arm Education

Computer Architecture Course

LAB 4

A Simple Pipeline

Issue 1.0

Contents

1 Introduction

1.1 Lab overview

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

- Identify the pipeline stages and placement of pipeline registers.
- Identify relevant control signals that need to be stored in pipeline registers.
- Modify a single-cycle processor (Arm Education Core) to implement a simple pipeline.
- Verify the functionality of the simple pipeline implemented in the Arm Education Core.
- Demonstrate the limitations of the simple pipeline implemented.

2 Requirements

Before attempting Lab 4, 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
- Introduction to Arm Education Core documentation
- Arm Education Core microarchitecture—completed Lab 3A and Lab 3B

Lab 4 is provided with **Lab4_simple_pipeline.zip**, which contains supplementary files used in the exercises of this lab.

3 Theory: Why pipeline a processor?

3.1 What is pipelining?

In our previous labs, we have used the single-cycle Arm Education Core that processes each instruction in one clock cycle, i.e., IF, ID, EXE, MEM, and WB processing stages are done in one **clk_en** cycle. Single-cycle Arm Education Core has a *Cycle Per Instruction* (CPI) of 1.

A *critical path* is the part of the circuit or logic that takes up the most time, and thus the clock frequency cannot be made shorter and in turn the computation faster unless this critical path is resolved. In reality, each logic and component single-cycle Arm Education Core would take some time to run—not to mention that sometimes Arm Education Core also has to access the memory twice (instruction and memory) within a single clock cycle. As a result, the single clock cycle needs to be in a long enough duration that accommodates at least two memory accesses and any critical paths.

How do we increase the clock frequency and throughput of Arm Education Core without significantly increasing the CPI as well? One such method to increase throughput is by using pipelining, where we arrange the different processing stages to be overlapped to exploit the "temporal" parallelism, as shown in Figure 1 below.

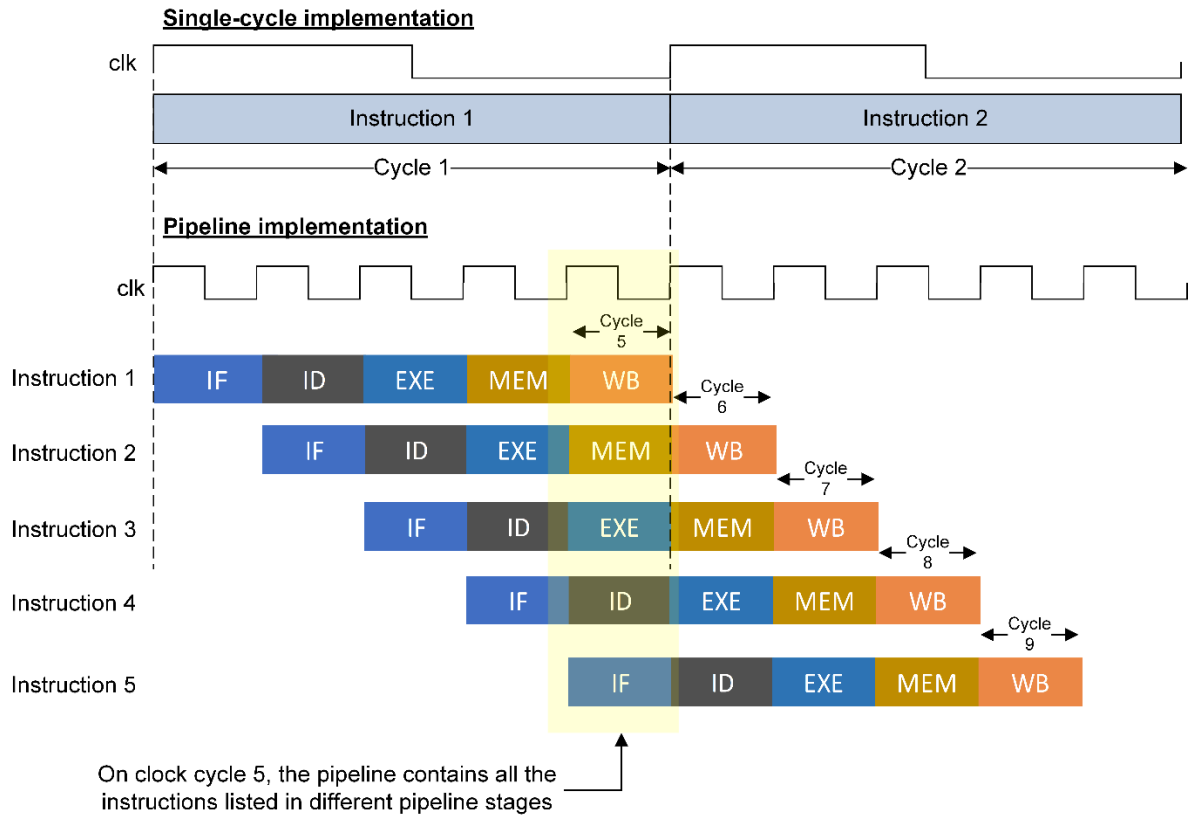


Figure 1: Single-cycle vs pipeline

Table 1 shows how the instructions propagate through each pipeline stage:

	IF	ID	EXE	MEM	WB
Cycle 1	Instruction 1	-	-	-	-
Cycle 2	Instruction 2	Instruction 1	-	-	-
Cycle 3	Instruction 3	Instruction 2	Instruction 1	-	-
Cycle 4	Instruction 4	Instruction 3	Instruction 2	Instruction 1	-
Cycle 5	Instruction 5	Instruction 4	Instruction 3	Instruction 2	Instruction 1
Cycle 6	-	Instruction 5	Instruction 4	Instruction 3	Instruction 2
Cycle 7	-	-	Instruction 5	Instruction 4	Instruction 3
Cycle 8	-	-	-	Instruction 5	Instruction 4
Cycle 9	-	-	-	-	Instruction 5

Table 1: Processing 5 instructions in pipeline stages

As shown in Figure 1, the initial pipeline latency in terms of clock cycle may increase (as in it takes 5 cycles before Instruction 1 finish executing, but the overall throughput (number of instructions executed at a given time) is higher compared to if all processing stages are executed in 1 clock cycle. In Figure 1, the single-cycle processor requires 2 long clock cycles to process 2 instructions. In contrast, the pipelined processor can process 5 instructions in 9 short clock cycles, which is almost the same time as 2 long clock cycles for the single-cycle processor. Since these short clock cycles are at a higher frequency, the pipelined version takes up less time and yet higher throughput compared to the single-cycle implementation.

