Detailed hardware description of i8051 microcontroller and analysis of ROM/RAM memory overflow. Simple example of external memory usage

- Pietro Alfonsi
- Edgardo Vittoria
- Giannantonio Frittella
- Andrea Rofini

i8051 Hardware Description

VHDL files provide an hardware description of the i8051 microcontroller, simulating the tasks of all the main components, like accumulator, CPU, RAM. We will analyze these files one by one.

i8051_lib.vhd

Here are defined common used constants, associated with predefined strings of bits. First of all, these are defined:

```
constant CD_16    : UNSIGNED (15 downto 0) := "----------------";
constant CD_12    : UNSIGNED (11 downto 0) := "------------";
constant CD_11    : UNSIGNED (10 downto 0) := "-----------";
constant CD_8     : UNSIGNED ( 7 downto 0) := "--------";
constant C0_8     : UNSIGNED ( 7 downto 0) := "00000000";
constant C1_8     : UNSIGNED ( 7 downto 0) := "00000001";
constant C7_8     : UNSIGNED ( 7 downto 0) := "00000111";
constant CM_8     : UNSIGNED ( 7 downto 0) := "11111111";
constant CD_7     : UNSIGNED ( 6 downto 0) := "-------";
constant C0_7     : UNSIGNED ( 6 downto 0) := "0000000";
constant C9_4     : UNSIGNED ( 3 downto 0) := "1001";
```

The name of the constants is structured in this way: the first `C` stands for *Constants* , followed by `D` if it is a *don't care* value, `0` if it is all zeros, `M` if it is all ones and a specific number if it is needed. Then it is followed by the number of bits, after the `_` .

Then we have Special Function Registers:

```
constant R_B    : UNSIGNED (7 downto 0) := "11110000";
constant R_ACC  : UNSIGNED (7 downto 0) := "11100000";
constant R_PSW  : UNSIGNED (7 downto 0) := "11010000";
constant R_IP   : UNSIGNED (7 downto 0) := "10111000";
constant R_IE   : UNSIGNED (7 downto 0) := "10101000";
constant R_SP   : UNSIGNED (7 downto 0) := "10000001";
constant R_P0   : UNSIGNED (7 downto 0) := "10000000";
constant R_P1   : UNSIGNED (7 downto 0) := "10010000";
constant R_P2   : UNSIGNED (7 downto 0) := "10100000";
constant R_P3   : UNSIGNED (7 downto 0) := "10110000";
```

```
11   constant R_DPL  : UNSIGNED (7 downto 0) := "10000010";
12   constant R_DPH  : UNSIGNED (7 downto 0) := "10000011";
13   constant R_PCON : UNSIGNED (7 downto 0) := "10000111";
14   constant R_SCON : UNSIGNED (7 downto 0) := "10011000";
15   constant R_SBUF : UNSIGNED (7 downto 0) := "10011001";
16   constant R_TCON : UNSIGNED (7 downto 0) := "10001000";
17   constant R_TMOD : UNSIGNED (7 downto 0) := "10001001";
18   constant R_TL0  : UNSIGNED (7 downto 0) := "10001010";
19   constant R_TL1  : UNSIGNED (7 downto 0) := "10001011";
20   constant R_TH0  : UNSIGNED (7 downto 0) := "10001100";
21   constant R_TH1  : UNSIGNED (7 downto 0) := "10001101";
```

`UNSIGNED (7 downto 0)` means that it is a 8-bit unsigned integer, with the 7th-bit being the most significant bit and the 0th the least significant.

Followed by the Instruction Set:

```
1   constant ADD_1  : UNSIGNED (4 downto 0) := "00101";
2   constant ADD_2  : UNSIGNED (7 downto 0) := "00100101";
3   ...
4   constant JMP    : UNSIGNED (7 downto 0) := "01110011";
5   ...
6   constant POP    : UNSIGNED (7 downto 0) := "11010000";
7   constant PUSH   : UNSIGNED (7 downto 0) := "11000000";
```

Then, all the operations supported by the ALU are defined:

```
 1   constant ALU_OPC_NONE   : UNSIGNED (3 downto 0) := "0000";
 2   constant ALU_OPC_ADD    : UNSIGNED (3 downto 0) := "0001";
 3   constant ALU_OPC_SUB    : UNSIGNED (3 downto 0) := "0010";
 4   constant ALU_OPC_MUL    : UNSIGNED (3 downto 0) := "0011";
 5   constant ALU_OPC_DIV    : UNSIGNED (3 downto 0) := "0100";
 6   constant ALU_OPC_DA     : UNSIGNED (3 downto 0) := "0101";
 7   constant ALU_OPC_NOT    : UNSIGNED (3 downto 0) := "0110";
 8   constant ALU_OPC_AND    : UNSIGNED (3 downto 0) := "0111";
 9   constant ALU_OPC_XOR    : UNSIGNED (3 downto 0) := "1000";
10   constant ALU_OPC_OR     : UNSIGNED (3 downto 0) := "1001";
11   constant ALU_OPC_RL     : UNSIGNED (3 downto 0) := "1010";
12   constant ALU_OPC_RLC    : UNSIGNED (3 downto 0) := "1011";
13   constant ALU_OPC_RR     : UNSIGNED (3 downto 0) := "1100";
14   constant ALU_OPC_RRC    : UNSIGNED (3 downto 0) := "1101";
15   constant ALU_OPC_PCSADD : UNSIGNED (3 downto 0) := "1110";
16   constant ALU_OPC_PCUADD : UNSIGNED (3 downto 0) := "1111";
```

