Last changed: 25/July/2003 by Ney Calazans

R8 PROCESSOR – ARCHITECTURE AND ORGANIZATION SPECIFICATION AND DESIGN GUIDELINES

Fernando G. Moraes & Ney L. V. Calazans

1 HISTORY

The R8 processor research effort is an initiative of the Hardware Design Support Group (GAPH). GAPH is a research group at the Computer Science Graduate Program (PPGCC), in the Faculty of Computer Science (FACIN), part of the Pontifical Catholic University of Rio Grande do Sul (PUCRS), located in Porto Alegre, capital of Rio Grande do Sul state, in Brazil.

In the start of 1997, a major revision of the then 13-year old Computer Science undergraduate curriculum started to be applied. The new curriculum significantly enhanced the hardware contents taught with regard to the previous one. Also, the hardware courses track in the curriculum was designed to expose students very early to hardware design using mostly professional tools. The reasoning to do so is exposed in detail in [1]. The basic idea is to propose a processor implementation as practical work to be developed along several semesters. First, the students have to produce a processor description from a detailed specification, implementing it in a hardware description language (currently, VHDL). Validation of their design is achieved by functional simulation. This happens in the third semester of the curriculum. A following course requires the students to start from their previous semester design and transform their implementation into a pipelined processor with the same architecture (register bank, instruction set, etc.). In a subsequent course, input-output capabilities and memory hierarchy considerations are added to the same design.

Every semester, a different processor specification is prepared. Specifications differ from one another in some aspects, the main ones being the requirement of unified memory or Harvard organization and obviously instruction set. However, all processors share common characteristics of being load-store, 16-bit RISC-like architectures, with a bank of 16 16-bit registers, and the instruction word length being fixed at 16 bits. These characteristics define the *Rx clan*. The first of these specifications was the R1 processor, proposed to the first class of students taking the Computer Organization course of the new curriculum, in the first semester of 1998. The R8 has been the eighth of these processors, and the R12 specification is currently under preparation.

Each new specification gives rise in fact to several processor implementations, and variations on the specification define a family of processors. The *R8 family* occupies a special position in the Rx clan, since it has been used in various research works outside the scope of undergraduate courses. Today, hardware implementations of the R8 basic architecture are available, and various implementations are validated at several abstraction levels. Up to four instances of the R8 processor have been implemented on FPGA-based prototyping boards, providing a multiprocessing platform to the research work on system-on-chip (SoC) and network-on-chip (NoC) subject areas.

This document is based on the original R8 specification handled to the students and is intended as the basis for formalizing the R8 family, providing a “gold” specification for its very first member, named simply R8. The definition of design methods and tools to support the R8 architecture must rely upon this specification, including the assembler and simulator tools, the C compiler and the embedded operating system kernel for the R8 processor. Also, it is expected that from this other specifications may branch to define other members of this processor family.

The rest of this document is organized as follows. Sections 2 to 4 present the basic Instruction Set Architecture (ISA) of the R8 processor. Next, Sections 5 to 10 detail a possible organization for the implementation of the R8 processor ISA. Section 11 give some hints on how to use the R8 processor software development environment available today, while Section 12 provide references to other documents.

2 GENERAL CHARACTERISTICS

- Load-store architecture: the logic and arithmetic instructions are executed among internal registers only, while the memory access instructions execute either the reading from (load) or the writing to (store) one memory position.
- Register bank: due to the load/store architecture option, the processor must have a relatively large set of data manipulation general-purpose registers, to reduce the number of memory accesses (this always represents a time penalty with regard to the processor internal operation). This characteristic differs from an accumulator based architecture, which keeps all data in memory, performing logic and arithmetic instructions among contents in memory and contents of one or a few special registers, named *accumulators*. Consider, for example the following C language line: **`for(i=0; i<1000; i++)`**. Here, in case '**`i`**' is stored in memory, 2,000 memory access operations are needed to execute this line, performing read and write operations at each iteration. In case '**`i`**' is first read and stored in a register, just register operations are needed, without using memory most of the time during the **`for`** command execution!
- Regular format for instructions: all instructions have exactly the same size, occupying 1 memory word each. The instruction contains the operation code and the operands specification, in case they exist.
- Few addressing modes.

Thus, the R8 processor is a RISC-like machine, but still missing some characteristics so common in most RISC processors, such as *pipelines*. The main specific organizational characteristics of this multi-cycle processor are:

- Address and data size are basically of 16 bits.
- Memory addressing is performed on a word basis (i.e. each memory address corresponds to the identifier of a position containing 16 bits of information).
- The register bank contains 16 general-purpose registers, each of 16 bits.
- There are 4 status flags named: negative, zero, carry, and overflow.
- The instruction execution takes place in 2 to 4 clock cycles, i.e. the average clock per instruction (CPI) for any program executed is always a number between 2 and 4.

3 INSTRUCTION SET

The very simple instruction set defines the following instruction classes:

- Logic and arithmetic binary instructions (2-operand): addition, subtract, AND, OR, Exclusive-OR.
- Logic and arithmetic binary instructions with short constants: addition and, subtraction.
- Unary logic and arithmetic instructions: left and right shifts and bitwise inversion.
- Half-word register loads with a constant: loads for higher half and for lower half.
- Stack pointer register initialization and return from subroutines.
- Null instruction: instruction that does nothing (useful for dimensioning waiting loops and specific memory space reservation).
- Halt instruction: to suspend the instruction execution flow.
- Load instruction: to read a memory word content to some general-purpose register.
- Store instruction: to store a 16-bit content in some general-purpose register to a place in memory.
- Jumps and subroutine calls with *relative* addressing, with short or long displacements and with absolute register addressing.
- Stack insertion and removal instructions: to manipulate values stores in the top of the stack maintained by the processor.

Table 1 - R8 processor instruction set architecture.

INSTRUCTION	INSTRUCTION FORMAT				
	15 – 12	11 - 8	7 - 4	3 - 0	SEMANTICS
ADD Rt, Rs1,Rs2	**0**	R target	R source1	R source2	Rt ← Rs1 + Rs2 ; Inz ; Icv
SUB Rt, Rs1,Rs2	**1**	R target	R source1	R source2	Rt ← Rs1 - Rs2 ; Inz ; Icv
AND Rt, Rs1,Rs2	**2**	R target	R source1	R source2	Rt ← Rs1 *and* Rs2 ; Inz
OR Rt, Rs1,Rs2	**3**	R target	R source1	R source2	Rt ← Rs1 *or* Rs2 ; Inz
XOR Rt, Rs1,Rs2	**4**	R target	R source1	R source2	Rt ← Rs1 *xor* Rs2 ; Inz
ADDI Rt, *cte8*	**5**	R target	Constant		Rt ← Rt + ("00000000" & constant) ; Inz ; Icv
SUBI Rt, *cte8*	**6**	R target	Constant		Rt ← Rt - ("00000000" & constant) ; Inz ; Icv
LDL Rt, *cte8*	**7**	R target	Constant		Rt ← RtH & constant
LDH Rt, *cte8*	**8**	R target	Constant		Rt ← constant & RtL
LD Rt, Rs1,Rs2	**9**	R target	R source1	R source2	Rt ← MEMP (Rs1+Rs2)
ST Rt, Rs1,Rs2	**A**	R target	R source1	R source2	MEMP (Rs1+Rs2) ← Rt
SL0 Rt, Rs1	**B**	R target	R source1	**0**	Rt[15:0] ← Rs1[14:0] & 0 ; Inz
SL1 Rt, Rs1	**B**	R target	R source1	**1**	Rt[15:0] ← Rs1[14:0] & 1 ; Inz
SR0 Rt, Rs1	**B**	R target	R source1	**2**	Rt[15:0] ← 0 & Rs1 [15:1]; Inz
SR1 Rt, Rs1	**B**	R target	R source1	**3**	Rt[15:0] ← 1 & Rs1 [15:1]; Inz
NOT Rt, Rs1	**B**	R target	R source1	**4**	Rt ← not (Rs1) ; Inz
NOP	**B**	0	0	**5**	No action
HALT	**B**	0	0	**6**	Suspends the instruction fetch and execution flow
LDSP Rs1	**B**	0	R source1	**7**	SP ← Rs1 (initializes the stack pointer)
RTS	**B**	0	0	**8**	PC ←MEMP(SP+1) ; SP←SP+1
POP Rt	**B**	R target	0	**9**	Rt ←MEMP(SP+1) ; SP←SP+1
PUSH Rt	**B**	R target	0	**A**	MEMP(SP)←Rt; SP←SP-1
JMPR Rs1	**C**	0	R source1	**0**	PC ← PC + Rs1 (does not depend upon any flags)
JMPNR Rs1	**C**	0	R source1	**1**	if (*n* =1) PC ← PC + Rs1
JMPZR Rs1	**C**	0	R source1	**2**	if (*z* =1) PC ← PC + Rs1
JMPCR Rs1	**C**	0	R source1	**3**	if (*c* =1) PC ← PC + Rs1
JMPVR Rs1	**C**	0	R source1	**4**	if (*v* =1) PC ← PC + Rs1
JMP Rs1	**C**	0	R source1	**5**	PC ← Rs1 (does not depend upon any flags)
JMPN Rs1	**C**	0	R source1	**6**	if (*n* =1) PC ← Rs1
JMPZ Rs1	**C**	0	R source1	**7**	if (z =1) PC ← Rs1
JMPC Rs1	**C**	0	R source1	**8**	if (*c* =1) PC ← Rs1
JMPV Rs1	**C**	0	R source1	**9**	if (*v* =1) PC ← Rs1
JSRR Rs1	**C**	0	R source1	**A**	MEMP(SP)←PC; SP←SP-1; PC← PC+ Rs1
JSR Rs1	**C**	0	R source1	**B**	MEMP(SP)←PC; SP←SP-1; PC← Rs1
JMPD displacement	**D**	**0**	Displacement (10 bits)		PC ← PC + **signal_ext &** displacement
JMPND displacement	**E**	**0**	Displacement (10 bits)		if (*n* =1) PC ← PC + **signal_ext &** displacement
JMPZD displacement	**E**	**1**	Displacement (10 bits)		if (z=1) PC ← PC + **signal_ext &** displacement
JMPCD displacement	**E**	**2**	Displacement (10 bits)		if (*c* =1) PC ← PC + **signal_ext &** displacement
JMPVD displacement	**E**	**3**	Displacement (10 bits)		if (*v* =1) PC ← PC + **signal_ext &** displacement
JSRD displacement	**F**	Displacement (12 bits)			MEMP(SP)←PC; SP←SP-1; PC← PC + **signal_ext &** displacement

