A Microprogrammable Processor Teaching Tool and its FPGA Implementation

Xavier Maldague,[1] Hélène Torresan

Electrical and Computing Engineering Department

Université Laval, Quebec City (Quebec), Canada G1K 7P4

Abstract - In this paper a microprogrammable processor teaching tool and its FPGA implementation are presented. The processor is based on a classical Am2900 architecture and it is implemented in the popular Xilinx Foundation™ Environment. The tool helps electrical and computing engineering students to familiarize practically with microprogramming through development of their own instruction set and experiments with signal flow within a processor. In addition to simulation, the complete configuration can be run on an Xilinx 4010XL FPGA. In the paper a short description of the tool is provided.

Keywords: microprogramming teaching tool, Am2900, FPGA implementation, microprogrammable processor, ALU, CCU, firmware

1. Introduction

A quick survey on the Internet reveals that microprogramming is taught in many universities including our own. In fact, microprogramming is generally included in *Digital circuit, Computer Architecture* or *Microprogramming* courses in electrical and compuer engineering programs and also in computer science [1, 2, 3]. Sometimes (as in the Université Laval), it is combined with a study of micro controllers to form a complete one-term course.

Nowadays, there are two broad classes of processor architectures, *hardwired* or *microprogrammed*. Both have advantages and drawbacks. As it is known, micropro-

1. maldagx@gel.ulaval.ca

gramming is the programming, at the lowest level, of the hardware architecture defining a given processor (microprogramming is also known as *firmware*). Once the architecture is defined, microprogramming enables flexibility and reverse compatibility since it is "just a matter" of reprogramming the given architecture to update the chip (by changing the firmware). On the other hand, decoding associated with microprogramming degrades the overall processor performance.The well known Intel 8051 (and later 80251) micro-controller is an example of a microprogrammed processor with compatibility with the former Intel 8048.

In *hardwired* architectures, subsystems are based on decoding logic and complex sequential and combinational networks. Advantages of this approach are: custom design, smaller (minimal) size, higher speed. It is generally considered that *hardwired* architectures are good for fixed (stable) designs and high production intended volumes. They materialized, for instance, as Reduced Instruction Set Computers (RISC) which are, of course, less flexible but faster.

Back to the classroom, the concerns in teaching microprogramming concepts are especially related to the practical aspects. Since for a given chip microprogramming is "buried in the silicon" by the processor manufacturer, it is not accessible. However, the current development of the embedded logic such as the availability of large Field Programmable Gate Arrays (FPGA) makes it possible to configure these circuits as microprogrammed processors. This approach was selected for the present project.

The selected microprogrammed architecture is the classical Am2900 described in [4, 5]. The Am2900 Family of products does not exist anymore, however, it is still "alive" as emulators, FPGA and code since as pointed out by D.E. White [5], "it was just too cool to lose as far as the design community is concerned”. Readers are referred to these references for more details on the discussed microprogrammed architecture since it is out of the scope of this paper to detail various Am2900 blocks here. The proposed teaching tool was designed and implemented in an FPGA (Field Programmable Gate Array) allowing students to:

- define instruction sets,
- experiment with a microprogrammable processor and also
- generate real signal activities at the FPGA outputs [6,7, 8].

In this text, the architecture of the microprogrammable processor teaching tool and its FPGA implementation are reviewed. One goal of this paper is to present the tool which is available for free (contact the authors) and is considered as an interesting addition to electrical, computier engineering and computer science curricula when microprogramming concepts are taught. Finally, it is worthwhile to mention the tool was developed in the popular Xilinx Foundation™ Environment commonly found in many universities (more on that in section 3). This is considered as a standard approach when compared with the TTL discrete components approach. Surprisingly, such TTL discrete components approaches are still sometimes seen in teaching microprogramming!

2. Processor Architecture

In a microprogrammed processor, the main (=macro) program consisting of several macro-instructions is translated into hardware activity through micro-instructions. Thus, each macro-instruction corresponds to a sequence of micro-instructions (see for example, Figure 1). All these micro-sequences constitute the instruction set to which the microprogrammable processor responds.

The microprogrammable processor used in this project is classical and based on a pipelined CPU (Central Processing Unit) with scratch pad registers [1-4]. Figure 2 (a) presents the general design: the main parts are the Computer Control Unit (CCU), the Arithmetic Logic Unit (ALU) with associated scratch pad registers, the Program Counter (PC), the Memory Address Register (MAR) and the bus to move around signals. A display unit is added to show FPGA outputs on LEDs (see section 4, below). In such a microprogrammed architecture, all the activities are managed by the CCU. The CCU stores the macro-instruction from the macro-program located in the macro-memory of the Instruction Register and then decodes and translates it into a succession of micro-instruction sequences located in the micro-memory (Figure 1). A micro-instruction is made up of various fields, each one addressing a particular structure within the processor. In fact the micro-instruction fields contain all the bits needed to control these structures. Since a pipeline is used here, both the micro-memory and the ALU being activated simultaneously, thus reducing the micro-cycle length. The pilpeline is a register that freezes the signals so that the ALU starts activity on the stable data while simultaneously the micro-memory fetches the next micro-instruction. The micro-cycle corresponds to the period of time needed to execute one micro-instruction.

Figure 2 and Table 1 show the processor architecture in greater detail. The **BUS** serves both for address and data transfers. It is three-state protected to avoid conflicts since multiple units are connected to it as shown. The **Memory-Address Register (MAR)** is the 8-bit register containing the address pointing out to the macro-memory, the address coming from the Program Counter (PC). The **Memory Buffer Register (MBR)** holds the 8-bit data read or to be written to the macro-memory at the address contained in the MAR. The **Macro-Memory** is 8-bit wide and 32-byte deep. This is sufficient for our teaching purpose. Restrictions on memory size is necessary to fit the design within the 4010XL FPGA (see section 3). In the present system only the 5 least significant bits of the MAR are used due to the macro-memory size ($2^5 = 32$), the other bits are ignored. Readings are made without delay while writings are executed at the clock falling edge since the address in the MAR is only available after a clock rising edge. The **Instruction Register (IR)** is a 16-bit register holding an 8-bit opcode and an 8-bit operand. For a typical macro-instruction, the PC points to the opcode while the operand is located at the PC+1 address. The 8-bit operand is divided into two 4-bit nibbles corresponding to the two scratch pad register addresses possibly used by the instruction ($2^4 = 16$ registers are available), Figure 3. To ease manipulation the PC is directly implemented as one of the scratch pad registers (PC = Register # 0). Subroutine FETCHINSTR (Figure 1 and Figure 4, below) loads the IR and updates the PC so that it points to the next macro-instruction (<PC>+2 -> <PC>).

Block **2910** is the CCU which is based on the Am2910 Supersequencer from AMD [4,5], Figure 2 (b). The 2910 recognizes sixteen instructions[1]. Essentially the purpose of the 2910 is to generate an address to the microprogram memory, based on the current macro-instruction. It comprises the following structures:

- Address Selection Multiplexer (MUX) with four sources of address,
- Incrementer (adds “1” asynchronously to the address available at the MUX output),
- micro-PC register (loads the incrementer output at the rising clock edge),

1. e.g. *Continue*, *Jump to subroutine, Three Way Branch*, etc. [5].

- Stack and Top-of-Stack pointer (save the return address for loop and jump to subroutine instructions),
- Counter/Zero-test (for loops a constant is loaded there from the Pipeline and decremented at each loop down to zero, Zero-test module detects this condition to end looping),
- Register (storage to hold a value from the Pipeline).

