# Communication Methods in Digital Systems Lab. 1

# Shared Bus System based on Z80 Core

## Introduction

This laboratory is intended to illustrate a process of using an HDL Core of the microprocessor Z80 (TV80) for modeling and programming the system. Using HDL simulation gives us a unique opportunity to observe the behavior of the microprocessor at different abstraction levels. At the hardware level, we can observe the sequence of signals during the execution of an instruction. At the instruction level or the program level the execution of instruction sequence is observed. Recorded waveform trace from simulation allows observing the sequence of data flow across the microprocessor.

Our goal is to observe the communication protocol between the system components. Later we replace the microprocessor core with a bus functional model. The BFM elementary interface cycles generated by the Z80 using behavioral constructs. This approach significantly simplifies the simulated model complexity and delivers the environment that simplifies the design and verification of developed peripheral systems

The TV80 microprocessor core is shown in Fig. 1. In the fisrt step we will create an environment that allows us to interact with it.

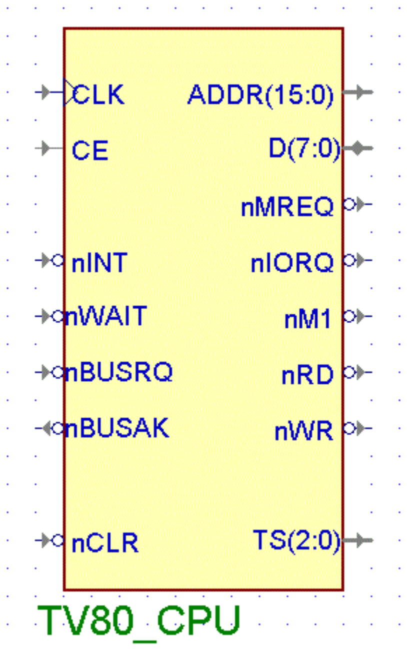

Fig. 1. The TV80 (Z80 equivalen core) schematic symbol

There are 6 different interface cycles that should be observed that are: FETCH, MEMRD, MEMWR, IOR, IOW, and INTERRUPT. It should be noted that the WAIT signal influences the interface cycle timing.

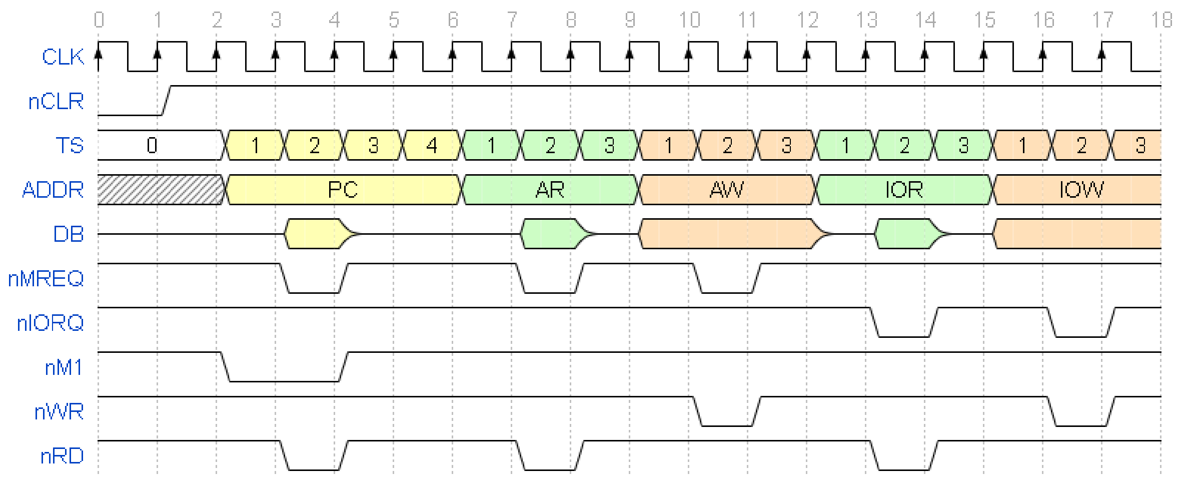

Fig. 2. The TV80 basic interface cycles

## Building a TV80 microprocessor system

The basic microprocessor system requires attaching at least random access memory (von Neuman architecture) where the program and data are stored. The memory for simulation purposes does not have to meet synthesis requirements. It is enough to meet the behavioral requirements of the microprocessor system. In the example there is defined memory that models behavior of typical level-triggered memory modules:

```
module RAM_4K(A, nOE, nWE, nCS, DQ);

input [11:0] A;
input nOE, nWE, nCS;
inout [7:0] DQ;

reg [7:0] MEM [4095:0];

always @(nOE or nWE or nCS or DQ or A)
    if(~nCS & ~nWE & nOE)
        MEM[A] = DQ;

assign DQ = (~nCS & nWE & ~nOE) ? MEM[A] : 8'hzz;

endmodule
```

