University of Trento
Department of Physics

An FPGA-based Intel 8080 Mockup Soft Microprocessor

Leonardo Cattarin

Laboratory of Advanced Electronics
Professor: Ricci Leonardo

1 Introduction

Digital computers allow to store, process and transmit information under the form of binary numbers. The scope of this work is to synthesize a computing system inspired on the model of the Intel 8080 Microprocessor, using a Xilinx Spartan 3A.

2 Components and Implementation

The whole computing system is composed by three main modules: CPU, RAM and LCD debug modules and by various other auxiliary modules. The CPU comprises various 8-bit registers and uses them to fetch data from the RAM module, process it and, eventually, write back on the RAM. The LCD Debug driver allows to access both the bytes stored in the RAM and the CPU registers from the LCD screen. North and East pushbuttons are stabilized and used along with a switch to change the CPU registers or RAM values shown on the display.

Both the CPU and RAM run on three different clocks: a 50 MHz "Master" synchronization clock, a 1kHz debug clock used to comunicate with the LCD driver independently from the other operations and a last clock, sourced via the South pushbutton and used to govern the timing of the actual CPU and RAM operations.

The RAM is pre-loaded with the data and instructions during the FPGA configuration phase.

2.1 CPU Module

The CPU module behaves like a state machine which fetches and executes one instruction at time from the RAM in the form of single-byte operation codes ("opcodes"). If the instruction requires it, the CPU can also fetch other bytes as parameters. Each instruction execution is divided into several states, kept track by the `state` register. Various other

