	Branch if zero
bc	Lable	Branch if carry
cla	-	Clear AC
cls	-	Clear all status flags
cma	-	Complement AC
sra	Immediate data	Shift right AC
sla	Immediate data	Shift left AC
inc	-	Increment AC
halt	-	Terminate program
inp	-	Input character to AC
out	-	Output character from AC
ski	-	Skip on input flag
sko	-	Skip on output flag
ion	-	Interrupt on
iof	-	Interrupt off
sfi	-	Set input flag
sfo	-	Set output flag

To branch to a line of assembly code, a symbolic destination must be placed at the beginning of the line of code. The comment symbol is //. A line that begins with // is considered a comment line. It is printed to the list (.lst) file but will not be encoded into the hex (.mem) and the binary (.dat) files. Comments may be added to lines that contain program code.



```
        ldc $3   // load immediatly AC by 3
t1:     add 900  // add memory contents of address 900 to AC
        dsz      // decrement and skip if zero
        bun t1:  // branch unconditionally to t1:
        sta 901  // store AC in memory location 901
        lda $15
t2:     sla 1    // shift right AC by 3
        bc t2:   // branch if carry to t2:
        halt     //halt
data:
        900 2
```

Figure 7: Example of a source code.

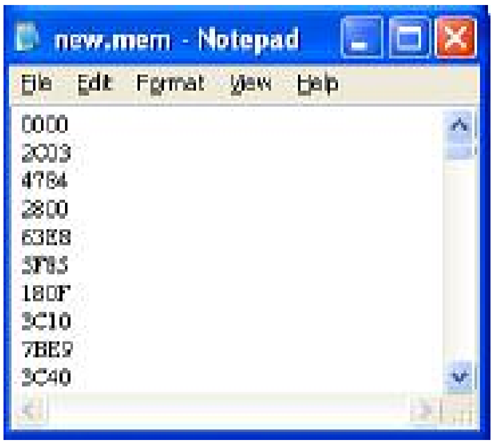


```
Enter the name of the file to be assembled.

File names must meet these conditions:
    They must be 1 to 12 characters long.
    They may not contain spaces.
    They may have a 3 character extension.

    Enter the file name here >> _
```

Figure 8: Assembler Window.



```
0000
2C03
4784
2800
63E8
3F85
180F
3C10
7BE9
3C40
```

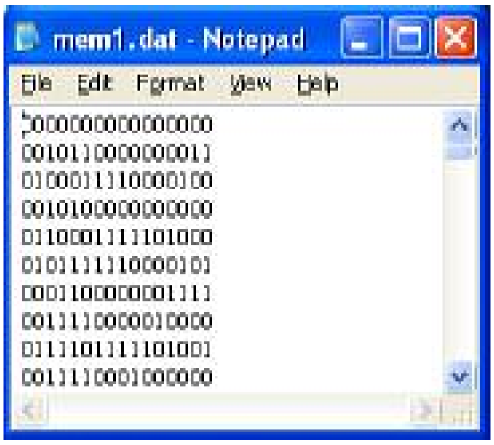
Figure 10: Beginning of Hex file.

Example:

To use the assembler, follow these steps:

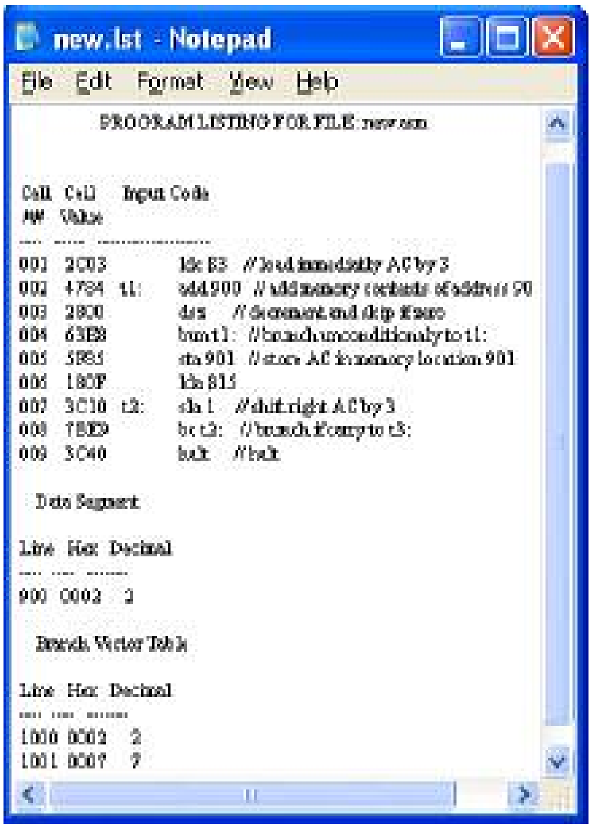
1. Write the program in Basic Computer Assembly language using a text editor such as Microsoft Notebook, see Fig. 7.
2. Save the file as text only, using a "*.asm*" extension.
3. Make sure your assembly language file is in the same directory as the Basic Computer Assembler.
4. Using Windows Explorer or a command window, start the assembler. Figure 8 shows the assembler window.
5. When the assembler comes up, enter the name of your file, being sure to include the *.asm* extension.

