# A RUDIMENTARY MACHINE.<br>EXPERIENCES IN THE DESIGN OF A PEDAGOGIC COMPUTER

Enric Pastor, Fermín Sánchez and Anna M. del Corral
Department of Computer Architecture
Universitat Politècnica de Catalunya
enric, fermin, anna @ac.upc.es

**Abstract**

This paper describes both the structure and the architecture of a pedagogic computer named "Máquina Rudimentaria". This computer has been designed to be used in a first course on logic design or computer architecture. Therefore, the orthogonality and simplicity have been the major goals. The description of the computer includes the definition of both the machine and the assembly language, the implementation of the data-path and the optimization of the control unit. A programming framework has also been designed to provide the development of small complexity programs in assembly language, its compilation and detailed simulation on the proposed architecture.

## 1 Introduction

The *Máquina Rudimentaria* (MR) [1] intends to be a simple pedagogic computer. The main objective pursued in the design is to guarantee the orthogonality and simplicity. Therefore, the MR may be used as a tool to teach basic concepts about structure and architecture of computers in the first courses of engineering or computer science studies. Since the design of the MR only includes basic concepts about digital systems, it can also be used as a complex case study in VLSI design courses. In that sense, the MR offers a common point that allows a natural progression from the study of logic design to the basic concepts of computer architectures and further into VLSI design.

Today, an extremely large number of processors exists designed both in the commercial market or in universities. So, is it worth to design new pedagogical processors instead of taking from the shelf some already existing design? The answer to this question is affirmative, because the complexity of the processor to be used strongly depends on the context in which should be applied.

In order to use a processor in an initial course on computer architecture, old commercial products like the VAX [2] or Motorola series should be disregard because they include large instruction sets and complex addressing modes. Processors in the Intel 80x86 family [3] should be also disregard because their lack of orthogonality and the number of side concepts that must be introduced (even the old 8086 is too cumbersome). Even more recent RISC processors, like the SPARC family, still are beyond the required complexity. Existing pedagogic processors designed in universities like the DLX [4] are typically well documented and allow to introduce advanced concepts on them. However, they are also still far too complex to be easily assimilated by students with very rudimentary knowledge about computer architecture.

Due to the fact that no suitable processor was found for our purpose, a few years ago the design of a new pedagogic computer was started. The main goals were simplicity and orthogonality. The instruction set should be complete enough to design medium complexity programs (following a RISC philosophy), but no subroutines or interruptions were included. Both the bus and the memory subsystem should be very simple, with emphasis in the design of the datapath and the control unit, and no I/O were incorporated. The result is the processor described in this paper.

## 2 The Architecture of the MR

### 2.1 Basic Structure

The MR is a classic von Neumann computer without any I/O facility (for the sake of simplicity). The memory is organized in 256 words of 16 bits, with separated ports for reading and writing, and a read/write control signal. The CPU is clearly divided into a data-path and a centralized Control Unit, described in sections 3 and 4 respectively.

The instructions of the MR work on integer numbers of 16 bits using two's complement encoding. Addresses have 8 bits matching with the size of the memory. The CPU is able to perform basic integer operations on data stored in the 8 registers available in the Register File. Register R0 is dedicated to store the constant 0. Additionally, there exist two condition *flags* for the sign (N) and for zero (Z) of the result.

| Instruction | Operation |
|---|---|
| ADDI Rs, #inmediate, Rt | Rt := Rs + inmediate |
| SUBI Rs, #inmediate, Rt | Rt := Rs − inmediate |
| ADD Rs1, Rs2, Rt | Rt := Rs1 + Rs2 |
| SUB Rs1, Rs2, Rt | Rt := Rs1 − Rs2 |
| ASR Rs, Rt | Rt := Rs >> 1 |
| AND Rs1, Rs2, Rt | Rt := Rs1 $\wedge$ Rs2 |
| LOAD base_addr(Ri), Rt | Rt := M[base_addr + Ri] |
| STORE Rs, base_addr(Ri) | M[base_addr + Ri] := Rs |

| Instruction | Branch condition | |
|---|---|---|
| BR target_addr | unconditional | 1 |
| BEQ target_addr | equal | Z |
| BL target_addr | less | N |
| BLE target_addr | less or equal | $N \vee Z$ |
| BNE target_addr | different | $\overline{Z}$ |
| BGE target_addr | greater or equal | $\overline{N}$ |
| BG target_addr | greater | $\overline{N \vee Z}$ |

Figure 1: The Assembly language for the MR.

