Computer Architecture Course

LAB 6

Stalls, Control Hazard, and PPA Estimation

Issue 1.0

Contents

1 Introduction

1.1 Lab overview

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

- Identify the need for a stall and a control hazard solution.
- Modify a pipelined processor (Arm Education Core) to implement a stall mechanism.
- Verify the functionality of the stall and control hazard solution mechanism.
- Calculate and compare the average CPI and execution time for single-cycle versus completed pipeline implementation.
- Produce and evaluate synthesis reports to compare the power, area (resource utilization), and timing for single-cycle versus completed pipeline implementation.
- Outline the limitation and potential computer architecture improvements that can be made to a pipelined implementation.

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 and forwarding paths—completed Lab 5

Lab 6 is provided with `Lab6_complete_pipeline.zip`. This zip file contains supplementary files used in the exercises of this lab and contains the Arm Education Core with forwarding paths, which was done in Lab 5.

3 Theory: Stalls

In the previous lab, we have observed that the pipelined Arm Education Core with forwarding paths would not be able to resolve a load-use data dependency without having a NOP instruction in between. For example:

```
LDURB  W10, [X1]

NOP

AND    X11, X10, X7
```

In most data processing instructions, the results are ready at the end of the EXE stage (MEM pipeline register). However, for the load instruction, the result is only ready at the end of the MEM stage (WB pipeline register)—loading from memory takes time. If the NOP instruction were to be removed, the next instruction (AND) would be at the end of its EXE stage when the load instruction finally loads its data. Even with a forwarding path, it will not be possible for AND to execute in time with the forwarded value just before the clock transitions.

One method to resolve this would be to wire the processor in such a way that it would automatically "stall" when the processor detects such load-use data dependencies. Consider the following example shown in the diagram below:

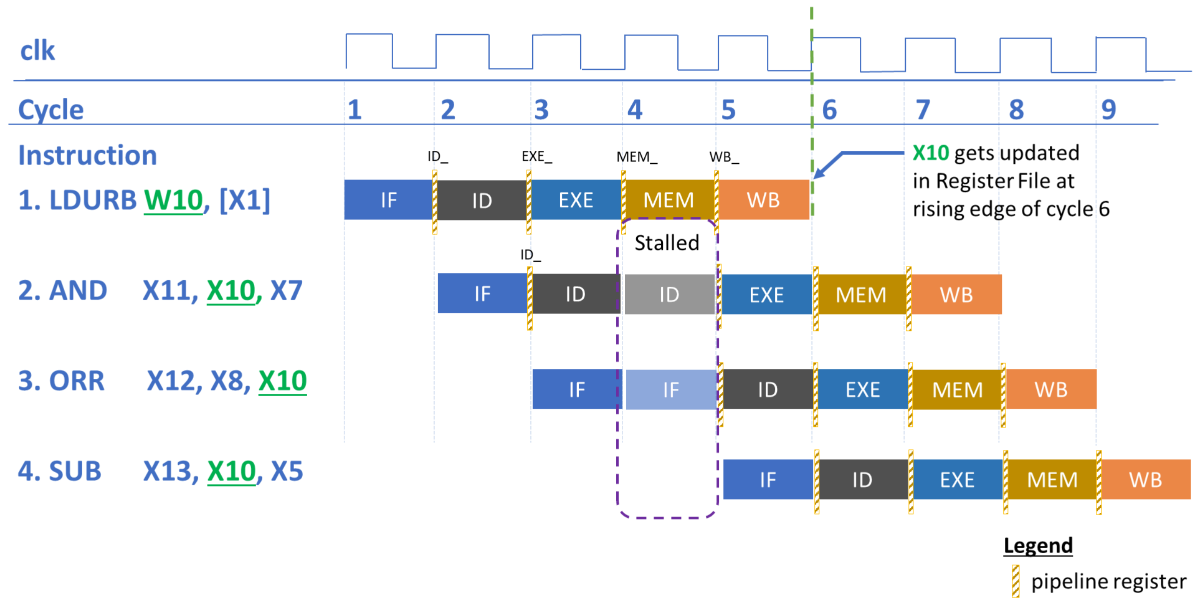


Figure 1: Stall mechanism

At cycle 4, the processor stalls until the data have managed to be loaded from memory. Stalling is basically holding up operations until the data from memory is ready. Therefore, the subsequent instructions (AND, ORR, SUB) are stalled too, until the LDURB memory access stage is complete. During the stall, we should not fetch any instruction from Instruction Memory too—this corresponds to the SUB instruction being delayed in Figure 1.

So what happens to the EXE stage at cycle 4 when the pipeline is stalled? Consider the following table (Table 1) where LDURB is the first instruction:

	IF	ID	EXE	MEM	WB
Cycle 3	ORR	AND	LDURB	-	-
Cycle 4 (stalled)	**ORR**	**AND**	☁	LDURB	-
Cycle 5	SUB	ORR	AND	☁	LDURB
Cycle 6	-	SUB	ORR	AND	☁

Table 1: Bubble in EXE stage for stalls

The EXE stage is unused at Cycle 4 when the pipeline is stalled. This is also known as a bubble, and it behaves like a NOP. The bubble is "inserted" by setting the control signals in the EXE stage to values that would avoid changing state (such as zero for some signals). Do recall that these control signals are also passed to the MEM and WB pipeline registers. The bubble propagates through the pipeline registers until it completes the WB stage.

4 Theory: "Branch bubble"

In the previous lab, we also encountered a Control Hazard. 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 by the time the next instruction is fetched.

There are several approaches to dealing with control hazards, such as delaying the branch, or even branch prediction. In this lab, we will explore one simple method, which is to assume the branch is not taken and continue fetching in subsequent instructions. Once we detect that the instruction requires branching, we replace the subsequent instructions with NOPs. In this lab, we call this the *branch bubble* method.

In theory, the core can detect that it needs to branch once an unconditional branch has been decoded at the ID stage, but for a conditional branch (e.g., `BNE`), the core relies on the result computed in the EXE stage to be able to decide if it needs to branch. The branch bubble mechanism aims to accommodate both unconditional branch and conditional branch by taking the worst-case scenario, where a decision to branch is only ready at the EXE stage. The branch bubble is a simple mechanism, and it is simple to implement to ensure that the instructions after the branch instruction that are not meant to execute will not execute, though this mechanism is also not the most efficient in terms of performance.

Consider the following examples shown in Figure 2 and Table 2.

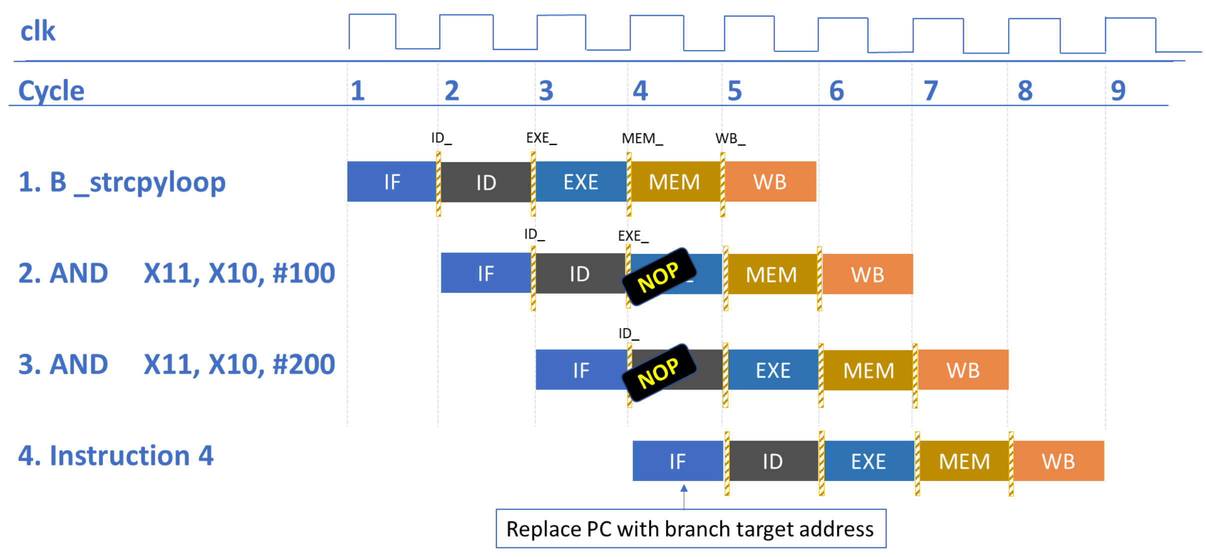


Figure 2: Branch bubble mechanism

	IF	ID	EXE	MEM	WB
Cycle 3	AND X1, X10, #200	AND X1, X10, #100	B _strcpyloop	-	-
Cycle 4	**_strcpyloop**	**NOP**	**NOP**	B _strcpyloop	-
Cycle 5	-	**_strcpyloop**	**NOP**	**NOP**	B _strcpyloop
Cycle 6	-	-	**_strcpyloop**	**NOP**	**NOP**

Table 2: Bubble in EXE stage for control hazards

At cycle 3, the branch instruction is in the EXE stage, and the core decides to branch. At cycle 4, the core fetches the branch target address and replaces the inputs of ID stage and EXE stage with NOPs.

