Chase Andersona
EE4363 MP3 REPORT

Code Modification:

For my code implementation I had to modify the wire assignments for many of the bypass/forwarding signals. By using these forwarding mechanics we do not require the use of nops to delay the operation of consecutive instructions with noted data dependencies. We also had to manipulate the Ain and Bin to check for the 3 respective bypass checks to see if the Ain/Bin could be using a previous element of the pipeline segments, in turn allowing for a more efficient pipeline structure. No modifications were made to the R-Type instruction set for the testing arrangement that did not look to require it, and we were still able to properly display the power of forwarding in contrast to non-nop data hazards. All of the bypass conditions were met with assistance from the textbook and its annotations. Most all of it made sense with reference to the correlated bit fields and through extra testing, the system seemed to be responding correctly.

Experiments:

My testing initially consisted of working through the faulty code with incremental adjustments to the each element to watch it affect both the GTKWAVE waveforms as well as implementing $display code lines in the verilog testbench that allowed me to also watch the register and data memory contents as the clocks cycles progressed. I thought that this was the most efficient way to confirm functionality of the system, for each time a data dependency was induced you were able to directly extract it through the resulting regs and mems. Then from the instruction memory contents that caused the failure you could inspect the GTKWAVE viewer and understand what caused the failure, pipeline latch not triggering the bypass or a misuse of rt or rs causing a trigger for the correct application of Ain/Bin. Most of the true analysis will be made in the Data Hazards section of this report.

Listed Initialized Instruction and Data Memory:

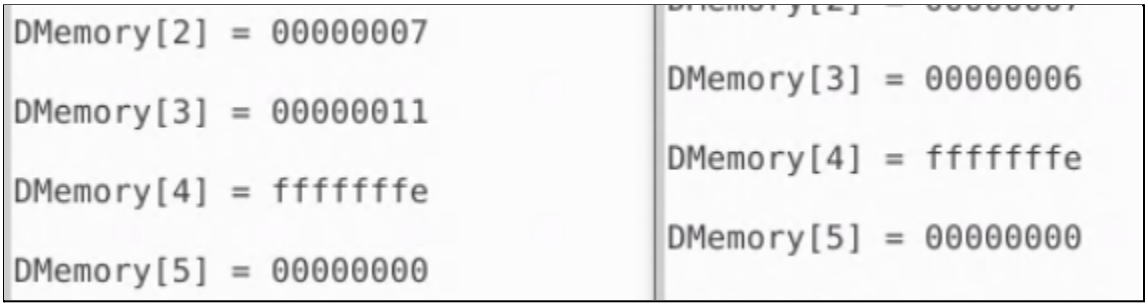
```
IMemory[0] = 32'h00412820; ADD $5 $2 $1
IMemory[1] = 32'h8ca30004; LW $3 0x4($5)
IMemory[2] = 32'haca70005; SW $7 0x5($5)
IMemory[3] = 32'h00602020; ADD $4 $3 $0
IMemory[4] = 32'h01093020; ADD $6 $8 $9
IMemory[5] = 32'hac06000c; SW $6 0xC($0)
IMemory[6] = 32'h00c05020; ADD $10 $6 $0
IMemory[7] = 32'h8c0b0010; LW $11 0x10($0)
IMemory[8] = 32'h00000020; ADD $0 $0 $0
IMemory[9] = 32'h002b6020; ADD $12 $1 $11

DMemory[0] = 32'h00000000;
DMemory[1] = 32'hffffffff;
DMemory[2] = 32'h00000000;
DMemory[3] = 32'h00000000;
DMemory[4] = 32'hfffffffe;
```

Data Hazards:

Starting from the beginning of instruction memory, with instruction 0 there exists data inputted into reg5. Then, in the consecutive two instructions 1 and 2 we can see that reg5 is in use for calculating memory locations. Within the old code this data was still successfully pushed and pulled from memory, also this was done successfully with the updated code, I believe that in the old code we can still see the use of Ain bypass wires being triggered which also will affect the code notwithstanding the changes made for MP3. This can be seen at the 3rd and 5th tick of the clock that bypassAfromMEM and bypassAfromALUinWB are both triggered respectively.