## 2.2 The Assembly Language of the MR

The instructions are of a fixed length of 16 bits (the word of the computer). The operation code is always stored in the two more significant bits, describing four types of instructions: Arithmetic-logical instructions, Data transfer instructions (two codes), and Branch instructions. The MR offers four basic addressing modes: *register* mode, *immediate* mode, *relative* mode (base address + offset) and *absolute address* mode.

### 2.2.1 Arithmetic-logical instructions

The arithmetic-logical instructions can use both the register and immediate addressing modes. When using the register mode the register file is accessed; when using the immediate mode a 5 bits constant coded in the instruction is accessed. All arithmetic-logical instructions store their result into the register file (Rt) and activate the condition flags N and Z. If the instruction writes its result into the register R0 (constantly set to 0) no result is stored but the condition flags are updated.

According to the source operands, two types of arithmetic-logical instructions exist: the first type reads all its source operands from the register file; the second reads one operand from the register file and the second from the instruction itself (immediate mode). The MR offers six arithmetic-logical instructions (see Figure 1) namely:

- **Register addition**: adds the contents of two registers (Rs1 + Rs2).
- **Register subtraction**: subtracts the contents of two registers (Rs1 - Rs2).
- **Arithmetic shift**: shifts a register one bit to the right with sign extension (Rs >> 1).
- **Logic AND**: bit level logic *and* between two registers (Rs1 $\wedge$ Rs2).
- **Immediate addition**: adds the contents of one register and the immediate operand extended to 16 bits (Rs + #n).
- **Immediate subtraction**: subtracts the contents of a register and the immediate operand (Rs - #n).

Figure 2(a) shows the encoding used for each instruction. All arithmetic-logical instructions use the same operation code (11). An additional field OP (bits 2-0) identifies each particular instruction (see Figure 2(b)). Bit $OP_2$ separates between register-register instructions ($OP_2$=1) and register-immediate instructions ($OP_2$=0).

### 2.2.2 Data transfer instructions

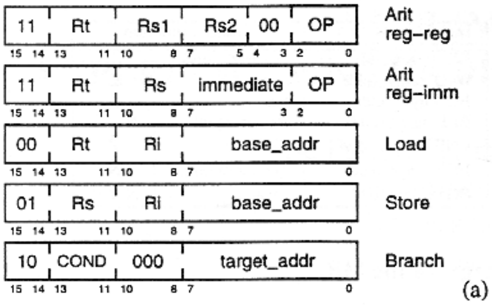
There exists only two data transfer instructions (see Figure 1). Load and Store instructions use the register and the relative addressing modes. The target memory address is computed by adding the contents of the eight less significant bits of an index register (Ri), the offset, to an 8 bits address stored in the instruction (base_addr), the base. The detailed format of each instruction is described in Figure 2(a):

- **Load**: reads the contents of the memory for the target address and stores the value in a register (Rt), updating the condition flags N and Z. If the destination register is R0, only the flags are modified. The operation code is 00.
- **Store**: writes the contents of a register (Rs) into the memory for the selected target address. The flags N and Z are not modified. The operation code is 01.

### 2.2.3 Branch instructions

All branch instructions use the absolute mode (8 bits) to indicate the branch target address. None of the branching instructions modifies the value stored in the condition flags. There exists six condition branch instructions and one unconditional branch instruction, as shown in Figure 1. Figure 2(a) describes the encoding used for each branch instruction. All of them share the operation code (10). The field COND (bits 13-11) identifies each individual instruction according to Figure 2(c) namely:

- **Branch if greater**: jumps if N=0 and Z=0.
- **Branch if greater or equal**: jumps if N=0.



| Arithmetic instruction | OP |
|---|---|
| Add reg-immediate | 000 |
| Sub reg-immediate | 001 |
| Add reg-reg | 100 |
| Sub reg-reg | 101 |
| Shift right | 110 |
| Logic AND | 111 |

(b)

| Branch instructions | COND |
|---|---|
| Unconditional | 000 |
| Equal | 001 |
| Less | 010 |
| Less or equal | 011 |
| Different | 101 |
| Greater or equal | 110 |
| Greater | 111 |

(c)

Figure 2: Machine language format for the MR.

- **Branch if less**: jumps if N=1.
- **Branch if less or equal**: jumps if N=1 or Z=1.
- **Branch if equal**: jumps if Z=1.
- **Branch if different**: jumps if Z=0.
- **Unconditional branch**: always jumps.

# 3 Data-path

The set of instructions described in Section 2 can be sequentially executed (not in a pipelined way) in the data-path described in Figure 3. The data-path contains six special registers:

- The program counter (PC),
- the instruction register (IR),
- the address register (R@),
- the ALU register (RA) and
- the sign and zero flags (RN and RZ).