5 Completing Arm Education Core's pipeline

5.1 Stall implementation

One way of implementing stalls in Arm Education Core is to have **a stall control signal**. Consider the following scenario shown in the diagram below.

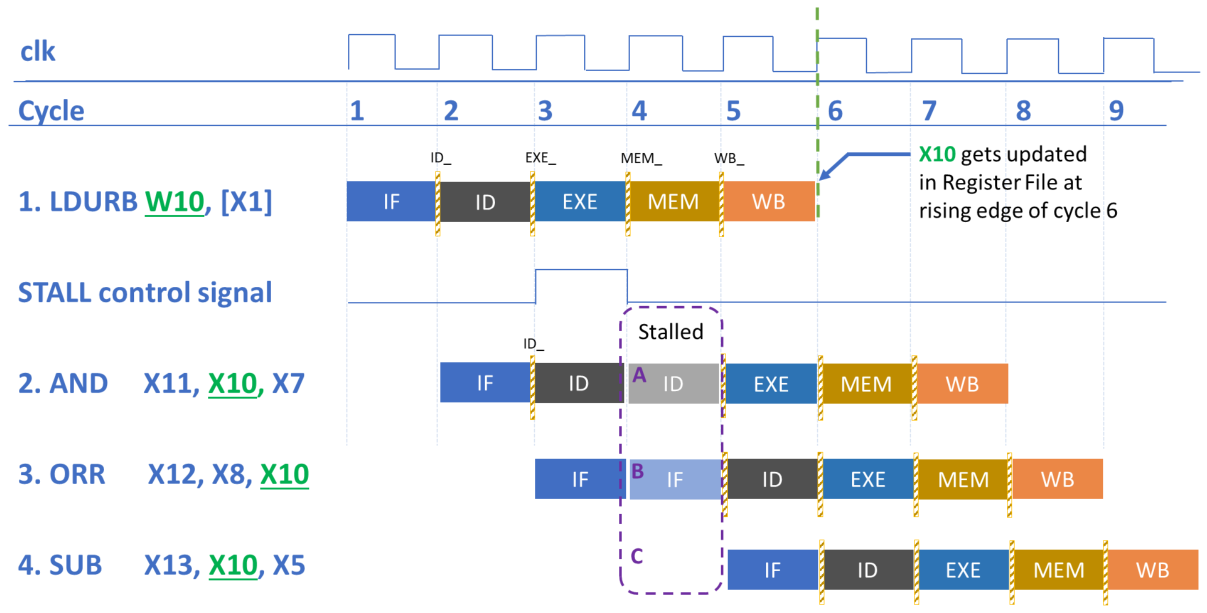


Figure 3: Implementing stalls in Arm Education Core

	IF	ID	EXE	MEM	WB
Cycle 3	ORR	AND	LDURB	-	-
Cycle 4 (stalled)	ORR	AND	☁	LDURB	-
Cycle 5	SUB	ORR	AND	☁	LDURB
Cycle 6	-	SUB	ORR	AND	☁

When and how to assert stall control signal

To detect load-use data dependencies by comparing the target/destination index for the load instruction (LDURB) and the register operand index of the next instruction (AND), both instructions need to be decoded first. Therefore, the earliest we could compare both instructions' register operands to detect hazard is when the earlier instruction is in the EXE stage and the next instruction is in the ID stage.

In the example in Figure 3, at cycle 3, the processor detects a load instruction in the EXE_ pipeline and a data dependency in the ID_ pipeline (by comparing register operand indexes). Upon such detection, the stall control signal is asserted high.

Stalled at cycle 4

Referring to Figure 3, when the stall control signal is high in cycle 3, the following needs to happen in the next clock cycle (Cycle 4) in `Educore.v`:

- The PC will not be updated with the next value—this corresponds to the stall labeled **[B]** in Figure 3, **where the PC will still point to the address to fetch the ORR instruction**. The PC, therefore, retains its same value.
- We will not update the `instruction[31:0]` in the ID stage—this also corresponds to the stall labeled **[A]** in Figure 3, **where instruction register will still point to address fetched (AND instruction)**. Do recall that `instruction[31:0]` in the ID stage gets its value from the Instruction Memory in the testbench in a combinatorial way. The Instruction Memory is synchronous to `mem_clk`, which is the same as `core_clk` in this lab. To ensure that `instruction[31:0]` remains the same, we can disable the `instruction_memory_en` signal in cycle 3 so that in cycle 4 `instruction[31:0]` remains the same.
- We will not update the ID_PC value—this corresponds to the stall labeled **[A]** in Figure 3.
- We want to set the EXE control signals to zero—this is to insert a bubble, see Table 1. These control signals should include any MEM access and WriteBack control signals.
- The delay of the SUB instruction labeled **[C]** in Figure 3 naturally happens due to the stalling of IF in **[B]**.

5.2 Branch bubble implementation

One way to implement the "Branch Bubble" in Arm Education Core is to have **a branch bubble control signal**. Consider the following scenario shown in the diagram below.

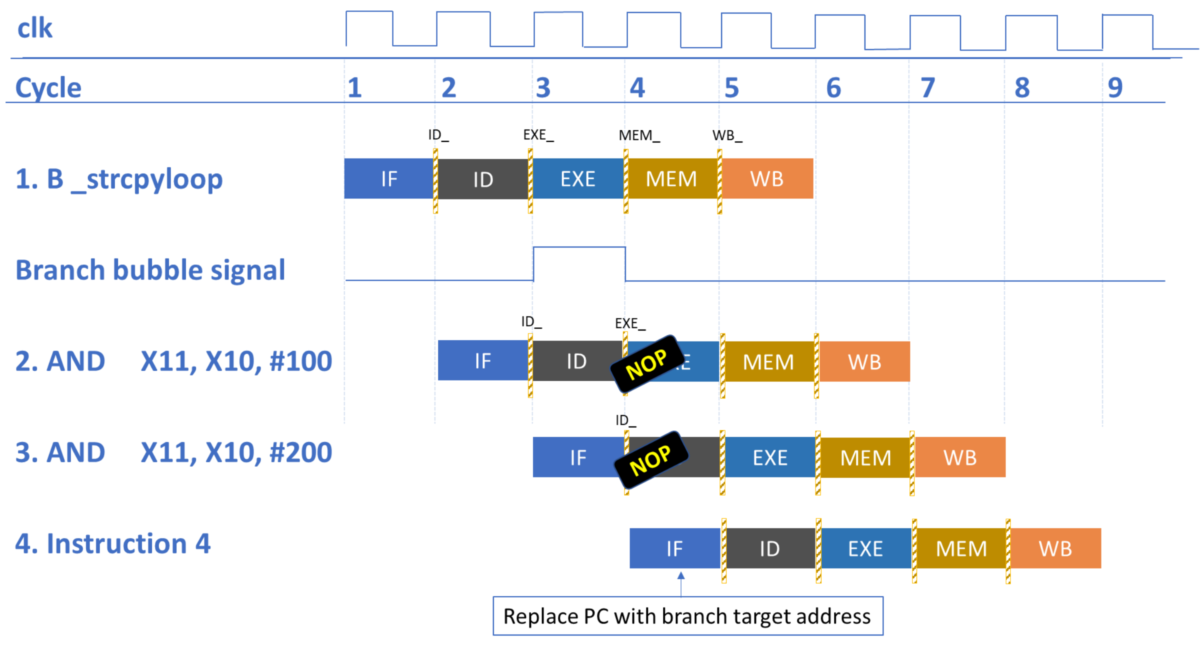


Figure 4: Branch bubble implementation in Arm Education Core

<u>When and how to assert branch bubble control signal</u>

We can technically detect that an instruction is a branch instruction at the ID stage, but for a conditional branch, for example, the decision on whether to branch or not depends on the EXE results. Therefore, we can assert the branch control signal at the EXE stage in `Educore.v`, when it detects:

- An unconditional branch (`br_taken` = 1 due to `br_condition_mux` from the Instructor Decoder being 1).
- A conditional branch and a match in PSTATE (which will cause `br_taken` = 1).
- If the `next_PC_mux` at the EXE stage specifies that the PC will be sourced from a register address (Unconditional Branch to Register, BR instruction, for example).

