Overcoming data-hazards using forwarding and stalling

Design

In prior laboratory sessions, I developed various parts of an ARM Legv8 processor and established a pipelined cycle framework for the processor. It was noted in earlier work that this setup was unable to manage data dependencies or perform hazard detection and forwarding. In the current lab, I have integrated additional hardware components designed to identify hazards and execute forwarding operations: these include the Hazard Detection Unit (HDU) and the Forwarding Unit. Refer to the accompanying diagram for more details:

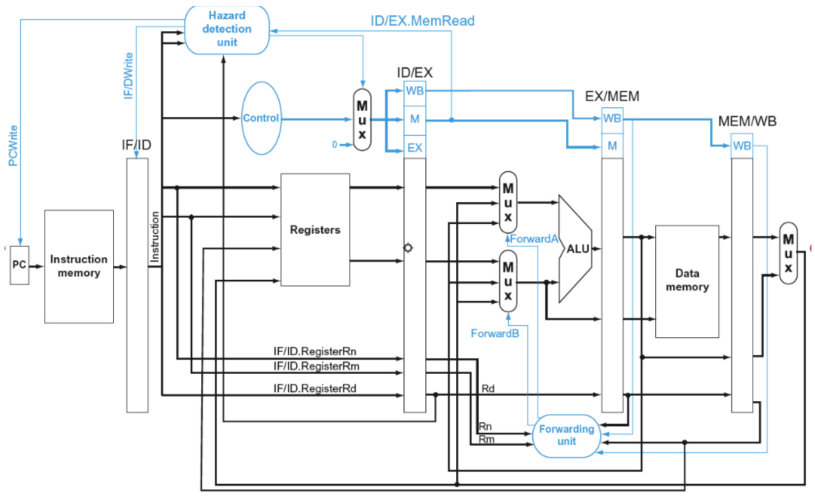

Figure 1: **Pipelined control overview showing the forwarding and hazard detection logic.** Note that this diagram leaves out some of the details of the full datapath. [Figure 4.7.4 from the textbook]

Figure 1: 5 stage pipelined ARM processor with forwarding and hazard detection logic

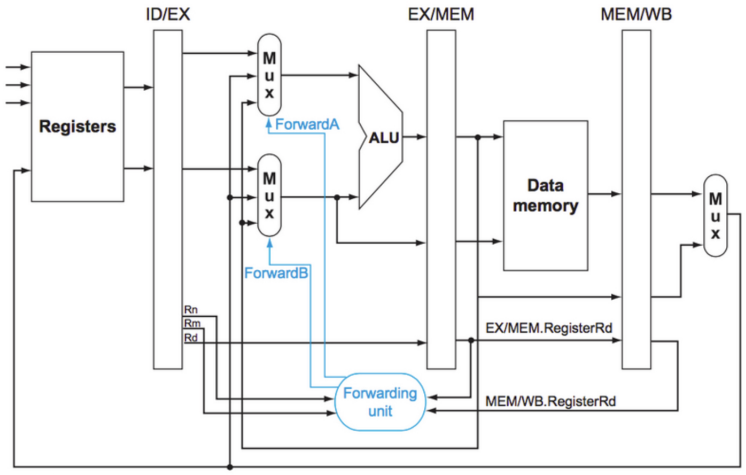

Figure 2: **Close-up of the forwarding circuitry in the datapath of Figure 1.** [Figure 4.7.2.B from the textbook]

Figure 2: Close-up of Forwarding circuitry from Figure 1.

pipelinedCPU1

The processor's design was meticulously informed by the preceding diagram. Central to the design process was the understanding of how each component interacted with the newly introduced forwarding and hazard detection mechanisms. Ensuring accurate connections and proper integration of any necessary intermediate logic was essential. Detailed signal labeling proved extremely helpful in minimizing connection-related design errors.

Modifications to the existing files from pipelinedCpu0 were minimal, largely confined to incorporating the forwarding and hazard detection systems and adjusting the connections between components. For instance, within the CPU Control module, there was no need to alter the existing logic; however, careful consideration was given to the redefined inputs and outputs. In this setup, all outputs from the CPU control were directed to a newly established multiplexer, named muxControl. Additionally, two fresh multiplexers, aluMux1 and aluMux2, were introduced prior to the ALU to facilitate the forwarding logic, serving the top and bottom multiplexers, respectively.