Four elemental parts can be distinguished in the data-path:

- **Register file control:** The register file has 8 16-bit registers. R0 is a special register that always contains a 0. The register file provides one read port and one write port that can be simultaneously used. It always provides the contents of the register corresponding to the read register input on the output, while the write must be explicitly controlled by the Control Unit with the write control signal WRt.

  The write register input is directly taken from the bits 13-11 of the instruction (target register in arithmetic-logical and load instructions). The read register input can be selected from three different inputs:

  - the fields that codify the source registers in arithmetic-logical instructions (bits 10-8 and 7-5),
  - the index register in load/store instructions (bits 10-8) and
  - the source register in store instructions (bits 13-11).

  The selection is done by means of the mux *SELREG*. Such a mux is controlled by the Control Unit by means of bits *CRs*.

- **Arithmetic-logical computation:** This part includes:

  - the Arithmetic-Logical Unit
  - a 16-bit register to store the first operand (it is always a register from the register file)
  - a mux to select the source of the second operand. The input of the mux is selected by the Control Unit. The second operand me be on the register file, the memory or the instruction register when it is immediate. In this case, the operand is sign-extended by using a sign-extension unit that has a 5-bit input that is sign-extended into a 16-bit result in the output.
  - the flags *Z* and *N*.

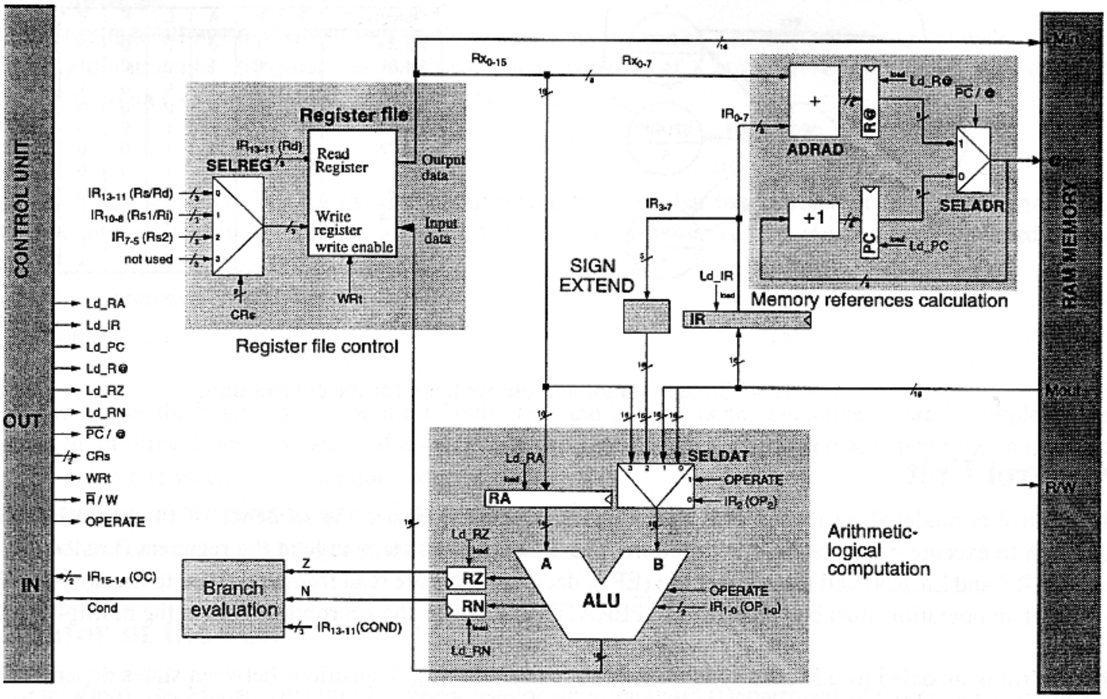


Figure 3: Data-path of the MR.

The ALU can perform all the operations required by the instructions (addition, subtraction, 1-bit shift right or and logical) and also allows the second operand to appear in the output (in order to activate the flags in the execution of load). The operation to be performed is controlled by means of three bits. The *OPERATE* bit proceeds from the Control Unit. This bit is used to decide if the second operand appears in the output (*OPERATE=0*) or an arithmetic-logical operation must be performed (*OPERATE=1*). In the last case, the operation is decided by two bits that are directly connected to the lower-weight bits of the *IR*. Thus, the field *OP* of the arithmetic-logical instructions directly selects the operation to be performed in the ALU. The ALU also calculates the values Z and N for the flags.