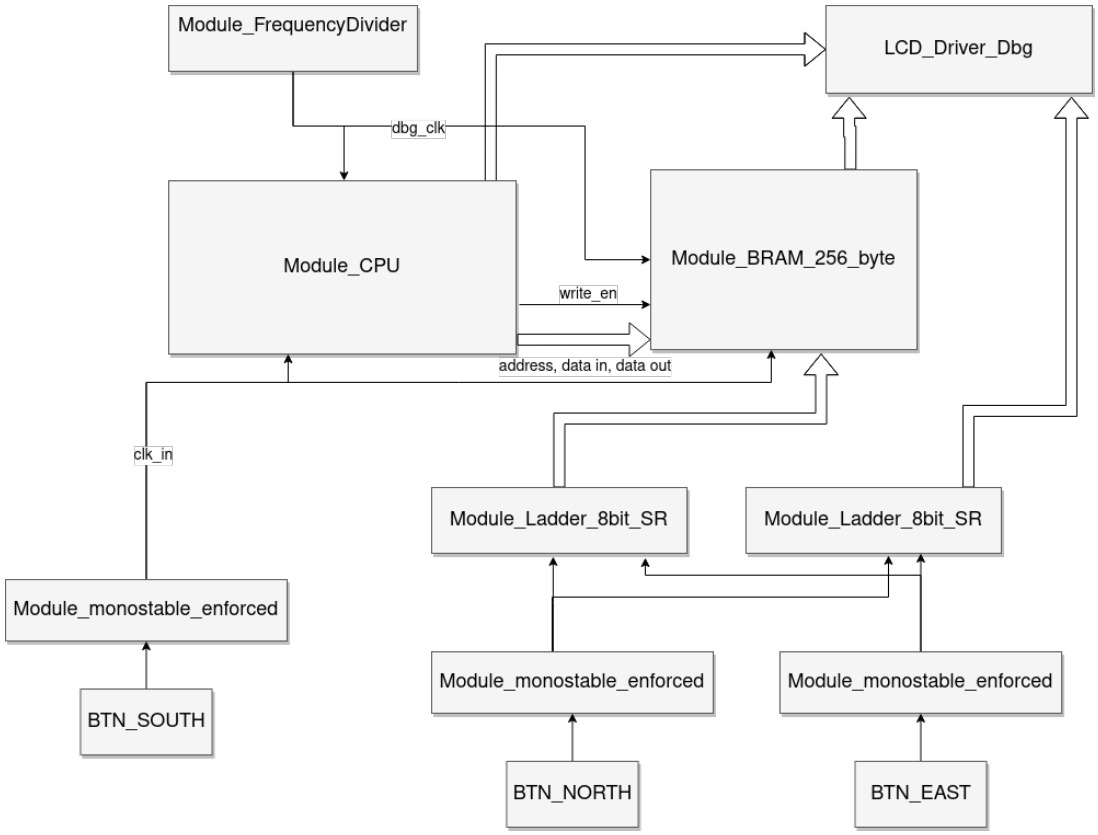

Figure 1: Modules scheme of the computing system

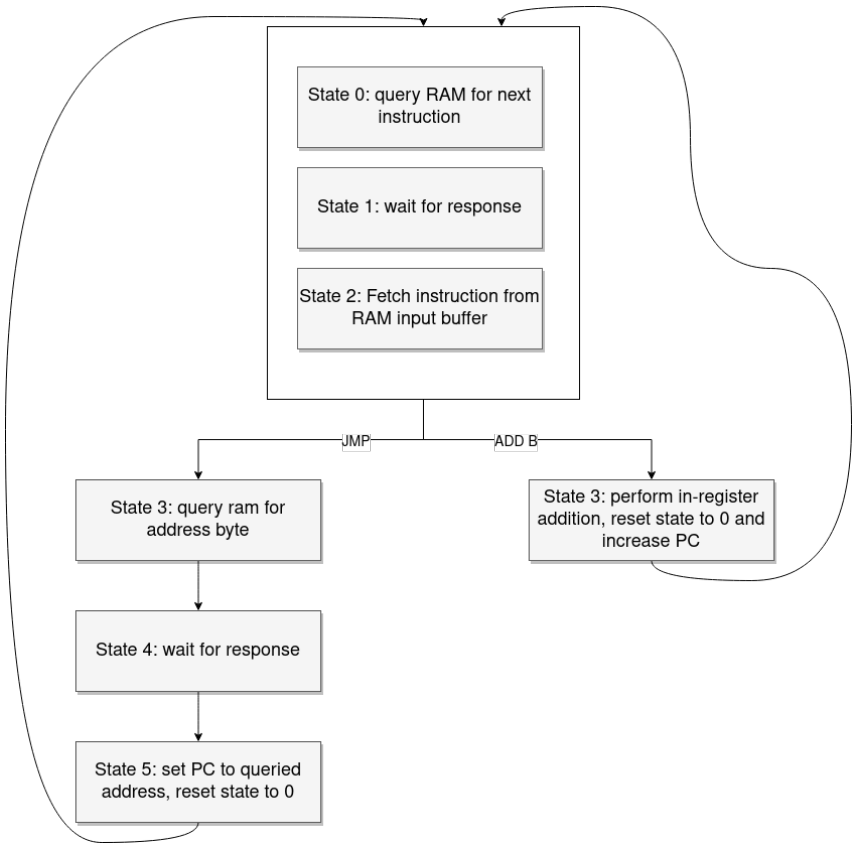

Figure 2: Fetch-execute cycle corresponding to JMP and ADD instructions

8-bit registers are present to help store, process and write data. At each `clk_in` edge risetime, the module behaviour is determined by the current instruction and state. When the last state corresponding to an instruction is reached, `state` register is reset to 0 and the process restarts by fetching the next instruction.

In general, the instruction fetch-execution cycle behaves as follows:

- `state` 0-2: The next instruction is fetched from the memory location pointed by the Program Counter (`PC`) and placed in the Instruction Register (`IR`).
- `state` 3- : The instruction is executed, eventually by reading or writing other bytes from memory or by operating on the internal registers. At the end, the `PC` address is incremented by one and `state` is reset to 0. The cycle then restarts

The following is a list of the internal CPU module registers:

- `PC` : (8-bit) Program Counter, contains the memory address to the next instruction.
- `IR` : (8-bit) Instruction Register, Contains the byte corresponding to the current instruction in execution
- `A` : (8-bit) Accumulator, Register used to store fetched data and used by default as result recipient by the ADD instruction
- `B,C` : (8-bit) General purpose registers
- `W,Z` : (8-bit) Temporary registers for internal instruction operations, not directly accessible via instructions
- `H,L` : (8-bit) Registers used to store memory addresses, for instance used to write from register to referenced memory location
- `data_addr,data_out,write_en` : Registers connected to output wirebuses, used to communicate read/write informations to RAM
- `flg_carry,flg_sign,flg_zero,flg_parity,flg_auxiliary` : Various flags registers used to keep track of the results of the instructions. `flg_auxiliary` is used as general-purpose flag to reduce the number of required instructions.