- The following conventions have been adopted in the above Table:

  RtH: **most significant eight bits of Rt**
  RtL: **least significant eight bits of Rt**
  &: **bit vectors concatenation**
  ←: **register assignment of a value or of contents of a memory position**

MEMP(x): **memory position contents at the address x**
Rt : **Rtarget [destiny]**
Rs1 : **Rsource1**
Rs2 : **Rsource2**
Inz: **activate the storage of the values of the negative and zero status flags**
Icv: **activate the storage of the values of the carry and overflow status flags**

4 ARCHITECTURE REGISTERS

The processor contains the following control and data storage registers:

- IR (*instruction register*): a 16-bit register that stores the present instruction code (*opcode*) and their specific operands.
- PC (*program counter*): a 16-bit register that contains always the memory address of the next instruction to be executed.
- SP (*stack pointer*): a 16-bit register that stores the top of stack address, controlling the calling and return from subroutines. The SP must be initialized by each program intending to use subroutines with the LDSP instruction, which loads a value in SP that represents the primary top of the stack for the program.
- R0 to R15 (*general-purpose registers*): 16 general-purpose data manipulation registers. The register bank formed by these registers has a write port and two read ports. This means that it is possible to write in some register while at the same time two data are read from other registers in the bank. The outputs of the register bank are the output busses *SOURCE1* (S1) and *SOURCE2* (S2).
- N, Z, C, V (*status flags*): 4 status *bits* controlling *signal*, *zero*, *carry* and *overflow* conditions of logic, arithmetic and memory instructions. These flags are used to store information to be tested by conditional jump instructions and conditional calls to subroutines. The flags state is determined by logic, arithmetic and memory read operations (loads).

Other registers may be necessary to implement the organization. Examples appear in Section 6 of this document, where a suggestion of organization is advanced.

5 RELATIONSHIP BETWEEN PROCESSOR AND EXTERNAL MEMORY

Figure 1a illustrates the relationship between the R8 processor and its external memory. The processor receives two signals from the external world: *clock*, intended to synchronize the internal events of the processor hardware; and *reset*, which puts the processor in a starting state, able to begin instruction execution by fetching instructions from the first address of the external memory, namely address **0000h**.

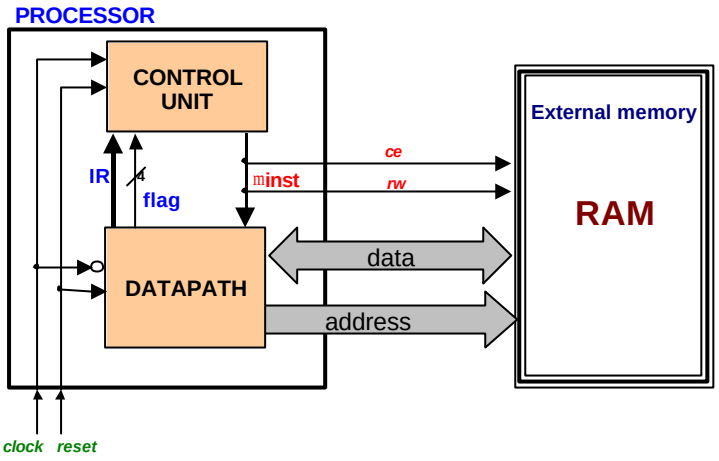
The R8 control unit generates microinstructions (µinst) to execute instructions in a clock cycle by clock cycle basis. Each microinstruction is responsible for commanding the actions executed by the datapath, such as register access selection, ALU operation and/or external memory access to fetch data or instructions.

After an instruction fetch, the datapath informs to the control unit the code of the current instruction under execution (the content of the ***IR*** register) and the values of the status flags (***flags***). The datapath is also responsible for the communication with the external memory. The signals employed for the exchange of information between the R8 processor and the external memory are: *data* (a 16-bit bidirectional bus transporting data and instructions) and *address* (a 16-bit bidirectional bus containing a memory addresses to access).

The memory access control is performed by the control unit through signal *ce* and *rw*. The signal *ce* indicates, when its value is 1, that an operation of information transfer is occurring between the processor and the memory, while signal *rw* indicates if this operation is a reading (*rw=1*) or writing (*rw=0*) operation.

It is important to note that the datapath and the control unit work at distinct edges of the clock signal. In one clock edge (e.g., the rising edge) the control unit generates a microinstruction, and in the next edge (in this case at the falling edge) the datapath executes the microinstruction, by modifying its registers. With this schema the control and data information will be sampled when they are effectively stable, for a sufficiently low clock frequency.

Figure 1b represents the first hierarchical level of the R8 processor, by means of the hardware description language. In this Figure the processor blocks are connected by *signals*, and are instantiated by means of *port map* commands.



PROCESSOR

```
entity processor is
      port( ck,rst: in std_logic;
            data: inout reg16;
            address: out reg16;
            ce,rw: out std_logic );
end processor;

architecture processor of processor is
      signal flag: reg4;
      signal uins: microinstruction;
      signal ir: reg16;
   begin

      dp: datapath
          port map( uins=>uins, ck=>ck, rst=>rst,
                    instruction=>ir,
                    address=>address, data=>data,
                    flag=>flag);


      ctrl: control_unit
            port map(uins=>uins, ck=>ck, rst=>rst,
                     flag=>flag, ir=>ir);
      ce <= uins.ce;
      rw <= uins.rw;
end processor;
```

(a) Block diagram of the processor-memory interface (b) VHDL description of the R8 processor first hierarchical level

Figure 1 – Relationship between the R8 processor and the eternal memory.

6 INSTRUCTION EXECUTION IN THE DATAPATH

The instruction execution in the R8 processor requires from 2 to 4 clock cycles. The distinct machine cycles are defined as follows:

- **Cycle 1 : instruction fetch.** Common to all instructions.
- **Cycle 2 : decoding and operand fetch.** Common to all instructions.
- **Cycle 3 : ALU operation.** Common to all instructions, except for **HALT** and **NOP**.
- **Cycle 4 : instruction execution.** Depends on the specific type of operation, may not exist.

6.1 Instruction Fetch Cycle

In this cycle occurs the reading of the instruction code pointed by the ***PC*** register in the external memory, storing the code in the ***IR*** register and incrementing the ***PC*** to point to the next instruction. In microprogramming pseudo code this is represented in a micro-assembler notation as **IR ← MEMP(PC); PC++;** Figure 2 illustrates the interconnection of hardware components necessary to the execution of the instruction fetch cycle.

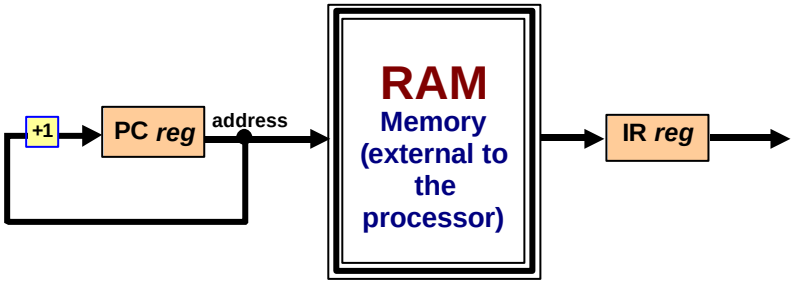


Figure 2 - Hardware needed to execute the instruction fetch.

