arm Education

Computer Architecture Course

LAB 3 (Part B)

Execution, Memory Access, and WriteBack

(with a single-cycle Arm Education Core)

Issue 1.0

Contents

1 Introduction

1.1 Lab overview

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

- Outline the Arm Education Core datapath for the Execution, Memory Access, and WriteBack stages.
- Identify the key components, control signals, and logic in the Execution, Memory Access, and WriteBack stages of Arm Education Core.
- Demonstrate the operation of the Barrel Shifter, Immediate Decoder, and ALU modules using simulation.
- Demonstrate Memory Access operations during load and store instructions using simulation.
- Identify the control signals and outputs of the WriteBack stage.

Note: This lab aims to give learners sufficient familiarization with Arm Education Core's microarchitecture so that learners have adequate knowledge to be able to implement RTL changes in the future labs. This lab will not focus on the detailed RTL design aspect of each module instantiated in Arm Education Core. For learning purposes, most instantiated modules are treated as black-boxes, and it suffices to know what these modules do at a high level.

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
- Introduction to Arm Education Core documentation

This lab requires lab_files.zip provided with Lab 3. (The Arm Education Core provided in lab_files.zip is similar to the one used in Lab 1 and Lab 2).

3 Recap: Single-cycle Arm Education Core datapath

The Arm Education Core provided with this lab is a single-cycle processor. The following diagram shows a simplified overview of the single-cycle Arm Education Core datapath. Note that not all the control signals and relevant combination logic are shown in this diagram, and this will be more apparent as we take a closer look at each processing stage. Lab 3A has covered exercises related to the IF and ID stages.

