A Very Simple Microprocessor

By Etienne SICARD
www.microwind.org

This application note gives an introduction to microprocessor architecture. The goal of the project is to build a 4-bit processor at logic level and then simulate the processor at layout level.

1. Introduction

The Very-Simple-Microprocessor is an updated version of the very popular SAP (Simple-As-Possible) computer architecture proposed by Albert P. Malvino [RefBook] in 1993 in his famous book "Digital Computer Electronics". The VSM computer introduces the basic concepts of microprocessor architecture in the simplest possible way. The VSM is very primitive, but already quite complex, as seen in figure 1.

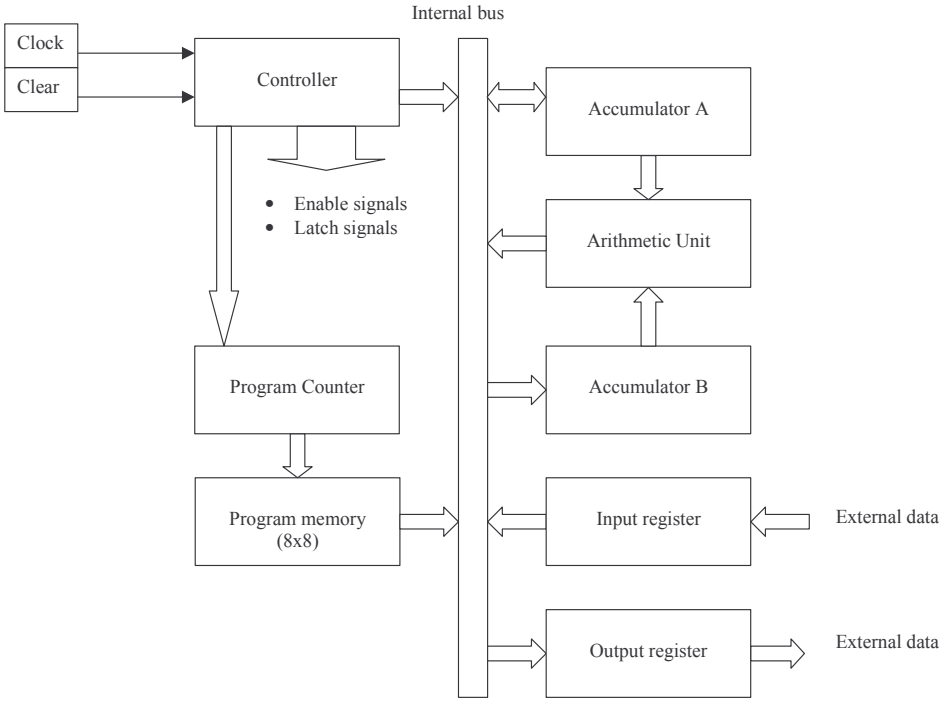


Figure 1: VSM basic architecture

The role of each block is described in table 1.

Block	Block	Size
Program Counter	The program counter counts from 0000 to 1111. It monitors the address of the active instruction. Initially, the program counter is set to 0000, so the microprocessor starts by the first instruction of the memory.	4 bits
Program Memory	The program memory stores the program. Each program line has an 8 bit format: the four most significant bits are the instruction itself, the least significant bits the data attached to he instruction, if necessary.	8x8 bit
Accumulator A	The accumulator A stores the intermediate results computed by the microprocessor. The accumulator is a 4-bit register. Upon request (EnableA), the accumulator result is placed on the internal bus.	4 bits
Accumulator B	The accumulator B is similar to accumulator B. Is is mainly used to supply the number to be added or substracted from accumulator B to execute an addition or a substraction.	4 bits
Arithmetic Unit	The Arithmetic Unit performs the operation S=A+B (Addition) Or S=A+~B+1 (Substraction)	4 bits
Input Register	The Input Register gives the opportunity to transfer data from the outside world inside the microprocessor.	4 bits
Output Register	The Output Register transfers the contents of the internal bus to the outside world. Usually, this instruction is performed at the end of the program to display the final result. The output register stores the output data on a fall edge of the clock. The output register is usually connected to an interface circuit which transfers or displays the result to the user.	4 bits

Table 1: The main blocks of the VSM architecture

The VSM circuit is based on a bus, called “internal bus” (IB). Each block listed in figure 2 may take the control of the bus, using a specific signal “Enable”. For example, the accumulator A has an enable signal *EnableA*. When *EnableA* is high, the 4 bits of the accumulator A go to the internal bus.

The list of enable signals is given in figure 2. The control of these enable signals is provided by the *MicroInstruction* block, which plays a fundamental role in the control of the microprocessor.

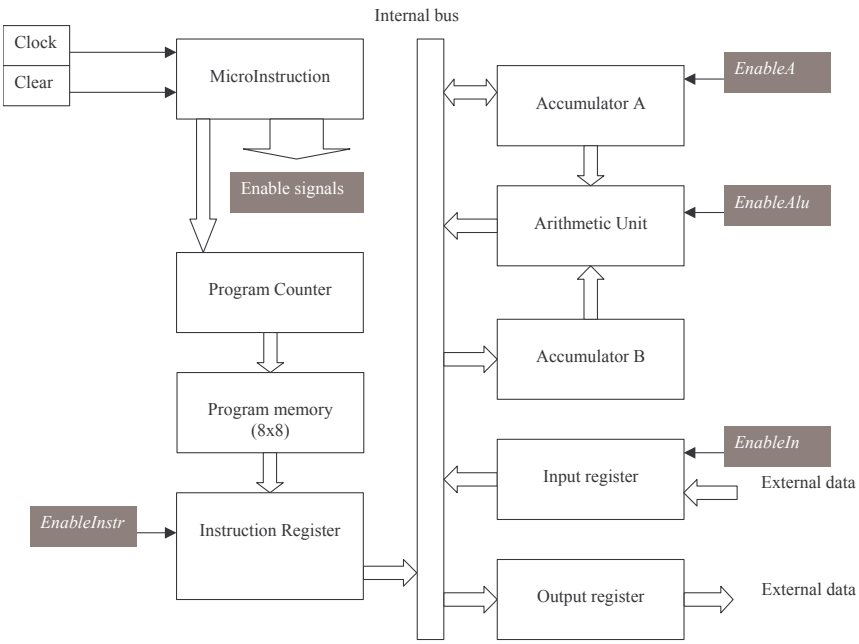


Figure 2: The controller generates "Enable" signals that allow one block to take the control of the bus

Enable Signal	**Description**
EnableA	Authorizes A to take control of the bus.
EnableAlu	Places the result of the arithmetic operation (ADD or SUB) on the bus
EnableInstr	Places the data part of the instruction (Four least significant bits) on the bus.
EnableIn	Transfers the contents of the external input on the internal bus.

Table 2: F our blocks may take the control of the internal bus, thanks to "Enable" signals

2. Instructions

The basic instructions are listed below. As the instruction is coded on 4 bits, only 16 different instructions may be considered.

No Operation (NOP=0000)

The No Operation instruction has no effect on internal register. However, this instruction is very interesting to understand how the basic clock controls work.

Addition (ADD=0001)

The contents of accumulator A is added to the data given as a parameter, and the result updates the accumulator A. The addition is performed on four bits. The carry is ignored. For example, considering that A=2, the instruction "ADD 3" corresponds to A=A+3, that is A=2+3. The final value of A is 5.

Subtraction (SUB=0010)

The contents of accumulator A is subtracted to data given as a parameter, and the result updates the accumulator A. The subtraction is performed on four bits. The carry is ignored.

Get Input (In=0011)

The input port is transferred to accumulator A.

Give Output (OUT=0100)

The accumulator A is stored on the output port. The output port is a four bit register that memorizes the output value and keeps it permanently available until its contents is refreshed by a new "Give Output" instruction.

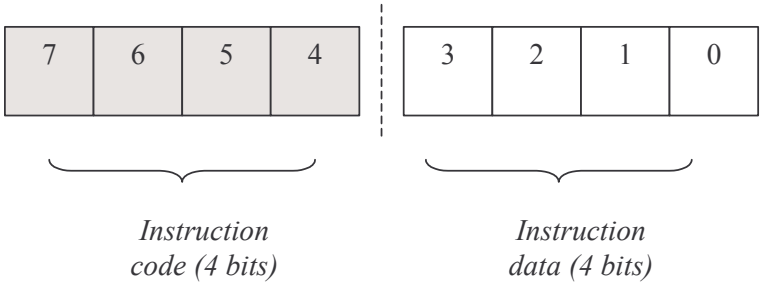
Load Instruction (LDA=0101)

LDA stands for "load the accumulator A with a value". For example, the instruction LDA 9 transfers the value 9 (1001 in binary format) to the accumulator A.

3. Program Memory

The program memory contains up to 8 bytes, where we store the instructions to be executed. Each instruction is coded on 8 bits. The most significant bits correspond to the instruction itself, while the least significant bits are the data. The following program loads the accumulator A with the value "2", then adds "1", and places the result to the output register.

The memory symbol is shown in figure 3. The memory has 8 registers, each register having 8 elementary memory cells. You can change the contents of the internal memory by a click at the desired logic cell. When you save the schematic diagram, you also save the memory contents. The memory symbol may be found in the basic symbol palette.



Mnemonic	**OpCode (binary)**	**OpCode (hexa)**
LDA 2	0101 | 0010	0x52
ADD 1	0010 | 0001	0x21
OUT	0100 | 0000	0x40
NOP	0000 | 0000	0x00

Table 3: The memory contains 8-bit information split into two parts: the 4 most significant bits for the instruction code and the least significant bits for the data. Some op-code examples are provided for LDA, ADD, OUT and NOP instructions

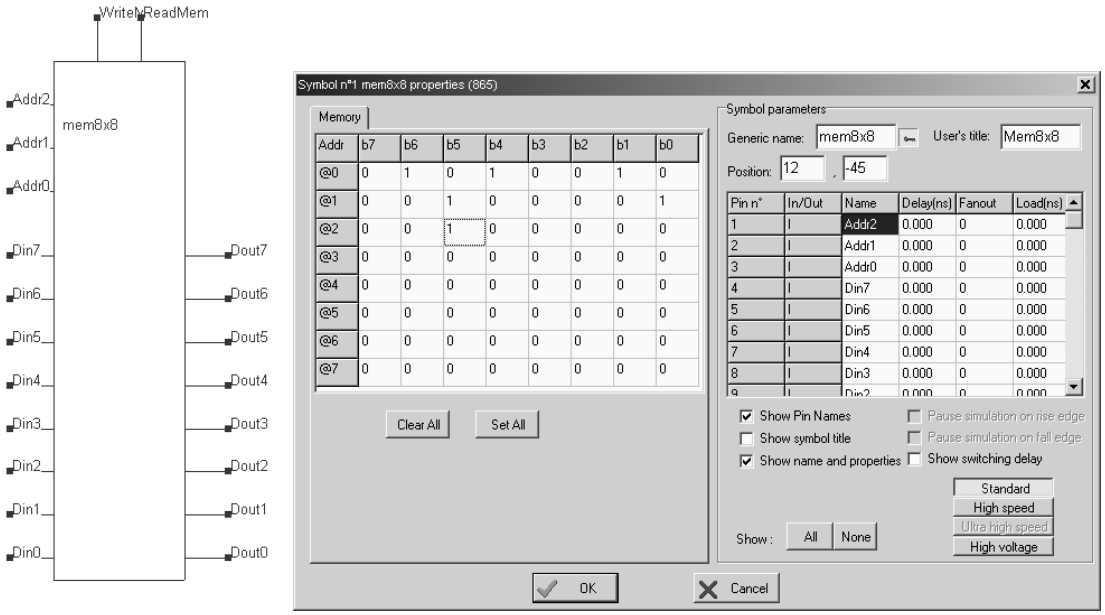


Figure 3: Storing the program in the memory(VSM-mem8x8macro.sch)

4. Executing the instructions

Introducing the micro-instructions

The mechanisms for executing the instructions are based on internal microinstructions. The execution of each instruction is based on four different time phases, as illustrated in figure 4. The reader should make the distinction between the microprocessor instruction itself such as "LDA 2" and the four internal microinstructions needed to complete the "LDA 2" instruction, called phase 0, 1, 2 and 4.

