Laboratory Report

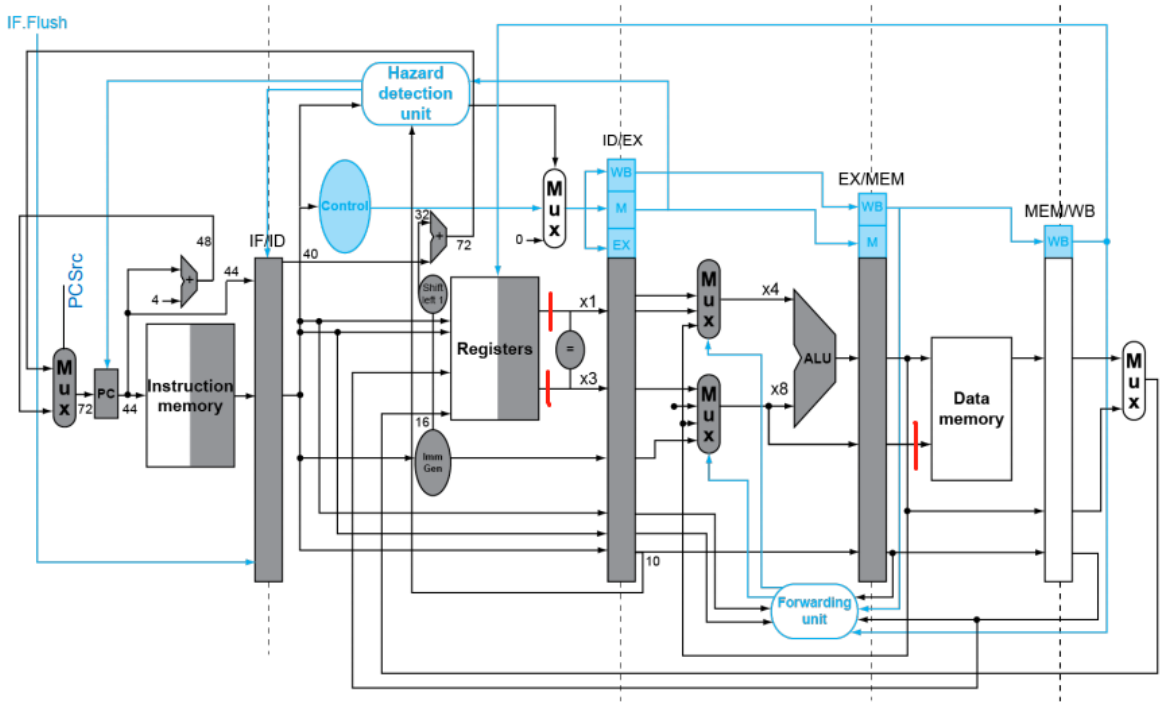
Lab 5

Resolving Hazards

Chunyu Wang, Weiqing Xu, Wenbo Yu
Date: November 18, 2022

1. Brief description of the modeling and implementation of the processor

1.1. Our modified pipelined processor is designed to solve data hazards and control hazards on the basis of previous design in Lab4. The most significant modification is to add a Forwarding Unit, a Hazard detection unit, and 5 muxes. The basic structure is shown below:



1.2. For Forwarding Unit, it mainly contains the control of 5 new muxes: The first two is S-to-ALU data hazards. S can be ALU (EX hazard) or Data Memory (MEM hazard). The basic logic to verify where there occurs these two kinds of hazards is shown below (only Forwarding A):

```
//EX
if (EX_MEM_RegWrite && (EX_MEM_RegRd!=0) && EX_MEM_RegRd == ID_EX_r1)
   Forwarding_A = 2'b10;
//MEM
else if (MEM_WB_RegWrite && (MEM_WB_RegRd!=0) && MEM_WB_RegRd == ID_EX_r1)
   Forwarding_A = 2'b01;
else
   Forwarding_A = 2'b00;
```

The next two part of Forwarding Unit is for S-to-Judge_After_register. Since we move the branch condition judgement part to ID stage in order to solve control hazard. Some new data hazards may occur when there is an R-type instruction followed by another one and then a branch instruction. The basic logic to verify where there occurs this kind of hazards is shown below (only read_data1):

```
if(IF_ID_branch && EX_MEM_RegWrite && (EX_MEM_RegRd!=0) && EX_MEM_RegRd == IF_ID_r1
    Forwarding_Com1 = 1'b1;
else
    Forwarding_Com1 = 1'b0;
```

