COL216 Computer Architecture

Lab Assignments 7 : CPU with Multi-cycle Design Style

Introduction

The objective of this assignment is to change the CPU design to follow a multi-cycle approach. In this approach, each instruction takes a certain number of clock cycles to complete, doing one primitive action in one cycle. The number of cycles taken by different instructions may be different. In such a design, an FSM is used as a controller. The state of this controller determines what action(s) is(are) required to be done in the current cycle. The datapath for such a design has some additional registers to store some of the intermediate results produced by the primitive actions. A multi-cycle design approach also offers a possibility of sharing resources across clock cycles.

In the previous assignment, an FSM was introduced to facilitate single step operation. This allows a program to be run either continuously or one clock cycle at a time. For the multi-cycle design of this asignment, adding one more mode of operation will be desirable – single instruction execution. In this mode the circuit will step through the required number of clock cycles to complete one instruction.

The current design will get enhanced in may different ways in the subsequent assignments. Therefore, it will be useful to make the design more modular at this stage, so that enhancements can be viewed at the level of modules. This would be achieved by structurally partitioning the design into smaller modules.

Thus, there are 4 tasks involved in this assignment.

1. Partition the design into modules.
2. Enhance the execution controller FSM.
3. Introduce an FSM for stepping through the primitive actions.
4. Modify the datapath.

Detailed Description

<u>Modular design</u>

So far, most of the students would have put the entire processor design as a single entity with its architecture in a single file. Now it is time to split the design into multiple modules with their entities+architectures organized in separate files. The main processor will have the remaining functionality plus instantiation statements for these modules. Some possible candidates for defining as separate modules are as follows.

- Register File:

  This will have two read ports and one write port, along with a dedicated port for PC. Each read port has an address input and a data output. The write port has an address input, a data input and a write enable. The PC port has a data input, a data output and a write enable, but no address input (since the address is fixed).

- ALU and Flags:

  This will have two operand inputs, a result output, a control input specifying the operation to be performed (presently, add or subtract only), flag outputs (presently, Z flag only) and a flag write enable input. Note that the ALU part is combinational but the flag part has storage.

- Instruction Decoder:

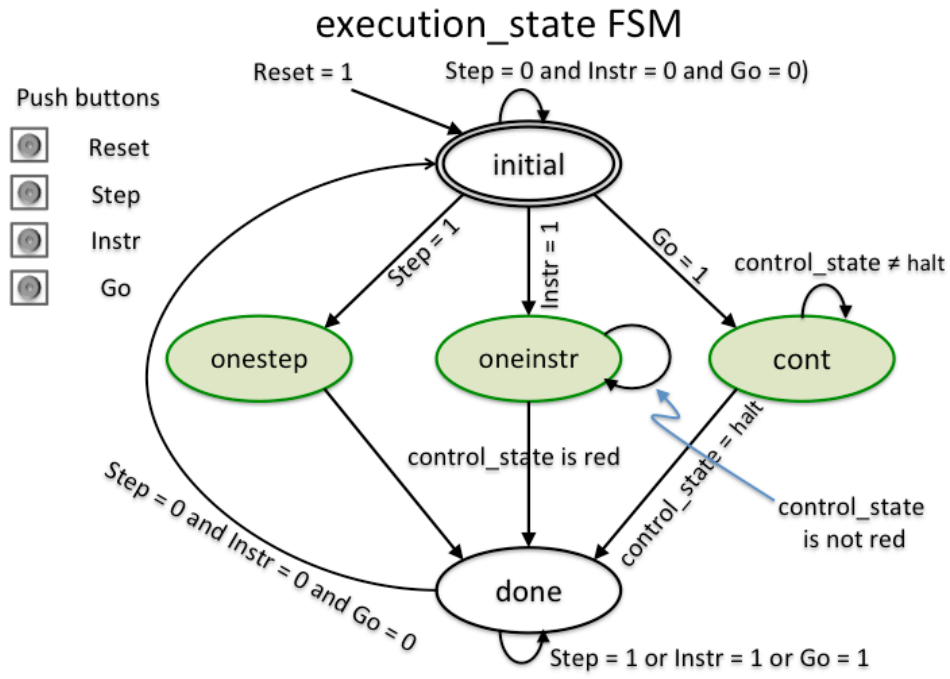
  The outputs of this module would indicate what is the current instruction and to what class it belongs. The inputs would be the relevant instruction fields required to produce these outputs. Include "halt" as an additional instruction class.