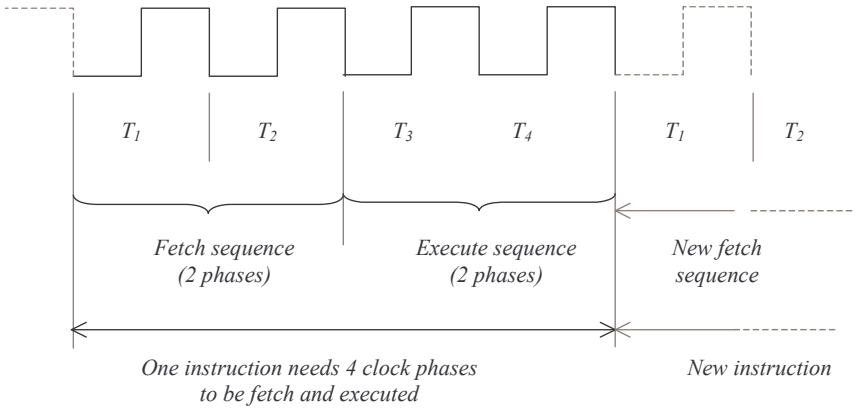


Figure 4: The execution of one instruction is based on six time phases including 6 microinstructions

In the execution of one instruction, we distinguish four phases, described as follows. The two first phases are called the fetch sequence. The corresponding microinstructions are independent of the user's instruction.

Phase	Name	Description
Phase 1	Address state	The contents of the memory is loaded by the instruction register.
Phase 2	Increment state	The program counter address is incremented. The register provides the microinstruction decoder with the instruction.
Phase 3	Execute step 1	Depending on the instruction, the microprocessor makes the first step of the execution phase.
Phase 4	Execute step 2	The microprocessor makes the second step of the execution phase

Table 4: The execution of one instruction is based on four time phases

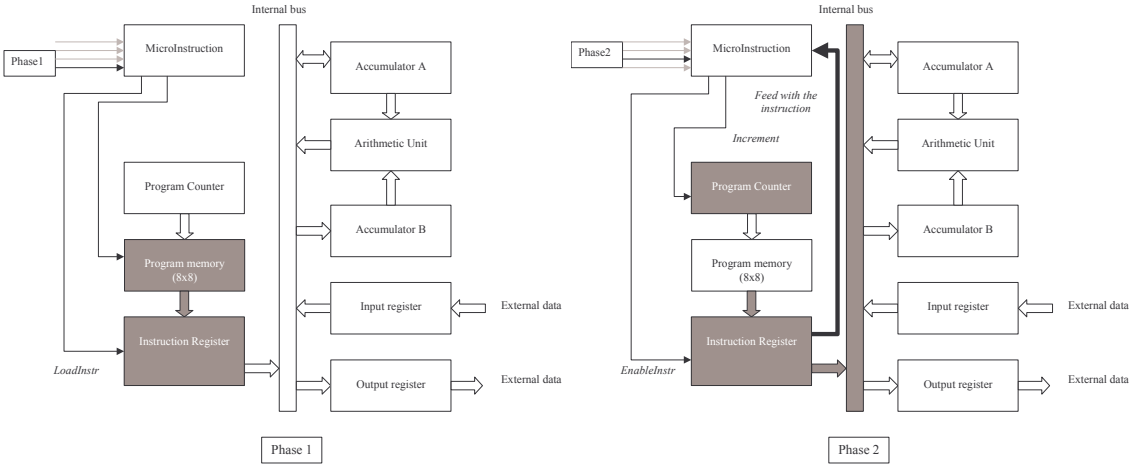


Figure 5: The execution of two first microinstructions representing the "fetch" sequence

No Operation (NOP=0000)

The Fetch sequence corresponds to the access to the memory (RedMem=1), the loading of the corresponding instruction (LoadInstr=1) during phase 1 (Figure 5). During phase 2, the stored instruction is sent to the microinstruction controller, while the counter is incremented. As the 'No Operation' instruction do not affect any internal register, the execution phases (Phase 3 and phase 4) do not correspond to any specific activity.

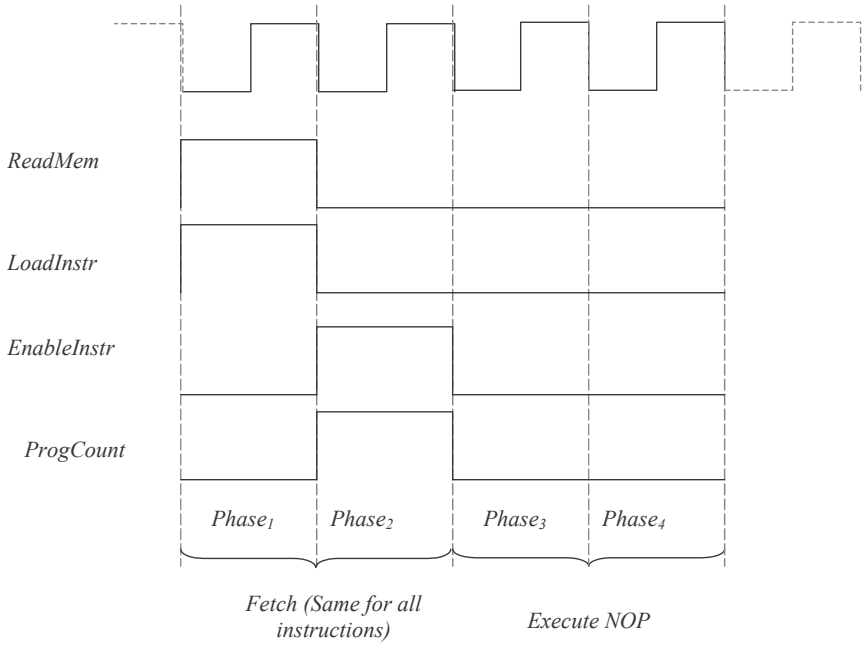


Figure 5: NOP fetch and execute sequence

Addition (ADD=0001)

The addition is performed between the accumulator A and the 4-bit data given as a parameter of the ADD instruction. Consequently, the addition is executed by storing the data in the accumulator B, then asking the arithmetic unit to produce the addition between accumulator A and accumulator B (Phase 4), and finally by transferring the result back to accumulator A on the rise edge of the clock during phase 4, as illustrated in figure 6.

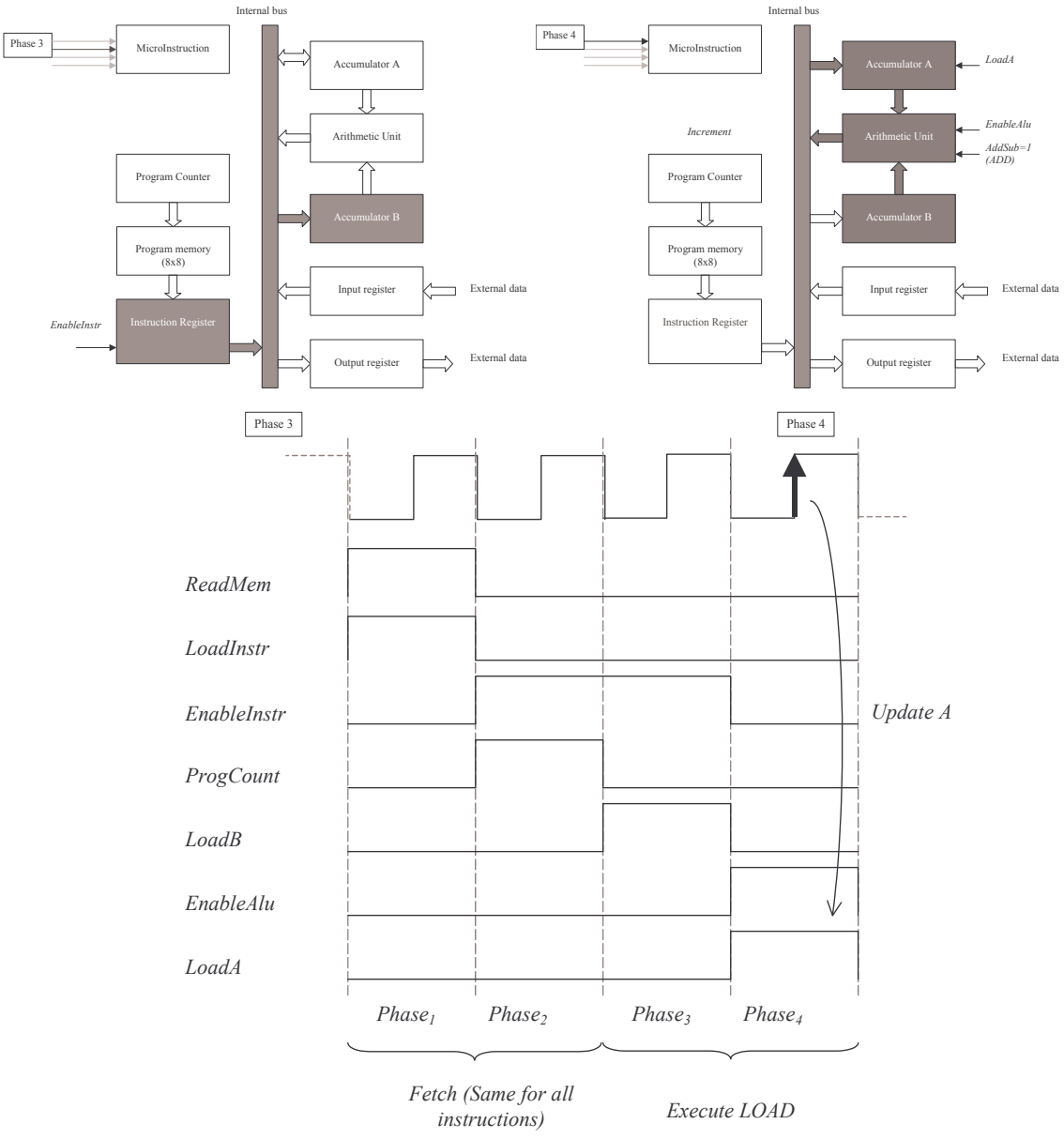


Figure 6: The execution the microinstructions corresponding to the ADD instruction

Subtraction (SUB=0010)

The execution phase of the subtraction instruction is identical to the one of the addition instruction. The only difference is that the “AddSub” is set to 0, which means “Substract”.

Get Input (In=0011)

The input port is transferred to accumulator A during phase 3 (Figure 6). There is nothing else to do in phase 4, where all registers remain inactive.