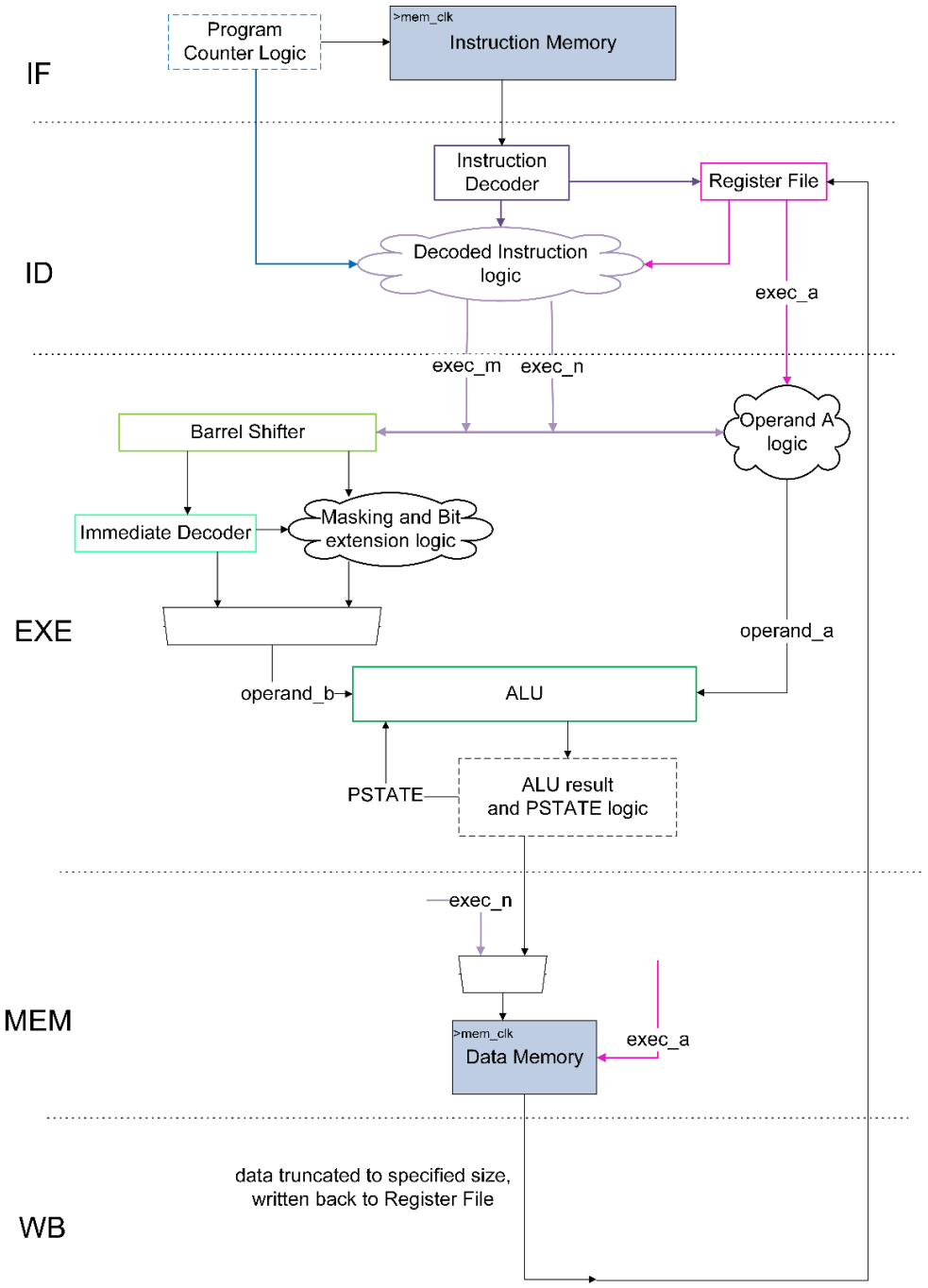


Figure 1: Simplified datapath of single-cycle Arm Education Core

4 Recap: Simulating Lab 3 instructions

The exercises in Lab 3A and Lab 3B require the simulation of specific assembly instructions. Follow these steps:

1. In your working directory, create a folder called **Lab_3** (**C:\Workspace\Lab_3**).

2. In **Lab_3** folder, unzip the folder lab_files.zip. This folder contains **test_Lab3.S** and **Educore-SingleCycle** folders.

3. Simulate test_Lab3.S on Arm Education Core by running the following commands in your Windows terminal:

```
cd Lab_3/lab_files

aarch64-none-elf-gcc -nostdlib -nodefaultlibs -lgcc -gdwarf-4
-Wa,-march=armv8-a -Wl,-Ttext=0x0 -Wl,-N -o test_Lab3.elf
test_Lab3.S

aarch64-none-elf-objcopy -O verilog test_Lab3.elf test_Lab3.mem

cd Educore-SingleCycle

iverilog -Wall -Wno-timescale -Wno-implicit-dimensions -I head/ -t
vvp -y src/ -s test_Educore src/* tests/* -o test_Educore.vvp

vvp test_Educore.vvp -lx2 +TEST_CASE=../test_Lab3.mem
```

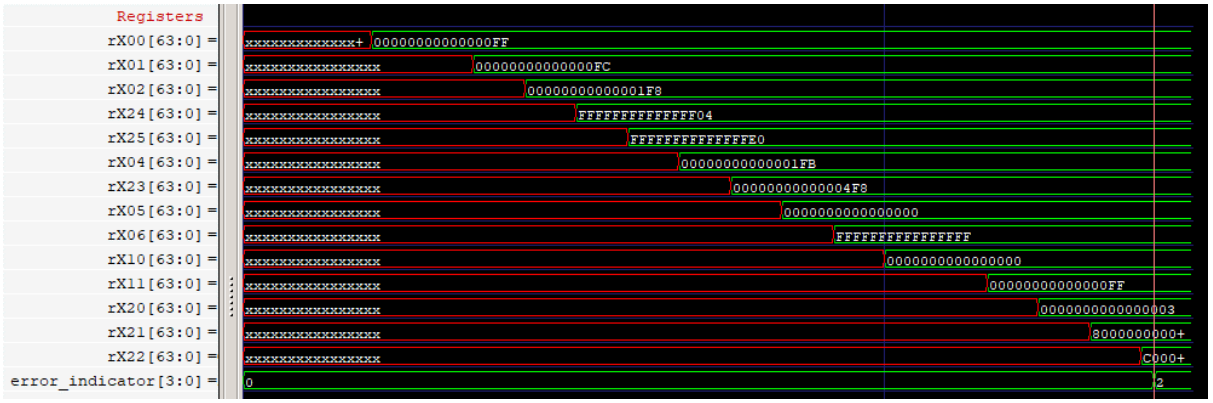
   You should receive the following message in your terminal:

```
[EDUCORE LOG]: Test case: ../test_Lab3.mem
LXT2 info: dumpfile dump.lx2 opened for output.
[EDUCORE LOG]: Apollo has landed
```

4. Run GTKWave to view the waveforms simulated with the provided filter file:

```
gtkwave dump.lx2 debug.gtkw
```

5. Compare the instructions in test_Lab3.S and ensure that the simulated waveforms are as expected, as shown in:



5 Execution stage

In Arm Education Core's Execution stage (EXE), the instruction is executed, and the results are calculated. Arm Education Core's EXE stage contains the largest collection of control signals.