Block **2901** is an asynchronous 8-bit 8-function ALU based on the 2901 ALU from AMD [4,5], Figure 2 (c). Computations take place on the rising clock edge (when the Pipeline settles) and results are saved in a register at the following falling clock edge. It is made up of the following structures:

- RAM shift (to shift one bit to left or right direction, useful for multiply and divide-by-two operations),
- Q Shift (similar to the RAM shift, enables dual-precision arithmetic in conjunction with the RAM shift),
- Q-Register (an ALU register at the output of the Q-shift unit),
- Scratch pad registers (register selection comes either from the macro-instruction operand - Figure 3 - or is imposed directly by the micro-instruction),
- ALU data source selector (selects the source of operands either from a scratch pad register or BUS or other ALU units),
- 8-function ALU (operations: +, -, swapped -[1], AND, inverted AND, OR, Exclusive Or, Identity; results can be fed back to a scratch pad register and status bit - Zero, Sign, Overflow, Carry out - can be transferred to the Micro Macro flag register),

1. implementation of “a - b” and “b - a.”

- Output Data Selector (transfers either the ALU output or a register content to the BUS),
- ALU logic (controls the ALU).

Block **2904** is based on the Am2904 from AMD [5] and serves to determine the condition value during a branch instruction. It comprises the following structures:

- Micro Macro flag register (memorizes ALU status bits and makes them available to take a decision based on their values; macro-flags are available at the macro-instruction level and micro-flags are available at the micro-instruction level),
- Conditional MUX (to select the condition of interest, for instance based on the Zero flag value), Polarity block (enables the NOT operation by inverting the condition value).

3. FPGA implementation

The microprogrammable processor described in the previous section is implemented in the Xilinx Foundation™ - software - environment used in many *Digital Circuit* classes including ours [9]. The three input methods available in Xilinx Foundation™ are used in this project. In the design, we mostly relied on the schematic editor for easy access to signals (through labels) and direct visualization of the overall architecture by the students. Also, HDL[1] editor and especially ABEL™[2] was employed since students becomes familiar with ABEL™ faster as compared to VHDL[3]. Typically, we offer students a three-hour crash course in ABEL™ and this has been proved to be sufficient for them

1. Hardware Description Language.
2. Advanced Boolean Expression Language.
3. VHSIC Description Language, VHSIC stands for Very High Speed Integrated Circuits.

to feel comfortable with this language. With respect to ABEL™, VHDL is more complex and requires a full term course to grasp. Moreover, ABEL™ sources can be encapsulated within a block called “macros” included into the overall schematic design. Finally, Logiblox modules written in VHDL were also written to implement MUX and memories.

In order to run the processor, the first step consists to update three files implementing the:

- micro-memory,
- macro-memory, and
- mapping prom.

Most of the student work deals with writing the micro-memory file defining the macro-instructions. In fact, since example files are already provided to students, they only need to update these after being familiar with the content (see examples provided in Figures 1, 4, 5, 6, 7).

As said before, **micro-instructions** must provide the control bits for all structures present in the processor. Table 2 shows how this is organized with the different fields for all the discussed structures. Students edit ABEL™ files according to these fields which implement the 70 control bits of the processor. Student work concerns essentially the development of new opcodes and thus elaboration of micro-instruction sequences implementing these. Such a study of microprogramming concepts is obviously the purpose of the project.

ABEL™ template files are made available to students so that no extensive knowledge of this language is needed. For instance, the following ABEL™ structure is employed to define micro-instructions:

when (M == ^hXX) THEN { ... }

where XX is the address of micro-instruction within the micro-memory (the micro-memory has a capacity of 256 micro-instructions) and { ... } corresponds to the micro-instruction body (there is one micro-instruction per micro-memory address). As an example, Figure 4 shows the ABEL™ code for the FETCHINSTR subroutine. FETCHINSTR uses micro-address 5 to 8. It is thus a 4 micro-instruction long subroutine. A few other micro-sequences are provided to students to help them with the system, for instance *ADD R1, R2* (to add two registers together, Figure 1), *LOAD R imm* (load register with an immediate data, Figure 5), etc.

The **macro-program** makes use of the available opcodes defined in the micro-memory to perform a given task. The minimum macro-instruction size is 16-bit long coded on two subsequent macro-memory addresses (since the macro-memory is 8 bit). As said before, the first byte defines the opcode and the second the operands (Figure 3). Moreover, only the last 5 bits of the opcode are used so that a maximum of $2^5 = 32$ opcodes can be defined. Although the minimum macro-instruction length is two bytes, there is no maximum. This minimum figure comes from the fact the FETCHINSTR subroutine increments de facto the PC by two since “normally” an instruction is made up of both an opcode and operands such as *ADD R1, R2. O*f course FETCHINSTR could be redefined differently. For longer macro-instructions (i.e. instructions of more than two bytes), it is the user’s responsibility to increment the PC as required. For instance macro-instruction *LOAD R imm* is defined in three bytes (Figure 5): opcode (one byte), operand (one byte, one nibble in fact since only one register is involved here) and data (one byte). Figure 6 lists a macro-program example: macro-programs are created as VHDL Logiblox and templates are available to help students to write these.

Finally, the **Mapping Prom** file performs the correspondence between a given defined opcode and the address of the first corresponding micro-instruction within the micro-memory. This LUT (look-up table) takes the 5 least significant bits of the macro-instruction opcode and translates them to an 8-bit long address. Here too an ABEL™ structure is employed to define the Mapping Prom LUT. For instance:

when (A == ^h01) THEN S = ^h19;

makes the correspondence between opcode number 1 and micro-sequence starting at micro-address 19 (“A” stands for opcode and “S” for address). A Mapping Prom file example is listed in Figure 7.

4. Simulation and execution

Once files defining micro-program, macro-program and Mapping Prom are ready, the processor can be entirely simulated with the simulator included in the Xilinx Foundation™ Environment [10]. Practice with this simulator was found particularly important to help students:

- understand what is going on and why,
- check it obtained values conform to what is expected,
- verify signal values.

The uinteresting thingh is that students can follow the progression of the signals directly on the screen since it matches the architecture layout of Figure 2 (thanks to the schematic editor available in Xilinx Foundation™). This is particularly useful at the debugging stage. Moreover, double-clicks on blocks enlarge them allowing to view their detailed content descriptions (example: listing of ABEL™ file, VHDL Logiblox). Timing diagrams of simulations can also be saved and printed for inclusion within a lab report.

Once the simulation is satisfying, it is possible to download the design in the 4010XL FPGA. Here again, the Xilinx Foundation™ Environment is used at this stage by providing the '.bit' configuration file to be downloaded onto the FPGA (with a utility program such as *gxsload* available from Xess Corporation [11]). The 4010XL FPGA is mounted on a XS40-010XL board available from Xess Corporation and connected to a Windows computer through the parallel port [11]. Eight inputs (example dip switches) and eight outputs (example LEDs) are connected to the 4010XL FPGA, see **Display** structure in Figure 2 (a). Two of the inputs are reserved (so that six are available for selection): one serves for the reset and the other for the clock. The clock can be connected to a debounced switch for manual step by step testing or to a square wave signal for automatic progression (e.g. 1 Hz). Out of the eight input bits, the six bit remaining input bits display of different values on the output LEDS. For instance the content of any scratch pad registers, the content of a macro-memory location or the output of the 2910 MUX (section 2). This last possibility provides a way to check which micro-instruction is being executed. After five clock rising edges at start-up, a reset is completed (thanks to the RESET micro-routine located at micro-memory address “0”) and the processor is ready to start its task. Execution next proceeds at each rising edge of the clock.

