Nick Robinson

Matt McDonald

EECE 3324

Final Project

The challenge was to create a partial simulated standard 5-stage MIPS architecture in verilog targeted against the HDL simulator Modelsim. The main challenges of the project could be broken up into the development of four pieces, a single cycle datapath, a multi-cycle 5 stage datapath, a forwarding unit, and a control hazard unit. Then, simulation and analysis showed us the benefit of a pipelined datapath over a single cycle one despite the extra complexity.

The first stage of the development was to implement a simple singe-cycle datapath. The end result matched up quite nicely with **Figure 1** in the project guidelines sheet. Most of this process involved the simple creation of components, such as the MIPS register bank, ALU, memory interface, and other datapath components. There were some challenges with register writing and timing and making sure that register writes happened before reading occurred. The ALU was fairly easy to implement with higher level verilog encapsulated the details inside the ALU. However, due to the nature of this project, there was no onboard memory inside of the MIPS CPU and instead, we needed to expose an external interface outside of the CPU package to a provided *memory.v* implemented memory module. After, it was merely a process of wiring components together and writing clean verilog. However, due to the profound lack of existence of a verilog environment that can provide basic code writing necessities in rapid development (not just accepting a miswired label as a new net and then throwing all sorts of logical errors and not telling the writer about it), most of this part of the project was spent renaming and matching up nets correctly.

After a single-cycle datapath was working properly, we had the challenge of making it faster, thus implementing a pipelined processor. In order to do this, we split up the 5 stages of

the processor into instruction fetch, instruction decode, execution, memory, and register writing. There were also registers added in between IF/ID, ID/EX, EX/MEM, MEM/WB that allowed for a pipeline of instructions to pour through the CPU. With all of these registers, the number of wires in the CPU module exploded and thus we attempted to implement a basic naming convention <REG_ONE>_<REG_TWO>_<COMPONENT_NAME> where all component names for the same signal in the database where the same and register names constant. This saved us quite a bit of time with simple net naming issues. However, in creating the pipeline, we opened a slew of issues emanating from control and data hazards. In order to fix these, we created two control units, a hazard control unit and a forwarding control unit.

The hazard control unit was implemented to control the insertion of NOPS when necessary to stall the pipeline when a specific instruction collided with another and the stalls were necessary to be in place until forwarding could take over and deliver the proper result. It inserted NOPs by stalling all Address register changes and also inserting NOPS into the control signals that propagated through the registers through the later pipeline so any other remnant data signals were simply DON'T CARE. Also on this note is the changes we needed to make to the jump and branch logic. When a branch or jump occurs in the pipeline, the basic datapath was changed to calculate branching at the Decode state so that it wouldn't have to wait until the execute state. By doing this, we only need to flush one instruction from IF/ID when a branch or jump occurs. Thus, we take the assume not jump/branch and flush afterwards approach, minimized as much as possible.

The forwarding unit was as simple as an algorithm. Although it rests in a logic execute state, it pulls from both MEM_WB and ID_EX to calculate when two instructions are colliding. If a second instruction is requesting the data of an instruction that is writing to the registers, the pipeline is stalled to a minimum so that before the data is written to the memory, is is forwarded to the needing instruction in the execute state. The actual replacement with correct values was

just implemented via two mux’s whose control signals emanate from the forwarding unit and only override the ALU inputs when certain criteria (forwarded values needed) are met.

The results of the project are as follows:

Total # of cycles: 123

Total # of instructions: 100

CPI: 1.23

Register file contents:

R0: 0x00000000 R1: 0x00000000

R2: 0x00000020 R3: 0x00000012

R4: 0x00000000 R5: 0x00000000

R6: 0x00000000 R7: 0x00000000

R8: 0x00000003 R9: 0x00000006

R10: 0x00000009 R11: 0x0000000c

R12: 0x0000000f R13: 0x00000012

R14: 0x00000015 R15: 0x00000018

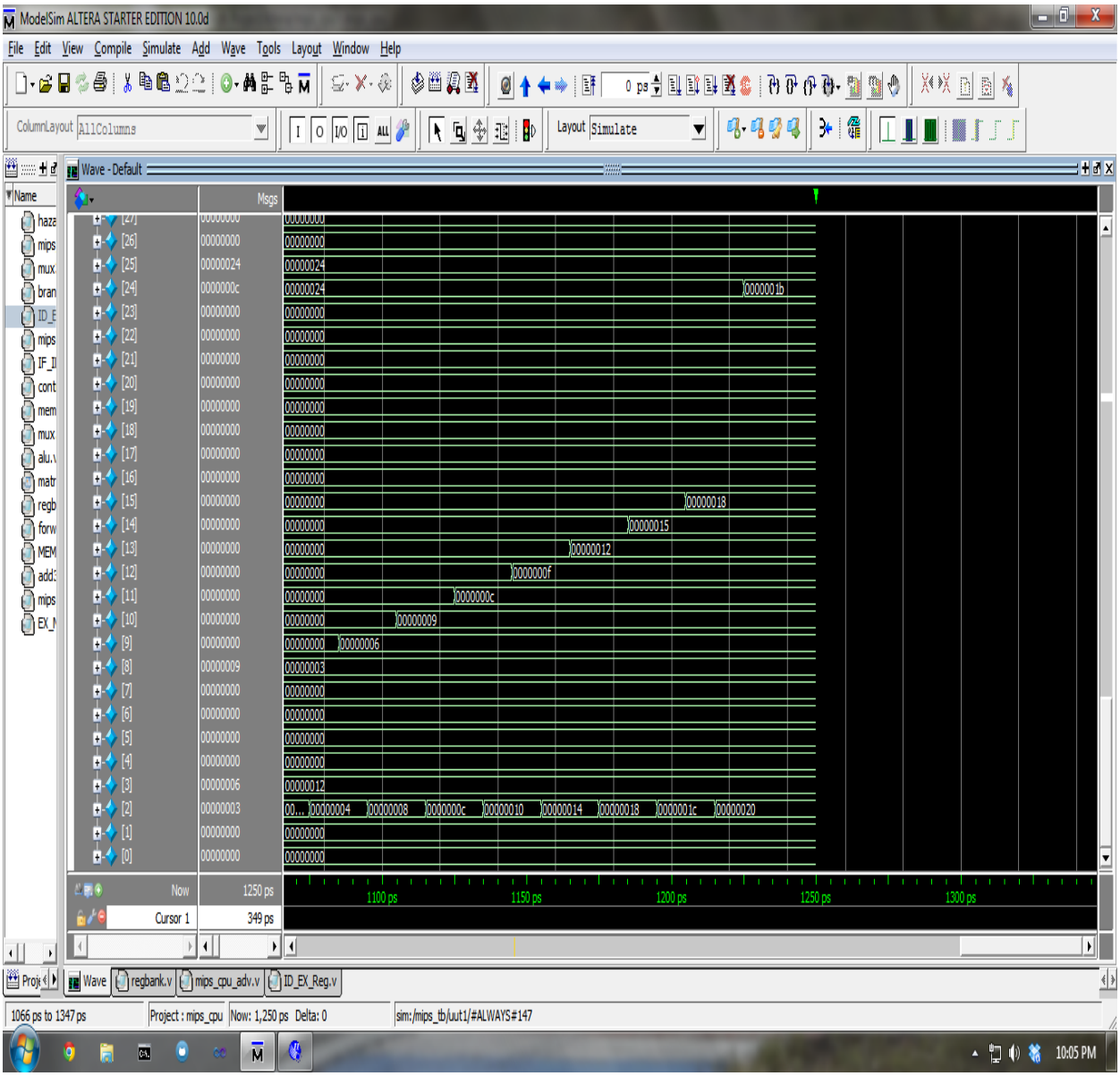
R16: 0x00000000 R17: 0x00000000

R18: 0x00000000 R19: 0x00000000

R20: 0x00000000 R21: 0x00000000

R22: 0x00000000 R23: 0x00000000

R24: 0x0000001b R25: 0x00000024


ModelSim ALTERA STARTER EDITION 10.0d
File Edit View Compile Simulate Add Wave Tools Layout Window Help
Wave - Default
Name
Msgs
Now
1250 ps
Cursor 1
349 ps
1100 ps
1150 ps
1200 ps
1250 ps
1300 ps
Wave
regbank.v
mips_cpu_adv.v
ID_EX_Reg.v
1066 ps to 1347 ps
Project : mips_cpu
Now: 1,250 ps Delta: 0
sim:/mips_tb/uut1/#ALWAYS#147
10:05 PM