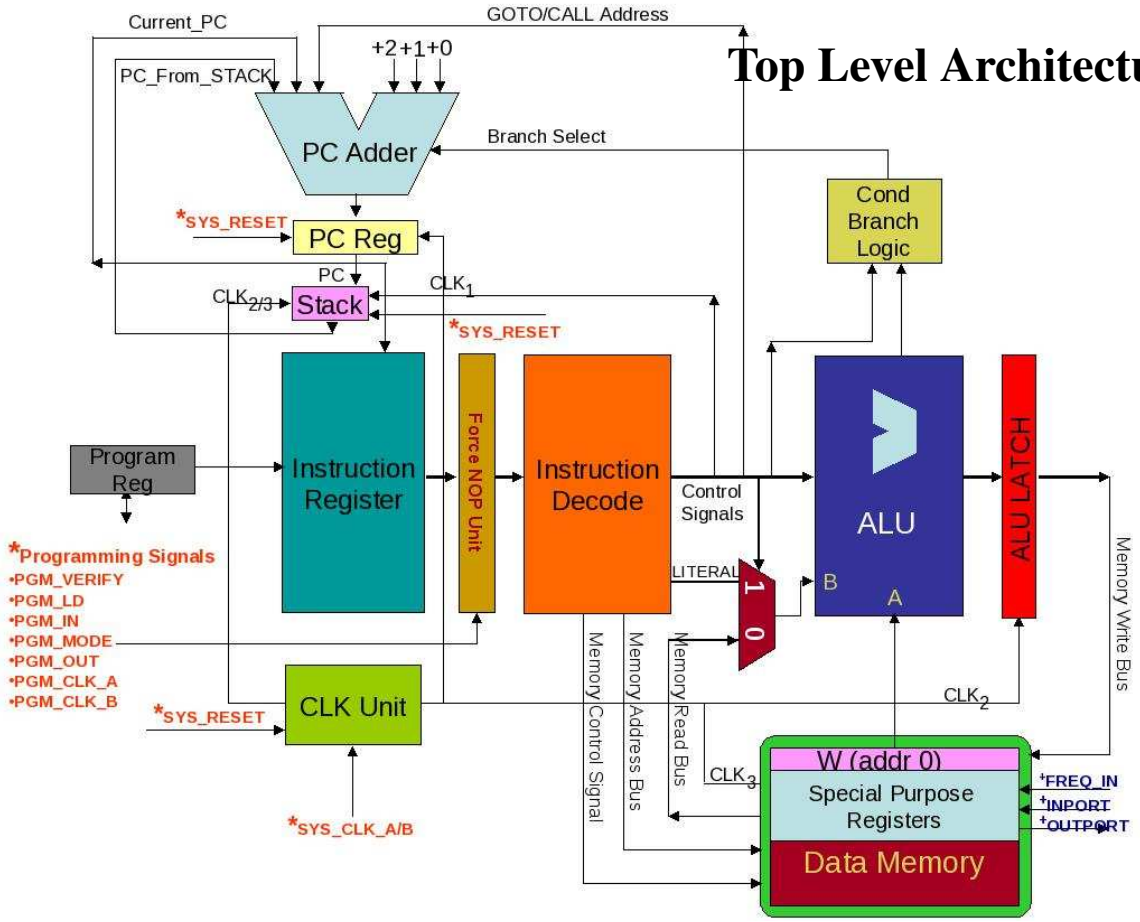
LAB Assignment #8 for ECE 443

Assigned: Wed., Oct 28, 2009
Due: Mon., Nov. 11, 2009

Description: Build Instruction Register/Memory and the PC logic and stack of the microcontroller

The microcontroller we are designing was originally designed by Dhruva Acharyya. The top-level architecture he derived for the microcontroller is shown below.



Red: external signals to control programming the microcontroller
Blue: external signals for runtime communication/operation

You have built the Instruction Decoder and ALU. This lab will focus on building the instruction memory (labeled Instruction Register above).

For the instruction memory, you will use BRAM (an embedded memory on the FPGA). The instruction memory is 256 words, where each word is 12-bits wide. So the address bus is 8-bit wide. The output of the BRAM replaces your IR register -- in other words, you will use the output bus from the core generator as the instruction register. You will use the Core Generator to generate a BRAM with these size parameters. You will also indicate that the outputs are to be 'registered', which will create the IR register for you automatically. To create a BRAM memory, follow these steps:

1) Under ‘programs/ISE xx/accessories’, click on ‘CORE Generator’
2) Select ‘new project’ under the file menu
3) Enter a name and directory
4) In the CGP form, select Virtex2P, xc2vp30, ff896, -7. Click Ok.
5) Expand ‘Memories & Storage’ in the list on the left in CORE Generator, then ‘RAMs & ROMs’
6) Select ‘Block Memory’ and Click Customize on the right.
7) Block Memory Generator dialog then appears, choose ‘Single Port RAM’ (the default) and name the component InstrMem - leave ‘Minimum Area’ selected for the Algorithm, click ‘Next’.
8) Set ‘Memory Size’ params ‘Write Width’ to 12, ‘Write Depth’ to 256, leave ‘Operating Mode’ set to ‘Write First’, ‘Enable’ set to ‘Always Enabled’ and ‘Output Reset Value’ at 0, click next.
9) The next screen allows you to add a register to the output of the BRAM, which will serve as the IR in your design. Select ‘Register Port A Output of Memory Core’ under ‘Optional Output Registers’-‘Port A’, leave other options at default and click next.
10) Leave options as is on the last screen and click ‘Finish’

The CORE Generator generates a set of files:

InstrMem.ngc: Binary Xilinx implementation netlist file containing the information required to implement the module in a Xilinx (R) FPGA.

InstrMem.vhd: VHDL wrapper file provided to support functional simulation. This file contains simulation model customization data that is passed to a parameterized simulation model for the core.

InstrMem.vho: VHO template file containing code that can be used as a model for instantiating a CORE Generator module in a VHDL design.

InstrMem.xco: CORE Generator input file containing the parameters used to regenerate a core.

Plus some other files...

The InstrMem.vho contains an example instantiation to be inserted into your VHDL code as described below.

```
component InstrMem
        port (
        clka: IN std_logic;
        dina: IN std_logic_VECTOR(11 downto 0);
        addra: IN std_logic_VECTOR(7 downto 0);
        wea: IN std_logic_VECTOR(0 downto 0);
        douta: OUT std_logic_VECTOR(11 downto 0));
end component;

your_instance_name : InstrMem
   port map (clka=>clka, dina=>dina, addra=>addra, wea=>wea, douta=>douta);
```

InstrMemUnit Module (module 1):

You then need to create a new ISE project that will serve as the instruction memory (InstrMemUnit) with the following parameters:

The **inputs** are as follows:

clk: 1-bit

reset: 1-bit

PC: 8-bits

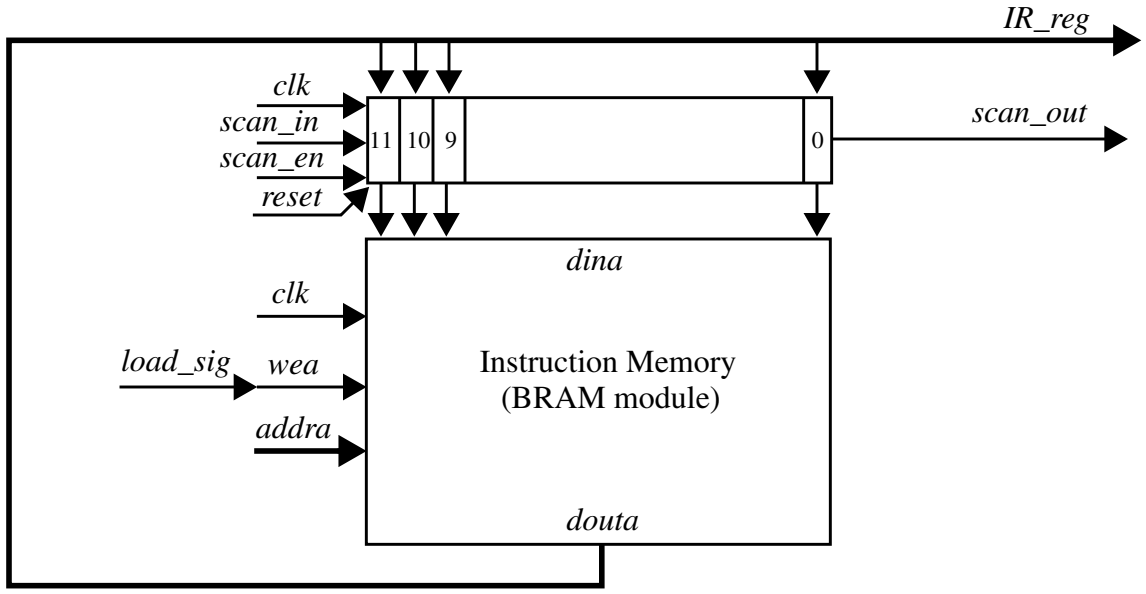
scan_in: 1-bit
scan_en: 1-bit
load_sig: 1-bit

The **outputs** are as follows:

IR_reg: 12-bit output
scan_out: 1-bit

Within this module, you should add an instance of the BRAM you created above using the template provided. To accomplish this, add the **InstrMem.xco** file as a source after you create the ISE project. You also need to connect the proper signals to the actual parameters as described below. *douta* of the memory module will serve as the IR register.