Of course, with only 6 inputs and 8 outputs, not much “useful” can be done with this implementation, but this is not our purpose! In fact, for the purpose of teaching microprogramming concepts, simulation in the Xilinx Foundation™ Environment is sufficient. However, the possibility to configure the FPGA and observe real hardware activity can be seen as the “ultimate microprogramming challenge”. Knowing this is available in the tool is thus attractive and makes it “complete,” since the hardware can be exercised by executing “real software” (as defined by the macro-program), although on a small scale.

5. In class deployment

Since the system is based on the same FPGA platform (the XS40-010XL board) and software tools (the Xilinx Foundation™ Environment) that are employed by our *Digital Design* course, substantial savings are made. This saves us also the requirement to define and develop our own support environment. Moreover, since students in the *Micro-programming* course followed the *Digital Design* course previously, they are already familiar with both the FPGA platform and the Xilinx Foundation™ environment. Thus they are able to go deeper into microprogramming concepts.

The system was tested repeatedly within our 75-student *microprogramming* class through both magistral demonstrations and home work (students are grouped in team of two). They were asked to do several exercises: get acquainted with the system, develop several new macro-instructions. Globally, students have found their experience with the system quite positive.

It is worthwhile to say that availability of the supporting hardware (the XS40-010XL board) was found to be useful although, as said above, not essential since the simulation performed in the Xilinx Foundation™ environment already provides an exhaustive testing of the mircroprogrammable design.

6. Conclusion

In this paper, a microprogrammable processor and its FPGA implementation were described. This system was developed as a teaching tool for microprogramming in our electrical and computer engineering curricula. Since the supporting hardware/software combination is common in many universities, it is believed the presented microprogramming teaching tool will be well received especially since it is available for free.

7. Acknowledgements

Supports of the Electrical and Computing engineering department of Université Laval and Programme Études-Travail of the Education Department of Quebec Province are acknowledged.

References

[1] C. M. Gilmore, *Microprocessors: Principles and Applications*, 2/e, McGraw-Hill Pub., 544 pp, 1996.

[2] I. Englander, *The Architecture of Computer Hardware and Systems Software: An Information Technology Approach*, 2/e, J. Wiley Pub., 784 pp, 2000.

[3] M. M. Mano, *Digital Design*, 3/E , Prentice Hall, 516 pp, 2002.

[4] D. E. White, *Bit-Slice Design: Controllers and ALU*s, 1996 with last edit July 7, 2001[1].

[5] *Am2900 Family*, Advanced Micro Devices, Inc. (http://www.amd.com/us-en/).

[6] E. Hwang, “Where’s the hardware?”, Circuit Cellar, issue **150**, p. 32-38, 2003.

[7] H. de Garis, M. Korkin, F. Gers, E. Nawa and M. Hough, “Building an artificial brain using an FPGA based CAM-Brain Machine”, *Applied Mathematics and Computation*, **111**[2-3]: 163-192, 2000.

[8] V. Sklyarov, “Reconfigurable models of finite state machines and their implementation in FPGAs”, *Journal of Systems Architecture*, **47**[14-15]: 1043-1064, Aug. 2002.

[9] R. S. Sandige, *Digital Design Essentials*, Prentice Hall Pub., 670 pp, 2002[2].

[10] See *www.xilinx.com/* (and XUP - Xilinx University Program) and online help in *Xilinx Foundation*™.

[11] available for download from *www.xess.com.*

1. This book is available on line at: `www10.dacafe.com/book/parse_book.php?article=BITSLICE/index.html`
2. Interrestingly, *Xilinx Foundation*™ *- student edition* software is bonded with this book.

Figure Captions

Figure 1	A simple macro-instruction and its equivalent sequence of five micro-instructions. Note < ..> denotes the content of “..”. PC, MAR, IR shown in Figure 2. Micro-code listing of the ADD R1 R2 micro-sequence is also provided (refer to Figure 4 for more explanations).
Figure 2	(a) Overall architecture of the Microprogrammable Processor Teaching Tool. MSB = Most Significant Byte, LSB = Least Significant Byte, OE = Output Enable. Snapshot of the Xilinx Foundation™ Environment as seen by the students and showing the overall architecture of the Microprogrammable Processor Teaching Tool. (b) Overall Am2910 CCU structure. (c) Overall Am2901 ALU structure [4,5].
Figure 3	Format of a 2-byte macro-instruction located at macro-memory address “i.” OPCOD designates the 5 bit opcode, AAAA and BBBB the scratch pad register addresses and xxx are unused bits. The macro-memory is eight bit wide as shown.
Figure 4	Micro-code listing of the FETCHINSTR micro-sequence with 4 micro-instructions (micro-memory addresses 5 to 8). Refer to Table 2 for Micro-instruction fields and subfields. Notes: ^h for hexadecimal, ^b for binary. See also Figure 1 for description of FETCHINSTR micro-sequence.
Figure 5	Micro-code listing of the *LOAD R imm* micro-sequence with 2 micro-instructions (micro-memory addresses ^h19 to ^h1A).

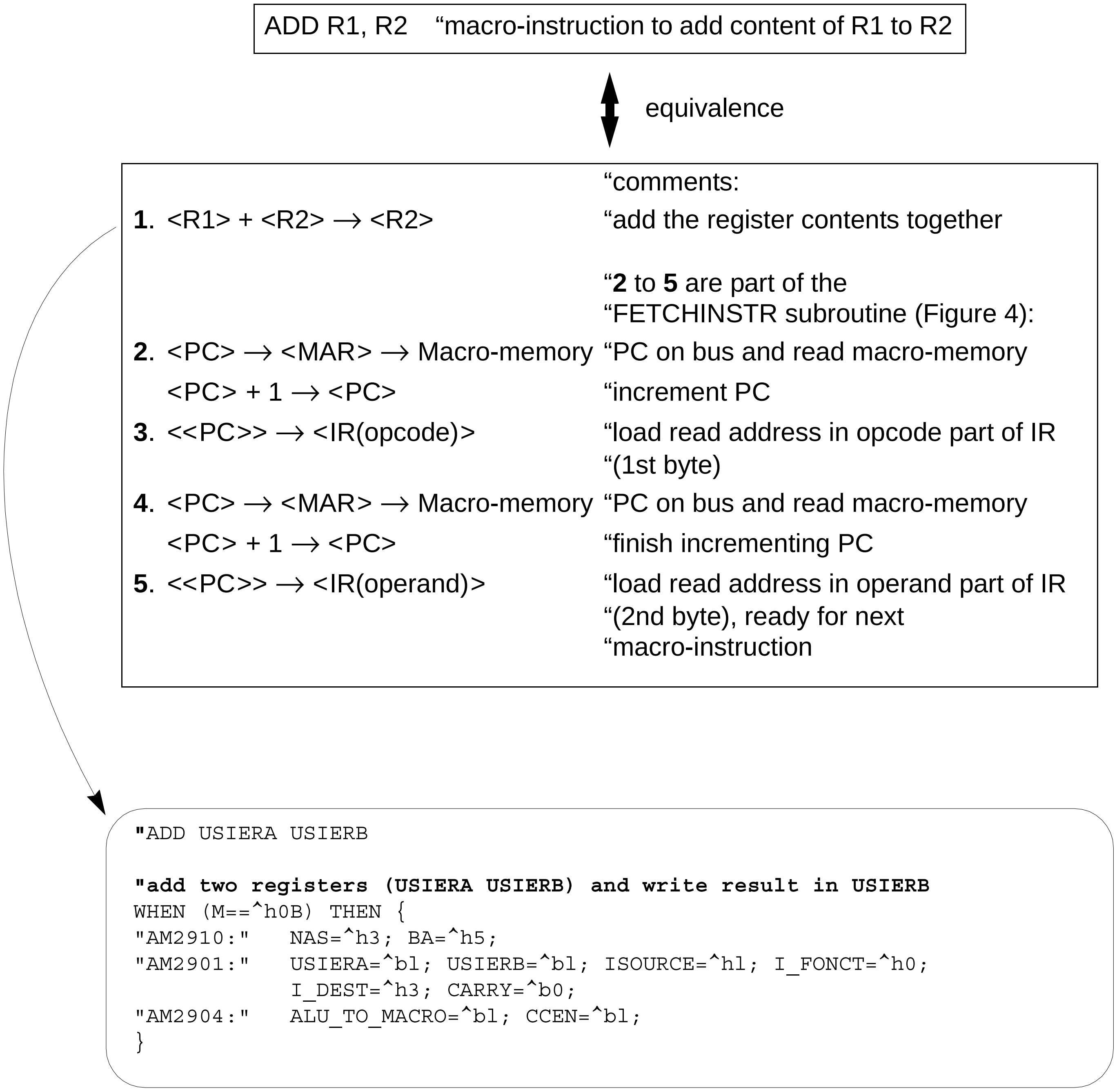
Figure 6	Example of a macro-instruction listing, two 3-byte macro-instructions and one 2-byte macro-instruction. Macro-memory addresses 0 to 7.
Figure 7	Example of an ABEL™ Mapping Prom listing showing defined opcodes 01 and 02 for macro-instructions *Load R IMM* and *ADD USEIRA USEIRB* respectively. *USEIRx* means operand comes from the macro-instruction (*x* = *A* or *B*).