3.2 Pipeline registers

In the pipelining method, the processing of instructions is broken into stages and the different stages of different instructions are overlapped (refer back to Figure 1).

To implement pipelining in a processor, we can insert *pipelining registers* to divide the processor logic into different pipeline stages. The pipeline registers are synchronous to a clock, and the pipeline registers allow synchronization and signaling of the logic between two stages.

Consider the example shown in the diagram below, where clk is an external clock source, C is the time delay through a register, and T is the delay due to the *critical path* of the combinational logic. The *critical path* is the path in the combinational logic that imposes the greatest delay in signal propagation from its inputs to its outputs. The overall minimum clock period that the circuit requires to be able to complete processing is equivalent to clock period = T + C.

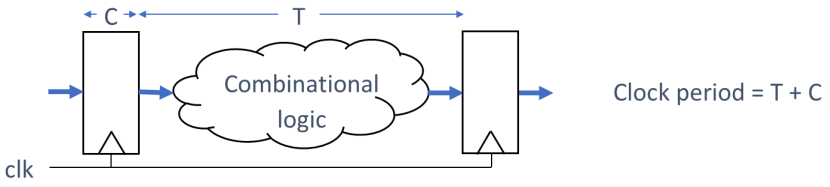


Figure 2: Clock period = T+C

From Figure 2, it is apparent that the processing clock period is dependent on the critical path. Now let's observe what happens if we were to divide the combinational logic into two, so that T of the critical path is now T/2, as shown in the diagram below.

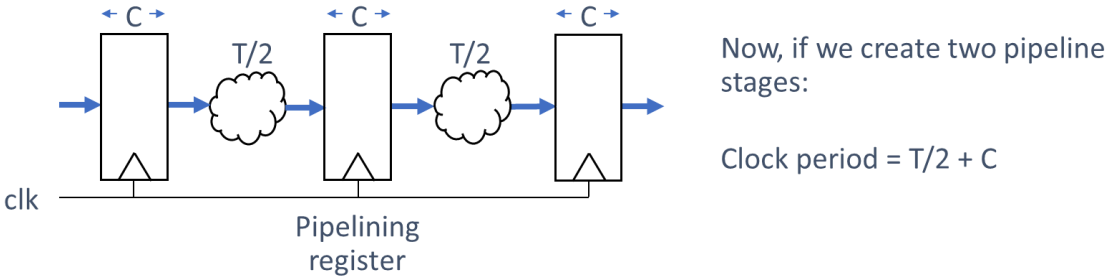


Figure 3: Clock period = T/2 + C

As shown in Figure 3, the minimum clock period is now = T/2 + C. If C is small, then we have approximately halved the clock period by implementing the pipeline, therefore causing the throughput to nearly double.

When inserting the pipeline registers, there need to be appropriate control signals so that the correct information is passed down to each stage of the pipeline. For example, an instruction's destination register needs to be carried down to the next pipeline stages until it reaches the WriteBack pipeline stage. We will look at this in detail in the next exercise.

4 Implementing pipeline in Arm Education Core

To implement pipelining in Arm Education Core, we need to:

1. **Identify the pipeline stages in Arm Education Core.**
   This is easy. Single-cycle Arm Education Core conveniently has 5 processing stages done in a single cycle:
   - IF—Instruction Fetch
   - ID—Instruction Decode
   - EXE—Execute
   - MEM—Memory access
   - WB—WriteBack

   To implement a pipeline in Arm Education Core, we could use these 5 processing stages as 5 pipeline stages instead.

2. **Identify where to place the pipeline registers.**
   This is also relatively straightforward—the pipeline registers should be placed in between each pipeline stage. The pipeline registers should be synchronized to a reference clock.

3. **Decide what kind of signals and control signals need to be in these pipeline registers.**
   This requires some understanding of Arm Education Core's microarchitecture. Do recall that there need to be appropriate control signals so that the correct information is passed down to each stage of the pipeline. For example, an instruction's destination register needs to be carried down to the next pipeline stages until it reaches the WriteBack pipeline stage.

4.1 Recap: Single-cycle Arm Education Core microarchitecture

In Lab 3A and 3B, we learned about the single-cycle Arm Education Core microarchitecture and data path, as shown in Figure 4.

We have learned that:

- The Instruction and Data Memory are in the testbench file and run on its own **mem_clk**.
- The Program Counter points to the address of the instruction to be fetched. The Program Counter value is passed on to the ID (Decoded Instruction Logic) and EXE stages.
- The instruction fetched is passed on to the Instruction Decoder, which outputs various control signals and variables. Many control and variable signals from the Instruction Decoder module go to the IF stage (i.e., Program Control Logic), ID stage (Register File and Decoded Instruction Logic), EXE, MEM, and WB stages.
- The Register File has 3 asynchronous read ports (N, M, A) and 2 synchronous write ports (one for write register and the other for load instructions).
- **exec_a**, **exec_m**, and **exec_n** correspond to values of operands in the instructions; these values are obtained once the instruction is decoded (ID) and are passed on to the EXE and MEM stages for further execution.

