5-STEP DESIGN METHODOLOGY FOR A GENERAL PURPOSE CPU USING STANDARD CPLDs/FPGAs

Karim Salman, Michael B. Anderton

Middle Tennessee State University

Abstract

We present a novel hardware Central Processing Unit (**CPU**) design methodology based on a 5-step approach. The method starts with a definition of the target **CPU** internal components and data and address size. The method is applicable to a higher level of abstraction and complexity. However, for ease of illustration a basic **CPU** with a minimum size instruction set is selected. The instruction set complies with the instruction set completeness criteria. The instruction format is likewise chosen to be simple and illustrates the way our methodology is implemented. The **CPU** is implemented on an Altera FPGA/CPLD Flex10K device using schematic approach with the Altera MAX+Plus II software CAD. The design was simulated and tested using Altera UP2 board.

Introduction

CPU design for engineering/engineering technology students varied widely in objectives and approach[1-7]. For a long time, block diagrams of simple **CPU**s have been used in beginning computer courses, mainly to allow students to visualize how a **CPU** functions. To meet this need, many textbook authors[1,6,7] have devised simple **CPU**s at the block diagram level to illustrate how instructions are executed and data are manipulated. Obviously, omitting many of the circuit details allows an overall understanding that is usually sufficient for students with little or no experience with digital circuits. However, students in accredited Electronics and Computer Engineering and Engineering Technology programs, have a more thorough background in digital circuit design. They are able to understand how instructions are decoded, what control signals are required for datapath operation, and how those control signals are generated. By examining this extra level of detail, students can better tie the new material to principles they have already learned. The low end approach is mainly descriptive of either a hypothetical ad hoc designs that remain to be implemented and tested[4,6,8], or description of operation and design criteria of available well established designs[10]. On the high end, the approach usually departs and concentrates on describing complex architectures. This paper describes an architecture that is amenable to hardware implementation. A 5-step design methodology is presented. Although the paper adopts an already described architecture[6],

the 5-steps described here allow a novice to be able to tackle, design and implement more challenging and advanced **CPU** architectures.

The design methodology

We begin by establishing some requirements for the **CPU** in mind. However, we are attempting to finally design and implement a simple architecture without sacrificing the important and basic issues in such design. For example we need first to address the following issues: word length, memory size, and registers. These can also be restated or modified without in another design once the methodology is adhered to. Since we are adopting a previously published design, the first two steps in our methodology would be to define our basic computer:

1. 16-bit data wordlength and 12-bit memory address. The wordlength size can address double precision data more readily than 8-bit designs, yet it adds little overhead in our methodology. The memory size will not be too prohibitive for hardware implementation. Our choice for the target hardware platform is Altera Flex10K FPGA/CPLD device[11]. The memory size can therefore be increased if a newer device is selected.
2. The hardware components will consist of :
    a. A memory unit with 2k words of 16-bits each.
    b. Ten registers.
    c. Two decoders: a 3x8 operation decoder and a 4x16 timing decoder.
    d. A 16-bit multiplexer based common bus.
    e. Control Logic Unit.
    f. Arithmetic and Logic Unit (**ALU**).

In addition to a 4-bit Sequence Counter (**SC**), there are 9-registers as shown in Fig. 1:

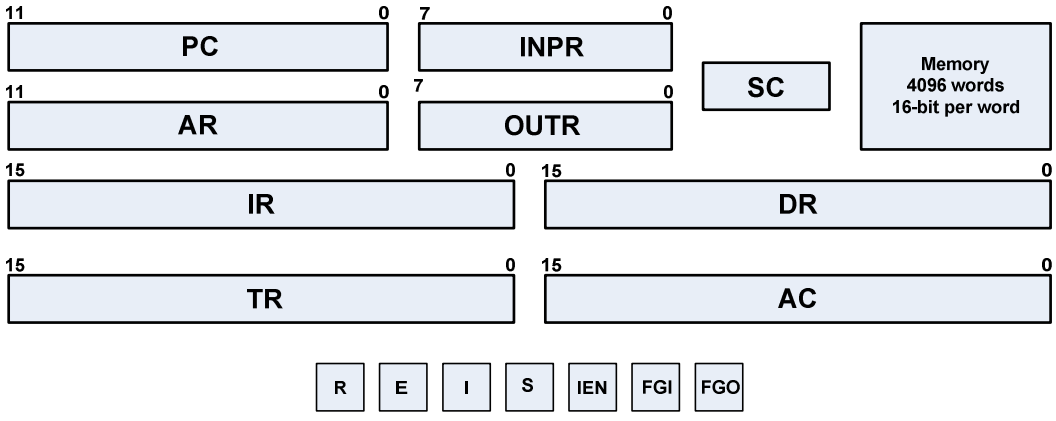

Fig. 1 Basic computer registers and memory.

- Two address registers, the Program Counter (**PC**), and the Address Register (**AR**). Both are 12-bit wide. The program counter takes charge of the instruction normal flow whereas the Address Register main duty is to

address the memory. This is an important concept in the design because it makes the design easy to implement from the hardware point of view, and allows interrupt handling without adding additional address registers.

- A 16-bit instruction register (**IR**). The instruction format that dictates this choice will be addressed later.
- A 16-bit accumulator (**AC**). The input to this accumulator is connected directly to the output of the Arithmetic and Logic Unit (**ALU**). The **AC** output permanently feeds and shares one leg of the **AL**U with the input register (**INPR**).
- Two additional 16-bit data registers, a Data Register (**DR**), and a Temporary Register (**TR**). The **DR** is permanently connected to the second leg of the **ALU**. Therefore, the **ALU** is physically separated from the main bus. This feature is important to avoid metastabilities in the sequential design.
- Two 8-bit registers. One is the Input Register (**INPR**), and the other one is the Output Register (**OUTR**). These facilitate communications with external input and output devices.
- A collection of 7 disjoint flip flops that collectively act as the **CPU** status register. The Interrupt Register (**R**), The Stop Register (**S**), The Carry Register (**E**), the Addressing Mode Register (**I**), the Interrupt Enable Register (**IEN**), the Flag Input Register (**FGI**), and the Flag Output Register (**FGO**). These are individually controlled as will be detailed later.