And finally, operations defined for the Decoder:

```
constant OPC_ADD_1  : UNSIGNED (6 downto 0) := "0000001";
constant OPC_ADD_2  : UNSIGNED (6 downto 0) := "0000010";
...
constant OPC_JMP    : UNSIGNED (6 downto 0) := "0101100";
constant OPC_JNB    : UNSIGNED (6 downto 0) := "0101101";
...
constant OPC_POP    : UNSIGNED (6 downto 0) := "1010101";
constant OPC_PUSH   : UNSIGNED (6 downto 0) := "1010110";
```

i8051_dbg.vhd

This file is used for debugging, as it prints every instruction executed in a *trace.out* file. It takes one input: `op_in`, the decoded instruction executed.

```
process(op_in)
        variable s : STRING (1 to 12);
        variable l : LINE;
        file f : TEXT is out "trace.out";
    begin
```

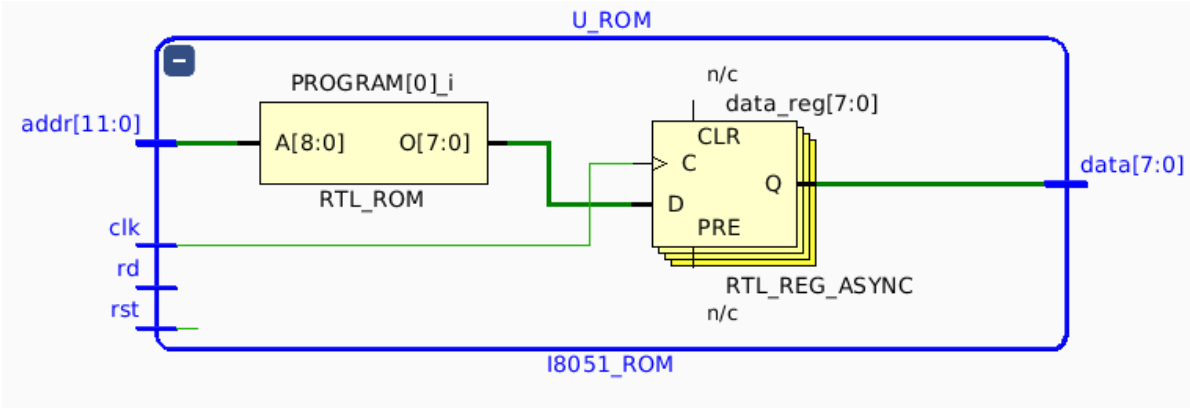
`s` is the string to be printed, `l` is the line in the file where to print and `f` is the output file. So we assign `s` to the decoded instruction:

```
case op_in(6 downto 0) is
...
    when OPC_ORL_1  => s := "ORL 1       "; opc <= ORL_1;
...
```

and then we print the line:

```
if( s /= "            " ) then
    write(l, s, LEFT, 7);
    writeline(f, l);
end if;
```

i8051_rom.vhd



This file models the 4 KB ROM of the i8051. Input signals are:

- `addr` : the memory address
- `clk` : clock signal
- `rd` : read operation
- `rst` : reset signal

And it outputs the data in the address.

It is generated automatically by the `i8051_mkr.c` file. This file takes the input HEX file, checks if it is valid and the fields are in the correct format. After that it writes the behavioral code of the .vhd file:

```
1   int main(int argc, char *argv[]) {
2
3       FILE *fh;
4       char buf[256];
5       unsigned char rom[4096];
6       unsigned prgSize = 0;
7
8       memset(rom, 0, sizeof(rom));
9       if( argc != 2 || (fh = fopen(argv[1], "r")) == 0 ) {
10
11          printf("file argument invalid\n");
12          exit(1);
13      }
14
15      while( !feof(fh) ) {
16
17          fgets(buf, sizeof(buf), fh);
18          if( Load(buf, rom, sizeof(rom), &prgSize) ) break;
19      }
20
21      WriteVhdl(rom, prgSize, argv[1]);
22      return 0;
23  }
```

The main operation is at line 21, `WriteVhdl` . This function will compose the right VHDL file, in this way:

```
1   void WriteVhdl(unsigned char *rom, unsigned prgSize, const char *fileName) {
2
3       FILE* fh = fopen("i8051_rom.vhd", "w");
4       char *binary;
5       unsigned i;
6
7       assert( fh != 0 );
8
9       fprintf(fh, "-- %s \n\n%s", fileName, RomHeader);
10      fprintf(fh, "    type ROM_TYPE is array (0 to %i)", prgSize - 1);
11      fprintf(fh, " of UNSIGNED (7 downto 0);\n\n");
12      fprintf(fh, "    constant PROGRAM : ROM_TYPE := (\n\n");
13      for(i=0; i<prgSize-1; i++) {
14
15          binary = ToBinary(rom[i]);
16          fprintf(fh, "\t\"%s\",%s\n", binary, Comment(binary));
17      }
18      binary = ToBinary(rom[prgSize-1]);
19      fprintf(fh, "\t\"%s\");%s\n%s", binary, Comment(binary), RomFooter);
20  }
```

It opens the output file at line 3, writes the declarations in lines 9-12 (you can see at line 10 that the array which composes the ROM has dynamic size, based on the `prgSize` variable derived from reading the HEX file) and then binary conversions of the HEX instructions are written in the file in lines 13 to 17. Finally the footer (the behavioral code) is written.