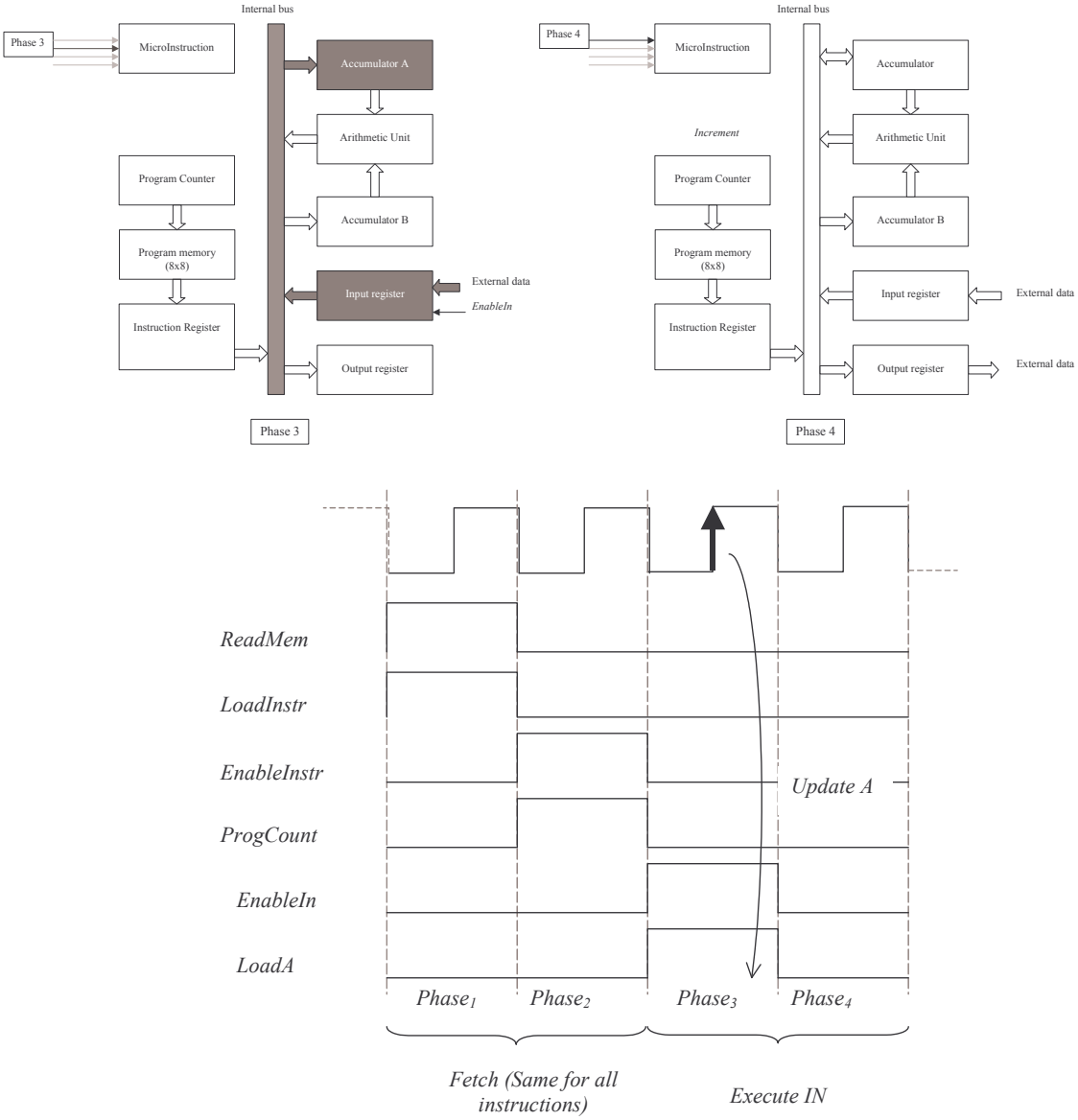


Figure 7: The execution the microinstructions corresponding to the IN instruction

Give Output (OUT=0100)

The contents of the accumulator A is transferred to the output port via the internal bus during phase 3. The output port memorizes the accumulator value and keeps it permanently available thanks to its four registers. The processor is inactive during phase 4.

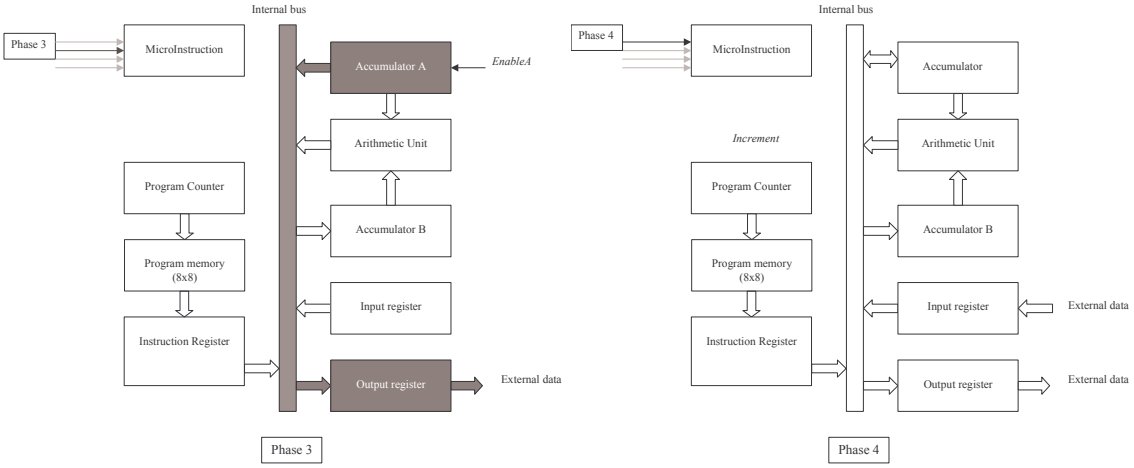
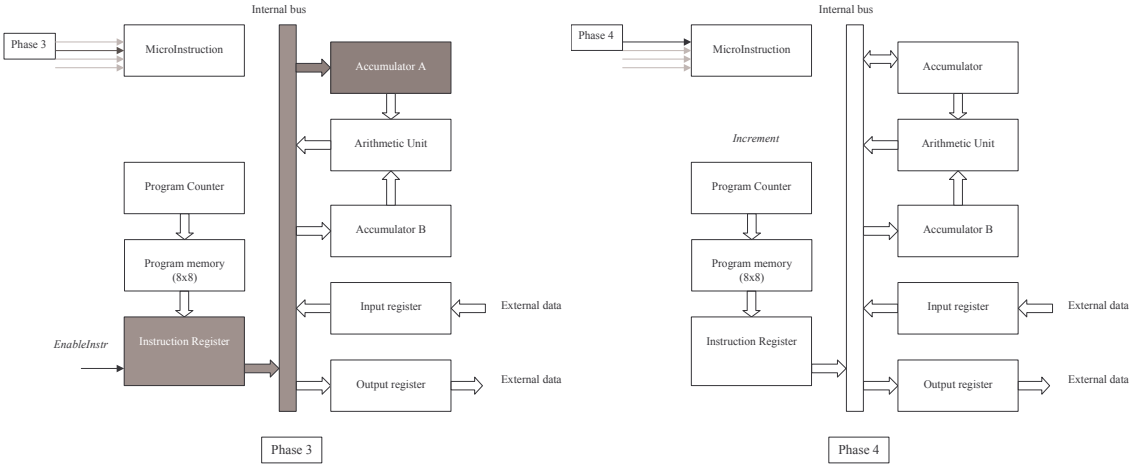


Figure 8: The execution the microinstructions corresponding to the OUT instruction

Load Instruction (LDA=0101)

The load instruction transfers the 4-bit data given as a parameter of the LDA instruction to accumulator A. For example, the instruction LDA 9 transfers the value 9 (1001 in binary format) to the accumulator A. In figure 7, the four least significant bits of the instruction register are placed on the internal bus and then transferred to accumulator A. As a result, the updated value of A is 1001. There is no activity during phase 4.



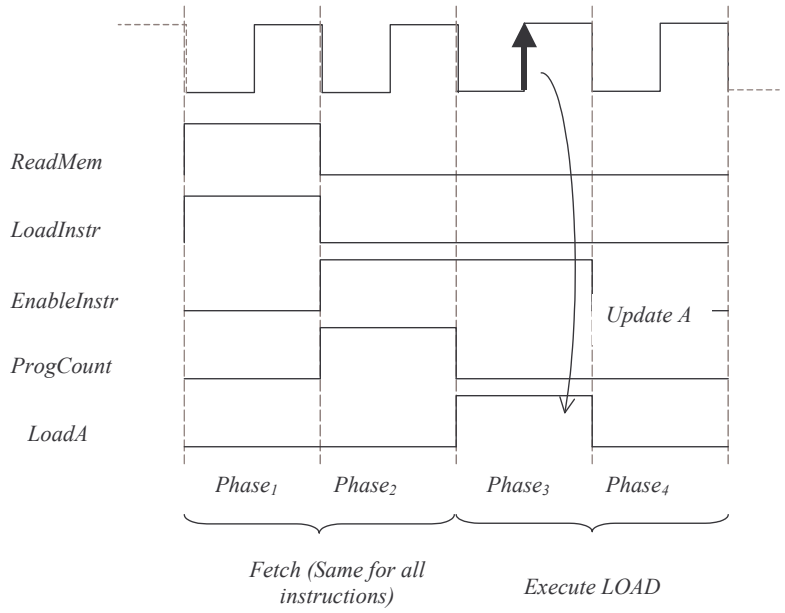


Figure 9: The microinstruction during phase 3 executes the load operation. During phase 4, the processor is inactive.

5. Basic Block design

We detail here the structure of each sub-block of the microprocessor.

Accumulator A

The accumulator is based on four edge-sensitive D-registers. The register output is permanently available through AluA0..AluA3 for the ADD and SUB operations. Meanwhile, the contents of A may be transferred to the internal bus when “EnableA” is asserted. We use three state inverters to build the access to the internal bus. The “latchA” signal authorized the transfer of input data (Here, a keyboard) to the D-registers at the fall edge of the main clock.

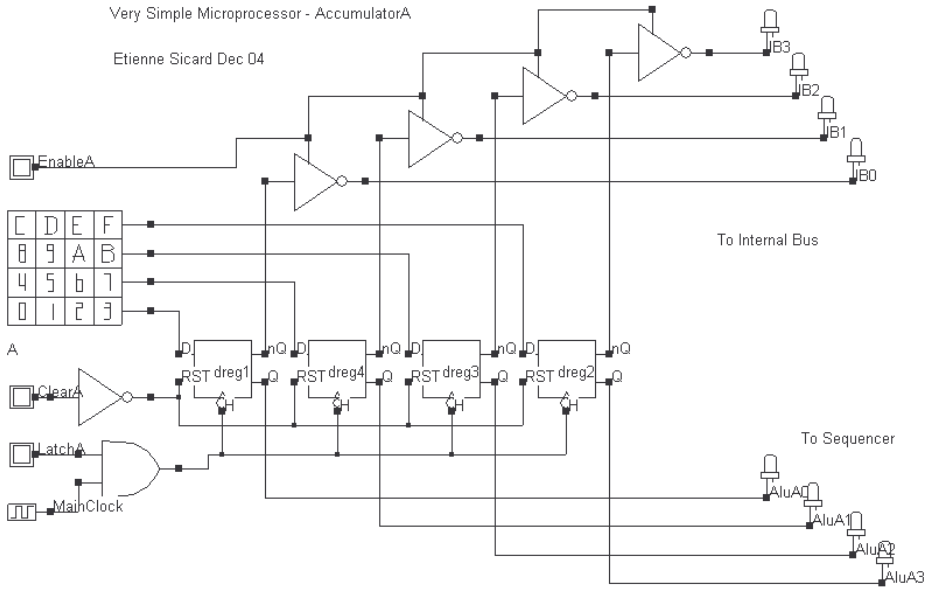


Figure 10: Structure of the accumulator A showing its connections to the internal bus and the arithmetic unit (Vsm-AccumulatorA.sch)

Accumulator B

As for accumulator A, the accumulator B is also based on four edge-sensitive D-registers. The register output is permanently available through AluB0..AluB3 for the ADD and SUB operations. The “latchB” signal authorized the transfer of input data (Here, a keyboard) to the D-registers at the fall edge of the main clock.