Referring back to Figure 4, when the bubble control signal is asserted in `Educore.v`, the following needs to happen:

At cycle 3 (branch bubble control asserted)

- We want to set the EXE control signals to zero at the next clock cycle—effectively a NOP. Any of the control signals that is to be updated in the EXE_ pipeline registers in the next clock cycle should be set to zero, or in some cases, a value to reflect no change. These control signals should include any MEM access and WriteBack control signals.
- The `next_instruction_valid` signal is deasserted. This will make the `instruction_valid` signal to be deasserted at the next clock cycle (cycle 4).
- Since the B instruction has completed the EXE stage by now, the IF stage at the next clock cycle (cycle 4) can be replaced with the branch target address. Therefore, we can disable the read of instruction from Instruction Memory for the next clock cycle by setting `instruction_memory_en` =0 at cycle 3.

At cycle 4 (branch bubble control deasserted)

- The control signals of the EXE_ pipeline registers have been updated with zero, or in some cases, a value that reflects no change, effectively behaving like the first NOP for Instruction 2 in Figure 4 (`NOP` replaces `AND X11, X10, #100`).
- The `instruction_valid` signal is deasserted. When this happens, we could replace the `instruction` signal in the ID stage with a NOP—see instruction 3 in Figure 4 (`NOP` replaces `AND X11, X10, #200`).

5.3 Task: Code modifications

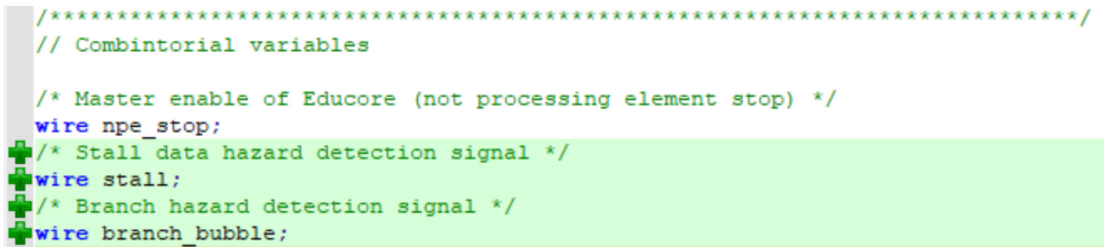
To implement stalls and branch bubbles in Arm Education Core, follow these instructions:

1. Create a working directory folder called `Lab_6`.

2. Unzip the provided `Lab6_complete_pipeline.zip` in your working directory `Lab_6` folder.

3. Open the `src\Educore.v` file—this file already has forwarding paths.

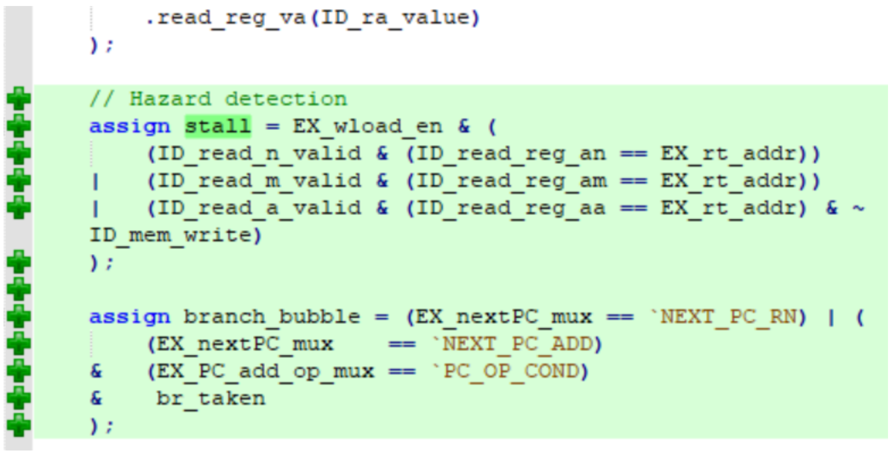
4. Add 2 control signals called `stall` and `branch_bubble` as shown below:



```
/************************************************************************/
// Combintorial variables

/* Master enable of Educore (not processing element stop) */
wire npe_stop;
/* Stall data hazard detection signal */
wire stall;
/* Branch hazard detection signal */
wire branch_bubble;
```

5. Add the following Hazard Detection Logic after the Register File module instantiation:



```
        .read_reg_va(ID_ra_value)
    );

    // Hazard detection
    assign stall = EX_wload_en & (
        (ID_read_n_valid & (ID_read_reg_an == EX_rt_addr))
    |   (ID_read_m_valid & (ID_read_reg_am == EX_rt_addr))
    |   (ID_read_a_valid & (ID_read_reg_aa == EX_rt_addr) & ~
    ID_mem_write)
    );

    assign branch_bubble = (EX_nextPC_mux == `NEXT_PC_RN) | (
        (EX_nextPC_mux    == `NEXT_PC_ADD)
    &   (EX_PC_add_op_mux == `PC_OP_COND)
    &    br_taken
    );
```

This hazard detection logic implements multiplexes to decide when to assert the `stall` and `branch_bubble` control signals. A stall is asserted when there is load-use data dependence, i.e., register index in the ID stage is the same as the target/destination register of the load instruction (previous instruction) at the EXE stage. This should also happen only if the registers are modifiable and when the next instruction is not a store instruction (`ID_mem_write` comes from the Instruction Decoder and is 1 when the instruction decoded writes to memory—a store instruction).

Recall from Lab 3A:

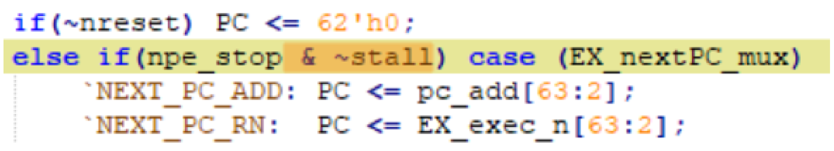
- The `EX_nextPC_mux` signal comes from the Instruction Decoder, and it specifies whether the next PC value is sourced from a register or an adder.

- The EX_PC_add_op_mux signal also comes from the Instruction Decoder. It specifies the next PC from the adder as the current one plus 4 (next instruction), or the value of this instruction's PC plus the branch offset conditional to PSTATE matching required state.
- `br_taken` is asserted when there is an unconditional branch instruction or the branch condition matches the PSTATE.

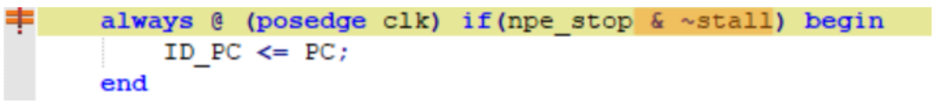
A branch bubble happens when the next PC value is sourced from a register (unconditional branch), or if a conditional branch is taken.

6. Modify `Educore.v` to include the `stall` and `branch_bubble` control signals as conditions:

   a. In the Instruction Fetch stage, we need to stall the PC and the PC value going into the ID stage—(see stall labeled **B** and **A** in Figure 3):



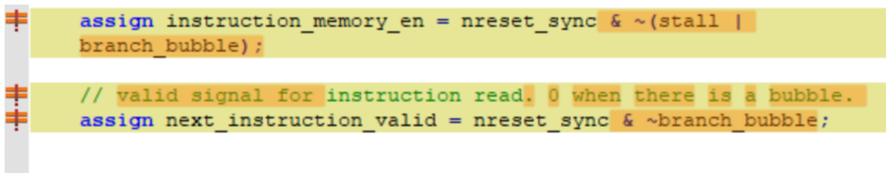
```
if(~nreset) PC <= 62'h0;
else if(npe_stop & ~stall) case (EX_nextPC_mux)
    `NEXT_PC_ADD: PC <= pc_add[63:2];
    `NEXT_PC_RN:  PC <= EX_exec_n[63:2];
```



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

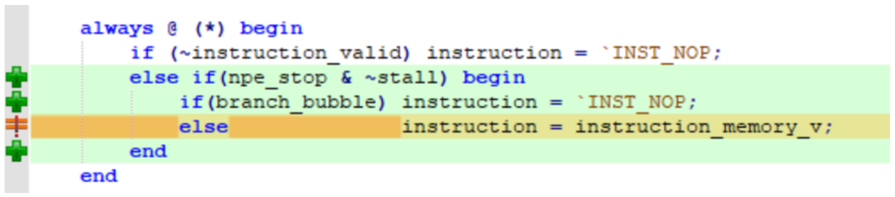
   b. When there is a stall or branch bubble, the `instruction` signal should not get updated by values from the Instruction Memory (see stall type **B** in Figure 3). We do this by disabling the `instruction_memory_en` signal so that no instruction will load from Instruction Memory to the `instruction` signal.
   When there is a branch bubble, we will also need to deassert `next_instruction_valid` signal so that the `instruction_valid` signal in the next clock cycle will be deasserted, causing `instruction` to be replaced with a NOP.