Below is an elaboration of the specific modifications undertaken and the rationale behind each change.

PC

A new signal, named PCWrite, has been introduced to the program counter (PC) module, originating from the newly implemented Hazard Detection Unit (HDU). The operational logic for this addition is straightforward: during the rising edge of the clock cycle, if the PCWrite signal is active, the PC module will accept and set its output to the provided address. If PCWrite is not active, the module will proceed to increment the counter as usual.

IF_ID

In the program counter (PC) module, an update similar to the previous one has been implemented. This update conditions the values within the module to change only when there's a rising edge of the clock and the IF_ID_Write signal is active. The IF_ID_Write signal is another output from the Hazard Detection Unit (HDU). The implementation of the Write signal can be seen below:

```
--On rising edge, assign intermediate value to out
elsif rising_edge(clk) and IF_ID_Write = '1' then
  IF_ID_pc_out    <= if_id_reg(95 downto 32);
  IF_ID_addr_out  <= if_id_reg(31 downto 0);
end if;
```

Figure 3: IF_ID Implementation

ID_EX

In this section, we modified this component by adding a more descriptive ending to the register/buffer. We explicitly determined what Rm, Rn was along with the already included Rd piece. The implementation of the addition of the signals can be seen below:

```
ID_EX_RegisterRn_in    : in std_logic_vector(4 downto 0);
ID_EX_RegisterRm_in    : in std_logic_vector(4 downto 0);
ID_EX_RegisterRd_in    : in std_logic_vector(4 downto 0);
```

Figure 4: ID_EX Implementation

hazardDetectionUnit (HDU)

The Hazard Detection Unit (HDU) features four inputs and three outputs, as depicted in the accompanying diagram. The implemented logic functions as follows: to stall the pipeline, the HDU sets the PCWrite (the write enable signal for the program counter) to 0, effectively pausing updates to the program counter. Concurrently, it stalls the IF_ID register, which receives data from the program counter, by setting the IF_ID_Write to 0. Furthermore, when the select signal for muxControl is 0, the ID_EX register inputs should be zeroed, overriding the signals from the CPUControl. Conversely, to maintain normal operation and not stall the pipeline, these signals are set to '1'. The implementation of the HDU can be seen below:

```
begin
    process (all)
    begin
        if ( ID_EX_memRead = '1' and ( (ID_EX_registerRd = IF_ID_registerRn1) or
           (ID_EX_registerRd = IF_ID_registerRm2) ) ) then
            -- stall the pipeline, dont write to the pc counter
            PCWrite <= '0';
            -- dont write to the IF_ID_reg as well because it gets its input from
            IF_ID_write  <= '0';
            -- also the mux control would be i.e dont read in from the CPUControl
            -- see the book for more
            muxControl <= '0';
        else
            PCWrite <= '1';
            IF_ID_Write <= '1';
            muxControl <= '1';
        end if;
    end process;
end behavioral;
```

Figure 5: HDU Implementation

muxCPU

The newly added multiplexer, placed immediately after the CPUControl, operates based on the select signal. If the select signal is '0', the ID_EX register is not populated with values from the CPUControl; instead, it is filled with zeros. When the select signal is '1', the values from the CPUControl are passed through to the ID_EX register. This configuration is part of the stall logic previously described. The implementation of creating a quick mux for the CPU can be seen below:

```
architecture dataflow of muxCPU is
  begin
  process(all)
    begin
    if (sel = '1') then
        CBranch_out   <=  CBranch_in;
        MemRead_out   <=  MemRead_in;
        MemtoReg_out  <=  MemtoReg_in;
        MemWrite_out  <=  MemWrite_in;
        ALUSrc_out    <=  ALUSrc_in;
        RegWrite_out  <=  RegWrite_in;
        UBranch_out   <=  UBranch_in;
        ALUOp_out     <=  ALUOp_in;
    else
        CBranch_out   <= '0';
        MemRead_out   <= '0';
        MemtoReg_out  <= '0';
        MemWrite_out  <= '0';
        ALUSrc_out    <= '0';
        RegWrite_out  <= '0';
        UBranch_out   <= '0';
        ALUOp_out     <= "00";
    end if;
  end process;
end dataflow;
```