- **Memory references calculation:** The memory references performed during the execution of a program are calculated in this module. The Control Unit selects between the Program Counter (*PC*) or the Addresses Register (*R@*) by means of the mux *SELADR*.

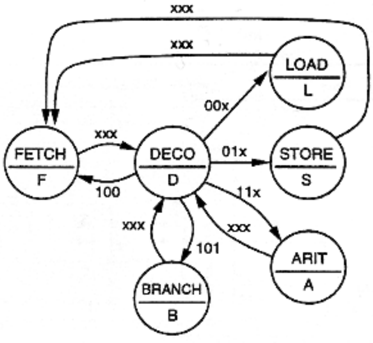
- **Branch evaluation circuit:** This specific combinational circuit generates one bit that reports to the Control Unit the result of the branch evaluation. Its inputs are the two condition flags and the three bits that identify the branch instruction (bits 13-11). If the instructions produces a branch, the branch target address is taken from the address register.

Memory references can be generated by three different events:

- **Fetching consecutive instructions**: Instructions are 16-bit length. Since a memory reference contains a 16-bit word, two consecutive instructions are stored in consecutive addresses. This module provides an adder to increment the *PC* (for the sake of simplicity, the adder is denoted as [+1] in Figure 3). This mechanism allows to make ready the fetch of the next instruction in sequence.

- **Execution of memory-reference instructions**: Load and store instructions reference a memory word by means of the relative addressing mode. Then,the address of such a word is calculated by adding an offset (contained in the bits 7-0 of the instruction) to a baseaddress (stored in a register of the register file). To calculate the address, the Control Unit selects the first input of the mux *SELREG* in order to read the index register from the register file.

- **Read the contains of a branch target address**: In order to store the branch target address in the *R@* register, bits 7-0 of the branch instruction (branch target address) are added to zero in the *ADRAD adder*. The Control Unit achieves a zero in the other input of the adder by selecting the first input of the mux *SELREG*. Since the value of bits 10-8 in branch instructions is '000', register R0 (which always contains a zero) is selected from the register file.



(a)

| Outputs UC | F | D | A | L | S | B |
|---|---|---|---|---|---|---|
| Ld_RA | 0 | 1 | 0 | 0 | 0 | 0 |
| Ld_IR | 1 | 0 | 1 | 0 | 0 | 1 |
| Ld_PC | 1 | 0 | 1 | 0 | 0 | 1 |
| Ld_R@ | 0 | 1 | 0 | 0 | 0 | 0 |
| Ld_RZ | 0 | 0 | 1 | 1 | 0 | 0 |
| Ld_RN | 0 | 0 | 1 | 1 | 0 | 0 |
| WRt | 0 | 0 | 1 | 1 | 0 | 0 |
| $\overline{\text{R}}$/W | 0 | 0 | 0 | 0 | 1 | 0 |
| $\overline{\text{PC}}$/@ | 0 | x | 0 | 1 | 1 | 1 |
| CRs | x | 1 | 2 | x | 0 | x |
| OPERATE | x | x | 1 | 0 | x | x |

(b)

Figure 4: State diagram and output table for the control unit.

# 4 Control Unit

The control is modeled by means of a state diagram, which indicates the sequence of the operations performed in the data-path to execute each machine instruction. The Control takes steps to load the registers (Ld_RA, Ld_IR, Ld_PC, Ld_R@, Ld_RZ and Ld_RN) and the register file (ERt), decides if data are read from or written to memory ($\overline{\text{R}}$/W), indicates to the ALU if an operation must be performed (OPERATE) and selects the appropriate data in the multiplexors ($\overline{\text{PC}}$/@ and CRs).

The control is modeled as a Moore synchronous state machine [5]. Transitions between states depend on the external inputs of the system. The next state is here decided by using both the operation code (bits 14 and 15 of the instruction) and the branch evaluation bit. Moore type external outputs are dependent on only the present state of the circuit, and are independent of the external inputs of the circuit. The operations to perform in the data-path on each state are defined in an output table, related to the output of each state in the Moore machine. Figure 4 shows the Moore machine and the output table of the Control Unit.

# 5 Assembly Language

An assembly language (LE) has been designed to simplify program development in the MR. The LE contains the basic elements to define instructions, labels, directives, expressions and macros.

### Instructions

The LE encodes each one of the available instructions by using a single mnemonic code. Figure 1 shows a complete list of mnemonic codes and the mechanisms to specify the operands.