Then three new files are generated in the directory. The hex and list files will have the same name as the assembly language file with extensions *.mem* (see Fig. 9), and *.lst* (see Fig. 10). The binary file have a "*mem1.dat*" name, as in Fig. 11.



```
0000000000000000
0010110000000011
0100011110000100
0010100000000000
0110001111101000
0101111110000101
0001100000001111
0011110000010000
0111101111101001
0011110001000000
```

Figure 11: Beginning of mem1.dat file.



```
        PROGRAM LISTING FOR FILE: new.asm


Cell  Cell    Input Code
M#    Value
----  -----   ---------------
001   2C03          ldc E3   //load immediately AC by 3
002   4784   t1:    add 900  // add memory contents of address 90
003   2800          dss      // decrement and skip if zero
004   63E8          bun t1:  //branch unconditionaly to t1:
005   3F85          sta 901  // store AC in memory location 901
006   180F          ldc 815
007   3C10   t2:    sla 1    //shift right AC by 3
008   7BE9          bc t2:   //branch if carry to t3:
009   3C40          halt     //halt

  Data Segment

Line  Hex  Decimal
----  ---  -------
900   0002    2

  Branch Vector Table

Line  Hex  Decimal
----  ---  -------
1000  0002    2
1001  0007    7
```

Figure 9: Listing file.

FPGA IMPLEMENTATION

Field programmable gate arrays (FPGAs) are a class of programmable logic devices based on an array of logic cells surrounded by a periphery of input/output cells. These programmable integrated circuits can be programmed in the field to implement specific design function.

The basic computer architecture is created from the ground up as a scalable architecture, covering the basic operations in 16-bit processor domain. Figure 12 illustrates the FPGA board (Xilinx Spartan-3).

The basic computer uses a four-character, seven segment LED display controlled by FPGA user-I/O pins. The interface for the seven-segment display is relatively simple, so it is used to display contents of system registers.

Four push button switches are used to moving-up registers sequence up/down, system reset, and user clock.

There are eight slide switches (SW_0-SW_7) in the system. Switches (SW_{4-7}) are used as input data, switches (SW_1 & SW_2) are used as external interrupts, while switch (SW_0) is used to indicate if the clock is a system clock or a user clock.

Figure 13 shows the design procedure of the basic computer.