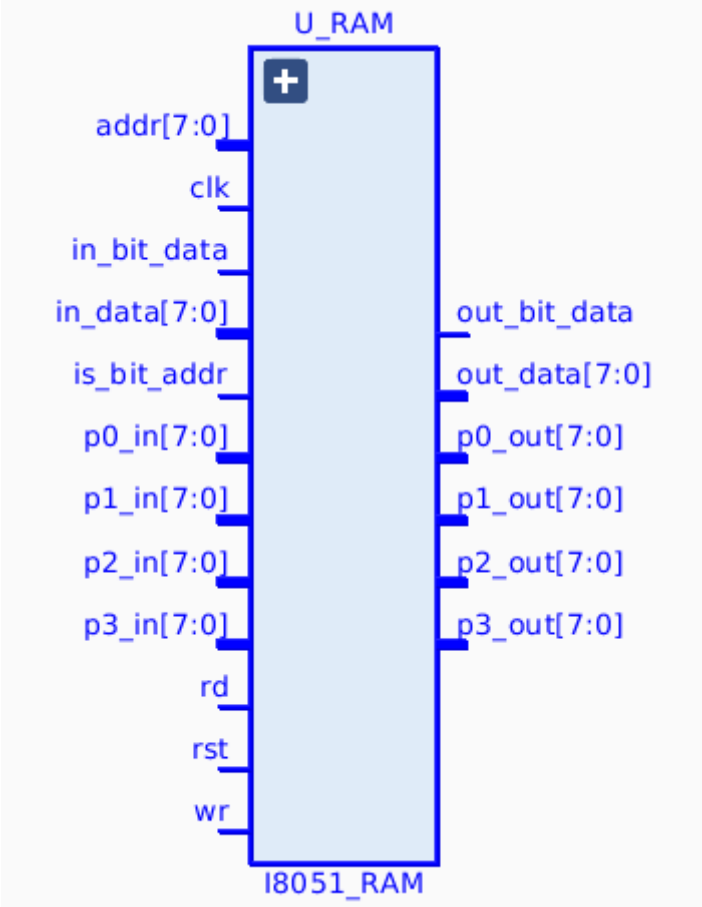
So we obtain all the instruction to execute the program:

```
1   type ROM_TYPE is array (0 to 399) of UNSIGNED (7 downto 0);
2   constant PROGRAM : ROM_TYPE := (
3   ...
4       "01111001", -- MOV_7
5       "00000000",
6       "11101001", -- MOV_1
7       "01000100", -- ORL_4
8       "00000000",
9       "01100000", -- JZ
10      "00011011",
11      "01111010", -- MOV_7
12  ...
```

At the end we have the code that drives the read of the the ROM lines, if the state is not `reset`, but in `read` and on a rising edge of the clock (lines 6-10), assigning the variable `data` to the integer associated to the binary instruction in order to retrieve the right instruction in the `PROGRAM` array:

```
process(rst, clk)
    begin
        if( rst = '1' ) then
                data <= CD_8;
        elsif( clk'event and clk = '1' ) then
            if( rd = '1' ) then
                data <= PROGRAM(conv_integer(addr));
            else
                data <= CD_8;
            end if;
        end if;
    end process;
```

i8051_ram.vhd



This file describes the i8051 RAM. It takes as input:

- `addr` : memory address to be requested
- `clk` : clock signal
- `rd` : read operation
- `wr` : write operation
- `rst` : reset signal
- `in_bit_data` : single bit to be written

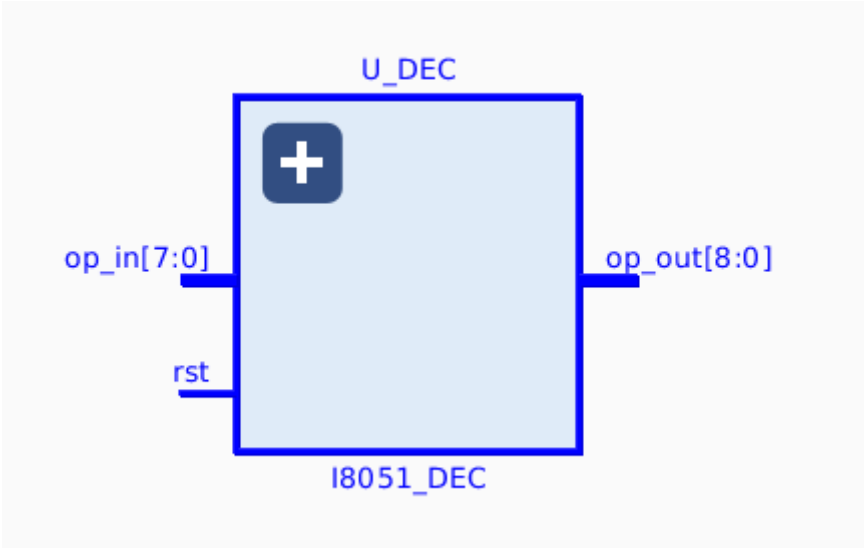
- `out_bit_data` : single bit to be read
- `is_bit_addr` : requesting only one-bit data
- `in_data` : single byte to be written
- `out_data` : single byte to be read
- `pX_in` : write access to the pX port (0,1,2, or 3)
- `pX_out` : read access to the pX port

RAM is implemented by reading the input signals and performing the right read/write operation:

```
1   if( rst = '1' ) then
2   ...
3   elsif( clk'event and clk = '1' ) then
4       if( rd = '1' ) then
5
6               if( is_bit_addr = '1' ) then
7
8                       GET_BIT(addr, v1);
9                       out_bit_data <= v1;
10              else
11
12                      GET_BYTE(addr, v8);
13                      out_data <= v8;
14              end if;
15        elsif( wr = '1' ) then
16
17              if( is_bit_addr = '1' ) then
18
19                      SET_BIT(addr, in_bit_data);
20              else
21
22                      SET_BYTE(addr, in_data);
23              end if;
24        end if;
25  end if;
```

Line 1 checks if a reset action is needed, that will put registers to 0 and RAM addresses data to *don't care* . If the `read` operation is needed, then it will check if it has to read a single bit or a byte, and then perform the operations `GET_BIT` or `GET_BYTE` and output the data. Analogously for the `write` operation.