3. A simple instruction format, as shown below. By dividing the instruction word into three fields allows the choice of three basic references: Memory, Register, and Input/Output. The memory instruction format can be subdivided into Direct and Indirect through the I-bit. The register instruction format makes use of the 12-least significant bits of the instruction word. Without decoding, these can address 11 register instructions and the Halt instruction. No memory reference is used here. The input/output addressing mode utilizes only part of the available code in the 12-least significant bits when the 4-most significant bits are all 1's. With three bit IR_{12-14} we are allowed to address 7-memory reference instructions. An instruction decoder is used to produce 8 control signals $D_0, D_1, ..., D_7$. The last signal determines whether the instruction is memory reference when $D_7 = 0$ or non-memory reference when $D_7 = 1$. However, these can also be in a direct or indirect mode depending on the status of the IR_{15} bit which is the **I**-bit. The instruction set, though looks too few, but they are sufficient to satisfy the general instruction-set-completeness requirements, that requires minimum set of instructions in each of the following categories:
4. *Arithmetic* (**ADD, INC, CLA**), *Logical* (**AND, CMA, CME, CLE**), and *Shift i*nstructions (**CIR, CIL**). The three arithmetic instructions are sufficient to address all necessary arithmetic operations. For example, subtraction can be achieved by 2's arithmetic through 1's complementing **AC** (**CMA**), and incrementing (**INC**). Multiplication and Division is achieved through successive additions and subtractions. Logical operations are likewise achieved through the Boolean two primitives **AND**, and any of the two instructions **CMA**, **CME**.

Arithmetic and logical shift operations are achieved through the two instructions **CIR** or **CIL** and proper choice of the E flip flop. Signed or unsigned arithmetic operations are delegated to a software task by testing and controlling the most significant bit of the accumulator.

- *Memory Reference Instructions*. Moving information to and from memory and processor registers can be achieved through the instructions **AND**, **ADD**, **LDA**, and **STA**.
- *Program Control Instructions* **BUN, BSA, ISZ, SPA, SNA**, and **SZA**, as well as *processor status check instructions* **SZE, SKI, SKO, ION**, and **IOF**.
- *Input and Output instructions* **INP** and **OUT**.

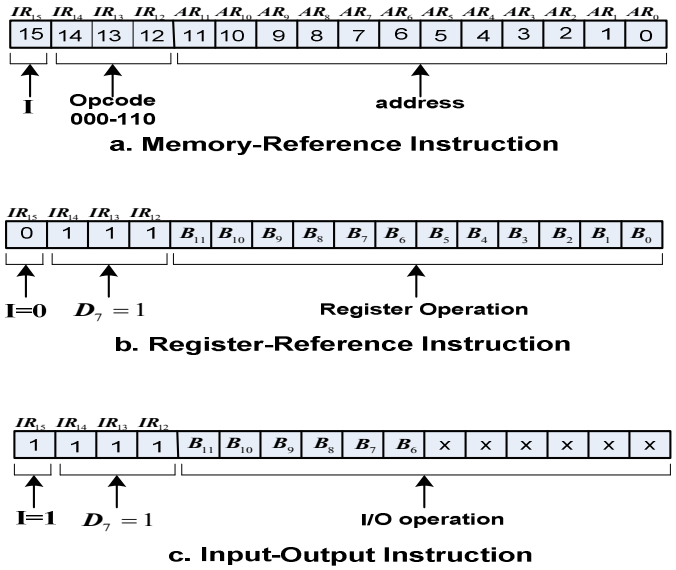

Fig. 2 Instruction format.

5. Control Unit Design. This step represents the bulk and the heart of the **CPU** design. In order to be able to address the complex structure and timing constraints associated with the control unit a sequential design approach must be attempted. This is due to the fact that unlike the **ALU**, which is essentially a combinational circuit, most of the operations in the control unit are basically sequential because they deal with data in or out of the registers at the transitions of the **CPU** clock. Using the *Register Transfer Language* can describe more easily the sequential operations entailed. The normal *Fetch*, *Decode*, and *Execute* phases can also be addressed quite effectively with this simple language. The use of this language can transform the assembly code instructions and operations down to the microprogrammable level in accordance with the transition of the master clock. Since each instruction has to pass through the above three phases in Table 1, and in order to restrict the operations of each instructions within specified time windows, an instruction sequence counter (**SC**) is used that will provide a number of timing pulses $T_0, T_1, ..., T_6$, Fig 3, depending on phase and the instruction being processed.

Table 1 Control Functions and Micro Operations for the Basic Computer.

