arm Education

Computer Architecture Course

LAB 5

Forwarding Paths

Issue 1.0

Contents

1 Introduction

1.1 Lab overview

At the end of this lab, you will be able to:

- Outline the limitations of a simple pipeline for Read-After-Write (RAW) data hazards.
- Describe the purpose and operation of forwarding paths.
- Identify the pipeline registers and signals that require forwarding paths.
- Modify the Arm Education Core to implement forwarding paths.
- Verify the functionality of the forwarding path implemented.
- Summarize the advantage and limitations of forwarding paths.

2 Requirements

Before attempting this lab, ensure that you have already completed the installation instructions in the *Getting Started Guide* provided with this course.

The prerequisites for this lab are:

- Familiarity with Arm assembly
- Verilog
- Arm Education Core with simple pipeline—completed Lab 4

Lab 5 is provided with `Lab5_forwarding_pipeline.zip`. This zip file contains supplementary files used in the exercises of this lab and contains the Arm Education Core with a simple pipeline, which was done in Lab 4.

3 Recap: Limitation of the simple pipeline

In our previous labs (Lab 4), we have implemented a simple pipeline and observed limitations whereby:

- The simple pipeline could not cope with Read-After-Write (RAW) data hazards. It was only able to output the correct results if NOPs were inserted in between the instructions with dependencies, which can decrease the throughput at a given time period. See Figure 1.
- NOPs were also used so that the program would not halt prematurely when the YIELD instruction was processed.

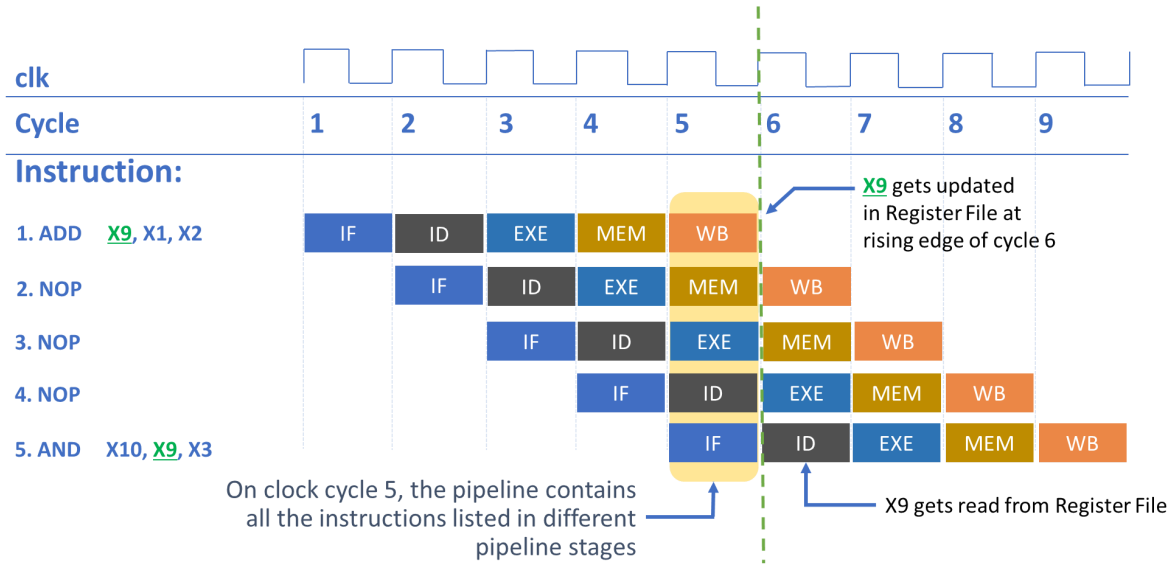

Figure 1: RAW data hazard in the pipeline being resolved with NOPs in the previous lab

There are other types of hazards such as the Write After Read (WAR), Write After Write (WAW), and Control Hazards. For the WAR and WAW hazards, the instruction sequence must be ensured to be in the right order, which the compiler can ensure for this to happen (Arm Education Core is an in-order core, so these hazards cannot happen). Control hazards are also known as branch hazards, where the processor does not know which instruction to fetch next because the branch decision has not been made in the time available to fetch the next instruction.

In this lab, we will explore a hardware architecture technique that can help resolve some RAW data hazards, which are called forwarding paths.