6.2 Decoding and Operand Fetch Cycle

In the second clock cycle of each instruction the register source operands are read from the register bank, independently of whether the instruction employs these registers or not. These operands are called *source1 and*

source2. The source registers values are respectively stored in the temporary registers *RA* and *RB*. Bits 7 to 4 of the IR register address register source1.

Register *source2* can be addressed by bits 3 to 0 or by bits 11 to 8 of the IR register. When the instruction under execution involves the *target* register address as a source operand, *IR bits 11 to 8 address source2.* Example: the ADDI instruction, when an immediate constant is added to the contents of a given register and the results are stored back in the same source register. Figure 3 illustrates the hardware components for the reading of source registers from the register bank and their interconnections.

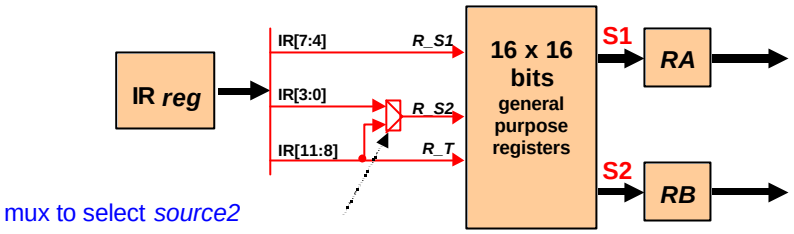


Figure 3 - Hardware to execute the operand fetch cycle. Instruction decoding is performed in parallel with instruction operand fetch by the control unit, which is not shown.

6.3 ALU Operation Cycle

- The ALU operation cycle is also common to all instructions, except for the *halt* instruction. Given the variety of instructions, it is necessary to insert multiplexers in the ALU inputs to correctly select operands.
- The ALU operation result is always store in the *RALU* register and, depending on the executed operation the flags values are stored in flip-flops (N, Z, C, V).
- Figure 4 illustrates the hardware components interconnection for the ALU operation.

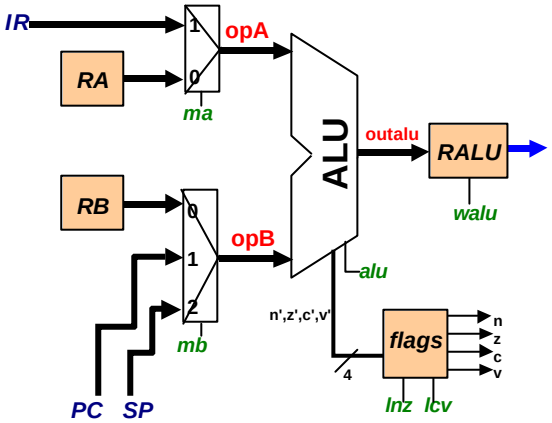


Figure 4 - Hardware to execute the ALU operation cycle.

- Table 2 below defines the value that ALU inputs *opA and opB* must assume, according to each instruction.

Table 2 – Alu operand input specification for each instruction of the R8 processor.

Instructions	OpA	OpB
ADD, SUB, AND, OR, XOR, LD, ST, SHIFTS, NOT, LDSP	RA	RB
ADDI, SUBI, LDL, LDH	IR	RB
RTS, POP	-	SP
PC relative jumps and subroutine calls	RA	PC
Absolute jumps and subroutine calls	RA	-
Short displacement jumps and subroutine calls	IR	PC

6.4 Instruction Execution Cycle

6.4.1 Arithmetic, logic and immediate addressing instruction execution

- The fourth clock cycle of arithmetic and logic instructions stores the ALU result register *RALU* in the register bank according to the destination register address. This cycle is called *write-back*.
- Figure 5 illustrates the hardware structure necessary for the execution of the fourth cycle for all logic and arithmetic instructions.

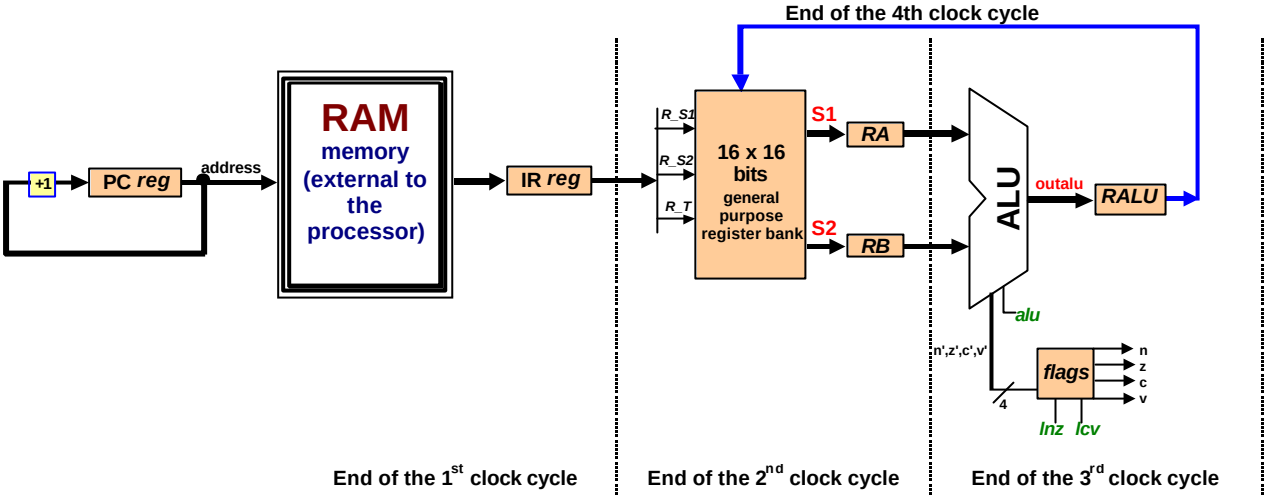


Figure 5 – Datapath section and memory to illustrate the execution flow for logic, arithmetic LDL and LDH instructions.

- The immediate addressing mode instructions imply that a given destination register (target) receives the result of a given operation between the destination register itself and an 8-bit constant. The immediate addressing mode instructions are:
  - Load the upper byte of a register (LDH): *Rt* ← *constant & RtL* (the target register receives a constant byte in the upper byte, maintaining the lower byte unchanged).
  - Load the lower byte of a register (LDL): *Rt* ← *RtH & constant* (the target register receives a constant byte in the lower byte, maintaining the upper byte unchanged).
  - Addition/subtraction with the immediate mode: addition/subtraction of the contents of a given register with an 8-bit constant: *Rt* ← *Rt +/- constant.* **Important**: the instruction execution implies completing with 0s the 8 more significant bits of the constant to generate a 16-bit register.
- **Important**: in order to load a general-purpose register with a 16-bit constant, two instructions LDH and LDL must be used in sequence. To read or write some data stored in a given 16-bit memory address 3 assembly language instructions are necessary: the first 2 to load the high and low parts of the address into a register (LDH and LDL, respectively) and a third instruction to execute the reading from or writing to memory (LD or ST). An example is given below:

```
XOR   R0,R0,R0      ; loads R0 with the constant 0000H
LDH   R1, #03H
LDL   R1, #27H      ; store the value 0327H in register R1
LD    R5, R1, R0    ; stores into R5 the contents of the memory address given by (R1+R0)
```

6.4.2 Memory read instruction execution (LD)

- The semantics of this instruction is as follows. The *destination* register receives the contents of the memory address given by the result of the sum of the two source registers (*sources*): *Rt* ← *MEMP(Rs1 + Rs2)*. One of the sources can be used as a base register, while the other is used as a displacement from that base address (*offset*).
- In the fourth clock cycle, the *RALU* register addresses the memory and the data read from memory is directly stored in the register bank in register addressed by the bits 11 to 8 of register IR (IR[11:8]).

- A possible datapath organization to allow the execution of this instruction is presented in Figure 6.