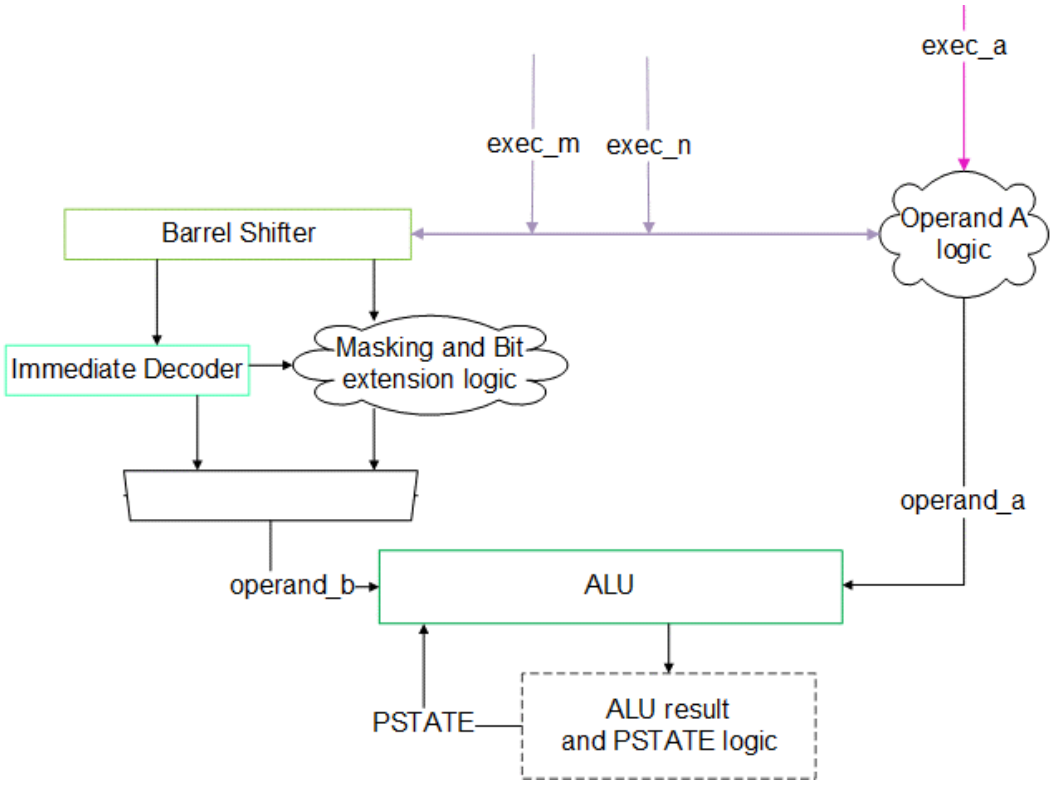


Figure 2: Arm Education Core EXE stage

In Figure 2, **exec_m** and **exec_n** comes from the Decoded Instruction Logic, which we covered in Lab 3A. Arm Education Core's EXE stage consists of 3 modules and 3 logics:

Component	Purpose

Operand A logic	Chooses the source of **operand**_a for the ALU module. Depending on the instruction, the source could either be **exec_n** (from Decoded Instruction Logic), **exec_a** (from Register File), or set to 0.
Barrel Shifter module	Implements bit rotations, logical shifts, and arithmetic shifts. Examples of instructions that require this are the LSL, LSR, ROR, and ASR instructions. Produces the right-shift amount and the result of the shifted operation.
Immediate Decoder module	For decoding logical instructions immediate values. Receives the right-shift amount from the Barrel Shifter. Produces two outputs based on **N**, **imms**, and **immr** inputs: **wmask** and **tmask**.
Masking and Bit extension logic	In Arm Education Core, the outputs from the immediate decoder are obtained and the appropriate bit masking or bit extension is performed when necessary. Depending on the instruction type, **operand_b** is then chosen to be either the Immediate Decoder value or the bit extended value.
ALU module	Conducts 3 logical operations (AND, OR, EOR) and 1 arithmetic operation (ADD) on 2 operands (**operand_a** and **operand_b**). Outputs NZCV flags.
ALU result and PSTATE logic	• Checks the PSTATE condition based on the ALU result. The PSTATE register stores the NZCV flags, and only certain instructions like ADDS, SUBS, ANDS, CCMP, and CCMN can update the new PSTATE value. Otherwise, it is updated with the value supplied within a field in the instruction. The PSTATE's Carry flag is fed back to the ALU. • Contains multiplexers to select which ALU result to move forward to the next processing stage (i.e., MEM).