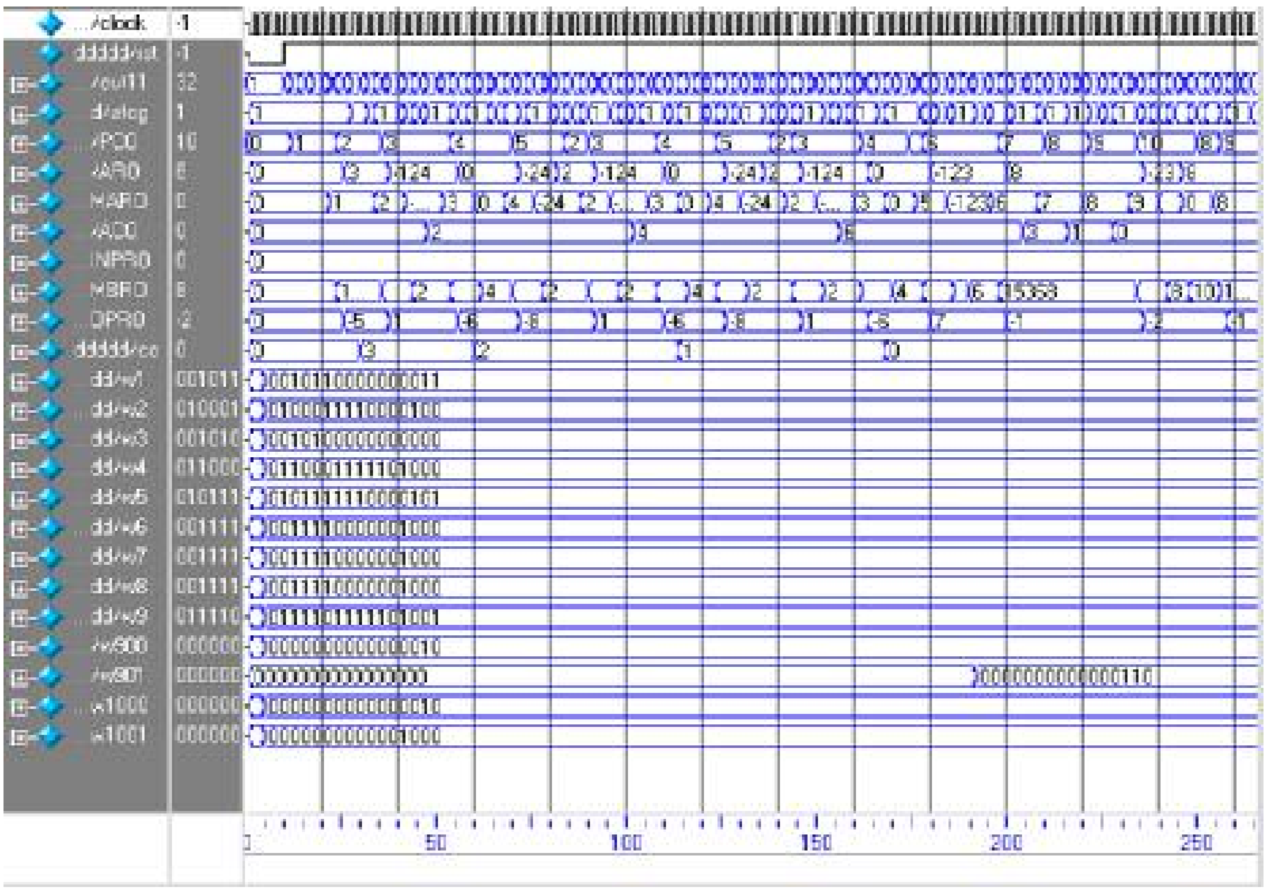


Figure 12: Waveforms Generated by the Program in Fig. 7.

Figure. 13: Spartan-3 board.

Table 8 shows the different characteristics of the FPGA board we have implemented on.

Table 8: Device Characteristics

Device: Xilinx Spartan-3 XC3S200FT256 Slices: 3584, 256-ball thin Ball Grid Array				
System gates	Logic cells	Multipliers blocks (18*18)	Digital Clock Managers	Max I/O Signals
200 k	4320	12	4	173
Select RAM		In-system programmable configuration PROM	Fast Asynchronous SRAM	
18K-bits Blocks	Max RAM K-bits			
12	216	2 M-bits	1 M-byte	