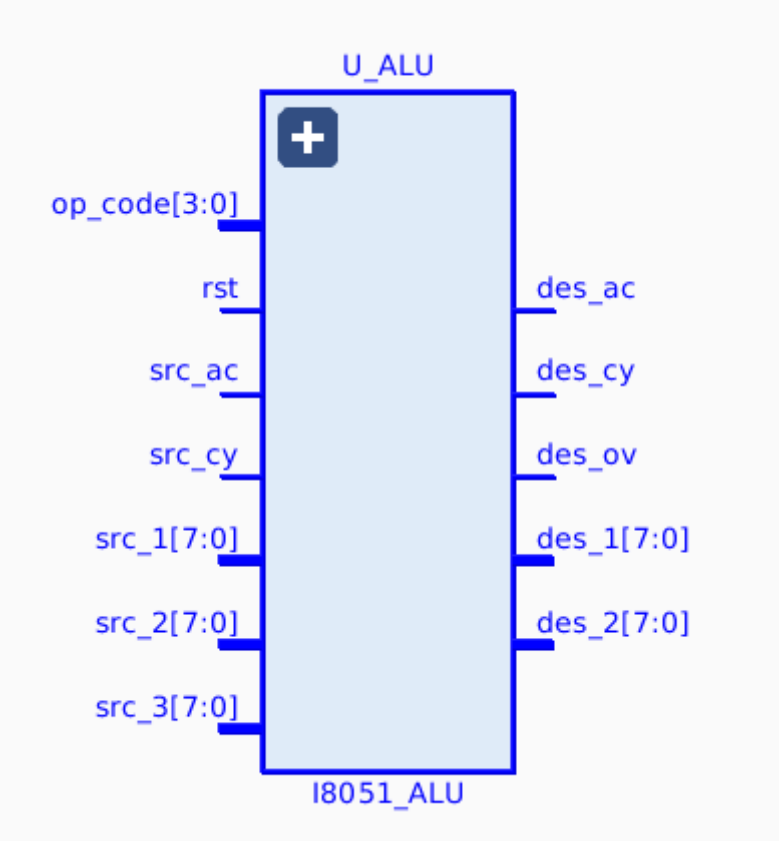
i8051_dec.vhd



This file implements the Decoder, which takes a variable-length instruction in the 8051 format and outputs a fixed-length instruction. It accomplishes the task in a combinational way, checking every case of the input instruction:

```
1   op_out <= ("00" & OPC_ERROR ) when rst = '1' else
2   ("01" & OPC_ACALL ) when op_in(4 downto 0) = ACALL  else
3   ("00" & OPC_ADD_1 ) when op_in(7 downto 3) = ADD_1  else
4   ("01" & OPC_ADD_2 ) when op_in(7 downto 0) = ADD_2  else
5   ...
6   ("11" & OPC_LCALL ) when op_in(7 downto 0) = LCALL  else
7   ("11" & OPC_LJMP  ) when op_in(7 downto 0) = LJMP   else
8   ("00" & OPC_MOV_1 ) when op_in(7 downto 3) = MOV_1  else
9   ("01" & OPC_MOV_2 ) when op_in(7 downto 0) = MOV_2  else
10  ("00" & OPC_MOV_3 ) when op_in(7 downto 1) = MOV_3  else
11  ...
```

i8051_alu.vhd



This file implements the ALU unit of the i8051, and its operations. Input signals are:

- `op_code` : code of the instruction to be performed, defined in the Lib file
- `rst` : reset signal
- `src_cy` , `des_cy` : carry in the 7th bit of the source/destination operand
- `src_ac` , `des_ac` : carry in the 4th bit of the source/destination operand
- `src_X` : first/second/third source operand of the alu
- `des_X` : first/second destination operand
- `des_ov` : overflow out of the alu

The file defines some internal functions to be used to compose the supported operations, like `DO_MUL` :

```
procedure DO_MUL(a, b : in UNSIGNED (7 downto 0);
                     r : out UNSIGNED (15 downto 0);
                     ov : out STD_LOGIC) is

variable v1 : UNSIGNED (15 downto 0);
begin
    v1 := a * b;
    r := v1;
    if( v1(15 downto 8) /= C0_8 ) then

        ov := '1';
    else

        ov := '0';
    end if;
end DO_MUL;`:
```

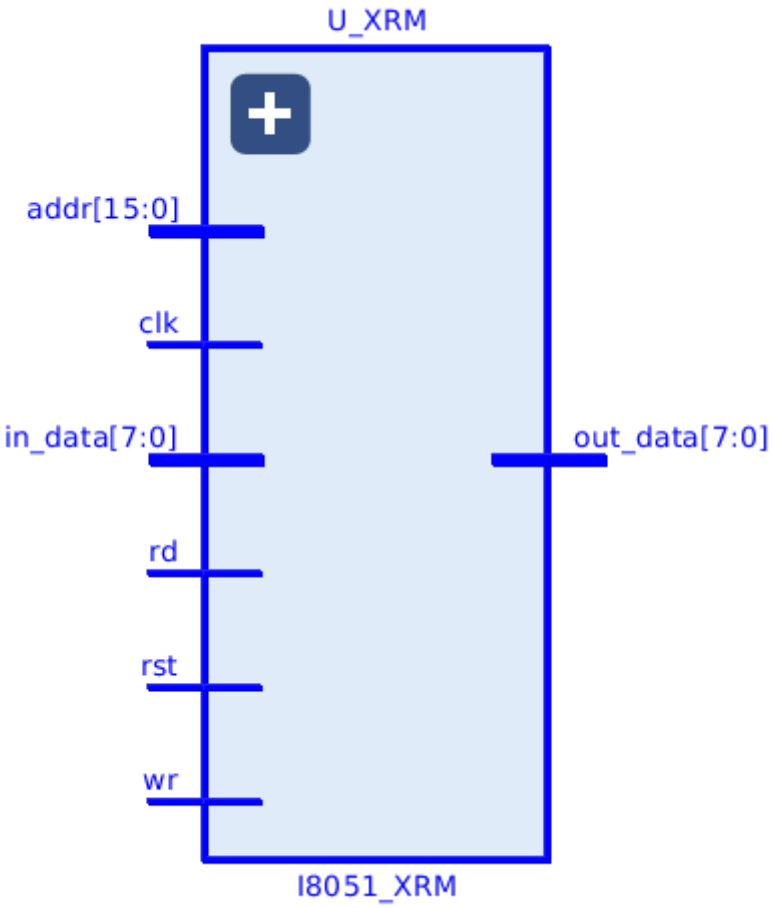
It performs multiplication between two numbers and checks for overflow, and it is used to implement the ALU operation `ALU_OPC_MUL`, the final multiplication between source operands:

```
case op_code is
...
            when ALU_OPC_MUL    =>
                    DO_MUL(src_1, src_2, v16, v_ov);
                    des_1 <= v16(7 downto 0);
                    des_2 <= v16(15 downto 8);
                    des_cy <= '-';
                    des_ac <= '-';
                    des_ov <= v_ov;
```