In the next exercises, we will observe the behavior of these modules and logics using the simulation in Recap: Simulating Lab 3 instructions.

5.1 Exercise: Barrel Shifter module

The Barrel Shifter implements bit rotations, logical shifts, and arithmetic shifts.

The following diagram shows Arm Education Core's Barrel Shifter inputs and outputs. Depending on the instruction type, the data source for **barrel_in** and **upper_in** can either be **exec_n** or **exec_m** from the Decoded Instruction Logic.

Figure 3: Barrel Shifter I/Os

In this exercise, we will demonstrate the function of the Barrel Shifter module by observing the module I/Os for:

- A bit rotation/shift instruction (**LSL X2, X1, #1**). The Logical Shift Left (immediate), also known as LSL (immediate) instruction, shifts a register value left by an immediate number of bits, shifting in zeros, and writes to the destination register. So, for example, a Logical Shift Left of value 2'b11 would be 3'b110. Note that LSL is also an alias for the Unsigned Bitfield Move (UBFM) instruction. For more information, see LSL instruction in Additional Reading.
- An add (shifted register) instruction (**ADD X23, X1, X0, LSL #2**). The shift is done on X0 and added to X1.

Based on the simulation results in Task: Simulating Lab 3 instructions, answer the following questions:

1. Fill in the signal values for the following instructions at the time when the correct results are available in their respective destination register:

Signal	Description	LSL X2, X1, #1	ADD X23, X1, X0, LSL #2
Correct destination register value		X2 =0000_0000_0000_01F8	X23 =0000_0000_0000_04F8
First operand register value		X1 = 0xFC	X1 = 0xFC
Second operand register value		-	X0 = 0xFF
shamt	Shift amount		
FnH	1–64 bits	1	1

	0–32 bits		
barrel_op	Specifies type of barrel shift operation	11	00
barrel_in	Value to be shifted, obtained from decoded instruction logic.		
upper_in	Value to be shifted, obtained from decoded instruction logic. Usually the same value as barrel_in, except for ROR or EXTR instructions.	0xFC	0xFF
right_shamt	Equivalent right shift amount		
barrel_out	Result after shift		

2. What module drives the **shamt**, **fnK**, and **barrel_op** signals?

3. Which processing stage and combi logic drives the **barrel_in** and **upper_in** signals?

4. **ADD X3, X1, X0, LSL #2** specifies a shift to the left by 2 bits for register X0, and this left shift amount is seen in signal **shamt**. What is the relationship between the right shift amount, register size, and signal **shamt**?

At this point, you may be wondering why the **shamt** value for instruction `LSL X2, X1, #1` is not 1. If you look up LSL in Base Instructions (alphabetical order), you will notice the following description:

64-bit (sf == 1 && N == 1 && imms != 111111)

LSL <Xd>, <Xn>, #<shift>

is equivalent to

UBFM <Xd>, <Xn>, #(-<shift> MOD 64), #(63-<shift>)

and is the preferred disassembly when imms + 1 == immr.

LSL is an alias of UBFM. The syntax for a UBFM instruction is `UBFM <Xd>, <Xn>, #<immr>, #<imms>`. In this case, `<immr>` = `-<shift MOD64>` = -(1) = 63 in 2s complement (0x3F). In other words, the LSL instruction gets assembled in a way where the left shift amount is already represented in a right shift amount that gets passed on to **shamt**. The Arm Education Core's barrel shifter recognizes such behavior.

5.2 Exercise: Immediate Decoder module

Arm Education Core's Immediate Decoder module decodes the **N**, **imms**, and **immr** bits that are encoded in some instructions, such as:

- Logical instructions with immediate values, e.g., AND X1, X0, #0x00003ffc00003ffc
- Bitfield move instructions, e.g., LSL X2, X1, #1.

The Arm Education Core's Immediate Decoder generates 2 masks (**wmask** and **tmask**) that can be used for various operations. In the Armv8-A ISA, **wmask** and **tmask** are used together or individually for the operation in various instructions. For example, for logical instructions with immediate values, **wmask** is the decoded immediate value. Some bitfield move instructions use both **wmask** and **tmask** to produce its results. In the Arm Education core, these masks are also used in other combination logic, which we will see in Exercise: Operand Logic and ALU module.

