Computer Organization

Iván de Jesús Deras Tábora

March 22, 2019

Period 1 2019

MIPS32SOC Part 5

Objectives

In this part of the project you'll be to simulate the SOC using verilator, also you'll synthesize the circuit using a MimasV2 FPGA board.

What you will need

In order to developed this part of the project you'll need the following:

1. Part 4 of the project completed
2. The clock generator module
3. The milliseconds counter

The clock generator and the milliseconds counter will be provided to you.

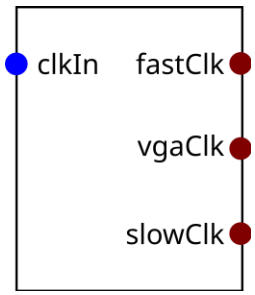
Your task

To achieve your goal you'll have to do the following

1. Use the new Instruction Memory module.
2. Use the Clock Generator module to generate three clock frequencies.
3. Use the new Milliseconds Counter module to generate the milliseconds counter.
4. Map the inputs and outputs to the FPGA pins. This is needed only for synthesis.

The Clock Generator module

The clock manager circuit is responsible of generating the cpu clock (50MHz) and the VGA clock (25MHz). The schematic is the following:



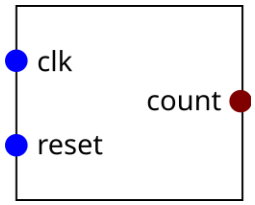
Signal	Type	Description	Bits
`clkIn`	Input	Input clock. This input should be mapped to the FPGA 100MHz clock signal.	1
`slowClk`	Output	Slow CPU clock signal (half the frequency of the fast clock frequency). This signal will be connected to the clock input of the Instruction Memory, Register File and the Program Counter register.	1
`vgaClk`	Output	VGA clock signal (25Mhz)	1
`fastClk`	Outpu	Fast clock signal (50Mhz). This will be conected to the Data Memory clock signal.	1

The VGA Text controller and the Data Memory module

The read port of these two modules should be synchronous now. That is the read operation should depend on the clock signal.

The Milliseconds Counter module

The circuit is responsible of generating milliseconds counter. The schematic is the following:



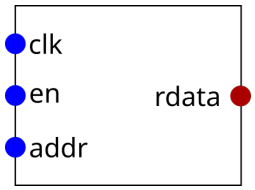
Signal	Type	Description	Bits
`clk`	Input	Clock signal	1
`reset`	input	Reset signal	1
`count`	Output	Counter output (the number of milliseconds)	32

The Keypad

Add an 8 bit input signal called `keypad` to the main module. This signal will contain the pushbutton states of the keypad. Be sure to put the keypad state in the upper 8 bits of the word (bits **31:24**). Remember our processor is BIG Endian.

The Instruction Memory module

The new Instruction Memory module is synchronous. That is for reading data it needs a clock pulse. The schematic of the circuit is the following:



Signal	Type	Description	Bits
`clk`	Input	Clock signal	1
`en`	Input	Enable signal	1
`addr`	Input	Address to read	11
`rdata`	Output	Output data	32

Mapping inputs and outputs to FPGA pins

To map input and outputs of the circuit to FPGA pins you'll be provided with a Digital circuit that contains all the MimasV2 input and output pins. You just have to copy the needed element to your circuit.

In order to be able to compile the generated verilog code with **verilator** for simulation and **Xilinx ISE** for synthesis, you'll have to follow this rules:

1. The input and output signals must follow this conventions:

Name	Type	**Pin mapping**	**Description**
`clk`	Input	FPGA 100MHz clock	The main clock signal
`reset`	Input	DIP Switch 8	The circuit reset
`keypad`	Input	6 pushbuttons + DIP Switch 1 + DIP Switch 2	The keypad
`red`	Output	VGA Red pins	The VGA red output
`green`	Output	VGA Green pins	The VGA green output
`blue`	Output	VGA Blue pins	The VGA blue output
`hsync`	Output	VGA HSync pin	The VGA hsync
`vsync`	Output	VGA VSync pin	The VGA vsync
`invalidPC`	Output	LED 0	Invalid Program Counter
`invalidOpc`	Output	LED 1	Invalid Opcode
`invalidAddr`	Output	LED 2	Invalid Address

2. The name of the output file must be `Mips32soc.v`

Implementation and Testing

In order to implement this part of the project follow these steps:

1. Add the Milliseconds counter module and the Keypad input and complete the I/O module. Remember the address `0xffff0004` should return the keypad state and the address `0xffff0008` should return the milliseconds counter. In the case of the keypad the state should be in the byte 0 of the word. Remember our processor is BIG endian.