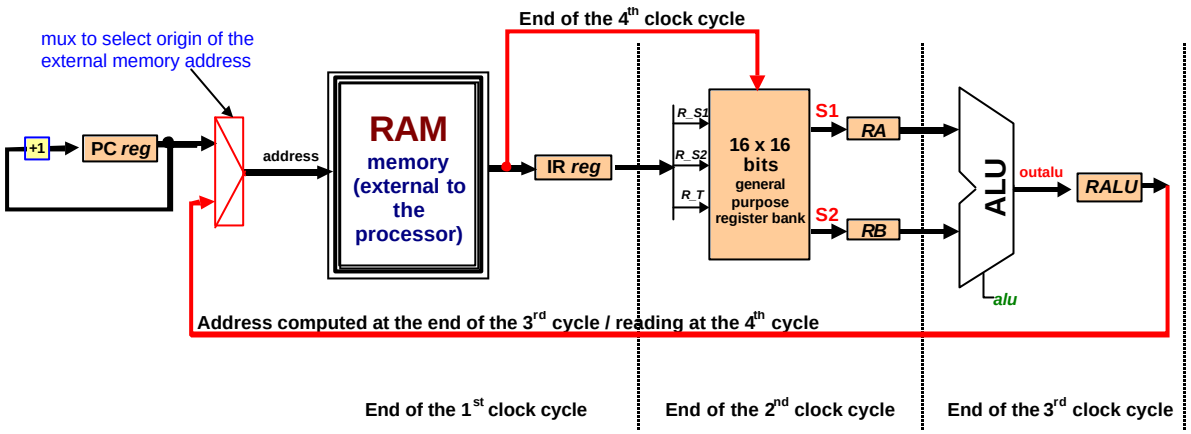


Figure 6 – Datapath section and memory to illustrate the LD instruction execution.

6.4.3 Memory write instruction execution (ST)

- The memory position addressed by the sum of the contents of two source registers (*sources*) receives the contents of the destination register (*target*): *MEMP(Rs1 + Rs2)* ← *Rt*.
- In the fourth clock cycle, the register addressed by IR[11:8] is read, writing the contents of this register in the address defined by the contents of the RALU register.
- A possible datapath organization to execute the memory store instruction is presented in Figure 7.

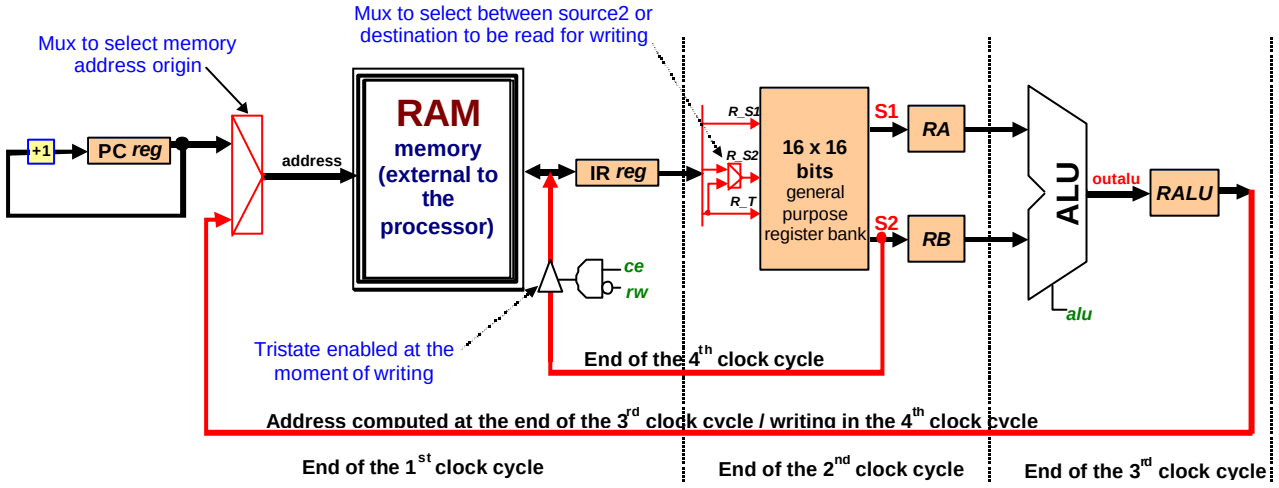


Figure 7 - Datapath section and memory illustrating the hardware needed to execute the ST instruction execution.

6.4.4 Conditional and unconditional jump instructions

- The destination address is computed in the third clock cycle of these instructions, being this information stored in the RALU register.
- If the jump is conditional, and the corresponding *flag* bit is clear (*flag*=0), the instruction ends in the third clock cycle.
- If a conditional jump is executed, the PC register must receive the contents of the *RALU* register.
- A possible datapath organization to execute conditional and unconditional jumps is presented in Figure 8.

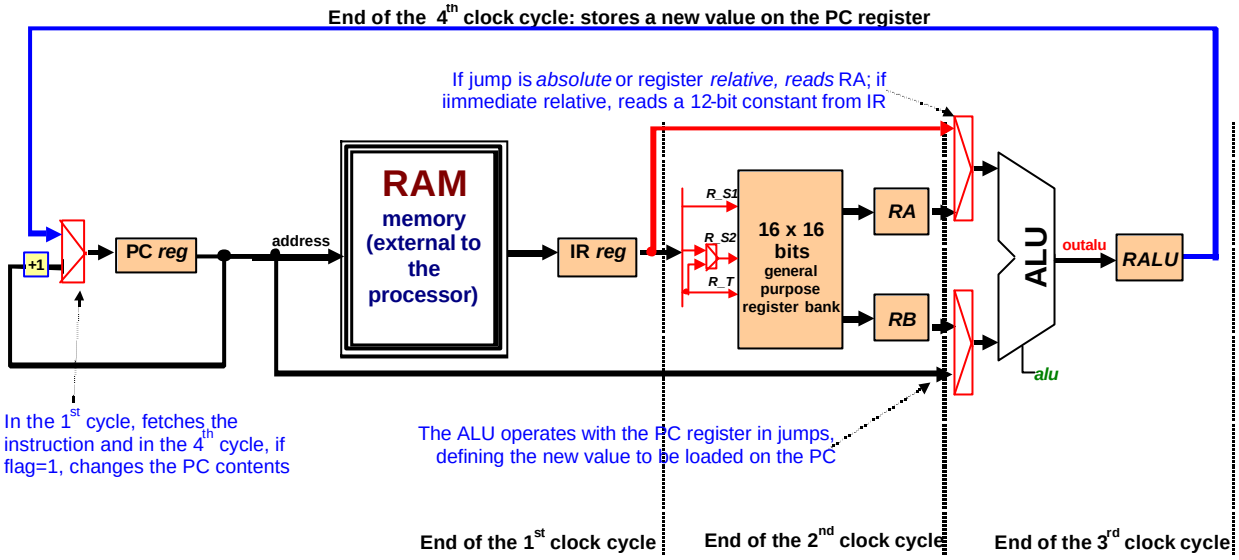
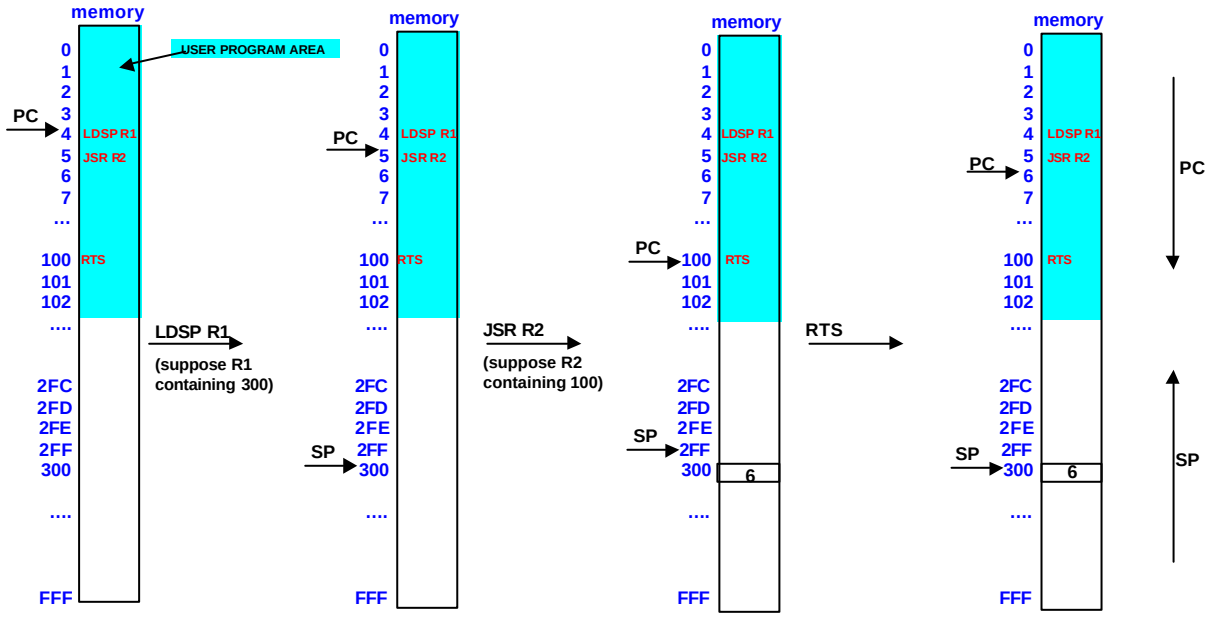


Figure 8 - Datapath section and memory to illustrate the hardware needed for the execution of unconditional and conditional jumps.