The following diagram shows Arm Education Core's Immediate Decoder inputs and outputs.

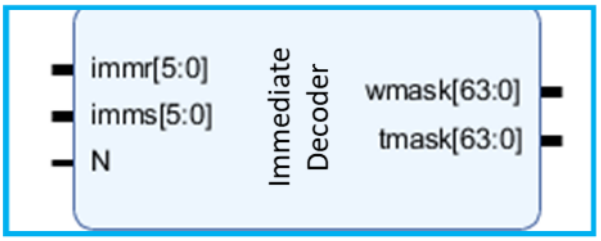


Figure 4: Immediate Decoder I/Os

Recap from previous lab:

- Logical and bitfield instructions encode immediate values using the N, imms, and immr bits. An element (sub-pattern of bits) is replicated across the register width.

N	imms[5:0]						Element size (bits)
0	1	1	1	1	0	X	2
0	1	1	1	0	X	X	4
0	1	1	0	X	X	X	8
0	1	0	X	X	X	X	16
0	0	X	X	X	X	X	32
1	X	X	X	X	X	X	64

- The immr bits just specify the number of right-rotations for the element, it can support up to 63 rotations (for 64-bit element).

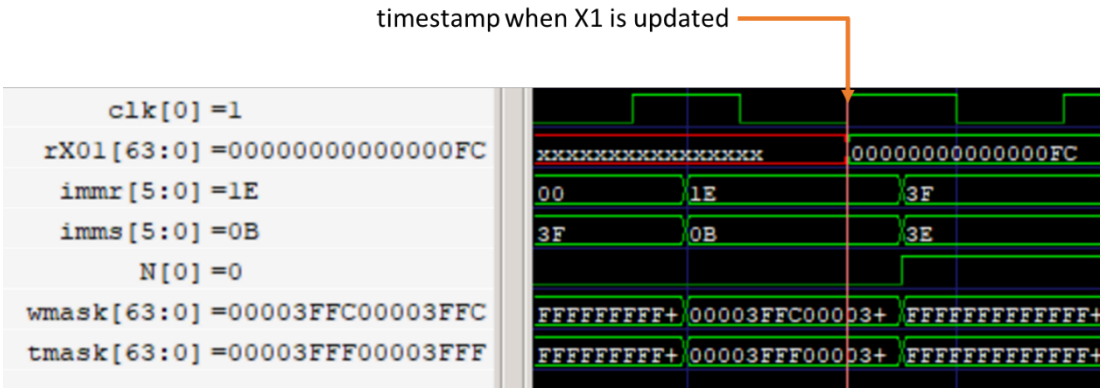
Logical immediate encoding example:

N	imms [5:0]	immr [5:0]	Element size	Number of 1s	Number of right rotation	Resulting element	Resulting 64-bit value
0	0xB	0x1E	32	12	30	0x0000_3ffc	0x00003ffc00003ffc

63	31 0
0x0000_3ffc	0x0000_3ffc

Based on the simulation results in Recap: Simulating Lab 3 instructions, let's observe the behavior of the Immediate Decoder module when certain instructions are executed by answering the following questions.

1. The following diagram shows the respective signals when register X1 is updated when the `AND X1, X0, #0x00003ffc00003ffc` is executed.



a. Which module drives the **N** and **imms** signals?

b. Which module in Arm Education Core drives the **immr** signal?

c. What is the value of **wmask**? Does this correspond to the expected theoretical result shown in the example in the previous page?

5.3 Exercise: Masking and Bit Extension Logic

In the **Educore** module, there are combinational logics for doing the appropriate masking and bit extension for:

- Bitfield instructions (BFM, SBFM, UBFM) and MOV instructions. Note that LSL is also an alias for the UBFM instruction.
- Add/Subtract shifted register instructions.
- Add/Subtract extended instructions.

Most of these advanced instructions are beyond the scope of this lab. Hence, for the purpose of this exercise, it is therefore sufficient to just understand what this combinational logic outputs (bitext_out), and we will observe the associated signal values when the ASR instruction is executed.

Based on the simulation results in Task: Simulating Lab 3 instructions, let's observe the behavior of the ALU modules and associated logic when certain instructions are executed by answering the following questions.

1. Consider the following ASR instruction (alias of SBFM instruction):

```
ASR   X25, X24, #3 // same as SBFM    X25, X24, #3, #63
```

The Arithmetic Shift Right (immediate), ASR instruction shifts a register value right by an immediate number of bits and shifts in copies of the sign bit in the upper bits. The ASR instruction is an alias of the Signed Bitfied Move (SBFM) instruction.