Fetch	$\bar{R}T_0$:	$AR \leftarrow PC$
	$\bar{R}T_1$:	$IR \leftarrow M[AR],\ PC \leftarrow PC + 1$
Decode	$\bar{R}T_2$:	$D_0, ..., D_7 \leftarrow Decode\ IR(12-14),$ $AR \leftarrow IR(0-11),\ I \leftarrow IR(15)$
Indirect	$\bar{D}_7IT_3$:	$AR \leftarrow M[AR]$
Interrupt	$\bar{T}_0\bar{T}_1\bar{T}_2(IEN)(FGI + FGO)$:	$R \leftarrow 1$
	RT_0:	$AR \leftarrow 0,\ TR \leftarrow PC$
	RT_1:	$M[AR] \leftarrow TR,\ PC \leftarrow 0$
	RT_2:	$PC \leftarrow PC + 1,\ IEN \leftarrow 0,\ R \leftarrow 0,\ SC \leftarrow 0$
Memory Reference:		
AND	D_0T_4:	$Dr \leftarrow M[AR]$
	D_0T_5:	$AC \leftarrow AC \wedge DR,\ SC \leftarrow 0$
ADD	D_1T_4:	$DR \leftarrow M[AR]$
	D_1T_5:	$AC \leftarrow AC + DR,\ E \leftarrow C_{out},\ SC \leftarrow 0$
LDA	D_2T_4:	$DR \leftarrow M[AR]$
	D_2T_5:	$AC \leftarrow DR,\ SC \leftarrow 0$
STA	D_3T_4:	$M[AR] \leftarrow AC,\ SC \leftarrow 0$
BUN	D_4T_4:	$PC \leftarrow AR,\ SC \leftarrow 0$
BSA	D_5T_4:	$M[AR] \leftarrow PC,\ AR \leftarrow AR + 1$
	D_5T_5:	$PC \leftarrow AR,\ SC \leftarrow 0$
ISZ	D_6T_4:	$DR \leftarrow M[AR]$
	D_6T_5:	$DR \leftarrow DR + 1$
	D_6T_6:	$M[AR] \leftarrow DR,\ if\ (DR = 0)\ then\ (PC \leftarrow PC + 1),\ SC \leftarrow 0$
Register-Reference:		
		$D_7\bar{I}T_3 = r$ ($common\ to\ all\ register - reference\ instructions$)
		$IR(i) = B_i$ ($i = 0,1,2,...,11$)
		$r = SC \leftarrow 0$
CLA	rB_{11}:	$AC \leftarrow 0$
CLE	rB_{10}:	$E \leftarrow 0$
CMA	rB_9:	$AC \leftarrow \overline{AC}$
CME	rB_8:	$E \leftarrow \bar{E}$
CIR	rB_7:	$AC \leftarrow shrAC,\ AC(15) \leftarrow E,\ E \leftarrow AC(0)$
CIL	rB_6:	$AC \leftarrow shlAC,\ AC(0) \leftarrow E,\ E \leftarrow AC(15)$
INC	rB_5:	$AC \leftarrow AC + 1$
SPA	rB_4:	$If\ (AC(15) = 0)\ then\ (PC \leftarrow PC + 1)$
SNA	rB_3:	$If\ (AC(15) = 1)\ then\ (PC \leftarrow PC + 1)$
SZA	rB_2:	$If\ (AC = 0)\ then\ (PC \leftarrow PC + 1)$
SZE	rB_1:	$If\ (E = 0)\ then\ (PC \leftarrow PC + 1)$
HLT	rB_0:	$S = 0$
Input-Output:		
		$D_7IT_3 = p$ ($common\ to\ all\ input - output\ instructions$)
		$IR(i) = B_i$ ($i = 6,7,8,9,10,11$)
		$p = SC \leftarrow 0$
INP	pB_{11}:	$AC(0-7) \leftarrow INPR,\ FGI \leftarrow 0$
OUT	pB_{10}:	$OUTR \leftarrow AC(0-7),\ FGO \leftarrow 0$
SKI	pB_9:	$If\ (FGI = 1)\ then\ (PC \leftarrow PC + 1)$
SKO	pB_8:	$If\ (FGO = 1)\ then\ (PC \leftarrow PC + 1)$
ION	pB_7:	$IEN \leftarrow 1$
IOF	pB_6:	$IEN \leftarrow 0$

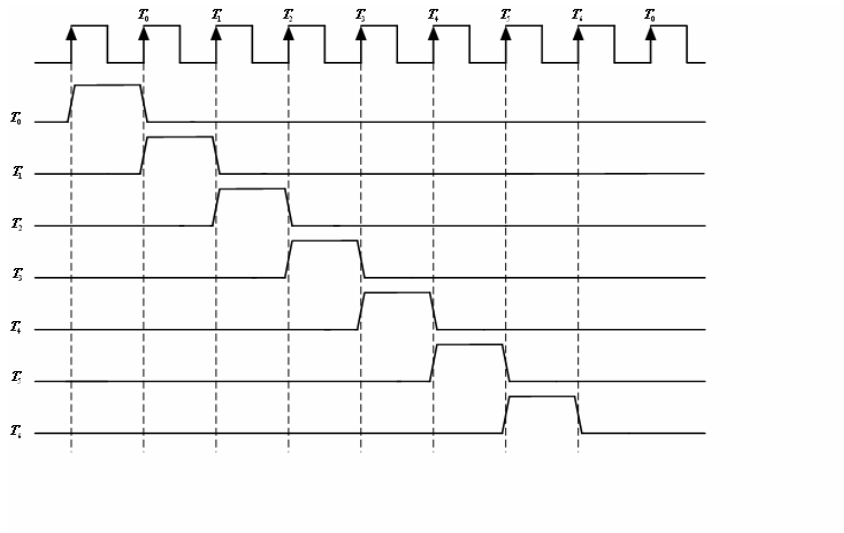

Fig. 3 T timing signals.

Each instruction will be translated into the microprogrammable level through the register transfer language. Table 1 describes all the operations needed for each instruction for the whole processor. Hence this table can be considered as the blue print for this processor. Once this table is formed the rest of the design is simply logical equation formation and implementation. Associated with this table is the internal **CPU** architecture. As depicted in Fig. 4, the bus system has a separate control unit.

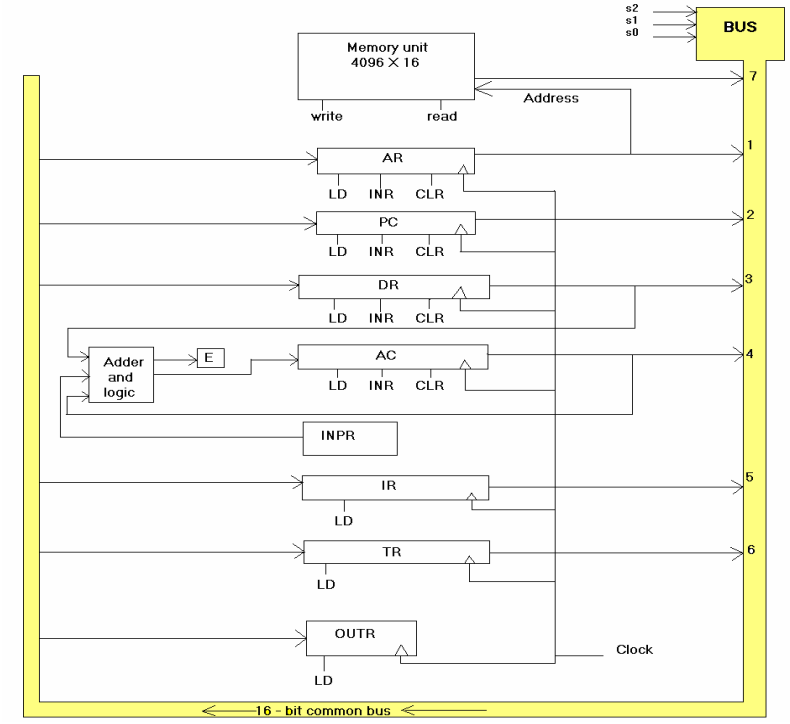

Fig. 4 Basic computer registers connected to a common bus.

Out of three approaches for the bus design, namely Open Collector, Tri State, or Multiplexer, the latter is adopted. Any of the others can be equally selected for the design without undue degradation. At any time only one device can assume control of the bus system. Hence an encoder system with three select signals will ensure that any device can control the bus during any clock cycle, Fig. 5. For example when $S_2S_1S_0 = 111$ the memory is being read and therefore the memory will assume control of the bus.

