

Novel assembler to facilitate the understanding of how processors work

H.Pomares^{*1}, I.Rojas¹, M.Damas¹, L.J.Herrera¹, A.Guillén¹, G.Rubio¹, A.Jiménez-Hubeaux² and P.Ibáñez³

¹Dpto. Arquitectura y Tecnología de Computadores, ETS de Ingeniería Informática y de Telecomunicación – University of Granada, Granada – Spain

²Student of Computer Engineering Projects of the ETS de Ingeniería Informática y de Telecomunicación of the University of Granada, Spain

³MSc. in Computer Science in the University of Granada, Spain

This paper presents a new, easy-to-use assembler that will facilitate the comprehension of how processors work. This assembler has been tested during the last year and will be used within the department of Computer Architecture and Computer Technology of the University of Granada, as a practice tool for courses related to the study of the structure and organization of computers. This assembler completes the software environment utilized during several years in the above-mentioned department and is freely available in the World Wide Web.

Keywords Didactic computer; Computer architecture; Assembler language;

1. Introduction: The elementary didactic computer CODE-2

In the first courses of Computer Science or any other computer related degree, when we try to teach how processors are built and how they work, teachers do often encounter the difficulty of having to make up simple dynamical models in order to facilitate the visualization and understanding of what is happening inside a computer or even a processor.

CODE-2 (Fig.1) is a very simple Von Neumann-type computer that is composed (as all computers of this type) of: input/output unit, control unit, processing unit and a memory for data and instructions, and which has been recently built in the department of computer architecture and computer technology of the University of Granada [1,2]. CODE-2 has all the basic elements of a RISC processor [3], but with only 16 machine instructions. Using and studying CODE-2 in the first courses, students can comprehend the information flux within the computer, and understand the basics of how computers work, without having to deal with the very complex structures that current processors have [4].

From the architectural point of view, the elements that can be accessed and should be known in order to write a computer program for CODE-2 are (see Fig. 2):

- A register file, comprised of 16 registers (r0...rF). All can be accessed as general-purpose registers, although rE is normally used as the stack pointer and rD as an address register (see the CODE-2 instruction set in Table 1).
- An arithmetic-logic unit (ALU), which can make additions, subtractions, the NAND logic operation and several kinds of logical shifts.
- Flags: zero (Z), sign (S), carry (C) and overflow (V).
- Main memory, with 64Kwords of 16 bits (128KB).
- Input ports; up to 256 input ports (from IP00 to IPFF).
- Output ports; up to 256 output ports (from OP00 to OPFF).

Table 1 shows the 16 instructions that can be executed in CODE-2 together with the parameters defined for each one of them, and a little explanation of what every instruction does. To understand the exact

* Contact Author: e-mail: hpomares@atc.ugr.es

machine language format of these instructions, which will be necessary for the reader to test the correctness of the output of the proposed assembler in Section 3, Fig 3 shows the five possible instruction formats defined for this processor.

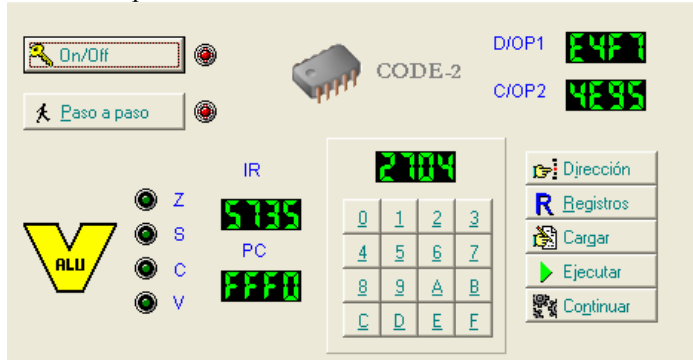


Fig. 1 Front panel of CODE-2.

2. Description of the designed assembler for CODE-2

An assembler is an application capable of translating a program written in assembler language into machine code, that is, a sequence of binary digits that the processor, in this case CODE-2, can understand. To implement an assembler we need a lexicon analyser and a syntactic analyser, such as LEX and YACC [5]. Apart from the 16 instructions defined in the instruction set of CODE-2, we have endowed our CODE-2 assembler with the following features:

2.1 General features of the CODE-2 assembler

One instruction per line: It is a condition of the assembler that there is only one instruction per line in the source file.

Case sensitiveness: The assembler is not case sensitive.

Number base system: Numerical values can be specified in

- Hexadecimal: H'2A
- Decimal: D'42
- Octal: Q'52
- Binary: B'00101010

Labels: The assembler deals with labels as if they were either memory addresses or integers. Labels cannot start by a number and when defined, they should end with a colon.

Comments: Every line in a source file can have a comment. Comments are preceded by a semicolon sign.