While `DO_MUL` performs the multiplication, `ALU_OPC_MUL` does the right destination port assignments to complete the operation.

Same thing happens for other operations supported by the ALU, e.g. addition and subtraction.

i8051_xrm.vhd



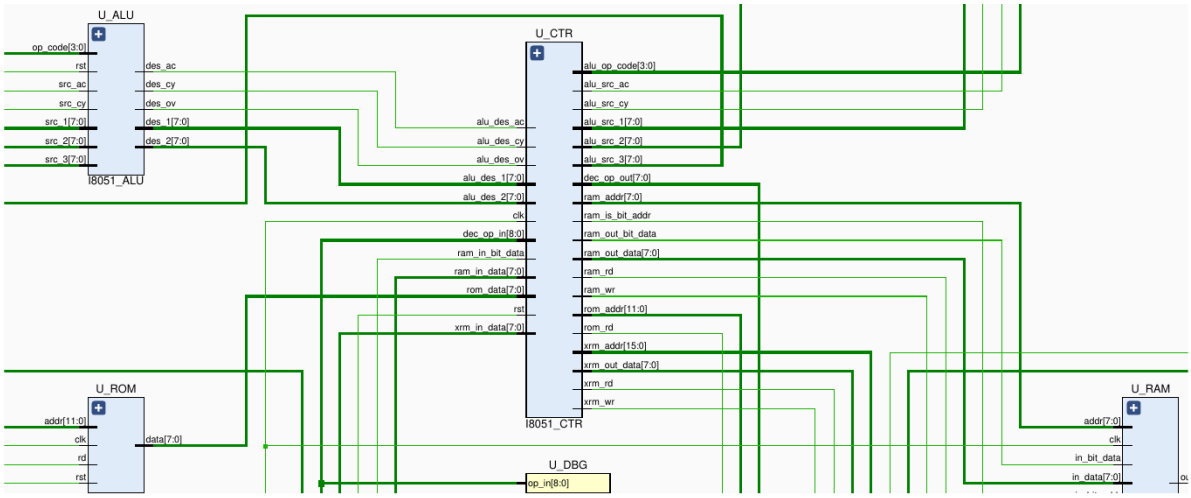
This file describes the external memory for the 8051. It takes a 16-bit address as input to identify one of the 64KB memory addresses, takes 8-bit input data to write and outputs 8-bit data when reading. Code is very similar to ROM:

```
1   process(rst, clk)
2       begin
3
4           if( rst = '1' ) then
5
6               for i in 0 to storage_size-1 loop
7
8                   xrm(i) <= CD_8;
9               end loop;
10
11              out_data <= CD_8;
12
13          elsif( clk'event and clk = '1' ) then
14
15              if( rd = '1' and conv_integer(addr) < storage_size ) then
16
17                  out_data <= xrm(conv_integer(addr));
18              elsif( wr = '1' and conv_integer(addr) < storage_size ) then
19
20                 xrm(conv_integer(addr)) <= in_data;
21              end if;
22          end if;
23      end process;
```

Line 6 is an addition with respect to the ROM file, as we reset also the data in the external memory when the `reset` signal is high. Also, when clock is high and a `read` or `write` is selected, we check if the address goes out of bounds of memory size. `xrm` identifies the right data in the memory, addressed by the conversion of the `addr` value:

```
1   type XRM_TYPE is array (0 to storage_size-1) of UNSIGNED (7 downto 0);
2   signal xrm : XRM_TYPE;
```

i8051_ctr.vhd



This file describes the processor of the i8051 microcontroller. As you can see from the image, this component is binded to all the others we have seen (RAM, ROM, external memory and ALU), as it is the starting point of every operation of the microcontroller. Input signals are:

- `rst` : reset signal
- `clk` : clock signal
- `ram_in_data` : data received from the RAM, after a request
- `ram_in_bit_data` : bit-data received from RAM
- `rom_data` : Data returned by ROM
- `xrm_in_data` : data returned from external memory
- `dec_op_in` fixed length operation code (type of instruction, it can be 6, 7 or 8 bit long), decoded by the Decoder.
- `alu_des_1` , `alu_des_2` , `alu_des_cy` , `alu_des_ac` and `alu_des_ov` : output data from ALU (operands, carry and overflow) as we saw in the ALU file

We can notice that input data for the processor is the output data from other components. These data are the ones returned after a request to either RAM, ROM, external memory or ALU. Output signals are all the input signals of the components we already saw previously. For example `rom_addr` to request a memory address in ROM, `dec_op_out` the variable length operation code sent to the Decoder, or `alu_src_1` for source operand in ALU and `alu_src_cy` for the carry in the 7th bit of the ALU.