Table 1 - Structures of the microprogrammable processor

Main structures	substructures
BUS	Bus
Macro-memory	Macro-memory
MAR	Memory Address Register
MBR	Memory Buffer Register
IR	Instruction Register
2910 (CCU)	CCU Logic, Counter, Zero-Test, Register, Stack, TOS Pointer, Incrementer, micro-PC Register, MUX
2901 (ALU)	Scratch pad registers, ALU Data Source Selector, 8-Function ALU, Output Data Selector, ALU Logic, Q Register, Q Shift and Ram Shift
2904	Micro Macro Flags, Conditional MUX, Polarity
Micro-memory	Micro-program Memory
Pipeline	Pipeline
Mapping Prom	Look-up table

Table 2 - Micro-instruction fields

Micro-instruction fields	Micro-instruction subfields
A. AM2910:	Next_Address_Select (NAS), Branch_Address (BA)
B. ALUCONST:	Constant made available as operand for ALU
C. MEMORY:	Read_Write
D. BUS:	Onto, Offof
E. AM2901:	A_Address, B_Address, Alu_Source, Alu_Fonction, Alu_Destination,Carry
F. AM2904:	Branch_Condition_Select, Polarity, Force_Condition, Flag_Source
G. RAM-shift:	Ram_In
H. Q-shift:	Q_In


ADD R1, R2 “macro-instruction to add content of R1 to R2

	“comments:
1. <R1> + <R2> $\longrightarrow$ <R2>	“add the register contents together
	“**2** to **5** are part of the “FETCHINSTR subroutine (Figure 4):
2. <PC> $\longrightarrow$ <MAR> $\longrightarrow$ Macro-memory	“PC on bus and read macro-memory
<PC> + 1 $\longrightarrow$ <PC>	“increment PC
3. <<PC>> $\longrightarrow$ <IR(opcode)>	“load read address in opcode part of IR “(1st byte)
4. <PC> $\longrightarrow$ <MAR> $\longrightarrow$ Macro-memory	“PC on bus and read macro-memory
<PC> + 1 $\longrightarrow$ <PC>	“finish incrementing PC
5. <<PC>> $\longrightarrow$ <IR(operand)>	“load read address in operand part of IR “(2nd byte), ready for next “macro-instruction

```
"ADD USIERA USIERB

"add two registers (USIERA USIERB) and write result in USIERB
WHEN (M==^h0B) THEN {
"AM2910:"   NAS=^h3; BA=^h5;
"AM2901:"   USIERA=^bl; USIERB=^bl; ISOURCE=^hl; I_FONCT=^h0;
            I_DEST=^h3; CARRY=^b0;
"AM2904:"   ALU_TO_MACRO=^bl; CCEN=^bl;
}
```

Figure 1 A simple macro-instruction and its equivalent sequence of five micro-instructions. Note < ..> denotes the content of “..”. PC, MAR, IR shown in Figure 2. Micro-code listing of the ADD R1 R2 micro-sequence is also provided (refer to Figure 4 for more explanations).