   Note: `` `INST_NOP `` instruction encoding is defined in `head/Educore.vh`.



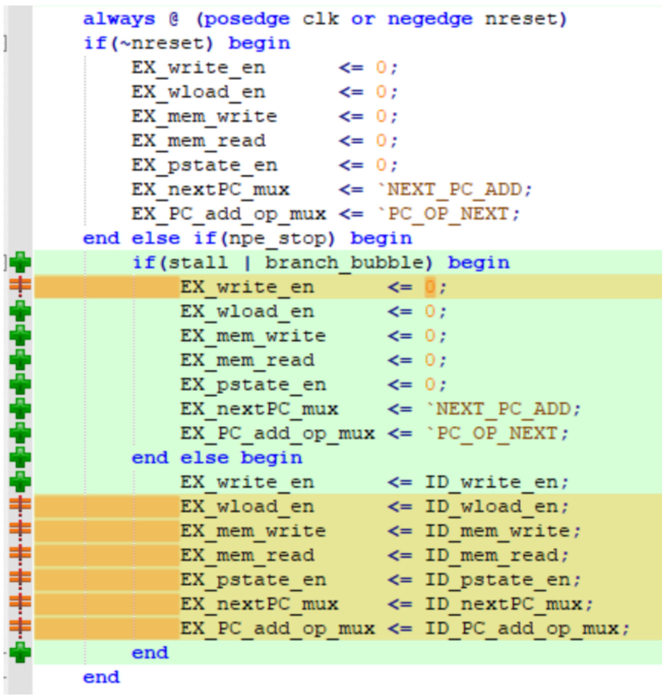
```
assign instruction_memory_en = nreset_sync & ~(stall |
branch_bubble);

// valid signal for instruction read. 0 when there is a bubble.
assign next_instruction_valid = nreset_sync & ~branch_bubble;
```



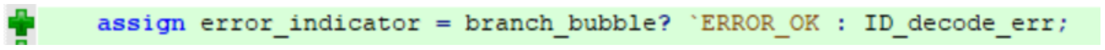
```
always @ (*) begin
    if (~instruction_valid) instruction = `INST_NOP;
    else if(npe_stop & ~stall) begin
        if(branch_bubble) instruction = `INST_NOP;
        else              instruction = instruction_memory_v;
    end
end
```

c. Don’t forget to introduce the bubbles at the EXE stage of the next clock cycle when there is a branch or stall—this is done by disabling the relevant control signals:



```
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
    if(stall | branch_bubble) 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 begin
        EX_write_en      <= ID_write_en;
        EX_wload_en      <= ID_wload_en;
        EX_mem_write     <= ID_mem_write;
        EX_mem_read      <= ID_mem_read;
        EX_pstate_en     <= ID_pstate_en;
        EX_nextPC_mux    <= ID_nextPC_mux;
        EX_PC_add_op_mux <= ID_PC_add_op_mux;
    end
end
```

7. Lastly, modify the `error_indicator` signal so that:



```
assign error_indicator = branch_bubble? `ERROR_OK : ID_decode_err;
```

This is a workaround so that it will not give us a false alarm when we run some of the testcases provided, for example, if there is a BRK instruction while waiting for the branch to execute.

8. Save the file.

6 Task: Verify and analyze completed Arm Education Core

In this task, we will simulate our complete Arm Education Core with Lab 5's testcase again but with added instructions that use conditional branch (see `_strcpyloop`). Arm Education Core should be working correctly for the branch instructions and load-use dependency instructions. The `Lab6_complete.S` has the following instructions:

```
.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

      MOVZ   X20, #0x013C


// store values in memory

      STURB  W5, [X0]

      STURB  W6, [X0, #1]

      STURB  W7, [X0, #2]

      STURB  W8, [X0, #3]

      STURB  WZR, [X0, #4]


// test load-store dependency

      LDURB  W10, [X0]

      STURB  W10, [X0, #5]


// 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]
      AND   X11, X10, X7
      ORR   X12, X8, X10
      SUB   X13, X10, X5

// test Branch
      B            _strcpyloop
      ADD          X11, X10, #100     // this shouldn't execute
      ADD          X11, X10, #200     // this shouldn't execute
      ADD          X11, X10, #300     // this shouldn't execute

_strcpyloop:
    LDRB     W21, [X0], #1
    STRB     W21, [X20], #1
    CMP      X2, #0
    BNE      _strcpyloop    // If not, repeat the _strcpyloop

_test:
      NOP
      NOP
      NOP
      NOP
      YIELD
```

Run the provided testcase in `Lab6_complete_pipeline.zip` using the bash script, by following these steps:

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

   ```
   cd Lab6_complete_pipeline
   ```

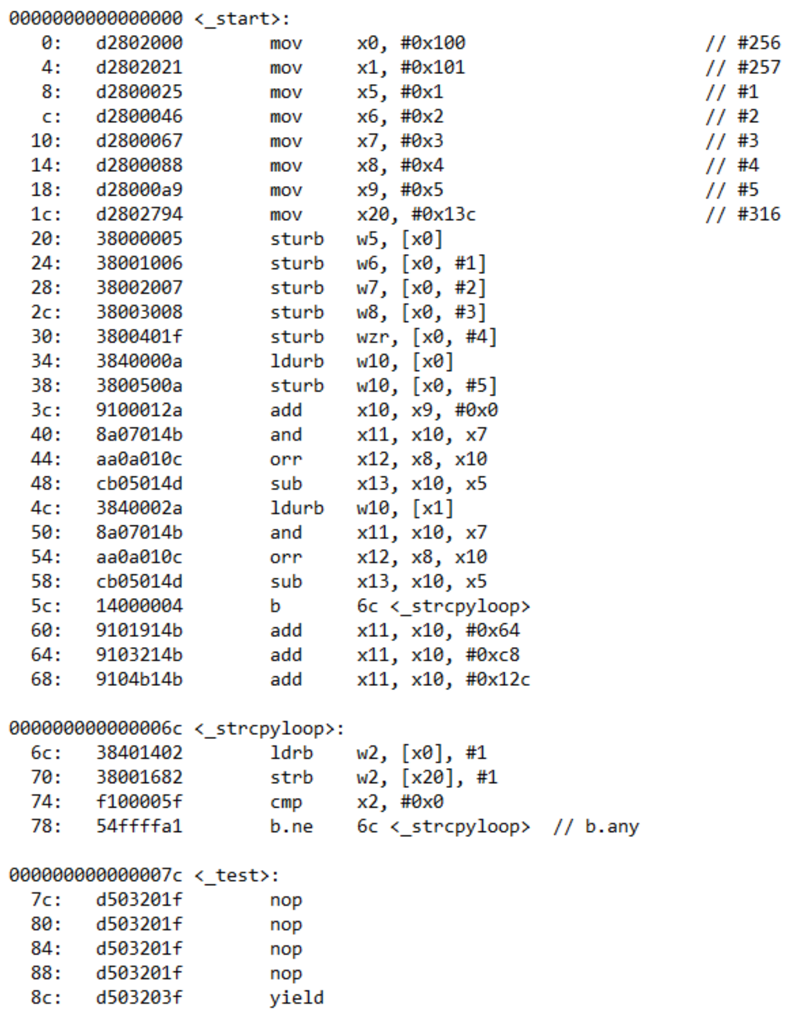
2. Run the run_tests.sh file by entering the following command:

   ```
   bash run_tests.sh
   ```

3. Use `objdump` to generate a human-readable encoding and corresponding program counter values for the Assembly instructions:

   ```
   cd sw/elf

   aarch64-none-elf-objdump -d test_Lab6_complete.elf >
   Lab6_complete.diassembly
   ```



```
0000000000000000 <_start>:
   0:   d2802000        mov     x0, #0x100                      // #256
   4:   d2802021        mov     x1, #0x101                      // #257
   8:   d2800025        mov     x5, #0x1                        // #1
   c:   d2800046        mov     x6, #0x2                        // #2
  10:   d2800067        mov     x7, #0x3                        // #3
  14:   d2800088        mov     x8, #0x4                        // #4
  18:   d28000a9        mov     x9, #0x5                        // #5
  1c:   d2802794        mov     x20, #0x13c                     // #316
  20:   38000005        sturb   w5, [x0]
  24:   38001006        sturb   w6, [x0, #1]
  28:   38002007        sturb   w7, [x0, #2]
  2c:   38003008        sturb   w8, [x0, #3]
  30:   3800401f        sturb   wzr, [x0, #4]
  34:   3840000a        ldurb   w10, [x0]
  38:   3800500a        sturb   w10, [x0, #5]
  3c:   9100012a        add     x10, x9, #0x0
  40:   8a07014b        and     x11, x10, x7
  44:   aa0a010c        orr     x12, x8, x10
  48:   cb05014d        sub     x13, x10, x5
  4c:   3840002a        ldurb   w10, [x1]
  50:   8a07014b        and     x11, x10, x7
  54:   aa0a010c        orr     x12, x8, x10
  58:   cb05014d        sub     x13, x10, x5
  5c:   14000004        b       6c <_strcpyloop>
  60:   9101914b        add     x11, x10, #0x64
  64:   9103214b        add     x11, x10, #0xc8
  68:   9104b14b        add     x11, x10, #0x12c

000000000000006c <_strcpyloop>:
  6c:   38401402        ldrb    w2, [x0], #1
  70:   38001682        strb    w2, [x20], #1
  74:   f100005f        cmp     x2, #0x0
  78:   54ffffa1        b.ne    6c <_strcpyloop>  // b.any

000000000000007c <_test>:
  7c:   d503201f        nop
  80:   d503201f        nop
  84:   d503201f        nop
  88:   d503201f        nop
  8c:   d503203f        yield
```

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

   ```
   cd ../../sim_dump/

   gtkwave test_Lab6_complete.mem.lx2 ../GTKW/debug.gtkw
   ```

5. Open the `Lab6_complete.disassembly` file. Use the program counter (left column) and encoding values to inspect the waveforms and check if they match the expected results of the Assembly code.