The architecture is defined behaviorally, starting with the definition of CPU states:

```
1   type CPU_STATE_TYPE is (CS_0, CS_1, CS_2, CS_3);
```

Those are: *reset controller* , *handle all interrupts* , *process instructions* and *execute state* and 8 execution stages to complete the wanted operation.

Then some utility procedures are defined, like functions to read and write from RAM, increment Program Counter or interact with the ALU. For example, to read from RAM:

```
1   procedure START_RD_RAM (a : UNSIGNED (7 downto 0)) is
2           begin
3               ram_addr <= a;
4               ram_is_bit_addr <= '0';
5               ram_rd <= '1';
6               ram_wr <= '0';
7           end START_RD_RAM;
```

The processor checks first if a reset signal is high, so it resets all the registers. If not present, it looks for an high signal for the clock, and it will look at the CPU state (line 9):

```
1    if( rst = '1' ) then
2   ...
3   elsif( clk'event and clk = '1' ) then
4
5               STOP_RD_ROM;
6               STOP_RD_WR_RAM;
7               STOP_RD_WR_XRM;
8
9               case cpu_state is
```

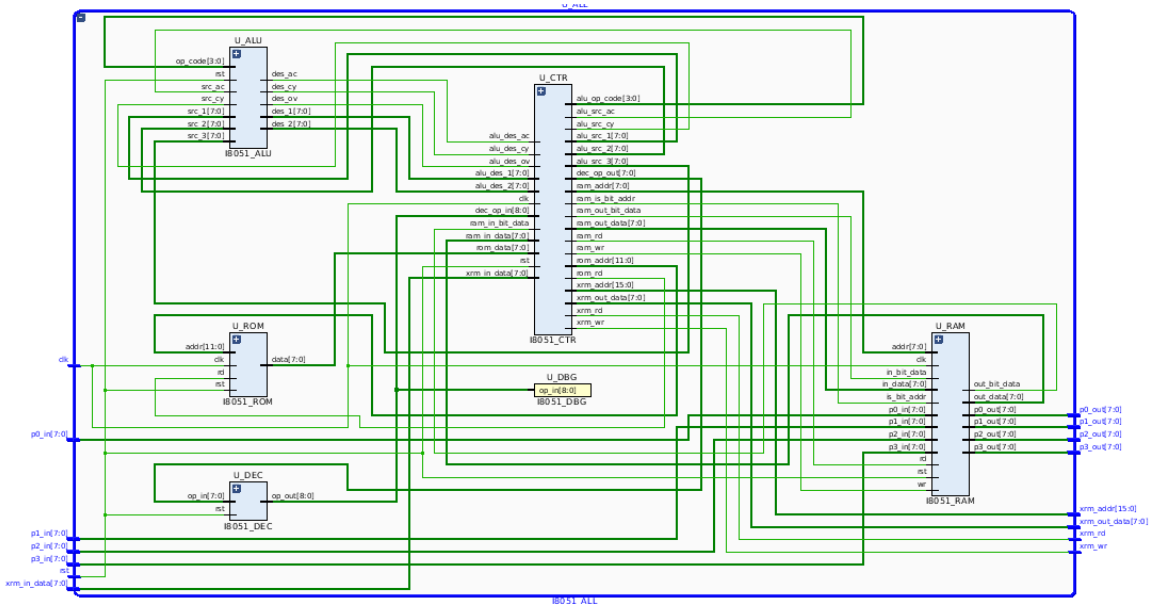
The main states are the `CS_2` and `CS_3`. In the first, the operation is fetched. In the second it is executed using all the execution states (example for `MOV1`):

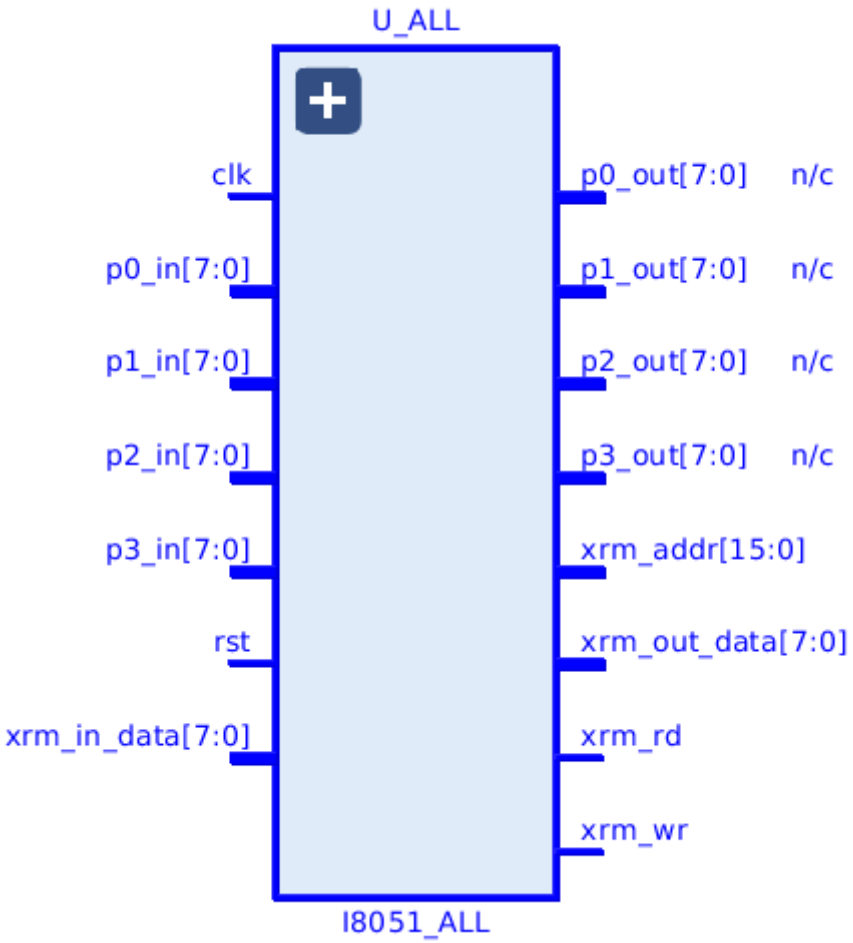
```
1    when CS_3 =>
2            case dec_op_in(6 downto 0) is
3   ...
4            when OPC_MOV_1  =>
5                                case exe_state is
6                                    when ES_0 =>
7                                        GET_RAM_ADDR_1(v8);
8                                        START_RD_RAM(v8);
9                                        exe_state <= ES_1;
10
11                                   when ES_1 =>
12                                       exe_state <= ES_2;
13
14                                   when ES_2 =>
15                                       ram_out_data <= ram_in_data;
16                                       START_WR_RAM(R_ACC);
17                                       exe_state <= ES_3;
18
19                                   when ES_3 =>
20                                       exe_state <= ES_4;
21
22                                   when ES_4 =>
23                                       exe_state <= ES_5;
```

```
24
25                                  when ES_5 =>
26                                      exe_state <= ES_6;
27
28                                  when ES_6 =>
29                                      exe_state <= ES_7;
30
31                                  when ES_7 =>
32                                      SHUT_DOWN_ALU;
33                                      cpu_state <= CS_1;
34                                      exe_state <= ES_0;
35                              end case;
```