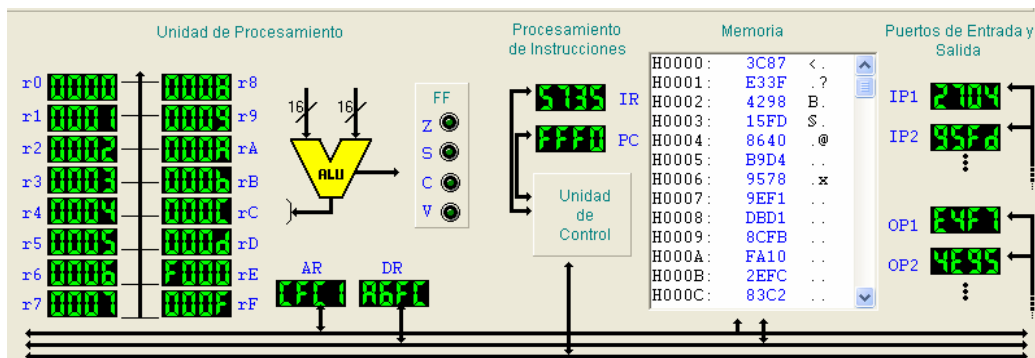
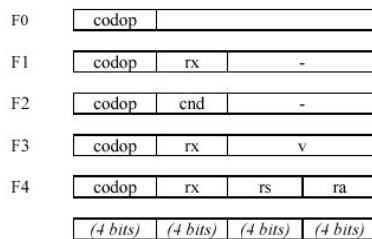


Fig. 2 Schematic internal structure of CODE-2.**Table 1** CODE-2 instruction set

Hex	Name	Mnemonic	Parms	Format	Operation
0	Load	LD	rx,[v]	F3	$rx \leftarrow M(rD+v)$
1	Store	ST	[v],rx	F3	$M(rD+v) \leftarrow rx$
2	Load Low Immediate	LLI	rx,v	F3	$rx(15:8) \leftarrow H'00;$ $rx(7:0) \leftarrow v$
3	Load High Immediate	LHI	rx,v	F3	$rx(15:8) \leftarrow v$
4	Input	IN	rx,IPv	F3	$rx \leftarrow IPv$
5	Output	OUT	OPv,rx	F3	$OPv \leftarrow rx$
6	Add	ADDS	rx,rs,ra	F4	$rx \leftarrow rs+ra$
7	Subtract	SUBS	rx,rs,ra	F4	$rx \leftarrow rs-ra$
8	NAND	NAND	rx,rs,ra	F4	$rx \leftarrow (rs-ra)'$
9	Shift Left	SHL	rx	F1	$C \leftarrow rx(15), rx(i) \leftarrow rx(i-1), i=15, \dots, 1;$ $rx(0) \leftarrow 0$
A	Shift Right	SHR	rx	F1	$C \leftarrow rx(0), rx(i) \leftarrow rx(i+1), i=0, \dots, 14;$ $rx(15) \leftarrow 0$
B	Shift Right Arithmetic	SHRA	rx	F1	$C \leftarrow rx(0), rx(i) \leftarrow rx(i+1), i=0, \dots, 14$
C	Branch	B-	cnd	F2	If cnd is true, $PC \leftarrow rD$
D	Call Subroutine	CALL-	cnd	F2	If cnd is true, $rE \leftarrow rE-1, M(rE) \leftarrow PC,$ $PC \leftarrow rD$
E	Return	RET	-	F0	$PC \leftarrow M(rE); rE \leftarrow rE+1$
F	Halt	HALT	-	F0	Halt

**Fig. 3** Instruction formats defined for CODE-2.

2.2 Assembler directives

We have provided our CODE-2 assembler with the following directives in order to make program writing easier:

ORG dir: When this directive is found in a source file, all the assembler instructions written from that moment will be mapped starting from the memory address *dir*. The programmer can use as many ORG directives as desired. All instructions will be mapped by default from the memory address H'0000.

LO(labelname): This function returns an 8-bit integer which represents the lower or least significant byte of the memory address/value represented by label *labelname*.

HI(labelname): This function returns an 8-bit integer which represents the higher or most significant byte of the memory address/value represented by label *labelname*.

INCLUDE "filename": This directive allows the programmer to include to their source file other program written in assembler. This directive can be inserted in any place in the source file and can be used as many times as desired.

EQU: This directive lets us define constants or any new names to the existing general-purpose registers. The format is: *name EQU expression*.

DW: This directive allocates one word (16 bits) in the main memory and can also initialize that location with a specified value. The format is: *name DW initial_value*

DATA value,[value...]: With this directive, we can store any 16 bit value in the CODE-2 main memory. To the right of the DATA directive we should write the integer, or a list of integers separated by commas, that we want to store in the memory.

#B- o #CALL- label: In order to facilitate to the programmer the way branches and calls are specified, this pseudo-instruction allows them to specify the branch target address or the subroutine target address by means of a label.

DR: This directive defines a new name for one of the CODE-2 registers and initializes it with a specified value. This directive can only be used at the beginning of the program, before any assembler instruction. The format is: *name DR value*.

