EE108B Lab 3

Vince Sparacio, Atsu Kobashi

Feb 26, 2013

Introduction

In order to implement this new pipelined processor with forwarding and stalls, we made changes to the decode.v file and the mips_cpu.v file. In decode.v, we added code to forward correctly from the EX and MEM stages, to deal with RAW hazards. We also added the logic to determine when it is necessary to stall when we have a load-use hazard. In mips_cpu, we just rewired the decode module to give it a few extra inputs needed to determine forwarding conditions. We also attempted to manually translate our pong assembly code from Lab 1 to the format needed in the irom file.

Design

For the MIPS processor, the changes we made were to add the following functionalities in the decode.v module. In terms of adding signals to the decode modules, we chose to add alu_result_ex,reg_write_data_mem,reg_write_addr_ex, reg_write_addr_mem, reg_we_ex, reg_we_mem, and mem_read_ex so that we could perform the following logic operatoins.

Forwarding from EX stage

We checked to see if the destination register of the EX stage was the same as either the rs or rt register in the decode stage, and forwarded if it was. We also had to make sure that the destination register was not the zero register, and that the reg_we was high for the EX stage.

Forwarding from MEM stage

We checked to see if the destination register of the MEM stage was the same as either the rs or rt register in the decode stage, and forwarded if it was. We also had to make sure that the destination register was not the zero register, and that the reg_we was high for the MEM stage. In addition, we needed to make sure that if there was a more recent instruction writing to that same register in the EX stage, that we would not forward the old value in the MEM stage and instead forward the value from the EX stage. (this part of the condition is shown in bold below)

For example, for rs: MEMForwardRs= (reg_we_mem && (reg_write_addr_mem != `ZERO) && **!(reg_we_ex && (reg_write_addr_ex != `ZERO) && (reg_write_addr_ex == rs_addr)) && (reg_write_addr_mem == rs_addr)**);

Stall condition

The stall condition was relatively simple in that we only had to check if there was a load-use hazard which would be caused by a memory-read in the MEM stage indicating a load instruction, and if the destination register of that load was either the rs or rt register in the decode.

Results

It seemed that the MIPS processor we built worked successfully since it forwarded and stalled correctly when we tested it with instructions that caused hazards. In the Simulations section we've attached the waveforms indicating the successful operation of the processor with instruction dependencies that required forwarding, and load-use hazards that required a stall.

However, we were not successful in translating the pong.s MIPS assembly code to the irom representation. We unfortunately did not use any code translator for this task. Even though pong did not work, we were able to run the assembly code from Lab 2 on our processor as a demonstration to the TA.

Conclusion

We thought that implementing the MIPS processor required a lot of careful thinking but in the end was not a lot of coding. On the other hand, the translation of the pong assembly file from Lab1 to the irom format was a lot of work that could have been made simpler with a translation program like the one posted on Piazza. We are confident that with a few more tweaks to our pong code, it would function fine on our processor.

Simulations

Definition of Pong ROMs:

pong_irom: the synthesized rom which includes last week's test mechanism

pong_irom_VS: the attempted pong implementation in mips; not functional, however, it is almost complete

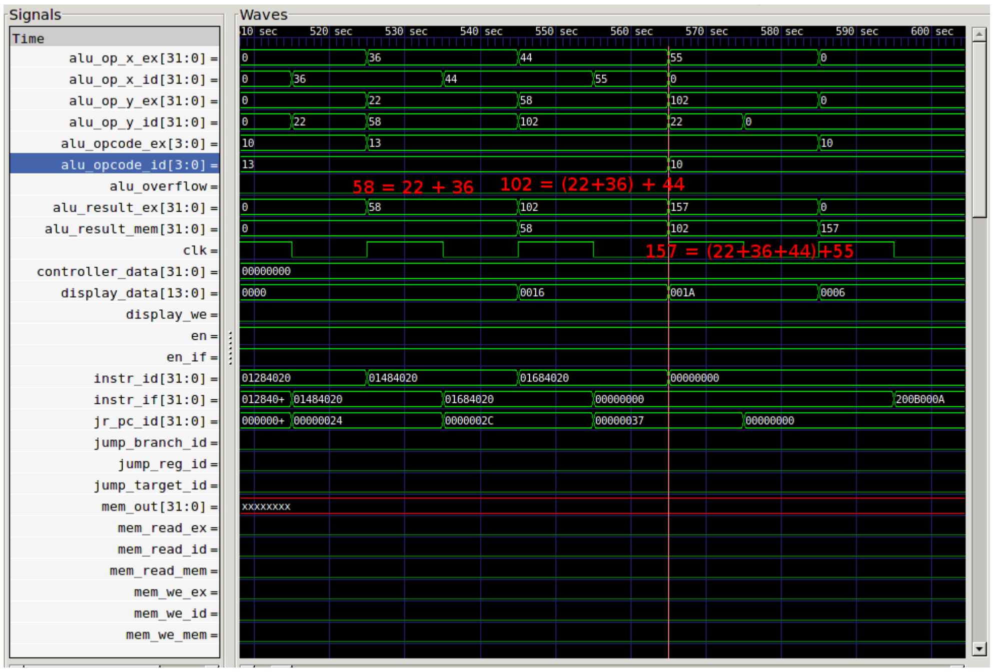
irom: this is what we used to test the pipelined processor

Below is a waveform demonstrating the correct forwarding of register values from the EX stage of one instruction to the deocde stage of the next. There were three consecutive instructions:

add $t0, $t0, $t1;

add $t0, $t0, $t2;

add $t0, $t0, $t3;
As you can see, the processor correctly forwarded the most recent values of t0 to the appropriate instructions.



We also did a similar test for load-use hazards, where the processor correctly stalled for one cycle.

Additional Questions

1) 74.892 Mhz is the maximum frequency, according to ISE

2) Our implementation needed to be changed drastically from that of lab 2. First was the use of pseudoinstructions in our python scripts that were converted into mips assembly code. As we found, this new format of entering instructions also created the chore of making sure that off of the inputs rs, rt, rd, etc. were in the appropriate locations. The next adjustment was how the rom interfaced with the screen and printed the output. Because we are no longer reading a python script, we needed to concatenate the color and x/y coordinates into one 32 bit number before storing the value in the address FFF00C.

Finally, because our new pipelined processor expects a stall after every branch operation, we had to revise our code to include nop’s after every branch operation.

3) Longest Combinational delay: 4.481ns; In order to cut down on this combinational delay, we can put a flip flop in the critical path to cut down on the delay per clock cycle.

4) In order to eliminate the stalls after a load instruction, it would be create several other hazards throughout the pipeline. Because the only solution would be forwarding (or switching the memory and execute stages), and forwarding a result backward in time causes system instability, it would not be wise to try and get rid of the stalls after a load.