6.4.5 Stack working and subroutine call instructions

The instructions that implicitly or explicitly manipulate the SP register (*stack pointer*) are: register contents insertion/removal on the stack (PUSH/POP), subroutine calls (*JSR*, *JSRR*, JSR), SP initialization (*LDSP*) and subroutine return (*RTS*). Figure 9 illustrates how the stack works.



- The program is stored from address *0* to address *N*. Thus, the program addresses grow with the memory addresses.
- The stack grows in a sense inverse of that of the program memory. The reason for this is to avoid interference with the program data area.
- The contents of the SP register are always interpreted as the address of the first free stack position for stacking new data.

Figure 9 – Illustrating the functioning of stack operations.

A subroutine call, also called jump to subroutine, is executed as follows:

- The destination jump address is computed in the third clock cycle, being stored in the *RALU* register.

- If the jump to subroutine is conditional and the respective *flag* is cleared (*flag*=0), the instruction is ended in the third clock cycle.
- If the jump to subroutine must be executed, the PC register must receive the contents of the *RALU* register, at the same time storing the previous PC contents on the top of the stack. Remember that the PC has been incremented in the fetch clock cycle (the first of the instruction), and this is the value stacked. After storing this value on the stack, the SP register is decremented, i.e.:

MEMP(SP)← PC;
SP← SP-1;
PC← *result at the RALU register (PC+offset* or *RS1* or *PC+RS1*)

- A possible datapath organization to execute subroutine calls is presented in Figure 10.

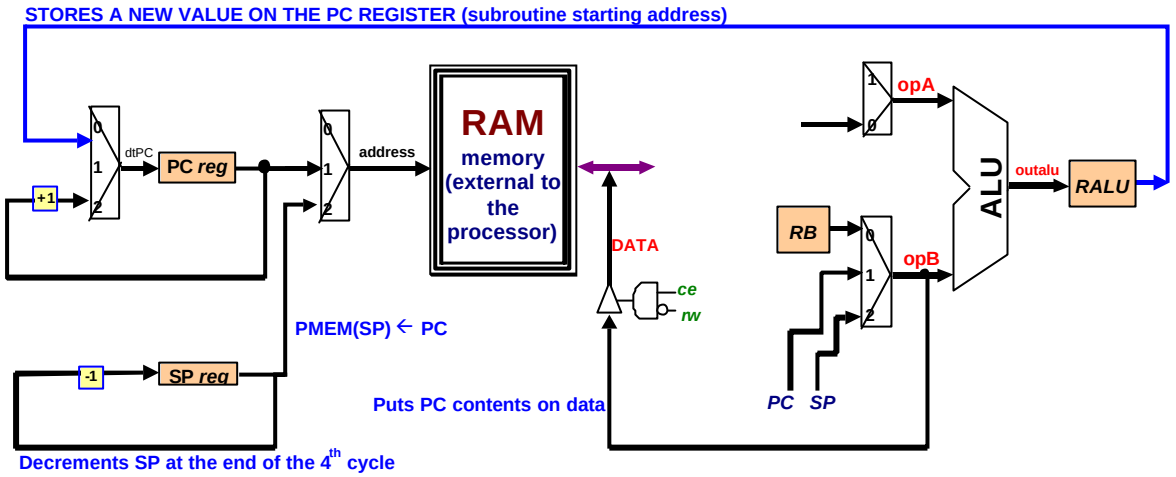


Figure 10 - Datapath section and memory to illustrate the execution of subroutine call instructions.

- The execution of the PUSH instruction is similar to the subroutine call instructions:

MEMP(SP)← Rt; (stored in RB)
SP← SP-1;

6.4.6 Return from subroutine and pop instructions (top of stack data retrieval)

- The fourth clock cycle of instructions RTS and POP address the data memory with the contents of the *RALU* register (with SP+1), storing the result of the memory reading either in the PC register (RTS) or in the register bank destination register (POP). The SP register is updated (SP ←SP+1).

7 DATAPATH ORGANIZATION

Aggregating the different previous figures, it is possible to obtain a complete R8 datapath that allows to implement all instructions defined for this architecture. This is depicted in

Figure 11. Some additional elements were added for controlling subroutines cal and return, due to the need of manipulating the SP register. The datapath sends to the control unit the output of the IR register as well as the contents of the status flags.

The datapath needs to receive ***18*** control signals from the control unit, organized into 4 signal classes:

- Register write enable signals (8): ***wPC***, ***wSP***, ***wIR***, ***wAB***, ***wALU***, ***wFlag***, ***wnz***, ***wcv***.
- External memory access controls (2): ***ce*** e ***rw***.
- Multiplexers control signals (7): ***mpc*** (PC register data origin), ***msp*** (SP register data origin), ***mad*** (selects which register addresses the external memory), ***mreg*** (register bank data origin), ***ms2*** (selects which portion of the IR register selects the second ALU operand), ***ma*** (ALU first operand data origin), ***mb*** (ALU second operand data origin).
- The operation executed by the arithmetic and logic unit (1): ***alu***.

Figure 12 illustrates the register bank organization, as a block diagram. It should be observed that the multiplexer responsible for the selection of the second source register (S2) is inside the register bank block.

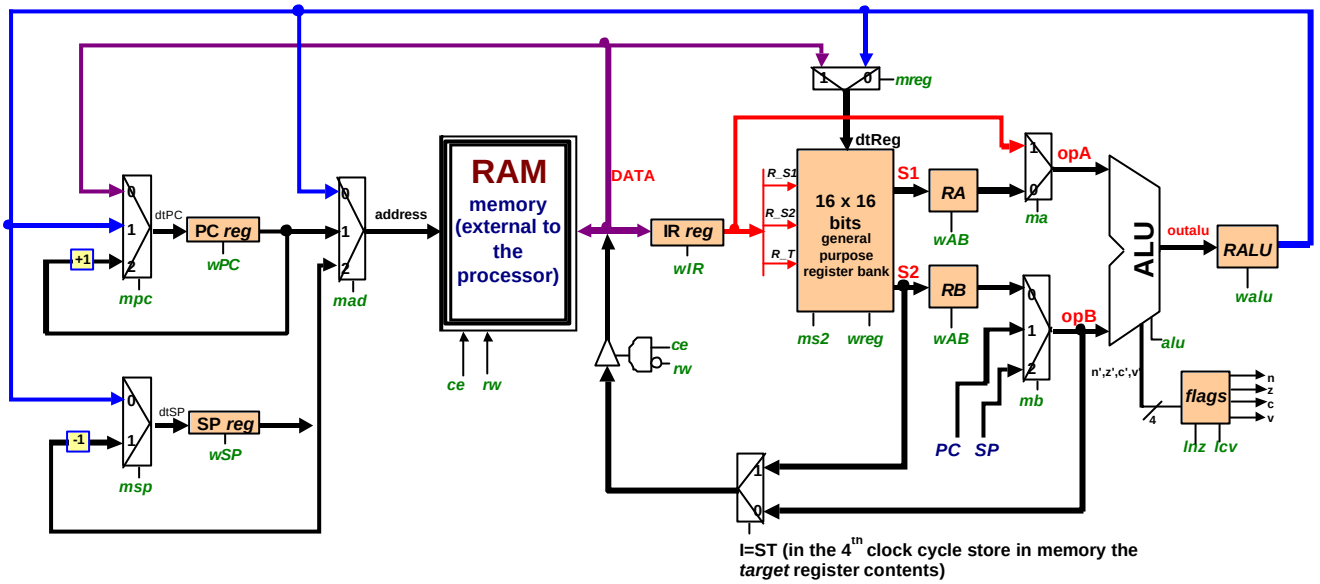


Figure 11 – Complete datapath organization (plus the external memory). Here are represented all 18 control signals that the control unit must generate and manage (in green, italics font). Clock and Reset signals are not represented, but are obviously necessary to command all registers.