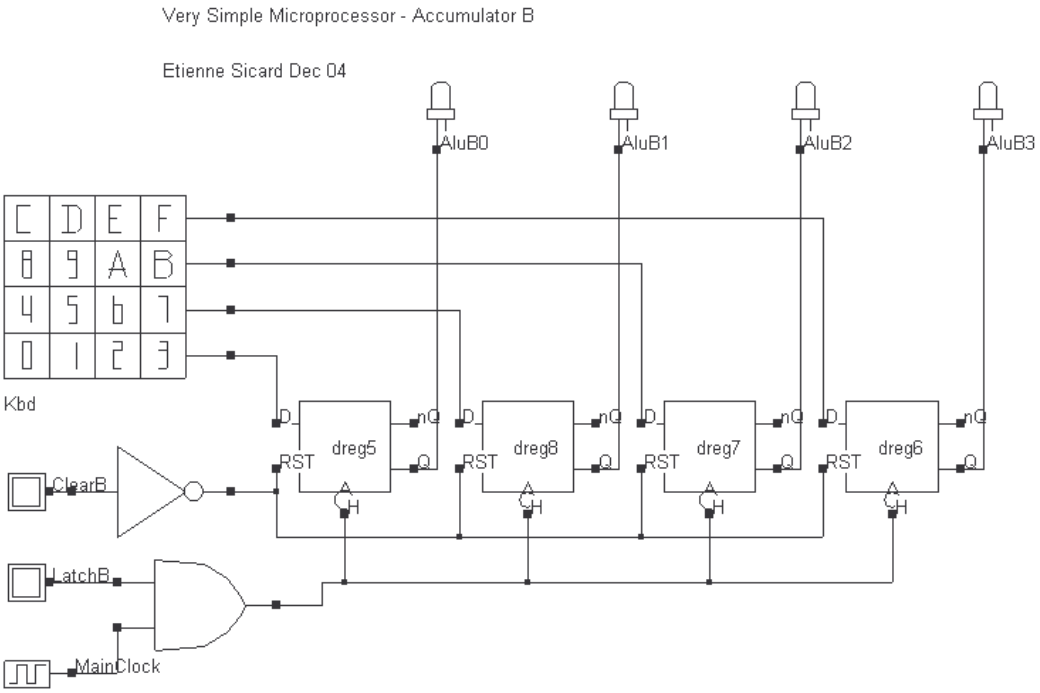


Figure 11: Structure of the accumulator B showing its connections the arithmetic unit (Vsm-AccumulatorB).sch

Add/subtract Block

The addition is based on full adder sub-circuits, that have been described in Microwind/Dsch user's manual. The internal structure of the full adder is a set of XOR gates for the "SUM" signal and a complex gate for the "Carry" signal, as shown in figure 12.

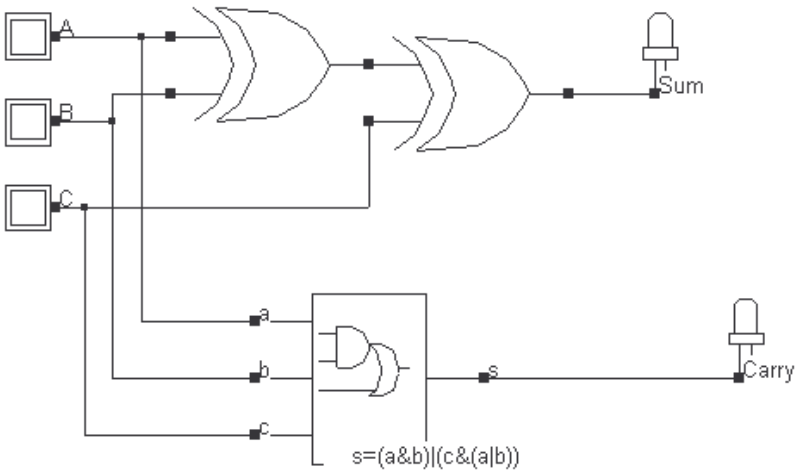


Figure 12: the internal structure of the FullAdder (Vsm-fullAdder.sch)

Adding two 4-bit information requires four full-adders chained as illustrated in figure 13. The carry signal propagates from the lower stage to the upper stage in order to perform the complete add operation.

To subtract two numbers (B-A in this case) and reusing the same full-adder circuits, we need to build two supplementary things:

- A circuit that creates the 2's complement of A
- A small circuit that sets the initial carry to 1.

One approach consists in implementing multiplexer circuits, that may be found in the symbol palette, advanced symbol menu, sub-menu "Switches". When "Sel" equals to 0, the input i0 is transferred to the output, otherwise, i1 is transferred to the output. Consequently, "AddSub"=0 corresponds to the transfer of A to the adder chain, (Add operation), while "AddSub"=1 corresponds to the transfer of ~A to the adder chain (Sub operation).

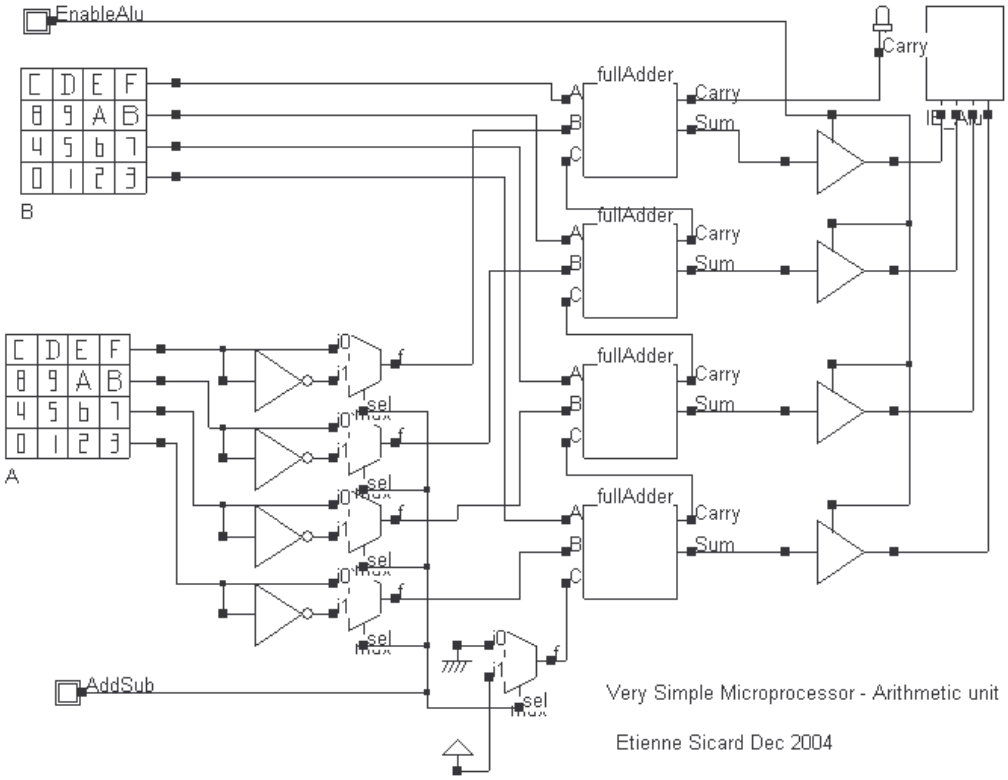


Figure 13: Structure of the arithmetic unit which may perform the ADD and SUB operations (Vsm-ArithmeticUnit.sch)

At that point, it sounds very interesting to connect the accumulators and the arithmetic unit in order to perform manually what the microprocessor will later do with its internal sequencer. The circuit made of the accumulator A, B and the arithmetic unit is shown in figure 14. The two keyboards serves as inputs A and B, the displays are placed on the output buses and the arithmetic unit connection to the internal bus.

Trying to operate this simple circuit is a very interesting introduction to the microprocessor operation. Below is the set of actions we need to perform sequentially in order to add two numbers:

- De-active the main Reset. Initially, the Reset pin is set to 0 (Default value at start), which corresponds to an active Reset. Both registers A and B are cleared (A=0, B=0). Nothing may work until you set the button "~MainReset" to 1.
- Load the desired value on A. Click on the lower keyboard named "A", for example 3. Click "LatchA" and wait at least one complete cycle of the main clock. The accumulator A stores 3 at the fall edge of the clock.
- Load the desired value on B. Click on the upper keyboard named "B", for example 2. Click "LatchB" and wait at least one complete cycle of the main clock. The accumulator B stores 2 at the fall edge of the clock. The arithmetic unit computes the sum A+B as "AddSub" is set by default to 0, which corresponds to the ADD instruction. However the result is not displayed as "EnableAlu" is 0.
- Set "EnableAlu" to 1 to display the result "5", as shown in figure 14.

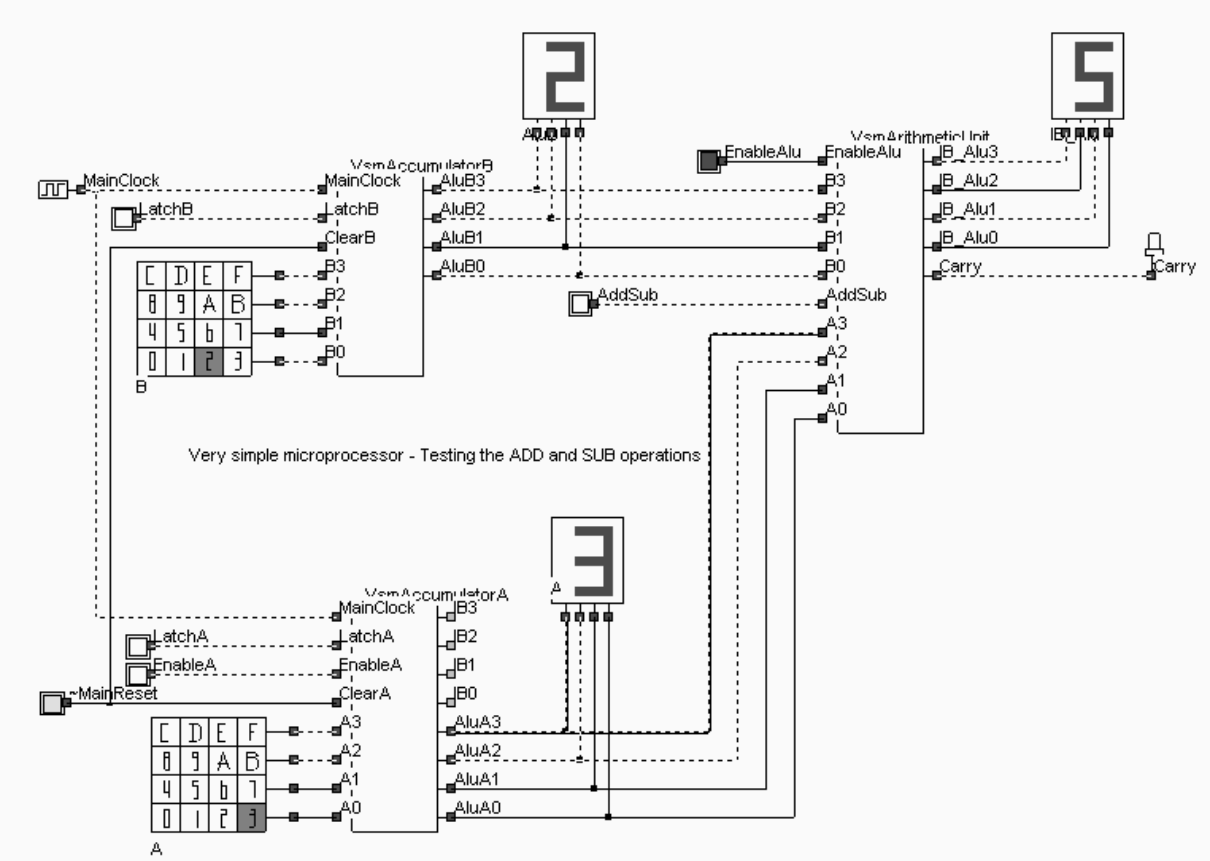


Figure 14: The connection between accumulators A, B and the arithmetic unit to test the ADD and SUB instructions (Vsm-RegARegBAlu.SCH)

The input register

The input register is a simple set of 3-state buffers, as shown in figure 15. There is no need for D-registers as the input will be directly transferred to the accumulator A.