Observations:

1. Let's inspect the load-use dependency scenario. According to the `Lab6_complete.disassembly` file, this starts at the Program Counter value of 0x48 (decimal 72). The corresponding `PC[61:0]` would have a value of decimal 18.

```
4c:    3840002a        ldurb    w10, [x1]
50:    8a07014b        and      x11, x10, x7
54:    aa0a010c        orr      x12, x8, x10
58:    cb05014d        sub      x13, x10, x5
```

   The following diagram shows the stall operation kicking in and the registers updated according to their expected values.

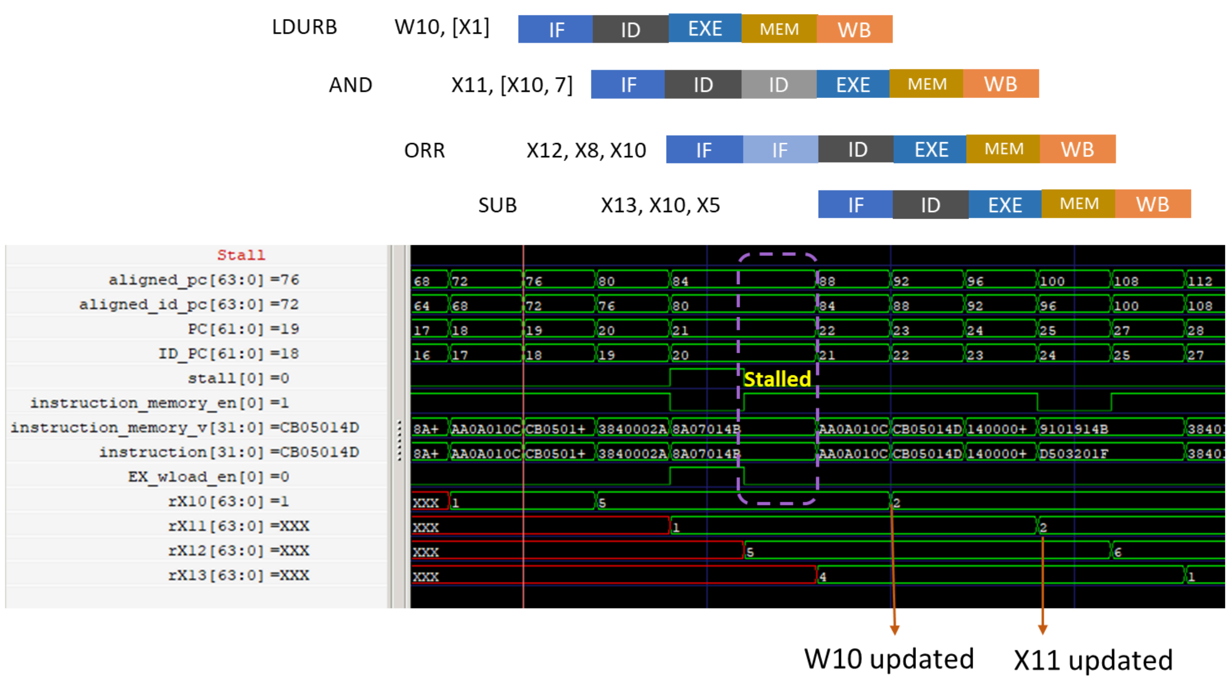


Figure 5: Stall operation working in Arm Education Core

2. Let's now inspect the branch hazard scenario. According to the `Lab6_complete.disassembly` file, the unconditional branch starts at the Program Counter value of 0x58 (decimal 88). The corresponding `PC[61:0]` would have a value of decimal 22.

```
  5c:   14000004        b       6c <_strcpyloop>
  60:   9101914b        add     x11, x10, #0x64
  64:   9103214b        add     x11, x10, #0xc8
  68:   9104b14b        add     x11, x10, #0x12c

000000000000006c <_strcpyloop>:
  6c:   38401402        ldrb    w2, [x0], #1
  70:   38001682        strb    w2, [x20], #1
  74:   f100005f        cmp     x2, #0x0
  78:   54ffffa1        b.ne    6c <_strcpyloop>
```

The following diagram shows the branch bubble operation kicking in and the registers updated according to their expected values.

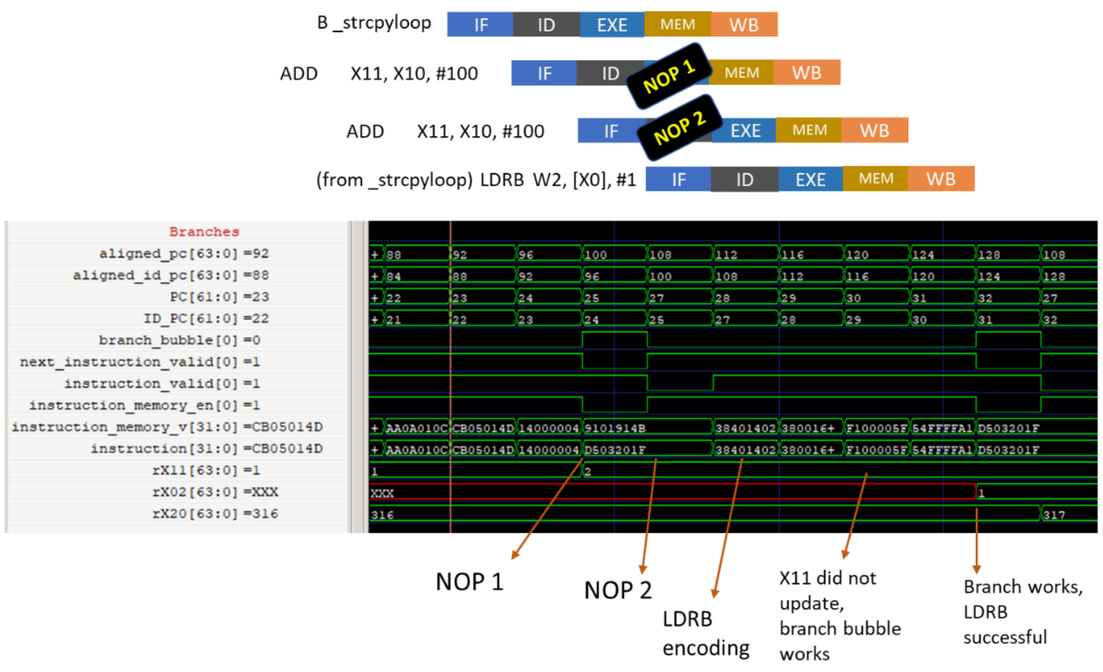


Figure 6: Branch bubble working in Arm Education Core

6.1 [Optional] Additional testcases

Lab 6 Arm Education Core comes with additional test Assembly source files that you can run. The Assembly instructions in these files are beyond the scope of this course, but you can use them to test that all instructions that Arm Education Core was meant to support work.

To run the additional testcases, follow these steps:

1. Copy all the .S files from `Lab6_complete_pipeline/additional_testcases/src` and paste in `Lab6_complete_pipeline/sw/src/`.
2. Run the run_tests.sh file by entering the following command:

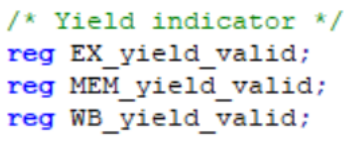
```
bash run_tests.sh
```

You should see each “Apollo has landed” message for each test. If any tests fail, ensure that the changes you have made in Task: Code modifications are correct.