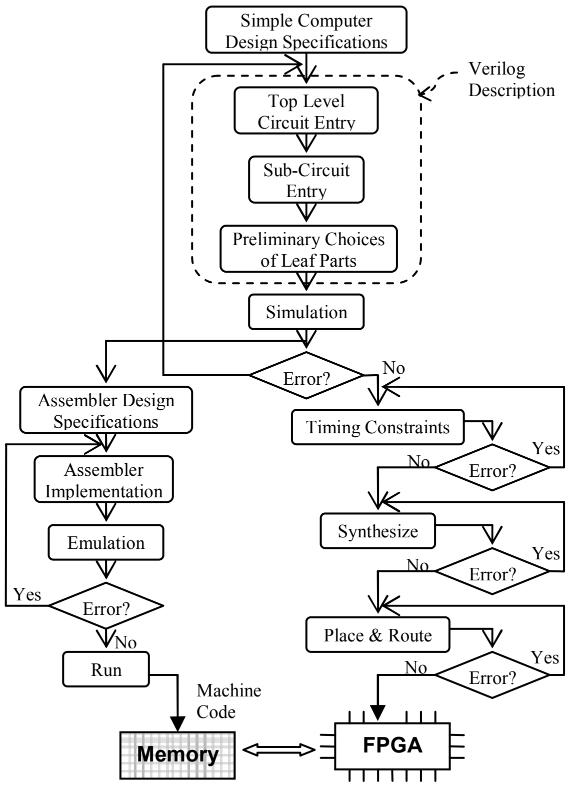


Figure 14: Design Flow Map.

Table 9: Macro Statistics

ROMs	16x7-bit
Registers	17
Counters	3
Multiplexers	7
Tri-states	27
Decoders	2
Adders/Subtractors	4

Tables 9 and 10 show the macro statistics and FPGA resources used respectively.

Table 10: Utilization Summary

Item	Used	Available	Percentage
Slices	433	3584	12%
Slice Flip Flops	114	7168	1%
4 input LUTs	802	7168	11%
Bonded IOBs	23	173	13%
GCLKs	2	8	25%

The circuit delay implemented on the FPGA is mostly due to routing delays, rather than logic block delays (Betz 2005). The average connection delay for this design is: 6.357 ns, the maximum pin delay is: 12.054 ns, the maximum operating frequency is 73.465 MHz, and the total memory usage is 90032 Kbytes.

CONCLUSIONS

- In this paper, the simulation of a basic computer and the implementation of an assembler has been presented.
- The micro-operation of the computer design and assembler design are implemented on an FPGA, since it offers good code density, easy customization, easily developed software, high performance and small area.
- The codes of the various modules are implemented and tested. The complete system can be achieved by linking up the modules together.
- The system has been tested with a program which utilizes every instruction as well as exercises the critical paths of the chip. It was found to be operational up to 73.465 MHz.

REFERENCES

Betz V. & J. Rose 2005, "FPGA Routing Architecture: Segmentation & Buffering to Optimize Speed & Density", *http://www.eecg.toronto.edu/~jayar/pubs/betz/fpga99 betz.pdf*

Ciletti M. D. 2002, "Advanced Digital Design with the Verilog HDL", *Prentice Hall,.1st edition, USA.*

Digilentic Co. 2004, "Spartan-3 Starter Kit Board User Guide", *April 26, http://www.digilentinc.com.*

Krid M. & D. S. M. Masmoudi 2005, "FPGA Implementation of a Feedforward Neural Network", *3rd Intr. Conf. on Systems, Signals & Devices, Vol.4, Tunisia.*

Mano M. M. 1993, "Computer System Architecture", *Prentice-Hall. 3rd edition, USA.*

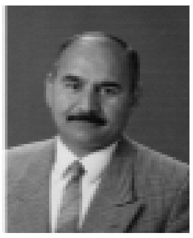
Muresan V, D. Crisu, & X. Wang 1997, "From VHDL to FPGA: A Case Study of a Fuzzy Logic Controller" *Proceedings of the Intr. Conf. of Young Lectures & PhD Students, pp(83-90), , Hungary.*

ST Microelectronics 2004, "Design Guid for Xilinx FPGA Power Management Systems", *Version 1.2, pp(1-11).*

Welch J.T. & J. Carletta 2000, "A Direct Mapping FPGA Architecture for Industrial Process Applications", *Proceedings of the 2000 IEEE Intr. Conf. on Computer Design: VLSI in Computers & Processors, ICCD2000, USA.*

.

AUTHOR BIOGRAPHY



Kasim Al-Aubidy received his BSc and MSc degrees in control and computer engineering from the University of Technology, Iraq in 1979 and 1982, respectively, and the PhD degree in real-time computing from the University of Liverpool, England in 1989. He is currently an associative professor in the Department of Computer Engineering at Philadelphia University, Jordan. His research interests include fuzzy logic, neural networks, genetic algorithm and their real-time applications. He is the winner of Philadelphia Award for the best researcher in 2000. He is also the chief editor of the Asian Journal of Information Technology. He has coauthored 3 books and published 54 papers on topics related to computer applications.