Old Code Waveforms

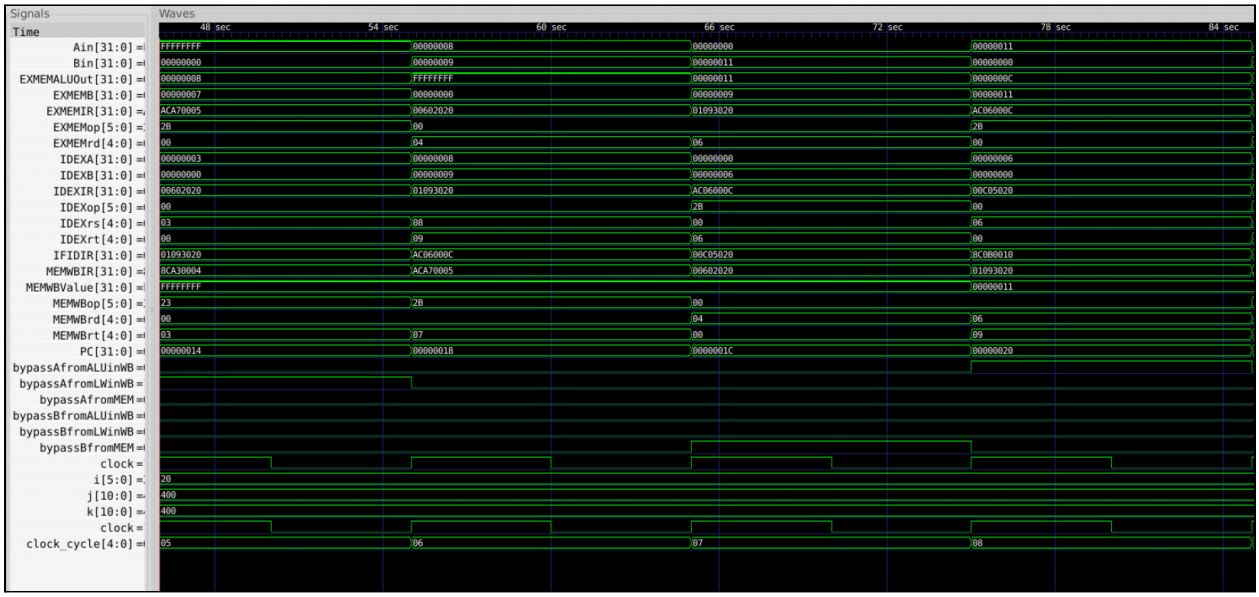
Next, we can look to 4 where the addition of the numbers 8 and 9, help in reg8 and reg9 respectively, are summed and placed into reg6, the hex number 0x11. Both new and old code versions properly place 0x11 into reg6, however the following data dependency is cause for a hazard. As can be seen in the following screenshot, instruction 5 stores the contents of reg6 into the data location 0xC(reg0) which is the address decimal 12 but is bit-shifted twice causing a resulting location of dataMem3. Here we can see the data hazard results:



New Code Left / Old Code Right

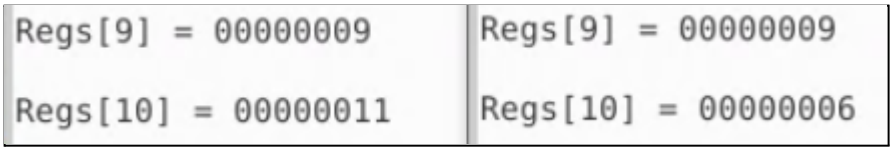
With this example we can clearly see that the data hazard has been caused by reg6 not being updated directly from the

additional instruction, and therefore still hold and place the old contents of 6 into the data memory defined. Here we can also see the bypass at work with this pair of instructions here in the new code's GTKWAVE:



New Code Where Bypass Assists Data Hazard

In the screenshot above we can see that the newly implemented bypassAfromALUinWB as well as the bypassBfromLWinWB both trigger along clock cycles 8 and 7 respectively showing exactly where the advantage of forwarding takes place to alleviate the system from a data hazard caused from dependency. Similarly instruction 6 also shows data hazards due to dependencies. We can see below:



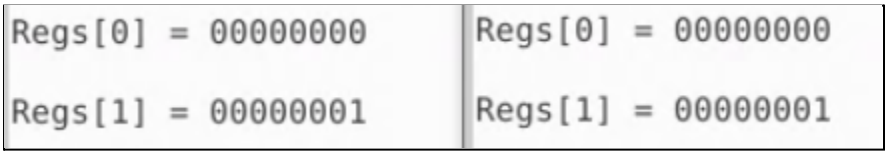
New / Old

Here we can see that the addition has still not written back to reg6 and therefore when we make an addition of reg6 and reg0 we still see a 6+0=6 which means that the pipeline has a data hazard that must be dealt with using the bypass/forwarding mechanism used in MP3. Now with instruction 7 we can see a simple load word instruction that has no issues with data hazards and will succesffuly load the contents of 0x10(reg0)

which when using the modified MIPS convention in this code, we will be loading in the word from address 16/2/2 which will be address 4, dataMem4[0xFFFFFFFE], and as we can see:

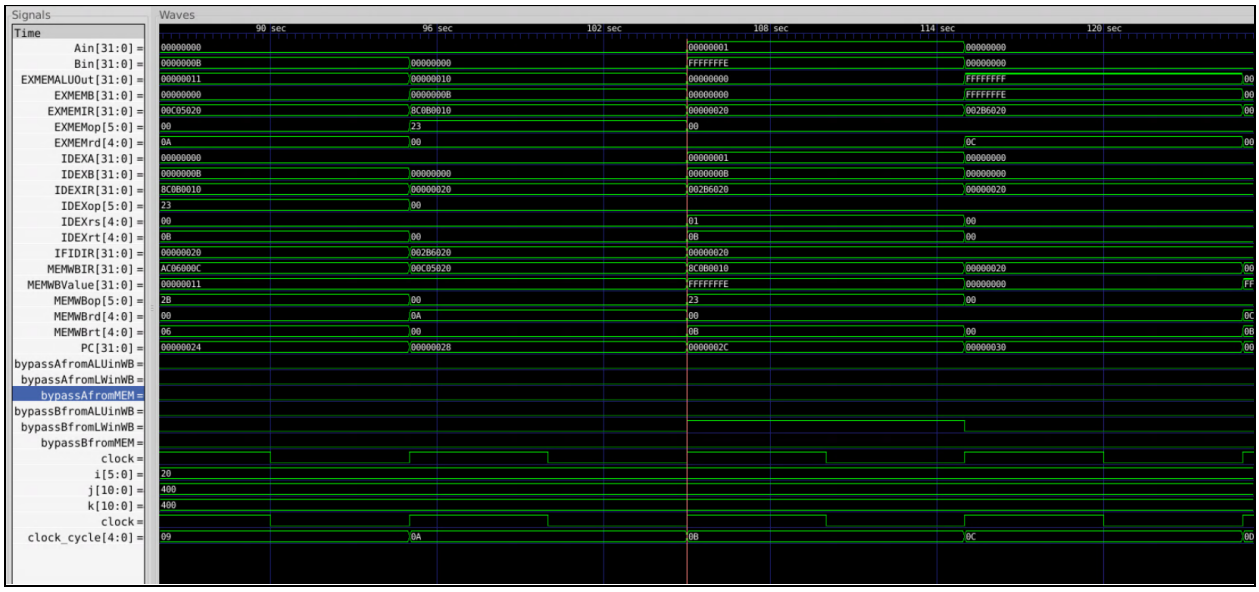
New / Old

This instruction was properly executed. Instruction 8 was simply just the addition of zeros and replaced with the zero that was already held inside reg0. We see the result of this operation in reg0:



New / Old

Finally, we have the last data dependency with instruction 9. Here this dependency is a data hazard for the required operation above for the contents of reg11 will not be complete by the time the code requests its data in the pipeline. Here we can see in the new code that the dependency is accounted for using the forwarding trigger of bypassBfromALUinWB which is not accounted for in the original old code. 1 + FFFFFFFE = FFFFFFFF, otherwise we get 11+1=12=0xC which is funnily enough also what is held inside of the 0xC/12th reg. In this screenshot it can be seen triggered at the 11th clock cycle as well as the resulting regs which were corrupted and corrected using bypass:



New Code with Triggered Bypass

Regs[11] = fffffffe	Regs[11] = fffffffe
Regs[12] = ffffffff	Regs[12] = 0000000c

New / Old Regs

Code Adjustments Explicit:

```
  // The bypass to input A from the MEM stage for an ALU operation
  assign bypassAfromMEM = (IDEXrs == EXMEMrd) & (IDEXrs!=0) & (EXMEMop==ALUop);
  // The bypass to input B from the MEM stage for an ALU operation
  assign bypassBfromMEM = (IDEXrt == EXMEMrd)&(IDEXrt!=0) & (EXMEMop==ALUop);
  // The bypass to input A from the WB stage for an ALU operation
  assign bypassAfromALUinWB = (IDEXrs == MEMWBrd) & (IDEXrs!=0) & (MEMWBop==ALUop);
  // The bypass to input B from the WB stage for an ALU operation
  assign bypassBfromALUinWB = (IDEXrt == MEMWBrd) & (IDEXrt != 0) & (MEMWBop ==
ALUop);
  // The bypass to input A from the WB stage for an LW operation
  assign bypassAfromLWinWB = (IDEXrs == MEMWBIR[20:16]) & (IDEXrs!=0) & (MEMWBop==LW);
  // The bypass to input B from the WB stage for an LW operation
  assign bypassBfromLWinWB = (IDEXrt == MEMWBIR[20:16]) & (IDEXrt!=0) & (MEMWBop==LW);

  // The A input to the ALU is bypassed from MEM if there is a bypass there,
  // Otherwise from WB if there is a bypass there, and otherwise comes from the IDEX
register
  assign Ain = bypassAfromMEM? EXMEMALUOut : (bypassAfromALUinWB | bypassAfromLWinWB)?
MEMWBValue : IDEXA;

  // The B input to the ALU is bypassed from MEM if there is a bypass there,
```

```
  // Otherwise from WB if there is a bypass there, and otherwise comes from the IDEX
register
  assign Bin = bypassBfromMEM? EXMEMALUOut : (bypassBfromALUinWB | bypassBfromLWinWB)?
MEMWBValue : IDEXB;
```