As a side note, the input/output operations usually require additional "buffering" times (blank states) to allow address informations delivery or data fetches.

2.2 RAM Module

The `Module_BRAM_256_byte` module is based on the usage of the "Block Ram" modules present in the Xilinx Spartan 3A. From the logical and Verilog code points of view, the Block RAM cells can be declared as arrays of n-bit registers. In this specific module, the Block ram used is organized as an array of 256 8-bit memory cells, corresponding to a memory with 8-bit addresses. The memory content is initialized using the Verilog command:

```
initial
begin
$readmemh("memory.data", RAM, 0, 255);
end
```

which prescribes how to load the RAM content from the file `memory.data` during the device configuration phase.

At each `clk_in` tick, the RAM either reads or writes the content at address `addr` according to `write_en` by using the wires/registers `data_in` , `data_out`

2.3 LCD driver Module

The `LCD_Driver_Dbg` module is used to visualize the content of both the RAM module and the internal CPU registers. The informations to be shown on display are inputed via the `addrInput, dataInput, CPU_interface, dbg_reg` wirebuses. The user can select via switch one of the two devices, and further use the North and East buttons to move between the memory addresses or CPU registers (implemented via a "Ladder" counter). These informations are then passed to the here considered module via `addrInput` and `dbg_reg` .

The LCD screen comprises a display data RAM used to store the shown characters, and communicates with the FPGA board via the following lines:

- `LCD_DB` : an 8-bit "Data bit" wirebus. Here is used in 4-bit mode, therefore the lower 4 bits are set to high. 4-bit mode requires to perform all the operations by sending separately the upper and lower "nible" of each 8-bit word.
- `LCD_E` : single-bit Read/write enable pulse, used to comunicate to the display that the other buses are ready for operating.
- `LCD_RS` : single-bit register select, set to 1 only during display memory operations.
- `LCD_RW` : read/write control, here permanetly set to 0 for write-only mode

Operating the LCD screen via Driver requires the following steps:

- Power-on initialization: initialization phase, the data bits are set and `LCD_E` is pulsed according to required patterns and timings.
- Display configuration: Here multiple commands are issued to configure the device:
  - Function set
  - Entry mode set: address counter is set to auto-increment
  - Display on, cursor and blinking are set off
  - Clear Display

  After issuing such commands, some time must be waited to allow the screen to function properly.
- Character writing: finally a cycle of character writing to the display RAM begins. For these commands, the `LCD_RS` must be set to 1 and `LCD_RW` to 0. Each character is represented by an 8-bit vlaue, and it must be transferred as indicated before, by sending first the upper nimble ant then the lower nimble. The writing address is then reset to the beginning and the process restarts.

The final output is:

```
RM AA XX
```

for the RAM debug mode, where AA stands for the 8-bit address and XX the relative value. All bytes are represented in hexadecimal notation.

Foir the CPU debug the output is:

```
CP RG XX
```

where RG stands for a register name, and XX is the content.

Disclaimer: the LCD driver module here used is based on a modified version of code not originally writtend by the author.

3 Instruction Set and Code Development

3.1 Instructions

The Insctructions implemented are freely based on the ones of the Intel 8080. To maintain the number of instruction as low as possible, many reduntant commands were removed. For instance, only `MOV` commands from and to the `B` register were implemented. Therefore, to perform copy operations, one needs to use only such instructions.

Similarly, to avoid the implementation of many different conditional jumps, a single `JC` conditional jump is implemented, based on the verification of the `flg_auxiliary` . To substitute the other checks, other commands where implemented for the specific purpose of setting the auxiliary flag according to the oter flags' values, such as `CHZ` (check zero).

It should be noted that the choice of a reduced instruction set yields to longer and more complex codes.