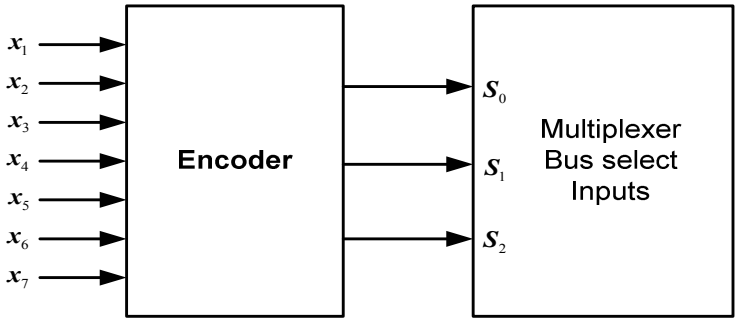

Fig. 5 Encoder for bus selection.

The data in the bus will be transferred to the destination register at the correct clock transition when the particular load signal is active. Several steps are also required to facilitate easy hardware implementation. For example, seven registers share the Load control signal (***LD***) feature, five registers share the Increment control signal (***INR***) feature, and six registers share the Clear control signal (***CLR***) feature. Hence it will be beneficial if all the three control signals are embedded in one 16-bit register model. Enabling / disabling these signals can represent any of the eight registers. Fig. 6 depicts the implementation of 1-bit of this register using Altera MAX+PLUS II FPGA/CPLD CAD[9] schematic approach.

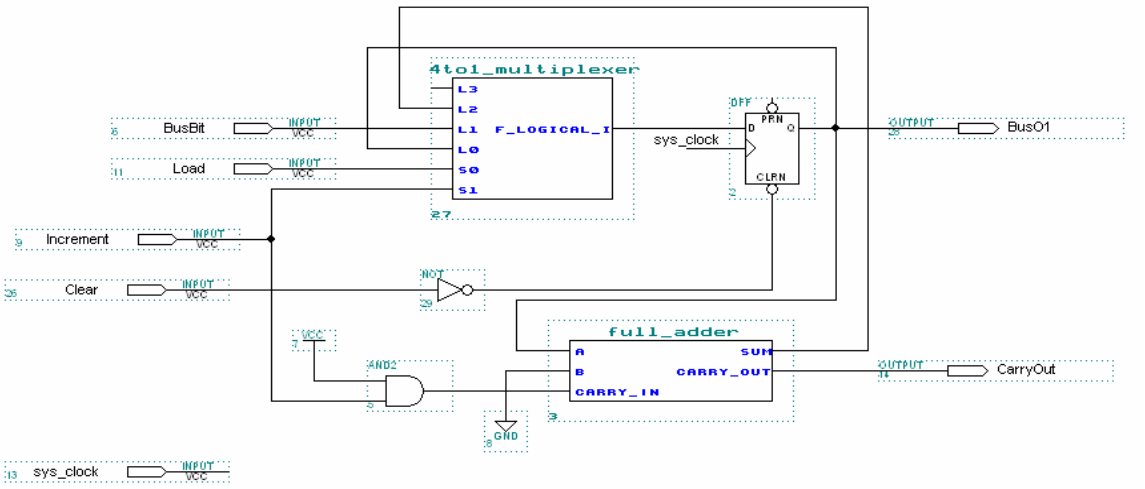

Fig. 6 1-bit of a 16-bit register with load, increment, and clear signals.

As an example, if the **IR** register was selected by the instruction during the fetch cycle, then the memory contents in the bus system should be loaded to this register when the LD of this register is active. Since the fetch cycle is common to all instructions, then we can see from Table 1 that two timing signals are required for this phase, T_0 and T_1. During T_0 the load signal LD of the AR register, i.e. LD_{AR} has to be active. The logic equation for this signal is derived by scanning Table 1 at the fetch cycle and during T_0. This results in the following equation: $LD_{AR} = \bar{R}T_0$. However, LD_{AR} can also be asserted during T_2 or even T_3 if we have indirect reference to the memory. Hence the total logic equation for LD_{AR} would be: $LD_{AR} = \bar{R}T_0 + \bar{R}T_2 + \bar{D}_7IT_3$. In the same way the other two signals can be derived. The INR_{AR} is asserted during the execution phase of the **BSA** instruction, thus: $INR_{AR} = D_5T_4$ and the CLR_{AR} signal is asserted during interrupt only: $CLR_{AR} = RT_0$. Similarly, we can scan for the **AC**: $LD_{AC} = D_0T_5 + D_1T_5 + D_2T_5 + (D_7IT_3 \square IR_{11}) + D_7\bar{I}T_3(IR_9 + IR_7 + IR_6 + IR_5 + IR_{11})$ but $IN_{AC} = D_7\bar{I}T_3 \square IR_5$ and $CLR_{AC} = D_7\bar{I}T_3 \square IR_{11}$. The memory read signal can be derived by scanning Table 1 for the following RTL: (any register) $\leftarrow M[AR]$ to obtain: Read = $\bar{R}T_1 + \bar{D}_7IT_3 + (D_0 + D_1 + D_2 + D_6)T_4$. The Boolean equation for the memory write signal is obtained in a similar way but scanning for $M[AR] \leftarrow$ (any register) instead. On the other hand, the bus control logic is derived from Tables 1 and Table 2.

Table 2 Encoder for Bus Selection Circuit.

Inputs							**Outputs**			**Register selected for bus**
x_1	x_2	x_3	x_4	x_5	x_6	x_7	S_2	S_1	S_0	
0	0	0	0	0	0	0	0	0	0	None
1	0	0	0	0	0	0	0	0	1	AR
0	1	0	0	0	0	0	0	1	0	PC
0	0	1	0	0	0	0	0	1	1	DR
0	0	0	1	0	0	0	1	0	0	AC
0	0	0	0	1	0	0	1	0	1	IR
0	0	0	0	0	1	0	1	1	0	TR
0	0	0	0	0	0	1	1	1	1	Memory

One of the following control signals $x_1, x_2, ..., x_7$ are assigned to each device that may control the bus. For example the memory would require x_7 to be asserted in order to carry out a read operation. The logic equation for this would be: $x_7 = \bar{R}T_1 + \bar{D}_7IT_3 + (D_0 + D_1 + D_2 + D_6)T_4$. The others are similarly derived. The **ALU** is designed by embedding Full Adders with logic between the **DR** and the **AC**, as shown in Fig. 7.