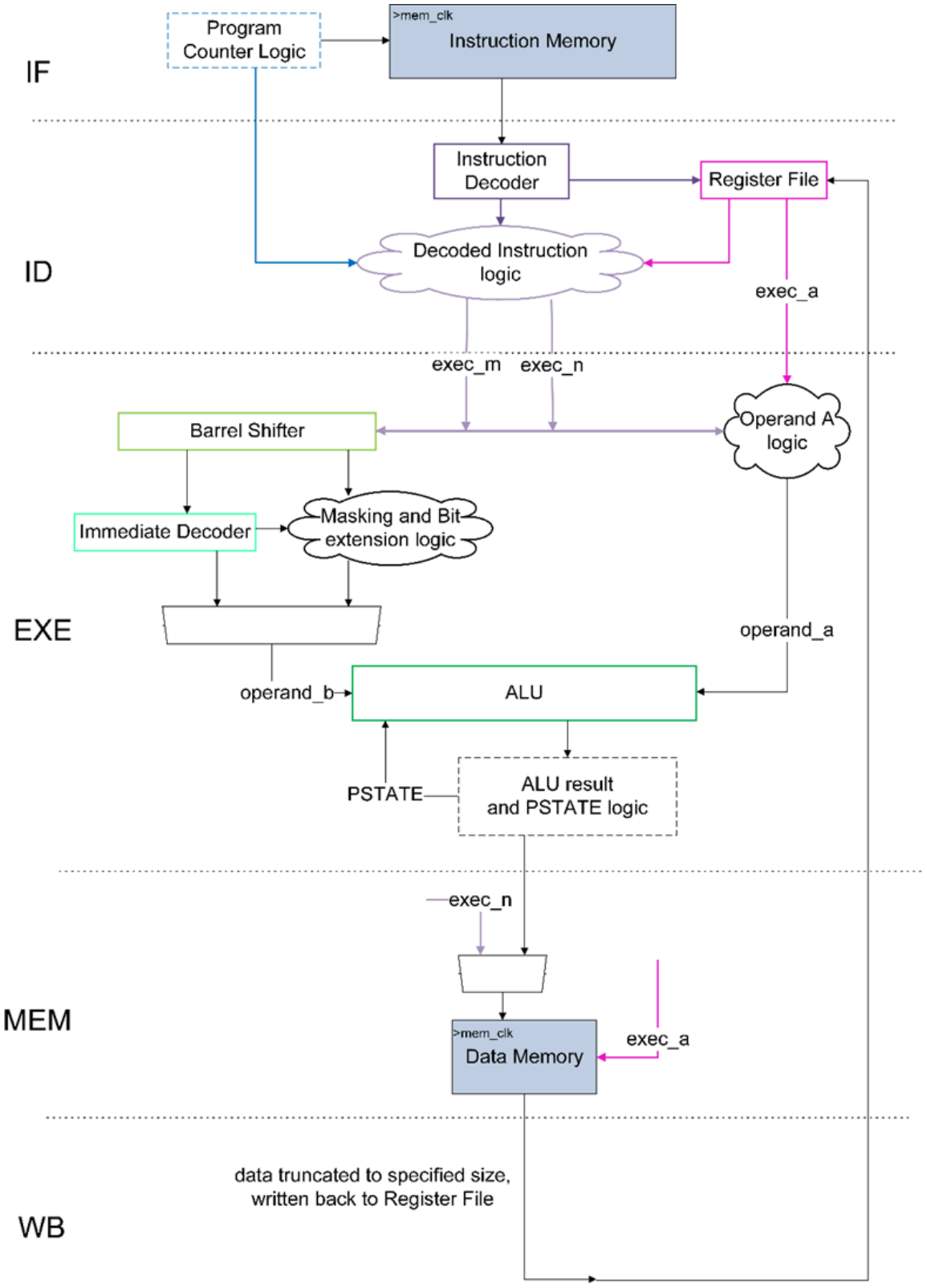
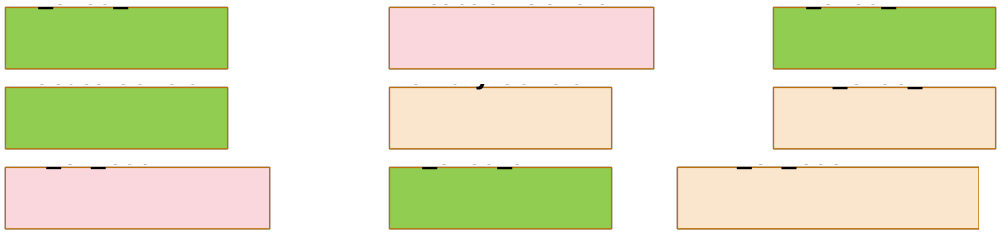


Figure 4: Overview of single-cycle Arm Education Core datapath (not all control signals are shown)

4.2 Exercise: Identifying signals for pipeline registers

To aid code readability, some variable names have a prefix that reflects the stage they are in. This is useful particularly if there are control signals that need to be passed on several stages to get to the WB stage, as an example.

Figure 5 on the next page shows a diagram of a pipelined Arm Education Core with its pipeline registers. The pipeline registers typically have a prefix that reflects the stage that they are currently in (usually the beginning of a stage). Having completed the previous labs and now familiar with Arm Education Core's microarchitecture, match the following blocks to the circled numbers in the pipeline registers in Figure 5:



Note:

- The **Execute Control** signals are all control signals used for the EXE stage, for example, control signals to multiplexes, combi logic, etc.
- To aid code readability, the signal that is being passed on to the next stage via a pipeline register will have a signal name prefix of the next pipeline stage. For example, **ID_PC** is the program counter value that is passed on to a pipeline register, which will then pass on the Decoded Instruction Logic in the **ID** stage.