4 Theory: Forwarding paths

One method to solve the RAW data hazard issue is by forwarding the result of an instruction to a certain pipeline stage of the next dependent instruction, as shown in Figure 2.

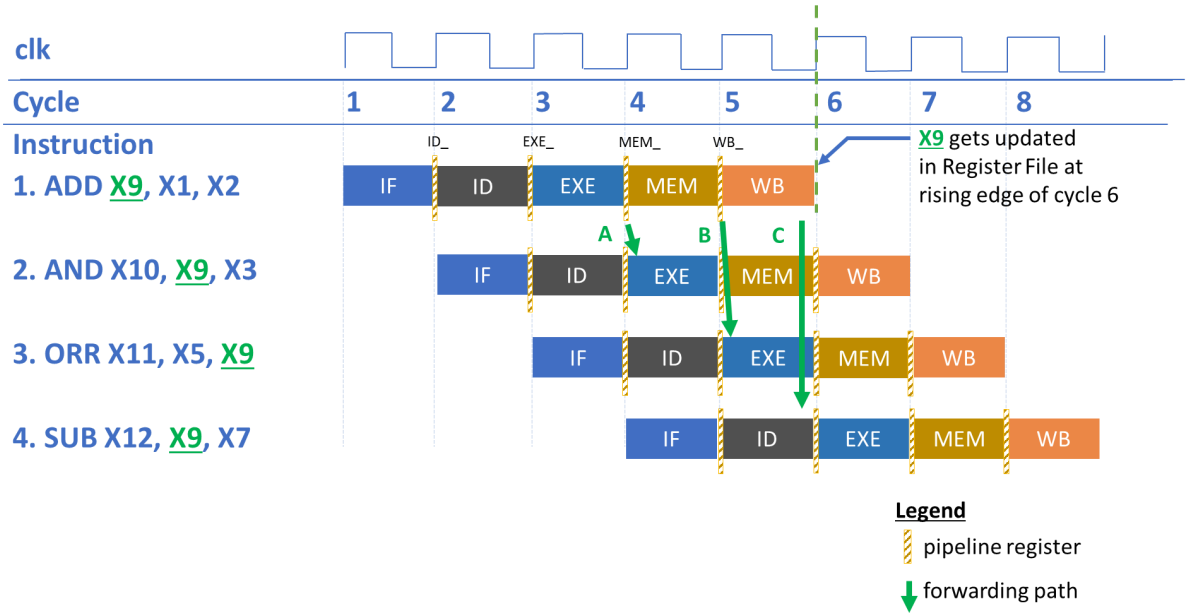

Figure 2: Forwarding paths in the pipeline

A. At the rising edge of the start of clock cycle 4, the EXE results for the ADD instruction is registered in the MEM pipeline register. The result in the MEM pipeline register is forwarded to the EXE stage of the AND instruction (see green arrow **A**). Forwarding path **A** (MEM_ -> EXE) bypasses the AND instruction having to wait for the ADD instruction to finish the WB stage in order to obtain the updated value of X9.

B. At the rising edge of the start of clock cycle 5, the result for the ADD instruction is now in the WB pipeline register. The result in the MEM pipeline now belongs to the result of the AND instruction instead. The ORR instruction is dependent on the ADD instruction. Therefore, the result in the WB pipeline register is forwarded to the EXE stage of the ORR instruction (see green arrow **B**). Forwarding path **B** (WB_ -> EXE) bypasses the ORR instruction having to wait for the ADD instruction to finish the WB stage.

C. During clock cycle 5, the result for the ADD instruction is to be written back to register X9—let's call the result to be written *write_reg_value*. The SUB instruction is still in the ID stage reading the old value of X9 because X9 will only be updated at the rising edge of cycle 6. (Do recall that in the case of Arm Education Core, the Register File has asynchronous reads and synchronous writes. Registers are read from ports N, M, and A in the ID stage). Therefore, *write_reg_value* needs to be forwarded to the register read value in the ID stage (see forwarding path **C**, write_reg_value -> read_reg_value).