The last is for Mem-to-Mem hazard. This one is for hazard on rs2 of sw instruction. The basic logic is shown below:

```
if (MEM_WB_RegRd == EX_MEM_r2 && EX_MEM_MemWrite)
    MemSrc = 1'b1;
else
    MemSrc = 1'b0;
```

1.3. For Hazard Detection Unit, we assume branch is always not taken, and flush when the assumption is wrong. The flush is executed through 3 control signals, PC_write, IF_ID_write, and Mux_Control. They are originally set to 1 when normal cases (no hazards). In the first part, we detect load-use hazards by the following logic:

```
if (ID_EX_MemRead && !ID_MemWrite) begin//LW HAZARD
    if (ID_EX_RegRd == IF_ID_r1 || ID_EX_RegRd == IF_ID_r2) begin
        PC_write=1'b0;
        IF_ID_write=1'b0;
        Mux_Control=1'b0;
    end
```

The second part is to solve the data hazards when the former instruction of a branch is R-type or lw. It could not be solved through forwarding unit so that we have to stall and wait for the former instructions to keep going. Notice that it should be stalled until there two instructions between lw and branch if data hazards occur. The logic is shown below:

```
else if (ID_branch && ID_EX_RegWrite) begin//BEQ HAZARD FOR R/LW
    if ((ID_EX_RegRd!=0) && (IF_ID_r1 == ID_EX_RegRd || IF_ID_r2 == ID_EX_RegRd))
    begin
        PC_write=1'b0;
        IF_ID_write=1'b0;
        Mux_Control=1'b0;
    End
else if (ID_branch && EX_MEM_MemRead) begin//BEQ HAZARD FOR LW (ADD THE SECOND NOP)
    if ((EX_MEM_RegRd!=0) && (IF_ID_r1 == EX_MEM_RegRd || IF_ID_r2 == EX_MEM_RegRd))
    begin
        PC_write=1'b0;
        IF_ID_write=1'b0;
        Mux_Control=1'b0;
    end
```

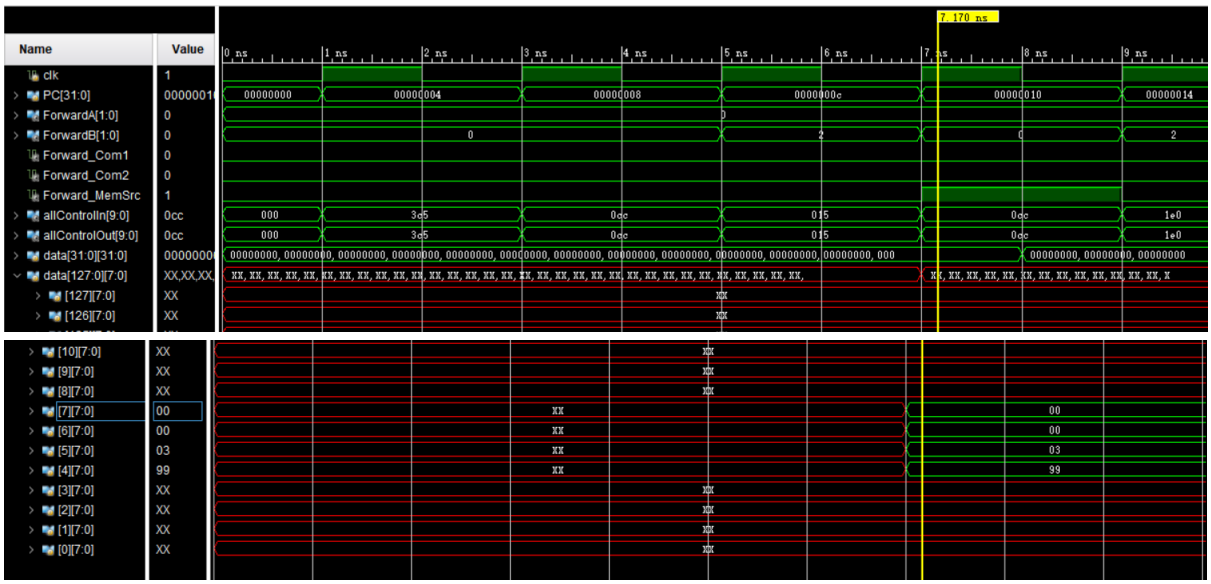
(Notice that for all the code text, we don't show the full code, so the grammar may be not correct)