Provide your answers here:

❶

❷

❸

❹

❺

❻

❼

❽

❾

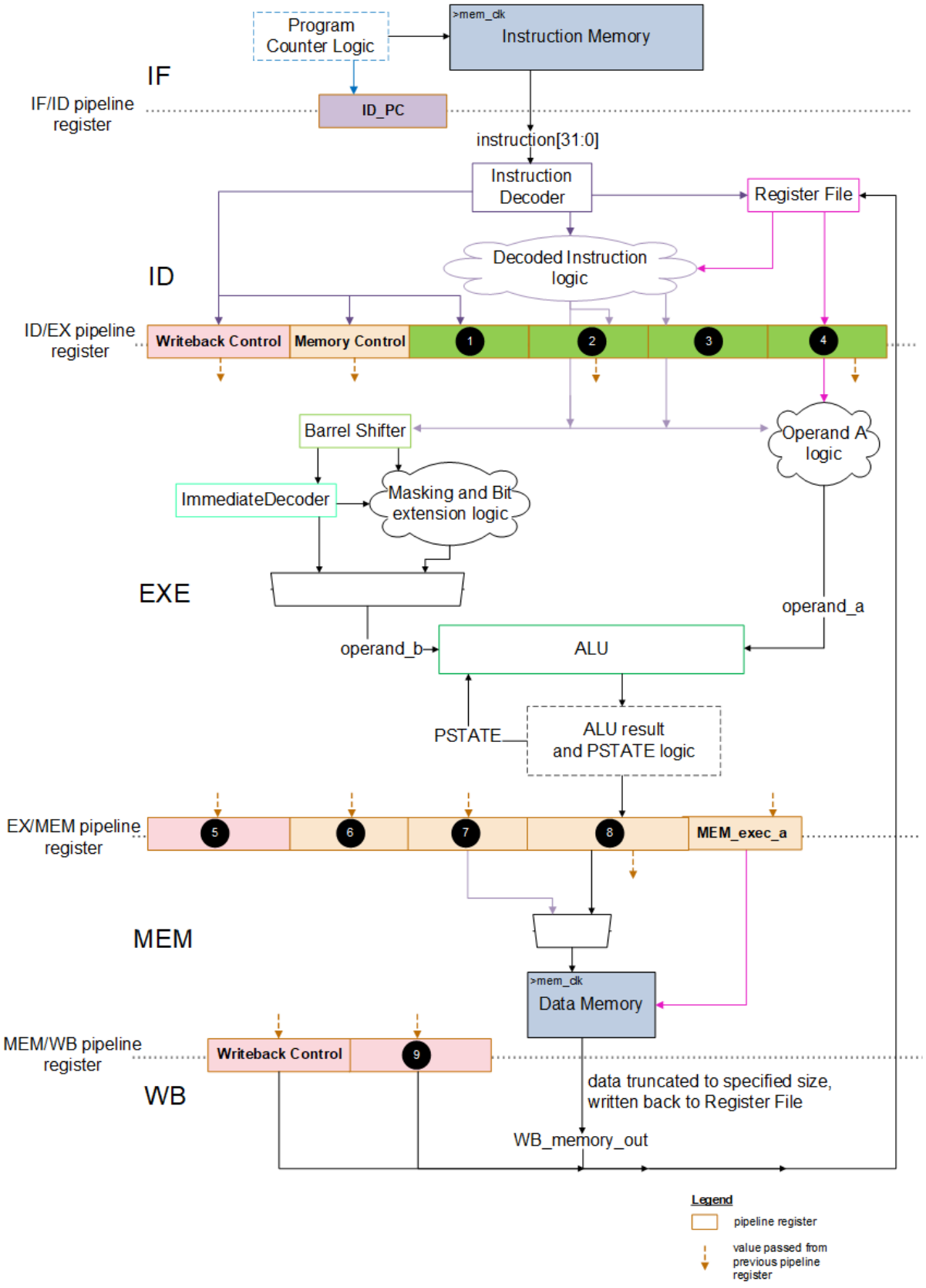


Figure 5: A simple pipeline in Arm Education Core

4.3 Code modifications for a simple pipeline

To implement pipeline registers and pass the relevant signals to the pipeline registers, we will use a modified **Educore.v** and **test_Educore.v** provided in the **Lab4_simple_pipeline** folder.

The testbench file provided with Lab 4 differs from the single-cycle Arm Education Core version in terms of the file **mem_clk** and **core_clk** ratios. Due to the pipelined design (see Figure 1 in What is pipelining?), **mem_clk** and **core_clk** are now able to run at the same clock and phase. It was not possible to do this in the single-cycle Arm Education Core.

```
always #2 mem_clk = ~mem_clk;
always #2 core_clk = ~core_clk;
```

Note: The pipelined version of Arm Education Core is optimized for **core_clk** = **mem_clk**. This means that both clocks must also be in the same phase. If the clocks are not in phase, then Arm Education Core will not work as intended.

Since **core_clk** = **mem_clk**, **instruction[31:0]** and **WB_memory_out** are specifically not pipeline registers because memory access needs at least 1 clock cycle. Therefore, as an example, for a load instruction:

At positive edge of clock cycle 1	**IF stage** **Educore** outputs **instruction_memory_**a to the memory
At positive edge of clock cycle 2	**ID stage** **memory** updates **instruction_memory_v** **instruction** = **instruction_memory_v**
At positive edge of clock cycle 3	**EXE stage**
At positive edge of clock cycle 4	**MEM stage** **Educore** outputs data**_memory_**a to the memory
At positive edge of clock cycle 5	**WB stage** **memory** updates **data_memory_v** **WB_memory_out** = truncated **data_memory_v**

The **Educore.v** file provided with Lab 4 differs from the single-cycle Arm Education Core RTL code in the following way:

1. Some variable names have a prefix to reflect the pipeline stage that they are in. Their corresponding connections have been defined and replaced appropriately.

   For example, in the single-cycle Arm Education Core, **read_reg_am** is connected to the Instruction Decoder. In the in **Educore.v** file for Lab4, **read_reg_am** has been renamed as **ID_read_reg_am** and has been made to connect to **EX_read_reg_am** pipeline register so that the signal can pass to the EXE stage. The following diagram also shows a small subset of such variable names in **Educore.v**:

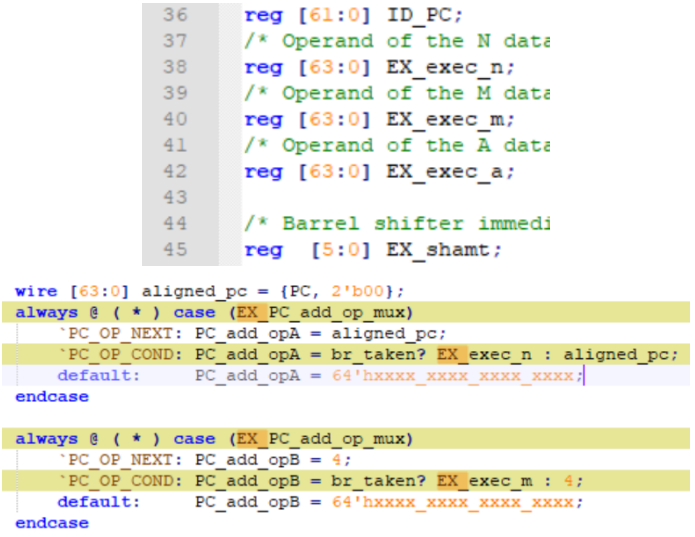


Figure 6: New variable names (left) and automatic replacement (right)