The following is the complete list of the instructions implemented:

- `NOP` , do nothin and fetch next instruction
- `JMP XX` , jump directly to address XX
- `JC XX` , jump conditionally to XX (if `flg_auxiliary` =1)
- `MVI B, XX` , copy immediately byte XX to B
- `MOV B,A` , `MOV B,C` , `MOV C,B` , `MOV B,H` `MOV H,B` , `MOV B,L` , `MOV L,B` , `MOV M(H),B` , `MOV B,M(H)` , various copy operations to/from B register (acts as a gateway)
- `ADD B` , `ADD M[H]` , Add to content of Accumulator A and save result in A
- `CMP B` , compare A with B. If A=B, set zero flag to 1. Otherwise set it to zero, and set carry flag to zero if A>=B or 1 otherwise
- `CHC` , `CHZ` , `CHS` , `CHP` , copy content of the various flag reghisters to the auxiliary one.
- `HLT` , do noting and stop, no more next instruction fetching.

3.2 Example: For Loop

The following example loops over various instructions to generate the natural numbers up to a certain value `UPPER_LIMIT` (excluded) and write them on memory starting from a chosen location. The idea is to fetch from memory a stored value of a `COUNTER` (which can be freely initialized), execute "loop" code, increase the counter and perform a

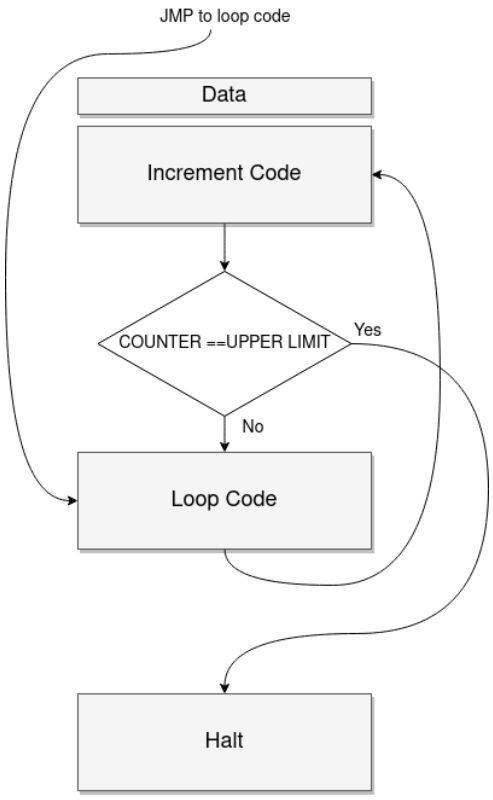

Figure 3: Loop Example Scheme

comparison with an `UPPER_LIMIT` . If the counter value is different than the limit, the code enters again the loop. Otherwise, it jumps directly to a halt command, and the program stops. The loop code fetches an initial address `NUMBERS_ADDR` and uses it to write the current counter value to the address calculated as `NUMBERS_ADDR + COUNTER` . In principle, this section could also be substituted by a subroutine call, allowing to implement more complicated operations.

The code is here represented first in pseudo-assembly, and then in explicit Hexadecimal form.

```
//jump directly to loop instructions
JMP PTR[LOOP]

//initial counter value, upper limit and memory location to write numbers.
COUNTER = 0
UPPER_LIMIT = 0A
NUMBERS_ADDR = 35

//code section for counter increment and conditional jump
#INCREMENT#

//load counter address in H
MVI B,PTR[COUNTER]
MOV H,B

//put 01 in B, use it to increase counter by 1, put result in B
```

```
MVI B, 01
ADD B
MOV B,A

//write counter to its memory position
MOV M[H],B

//load upper limit memory location in B, and then the limit itself
MVI B,PTR[UPPER_LIMIT]
MOV H,B
MOV B,M[H]

//compare A (counter) with B(upper limit)
//if A=B, zero flag is set to 1
CMP B

//check if zero flag is 1 (A=B) and jump to end program if true
CHZ
JC PTR[AFTER_LOOP]

#LOOP#
//puts counter address in H
MVI B,PTR[COUNTER]
MOV H,B

//puts counter in B, A and C
MOV B, M[H]
MOV A,B
MOV C,B

//load starting address for numbers writing
MVI B,PTR[NUMBERS_ADDR]
MOV H,B
MOV B,M[H]

//calculates address to write counter value, put in H
ADD B
MOV B,A
MOV H,B

//re-load counter value in B and A from C and write it to memory
MOV B,C
MOV A,B
MOV M[H],B

//jump to increment routine
JMP [INCREMENT]
```

```
#AFTER_LOOP#
HLT

#NUMBERS_ADDR#
```