In every execution state, registers can be assigned values and utility procedures can be called. Then the next execution state is called, until the last one where the CPU state is changed to `CS_1` (which is *interrupt handling* , not implemented in this solution, so it automatically switches to `CS_2`).

i8051_all.vhd





This file models the whole microcontroller, at an higher level of abstraction with respect to the other file we saw. Single components are defined only as inputs and outputs, defined as described previously (for example ALU):

```
component I8051_ALU
        port(rst     : in  STD_LOGIC;
             op_code : in  UNSIGNED (3 downto 0);
             src_1   : in  UNSIGNED (7 downto 0);
             src_2   : in  UNSIGNED (7 downto 0);
             src_3   : in  UNSIGNED (7 downto 0);
             src_cy  : in  STD_LOGIC;
             src_ac  : in  STD_LOGIC;
             des_1   : out UNSIGNED (7 downto 0);
             des_2   : out UNSIGNED (7 downto 0);
             des_cy  : out STD_LOGIC;
             des_ac  : out STD_LOGIC;
             des_ov  : out STD_LOGIC);
    end component;
```

Then ports are mapped to every component in order to create all the links between components (example for ALU):

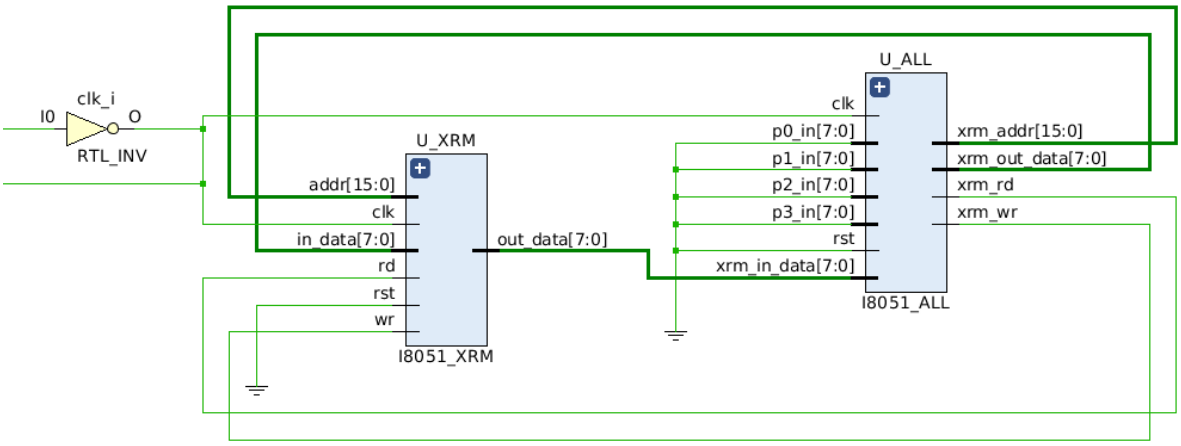
```
U_ALU : I8051_ALU port map(rst,
                           alu_op_code,
                           alu_src_1,
                           alu_src_2,
                           alu_src_3,
                           alu_src_cy,
                           alu_src_ac,
                           alu_des_1,
                           alu_des_2,
                           alu_des_cy,
                           alu_des_ac,
                           alu_des_ov);
```

Input signals of the whole component are:

- `rst` : reset signal
- `clk` : clock signal
- `pX_in` : input data for the four ports (P0, P1, P2 and P3)
- `pX_out` : output data of the four ports
- `xrm_in_data` , `xrm_out_data` , `xrm_rd` , `xrm_wrt` and `xrm_addr` define the signal to interact with the external memory, and decide if it is a read or write operation and the address requested.