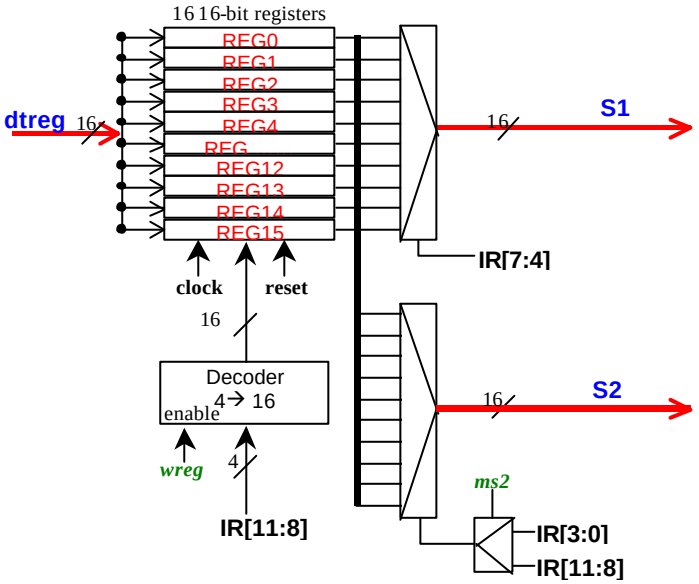


Figure 12 – Block diagram of the general-purpose register bank.

8 CLOCK CYCLES NEEDED TO EXECUTE INSTRUCTIONS

Given the R8 processor datapath organization description presented in the previous Section, it is possible to summarize the execution time for all instruction in terms of number of clock cycles to execute each instruction. This is shown in Table 1 below.

Table 1 – Number of clock cycles taken to fetch and execute each instruction in the organization of the R8 processor.

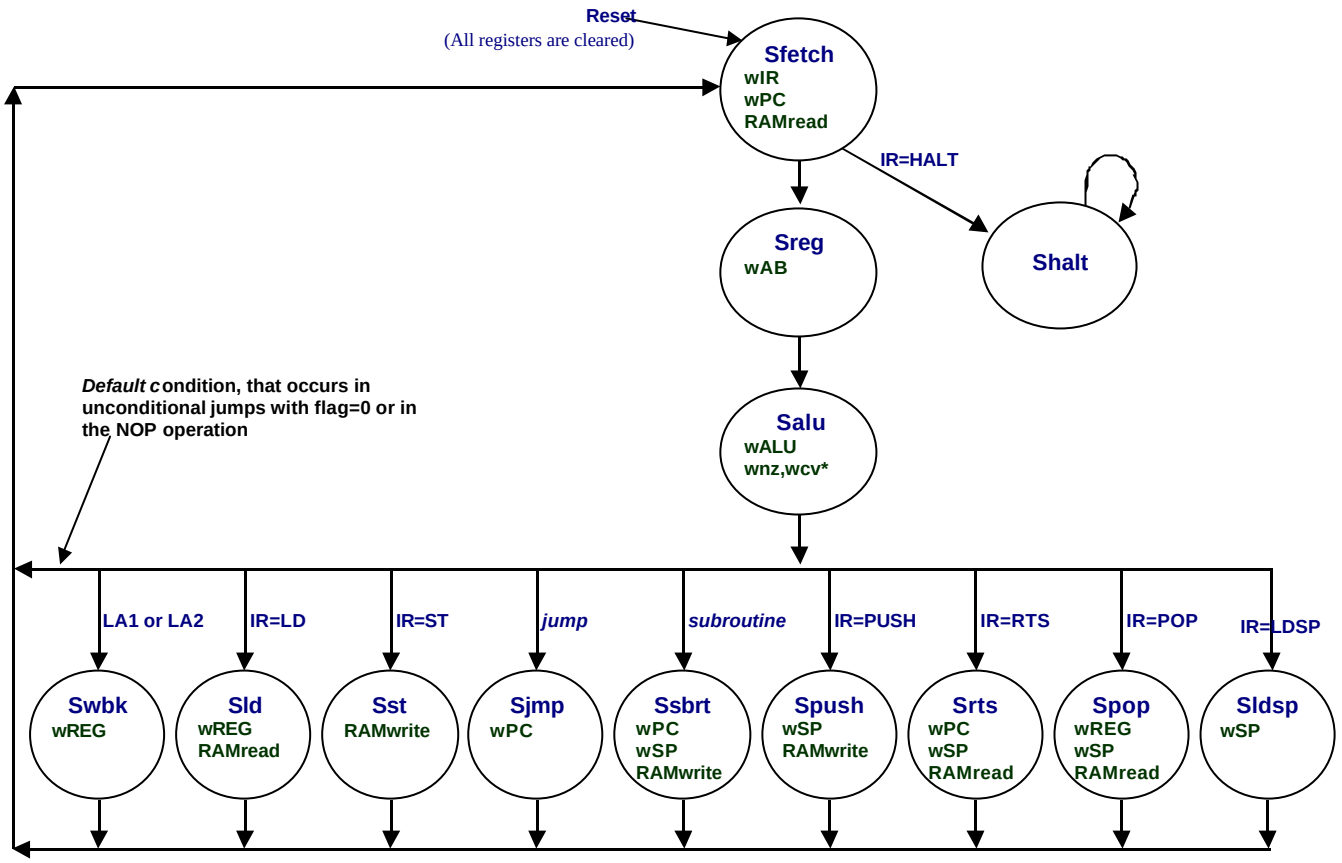
INSTRUCTION	# OF CLOCK CYCLES	INSTRUCTION	# OF CLOCK CYCLES
ADD	4	POP	4
SUB	4	PUSH	4
AND	4	JMPR	4
OR	4	JMPNR	3/4
XOR	4	JMPZR	3/4
ADDI	4	JMPCR	3/4
SUBI	4	JMPVR	3/4
LDL	4	JMP	4
LDH	4	JMPN	3/4
LD	4	JMPZ	3/4
ST	4	JMPC	3/4
SL0	4	JMPV	3/4
SL1	4	JSRR	4
SR0	4	JSR	4
SR1	4	JMPD	4
NOT	4	JMPND	3/4
NOP	3	JMPZD	3/4
HALT	2+	JMPCD	3/4
LDSP	4	JMPVD	3/4
RTS	4	JSRD	4

9 CONTROL UNIT

In order to execute any of the R8 processor instructions it is necessary to define a finite state machine to control the execution in the datapath. Figure 13 illustrates such an FSM, where at each clock the next state is a function of the preset state and of the instruction stored in register IR. In the same Figure is indicated each of the registers that are expected to receive new information at each state. Memory accesses are equally indicated (through the use of *RAMread* and *RAMwrite* commands).

The functions assigned to the 13 states are:

- Sfetch: first clock cycle of any instruction, instruction fetch;
- Srreg: second clock cycle of any instruction, except HALT, source registers reading and instruction decoding;
- Shalt: second clock cycle of instruction HALT, stops fetch–execute cycles and awaits for reset;
- Salu: third clock cycle, ALU operation;
- Swbk: fourth clock cycle, store ALU result back in the destination register;
- Sld: fourth clock cycle, read data from memory and store it in the destination register;
- Sst: fourth clock cycle, write destination register data contents to memory;
- Sjmp: fourth clock cycle, changes PC register contents in unconditional or conditional jumps, when flag=1;
- Ssbrt: fourth clock cycle , jump to subroutine;
- Spush: fourth clock cycle, puts source register contents in the top of the stack;
- Srts: fourth clock cycle, return from subroutine;
- Spop: fourth clock cycle, retrieves contents in the top of the stack to the destination register;
- Sldsp: fourth clock cycle, initializes the stack pointer register (top of the stack);



- LA1 - type 1 logic or arithmetic instruction – unary or binary instructions
- LA2 - type 2 logic or arithmetic instruction – instructions with a single source register
- *Jump*: execution of the 15 jump instructions
- *Jump to subroutine*: JSR, JSRR, JSRD
- * - writing of the flags in the Salu state depends on the specific instruction

Figure 13 – Control finite state machine for the R8 processor.

9.1 Instruction decoding / ALU operation definition

The first action performed by the R8 Control Unit after instruction fetch is instruction decoding.