2. Any control signals that are to be passed on to several pipeline stages are also defined in **Educore.v**, such as the **Execute Control**, **WriteBack Control** and **Memory Control** signals in Figure 5. These control signals include the:
   - Load and destination addresses (**rt_addr**, **rd_addr**).
   - Read and write enable signals (**read, write_en, wload_en**).
   - Any other signals needed for data truncation and selection at the WriteBack stage (**size**, **sign_ext, load_fnH**).

   The following diagram shows a subset of such signals—for example, **write_en** will be passed to the EXE stage (**EX_write_en**) from the Instruction Decoder and then to the MEM stage (**MEM_write_en**), followed by the WB stage (**WB_write_en**), which then wraps back to the Register File, as shown in Figure 5.

```
/* Register index of register to be written with memory loaded value */
reg  [4:0] EX_rt_addr;
reg  [4:0] MEM_rt_addr;
reg  [4:0] WB_rt_addr;
/* Register index of register to be written with execute stage output */
reg  [4:0] EX_rd_addr;
reg  [4:0] MEM_rd_addr;
reg  [4:0] WB_rd_addr;

/* Write enable for memory loaded value */
reg        EX_wload_en;
reg        MEM_wload_en;
reg        WB_wload_en;

/* Write enable for execute stage output value */
reg        EX_write_en;
reg        MEM_write_en;
reg        WB_write_en;
```

Figure 7: Example control signals to be passed on to several pipeline stages

There are some signals that only propagate to the MEM stage rather than all the way to the WB stage, as shown in the following code snippet.

```
// To memory stage
/* Memory write signal */
reg        EX_mem_write;
reg        MEM_write;
/* Select source of memory address. (Register (N-port) or Execute out (i.e ALU)) */
reg        EX_mem_addr_mux;
reg        MEM_addr_mux;
/* Base register for memory access (N port register value) */
reg [63:0] MEM_rn_value;
/* Register value to be stored in memory (A port register value) */
reg [63:0] MEM_rt_value;
```

Figure 8: Control signals that propage to MEM stage

3. All pipeline registers are synchronized to the positive edge of **clk**. (Wherever appropriate, the pipeline registers are in sync to the negative edge of **nreset** or enabled by **npc_stop**.)

4. In the system-wide assignment, there are new signals, called **instruction_valid** and **next_instruction_valid**. **next_instruction_valid** is only high when the processor is not in reset. Then, at the next clock cycle, **next_instruction_valid** value is passed on to the current instruction valid signal (**instruction_valid**). An instruction is only registered in register **instruction** if the **instruction_valid** is high. The role of these signals will be more apparent when we deal with potential hazards in the next lab.

4.3.1 Exercise

Follow these steps:

1. Unzip **Lab4_simple_pipeline.zip** in your working directory **Lab_4** folder.

2. Open the **src\Educore.v** file. This file is incomplete—it is missing some code that you will need to complete in this exercise.

3. Search the **Educore.v** file for the string "…" (using Ctrl+F, for example). If you search from the top of the file, the first search result will give you:

```
always @ (posedge clk) if(npe_stop) begin
    ID_PC <= ...;
end
```

4. Replace "…" with signal **PC**. This means that the value PC will be passed to pipeline register ID_PC at every positive edge of **clk** (when **npe_stop** is high due to **clk_en** being high. **npe_stop** will be low when there is a non-zero **error_indicator**).

```
always @ (posedge clk) if(npe_stop) begin
    ID_PC <= PC;
end
```

5. Continue searching for string "…" and complete the missing code:
    a. For ID/EX **Execute Control** and **EX_exec_a** pipeline register:

```
always @ (posedge clk or negedge nreset)
if(~nreset) begin
    EX_write_en      <= 0;
    EX_wload_en      <= 0;
    EX_mem_write     <= 0;
    EX_mem_read      <= 0;
    EX_pstate_en     <= 0;
    EX_nextPC_mux    <= `NEXT_PC_ADD;
    EX_PC_add_op_mux <= `PC_OP_NEXT;
end else if(npe_stop) begin
    EX_write_en      <= ID_write_en;
    EX_wload_en      <= ID_wload_en;
    EX_mem_write     <= ID_ ...;
    EX_mem_read      <= ID_ ...;
    EX_pstate_en     <= ID_pstate_en;
    EX_nextPC_mux    <= ID_nextPC_mux;
    EX_PC_add_op_mux <= ID_PC_add_op_mux;
end

always @ (posedge clk) if(npe_stop) begin
    EX_exec_a          <= ID_ra_value;
    EX_shamt           <= ID_shamt;
    EX_imm_sz          <= ID_imm_sz;
    EX_imm_n           <= ID_imm_n;
    EX_FnH             <= ID_ ...;
    EX_barrel_op       <= ID_barrel_op;
    EX_barrel_in_mux   <= ID_barrel_in_mux;
    EX_barrel_u_in_mux <= ID_barrel_u_in_mux;
    EX_bitext_sign_ext <= ID_bitext_sign_ext;
    EX_alu_op_a_mux    <= ID_alu_op_a_mux;
    EX_alu_op_b_mux    <= ID_ ...;
    EX_wtmask          <= ID_ ...;
    EX_invert_b        <= ID_invert_b;
    EX_alu_cmd         <= ID_ ...;
    EX_out_mux         <= ID_out_mux;
    EX_condition       <= ID_condition;
    EX_pstate_mux      <= ID_pstate_mux;
    EX_br_condition_mux <= ID_ ...;
```

b. For ID/EX **Memory Control** and **WriteBack Control** pipeline register:

```
// to memory stage
EX_mem_size     <= ID_ ...;
EX_mem_sign_ext <= ID_mem_sign_ext;
EX_mem_addr_mux <= ID_mem_addr_mux;

// to writeback stage
EX_rt_addr  <= ID_rt_addr;
EX_rd_addr  <= ID_ ...;
EX_load_FnH <= ID_load_FnH;
```