Figure 6: muxCPU Implementation

ALUMux1

I stuck to the diagram for guidance, ensuring that the appropriate input signals from the forwardingUnit and other modules were accurately fed into the system. The implementation of this 2-bit selector mux can be seen below:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity mux2to3 is -- Two by one mux with 5 bit inputs/outputs
port(
    in0    : in STD_LOGIC_VECTOR(63 downto 0); -- sel == 00
    in1    : in STD_LOGIC_VECTOR(63 downto 0); -- sel == 01
    in2    : in STD_LOGIC_VECTOR(63 downto 0); -- sel == 10
    sel    : in STD_LOGIC_VECTOR(1 downto 0); -- selects in0 or in1 or in 2
    output : out STD_LOGIC_VECTOR(63 downto 0)
);
end mux2to3;

architecture dataflow of mux2to3 is

begin
    with sel select
        output <= in0 when "00",
                  in1 when "01",
                  in2 when "10",
                  x"----------------"      when others;
end dataflow;
```

Figure 7: ALUMux Implementation

ALUMux2

The same as the one above.

forwardingUnit

The logic for this module was sourced from the textbook by Patterson and Hennessy, in addition to the lab specifications. The fundamental concept is as follows:
- EX/MEM to EX forwarding occurs when forwardA is set to ”10”.
- MEM/WB to EX forwarding happens when forwardA is set to ”01”.
- No forwarding takes place when forwardA is set to ”00”.

The same logic applies to the signal forwardB. Most of the logic was given in the spec, the rest was found in the textbook. The implementation of the forwarding unit can be seen below:

```
architecture behavioral of forwardingUnit is

begin

    process (all)
    begin
        -- EX hazard
        if ( EX_MEM_regWrite = '1'
            and (EX_MEM_RegisterRd /= "11111")
            and (EX_MEM_RegisterRd = ID_EX_registerRn1) ) then
            forwardA <= "10";
        -- MEM hazard
        elsif (MEM_WB_regWrite = '1'
            and (MEM_WB_RegisterRd /= "11111")
            and not (EX_MEM_regWrite = '1' and (EX_MEM_RegisterRd /= "1111")
            and (EX_MEM_RegisterRd = ID_EX_registerRn1))
            and (MEM_WB_RegisterRd = ID_EX_RegisterRn1) ) then
            forwardA <= "01";
        else
            forwardA <= "00";
        end if;

        if ( EX_MEM_regWrite = '1'
            and (EX_MEM_RegisterRd /= "11111")
            and (EX_MEM_RegisterRd = ID_EX_RegisterRm2) ) then
            forwardB <= "10";
        -- MEM hazard
        elsif (MEM_WB_regWrite = '1'
                and (MEM_WB_RegisterRd /= "11111")
                and not (EX_MEM_regWrite = '1' and (EX_MEM_RegisterRd /= "1111")
                and (EX_MEM_RegisterRd = ID_EX_registerRm2))
                and (MEM_WB_RegisterRd = ID_EX_RegisterRm2) ) then
                forwardB <= "01";
        else
            forwardB <= "00";
        end if;
    end process;
end behavioral;
```

Figure 8: Forwarding Unit Implementation

Results and Discussions

The testbench for pipelinedCpu was very simple. We have a clock and rest signals that change value over time. I didn't test any other modules as they were tested in previous labs.

p1

Here I run the test's instruction memory p1_imem with pipelinedCPU1. Here are the instructions for reference:

```
LDUR  X9, [XZR, 0]
ADD   X9, X9, X9
ADD   X10, X9, X9
SUB   X11, X10, X9
STUR  X11, [XZR, 8]
STUR  X11, [XZR, 16]
NOP
NOP
NOP
NOP
```

For reference, here are the initial values of the registers above and the contents of dmem with hex values:

```
XZR = x"0000000000000000"
x9  = x"0000000000000001"
x10 = x"0000000000000000"
x11 = x"0000000000000000"
x12 = x"0000000000000000"

x19 = x"0000000000000000"
x20 = x"0000000000000000"
x21 = x"0000000000000000"
x22 = x"0000000000000000"