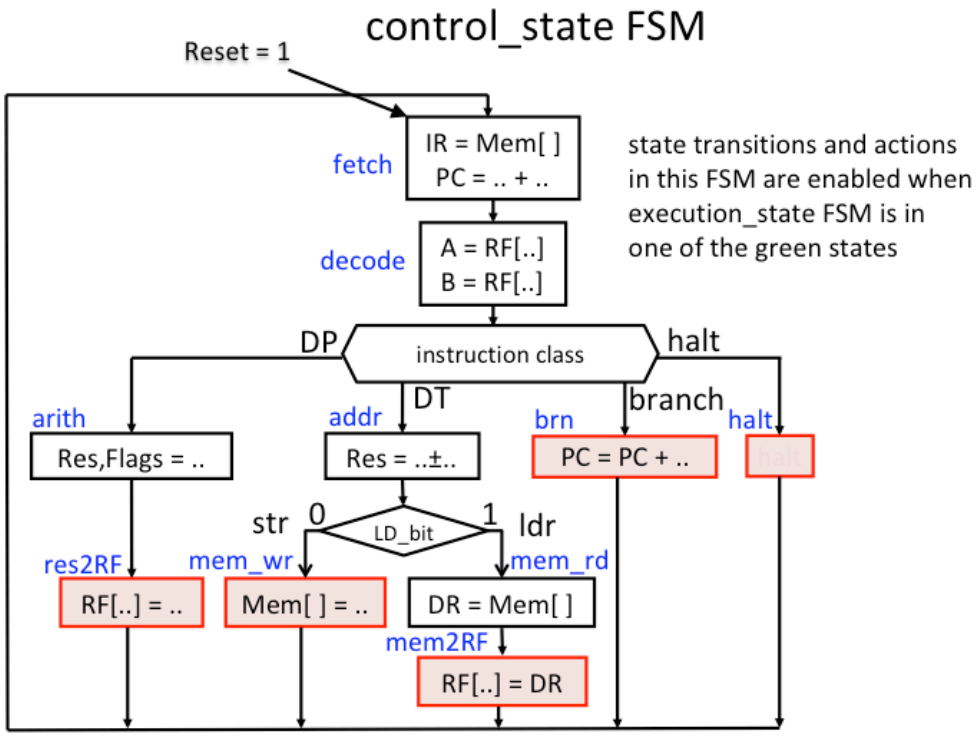
- Other adders/subtractor:

  Load/store instructions require offset addition or subtraction. PC requires incrementing and branch offset addition. These operations could be done by the ALU itself, but if that is not the case, then it is desirable to define separate components for these. Note that branch offset addition is of the form PC + 4 + offset. This is different from plain addition and would require the ALU to support a third operation. However, since all the three quantities in this expression are multiples of 4, we can actually compute PC(31 downto 2) + 1 + offset(31 downto 2) and concatenate two '0' bit with the result. This simply amounts to a plain addition with initial carry and does not add any complexity to the ALU.

FSMs for execution control and main control

State transition diagrams for the two controllers, labelled as "execution_state FSM" and "control_state FSM" are shown here.





The execution_state FSM is an extension of the one designed for Lab Assignment 6. An additional state "oneinstr" and a push button labelled "Instr" are introduced here to allow the processor to step through 3, 4 or 5 cycles as required for complete execution of the current instruction. The FSM continues in oninstr state until the current instruction reaches the last cycle of its execution. This condition is checked by looking at the state of the other FSM (control_state FSM). The states corresponding to the last cycle of an instruction are shown in red colour. The control_state FSM has one additional state "halt", as compared to the controller discussed in Lecture 9 (slides 29-31). Further, state transitions and actions in this FSM are enabled when execution_state FSM is in a state shown in green colour.

These two FSMs should be created as separate components.

Datapath modifications

Introduce additional registers in the datapath as shown in slide 20 of Lecture 9. These registers are referred to as temporary registers in the following discussion. These may be described behaviourally (signal declaration plus assignment in clocked process(es)) rather than describing as separate components (entities, architectures and instantiations). After separating the modules to make the design modular, as discussed earlier, the remaining datapath functionality includes the following.

(a) Concurrent assignment describing the combinational part feeding into the other modules and into the temporary register inputs.

(b) Clocked process(es) for making assignments to the temporary registers in appropriate control states.

It should be noticed that the temporary registers hold the portion of the processor state that is visible at microarchitectural level but not at ISA level, (meaning simply that these registers are not visible to an assembly language programmer). In the present context, an assembly language programmer sees only the contents of memory, register file (including program counter) and Z flag, which are now separate modules as per the modular design approach discussed earlier.

Resource Sharing

In this design, the following resource sharing options exist, as discussed in Lecture 9.

Replacing the split memory structure (Instruction Memory + Data Memory) by a single unified memory.

Subsuming the functionality for load/store offset addition/subtraction, PC incrementing and branch offset addition within the ALU.

These options may or may not be exercised in this assignment. However, having a unified memory would be essential for Assignment 14.

Overall Scope

The assignment involves designing, simulation, synthesis, testing on the FPGA board and reporting resource usage and performance results. The work is likely to spill beyond one week. The next four assignments are relatively simpler and will provide ample time to catch up.