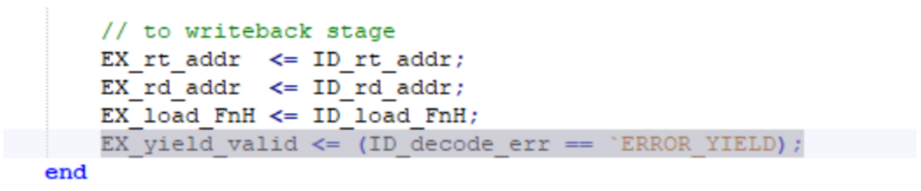
7 [Optional] Task: Insert NOPs automatically for YIELD instruction

In the previous labs, we usually had to insert 4 NOPs before a YIELD instruction so that the simulation would not terminate prematurely. One workaround is to modify Arm Education Core to automatically NOPs. The following code snippets show an example of a workaround in `Educore.v`:

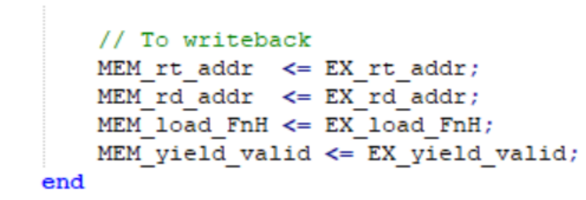
1. Declare yield valid signals that will be propagated through the pipeline stages:



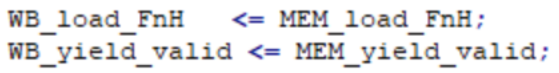
```
/* Yield indicator */
reg EX_yield_valid;
reg MEM_yield_valid;
reg WB_yield_valid;
```



```
        // to writeback stage
        EX_rt_addr   <= ID_rt_addr;
        EX_rd_addr   <= ID_rd_addr;
        EX_load_FnH <= ID_load_FnH;
        EX_yield_valid <= (ID_decode_err == `ERROR_YIELD);
    end
```

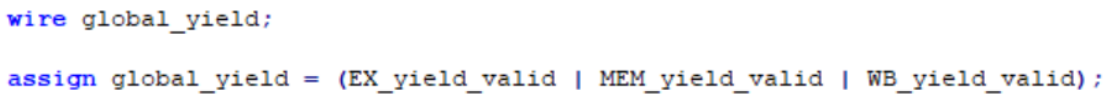


```
        // To writeback
        MEM_rt_addr   <= EX_rt_addr;
        MEM_rd_addr   <= EX_rd_addr;
        MEM_load_FnH <= EX_load_FnH;
        MEM_yield_valid <= EX_yield_valid;
    end
```



```
WB_load_FnH    <= MEM_load_FnH;
WB_yield_valid <= MEM_yield_valid;
```

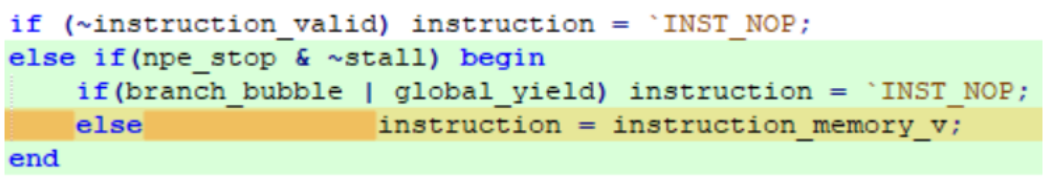
2. Declare a global yield signal



```
wire global_yield;

assign global_yield = (EX_yield_valid | MEM_yield_valid | WB_yield_valid);
```

3. Add a global yield signal as a condition to fetch NOPs.



```
if (~instruction_valid) instruction = `INST_NOP;
else if(npe_stop & ~stall) begin
    if(branch_bubble | global_yield) instruction = `INST_NOP;
    else                           instruction = instruction_memory_v;
end
```

4. Save the file.

8 Exercise: Performance, Power, and Area estimation and comparison

8.1 Average CPI and throughput

1. One of the ways of comparing performance in different processors is by using benchmark programs. For example, there are 3 popular benchmarks, which are the Dhrystone, CoreMark, and SPEC (by Standard Performance Evaluation Corporation).

   Let's assume that we have a benchmark program that consists of the following:

   - 25% loads
   - 10% stores
   - 13% branches, of which 40% of the branches are taken
   - 52% data processing instructions, of which:
     - 20% of these data processing instructions are followed by another data processing instruction that has dependence (requires a forwarding path)
     - 30% of these data processing instructions are preceded by a load instruction that it is dependent on.

   Based on the data above:
   CPI of loads = 1
   CPI of stores = 1
   CPI of branches = 0.6(1) + 0.4(number of cycles for branch to complete)
   CPI of data processing instructions = 0.50(1) + 0.2(number of cycles needed when there is data dependence) + 0.3(number of cycles needed when there is load-use data dependence/to stall loads)

   Average CPI
   = sum of (fraction of time that instruction is used × CPI for instruction type).
   = 0.25(CPI of loads) + 0.1(CPI of stores) + 0.13(CPI of branches) + 0.52(CPI of data processing instructions).

   a. Determine the average CPI for this benchmark in a single-cycle Arm Education Core.

b. Determine the average CPI for this benchmark in the simple pipelined Arm Education Core in Lab 4 (no forwarding path).
   Assuming there is no forwarding path and you can only get the correct results by introducing NOPs in the code, the CPI is therefore:

   CPI of branches = 0.6(1) + 0.4(1+2) = 1.8

   CPI of data processing of instructions = 0.50(1) + 0.2(1+3NOPs) + 0.3(1+3NOPs) = 2.5

c. Determine the average CPI for this benchmark in the complete pipelined Arm Education Core (with forwarding paths, stalls, and branch bubbles).

   CPI of branches = 0.6(1) + 0.4(1+2NOPs) = 1.8

   CPI of data processing of instructions = 0.50(1) + 0.2(1) + 0.3(2) = 1.3

d. Assume that the `mem_clk` is running at 10µs per clock cycle. As seen in the previous labs's simulation, the single-cycle Arm Education Core minimum `core_clk` cycle period needs to be 2 times the `mem_clk` in order to function correctly. On the other hand, the pipelined Arm Education Core can perform at the same speed as `mem_clk`.
   Based on these assumptions, calculate the average execution time for the:
   a. single-cycle Arm Education Core
   b. pipelined Arm Education Core with no forwarding path, stalls, or branch bubbles (Lab 4 version)
   c. completed Arm Education Core version.
   Average Execution time = average CPI × time.

e. Compare the average CPI and execution time of all 3 Arm Education Core implementations. Explain and discuss which implementation had the best and worst performance in terms of average execution time.

8.2 Synthesis

When designing a processor, the Power, Performance, and Area (PPA) are factors that need to be considered and delicately balanced. In the previous exercise, we estimated the performance of the 3 types of Arm Education Core in terms of execution time.

Power and Area are better estimated when using an *Application Specific Integration Circui*t (ASIC) design flow, which will require EDA tools and component libraries. In this exercise, we will generate a synthesis and implementation report to get a rough idea of the impact of the 3 microarchitecture types on Power and Area using an FPGA flow. Estimating Area using FPGA is not accurate; therefore, in this exercise, we will aim to get a ballpark estimate by observing the percentage of resource utilization, such as flip-flops and LUTs. Do bear in mind that you will need to take into account of each target (FPGA, CMOS, etc.), and one design may fit a target technology better than another.

In this exercise, we will use the Xilinx Vivado WebPACK software. Because we are synthesizing just the processor core itself without any wrapper around the core module, you will need at least 300 I/Os (corresponds to approximately all of Arm Education Core I/O signal bits). Therefore, you will need to choose a suitable FPGA that has a minimum of 300 I/O pins, such as the Xilinx FPGA part number xc7a75tfgg676-3. The Vivado WebPACK software by Xilinx can be downloaded for free, see the *Getting Started Guide*.

Lab 6 comes with a folder called `synthesis_exercise`. This folder contains all 3 Arm Education Core types (single cycle, a simple pipeline with no forwarding path/stalls/bubbles, and complete pipeline).

8.2.1 Setting up Vivado HDL WebPACK for synthesis

Synthesis of Arm Education Core is an exercise in one of the labs. We recommend using Xilinx Vivado HDL WebPACK (free) for synthesis.

To install Vivado HDL WebPACK, follow these steps:

1. Download the Vivado HDL WebPACK listed in Software requirements. You may need to create an account with Xilinx first.
2. Unzip the downloaded file and run the setup file on your Windows machine.
3. Accept all the default choices as you click through the installation.