2. Data Hazards and Control Hazards

2.1. EX Data Hazard

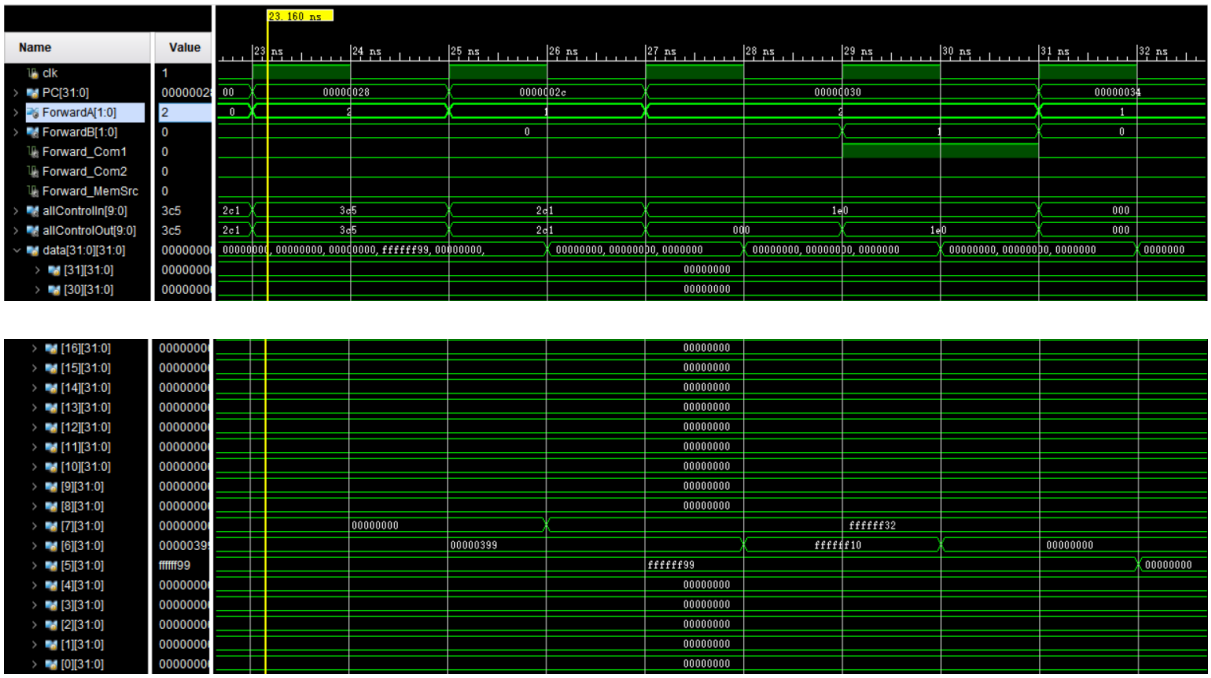
addi t1 x0 0x399
sw t1 4(x0)
...

t1 is modified in the first instruction, and it's called in the second instruction. This is an EX hazard. Forward_B becomes 10 to provide data in t0 in advance. When PC = 0x10, we see that data have been saved in data memory, indicating that the forwarding path has succeeded.