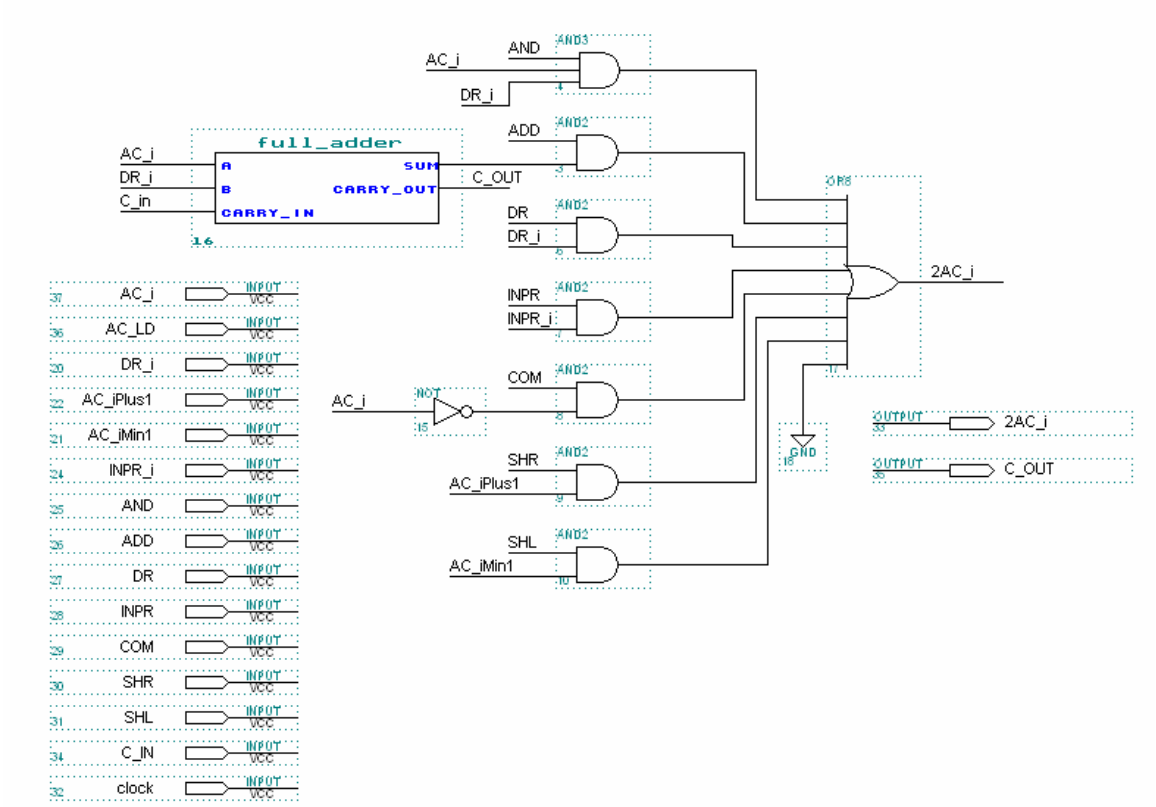

Fig. 7 AC Adder and logic, 1-bit.

As the design was progressing higher hierarchies were developed. For example, Fig. 8 depicts the control logic for the **PC** Register. This logic was then modularized into a symbol, Fig. 9. This module was then placed with all the other control modules associated with the other registers and timing signals, Fig. 10, and modularized again into a master Control Module as shown in Fig. 11. This hierarchal approach continues into Fig. 12 to include the Control Module, Register Module, memory unit, S flip-flop, and the bus multiplexer, and then finalized in Fig. 13, the top level module representing a complete **CPU**.

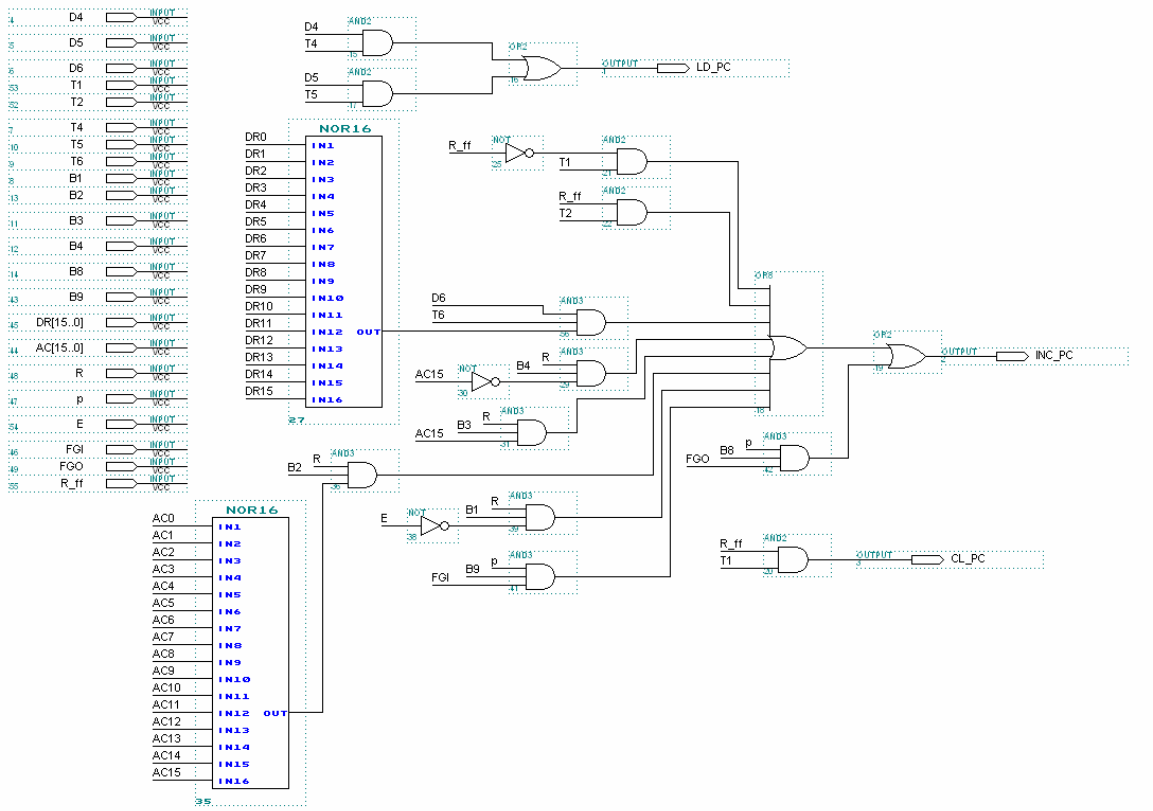

Fig. 8 Control logic for the PC Register.