Note: **rd_addr** is the destination register address for data processing instructions. **rt_addr** is the target address for load instructions.

c. For ID/EX **EX_exec_n** and **EX_exec_m** pipeline registers:

```
always @ (posedge clk) if(npe_stop) begin
    case (ID_exec_n_mux)
        `FEXEC_N_PC:      EX_exec_n <= aligned_id_pc;
        `FEXEC_N_PC_PAGE: EX_ ... <= {aligned_id_pc[63:12],12'h000};
        `FEXEC_N_RN:      EX_exec_n <= ID_rn_ ...;

    endcase

    case (ID_exec_m_mux)
        `FEXEC_M_IMM:     EX_exec_m <= ID_immediate;
        `FEXEC_M_RM:      EX_ ... <= ID_rm_value;
```

d. For EXE/MEM **Memory Control**, **WriteBack Control**, **MEM_exec_n**, **MEM_exec_a**, and **MEM_exec_m** pipeline register:
Note: MEM_exec_n is named **MEM_rn_value**, and **MEM_exec_a** is named **MEM_rt_value** in the code.

```
always @ (posedge clk or negedge nreset)
if(~nreset) begin
    MEM_read     <= 1'b0;
    MEM_write    <= 1'b0;
    MEM_wload_en <= 1'b0;
    MEM_write_en <= 1'b0;
end else if(npe_stop) begin
    MEM_read     <= EX_mem_read;
    MEM_write    <= EX_mem_ ...;
    MEM_wload_en <= EX_wload_en;
    MEM_write_en <= EX_ ...;
end
```

```
always @ (posedge clk) if(npe_stop) begin
    MEM_rn_value <= EX_exec_n;
    MEM_rt_value <= EX_exec_a;
    MEM_size     <= EX_mem_ ...;
    MEM_sign_ext <= EX_mem_sign_ext;
    MEM_addr_mux <= EX_mem_addr_mux;

    case (EX_out_mux)
        `EX_OUT_ALU:    MEM_ex_out <= EX_FnH? alu_out    : {{32{1'b0}},alu_out[31:0]};
        `EX_OUT_CND:    MEM_ex_out <= EX_FnH? cnd_out    : {{32{1'b0}},cnd_out[31:0]};
        `EX_OUT_CTRL:   MEM_ ... <= ctrl_out;
        `EX_OUT_PC_4:   MEM_ ... <= aligned_id_pc;
        default:        MEM_ex_out <= 64'hxxxx_xxxx_xxxx_xxxx;
    endcase
```

```
MEM_rt_addr  <= EX_rt_ ...;
MEM_rd_addr  <= EX_rd_ ...;
MEM_load_FnH <= EX_load_FnH;
```

e. For MEM/WB pipeline registers:

```
always @ (posedge clk or negedge nreset)
if(~nreset) begin
    WB_wload_en <= 1'b0;
    WB_write_en <= 1'b0;
end else if(npe_stop) begin
    WB_wload_en   <= MEM_wload_en;
    WB_write_en   <= MEM_write_ ...;
end

always @ (posedge clk) if(npe_stop) begin
    WB_ex_out     <= MEM_ex_ ...;
    WB_rt_addr    <= MEM_rt_addr;
    WB_rd_addr    <= MEM_rd_addr;

    WB_read       <= MEM_read;
    WB_size       <= MEM_size;
    WB_sign_ext   <= MEM_sign_ext;
    WB_load_FnH   <= MEM_load_FnH;
end
```

```
always @ (*) begin
    WB_memory_out =  WB_read ? ld_ext_ ... :
    64'hxxxx_xxxx_xxxx_xxxx;
end
```

6. Save the changes you made in **Educore.v**.

5 Verify and Analyze pipelined Arm Education Core

5.1 Task: Running simulation using the provided bash script

The following diagram shows the structure of the bash script (run_tests.sh) and makefiles that were provided in the **Lab4_simple_pipeline** folder.