The operation of the ASR instructions is shown in the following pseudocode:

```
bot = RotateRight(X24,immr) AND wmask
top = ReplicateSign(X24)
X25 = (top AND NOT(tmask)) OR (bot AND tmask)
```

where

- **X25** and **X24** are the registers
- **wmask** and **tmask** are outputs from the Immediate Decoder.
- **immr** is the number of rotations
- **RotateRight()** is a function that rotates the bits to the right according to **immr**. For example: **RotateRight(01101, 1) = 10110**
- **ReplicateSign()** is a function that sign extends the sign bit across the register size. In this case, X24 has a 2's complement value and the sign bit is 1. Therefore, top is a 64-bit register of 1s.

a. When the ASR instruction is executed, register X24 has the value 252 (2's complement shown below) and the **wmask** and **tmask** values are also shown below.

X24	0xFFFF_FFFF_FFFF_ FF04
immr	0x3
wmask	0xFFFF_FFFF_FFFF_FFFF
tmask	0x1FFF_FFFF_FFFF_FFFF

Based on the above pseudocode,

i. What is the value of `RotateRight(X24,immr)`?

ii. What is the value of `bot AND tmask`?

iii. What is the value of `top AND NOT(tmask)`?

iv. The value of (top AND NOT(tmask)) OR (bot AND tmask)? Does this match the value of X25 in the simulation?

b. Which module in **Educore.v** does the shift rotation? (**Hint**: Does barrel_out signal have the same value observed in `RotateRight(X1,immr)`? Check the waveforms you simulated from Recap: Simulating Lab 3 instructions)

c. Is the value of (`temp AND tmask`) equals **bitext_out**?

At this point, you may be wondering what is the purpose of **wmask** if it's just all 1s in this case. Although in this case of the ASR instruction the purpose of **wmask** is not as apparent, the SBFM instruction is, in fact, a base instruction to several aliases such as LSL, LSR, and other instructions that are out of scope of this lab (SBFX, SXTH).

5.4 Exercise: Operand Logic and ALU module

Operand Logic

The Operand Logic chooses the source for Operand A and Operand B, which feeds into the ALU module.

- The ALU Operand A is sourced from **exec_n**, **exec_a**, or cleared.
- The ALU Operand B from the bit extension logic or the **wmask** from the immediate decoder.

ALU module

Arm Education Core's ALU module has the following inputs and outputs:

Figure 5: ALU module I/Os

The ALU takes operand A and operand B and carries out its operation depending on the **cmd** option. See ALU I/Os for signal descriptions. The ALU then outputs its result and corresponding N, Z, C, V flag. The carry flag (C, which is PSTATE[1]) is fed to the ALU module via the **carry_in** input.

Answer the following questions:

1. The **wtmask[0]** signal is a control signal from the Instruction Decoder module. When **wtmask=1**, it applies bit clearing on ALU operand A source and bit masking on operand B source before ALU operation.
   The **wtmask** signal is set to high for Move Immediate instructions and Bitfield Immediate instructions.

   For the `MOVZ X0, #0xff` instruction, fill in the following signal values at the time when the correct results are available in the destination register:

Instruction decoder	wtmask	

Masking & Bit Extension Logic	bitext_out	
Operand Logic	alu_op_a	
	alu_op_b	
ALU module	alu_out	
	alu_flags (NZCV)	

2. Provide a snapshot showing the **operand_a**, **operand_b** signals, and ALU I/Os when the `ADD X4, X1, X0` instruction is being executed. Ensure that the **alu_out** shows the expected results of X4.

5.5 Exercise: ALU result and PSTATE logic

Arm Education Core's ALU result logic produces an output (**ex_out**) to be passed on to the next processing stage (MEM or WB).

