CAD8 Controller FALL 2009

Assignment

Design the controller of your microprocessor.

Description

The first step in designing the controller is to identify and define the control signals, which should already be documented in CAD7. The controller for the baseline architecture is simplified by the fact that all instructions are one word long. In the simplest implementation of the microcontroller, the instruction decode can be done without finite state machines (FSMs). If FSMs are used for jump and branch, or other instructions (e.g. interrupt processing) which require two cycles in the execute stage, the FSMs can be very simple.

Implementing the control in a pipelined machine which has a separate decode stage (unlike ours) means having the necessary logic to set the control bits to their respective values in each stage for each instruction. This is accomplished by decoding the opcode from the IR and the condition bits (in case of a branch or jump) to create the control bits which are stored in the Control Register. These control bits dictate the function of the *EXECUTE* stage in the next cycle. Simple control is also required for the fetch stage; the details of which depend upon your timing scheme for memory access and write back.

In the 427 Baseline Architecture, decode and execute are in the same stage (to avoid data dependencies). The decode is so simple and fast that it does not add much delay to the execute stage. You can implement the control (decode) logic without a control register, but be certain that any lines controlling writing to data memory or register files are hazard-free.

Your data memory interface (and sometimes your program memory interface) will have associated control signals in addition to data and address busses. These control signals should also be driven by your control unit. You may also need a few flip-flops to save certain control bits for one more clock cycle; for example, if you write back results into the register file on the positive edge of the clock after the completion of the execute stage, you may need to latch the Write_Enable signal on the falling edge of the clock so that you do not use the control signal value of the next instruction. Consider making some or all of the flip-flops in your controller scannable. If you don't have an external program memory interface, scanning the IR will be important. Scanning the PSR is also a good idea. Base your decision to scan other flip-flops, if present, by considering their controllability and observability. Synopsys's *Design Compiler* has the ability to automatically insert scan chains but hard instantiation of scan flip-flops (or inference of scan with << and >> operators) will prove simpler for your project. Finally, keep in mind that D flip-flops and scan flip-flops are expensive in terms of area. You should be able to clearly identify the need for any flip-flop in your controller; then make sure that you don't accidentally infer more flip-flops by checking your synthesis result.

Procedure

Modeling

Describe your controller's functionality using the Verilog hardware description language. Simulate this Verilog description of your controller along with your datapath and memories prior to synthesis to ensure correct operation. Guidelines for writing good, synthesizable Verilog will be discussed in lecture. Pay attention to these guidelines since it's easy to generate poor gate-level implementations if you're unaware of some of the pitfalls. Partitioning your controller into smaller modules can be helpful in monitoring the quality of the synthesized result, particularly if you're new to synthesis.

Synthesizing and Compiling

Having tested the Verilog model, you can then synthesize it in Synopsys's *Design Compiler*. Synopsys will take the Verilog description along with timing/area constraints (provided by you) and attempt to generate a gate-level netlist that meets these constraints. You should set the timing constraints based on your knowledge of the timing of your datapath and any estimates you may have for associated peripheral

circuits (including memories). If you do choose to partition your design, you should set constraints only at the highest level. Once you've synthesized a gate-level netlist, you have the option of checking correct functionality prior to APR by simulating in *NCVerilog*. Since APR is relatively simple for controllers of this size, you can skip this step and verify functionality with better delay information after completing APR.

APR with Cadence's *SOC-Encounter* should be relatively simple. With your chip-level floorplan in mind, choose a good aspect ratio for your controller and distribute pins along the periphery in such a way to ease routing at the chip-level when you route the controller, datapath, and other large blocks. Once you've completed APR (and all functional verification), export your layout to *ICFB* and run DRC and LVS. The schematic you use for *NCVerilog* should contain the controller, datapath, and memories. You should not, however, create a layout yet to match this schematic. This will be done as part of your final project, top-level integration. You can use the on-chip memory models in the class directory for off-chip memory as well if you are unable to find Verilog descriptions of the off-chip memory you've selected. In this case, you'll need to modify size and delay parameters for the memory model to better mimic the external memory you've selected. Export SDF from *SOC-Encounter* and verify that the synthesized controller works with your datapath. You're only required to show proper operation of ***ibm13/assembler/testfile.asm*** for cad8 but you may want to write additional patterns to check other functionality (e.g. scan chain, extra instructions, interrupts, etc.).

Comments

- Plan your team's time in advance. It might take a while before you get used to Verilog syntax. Debugging could be time-consuming, since Verilog does not give any details about your bugs.

Requirements

Please turn in the following in your **cad8** directory:

- Synthesizable Verilog model(s) for your controller with associated testbench (*.v). Put comments in your code to ease readability.
- *NCVerilog* simulation (*.apr.v + *.apr.sdf) of all the instructions in the postscript format. You may have additional instructions, interrupts or special operating modes that are not part of the basic requirements. While you should design your controller to support any of these extras, you do not have to demonstrate their proper operation for this assignment. You will have to show proper operation of these for your final project demo, but not for **cad8**.
- Gate-level netlist for your controller (*.apr.v).
- Synopsys timing/area constraints for your controller. Include info in your README describing how you derived the timing constraints.
- The timing report from Design Compiler. Discuss the worst-case path and any paths that don't meet timing constraints (*.dc.rpt).
- Complete schematics including the controller, datapath, and memories.
- Complete layout for the controller only (don't create a layout yet for the controller integrated with datapath, memories, etc.)
- DRC and LVS reports for the controller. You do not need to run PEX.
- A README file describing your controller design. Discuss how you arrived at your timing constraints, any extra instructions, features, etc. that are not part of the baseline and any other design considerations you took into account.

Deadline

You need to turn CAD8 in by **Wednesday, November 18, 2009, 7pm**.