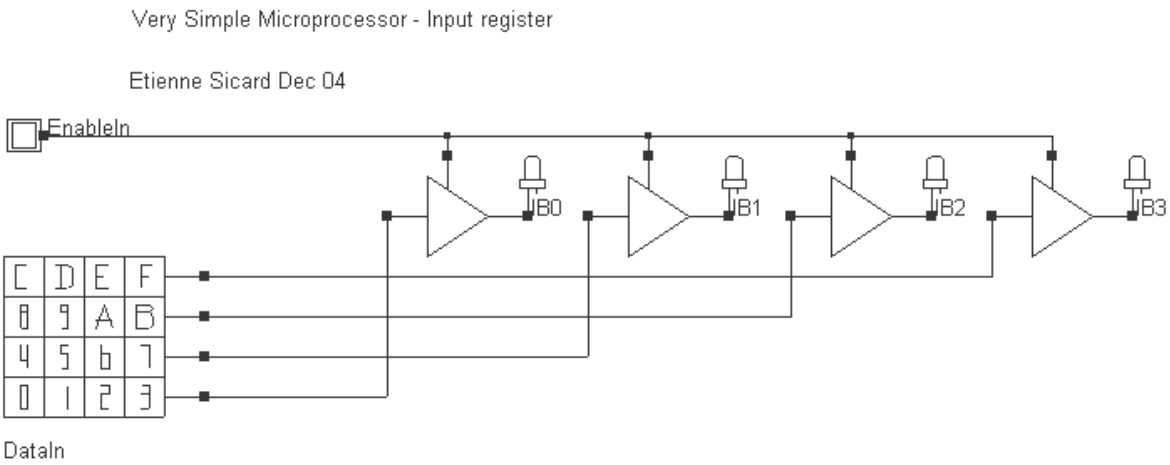


Figure 15: The input register (Vsm-InRegister.SCH)

The output Register

The output register is based on D-register cells as shown below. On the positive edge of the clock, the data is saved in the registers. It is very important that the data is stored on the positive edge of the clock during phase 3, and not on the negative edge, that would rise synchronization conflicts. Therefore, a NAND gate is used to make the circuit sensitive to the rise edge of the main clock, as seen in figure 16.

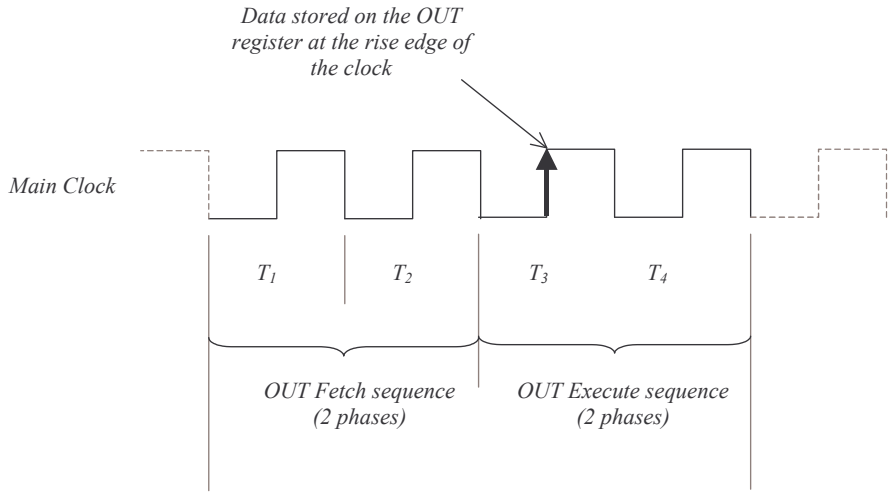


Figure 16: The output register must store the data at the rise edge of phase 3

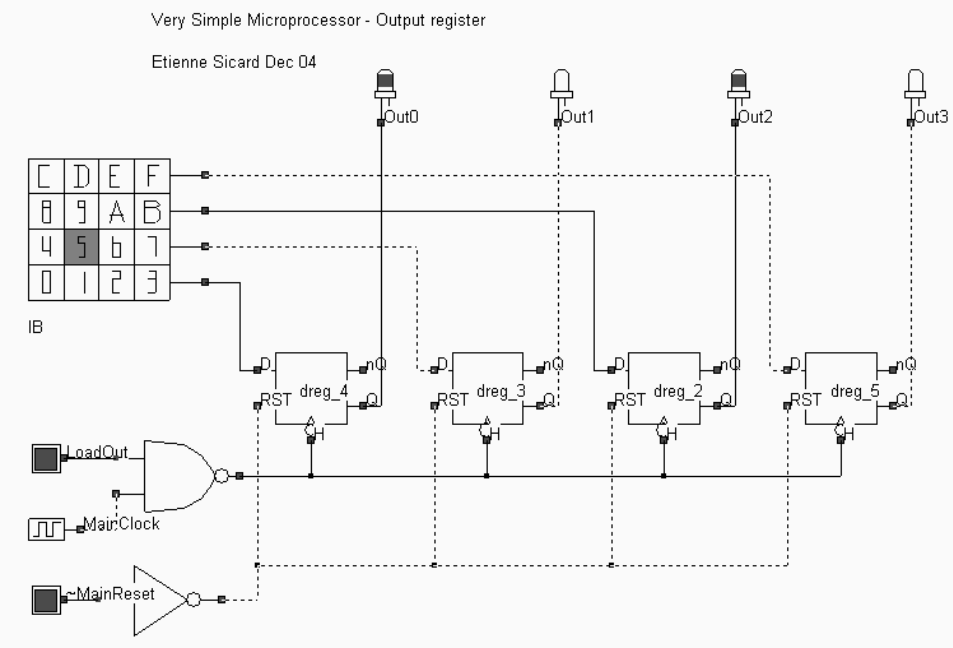


Figure 17: Internal structure of the output register (Vsm-OutRegister.SCH)

A manual microprocessor

In this paragraph, we propose to build a "manually-controlled" microprocessor which consists in accumulators A, B, the input and the output register. The goal of the simulation reported in figure 18 is to transfer the input information (DataIn) to the output port (DataOut). To perform this transfer, we need to enable the input port (EnableIn), and then Enable the output port. At the next rise edge of the main clock, the contents of the input keyboard (5 in this case) will appear in the display corresponding to the output. Several other transfers may be performed:

- Input register to accumulator A
- Input register to accumulator B
- Result of the addition of A and B to the output port

The arrow symbol (Symbol menu "Advance", symbol "Arrow") is used to ease electrical connections between clock and reset signals. In the basic example shown in figure 18, the connection is built automatically between all arrows having the same name. Double click the arrow symbol to access the arrow name which identifies the electrical net. In the example, we build two different electrical connections, one called "Clk", the other called "Rst". Notice that the electrical node names are not case sensitive.

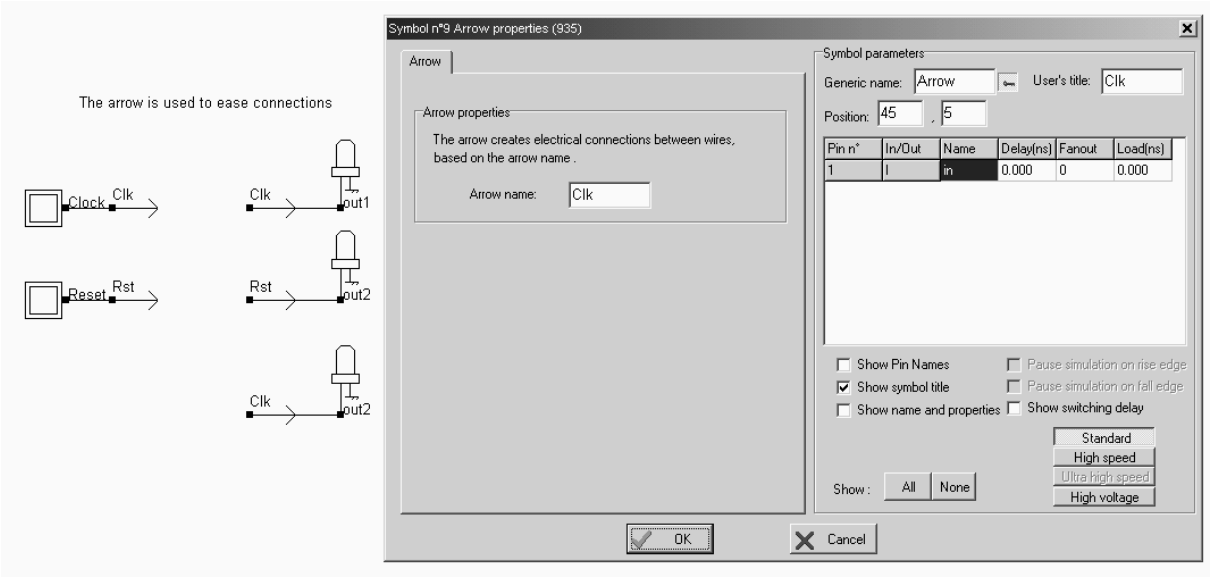


Figure 18: Building arrow connections to ease the electrical wiring of main signals (Vsm_arrow.SCH)

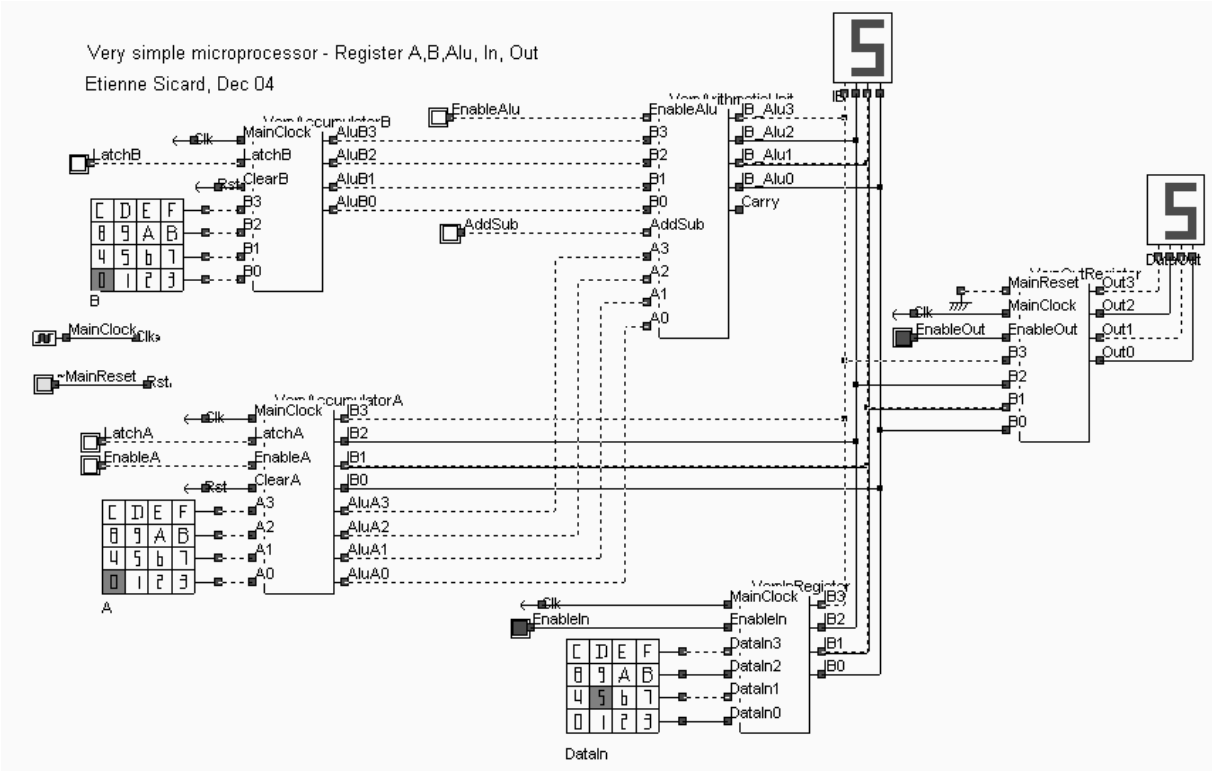


Figure 19: A manually-controlled microprocessor (Vsm-RegARegBAluInOut.SCH)

The Phase Generator

In order to transform the previous 'manual' microprocessor into a fully programmable microprocessor, we need to build several circuits to create the appropriate control signals. Firstly, the phase counter must produces the four phase signals Phase0 to Phase3, at the negative edge of the clock.