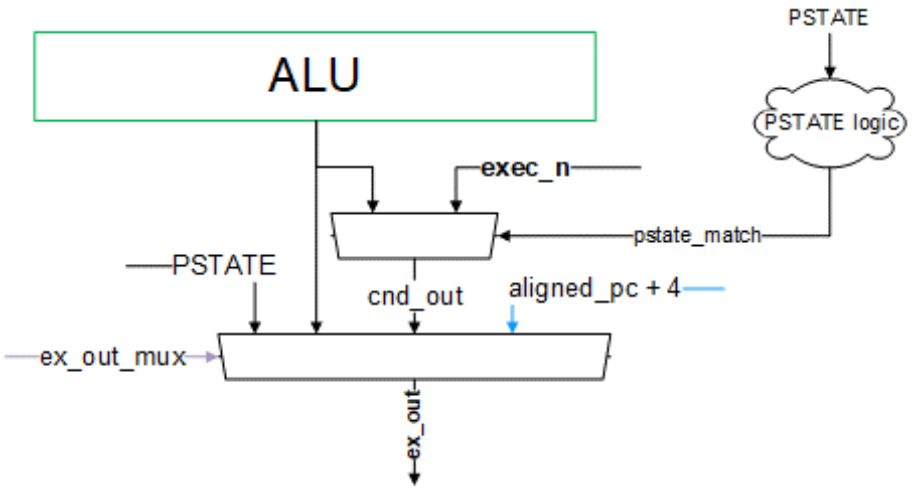


Figure 6: Overview of ALU and PSTATE logic

PSTATE stands for Processor State. The register fields in a traditional *Current Processor Status Register* (CPSR) in Armv7-A are referred to collectively as the PSTATE in Armv8-A (AArch64). In Arm Education Core, the PSTATE fields only include the NZCV flags.

The PSTATE logic selects the source for PSTATE and checks for the condition of PSTATE.

The output to the execution stage could be produced from:

- Value of PSTATE signal (Flags NZCV, signal **ctrl_out** in **Educore.v**)
    - PSTATE[3] — N flag
    - PSTATE[2] — Z flag
    - PSTATE[1] — C flag. This flag is fed to the ALU module via the **carry_in** ALU input.
    - PSTATE[0] — V flag
- Value of ALU result (signal **alu_out**)
- Deferred Conditional result (see CSEL instruction and the like—these instructions are also supported by Arm Education Core but is beyond the scope of this course)
- Value of PC+4 (The address of the next instruction, signal **aligned_pc**+4)

The signal **ex_out** is the ultimate output of the execution stage that gets passed on to the MEM processing stage.

Based on the simulation results in Task: Simulating Lab 3 instructions, answer the following question:

1. When `SUBS X6, X5, #0x1` instruction is executed, what is the PSTATE value?

2. Which flag (NZCV) is being asserted in `SUBS X6, X5, #0x1` and why?

6 Memory access stage

The Memory access (MEM) stage is accessed for Data Transfer-type instructions such as load and store instructions. The following code snippet shows the Verilog code for the Memory Access stage:

```
// Memory stage
assign data_memory_out_v = rt_value;
assign data_memory_a = (mem_addr_mux == `MEM_ADDR_RN)? rn_value : ex_out;
assign data_memory_s = mem_size;
assign data_memory_read  = mem_read;
assign data_memory_write = mem_write;
```

The following table shows the signal descriptions that are also the **Educore** module I/Os:

Signal	Educore I/O	Description
data_memory_out_v	Output	Data memory write value.
data_memory_a	Output	Data memory address.
data_memory_s	Output	Data memory read/write size.
data_memory_read	Output	Data memory read control.
data_memory_write	Output	Data memory write control.
data_memory_in_v	Input	Data memory read value.

The data memory address to read from or write to depends on the type of instruction—the address could be specified in the instruction itself (**rn_value**), or it could be the result of **ex_out**.

Do remember that the memory in the testbench file reads and writes at the positive edge of **mem_clk**, which is at a higher frequency than **core_clk**.

6.1 Exercise

Answer the following questions:

1. By referring to the code in **Educore.v**:
    a. What module drives the signal **mem_size**, **mem_read**, **mem_write**, and **mem_addr_mux**?

    b. What does the signal **rt_value** contain?

2. In Recap: Simulating Lab 3 instructions, the following code was simulated

```
// Store Load instructions
MOVZ   X10, #0
STUR   X0, [X10]
LDUR   X11, [X10]
```

Note: Arm Education Core only supports a limited subset of the Armv8-A ISA. It does not support the LDR and STR register instruction, and therefore as a replacement, we have used the STUR and LDUR instructions, which would give the same outcome.