Now, consider the following load-store scenario:

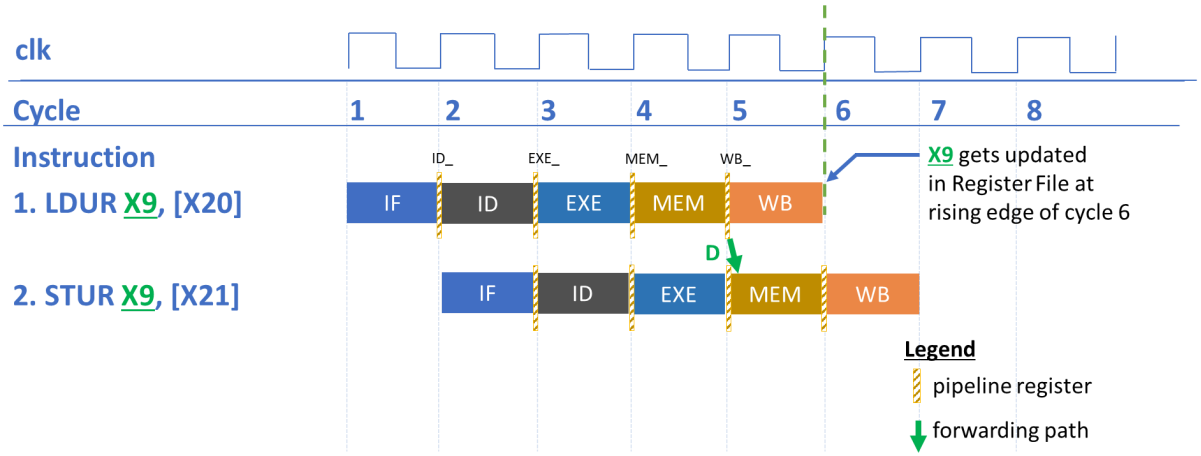

Figure 3: Forwarding path for load-store dependency scenario

D. At the rising edge of the start of clock cycle 5, the value loaded in the MEM (LDUR instruction) is now in the WB pipeline register. The STUR instruction is dependent on the LDUR instruction and stores data in the memory in the MEM stage. Therefore, the result in the WB pipeline register is forwarded to the MEM stage of the STUR instruction (see green arrow **D**). Forwarding path **D** (WB_ -> MEM) bypasses the STUR instruction having to wait for the LDUR instruction to finish the WB stage.

5 Implementing forwarding paths in Arm Education Core

In the previous labs, we have learned that Arm Education Core's Register File has three read ports N, M, and A, from which there are individual data paths for each port:

- Port N is usually for the register of the first operand. This feeds into the datapath leading to `exec_n`.
- Port M is the register of the second operand. This feeds into the datapath leading to `exec_m`.
- Port A is usually for the register that contains the value to be stored in a store instruction. This feeds into the datapath leading to `exec_a`.

Therefore, we need to:

1. Implement forwarding path combinatorial logic to pass the appropriate MEM/WB pipeline register to the exec_n, exec_m, exec_a values—see Figure 4.
2. Ensure that if a read address and write address for the Register File are equal, then the read value takes the value to be written (path C in Figure 2).

Notice that in Figure 4, the **EX_exec_n** pipeline register is now replaced with **EX_fexec_n**. **EX_fexec_n** feeds into the forwarding path logic, which will then select whether **EX_exec_n** should be from the EX pipeline register (**EX_fexec_n**) or from the MEM and WB pipeline registers, likewise for **EX_exec_a** and **EX_exec_m**.

To implement forwarding paths in Arm Education Core, we will use `Lab5_forwarding_pipeline.zip` provided with Lab 5. In this zip folder, the following files differ from the simple pipeline Arm Education Core RTL code from Lab 4:

- Educore.v
- InstructionDecoder.v
- RegisterFile.v

These differences will be described in the next task.