The counter must be reset by a signal “Clear” active low. The design of the phase counter is based on edge sensitive latch and XOR gates, as shown in figure 20.

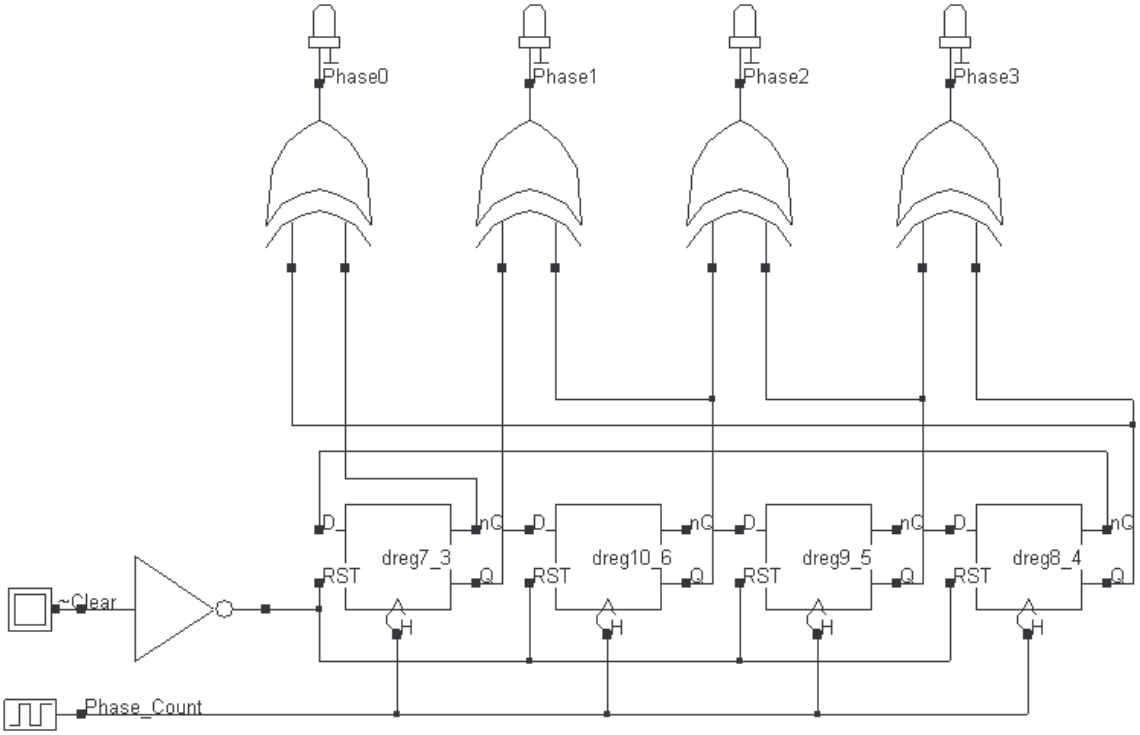


Figure 20: The phase counter structure(Vsm-RingCounter4.sch)

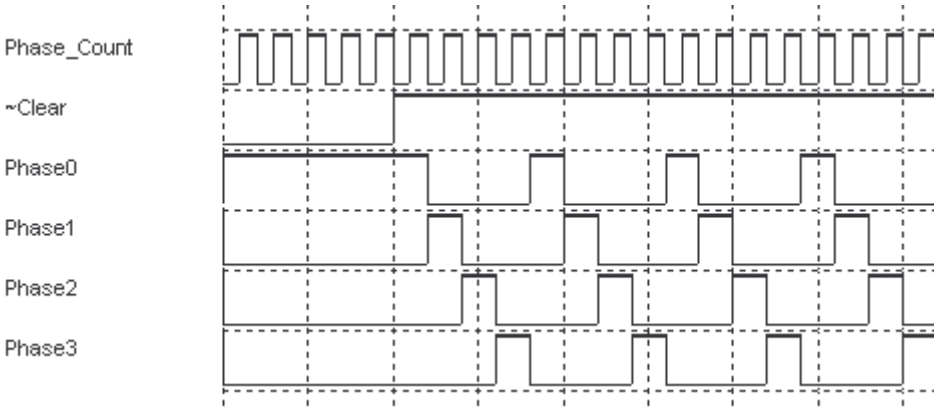


Figure 21: Simulation of the phase counter (Vsm-RingCounter4.sch)

When the clear signal becomes inactive (high value), the phases appear sequentially .

Program Counter 0 to 15

The program counter plays a very important role in the microprocessor as it supplies the main program memory with the address of the active instruction (Figure 22). At start, the program counter is 0. Then, at the end of each instruction, the program counter is incremented, in order to select the next instruction.

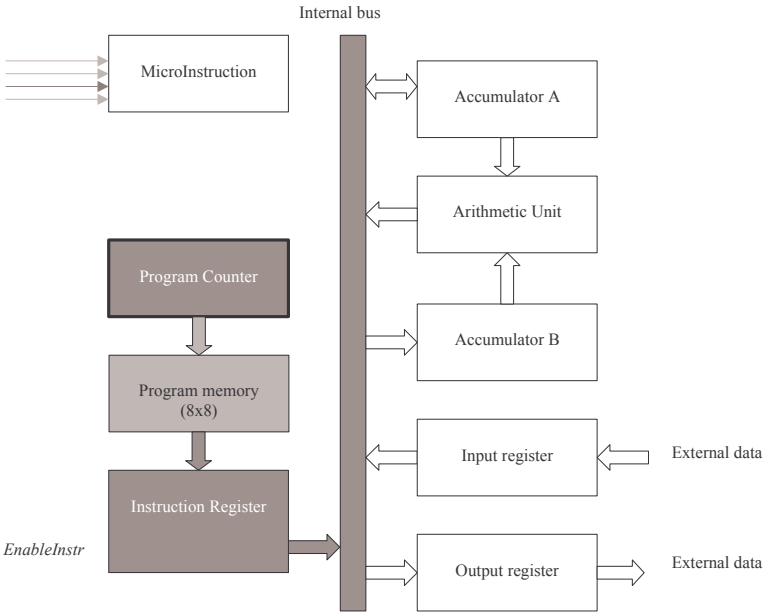


Figure 22: the program counter supplies the program memory with the address of the active instruction

One simple way to build a counter from 0 to 15 is to chain edge-sensitive D-registers, as shown below. The circuit is very simple, but works asynchronously. This means that due to propagation delays between stages, some intermediate results appear on the display during a very short period of time. The glitches have no impact in our microprocessor design as the circuit counter is incremented during the phase 2 of the microinstruction sequence, and is only exploited at the next instruction during phase 1, to load the instruction register.

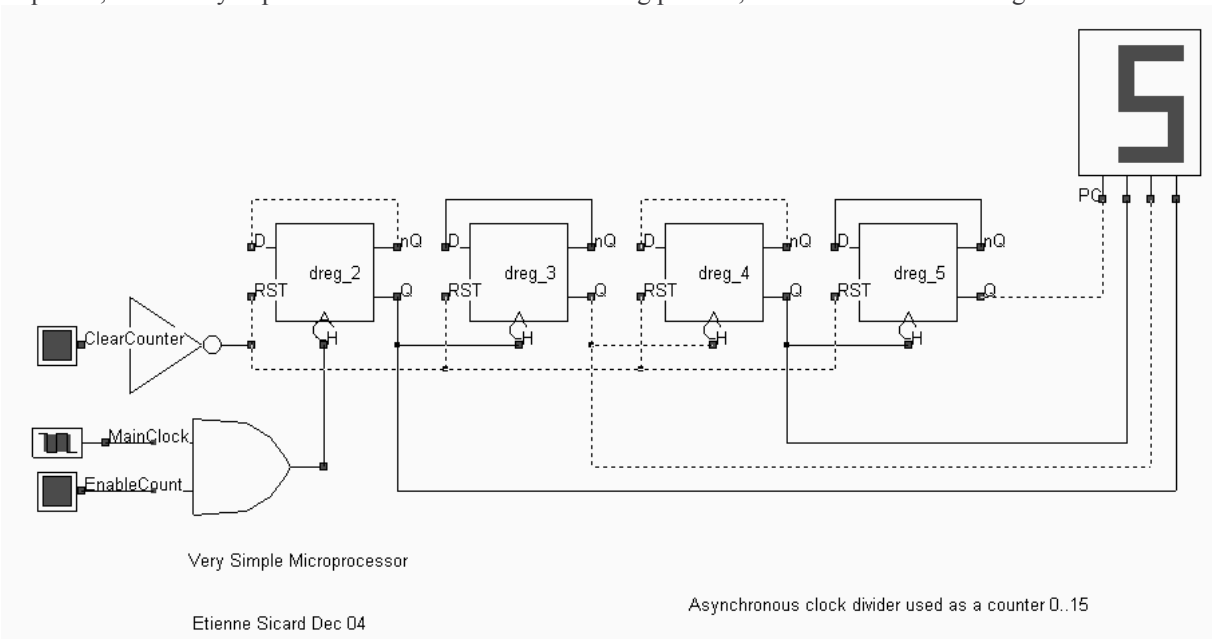


Figure 23: the program counter at work. Counting is enabled only during phase 2, at the fall edge of the main cloc (Vsm-Counter16.SCH)

The Instruction Register

The instruction register stores the contents of the program memory. The 8-bit information is split into two parts: the most significant bits correspond to the instruction itself, while the least significant bits are the data. The instruction code is stored by the four D-register circuits situated on the bottom of figure 23, in order to be permanently available for the microinstruction decoder, while the data is stored in four separate D-register cells, and can be made available on the internal bus. The instruction register gives a copy of the current instruction and releases the main memory, which can be accessed later for both read or write operation.