Within the *InstrMemUnit* module, you also need to implement a **scan register** and connect it as shown below:



A **scan register** is a special register that has two input paths, the standard (parallel) path that drives the D inputs as shown above with the IR bus and a serial input path given by *scan_in*. The scan register operates as a regular register when *scan_en* is low. When *scan_en* is asserted, it shifts the value present on *scan_in* into the MSB and all other bits are shifted to the right, i.e., it effectively becomes a shift register. The *scan_out* port will not be used in this laboratory but needs to be present in the module.

ScanEmulation Module (module 2):

You need to implement a state machine in this module that will scan an instruction into the scan register using the *scan_in* path as described below. The basic idea is that we are going to use the scan register to load up the instruction memory. As instructions are entered into hyperterminal, they are converted into a binary value (using a UART driver that I will provide) and passed to the *ScanEmulation* module. The state machine in *ScanEmulation* module will serially scan the instruction into the scan register located in the *InstrMemUnit* module.

The instruction will be saved in instruction memory at the address given by *scan_PC_reg* -- a register created and maintained within the *ScanEmulation* module. Therefore, during the instruction

load operation, your state machine will update *scan_PC_reg* so that it generates a sequence of addresses in order from 0 to *n*, where *n < 256*. To write the instruction memory, you need to place the address and data on the appropriate inputs of the BRAM memory and then enable the write operation by asserting *load_sig* for one cycle. NOTE: Although you will NOT be reading instructions out of instruction memory in this assignment, you will need to do this eventually. In order to read, you need to place the address on the BRAM address inputs and wait for **TWO** clock cycles before the data will be available.

We call the second module *ScanEmulation* because for a normal implementation of the microcontroller, the signals *scan_in*, *scan_out*, *scan_enable* and *load_sig* would be connected to external pins (they would not be driven by internal logic as described above). Since wiring the board in this fashion is not practical, in your implementation, you will instead write a FSM that **emulates** the external (off-chip control) of these signals. By emulation, I mean your state machine will toggle these signals in the same manner they would be toggled by external instrumentation such as a pattern generator.

As I indicated, I will provide a driver called UART_InstrRead that is designed to read a sequence of instructions, converts them into binary and asserts a signal when it is ready to be processed by your *ScanEmulation* module. The asserted signal is called *go_scan_emulation* -- it will be pulsed for one clock cycle when the binary version of the instruction entered by the user is available. This event should trigger your state machine to carry out the following sequence of operations. First, the *scan_enable* signal is asserted and held high during the scan operation given as follows. For each bit of the instruction, starting with the low order bit, you need to drive the *scan_in* signal with that bit for one clock cycle. After the last bit is scanned in, you need to assert the *load_sig* signal for one clock cycle to save the instruction into instruction memory. After you save it, you should increment *scan_PC_reg* by 1 and go back to the initial state and wait for the next instruction. Note that you need to declare and implement *scan_PC_reg* as a register in your module, similar to way you handled the IR register from lab #7.

The **inputs** to *ScanEmulation* are as follows:

clk: 1-bit
reset: 1-bit
Instr_In_Reg: 12-bit output (connected to *ScanIRIn_reg* in UART_InstrRead driver)
go_scan_emulation: 1-bit

The **outputs** are as follows:

scan_in: 1-bit
scan_en: 1-bit
load_sig: 1-bit
scan_PC_reg: 8-bits

Although the ALU and decoder and not the focus of this lab, you must leave them in the VHDL code. You should add two modules to the driver code that I provide, InstrMemUnit and ScanEmulation. See the comments in the driver code. IMPORTANT: Be sure to connect the PC port in InstrMemUnit to *PC* in my driver and NOT to the output of the ScanEmulation module's *scan_PC_reg*. For example, in the **port map** statement, you should have ...PC=>PC... in the InstrMemUnit instantiation.

For the demonstration, you will use hyperterminal and enter a series of instructions (one per line). I have developed the driver so that the contents of your instruction memory module will be dumped to hyperterminal once the "enter pushbutton" is pressed on the FPGA.

Laboratory Report Requirements:

NOTE: You will loose 10 pts/day every day the lab report is late. If you miss the demo, then the earliest you can do it is the next class period. For each class period that you are late, you will loose 20 pts.

1) Turn in a commented copy of your VHDL code along with a schematic.
2) Write a test bench and run simulation(s) that shows the inputs and output behavior of the circuit (be sure to include a set of 'sample' waveforms in your report).
3) Be prepared to give a demonstration in class on Wed using my UART driver code as described above.

Grading:
Your lab grade will consist of two parts. The first part is associated with the in-class demo, and is worth 50% of the total grade (50 pts). Successful demonstration of the lab's stated requirements is worth 50 pts. Partial implementations will be given only partial credit. The second portion of the lab grade is derived from your lab report. Correct implementation counts for 15 pts (of the 50 pts). Well documented simulation results are also worth 15 pts. The remaining 20 pts will be given according to how well the VHDL code is written and documented (comments). Bonus points will be given to any implementation feature that goes above and beyond the requirements.