...
add t2 t0 t3
and t1 t2 t3

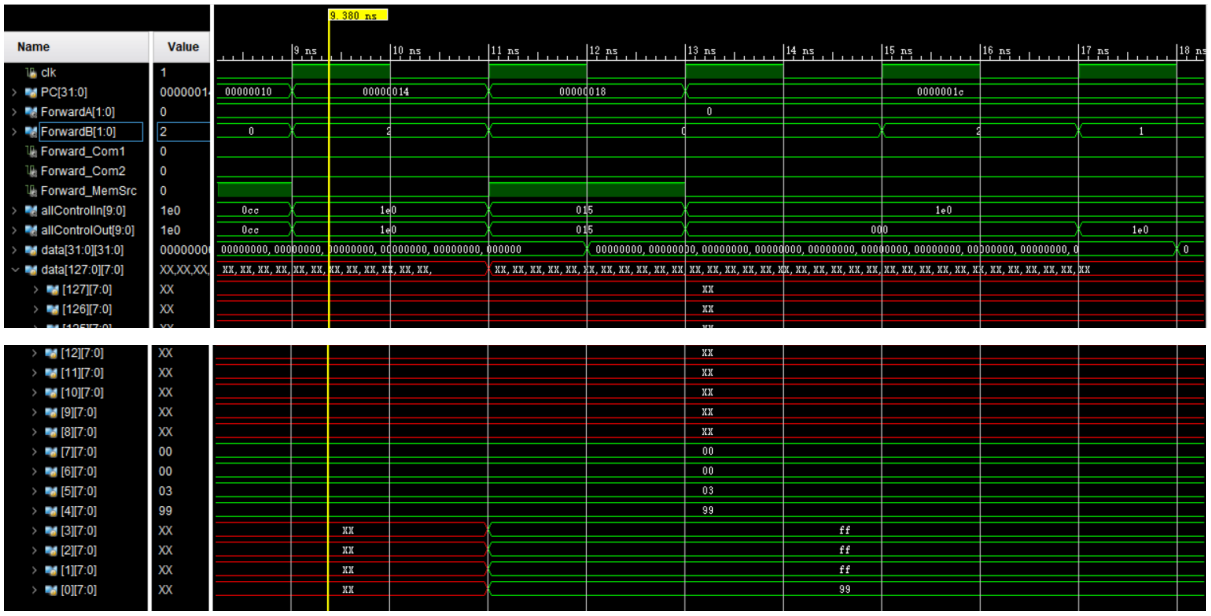
andi t1 t2 0
sub t0 t1 x0
...



In this case, Forward_A becomes 10 to deal with the EX hazards, and contents in t2 & t1 are used by forwarding paths.

2.2. Load-use Data Hazard

...
lb t0 4(x0)
sw t0 0(x0)
...



We can see that PC = 0x1c has been extended, and the result is saved as 0xffffff99 in data memory. Therefore, you see that the load-use hazard is resolved.

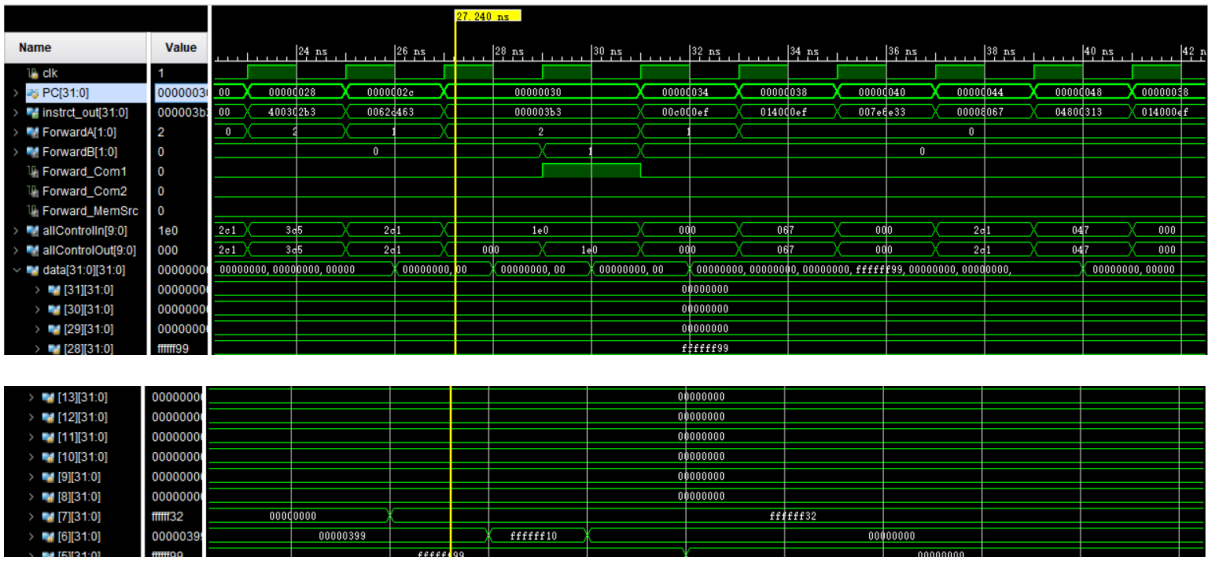
2.3. Flush

...

sub t0 t1 x0

bge t0 t1 right_branch

...



When PC = 0x30, we see that the pc goes on for two clock cycles. This is because of Flush. We assume the branch isn't taken at first, but the fact is that it's taken, so we need to change the target pc. We see that if branch is not taken, x7 should be changed to 0, but the data in x7 remains, meaning that the flush is added and the design is good.

3. Schematic