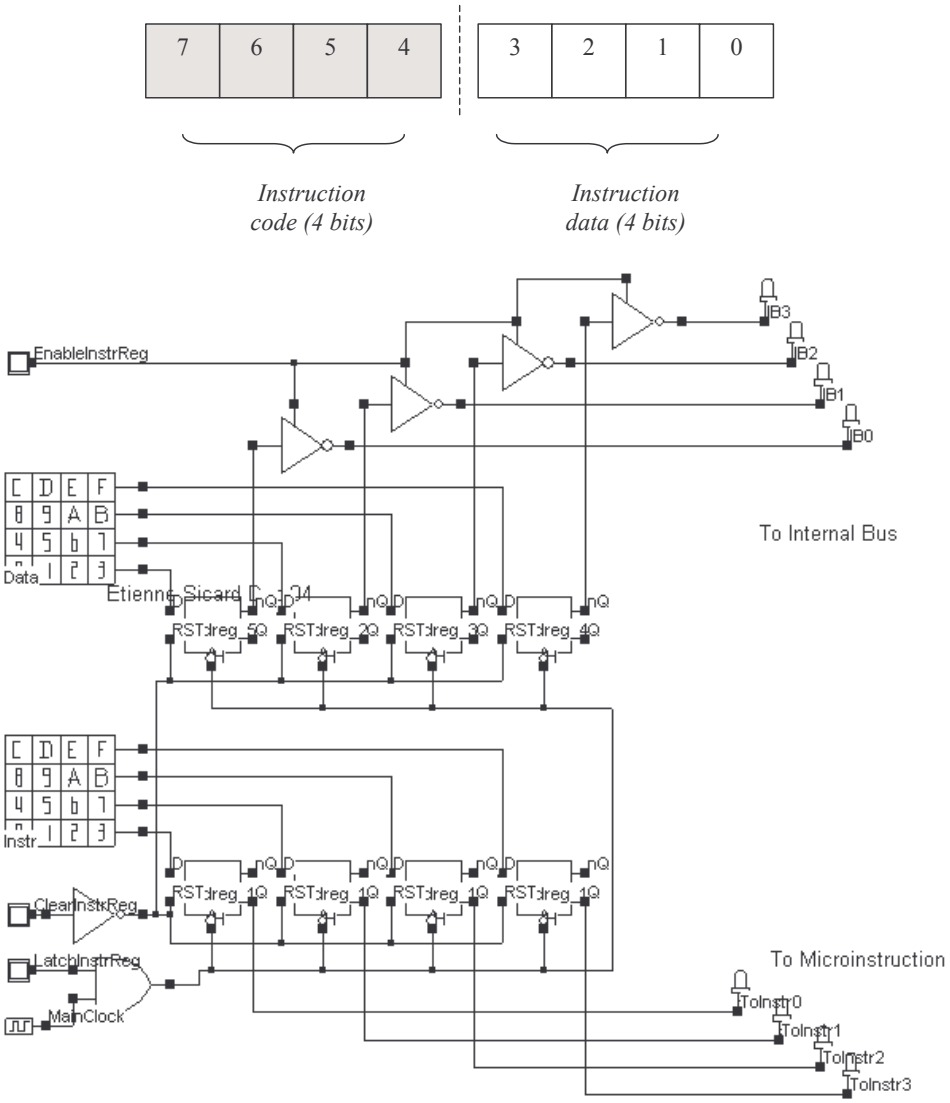


Figure 24: the instruction register stores the contents of the memory and separates the code part (Lower registers) from the data part (upper registers) (Vsm-InstructionReg.SCH)

The MicroInstruction Controler

The microinstruction is the 'heart' of the microprocessor. It controls the most important signals such as the 'Enable' and 'latch' controls. The input of the microinstruction register is the instruction code itself, plus the phase information. The four input AND gates serve as instruction decoders. For example, the instruction 0000 turns on the upper AND gate, which corresponds to the NOP microinstructions. Notice that pahes0 and phase1 are not connected to the instruction decoder. This is because the two first phases are not dependent on the instruction itself. Then, depending on the type of instruction, the desired signals are set to 1 if active, or kept at 0 to be inactive.

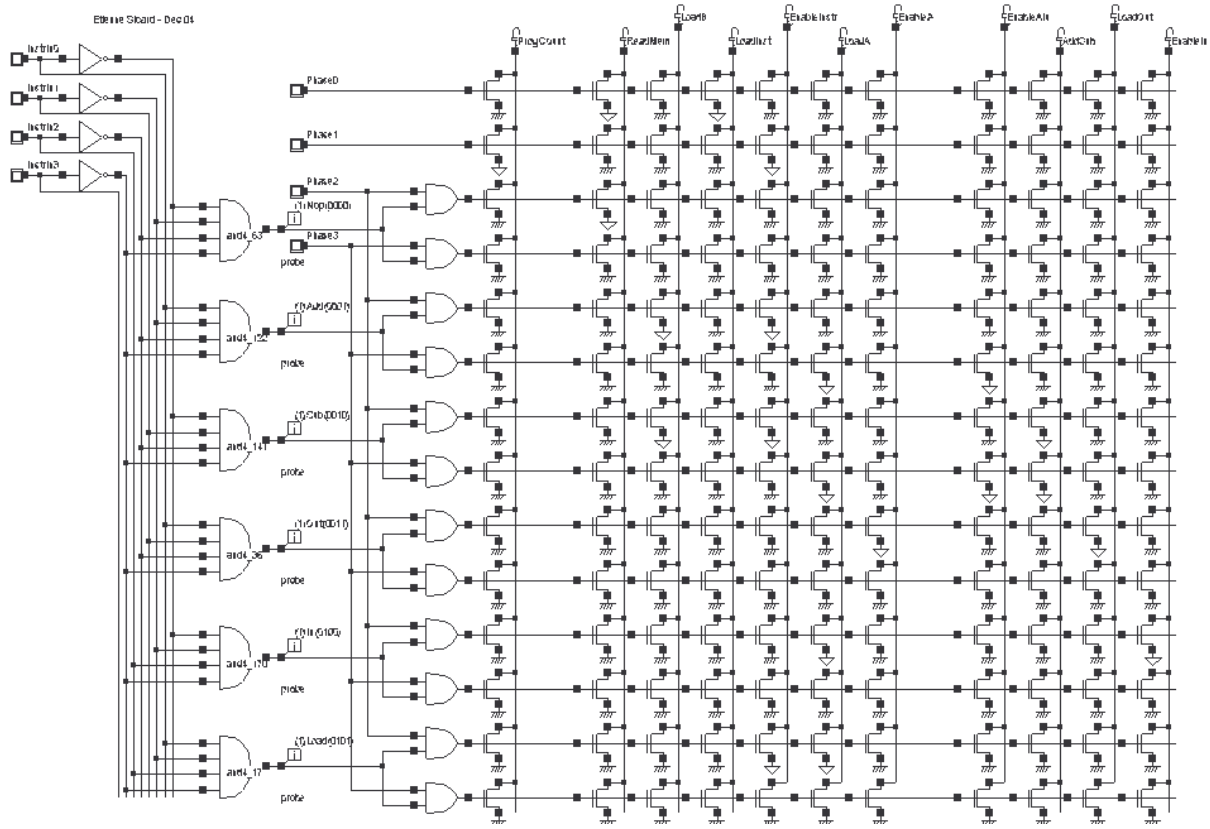


Figure 25: the microinstruction register stores the sequence for the two last phases for each instruction

The Complete Microprocessor

It is time now to connect all sub-circuits together and test the whole microprocessor. Each sub-block has been embedded into a symbol where only the input and output pins appear. The complete circuit is shown in figure 26. However, we should keep in mind that this is only a "very simple" and very low complexity microprocessor. Before starting the simulation, we must write the program inside the memory.

In the example shown in figure 26, three instructions have been written:

Mnemonic	**OpCode (binary)**	**OpCode (hexa)**
LDA 1	0101 | 0001	0x51
ADD2	0010 | 0010	0x22
OUT	0100 | 0000	0x40

Table 6: the code stored into the program memory

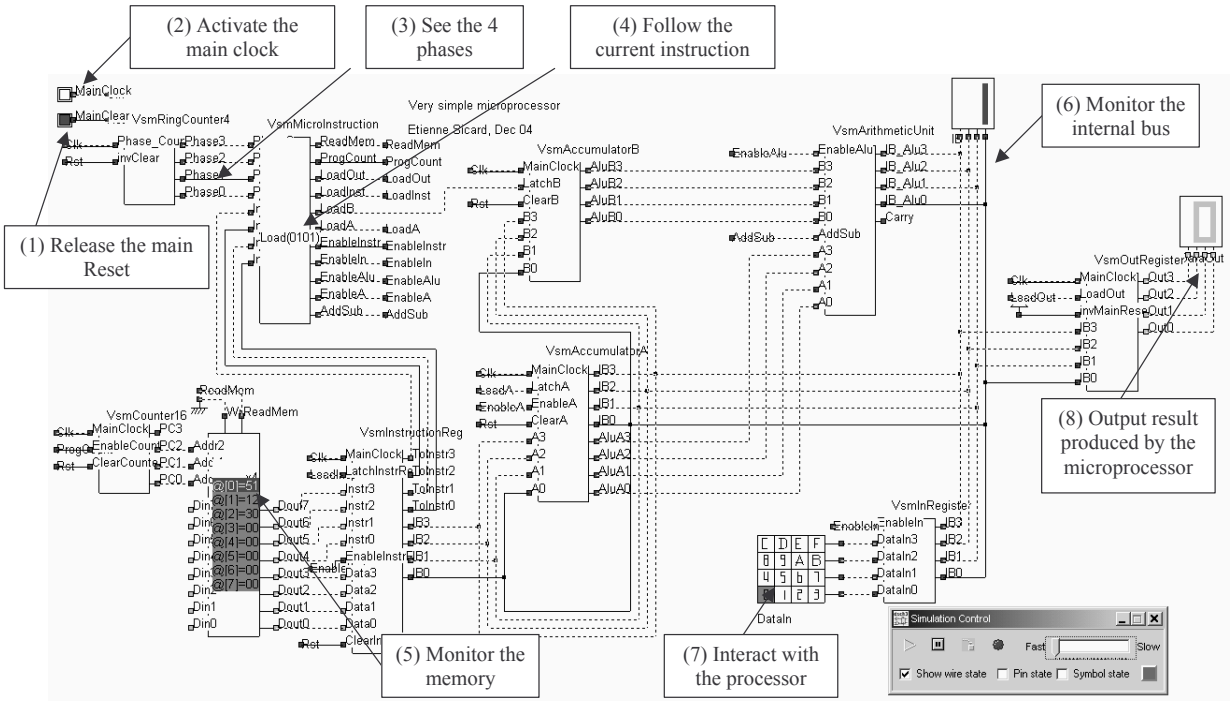


Figure 26: the microprocessor circuit ready for simulation (Vsm-Microprocessor.SCH)

Once simulation has started, there are several things to do in order to run the code:

- De-active the reset (1)
- Click the main clock (2)
- At each active edge of the clock, observe the phase counter shift from phase0 to phase1, phase2, phase3 and back to phase0 (3).
- Starting phase 2, the instruction is loaded to the microinstruction controller. The active instruction appears as shown in (4), which corresponds here to “Load (0101)”.
- You can monitor the memory contents and the active memory location (5).
- Also worth of interest is the monitoring of the internal bus (6).
- If required by the program, you can enter data to the keyboard by dataIn (7)
- If an instruction “OUT” is running, the result, should appear on the output display (8).

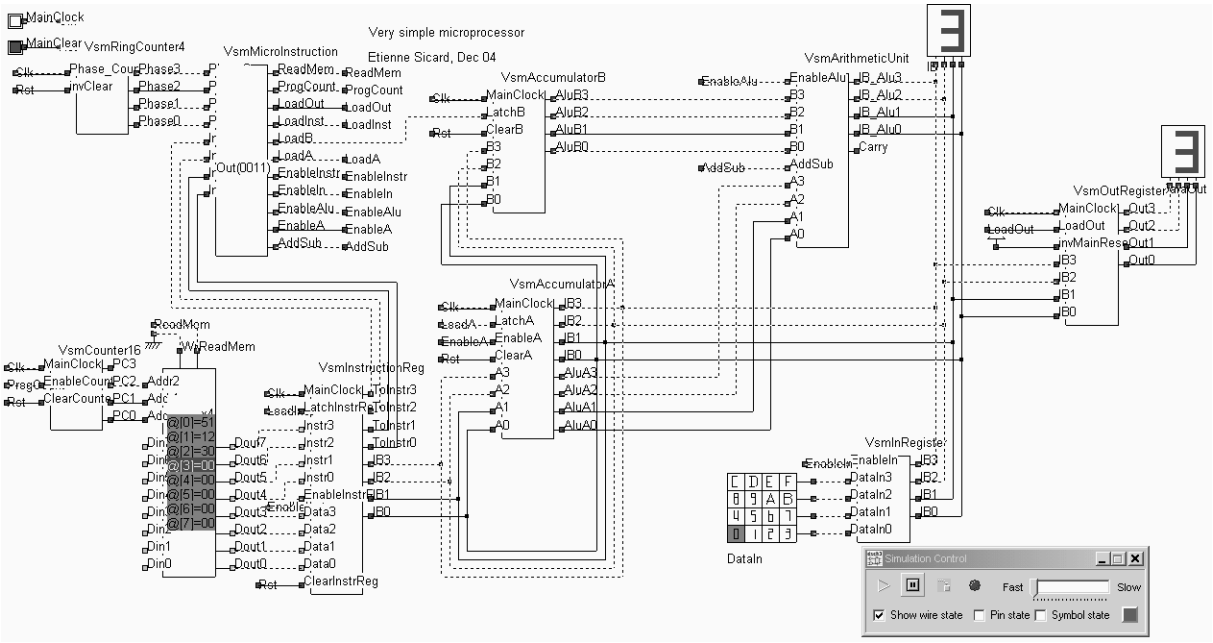


Figure 27: Final result for the addition of 1 and 2 using the program proposed in table 6 (Vsm-Microprocessor.SCH)

Memory Move

One important feature NOT handled by the very simple microprocessor is the memory move (MOVE). This instruction consists in transferring the contents of one memory location to accumulator A, or its opposite. Why didn't we build this functionality in the first version of our processor? Because the structure of the memory control and access must be deeply modified and lead to a significant amount of supplementary hardware.