- As an implementation suggestion, it is possible to create in the *package* of the R8 processor definitions an enumerated type containing all possible instruction mnemonics. Originally the R8 processor contains **40** instructions. However, it is possible to group some of these, e.g. the jump instructions, which may be grouped in three distinct classes: relative jumps (Rjump), absolute jumps (jump) and displacement jumps (Djump). Instruction mnemonic count can then be reduced to **28** instruction classes.

```
type instruction is
 (add,  sub,  and_i,   or_i,   xor_i,  addi,  subi,  ldl,   ldh,   ld,  st,  sl0, sl1, sr0,  sr1,
      not_i, nop, halt, ldsp, rts,  pop,   push,  Rjump, jump, Djump, jsrr,  jsr, jsrd);
```

- Instruction decoding. It is possible to observe the effect of the above instruction mnemonic grouping as 5 relative jumps are now referred by a single mnemonics “Rjump”, 5 absolute jumps become "jump" and the 5 jumps with short displacement become "Djump". This simplification tends to reduce the HDL code size to be written.

```
i <= add   when ir(15 downto 12)=0  else
     sub   when ir(15 downto 12)=1  else
     ……
     sr1   when ir(15 downto 12)=11 and ir(3 downto 0)=3  else
     ……
     Rjump when ir(15 downto 12)=12 and  ( ir(3 downto 0)=0 or
                 (ir(3 downto 0)=1 and fn='1') or (ir(3 downto 0)=2 and fz='1') or
                 (ir(3 downto 0)=3 and fc='1') or (ir(3 downto 0)=4 and fv='1') )   else
     jump  when ……
     Djump when …
     …
```

```
        jsr   when ir(15 downto 12)=12 and ir(3 downto 0)=11 else
        jsrd  when ir(15 downto 12)=15;

    uins.ula <= i;       --- ********** ALU operation
```

- Type 1 logic and arithmetic instruction decoding:

```
    inst_la1 <= '1' when i=add or i=sub or i=and_i or i=or_i or i=xor_i or i=not_i or i=sl0 or
                         i=sr0 or i=sl1 or i=sr1 else
                '0';
```

- Type 2 logic and arithmetic instruction decoding (Rt in the right size of micro instructions expressions):

```
    inst_la2 <= '1' when i=addi or i=subi or i=ldl or i=ldh else
                '0';
```

9.2 Multiplexers control

The multiplexers control signals (totalizing 7 signals) depend on the FSM present state and or in the current instruction code. It is recommended to locate the multiplexers and their control in

Figure 11. The control signals are preceded below by the "*uins.*" suffix that designates the microinstruction that passes from the control unit to the datapath.

1. PC data input origin control (depends on the FSM state):

```
    uins.mpc <= "10" when EA=Sfetch else    -- in the fetch increments PC
                "00" when EA=Srts  else     -- when returning from subroutine fetch new PC
                                            -- value in memory
                "01";                       -- by default load PC with ALU output
```

2. SP data input origin control (depends on the current instruction only):

```
    uins.msp <= '1' when  i=jsrr or i=jsr or i=jsrd or i=push else  -- post-decrement
                '0';
```

3. Memory addressing origin control (depends on the FSM state):

```
    uins.mad <= "10" when EA=Spush or  EA=Ssbrt else  -- in jump to subroutine, SP addresses
                                                      -- memory
                "01" when EA=Sfetch else               -- in the fetch, PC addresses memory
                "00";                                  -- by default: used in LD/ST
```

4. Register bank write data origin control (depends on the current instruction only):

```
    uins.mreg <= '1' when i=ld or I=pop else '0';     -- write in register bank element the contents
                                                      -- coming from memory
```

5. Choice of the second source operand address for register bank (depends on the instruction and on the FSM state). The second source operand address (source2) receives the address of the destination register when executing type 3 logic or arithmetic instructions or a memory write.

```
    uins.ms2 <= '1' when inst_la2='1' or i=push or EA=Sst else '0';
```

6. ALU operands origin (depends only upon the instruction code):

```
    -- first ALU operand is the IR during a type 2 logic or arithmetic instruction or jump/jsr with short displacement
     uins.ma <= '1' when inst_la2='1' or i=jumpD or i=jsrd else '0';


    -- second multiplexer
     uins.mb <= "01"  when  i=rts or i=pop   else      -- to increment the SP register
                "10"  when  i=jumpR or i=jump or i=jumpD or i=jsrr or i=jsr or i=jsrd  else
               "00" ;
```

In short, three main parts compose the control unit:

1) Instruction decoding.
2) Multiplexers control generation.
3) Control finite state machine, to generate control signals for register load enabling, reading from and writing to memory.

10 EXAMPLE SIMULATION OF AN INSTRUCTION SEQUENCE EXECUTION

The simulation of Figure 14 illustrates the execution of the 6 instructions in the program code below:

```
end     instruction
0128    7190
0129    8101;  R1 ← 0190          (400 in decimal)
012A    73AA
012B    83BB;  R3 ← BBAA
012C    AD01;  records the contents of register 14 in the address contained in register 1 (190H or 400)
012D    9F10;  reads the contents of the address contained in register 1, storing this in register 15
```

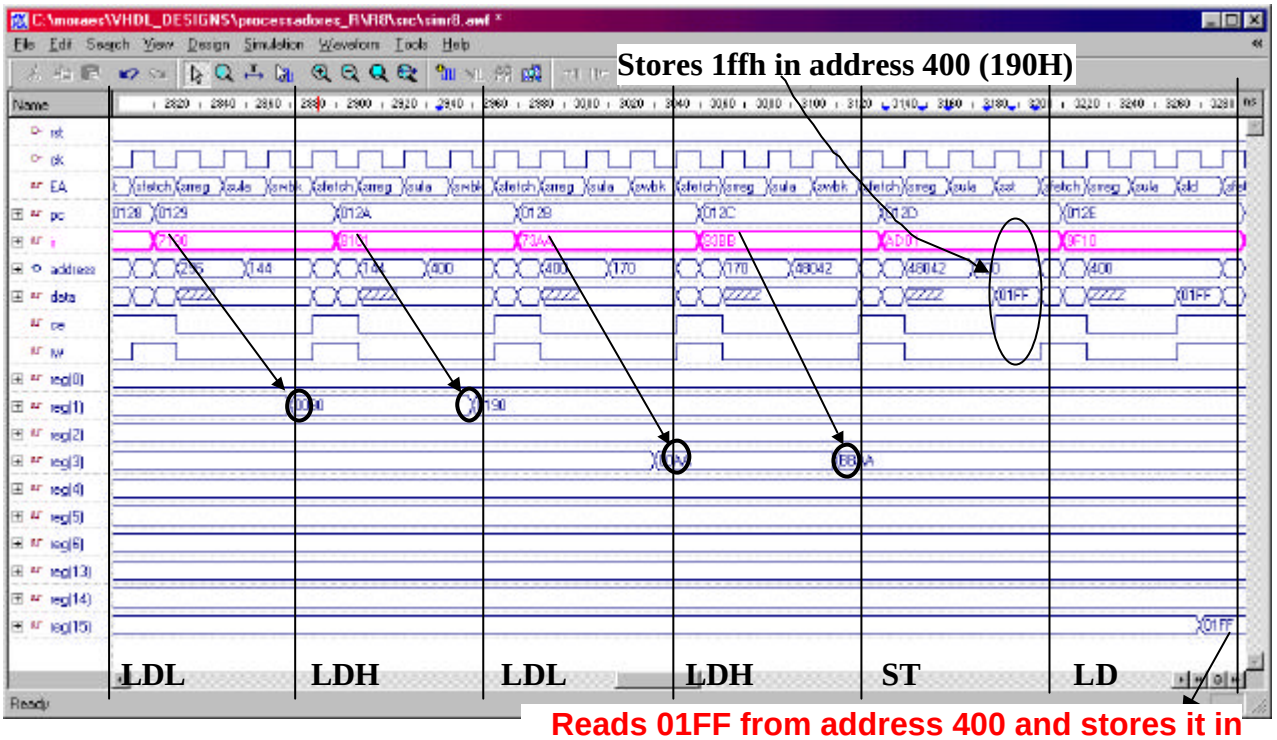


Figure 14 – A simulation of 6 instructions of the R8 pr ocessor.