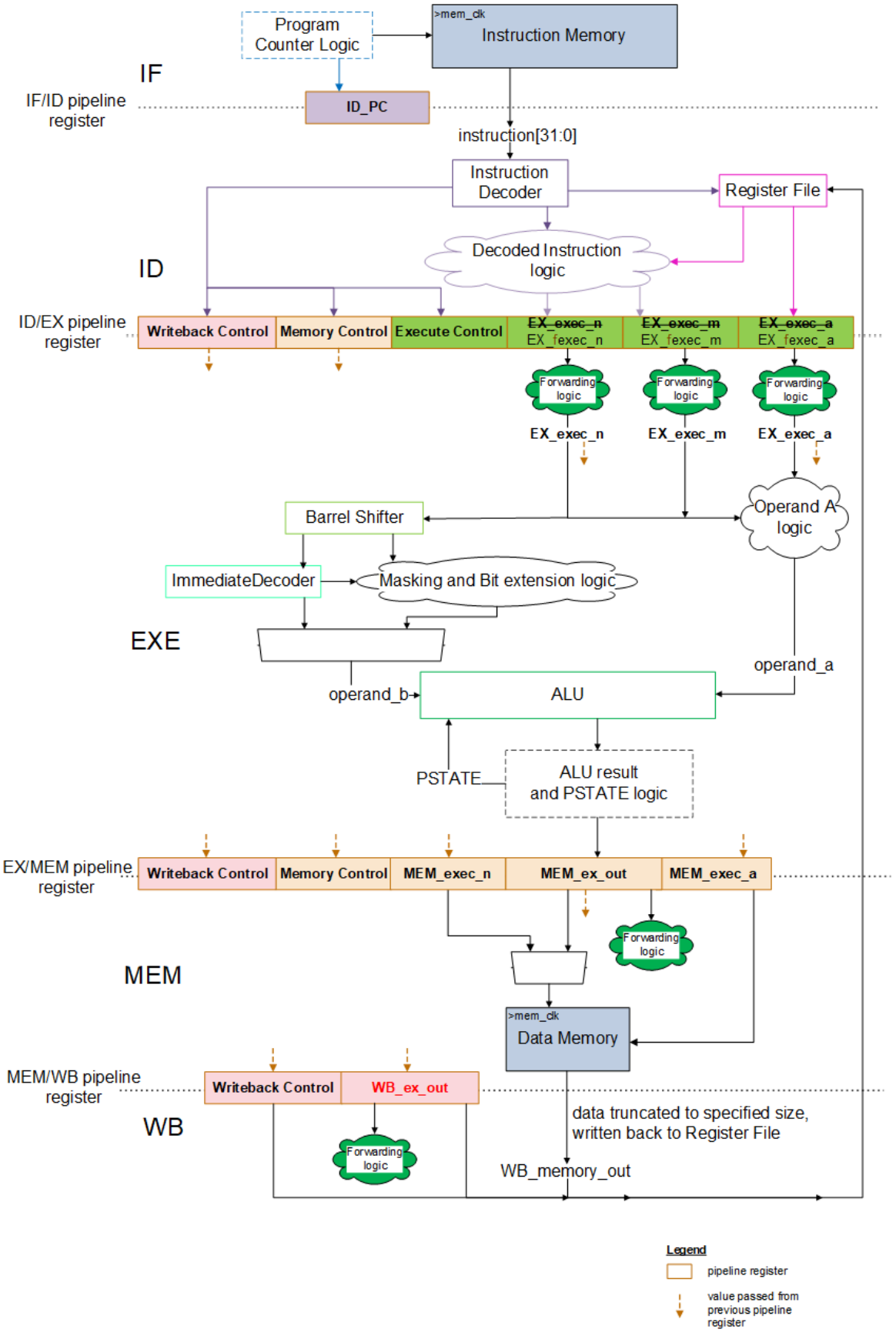

Figure 4: Pipelined Arm Education Core with forwarding path

5.1 Task: Code modifications

To do the code modifications for forwarding path implementation in Arm Education Core, follow these instructions:

1. Create a working directory folder called `Lab_5`.

2. Unzip the provided `Lab5_forwarding_pipeline.zip` in your working directory `Lab_5` folder.

3. Open the `src\RegisterFile.v` file and take notice of the following changes compared to the previous lab:

```
assign read_reg_vn =
    (|an_read_s & (an_read_s == write_s))? write_reg_v :
    (|an_read_s & (an_read_s == wload_s))? wload_reg_v : (
      (rX00 & {64{an_read_s[ 0]}}) | (rX01 & {64{an_read_s[ 1]}}))
```

```
assign read_reg_vm =
    (|am_read_s & (am_read_s == write_s))? write_reg_v :
    (|am_read_s & (am_read_s == wload_s))? wload_reg_v : (
      (rX00 & {64{am_read_s[ 0]}}) | (rX01 & {64{am_read_s[ 1]}}))
```

```
assign read_reg_va =
    (|aa_read_s & (aa_read_s == write_s))? write_reg_v :
    (|aa_read_s & (aa_read_s == wload_s))? wload_reg_v : (
      (rX00 & {64{aa_read_s[ 0]}}) | (rX01 & {64{aa_read_s[ 1]}}))
```

These changes are to implement forwarding path **C** in Figure 2, whereby if a read address and write address for the Register File are similar, then the read value takes the value to be written. This is applicable for ports N, M, and A. Do recall that there are two write ports, one for write register and the other for writeback due to load instructions.