Assuming that the MOVE operation transfers the contents of one memory location to A, we need to perform the following sequence: during phase 3, we need to have access to a *new* memory contents, which is not the one stored in the program counter. This means that a new bus access must be provided in the processor from the memory to the internal bus, without altering the contents of the instruction register. The differences between the two structures is displayed in figure 28.

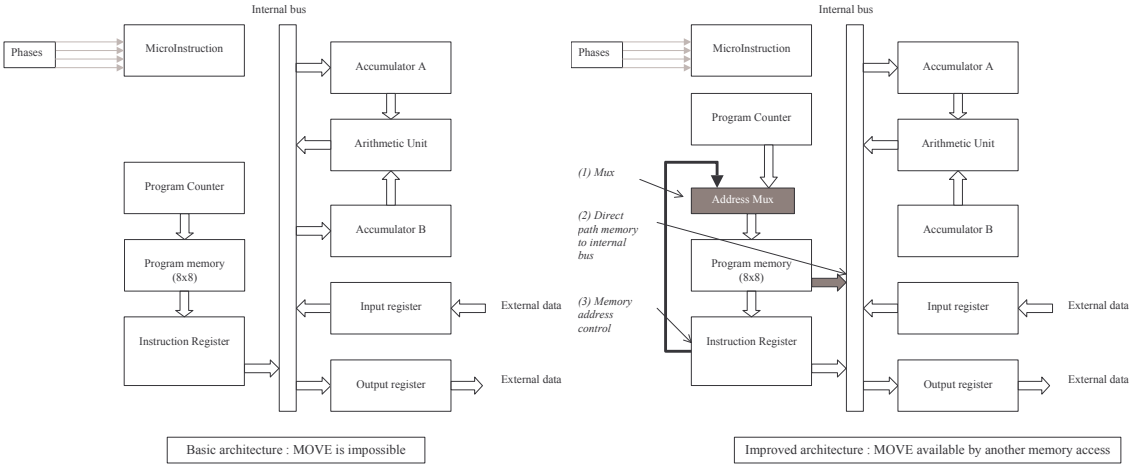


Figure 28: Modifying the microprocessor to handle the MOVE instruction

In practice, the MOVE circuit can be realized by adding a direct path to the internal bus (With its appropriate Enable control), a 4-bit address bus from the instruction register to the memory address, and a ProgramCount/Register address multiplexer.

Physical Implementation

Description of the design flow

The VSM processor has been described and simulated at logic level using DSCH, and saved under the name **vsm-microprocessor.SCH**. It can be converted automatically into layout using MICROWIND. The design flow is detailed in figure 29. We create the VERILOG description of the VSM processor using the command **File →Make Verilog File**. The text file **vsm-microprocessor.TXT**, which includes the VERILOG description, serves as the input of MICROWIND, to drive the automatic compilation of the circuit into layout. In MICROWIND, the command is **Compile →Make Verilog File.**

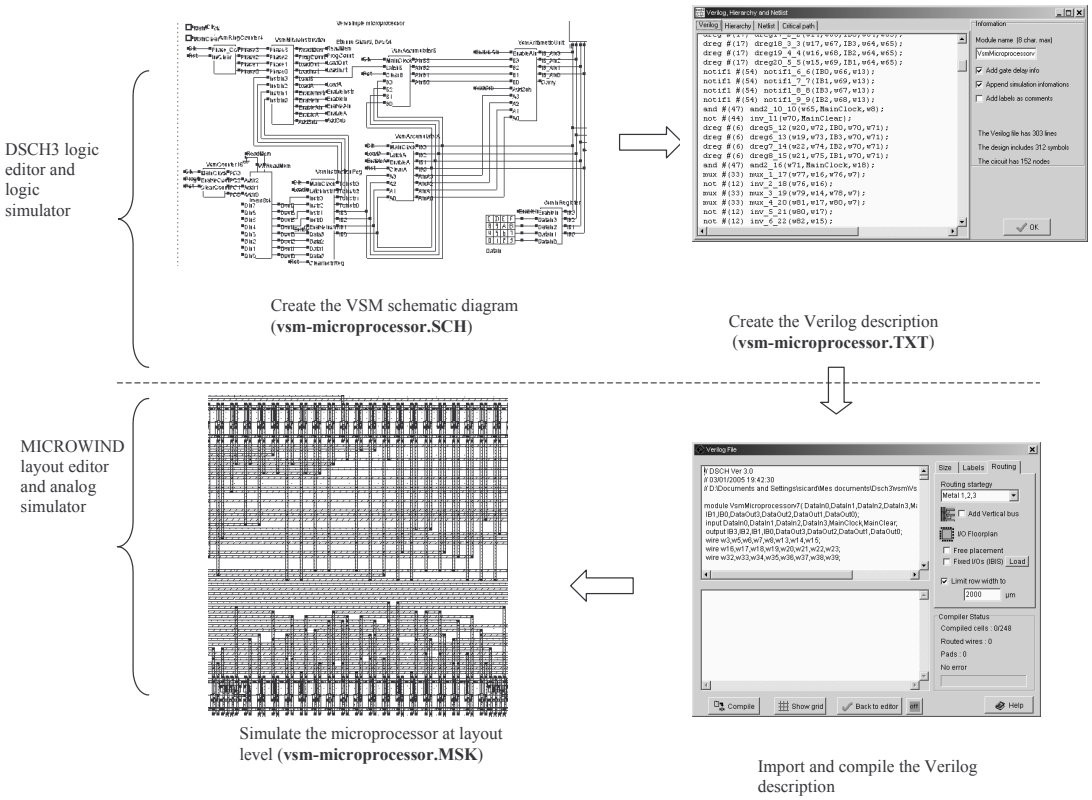


Figure 29: Compiling the logic circuit from its logic description, through the VERILOG format

VERILOG translation

In its basic version, the microprocessor includes 312 primitives. This relatively small number of devices is due to the fact that the memory is not handled in that translation. The memory symbol used in the microprocessor design is a macro-cell that is ignored during the translation in VERILOG.

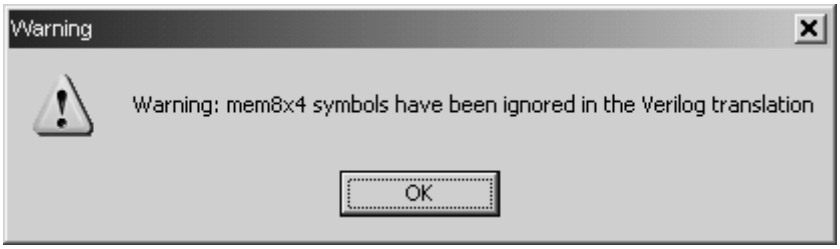


Figure 30: Warning concerning the memory macro that has not been translated into a VERILOG standard description

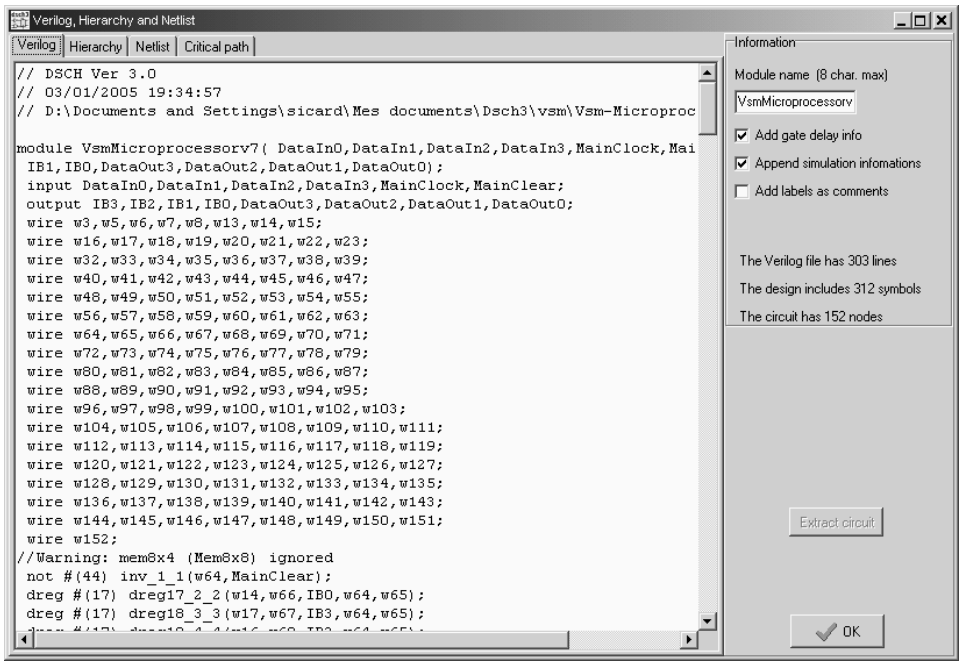


Figure 31: A partial view of the VERILOG description of the 4-bit microprocessor (vsm-microprocessor.SCH)

To generate a complete layout of the microprocessor, we need to design a cell-based 8x8 bit memory that work exactly as the memory macrocell. This can be done by constructing an array of 8x8 register cells based on very simple ring inverters as shown in figure 30. The access to write the information in the memory cell is provided by the nmos in series controlled by 'Write' (N1).

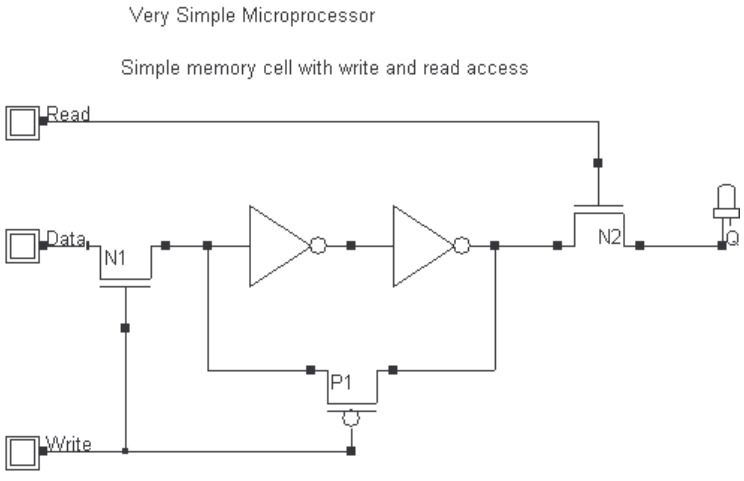


Figure 32: design of a very simple memory cell based on two ring inverters (vsm-memorycell.sch)

The complete RAM design includes the address decoder (3 address lines to 8 select lines) and the circuit to read and write 8 bit data in the memory.

Conclusion

In this chapter, a very simple 4-bit microprocessor has been constructed, and 5 instructions have been implemented. This gives the foundations for building more complex processors with extended instruction set, more sophisticated exchanges between the main memory and the accumulators, more powerful arithmetic unit, in order to build a more attractive microprocessor.

References

[1] A. P. Malvino, J. A. Brown “Digital computer electronics”, Third Edition, Glenco-Macmillan, ISBN 0-02-800594-5, 1992, USA