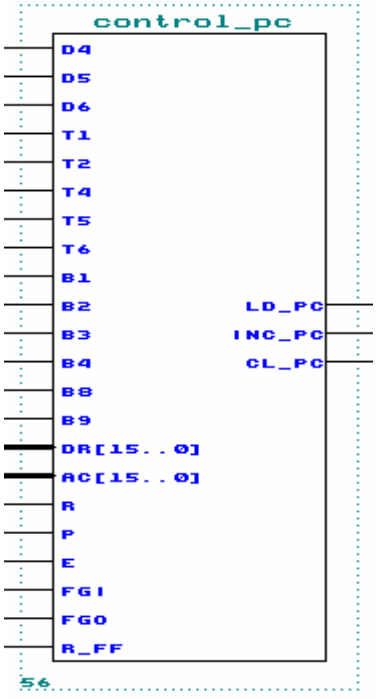

Fig. 9 Control Module for the PC Register.

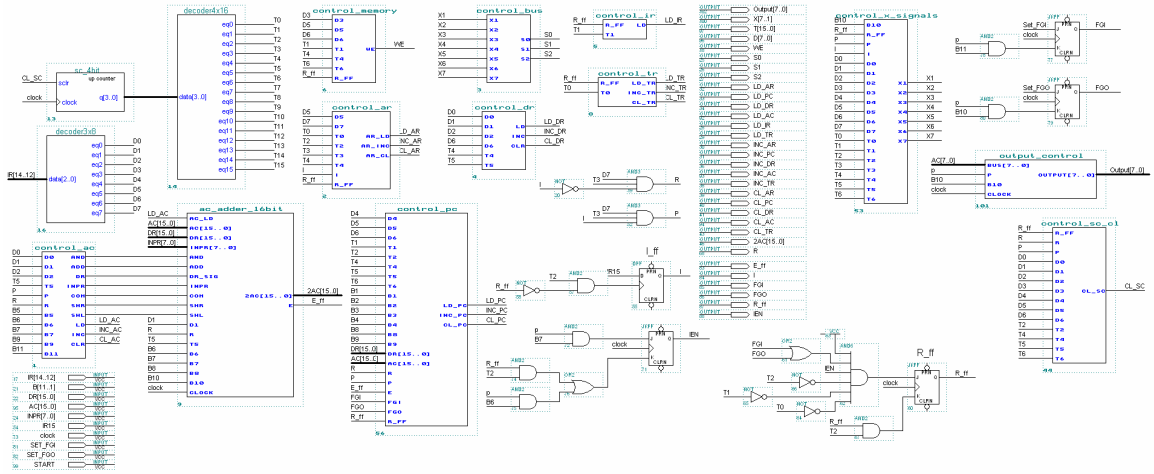

Fig. 10 Flip-flops, timing signals, and a module for each register's control signals.

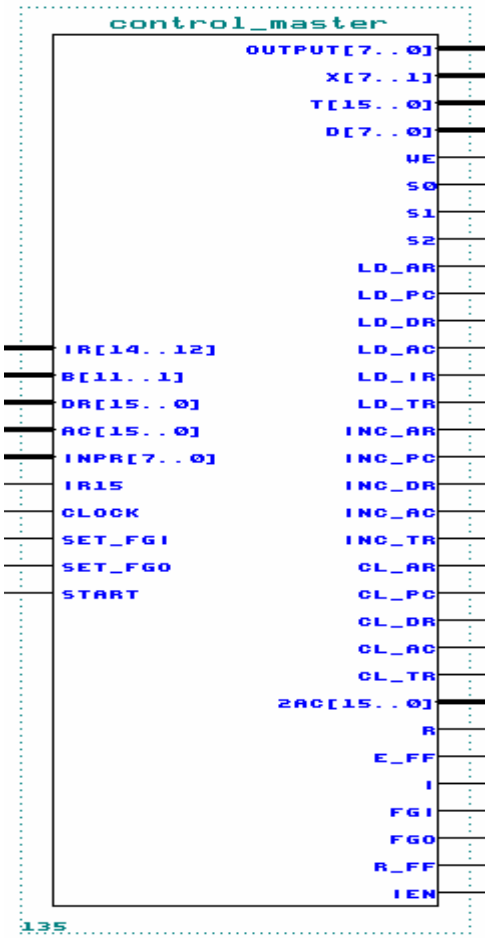

Fig. 11 Control Module.