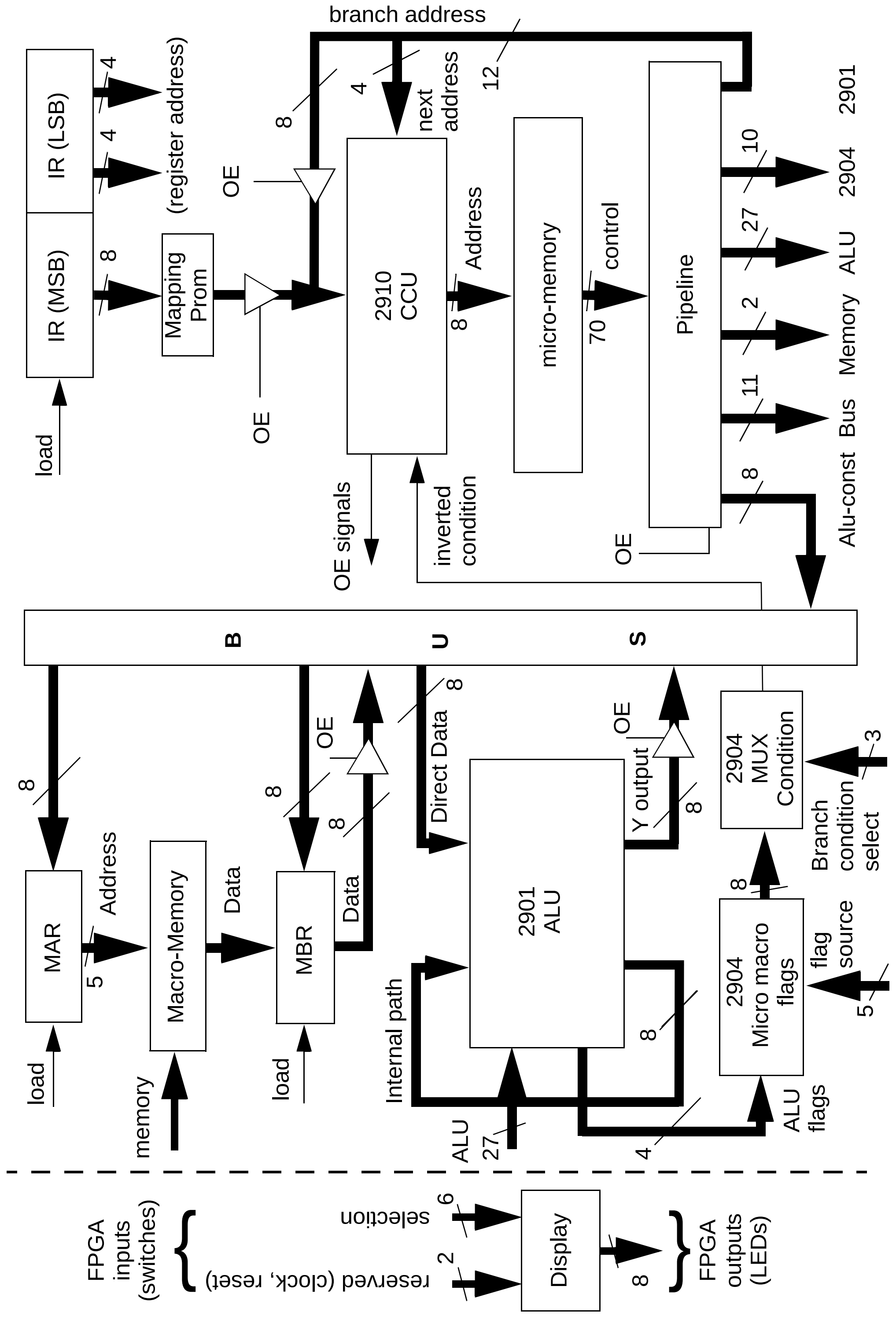


Figure 2 (a) Overall architecture of the Microprogrammable Processor Teaching Tool. MSB = Most Significant Byte, LSB = Least Significant Byte, OE = Output Enable.

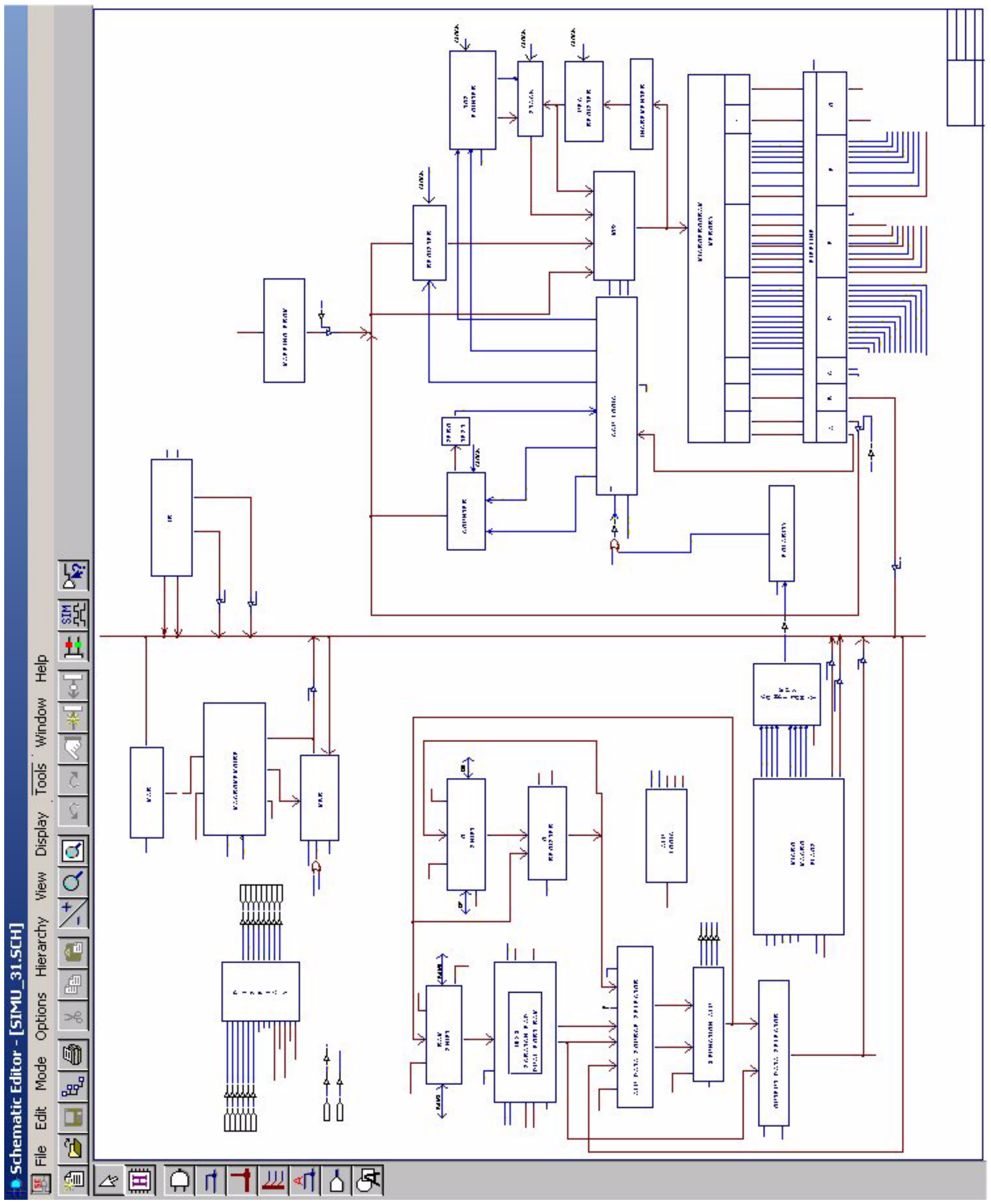


Figure 2 (a) Snapshot of the Xilinx Foundation™ Environment as seen by the students and showing the overall architecture of the Microprogrammable Processor Teaching Tool.