Output signals are ports values (P0, P1, P2 and P3) and external memory address, data and operation.

i8051_tsb.vhd



This is the testbench file, which contains one 8051 microcontroller and one external memory. This sits at an even higher level of abstraction and it simulates the whole system. Components are defined with their signals (example for external memory):

```
component I8051_XRM

        generic(storage_size : INTEGER := 2048);

        port(rst      : in  STD_LOGIC;
             clk      : in  STD_LOGIC;
             addr     : in  UNSIGNED (15 downto 0);
             in_data  : in  UNSIGNED (7 downto 0);
             out_data : out UNSIGNED (7 downto 0);
             rd       : in  STD_LOGIC;
             wr       : in  STD_LOGIC);
    end component;
```

This implementation does not support cycle/timing, so it is manually set with a period of 41.6 ns, a little slower than the real 8051 microcontroller:

```
rst <= '0' after 50 ns;
clk <= not clk after 41.6 ns;
```

Finally ports are mapped:

```
U_XRM : I8051_XRM port map (rst, clk, addr, out_data, in_data, rd, wr);
```

At this point, the whole system is composed and execution of user programs can be simulated.

Exceeding ROM size

The 8051 microcontroller has an internal 4KB ROM, as we saw previously. Program size can exceed this size. This limit is checked when generating the ROM file by the `i8051_mkr.c` file. The first step to generate the ROM file is to load the input HEX file. Then, it will check every line of the input file to see if it's a valid HEX record, and loads the *data* field of the record into the ROM. While doing it, it will check for the program size reached thus far:

```
if( (*prgSize) >= romSize || (base + len) >= romSize ) {

        printf("program too large\n");
        exit(1);
    }
```

For every line of the HEX file, it is checked whether the Program Size (`prgSize`) or if the base address and length (`base + len`) of the instruction to be loaded exceeds the 4KB limit (`romSize`).

`romSize` is a parameter of the Load function (where this code is executed) and is declared and passed by the `main` function:

```
// Size of ROM defined in main function (line 193 in i8051_mkr.c file)
unsigned char rom[4096];
...
// Call to Load function (size is passed with sizeof function, line 206)
if( Load(buf, rom, sizeof(rom), &prgSize) ) break;
...
// Definition of Load function, where the romSize variable is defined (line 48)
int Load(char *buf, unsigned char *rom, unsigned romSize, unsigned *prgSize)
```

So if a program is too large to fit in ROM, it cannot be loaded and the .vhd file cannot be generated. To test this behavior we created a 'large' file with lots of variables and calculations, we compiled it for the *mmcs51* family and tried to generate the ROM vhd file from the HEX file:

```
es_student@sohrabi-VirtualBox:~/workspaces/i8051/divmulPROVA/TEST3$ sdcc -mmcs51 --iram-size 128 eq.c
es_student@sohrabi-VirtualBox:~/workspaces/i8051/divmulPROVA/TEST3$ packihx eq.ihx > eq.hex
packihx: read 153 lines, wrote 267: OK.
es_student@sohrabi-VirtualBox:~/workspaces/i8051/divmulPROVA/TEST3$ ../../hex2rom/Release/Hex2Rom2.o eq.hex
program too large
```

This check can also be done (preemptively) with the *SDCC* compiler using the `--code-size 4096` flag:

```
es_student@sohrabi-VirtualBox:~/workspaces/i8051/divmulPROVA/xrm_test$ sdcc -mmcs51 --iram-size 128 --code-size 4096 eq.c
?ASlink-Error-Insufficient ROM/EPROM/FLASH memory.
```

While the original 8051 microcontroller has external memory for user programs, this model only provides a file for external RAM, so we can extend the size of data in our programs but not program size. In order to solve this problem, a first naive solution is to increase the capped limit of the ROM in the code. Code will fit in the simulated memory, but it will not fit in a real world 8051 processor. Otherwise, a vhd file to represent external program memory has to be created to obtain an optimal solution.

Exceeding RAM size

Similar problem as the one we just described can be applied to RAM size. The *SDCC* compiler has the option to check this limit. with the `--iram-size 128` flag that checks the program code for variables memory allocation

```
es_student@sohrabi-VirtualBox:~/workspaces/i8051/divmulPROVA/rom_overflow$ sdcc -mmcs51 --iram-size 128 --code-size 4096 --xram-size 2048
  eq.c
eq.c:127: warning 85: in function main unreferenced local variable : 'test12'
?ASlink-Error-Could not get 2660 consecutive bytes in internal RAM for area DSEG.
```

But we can overcome this limit, using the external RAM. We can define variables that we want to write in external RAM prepending the `__xdata` keyword to the variable definition:

```
unsigned char __xdata buffer[2048];
```

As you can see from the previous error message, a 2048 bytes variable cannot fit in a 128 byte RAM memory, but can in the external RAM. So now the program compiles (source code of the program):

```
es_student@sohrabi-VirtualBox:~/workspaces/i8051/divmulPROVA/xrm_test$ sdcc -mmcs51 --iram-size 128 --code-size 4096 --xram-size 2048 eq.c
es_student@sohrabi-VirtualBox:~/workspaces/i8051/divmulPROVA/xrm_test$
```

You can see the *SDCC* has a flag to check external size memory too, `--xram-size 2048`. If we try to exceed this limit, we get a similar error as the one we saw before:

```
es_student@sohrabi-VirtualBox:~/workspaces/i8051/divmulPROVA/xrm_test$ sdcc -mmcs51 --iram-size 128 --code-size 4096 --xram-size 2048 eq.c

?ASlink-Error-Insufficient EXTERNAL RAM memory.
```

Other Projects

There are not many alternative projects that try to simulate the 8051 microcontroller hardware. While the Dalton Project is the main result we get, when searching for an hardware simulation, it has some limitations. Notably, interrupt handling is not implemented, also timing/cycling is not present and there is no support for peripheral devices.

An alternative project that tries to improve the implementation is the OpenCore project. It is written in Verilog, not VHDL, but still can be used in Vivado to simulate user programs. It provides the functionalities not supported by Dalton Project while being notably more efficient.