2. Run the tests from part 4, all of them must succeed. In order to run the tests add the following code in the file `test_MIPS32SOC.cpp`

Function prototypes

```
// The milliseconds counter use this function
static int msCount;

int millis() {
    return msCount;
}
```

If some of the test failed, you have introduced a regression, so you have to FIX it before moving on.

3. Test the I/O module following these steps:

   (a) Add this code at the end of file `tests-code.cpp`

   I/O test code

```
// File 'test_io.S'
const uint32_t test_io_code[] = {
    /*0x00400000: */ 0x8c880000, //lw  t0, 0(a0)
    /*0x00400004: */ 0x8c890004, //lw  t1, 4(a0)
    /*0x00400008: */ 0x8c8a0008, //lw  t2, 8(a0)
    /*0x0040000c: */ 0x08100003, //j  0x40000c
};
const int TEST_IO_CODE_SIZE = 4;
```

   (b) Add this code at the end of the file `test_MIPS32SOC.cpp`

   I/O test case

```
TEST_CASE("'lw' I/O memory test") {
    VMIPS32SOC m;
    DECLARE_MIPS32_REGS(m);
    randomizeRegisterFile(m);
    a0 = 0xffff0000;
    setProgramCode(m, test_io_code, TEST_IO_CODE_SIZE);
    m.keypad = 0x53;
    msCount = 0xdeadbeef;
    reset(m);
    REQUIRE(pc == CODE_BASEADDR);
    CHECK_ERROR_SIGNALS(m);
    REQUIRE(m.MIPS32SOC->invalidAddr == 0);
    clockPulse(m);
    CHECK(t0 == 0);
    CHECK_ERROR_SIGNALS(m);
    REQUIRE(m.MIPS32SOC->invalidAddr == 0);
    clockPulse(m);
    CHECK(CHex(t1) == CHex(0x53000000));
    CHECK_ERROR_SIGNALS(m);
    REQUIRE(m.MIPS32SOC->invalidAddr == 0);
    clockPulse(m);
    CHECK(CHex(t2) == CHex(0xdeadbeef));
}
```

   (c) Compile and run the test cases. The new test case should succeed if you implemented the I/O module correctly.

4. Add the Clock Generator module. Remember the `clkIn` signal will be connected to main clock input. The `slowClk` signal will be connected to the clock signal of the **Register File**, **Instruction Memory** and **Program Counter**. The `vgaClk` is the VGA protocol clock signal (the signal is called `vclk`) and the `fastClk` is connected to the **Data Memory** and **VGA Memory** clock (in the case of the VGA memory the signal is called `clk`).

5. Add the new **Instruction Memory** module. Also be sure to remove the **AsyncROM** module.

6. Change the Data Memory read port from asynchronous to synchronous. The code should look like the following, basically the `assign` to the signal `rdata` is now inside the `always` block.

Data Memory

```
reg [31:0] memory[0:2047] /*verilator public*/;

always @(posedge clk)
begin
  if (en) begin
    if (memWrite[3]) memory[addr][7:0] <= wdata[7:0];
    if (memWrite[2]) memory[addr][15:8] <= wdata[15:8];
    if (memWrite[1]) memory[addr][23:16] <= wdata[23:16];
    if (memWrite[0]) memory[addr][31:24] <= wdata[31:24];

    rdata <= memory[addr];
  end
end
```

7. Change the VGA Memory read port from asynchronous to synchronous (The VGA memory module is called `DualPortVGARam`). The changes are similar to those you made to the **Data Memory**. The `assign` to the signal `rda` is now inside the `always @ (posedge clka)` block. There's a second always block but don't touch that.

8. Forward the next PC value to the PC Decoder. Remember when the system is on reset state the next PC value is `0x400000`.

9. In the `CMakeLists.txt`, change the name `AsyncROM.v` to `InstMem.v`

10. In the file `test_MIPS32SOC.cpp`, change the `#include "VMIPS32SOC_AsyncROM.h"` to `#include "VMIPS32SOC_InstMem.h"`

11. The tests should compile now. If you run the tests almost all of them will fail, that's OK. At least one should pass. The problem is that now our proccesor needs two clock cycles to execute every instruction.

12. In the file `test_MIPS32SOC.cpp` change the name of the function `clockPulse` to `oneClockPulse`, then create a new function called `clockPulse` which calls two times the `oneClockPulse` function. Also change the reset function as shown in the following code snippet.

reset function

```
void reset(VMIPS32SOC& m) {
    m.reset = 1;
    m.clk = 0;
    clockPulse(m);
    m.reset = 0;
}
```

Be sure to move the `reset` function after the `clockPulse` function. If you don't do this the compiler will generate an error.

13. Compile and test again. Now `51` out of `53` tests should pass, if not check the above steps. The only tests that should failed at this point are `'invalid global address' test` and `'invalid stack address' test`

14. FIX the test cases. Look at the output from the tests. In the error report there is a line number. Go to that line number, it should have something like `REQUIRE(m.MIPS32SOC->invalidAddr == 1);` before that line insert a call to `oneClockPulse(m)`. Do this on both lines, then compile again and run the tests. Now all of them should succeed. If not check the above steps.