This is the initial memory state:

00	C3	15	00	0A	35	06	02	60	06	01	80	47	70	06	03	60
10	46	B8	CC	DA	27	06	02	60	46	78	48	06	04	60	46	80
20	47	60	41	78	70	C3	05	76	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
...																
F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

While the following is the memory state at the end of the program execution (when the halt `HLT` instruction is reached):

00	C3	15	00	0A	35	06	02	60	06	01	80	47	70	06	03	60
10	46	B8	CC	DA	27	06	02	60	46	78	48	06	04	60	46	80
20	47	60	41	78	70	C3	05	76	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	01	02	03	04	05	06	07	08	09	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
...																
F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

As it can be seen, after the executon, the values `0x00, 0x01, .., 0x09 < 0x0A` have been written in memory starting from address `0x35` , as expected.

4 Conclusion

A computing system inspired by the Intel 8080 was syntethized using a Xilinx Spartan 3A, and a simple Loop Example was developed and tested.

5 Appendix: Processed Pseudo-Asm

The following pseudo-asm has been commented with the scope of illustrating how the various instructions are mapped to binary memory form. The curl brackets { } represent the memory addresses of the positions occupied by the instructions/data, the brackets ( ) contain the number of bytes occupied and the square [ ] contain the hexadecimal values corresponding to instructions and data.

```
//jump directly to instructions
{00 01}JMP PTR[LOOP] (2) [C3 15]

//memory location of counter value and initial address for number writing
{02}COUNTER = 0 (1) [00]
```

```
{03}UPPER_LIMIT (1) [0A]
{04}NUMBERS_ADDR (1) [35]

//increment routine
#INCREMENT# {05}
//load in H counter address
{05 06}MVI B,PTR[COUNTER] (2) [06 02]
{07}MOV H,B (1) [60]

//increase counter by 1, put it in B (and A)
{08 09} MVI B, 01 (2) [06 01]
{0A} ADD B (1) [80]
{0B} MOV B,A (1) [47]

//write counter to its memory position
{0C}MOV M[H],B (1) [70]

//load upper limit in B
{0D 0E}MVI B,PTR[UPPER_LIMIT] (2) [06 03]
{0F}MOV H,B (1) [60]
{10}MOV B,M[H] (1) [46]

//compare A (counter) with B(upper limit)
{11}CMP B (1) [B8]
//check if zero flag is 1 (A=B) and jump to end program if true
{12}CHZ (1) [CC]
{13 14}JC PTR[AFTER_LOOP] (2) [DA 27]

#LOOP# {15}
//puts counter value in B, A and C
{15 16}MVI B,PTR[COUNTER] (2) [06 02]
{17}MOV H,B (1) [60]
{18}MOV B, M[H] (1) [46]
{19}MOV A,B (1) [78]
{1A}MOV C,B (1) [48]

//load initial address for writing
{1B 1C}MVI B,PTR[NUMBERS_ADDR] (2) [06 04]
{1D}MOV H,B (1) [60]
{1E}MOV B,M[H] (1) [46]

//calculates address to write counter value, put in H
{1F}ADD B (1) [80]
{20}MOV B,A (1) [47]
{21}MOV H,B (1) [60]

//re-load counter value in B and A from C and write it to memory
{22}MOV B,C (1) [41]
```

```
{23}MOV A,B [78]
{24}MOV M[H],B (1) [70]

//jump to increment routine
{25 26}JMP [INCREMENT] (2) [C3 05]

#AFTER_LOOP# {27}
{27}HLT (1) [76]

#NUMBERS_ADDR#{35}
```