4. Open the `src\InstructionDecoder.v` file and take notice that the Instruction Decoder now has the following additional outputs:

```
    // To Execute stage
    output reg        read_n_valid,
    output reg        read_m_valid,
    output reg        read_a_valid,
    output reg  [5:0] shamt,
```

These additional ports are merely active-HIGH signals, which indicate that the instruction has valid register operands. For example, for an instruction ADD X1, X2, X3, register X1 is the destination register, so we don't need to forward register X1 value to the later pipeline stages (as part of the forwarding paths), and thus the `read_a_valid` is LOW.

5. Open the `src\Educore.v` file and take notice of the following changes:
    a. Since the Instruction Decoder module has additional outputs, the following wires are defined:

```
// To Execute stage
wire        ID_read_n_valid;
wire        ID_read_m_valid;
wire        ID_read_a_valid;
wire  [5:0] ID_shamt;
```

and are connected to the Instruction Decoder Module:

```
    // To Execute stage
    .read_n_valid(ID_read_n_valid),
    .read_m_valid(ID_read_m_valid),
    .read_a_valid(ID_read_a_valid),
```

    b. We have also declared registers **EX_fexec_<n/m/a>** and **EX_exec_<n/m/a>** so that it is similar to what is shown in Figure 4.

6. Modify Educore.v so that register indexes and register valid signals from the Instruction Decoder are pipelined into the EX registers in the next clock cycle by adding the following code:
    a. Declare the following EX registers in the //Register declaration section of Educore.v, as shown in the green section below:

```
/*****************************************************************************/
// Register declaration

/* Instruction currently being decoded */
reg [31:0] instruction;
/* The program counter (pointing to the next instruction to be fetched) */
reg [61:0] PC;
/* Processor state register (Holds NZCV flags`) */
reg [3:0] PSTATE;
/* The program counter / address of currently decoded instruction (previous value of PC) */
reg [61:0] ID_PC;
/* Operand of the N datapath. This could be the ID_PC or the register value from port N */
reg [63:0] EX_fexec_n;
/* Operand of the M datapath. This could be an immediate value or a register value from port M */
reg [63:0] EX_fexec_m;
/* Operand of the A datapath. This is always the register value from port A */
reg [63:0] EX_fexec_a;

/* Signal indicating that operands from the respective ports could be subject to forwarding. */
/* (i.e Valid modifiable register value) */
reg        EX_read_n_valid;
reg        EX_read_m_valid;
reg        EX_read_a_valid;

/* Register indices from the respective ports to enable and check forwarding */
reg  [4:0] EX_read_reg_an;
reg  [4:0] EX_read_reg_am;
reg  [4:0] EX_read_reg_aa;
```

    b. Now connect these newly declared pipeline registers to the corresponding value from the ID stage at the positive edge of the core clock, as shown in the green section below:

```
    always @ (posedge clk) if(npe_stop) begin
        case (ID_fexec_n_mux)
            `FEXEC_N_PC:      EX_fexec_n <= aligned_id_pc;
            `FEXEC_N_PC_PAGE: EX_fexec_n <= {aligned_id_pc[63:12],12'h000};
            `FEXEC_N_RN:      EX_fexec_n <= ID_rn_value;
        endcase

        case (ID_fexec_m_mux)
            `FEXEC_M_IMM:     EX_fexec_m <= ID_immediate;
            `FEXEC_M_RM:      EX_fexec_m <= ID_rm_value;
        endcase

        EX_fexec_a          <= ID_ra_value;
        EX_read_n_valid     <= ID_read_n_valid;
        EX_read_m_valid     <= ID_read_m_valid;
        EX_read_a_valid     <= ID_read_a_valid;
        EX_read_reg_an      <= ID_read_reg_an;
        EX_read_reg_am      <= ID_read_reg_am;
        EX_read_reg_aa      <= ID_read_reg_aa;
        EX_shamt            <= ID_shamt;
```