DMEM(0x0)  = 1
DMEM(0x8)  = 0
DMEM(0x16) = 0
DMEM(0x24) = 0
```

After running make p1 and opening the ghw file with GTKWave, this is what I observed:

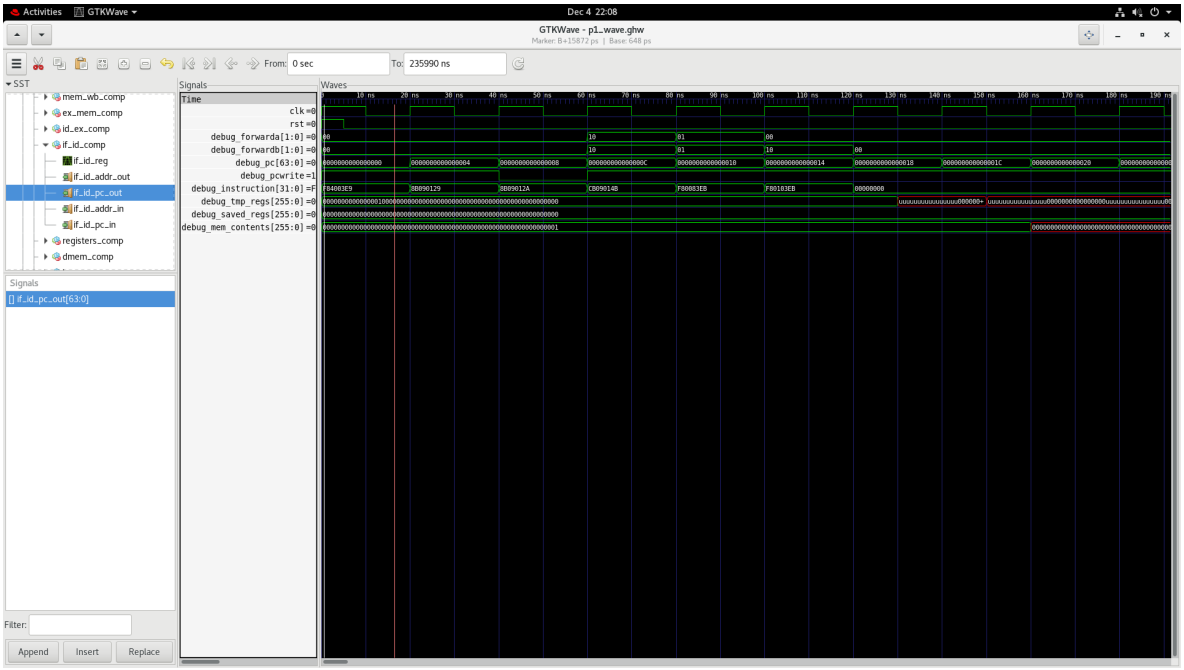

Figure 9: p1 GTKWave results

Cycle ->	1	2	3	4	5	6	7	8	9	10	11	
LDUR X9, [XZR, 0]	IF	ID	EX	MEM	WB							
ADD X9, X9, X9		IF	ID	EX	MEM	WB						
ADD X10, X9, X9			IF	ID	EX	MEM	WB					
SUB X11, X10, X9				IF	ID	EX	MEM	WB				
STUR X11, [XZR, 8]					IF	ID	EX	MEM	WB			
STUR X11, [XZR, 16]						IF	ID	EX	MEM	WB		
NOP												
NOP												
NOP												
NOP												
Instruction	F84003E9	8B090129	8B09012A	8B09012A (stall)	CB09014B	F80083EB	F80103EB	0	0	0	0	0
Cycle ->	1	2	3	4	5	6	7	8	9	10	11	12
LDUR X9, [XZR, 0]	IF	ID	EX	MEM	WB							
ADD X9, X9, X9		IF	ID		EX	MEM	WB					
ADD X10, X9, X9			IF		ID	EX	MEM	WB				
SUB X11, X10, X9					IF	ID	EX	MEM	WB			
STUR X11, [XZR, 8]						IF	ID	EX	MEM	WB		
STUR X11, [XZR, 16]							IF	ID	EX	MEM	WB	
NOP												
NOP												
NOP												
NOP												

Figure 10: CREDIT: Remy Ren. Logical walk through each instruction and the stages/hazards at each cycle.

FULL DISCLOSURE: After countless hours in the Lab and after talking and working on my implementation with Duc for many more hours, I have encountered issues that are unknown and remain unknown. Current speculations is a timing issue with ID/EX's ALU Op code that compounds into a few undefined errors later in the program.

However, with whatever I have so far, I can walk through the process that should work once that one issue has been solved.

Now, to look at what I do see happening in my simulation of GTKWave:

- The clk and rst cycles are working perfectly according to spec
- Throughout the program, PC is incrementing by 4 correctly
- PC write is working properly, only going "low" for one cycle before going back to "high" for the rest of the program.
- Instruction seems to be going through properly as it aligns with the partial solution from the spec: "F80103EB" at PC instruction 0x14.
- ForwardA and ForwardB are changing depending on the hazard detected (or no hazard).

Now for p2, here are the instructions given:

```
CBNZ X9 2
LSR X10, X10, 2
ANDI X11, X10, 15
LSL X12, X12, 2
ORR X21, X19, X20
SUBI X22, X21, 15
```

Using the same DMEM and initial values for registers, here is the GTKWave produced by my implementation:

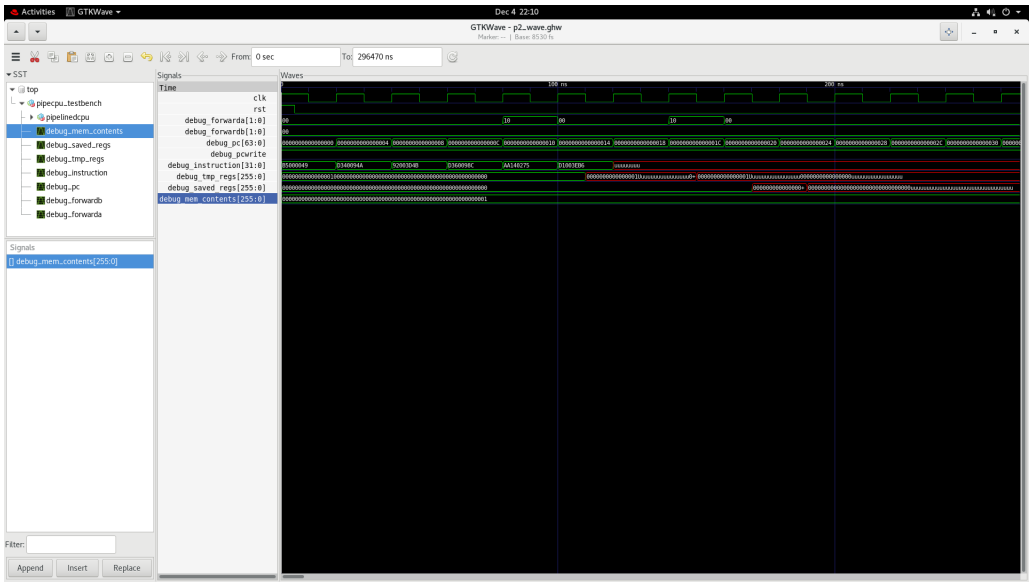

Figure 11: p2 GTKWave results

QUESTION: As seen from the GTKWave, there are lots of errors "undefined" after a certain point in this program. At a first look, it may seem like an issue with the implementation, but looking deeper, we can see what the code is supposed to do and any errors that may arise from pipelining this specific sequence of instructions. Upon further investigation, it can be observed that the first instruction is CBNZ X9 2. This is a branch that should be taken, however, since the fact that the branch should be taken is only computer in the MEM stage, the next two lines are already computed and stored in the IF/ID/EX buffers/registers. Once the branch is jumped and the code starts to execute, the prev instructions haven't been FLUSHED out, leading to errors. To fix this, a more sophisticated implementation must be done that includes the ability to flush out a set of instructions that aren't supposed to be run/used because of a branch jump.

Conclusion

Throughout this lab, I learned a lot about implementing hazard detections for a pipelined CPU. I learned how to design and map all the wirings and connections between components. Although I was ultimately not able to achieve the properly expected results from the GTKWave simulation due to an unfortunate series of events that led to the eventual uncertain unknown that is causing a series of unwanted results to be currently shown. Through hard debugging with Duc (TA) and countless hours working with my peers, I believe my implementation logic works perfectly, but because of a timing error with ID/EX and the ALU Op code, one cycle of instructions was skipped and a few others were "misaligned", thus leading to undefined errors. With a little more time and more help from the Professor and TAs, I hope to be able to identify and patch this unfortunate bug in my implementation. All-in-all, I believe this was a success and a great learning experience.