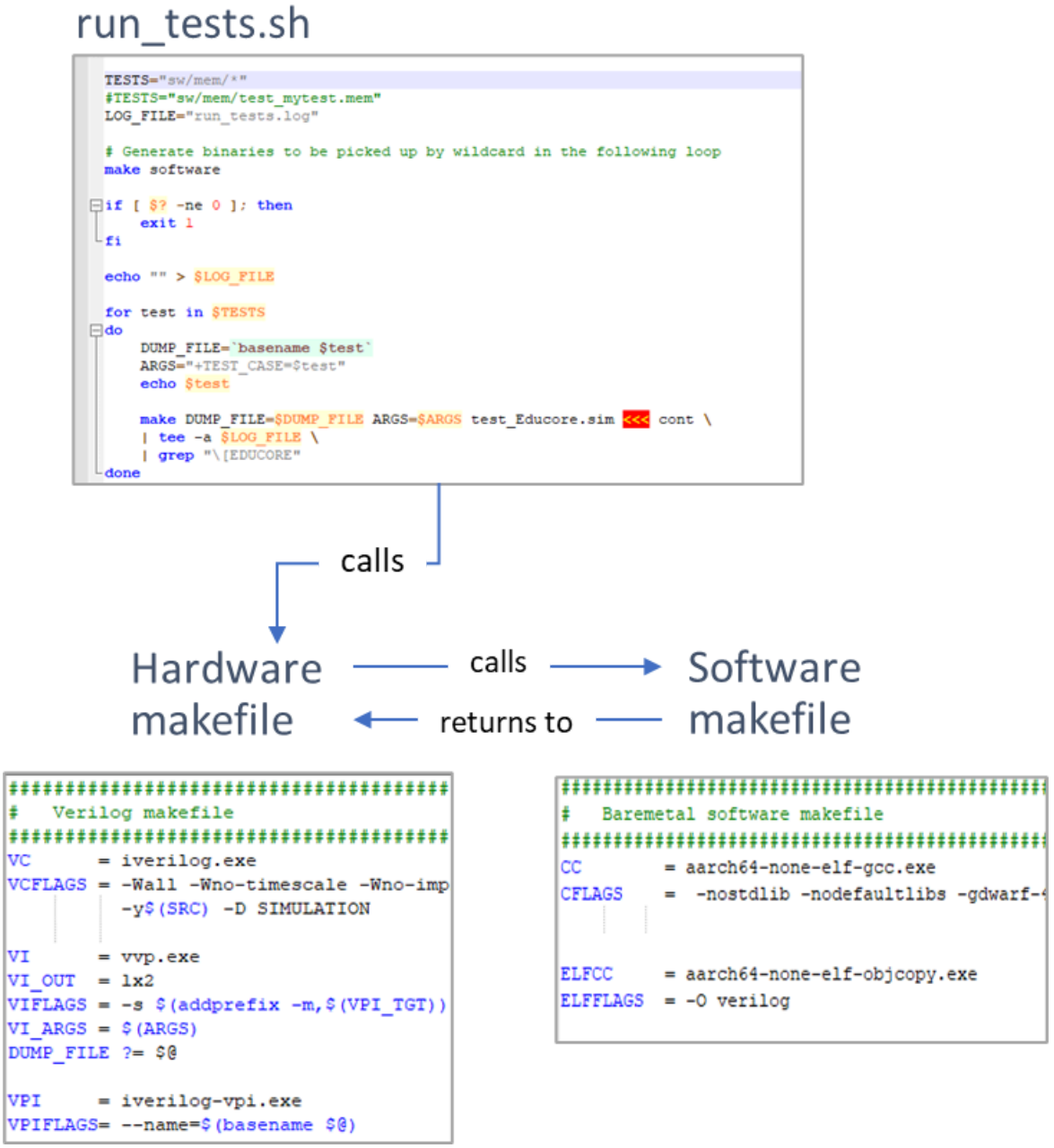


Figure 9: Makefiles and bash script

When the user runs the **run_tests.sh** script, it will call the hardware makefile first, which is at the same folder location as **run_tests.sh**. The hardware makefile will then call the software makefile in the **sw/** directory. The software makefile runs the **gcc** and **objcopy** commands. The hardware makefile compiles the RTL and simulates based on the Assembly tests .mem files generated by the software makefile.

The makefiles and **run_tests.sh** script expect the following file structure in order to work:

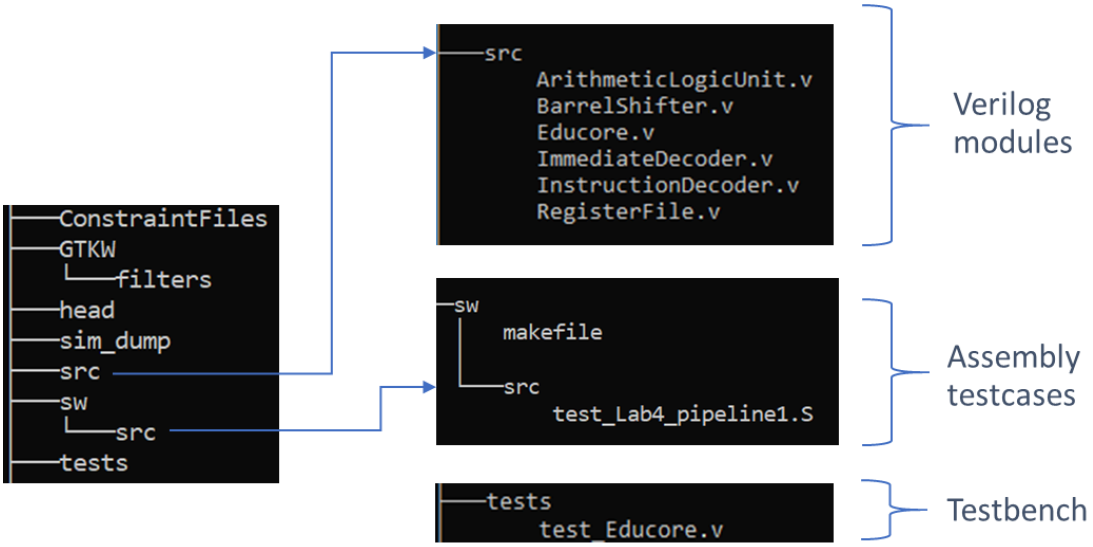


Figure 10: Folder tree of Lab4_simple_pipeline.zip

To run the provided testcase using the bash script, follow these steps:

1. Open a terminal (WSL or MSYS2 on Windows) and change directory to the folder.

   ```
   cd Lab4_simple_pipeline
   ```

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

   ```
   bash run_tests.sh
   ```

Observation:

- The waveforms dumps are placed in the **sim_dump** folder.
- The Assembly test file should complete with “Apollo has landed” message. The waveforms generated in **sim_dump/**folder is ready for inspection.

5.2 Exercise: Analyzing the simple pipeline in Arm Education Core

In this exercise, we will take a closer look at the behavior of the simple pipeline implemented in Arm Education Core.