8.2.2 Running synthesis

For each Arm Education Core type, do the following steps:

1. Launch Xilinx Vivado.
2. Select **Project** > **New** > **Next**.
3. Store project location appropriately and select **Next**.
4. Select **RTL project** > **Next**.
5. Select **Add Directories** and select the /head and /src folders of the Arm Education Core folder. Select **Open** and click **Next**.
6. Select **Add Files** and select the `test.xdc` file provided in the `synthesis_exercise` folder (clock period is set to 60 ns). Select **Open** and click **Next**.
7. Search for xc7a75tfgg676-3. Select it and click **Next** and **Finish**.
8. Click on **Run Implementation** in the **Flow Navigator bar** on the left. This will kick off both synthesis and implementation. Select **OK** > **OK**.
9. Once the implementation has completed (it will take some time), expand **Open Implemented Design** and click **Report Utilization** on the **Flow Navigator bar** on the left. Use the default settings and click **OK**. Take note of the utilization of resource table in the Summary tab.
10. Click **Report Power** on the **Flow Navigator bar** on the left. Select the **Switching** tab. Under **Simulation Settings**, add the following core Simulation activity file (.saif) and click **OK**. Take note of the on-chip power graph in the Summary tab.

    ```
    Synthesis_folder/<educoreversion>.saif
    ```

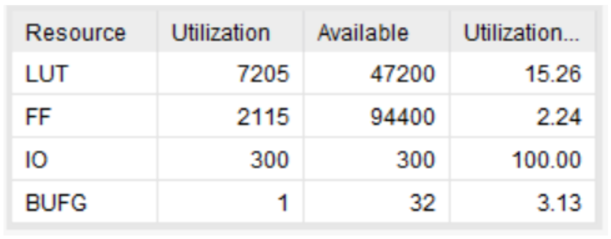
    Note: The .saif of *Switching Activity Interchange Format* file contains toggle counts on the signals of the design and timing attributes that specify time durations for signals at level 0, 1, X, or Z. The .saif files provided were generated based on test_Lab3.s, which works on all 3 core types, and were simulated in advance for each core type. For information on how to generate .saif file, see https://www.xilinx.com/support/answers/52632.html

11. Click on **Report Timing Summary** on the **Flow Navigator bar** on the left. Use the default settings and click **OK**. Take a note of the Worst Negative Slack value.
12. Repeat all these steps for the next Arm Education Core type.

Observations (clk = 60 ns):

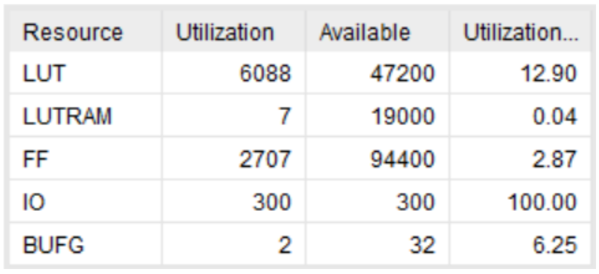
Post-implementation Utilization report

Single-cycle



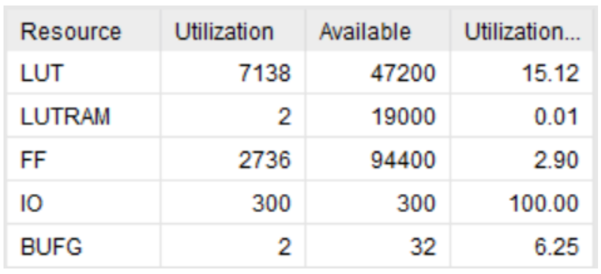
Resource	Utilization	Available	Utilization...
LUT	7205	47200	15.26
FF	2115	94400	2.24
IO	300	300	100.00
BUFG	1	32	3.13

Simple pipeline (no forwarding path, no stalls, no branch bubble)



Resource	Utilization	Available	Utilization...
LUT	6088	47200	12.90
LUTRAM	7	19000	0.04
FF	2707	94400	2.87
IO	300	300	100.00
BUFG	2	32	6.25

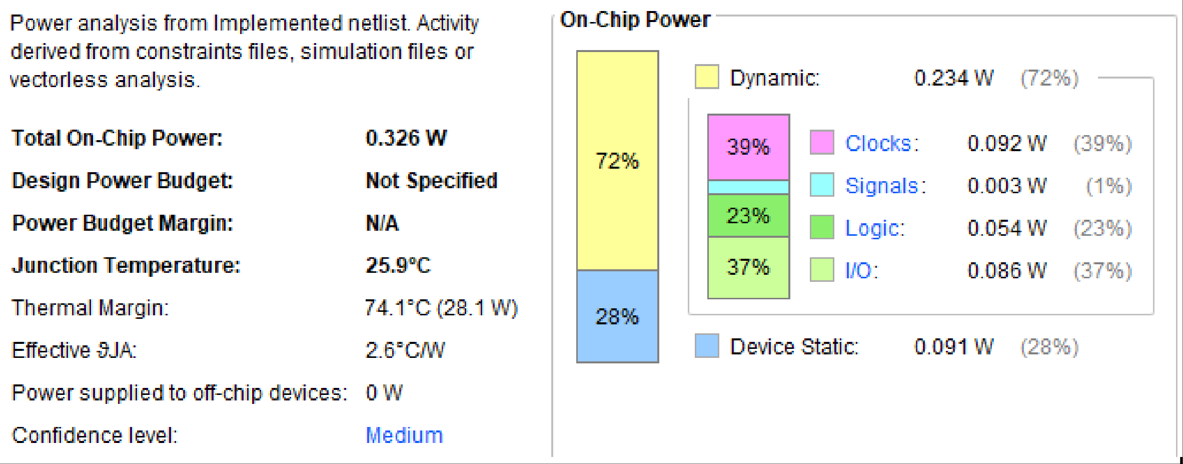
Completed pipeline Arm Education Core



Resource	Utilization	Available	Utilization...
LUT	7138	47200	15.12
LUTRAM	2	19000	0.01
FF	2736	94400	2.90
IO	300	300	100.00
BUFG	2	32	6.25

Post-implementation Power report

Single cycle



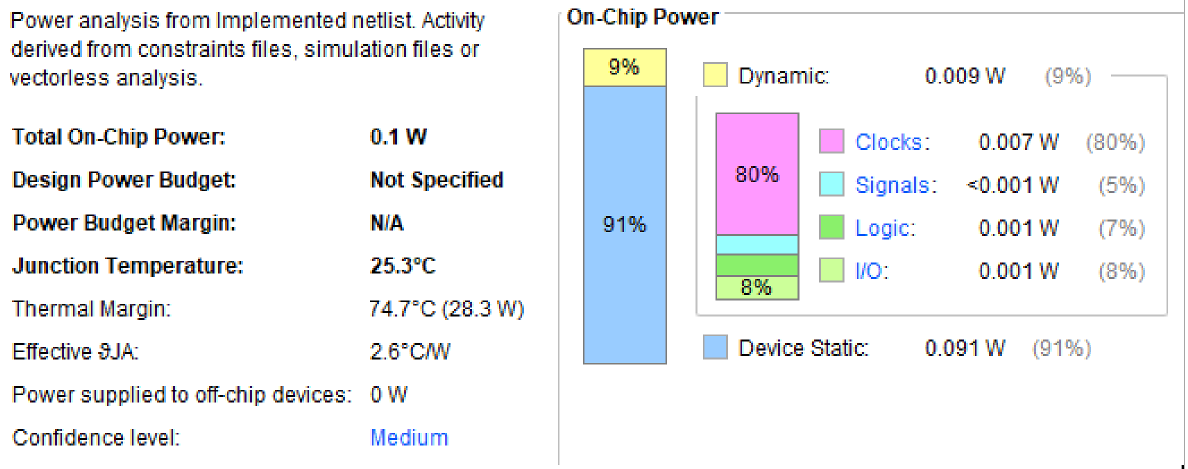
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.326 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.9°C**
Thermal Margin:	74.1°C (28.1 W)
Effective ϑJA:	2.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

On-Chip Power

🟨	Dynamic:	0.234 W	(72%)
🟪	Clocks:	0.092 W	(39%)
🟦	Signals:	0.003 W	(1%)
🟩	Logic:	0.054 W	(23%)
🟩	I/O:	0.086 W	(37%)
🟦	Device Static:	0.091 W	(28%)

Simple pipeline (no forwarding path, no stalls, no branch bubble)