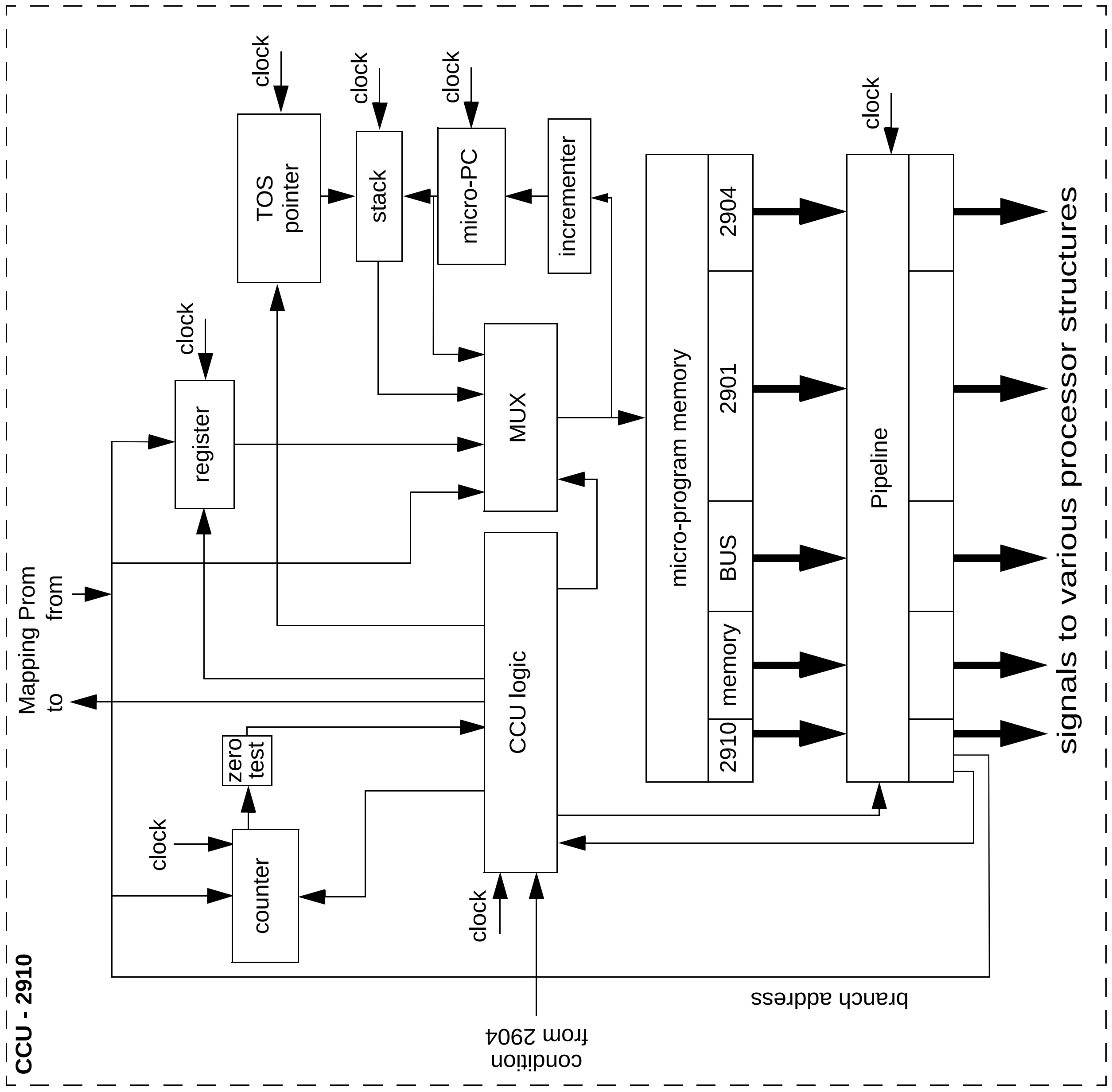


Figure 2 (b) Overall 2910 CCU structure.

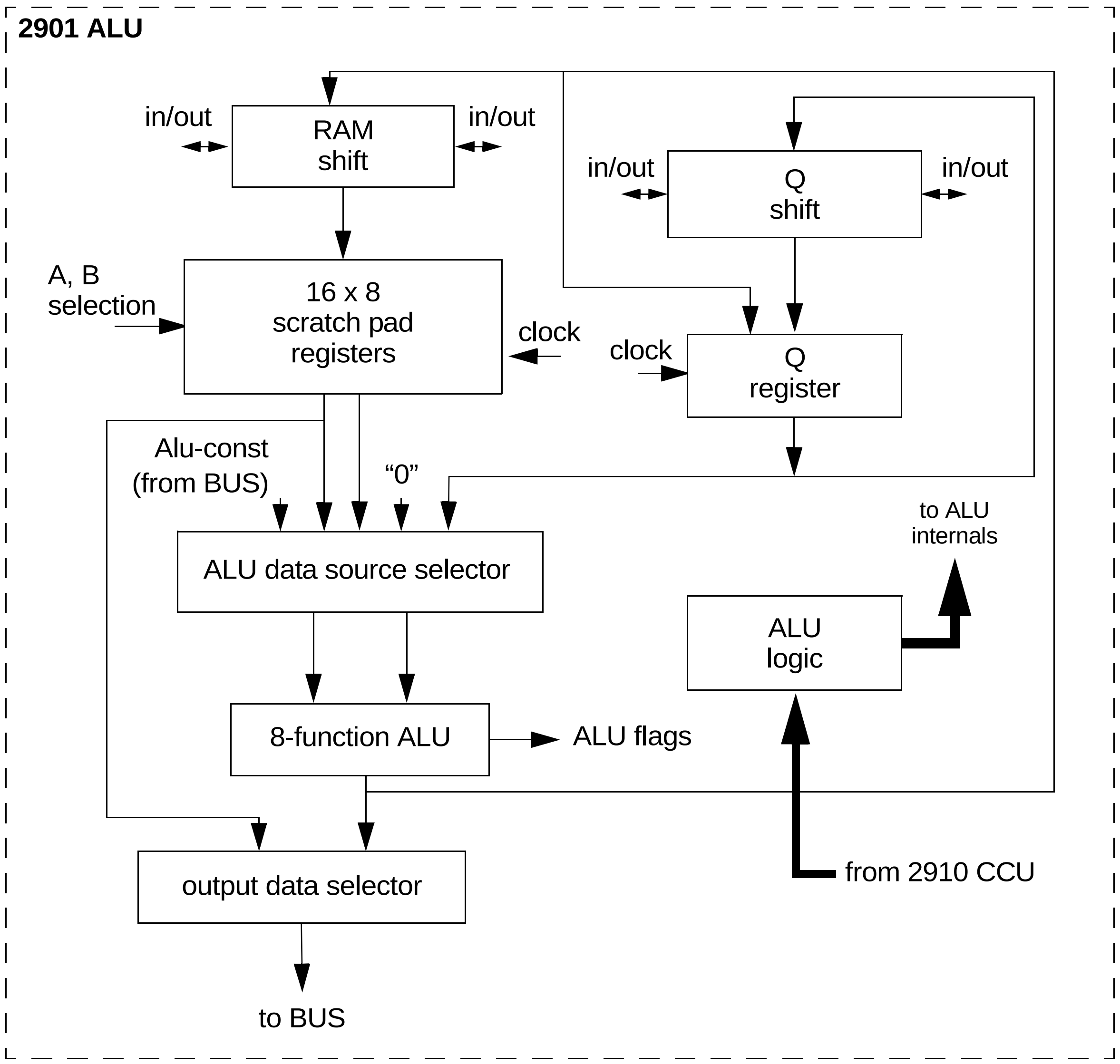


Figure 2 (c) Overall 2901 ALU structure [4,5].