In Task: Running simulation using bash script, one of the testcase provided is **sw/src/test_Lab4_pipeline1.S**, which has the following code:

```
.global _start

.text

_start:

            MOVZ   X0, #0xf

            MOVZ   X1, #0xe

            MOVZ   X2, #0xd

            MOVZ   X3, #0xc

            MOVZ   X4, #0xb


            ADD    X5, X0, #1

            ADD    X6, X1, X2

            SUBS   X7, X0, X1


_test2:

            ADD    X9, X1, X2

            AND    X10, X9, X3

            ORR    X11, X5, X9

            SUB    X12, X9, X7


            NOP

            NOP

            NOP

            NOP


            YIELD
```

Use the following command to inspect the generated waveforms:

```
cd sim_dump/

gtkwave test_Lab4_pipeline1.mem.lx2 ../GTKW/waveform_Lab4_pipeline1.gtkw
```

If you inspect the generated waveforms in sim_dump\test_Lab4_pipeline1.mem.lx2 using GTKWave, you will observe the following behavior:

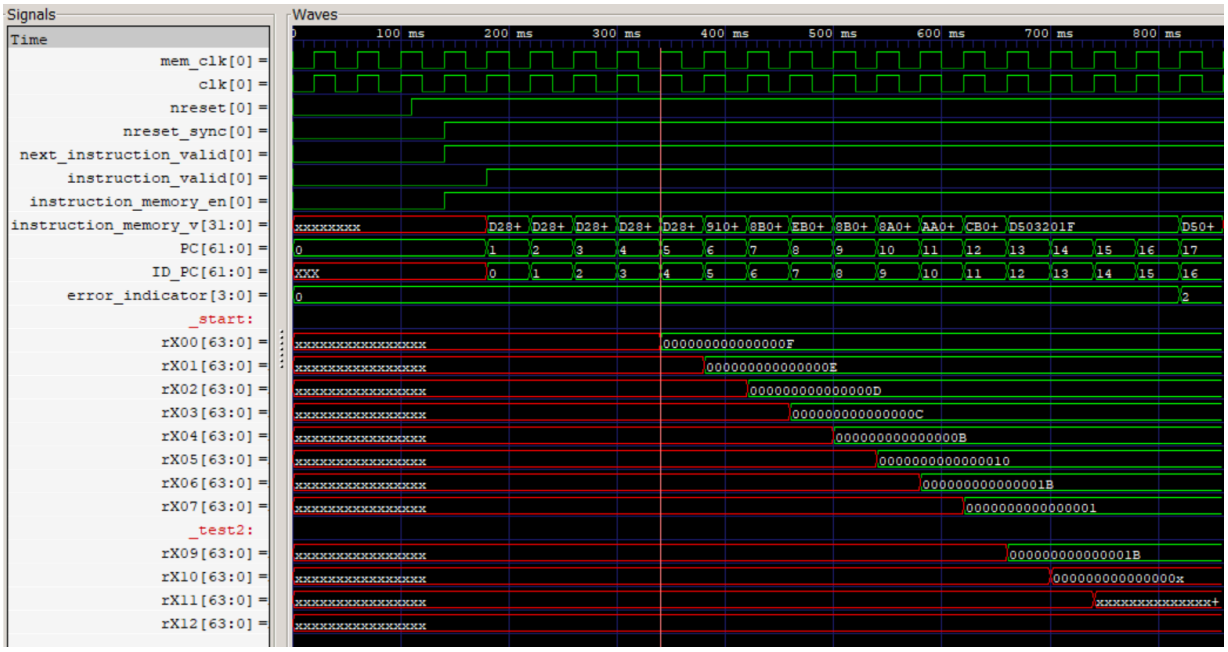


Figure 11: Waveforms from test_Lab4_pipeline1.mem.lx2

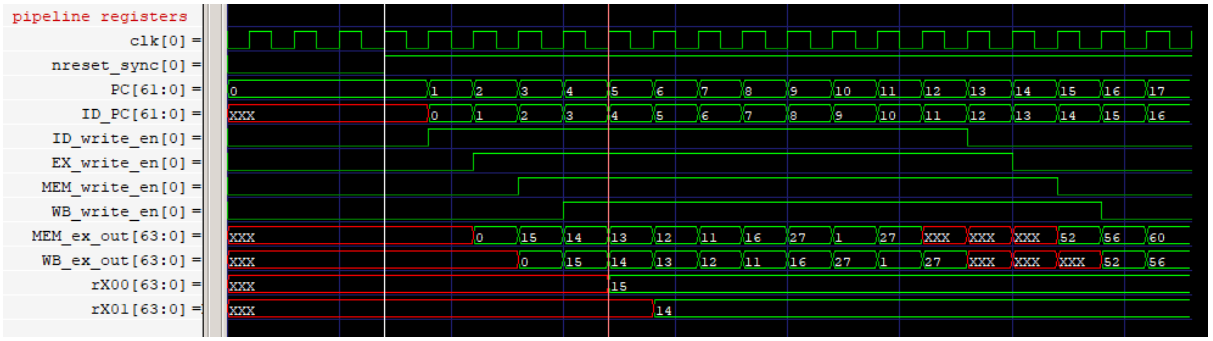


Figure 12: Waveforms of some pipeline registers

Observations:

- The first instruction takes 5 clock cycles to complete.
- The write_en signal propagates through the ID, EXE, MEM, and WB stages at each rising edge of the clock, as shown in Figure 12. Same case for **PC** and **ID_PC**.

Answer the following questions:

1. Why is register X0 only updated at the end of clock cycle 5, but the instructions beyond that seem to update after every clock cycle?

2. Registers X10, X11, and X12 are showing Xs. What are the expected results of these registers?

3. Now, modify the **_test2** section of the Assembly code so that there are NOPs between each instruction, like so:

```
_test2:
            ADD    X9, X1, X2
            NOP
            NOP
            NOP
            AND    X10, X9, X3
            ORR    X11, X5, X9
            SUB    X12, X9, X7

            NOP
            NOP
            NOP
            NOP

            YIELD
```

Now rerun the simulation using the **run_tests.sh** file. Observe the waveform outputs.
Are registers X10, X11, and X12 now showing the expected values? Explain the purpose of the 3 NOPs that you have inserted between the ADD and AND instructions.

4. The YIELD instruction does not depend on any prior register data. What then is the purpose of the NOP instructions just before YIELD?

5. Based on your previous answers, summarize 1 advantage and 1 limitation of the simple pipeline implemented.

6 Summary

In this lab, we have learned the benefits of a pipelined implementation versus a single-cycle implementation. We have implemented a simple pipeline design in Arm Education Core, with increased throughput. However, there are several limitations to the simple pipeline design, as discovered in the analysis exercise. In the next lab, we will look into how we can overcome these limitations.