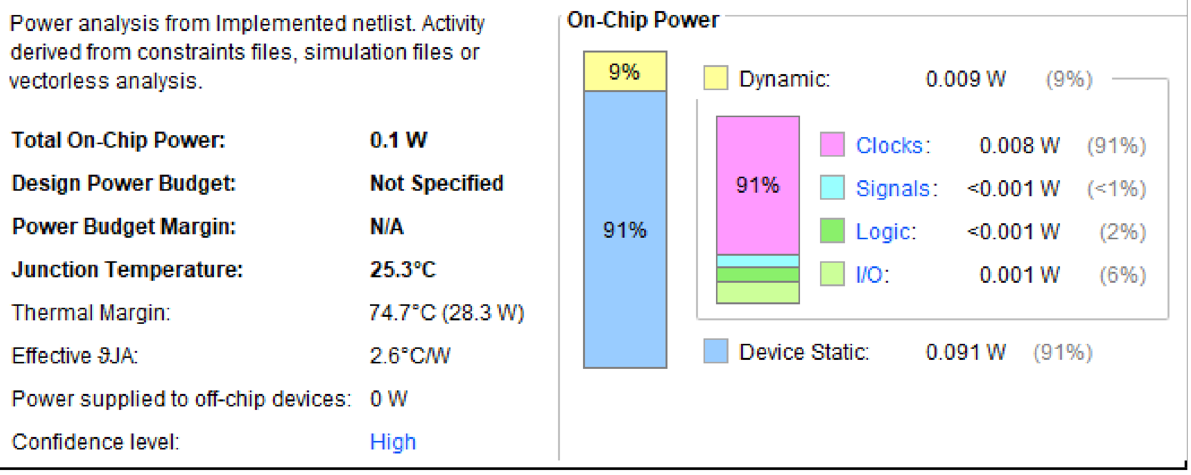
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.1 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.3°C**
Thermal Margin:	74.7°C (28.3 W)
Effective ϑJA:	2.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

On-Chip Power

🟨	Dynamic:	0.009 W	(9%)
🟪	Clocks:	0.007 W	(80%)
🟦	Signals:	<0.001 W	(5%)
🟩	Logic:	0.001 W	(7%)
🟩	I/O:	0.001 W	(8%)
🟦	Device Static:	0.091 W	(91%)

Completed pipeline Arm Education Core



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

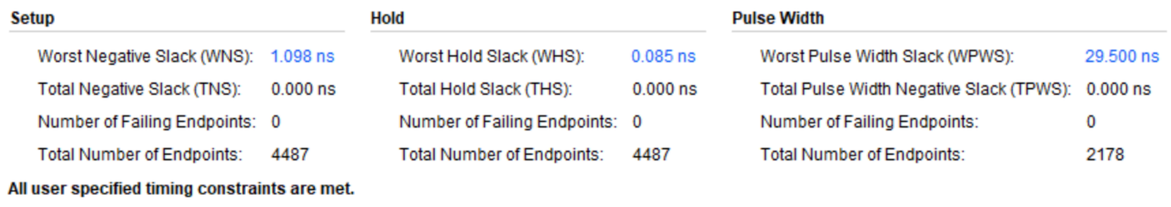
Total On-Chip Power:	**0.1 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.3°C**
Thermal Margin:	74.7°C (28.3 W)
Effective ϑJA:	2.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	High

On-Chip Power

🟨	Dynamic:	0.009 W	(9%)
🟪	Clocks:	0.008 W	(91%)
🟦	Signals:	<0.001 W	(<1%)
🟩	Logic:	<0.001 W	(2%)
🟩	I/O:	0.001 W	(6%)
🟦	Device Static:	0.091 W	(91%)

Post-implementation Timing report

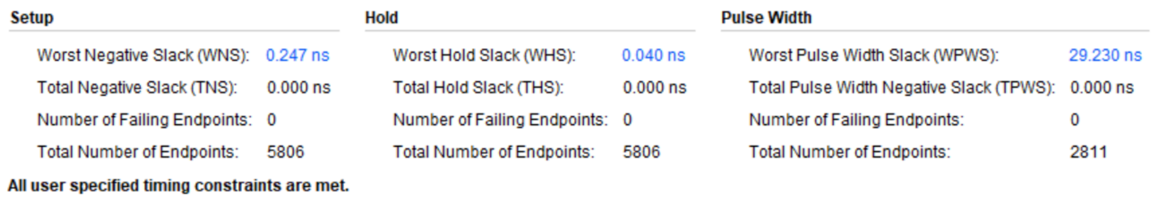
Single cycle



Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1.098 ns	Worst Hold Slack (WHS):	0.085 ns	Worst Pulse Width Slack (WPWS):	29.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	4487	Total Number of Endpoints:	4487	Total Number of Endpoints:	2178

All user specified timing constraints are met.

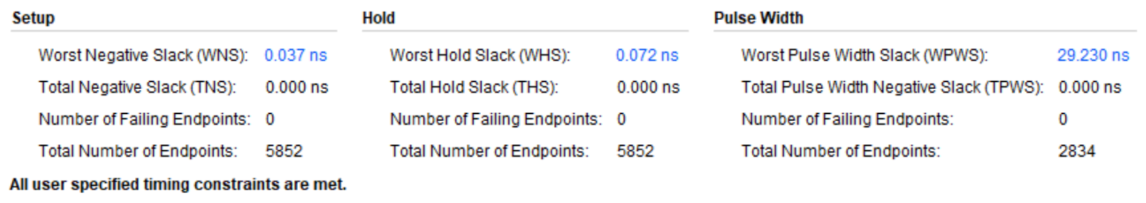
Simple pipeline (no forwarding path, no stalls, no branch bubble)



Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.247 ns	Worst Hold Slack (WHS):	0.040 ns	Worst Pulse Width Slack (WPWS):	29.230 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	5806	Total Number of Endpoints:	5806	Total Number of Endpoints:	2811

All user specified timing constraints are met.

Completed pipeline Arm Education Core



Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.037 ns	Worst Hold Slack (WHS):	0.072 ns	Worst Pulse Width Slack (WPWS):	29.230 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	5852	Total Number of Endpoints:	5852	Total Number of Endpoints:	2834

All user specified timing constraints are met.

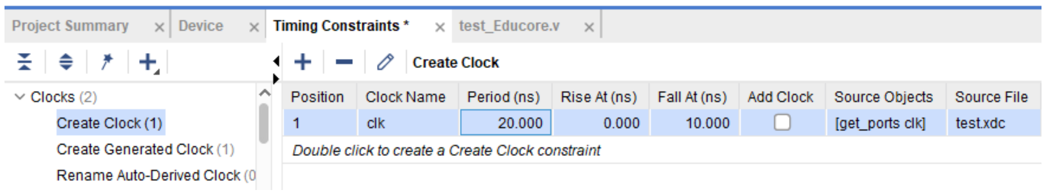
Based on the reports generated and calculated execution time in Average CPI and throughput, answer the following questions:

1. Complete the rankings following the table (where 1 being the best and 3 being the worst) and comment on your observations.

Core type	Performance (in terms of Execution time)	Power	Resource Utilization
Single cycle			Has the least flip-flop usage.
Simple pipeline			Flip-flop utilization in a medium range.
Completed pipeline			Has the most flip-flop usage.

Table 3: Comparison of cores for clk=60 ns (all cores met specified timing constraints at 60 ns clock)

2. In the timing constraint file, the clock period is set to 60 ns. The simple pipeline and completed pipeline cores are also capable of performing at a clock period of 20 ns. However, would the single-cycle core be able to do so as well?
   To answer this question, open the Single-Cycle Vivado project file and expand **Open Implemented Design** and click on **Edit Timing Constraints**. Double-click on clk in the Timing Constraints tab and adjust the clock to 20 ns. Then, click **Apply** at the bottom of the Timing Constraints tab and click on **Run Implementation** to rerun implementation. Click **Save** or **OK** if there is a prompt to save project.



Position	Clock Name	Period (ns)	Rise At (ns)	Fall At (ns)	Add Clock	Source Objects	Source File
1	clk	20.000	0.000	10.000	☐	[get_ports clk]	test.xdc

Double click to create a Create Clock constraint

   Provide a snapshot of the timing summary. Based on the timing summary report results, what can you infer about the single-cycle implementation in terms of clock speed and critical path in comparison to the complete pipelined Arm Education Core?

3. Which implementation do you think has the best compromise and explain why?

9 Further architecture improvements

1. Explain one advantage and disadvantage of the branch bubble approach.

2. What other control hazard mechanisms could be employed on Arm Education Core that could increase performance?

3. In any computer system, access to the main memory takes time, which can impact performance. Discuss what other improvements can be made on Arm Education Core to increase performance. For example, how would a cache or Instruction-Level Parallelism help boost performance?

10 Summary

Throughout the labs in this course, we have evolved Arm Education Core from a single-cycle implementation to a pipelined design, with forwarding paths, stalls, and a branch bubble mechanism. We have seen how such microarchitecture modifications could improve performance and throughput and discussed its potential impact on power and area (utilization of resource) as well. These are just examples of a few basic computer architecture techniques—there are many other techniques, and even today, the computer architecture field is an ongoing development and research field. Power, Performance, and, Area are key factors; however, there are also other design goals that need to be considered, like Security, Verification, Safety/Reliability (like fault-handling), and cost.