### Labels

Labels are a useful mechanism to define references to instructions or data without requiring the knowledge of its precise location in the memory space. A label is defined at the beginning of a line by using an identifier followed by the symbol ":". The assembly process, in charge of translating each instruction to machine language, computes in an initial step the precise address corresponding to each label and stored both, the label and the actual address, in a *symbol table*. A second step uses the contents of the symbol table to complete the translation of all instructions containing labels.

### Directives

Directives are indications introduced in the program to drive the assembly process. There exists two basic types of directives in the LE of the MR:

- **Directives to define variables and constants**: used to reserve and initialize memory space. They are also used to assign values to the symbols used along the program.

  | | |
  |---|---|
  | *.dw* $n_1$ $\{, n_2, ..., n_N\}$ | initializes consecutive positions of the memory to the specified values. |
  | *.rw* $n$ | reserves $n$ consecutive words in the memory without any initial value. |
  | *identifier* $= n$ | defines an equivalence between a symbol and a numeric value. |

- **Directives for execution control**: Used to indicate the first and the last instructions to be executed.

  | | |
  |---|---|
  | *.begin* identifier | defines the address of the first instruction to be executed by the program. |
  | *.end* | indicates the end of the program. |

### Arithmetic Expressions

The assembly language allows the use of arithmetic expressions instead of numeric values. Arithmetic expressions are evaluated during the assembly process. An expression may consists of a numeric value, a label, or the addition (+), subtraction (−), product (×) or division (/) of two expressions.

### Macros

A *macro* or "pseudo-instruction" is a set of parametrizable instructions that can be referenced at any point of a program. There exists two special directives to define a macro; the body (instructions) of the macro must be placed between both directives:

*.def* name {parameter{, parameter}}

{ body of the macro }

*.enddef*

Directive *.def* indicates the beginning of the macro definition and sets its name. Directive *.enddef* completes the macro. Parameters can be of three different types, and must be preceded by the symbols $d, $i o $, depending on its type:

- $\$dn$: the parameter references a memory address,
- $\$in$: the parameter references an immediate constant, and
- $\$n$: the parameter references a register in the register file.

## 6 Simulator of the MR

A graphic simulator of the MR architecture has been developed in the DAC (Department of Computer Architecture). The simulator includes an assembler program to translate assembly code to machine code. Simulator is available via `ftp` at the address `ftp://ftp.ac.upc.es/pub/archives/mr`. Versions for DOS and WINDOWS environments can be freely downloaded.

An assembly program must be written into a file with the extension `''.asm''`. The assembly process generates another file, with th same name but with extension `''.cod''`. This file contains the translation to MR machine language. The program that performs such a translation is denoted postassembler.

When macros are used, the assembly process requires an additional step to expand macros' code. A program that uses or contains macros must be written into a file with the extension `''.mr''`. The macros' definition may be included in the same file or in another one with the same extension. The program that translate macros and generates MR assembly code is denoted preassembler.

The execution of a program contained in a file with extension `''.cod''` can be simulated by using the tool described in this section. The simulator allows the user to stop the execution of a program at any clock cycle or after finishing every instruction. The user may run the program until the end or stop its execution by using break points. The contains of the register file and the memory are presented at each moment. The buses that are active at each step are presented in white color (instead of the normal grey), as well as the contains of all the registers of the datapath and the value of the output signals of the Control Unit. The current state of the Control Unit is represented shadowed in the state diagram that is below the data path. The window on the right contains the user program. The user may expand or unexpand macros (macros may be nested) and put or get break points at any place in the program.

Simulator also allows to build timing diagrams with the control signals and buses of the datapath selected by the user. This feature makes easy to the students understanding in detail how the MR works.

## 7 Conclusions

This work has presented an overview of the architecture and design of the pedagogic computer named *Máquina Rudimentaria*. The MR is a useful tool to introduce the basic principles about computer structure and architecture in a first course in an Engineering or Computer Science degree. Its simple design allows a natural and fluent connection from the study of digital circuits to assembly language and computer architecture. The existence of a programming environment and a detailed simulator eases the compression of its operation (available at `ftp://ftp.ac.upc.es/pub/archives/mr`).

## References

[1] F. Tirado et al. *Fundamentos de Computadores*. Ed. Síntesis, 1998.

[2] E. F. Sowell. *Programming in Assembly Language VAX-11*. Addison-Wesley, 1987.

[3] C. L. Morgan and M. Waite. *Introducción al Microprocesador 8086/8088*. McGraw-Hill, 1984.

[4] J. L. Hennessy and D. A. Patterson. *Computer architecture. A Quantitative Approach, 2nd ed.*. Morgan Kaufmann Publishers. San Mateo, Calif., 1996.

[5] R. S. Sandige. *Modern Digital Design*. McGraw-Hill, 1990.