7. Modify the Execute stage in Educore.v so that it is as shown below:

```
/*****************************************************************************/
// Execute stage

    // Forward detection
    assign EX_exec_n = EX_read_n_valid? (
        (MEM_write_en & (MEM_rd_addr == EX_read_reg_an))? MEM_ex_out :
        (WB_write_en  & (WB_rd_addr  == EX_read_reg_an))? WB_ex_out  :
        (WB_wload_en  & (WB_rt_addr  == EX_read_reg_an))? WB_memory_out : EX_fexec_n
    ) : EX_fexec_n;
    assign EX_exec_m = EX_read_m_valid? (
        (MEM_write_en & (MEM_rd_addr == EX_read_reg_am))? MEM_ex_out :
        (WB_write_en  & (WB_rd_addr  == EX_read_reg_am))? WB_ex_out  :
        (WB_wload_en  & (WB_rt_addr  == EX_read_reg_am))? WB_memory_out : EX_fexec_m
    ) : EX_fexec_m;
    assign EX_exec_a = EX_read_a_valid? (
        (MEM_write_en & (MEM_rd_addr == EX_read_reg_aa))? MEM_ex_out :
        (WB_write_en  & (WB_rd_addr  == EX_read_reg_aa))? WB_ex_out  :
        (WB_wload_en  & (WB_rt_addr  == EX_read_reg_aa))? WB_memory_out : EX_fexec_a
    ) : EX_fexec_a;

    // ------------------------- Barrel shifter ------------------------- //
```

These changes are to implement forwarding paths **A** and **B** in Figure 2. This modification implements multiplexes so that **EX_exec_<n/m/a>** can get forwarded the appropriate values from either the MEM stage (`MEM_ex_out`, i.e., result of EXE) or the WB stage (`WB_ex_out`/`WB_memory_out`, i.e., results of MEM stage, depending on whether it is a data processing instruction or load/store instruction). The conditions for these multiplexes include:

- If the register indexes are valid for each N, M, A port (`EX_read_<n/m/a>_valid`)
- If there is data dependency, i.e., the destination register address (rd_addr) or target register address (rt_addr for load) in the MEM and WB stages match the current register index at EXE stage in any of the N, M, A ports (e.g., `MEM_rd_addr == EX_read_reg_aa`)
- If the instruction will update/write to the Register File (`write_en` or `wload` signals).

8. To account for load-store dependency scenarios, modify the Memory stage in Educore.v so that it is as shown below:

```
/*****************************************************************************/
// Memory stage

// Check forwarding of loads to stores
assign data_memory_out_v =
    ((WB_rt_addr == MEM_rt_addr) & WB_wload_en)? WB_memory_out : MEM_rt_value;

assign data_memory_a = (MEM_addr_mux == `MEM_ADDR_RN)? MEM_rn_value : MEM_ex_out;
```

These changes are to implement forwarding path **D** in Figure 2.
`data_memory_out_v` is the value to be written to the data memory, typically from a store instruction. If the instruction before is a load instruction (`WB_load_en`) and there is register data dependence, then the stored value is from the previous instruction.

9. Save the modified files.

10.

6 Exercise: Verify and analyze forwarding paths

In this exercise, we will take a closer look at the behavior of the forwarding paths implemented in the pipelined Arm Education Core. We will achieve this by simulating Arm Education Core with `test_Lab5_forwardingpaths.S`, which has the following code:

```
.global _start

.text

_start:

//place move instructions here

      MOVZ   X0, #0x0100

      MOVZ   X1, #0x0101

      MOVZ   X5, #0x1

      MOVZ   X6, #0x2

      MOVZ   X7, #0x3

      MOVZ   X8, #0x4

      MOVZ   X9, #0x5


// store values in memory

      STURB  W5, [X0]

      STURB  W6, [X0, #1]

      STURB  W7, [X0, #2]

      STURB  W8, [X0, #3]


// test load-store dependency

      LDURB  W10, [X0]

      STURB  W10, [X0, #4]


// test data dependency

      ADD    X10, X9, #0

      AND    X11, X10, X7

      ORR    X12, X8, X10

      SUB    X13, X10, X5
```

```
// test load-use dependency
      LDURB  W10, [X1]
      NOP
      AND    X11, X10, X7
      ORR    X12, X8, X10
      SUB    X13, X10, X5

// test Branch
      B            _test
      ADD          X11, X10, #100      // this shouldn't execute
      ADD          X11, X10, #200      // this shouldn't execute
      ADD          X11, X10, #300      // this shouldn't execute
_test:
      ADD          X11, X10, #5
      NOP
      NOP
      NOP
      NOP
      YIELD
```