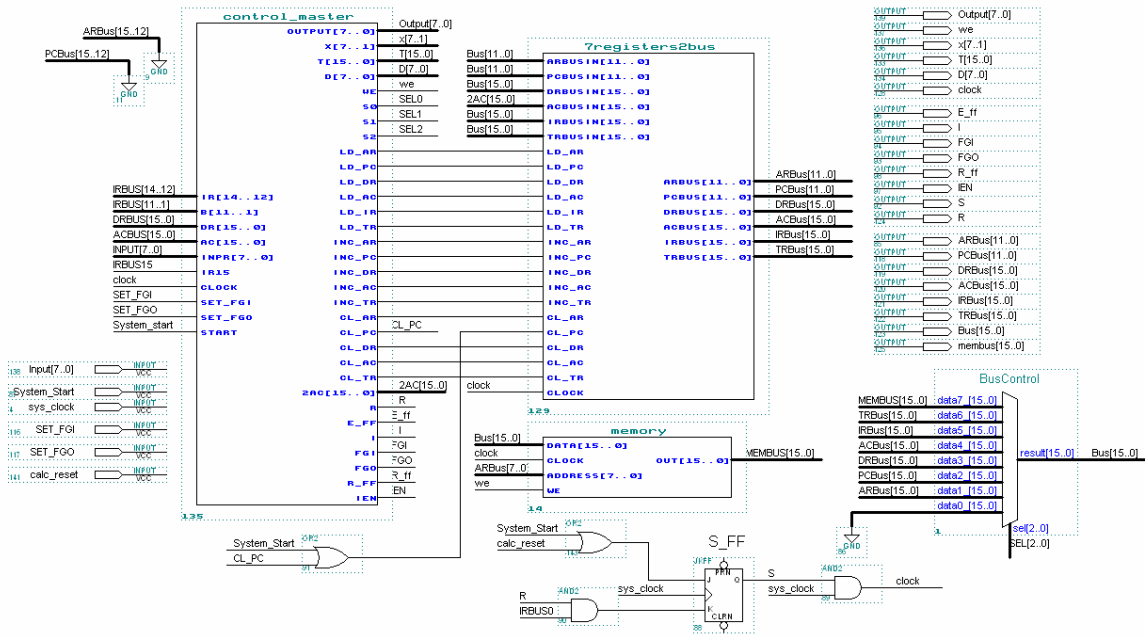

Fig. 12 Control Module, Register Module, memory unit, S flip-flop, and the bus multiplexer.

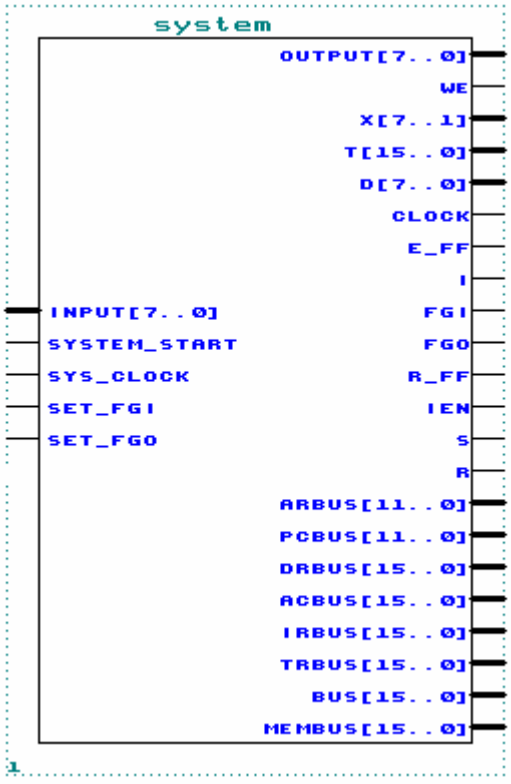

Fig. 13 Complete **CPU** Module.

The design was tested in two ways. First, it was simulated, Fig. 14, using MAX+Plus II[9] software with a small program listed in Table 3. After a successful simulation, the design was downloaded onto the Altera UP2 board, Fig. 15, and tested with the same program listed in Table 3. The LED's and 7-segment displays on the UP2 were utilized to observe the testing.

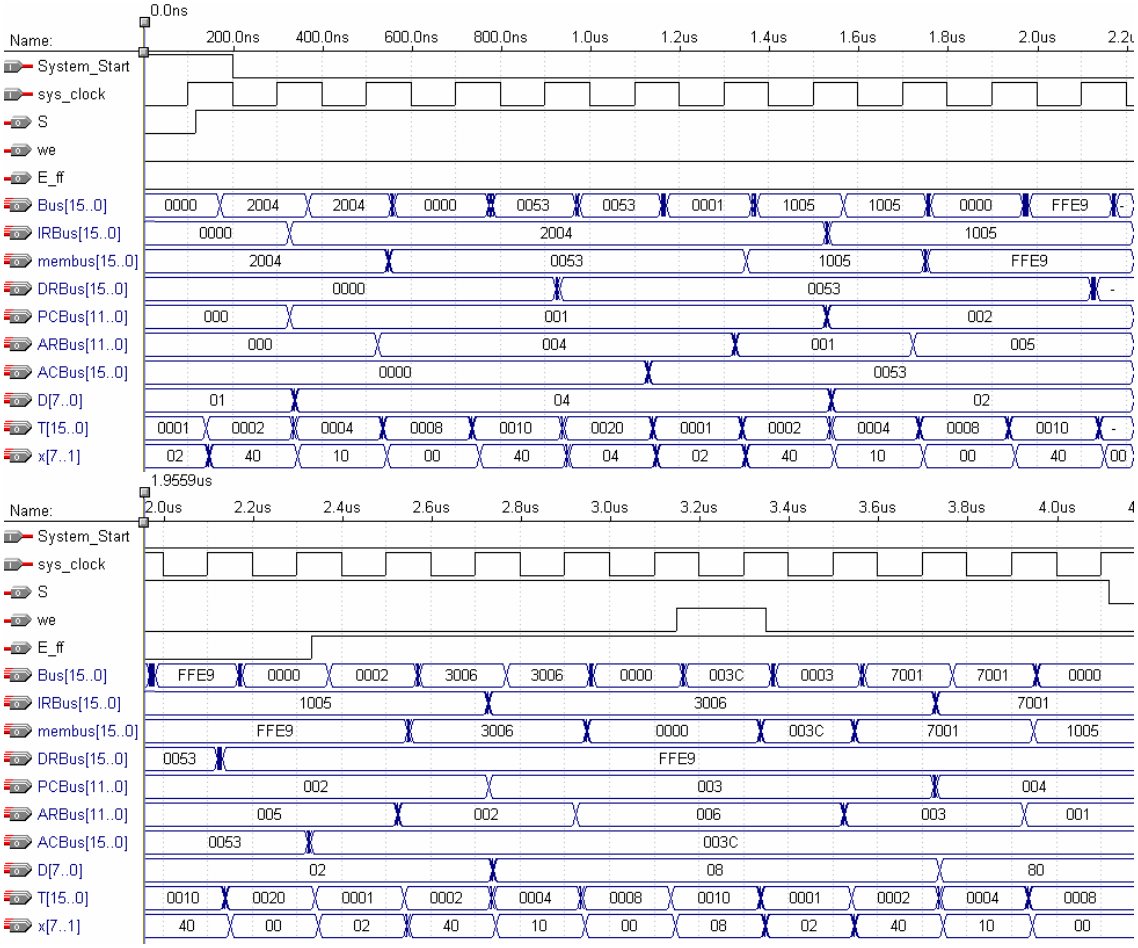

Fig. 14 Simulation of program in Table 3.