Based on the waveform simulation you have done in Recap: Simulating Lab 3 instructions:

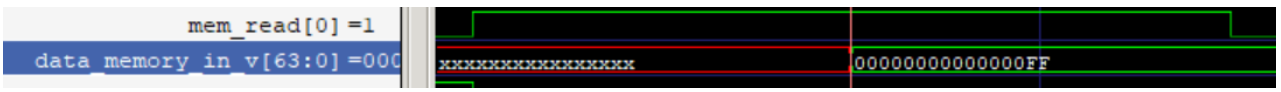
a. What is the result of X11 (register rX0B in simulation)?

b. **mem_read** is asserted to 1 during the execution of which instruction?

c. **mem_write** is asserted to 1 during the execution of which instruction?

d. What is the value of **data_memory_out_v** during the STUR instruction?

e. During the LDUR instruction, when mem_read=1, there is a delay for data_memory_in_v is only updated with the correct value (0xFF), as shown in the diagram below. Why is this so?



7 WriteBack stage

In the WriteBack stage, the results obtained from data_memory_in_v are truncated to the specified size before it is written back into the destination General Purpose Register of the Register File module. The following code snippet shows the WriteBack stage. Notice that **mem_size**, **mem_read**, and **data_memory_in_v** are being used in both MEM and WB stages.

```
/*****************************************************************************/
// WriteBack stage
// Choose data to write back
reg [63:0] ld_ext_memory_out;
always @ ( * ) begin
    case (mem_size)
        `LDST_SIZE_08: ld_ext_memory_out = {{56{mem_sign_ext & data_memory_in_v[
7]}}, data_memory_in_v[ 7:0]};
        `LDST_SIZE_16: ld_ext_memory_out = {{48{mem_sign_ext &
data_memory_in_v[15]}}, data_memory_in_v[15:0]};
        `LDST_SIZE_32: ld_ext_memory_out = {{32{mem_sign_ext &
data_memory_in_v[31]}}, data_memory_in_v[31:0]};
        default:       ld_ext_memory_out = data_memory_in_v;
    endcase

    if(~mem_load_FnH)
        ld_ext_memory_out = {{32{1'b0}}, ld_ext_memory_out[31:0]};
end

assign memory_out = mem_read? ld_ext_memory_out : 64'hxxxx_xxxx_xxxx_xxxx;
```

7.1 Exercise

By referring to the code in **Educore.v**, answer the following questions:

1. What module does signal **memory_out** from the WB stage drive?

2. What module drives signal **mem_load_FnH**?

8 Summary

In this lab, we have looked at key components, modules, control signals, and combinational logic in the EXE, MEM, and WB stages.

Having completed the exercises, you would have noticed by now that some signals from the IF and ID stages are also being used in EXE, MEM, and WB stages. Likewise, some signals from the MEM stage are also used in the WB stage. For example:

- The Program Counter value is also used in the EXE stage.
- Many control signals from the Instruction Decoder module go to EXE, MEM, and WB stages.
- **exec_a**, **exec_n**, and **exec_m** get passed on to EXE and MEM stages.

It is good to take note of the above points as we move on to the next lab where we start to implement pipelining in Arm Education Core.

9 Additional reading

ASR instruction

- https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ASR--immediate---Arithmetic-Shift-Right--immediate---an-alias-of-SBFM-?lang=en
- https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/SBFM--Signed-Bitfield-Move-?lang=en

LSL instruction

- https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/LSL--immediate---Logical-Shift-Left--immediate---an-alias-of-UBFM-?lang=en
- https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/UBFM--Unsigned-Bitfield-Move-?lang=en

10 Appendix

10.1 ALU I/Os

Input Signals

Signal	Size (bits)	Description
operand_a	64	The first of the two operands to the ALU.
operand_b	64	The second of the two operands to the ALU.
carry_in	1	The carry flag from the PSTATE register to allow “BIG” arithmetic (>64 bits).
invert_b	1	ACTIVE HI—Bit invert operand_b before applying the operation.
FnH	1	ACTIVE HI—Indicate whether the operation, result & operands are 64-bit width, otherwise 32-bit.
cmd	3	Specifies the operation carried out on the two operands: • bit-wise AND • bit-wise OR • bit-wise EOR • arithmetic ADD with carry clear • arithmetic ADD with carry set • arithmetic ADD with carry matching carry_in

Output Signals

Signal	Size (bits)	Description
result	64	Result of the operation carried out on the operands A & B
flags_nzcv	4	Extra bits indicating certain properties about the result. The flags reflect the specified size by FnH. Bit [3]: N — MSB is set

		Bit [2]: Z—Result bits from MSB to bit 0 are all zeros Bit [1]: C—If set, carry out occurred at MSB Bit [0]: V—Overflow occurred on arithmetic operation Msb is 63 if FnH is set, otherwise it is bit 31.