Run the provided testcase in `Lab5_forwarding_pipeline.zip` using the bash script, by following these steps:

1. Open a Windows terminal and change directory to the folder.

   ```
   cd Lab5_forwarding_pipeline
   ```

2. Run the run_tests.sh file by entering the following command:

   ```
   bash run_tests.sh
   ```

3. Use the following command to inspect the generated waveforms:

   ```
   cd sim_dump/
   gtkwave test_Lab5_forwardingpaths.mem.lx2 ../GTKW/debug.gtkw
   ```

4. Inspect the waveforms and check if they match the expected results of the Assembly code.

Observations:

The following diagram shows a snapshot of the simulated waveforms:

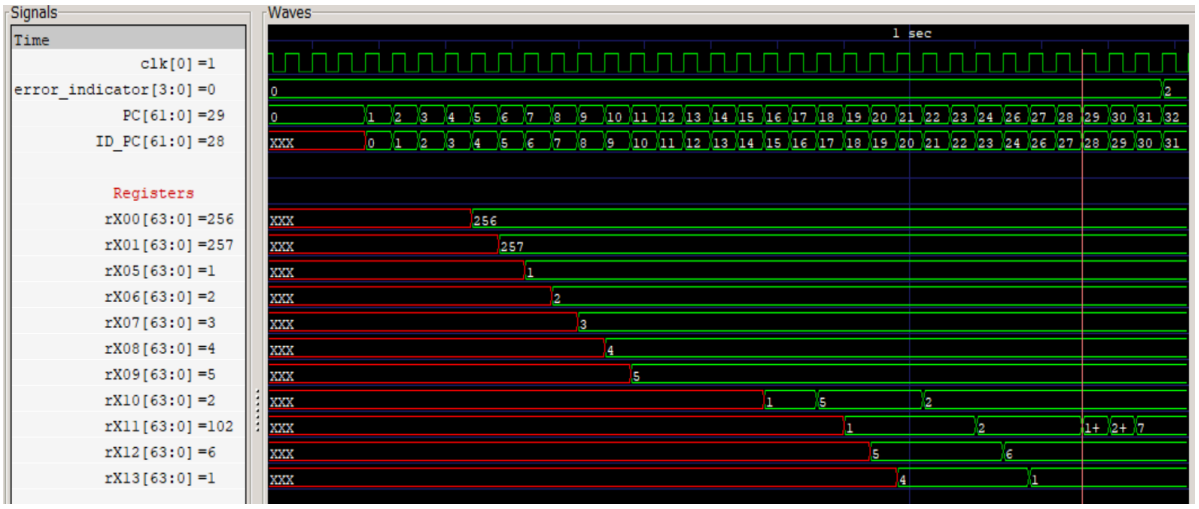

Figure 5: Waveforms from test_Lab5_forwardingpath1.mem.lx2

The load-store dependency problem is also resolved, as shown in Figure 6.

Figure 6: Load-store dependency resolved with the forwarding path

Answer the following questions:

1. What is the purpose of the following NOP instruction in the code:

```
// test load-use dependency
      LDURB  W10, [X1]
      NOP
      AND    X11, X10, X7
      ORR    X12, X8, X10
      SUB    X13, X10, X5
```

   Modify the testcase and remove the NOP between the LDURB instruction and AND instruction. Resimulate and observe the waveforms. (**Hint**: Look specifically at the contents of register X11). What happens? Explain your answer.

2. For the branch instruction, why are the 2 ADD instructions after `B _test` being executed when it should have branched immediately to `ADD X11, X10, #5`? Why isn't the `ADD X11, X10, #300` being executed?

3. Name 1 advantage and 2 limitations that are not solved by the forwarding paths.

7 Summary

In this lab, we have implemented forwarding paths in Arm Education Core and demonstrated how it can increase the throughput in a given time if there are data hazards. However, there are still some limitations, and we will address these in the next lab.