11 EXAMPLE PROGRAM TO TEST ALL INSTRUCTIONS

The object code below corresponds to an example text file that is read by the VHDL simulation test bench during simulation of the R8 processor. It is associated with the corresponding assembler code in the comments part (after the ";" symbol). The file contains *n* lines, each with 9 characters in the "xxxx yyyy" format, where xxxx is a 16-bit address (4 hexadecimal digits) and yyyy is a 16-bit instruction (4 hexadecimal digits). This text file is loaded in memory by the test bench during reset activation, typically at the start of the processor simulation.

```
0000 7108    ;
0001 8110    ;  LOAD R1, #1008
0002 7234    ;
0003 8212    ;  LOAD R2, #1234
0004 73DC    ;
0005 83FE    ;  LOAD R3, #FEDC
0006 0412    ; sum:     result in R4: 223C
0007 1512    ; subtrai: result in R5: FDD4
0008 2612    ; and:     result in R6: 1000
0009 3712    ; or:      result in R7: 123C
000A 4812    ; xor:     result in R8: 023C
000B 5101    ; immediate sum 01 to R1  - 1009
000C 510F    ; immediate sum 0F to R1  - 1018
000D 51FF    ; immediate sum FF to R1  - 1117  (1130ns)
000E 6201    ; immediate subtract 01 from R2  - 1233
000F 6204    ; immediate subtract 04 from R2  - 122F
0010 62FF    ; immediate subtract FF from R2  - 1130
0011 7DFF
0012 8D01    ;  LOAD RD, #01FF
0013 B0D7    ;  load the stack pointer with the contents of register 14 (511 in decimal)  (1610ns)
0014 7100
0015 8101    ;  R1 <- 0100
0016 C01B    ; ********* jump to subroutine pointed by R1 (address 0100)  *********
0017 B005    ; nop
0018 70FF
0019 80FF    ; R0 <- FFFF      / sets the negative flag
001A 50FF    ; R0 <- R0 + FF   / sets the overflow flag
001B 4000    ; RO <- R0 xor RO / seta the zero flag
001C 7730    ;
001D 8700    ; R7 <- 0030
001E C077    ; jumps to the address pointed by R7 (30H) if zero flag set

0030 7710
0031 8700
0032 C070    ; inconditional relative jump to address 33H+10H=43H

0043 D050    ; jump to address 44H+50H=94H

0094 B006    ; ********** HALT HALT ******* occurs at 4800 ns simulation time ***************

0100 B10A    ; SUBROUTINE THAT TESTS PUSHING AND POPPING REGISTERS TO/FROM the STACK
0101 B20A
0102 B30A
0103 B40A    ; push registers 1 to 4
0104 7109
0105 8100
0106 C01A    ; when here, call another subroutine using PC-relative addressing mode (110H-107H=09H)
0107 B409
0108 B309
0109 B209
010A B109    ; retrieve registers 1 to 4 from the stack
010B B008    ;  ****** rts *****************      (4000ns simulation time)

0110 4111    ; SUBROUTINE THAT employs xor instruction to clear registers 1 to 4 - 2490ns
0111 4222
0112 4333
0113 4444    ; simulation time: 2730ns
0114 F013    ; subrotine PC-relative, jump over 13 words, going to address 0128
0115 B008    ;  ****** rts *****************

0128 7190    ; SUBROUTINE THAT TESTS LOAD AND STORE INSTRUCTIONS
0129 8101    ; r1 <- 0190      (400 in decimal)
012A 73AA
012B 83BB    ; R3 <- BBAA
012C AD01    ; stores the contents of register 14 at the addresss contained in register 1 (190H or 400)
012D 9F10    ; reads contents of address contained in register 1, storing it in register 15 (3250ns)
012E B230
012F B220
0130 B221
0131 B221    ; tests sl0 and sl1 instructions -  R2 after this must contain BAA3
0132 B422
0133 B442
0134 B443
0135 B443    ; tests sr0 and sr1 instructions - R4 after this must contain CBAA - simulation time: 4000ns
0136 B104    ; not     - R1 after this must contain FFFF
0137 B008    ;  ****** rts *****************
```

It is recommended that R8 users **write assembly code programs for their application**, generating automatically the object code using the assembler/simulator combination. The assembler/simulation tool, as well as some more detailed documentation on how to use it is available together with the processor specification and implementation files. Figure 15 shows the Graphical User Interface (GUI) main window of the R8 assembler/simulator. To the left of the Figure appears the Memory Table, containing in each line one assembly instruction, the memory address chosen by the assembler where to locate this instruction and the associated object code actually store at this position. In the center of the Figure is located the Symbol Table for the source code program. For each label appearing in the program, regardless if this refers to data or program information, the associated memory address and the value of the label are associated in this table. To the right of Figure 15 are located the Register Table, which displays the contents of each general purpose register (from R0 to R15), and of each of the main control registers, IR, PC and SP. In the lower part of the window are located the control buttons *Step*, *Run*, *Pause*, *Stop* and *Reset*, as well as the simulation speed control buttons *Slow*, *Normal* and *Fast*. The values of the qualifiers are displayed in the rightmost inferior part of the window, using check boxes, where a checked box is associated with a value 1 for the corresponding flag.

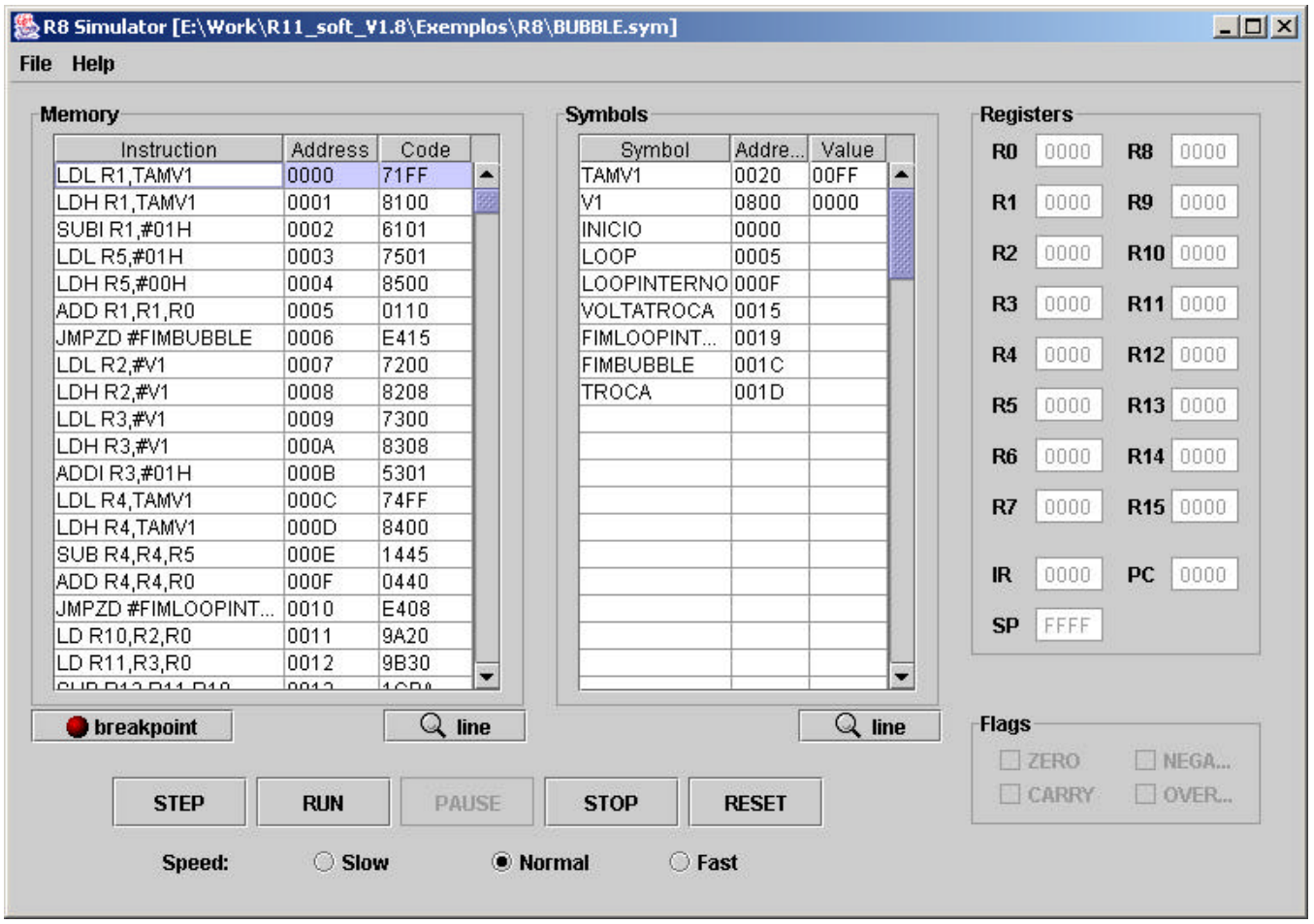


Figure 15 - R8 Assembler/Simulator Graphic User Interface.

The assembly tool accepts as input a text file written in the R8 assembly language. Given a file named <file>.asm, three output files are generated after processing by the assembler tool:

- <file>.hex – This is an Intel hex format file, a text file representing the object code and the application data contents in a hexadecimal format useful for downloading program and data to one of the prototyping boards for which R8 is available as an embedded processor;
- <file>.sym – This is a binary file used by the simulator;
- <file>.txt – This is a file containing the object code and the application data contents in a simple text format, adequate for use as input to a test bench running within a VHDL simulator (we have already used Aldec Active-HDL and Mentor Modelsim simulators). This format is illustrated above in Section 11.

The R8 assembler is mostly invisible to the user, since it is automatically called by the simulator interface when a file with .asm extension is read from disk. The above mentioned 3 output files are generated at the time of

loading the assembler file. If errors are found during the assembler execution phase, these are saved on a message log file. The simulator reads this file after the assembler execution, so that the errors can be presented to the user. In this last case, simulation is obviously not allowed to proceed, since an object file cannot be correctly generated.

12 REFERENCES

[1] N.L.V. Calazans, and F. G. Moraes. “Integrating the Teaching of Computer Organization and Architecture with Digital Hardware Design Early in Undergraduate Courses.” IEEE Transactions on Education, vol. 44, no. 2, pp 109-119. May 2001.