Support files

The support files contains the following:

1. The `sim/` folder. This folder contains all the C++ source files needed to build an executable file for the simulator. In the next section we'll build the simulator.
2. The `tools/` folder. This folder contains the `elf2mif` tool, you have to compile it then copy it to the `/usr/bin` system folder. There is a compiled version of `elf2mif`, you can use that.
3. The `xilinx/` folder. This folder contains the files needed to synthesize the circuit for the MimasV2 FPGA board.
4. The `mif_samples/` folder. Contains some sample MIF files. You can use them to test the simulator and the synthesized circuit.
5. The `lib/` folder. Contains the new library for the game.

Compile the simulator and run the game

To compile the simulator follow these steps:

1. From the provided support files copy the `sim/` into a `cpp/` subfolder. You have to create the `cpp/` folder in the top directory.
2. Declare as output the signals `invalidOpcode`, `invalidPC` and `invalidAddr`.
3. Rename the input signals `keypad` and `reset`, to `keypadIn` and `resetIn` in the top module and the UCF file. **NOTE:** Take into account that after this changes the simulator will failed compile. In order to compile it again you have to fix the signal names in the file `Mips32SocSim.cpp`
4. Declare local wires for `keypad` (the keypad is an 8-bit signal) and `reset` signals.
5. Add the following code to the top module:

   Reset and Keypad

   ```
   `ifdef SYNTHESIS
       assign reset = ~resetIn;
       assign keypad = ~keypadIn;
   `else
       assign reset = resetIn;
       assign keypad = keypadIn;
   `endif
   ```

6. Install the SDL2 development files for you distribution (In Ubuntu and Mint the package is called `libsdl2-dev`)
7. Add the following lines (before the `reg` declaration) to the DualPortVGARam module.

   Lines

   ```
   `ifdef verilator
       /* verilator lint_off UNUSED */
       import "DPI-C" function int updateVGADisplay();
       integer dummy;
   `endif
   ```

Also add these lines inside the `if (enablea)` statement:

Lines

```
`ifdef verilator
    if (wea != 4'b0000)
        dummy <= $c( updateDisplay() );
`endif
```

8. The new Instruction Memory uses 2048 words. You have to change the PCDecoder to allow the use of the 2048 words.
9. From the support files copy the `lib/libtinymips.a` into your game `lib/` folder.
10. In the `Makefile` change the flag `-fdelayed-branch` to `-fno-delayed-branch` in following line:

Makefile

```
CFLAGS=-I../include -G 0 -Os -Wall -ffreestanding -fno-stack-protector \
       -nostdinc -nostdlib -fdelayed-branch -fno-builtin
```

11. Then type the following commands inside the game folder (here we assume that the game executable file is called `game.elf`):

Compiling the Game

```
$ make clean
$ make
$ elf2mif game.elf code.mif data.mif 2048 1024
```

12. In the `CMakeLists.txt` file from the `cpp/sim` folder change the following line:

Compiling the Game

```
set(VERILOG_SRCDIR "${CMAKE_SOURCE_DIR}/../../part5/verilog")
```

This variable should point to the folder where the verilog sources are.

13. Use the following commands to compile the MIPS32SOC simulator (your working directory should be the top folder):

Compiling the Simulator

```
$ mkdir build-sim
$ cd build-sim
$ cmake ../cpp/sim
$ make
```

14. Run the MIPS32SOC simulator with the following command (asumming you are inside the `build-sim` folder):

Generating Xilinx ISE Project

```
$ ./Mips32SocSim --font font_rom.mif --mif code.mif:data.mif
```

Be sure to copy the files `font_rom.mif`, `code.mif` and `data.mif` inside the `build-sim` folder.

Synthesize the circuit

1. Be sure to declare any signal used in an `always` block as `reg`. If you failed to do this Xilinx ISE will generate an error during circuit synthesis.
2. Copy the `code.mif` and `data.mif` files generated from your game to the `verilog/cpu/` folder.
3. From the provided support files copy the `xilinx/` into the top folder.
4. Use the following commands to generate the Xilinx project file:

   Generating Xilinx ISE Project

   ```
   $ cd xilinx/project
   $ sh ../scripts/gen_xilinx_prj.sh part5/verilog > MIPS32SOC.xise
   ```

   Replace the path `part5/verilog` with the folder path that contains the verilog sources.
5. Open the project file `MIPS32SOC.xise` using Xilinx ISE and generate the programming file.
6. Load the `MIPS32SOC.bin` into the MimasV2 board using the Mimas Configurator Tool.
7. Test and you are done !