2.3 Output files generated by the assembler

The assembler we have designed for CODE-2 generates several output files as the result of the compilation process of a source program. All output files are ASCII files which can be edited and easily understood by the student. There are four main output files:

Intel HEX format file (.HEX): This is the standard format for specifying (E)EPROM memory values. Every line of this file, which is called a record, has the format:

```
RECORD MARK'; LOAD RECLen; OFFSET; RECTYP; INFO or DATA; CHKSUM
```

where RECORD MARK is a colon that indicates the beginning of a new line, LOAD RECLen specifies the number of data bytes of the record, OFFSET is a 16-bit number that specifies the memory address from which this record is to be stored, RECTYP indicates which type of record this one is about (data record, end-of-file record, etc), INFO or DATA contains the actual information of the record, that is, the actual memory values that we want to store, and finally CHKSUM is used to ensure the integrity of the register, so that the addition of all fields, including this one, must be zero.

Easy Hexadecimal Code file (.EHC): This is a simplified version of the .HEX format, which is much more understandable for the students and also permits comments to be inserted. This file specifies only one memory position per line, and the format of each line is: OFFSET DATA; COMMENTS. An example of this file format can be seen later in the next section.

Error file (.ERR): This file contains all error messages and warnings generated by the assembler. Again, an example of these messages will be given in the next section.

Tabular file (.TAB): This file is a slightly different version of the .EHC which is specially generated to facilitate the final program presentation to the teacher. This is a semicolon separated file which can be easily integrated into Microsoft Word™ in order to generate a table of the form: *Label-MemoryAddress-Mnemonic-HexCode-Comments*, where *mnemonic* and *comments* are exactly the instructions and the comments that the programmer wrote in the source file before the assembling process.

3. Examples

The CODE-2 assembler we are presenting here has been successfully and extensively used in subjects related to Computer Science Introduction in the Computer & Electrical Engineering School of the University of Granada. This assembler is available via web from the web site: <http://atc.ugr.es/~hector/ic>.

Figure 4 shows a simple example of the designed assembler. On the left part of the screen, the programmer can write the CODE-2 source code. At the bottom, all error messages are displayed. On the right part, the assembled instructions are shown, using the aforementioned .EHC format. It should be

noted that in this example since the label *hre* has been misspelled, the assembler does not know all information to generate some instructions and therefore replaces all unknown hexadecimal values by interrogation signs. Finally, Fig. 5 shows an error-free complete program written for CODE-2 with the corresponding output file generated. In this example we can see some of the features that we have mentioned in the previous section and which are self-explanatory.

The screenshot shows the CODE-2 assembler interface with two panes: 'CÓDIGO ENSAMBLADOR' (Assembly Code) and 'CÓDIGO HEXADECIMAL (.ehc)' (Hexadecimal Code). The assembly code pane shows lines 1-11, with line 9 containing a misspelled label 'hre'. The hexadecimal code pane shows lines 1-8 with some values replaced by '??'. A status bar at the bottom displays three error messages: 'Error Semántico: Error (111) el valor inmediato es mayor de 255 en C:\qgtmp\EX1R.asm en la línea 3', 'Error Sintáctico: Error (7) esperaba un registro o un identificador y en cambio aparece h'01, en C:\qgtmp\EX1R.asm en la línea 7', and '>> Error Semántico: Error (113) la etiqueta hre no ha sido definida en C:\qgtmp\EX1R.asm en la línea 9'.

Fig. 4 The CODE-2 assembler: Error messages and temporal assembled code

The screenshot shows the CODE-2 assembler interface with two panes: 'CÓDIGO ENSAMBLADOR' (Assembly Code) and 'CÓDIGO HEXADECIMAL (.ehc)' (Hexadecimal Code). The assembly code pane shows lines 1-29, including a loop structure and a data section. The hexadecimal code pane shows lines 12-28, corresponding to the assembly code. The status bar at the bottom is empty, indicating no errors.

Fig. 5 The CODE-2 assembler: Final program in machine code.

4. Conclusions

In this paper we have presented an assembler for CODE-2. This assembler has been specifically designed to and contributes to make computer operation less difficult to understand. The proposed assembler has been tested during the last year and has proved successful within the department of Computer Architecture and Computer Technology of the University of Granada, as a practice tool for courses related to the study of the structure and organization of computers.

References

- [1] A. Prieto and A. Lloris, *Introducción a la Informática*, 3th edition, McGraw Hill, 2004
- [2] J. Díaz, *Prototipo hardware de CODE-2*, proyecto fin de carrera, Universidad de Granada, 2002.
- [3] C. Hamacher, Z. Vranesic and S. Zaky, *Organización de Computadores*, McGraw Hill, 3th edition, 2000.
- [4] W. Stallings, *Computer Organization & Architecture: Designing for Performance*, Prentice-Hall, 6th edition, 2003.
- [5] A. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1988.
- [6] A. Jiménez-Hubeaux, *Ensamblador para CODE-2*, proyecto fin de carrera, Universidad de Granada, 2005.