The essential goal is to build a memory module that enables easy preparation of test rograms. For this purpose, there is defined as a set of tasks that behavior is similar to the assembly language compiler. There are the following tasks defined for direct controlling of memory content:

```
task WR_MEM;
task WR_MEM_W;
task ORG_MEM;
task INIT_MEM;
```

Before memory can be used it is necessary to initialize its content using the INIT_MEM task. This task set all memory cells to the given initial value. It can be compared with memset function in C. To mimic the behavior of assemblers there is a defined current write address. It is initialized by INIT_MEM to point 0. It can be moved to any valid location inside the memory using ORG_MEM task. All the memory write task will continue from the new location. The WR_MEM writes an 8-bit word at the current location and increments the current address. The WR_MEM_W write the 16-bit word in a little-endian fashion and adjust the current address (adds 2). Those functions exactly follow the translation of instructions.

## Verilog based inlined assembley language

During the observation of the system behavior it is convenient to design and assemble language based memory content generator. It is not simple to generate instructions from numerical pattersn but rather we expect using symbolic neamse instead that are easy to understand.

The instruction symbolic names definition is gathered in 8085_instr_set.v. There are defined mnemonics and respective operation codes. This idea significantly simplifies building the program. One should be aware that there are only operation codes that require the proper forming of arguments. Depending on instruction there is a need for an additional byte or a word to be defined after operation code. An exemplary fragment of the program is written as follows:

```
    M0.WR_MEM(`LXI_SP); M0.WR_MEM_W(16'h800); //Initialize stack
    M0.WR_MEM(`MVI_A); M0.WR_MEM(8'd12);
    M0.WR_MEM(`DAD_H); //RES = 2RES
    M0.WR_MEM(`RLC);
```

Here we have examples of instructions that require arguments like LXI SP,<arg16> or MVI A,<arg8>, or do not need arguments like DAD H or RLC.

The assembly compilation is based on two passes because not all information can be gathered after the first pass. The problem is jump instruction (conditional and simple) to the locations that follow the current instruction. At the moment of analysis, the destination address is unknown except of its symbolic name. The destination address is discovered after encountering the label definition

[<label_name>:]. To enable the implementation of jumps a set of tasks handling labels is defined. There are the following tasks

```
task INIT_LABEL;
task WR_LABEL;
task ASSIGN_LABEL;
task UPDATE_LABEL;
```

The idea of using those tasks is a combination of high-level language syntax with operations of assembler language. The Verilog language does not allow the automatic definition of variables different than wire. It is required to define all labels as 24-bit registers. Before placing a program statement all labels must be initialized using INIT_LABEL task coll for each label definition:

```
M0.INIT_LABEL(MUL_SUB);
```

There is a limit of 128 labels that completely satisfies our experiments.

Each time the instruction refers to the label the WR_LABEL task is used:

```
M0.WR_MEM(`JNC); M0.WR_LABEL(MUL_SA);
```

When the label definition is encountered the ASSIGN_LABEL task is used:

```
M0.ASSIGN_LABEL(MUL_SUB); //MUL_SUB:
```

After defining all instructions the labels addresses update should be requested

```
M0.UPDATE_LABEL(MUL_SUB);
```

The UPDATE_LABEL replace symbolic names with addresses discovered during the compilation process of assembly statements.

The exemplary short program can be put down in the following form

```
reg [23:0] MAIN_STOP;
initial begin
    M0.INIT_MEM(8'd0);
    M0.INIT_LABEL(MAIN_STOP);
    M0.WR_MEM(`LXI_SP); M0.WR_MEM_W(16'h800); //Initialize stack
    M0.ASSIGN_LABEL(MAIN_STOP); //MAIN_STOP:
    M0.WR_MEM(`JMP); M0.WR_LABEL(MAIN_STOP);
    M0.UPDATE_LABEL(MAIN_STOP);
end
```

In order to separate the program from the main body of the testbench it can be included using the following statement:

```
//----------------------------------------------------------
//  Included program
//----------------------------------------------------------
`include "tv80_mul.v"
```

Now you can try implementing other programs and see how it is executed.

## Tasks

1. Create a simple memory model and discover all bus cycles using short test programs. Use the include file 8085_instr_set.v for symbolic names of instructions. Triggering interrupt requires creating additional logic that asserts the nINT line. Evaluate the functionality of nWAIT, nBUSRQ, and nBUSAK.
2. Based on the discovered properties of interface cycles build the bus functional model of the CPU. The BFM should implement tasks enabling the functionality of all interface cycles named after its operation e.g. FETCH.
3. Implement a synthesizable 8-bit bidirectional programmable input-output port. Verify its operation correctness using developed BFM tasks. Check response correctness in all situations.
4. Create a 16-bit periodic timer with interrupt notifications. Verify its operation
5. Create an arbitration unit to the common memory block (1kB). Control access flow using the nWAIT signal. Implement constant and round-robin priority. Verify functionality using BFM tasks.