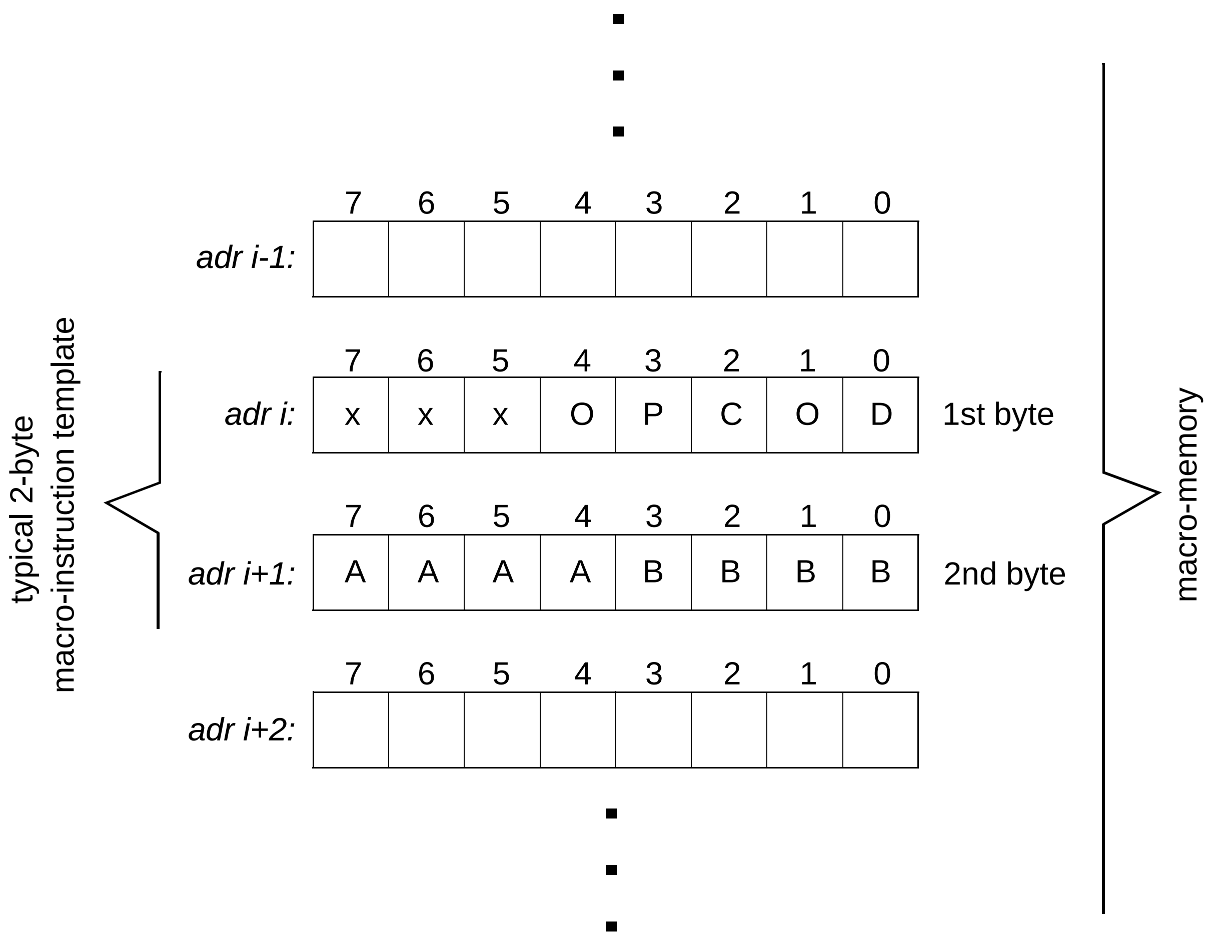


Figure 3 Format of a 2-byte macro-instruction located at macro-memory address “i.” OPCOD designates the 5 bit opcode, AAAA and BBBB the scratch pad register numbers and xxx are unused bits. The macro-memory is eight bit wide as shown.

```
" FETCHINSTR

" PC on bus, increment PC (R0), read macro-memory at PC value and save it in MBR:
WHEN (M==^h05)  THEN {
"AM2910:"             NAS=^h1; BA=^h0;
"MEMORY:"             READ=^b1;
"BUS:"                ALU_OE=^b1; ADR_LD=^b1;
"AM2901:"             AA=^h0; AB=^h0; I_SOURCE=^h3; I_FONCT=^h0; I_DEST=^h2;
                      CARRY=^b1;
}                                    micro-instruction address

" Save MBR content in most significant byte of IR (= Opcode):
WHEN (M==^h06)  THEN {
"AM2910:"             NAS=^h1; BA=^h0;
"BUS:"                MBR_OE=^b1; IR_MSB_LD=^b1;
"AM2901:"             NOP=^b1; I_SOURCE=^h0; I_FONCT=^h0; I_DEST=^h1; CARRY=^b0;
}

" PC on bus, increment PC (R0), read macro-memory at PC value and save it in MBR:
WHEN (M==^h07)  THEN {
"AM2910:"             NAS=^h1; BA=^h0;
"MEMORY:"             READ=^b1;
"BUS:"                ALU_OE=^b1; ADR_LD=^b1;
"AM2901:"             AA=^h0; AB=^h0; I_SOURCE=^h3; I_FONCT=^h0; I_DEST=^h2;
                      CARRY=^b1;
}

" Save MBR content in least significant byte of IR (= Operand):
WHEN (M==^h08)  THEN {
"AM2910:"             NAS=^h2;
"BUS:"                MBR_OE=^b1; IR_LSB_LD=^b1;
"AM2901:"             NOP=^b1;  I_SOURCE=^h0; I_FONCT=^h0; I_DEST=^h1; CARRY=^b0;
}
```

Figure 4 Micro-code listing of the FETCHINSTR micro-sequence with 4 micro-instructions (micro-memory addresses 5 to 8). Refer to Table 2 for Micro-instruction fields and subfields. Notes: ^h for hexadecimal, ^b for binary. See also Figure 1 for FETCHINSTR description.

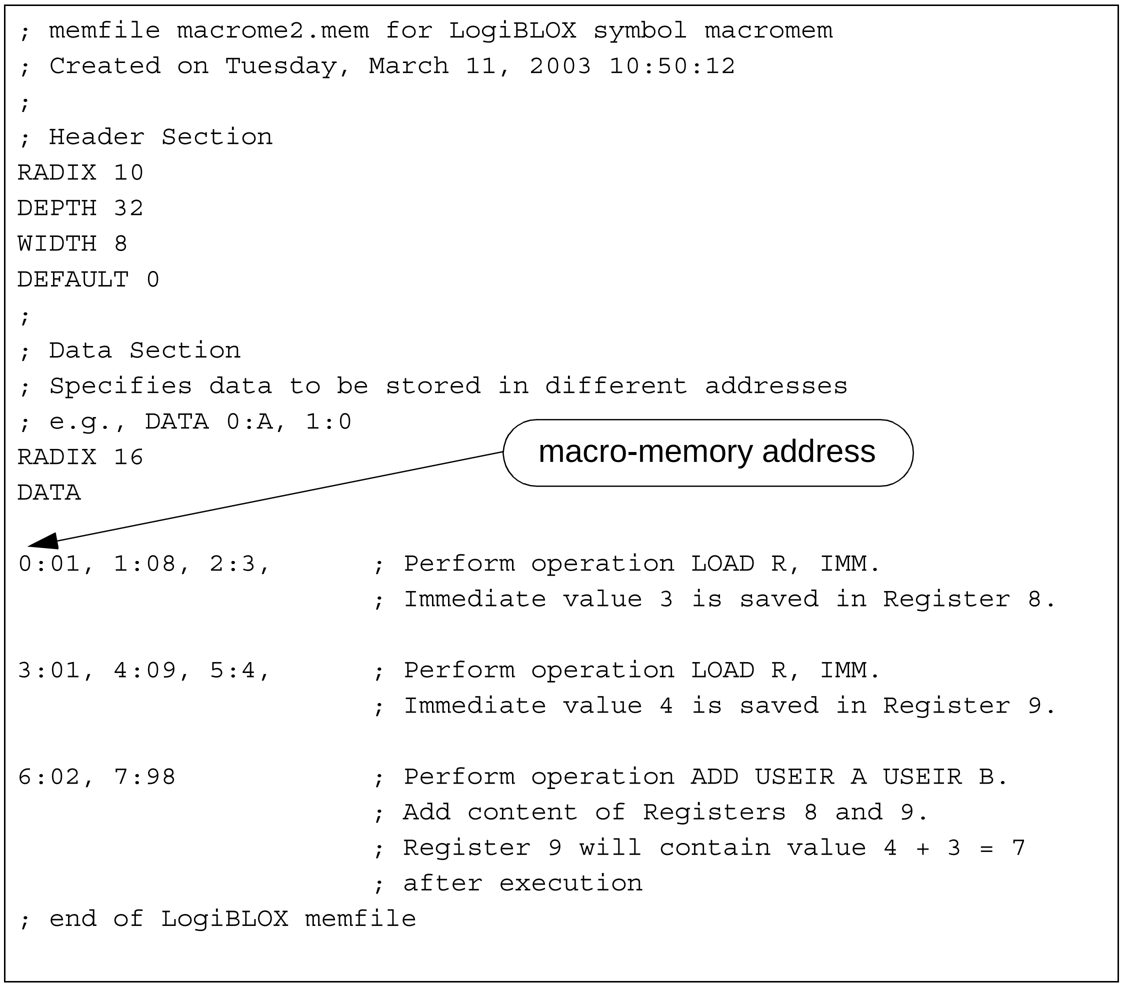
```
"LOAD R imm

" increment PC, prepare reading immediate value located at PC value in macro-
memory through the MEMORY BUFFER
WHEN (M==^h19) THEN {

"AM2910:"   NAS=^hl; BA=^h0;
"MEMORY:"   READ=^b1;
"BUS:"      ALU_OE=^bl; ADR_LD=^bl;
"AM2901"    AA=^h0; AB=^h0; I_SOURCE=^h3; I_FONCT=^h0; IDEST=^h2;
            CARRY=^bl;
}
" immediate value on bus and loaded in corresponding register (USIERB)
WHEN (M==^hlA) THEN {

"AM2910:"   NAS=^h3; BA=^h5;
"BUS:"      MBR_OE=^bl;
"AM2901"    USIERB=^bl; I_SOURCE=^h0; I_FONCT=^h0; I_DEST=^h3; CARRY=^b0;
"AM2904"    CCEN=^bl;
}
```