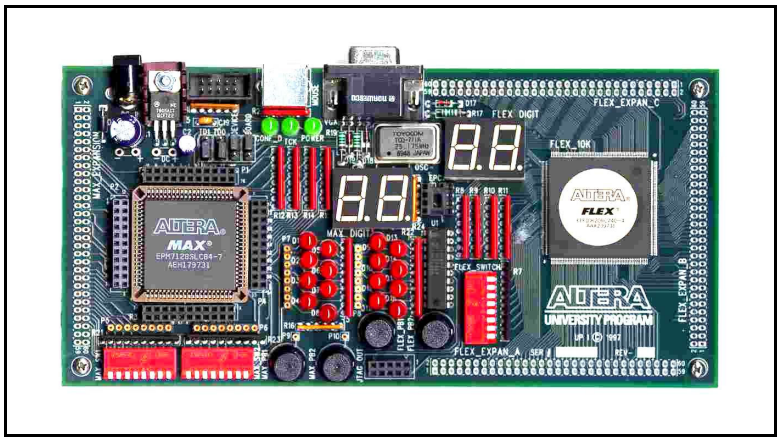

Fig. 15 Altera UP2 board

As depicted in Fig. 14, the program starts when System_Start is pulsed, which sets the **S** flip-flop. The **PC** Register is transferred to the **AR** Register when $T_0 = 1$. When $T_1 = 1$ the instruction in memory pointed to by **AR** is loaded into the **IR** Register to be decoded, and the **PC** Register was incremented to 001h in preparation for the next time $T_0 = 1$. The **PC** Register started at 000h, when loaded into **AR** pointed to the first location in memory. At that location is the opcode 2004h; load **AC** with the operand, 0053h, located at address 004h. The next instruction adds a second operand from memory at location 005h to the first operand already in **AC**. With the next instruction the result, 003Ch, is then stored in memory at location 006h. The write signal *we* can be clearly seen while **IR** contains 3006h, **AR** point to 006h, and AC contains the sum 003Ch. The last instruction is **HALT**, 7001h, which clears the **S** flip-flop and halts the **CPU**.

Table 3 Assembly Language program to add two numbers

Location	Label	Instruction	Opcode	Comments
		Org 0		/origin of program is location 0
000		LDA A	2004	/Load operand from location A
001		ADD B	1005	/Add operand from location B
002		STA C	3006	/Store sum in location C
003		HLT	7001	/Halt computer
004	A,	DEC 83	0053	/Decimal operand
005	B,	DEC -23	FFE9	/ Decimal operand
006	C,	DEC 0	0000	/Sum stored in location C
		END		/End of symbolic program

Conclusions

We have presented a new 5-step methodology for hardware **CPU** design. The computer was first defined with internal registers, bus and memory size. The instruction set was selected with minimum number of instructions and simple instruction addressing formats but complies with instruction set completeness requirements. In order to move to the lower microprogrammable level, where signals can be generated, a micro instruction table was created. The table detailed all the actions taken for each instruction and for the whole instruction set. This was possible through utilization of the Register Transfer Language constructs. This language was used since it transforms into hardware details the register operations inside the **CPU** that is essentially sequential in nature. The table also explains the usual three phases encountered by each instruction. The fourth step in the methodology was reduced to a simple and repetitive process of scanning the micro instruction table for each register and instruction actions. A general model for the registers was designed such that all the possible operations were confined to the assertion/de-assertion of three control signals, the load signal **LD**, the increment signal

INC and the clear signal **CLR**. Examples of logic equations formation were illustrated. The control unit was formed from the aggregate of all the logic equations for all the registers and their control signals. The complexity of the hardware was further reduced to a hierarchical design using the industry standard Altera MAX+Plus II[9] software design environment. The design was downloaded to an FPGA/CPLD platform. The platform was a prototyping board UP2 developed by Altera Inc. The design was tested through simulation runs of a representative program. The **CPU** hardware was also tested successfully for the same program on the UP2 board with 7-segment displays and LEDs representing inputs and outputs. The entire design utilized about 13% of the available Logic Cells (**LC**) count that amount to about 70,000 gates. The rest of the **LC**s were available to use to enhance the design with more logic designs or peripherals without having to add additional external logic. The design can be considered a very successful embedded microprocessor soft core design or System on a Chip (**SoC**). Of course much denser FPGAs are now available where more powerful soft microprocessors can be implemented. The 5-step methodology introduced can be used for more complex designs. The main emphasis again is on the creation of the micro instruction table. Hence, there are no restrictions as to the size or number of registers used, or the size of the memory.

References:

1. Andrew S. Tanenbaum, "Structured Computer Organization", Prentice Hall, 1999, ISBN 0-13-095990-1
2. William Stallings, "Computer Organization and Architecture", Prentice Hall, 2000, ISBN 0-13-081294-3
3. IRV Englander, "The Architecture of Computer Hardware and Systems Software", Wiley, 2003, ISBN 0-471-07325-3
4. Sajjan Shiva, "Computer Design and Architecture", Marcel Dekker, 2000, ISBN 0-8247-0368-5
5. Alen Clements, "The Principles of Computer Hardware", Oxford, 2000, ISBN 0-19-856454-6.
6. M. Morris Mano, "Computer System Architecture"; Prentice Hall, 1993; ISBN 0-13-175563-3.
7. James O. Hamblen and Michael Furman; "Rapid Prototyping of Digital Systems'; Kluwer Academic Publishers; 2001; ISBN 0-7923-7439-8.
8. Rex N. Fisher, "Design and Implementation of a Simple 8-Bit CPU", 2000. http://www.rexfisher.com/P8/P8_TOC.htm
9. MAX+Plus II Programmable Logic Development System and Software. http://www.altera.com/literature/ds/dsmii.pdf
10. V. Korneev, and K. Kiselev, "Modern Microprocessors", Charles River Media, 2004, ISBN 1-58450-368-8.
11. Altera University Program UP2 Education Kit. http://www.altera.com/literature/univ/upds.pdf.

K. SALMAN is a Professor in the department of ETIS at Middle Tennessee State University. He holds Ph.D. in electrical engineering from Brunel University, UK, and a member of IEEE.. His main line of research is in embedded microprocessors and ciphering.

MICHAEL B. ANDERTON is a Graduate Assistant at Middle Tennessee State University where he received his B.S. in Engineering Technology. His undergraduate senior project involved work with the Residue Number System. His graduate research interest includes System-on-chip (SoC) technology.