Figure 5 Micro-code listing of the *LOAD R imm* micro-sequence with 2 micro-instructions (micro-memory addresses ^h19 to ^h1A).


macro-memory address

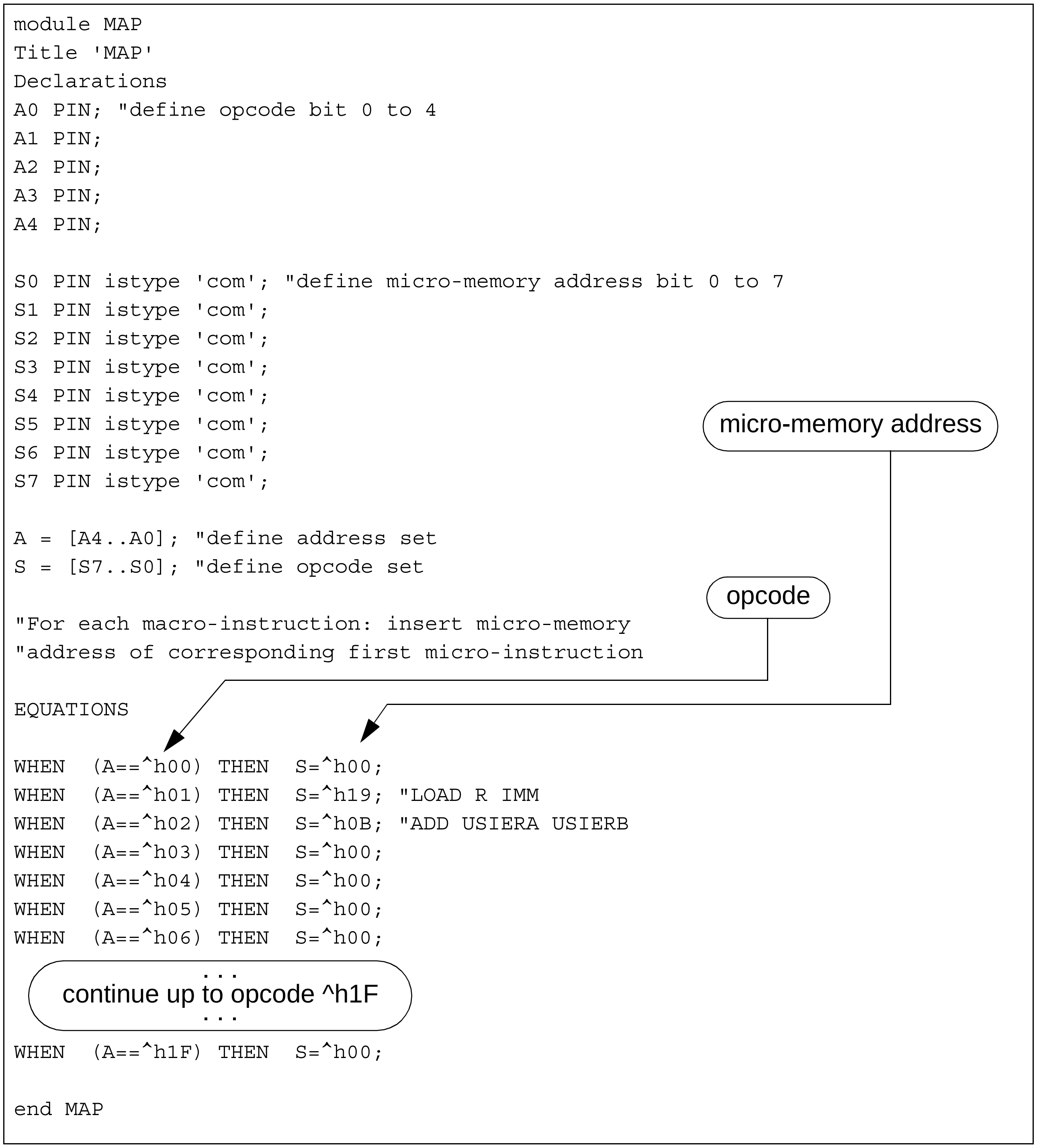
```
; memfile macrome2.mem for LogiBLOX symbol macromem
; Created on Tuesday, March 11, 2003 10:50:12
;
; Header Section
RADIX 10
DEPTH 32
WIDTH 8
DEFAULT 0
;
; Data Section
; Specifies data to be stored in different addresses
; e.g., DATA 0:A, 1:0
RADIX 16
DATA

0:01, 1:08, 2:3,      ; Perform operation LOAD R, IMM.
                      ; Immediate value 3 is saved in Register 8.

3:01, 4:09, 5:4,      ; Perform operation LOAD R, IMM.
                      ; Immediate value 4 is saved in Register 9.

6:02, 7:98            ; Perform operation ADD USEIR A USEIR B.
                      ; Add content of Registers 8 and 9.
                      ; Register 9 will contain value 4 + 3 = 7
                      ; after execution
; end of LogiBLOX memfile
```

Figure 6 Example of a macro-instruction listing, two 3-byte macro-instructions and one 2-byte macro-instruction. Macro-memory addresses 0 to 7.


micro-memory address
opcode
continue up to opcode ^h1F

```
module MAP
Title 'MAP'
Declarations
A0 PIN; "define opcode bit 0 to 4
A1 PIN;
A2 PIN;
A3 PIN;
A4 PIN;

S0 PIN istype 'com'; "define micro-memory address bit 0 to 7
S1 PIN istype 'com';
S2 PIN istype 'com';
S3 PIN istype 'com';
S4 PIN istype 'com';
S5 PIN istype 'com';
S6 PIN istype 'com';
S7 PIN istype 'com';

A = [A4..A0]; "define address set
S = [S7..S0]; "define opcode set

"For each macro-instruction: insert micro-memory
"address of corresponding first micro-instruction

EQUATIONS

WHEN  (A==^h00) THEN  S=^h00;
WHEN  (A==^h01) THEN  S=^h19; "LOAD R IMM
WHEN  (A==^h02) THEN  S=^h0B; "ADD USIERA USIERB
WHEN  (A==^h03) THEN  S=^h00;
WHEN  (A==^h04) THEN  S=^h00;
WHEN  (A==^h05) THEN  S=^h00;
WHEN  (A==^h06) THEN  S=^h00;
              . . .
              . . .
WHEN  (A==^h1F) THEN  S=^h00;

end MAP
```

Figure 7 Example of an ABEL™ Mapping Prom listing showing defined opcodes 01 and 02 for macro-instructions *Load R IMM* and *ADD USEIRA USEIRB* respectively. *USEIRx* means operand comes from the macro-instruction (*x* = *A* or *B*).