arm Education

Computer Architecture Course

LAB 3 (Part A)

Instruction Fetch and Decode

(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:

- Identify the top-level I/O signals and modules in the Arm Education Core.
- Outline the Arm Education Core datapath for the Instruction Fetch and Instruction Decode stages.
- Identify the key components, control signals, and logic in the Instruction Fetch and Instruction Decode stages of Arm Education Core.
- Demonstrate the behavior of the Program Counter including during a branch instruction using simulation.
- Outline the purpose of the Instruction Decoder module including how some of its outputs are used as control signals or passed to other key components in the Arm Education Core.
- Outline and demonstrate how register values are read from and written to the Register File module using simulation.

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.

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

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).

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

3 Single-cycle core overview

The Arm Education Core provided with this lab is a single-cycle processor. It is *one* processor implementation of a subset of Armv8-A instructions. It was developed solely for demonstration and education purposes, with limited functionality.

A single-cycle processor is a processor that processes its instruction in a single clock cycle.

3.1 Processing stages of single-cycle Arm Education Core

The single-cycle Arm Education Core implements 5 stages of processing **in a single clock cycle**, which are:

- **IF—**Instruction is fetched.
- **ID**—Instruction is decoded.
- **EXE**—Instruction is executed, and the result is calculated.
- **MEM**—Memory is accessed for Data Transfer-type instructions (Load/Store)
- **WB**—Results obtained are Written Back into the General-Purpose Registers in the Register File module.

3.2 Types of instructions supported

Arm Education Core supports a subset of Armv8-A Instructions that have the following type:

- Data processing instructions (immediate or register). Some instructions are capable of PC-relative addressing, or shift/extend register.
- Load/stores, including literal, unscaled, post-index, and pre-index types.
- Branches including conditional branch, unconditional branch.
- System register move instructions like **MRS** and **MSR**.
- NOP and YIELD.

For more information on supported instructions, see the Introduction to Arm Education Core documentation.

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

3.3 Single-cycle Arm Education Core datapath

The following diagram shows a simplified overview of the single-cycle Arm Education Core datapath. In this lab, we will go through each stage and investigate the characteristics and operations of key modules and logic. 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.

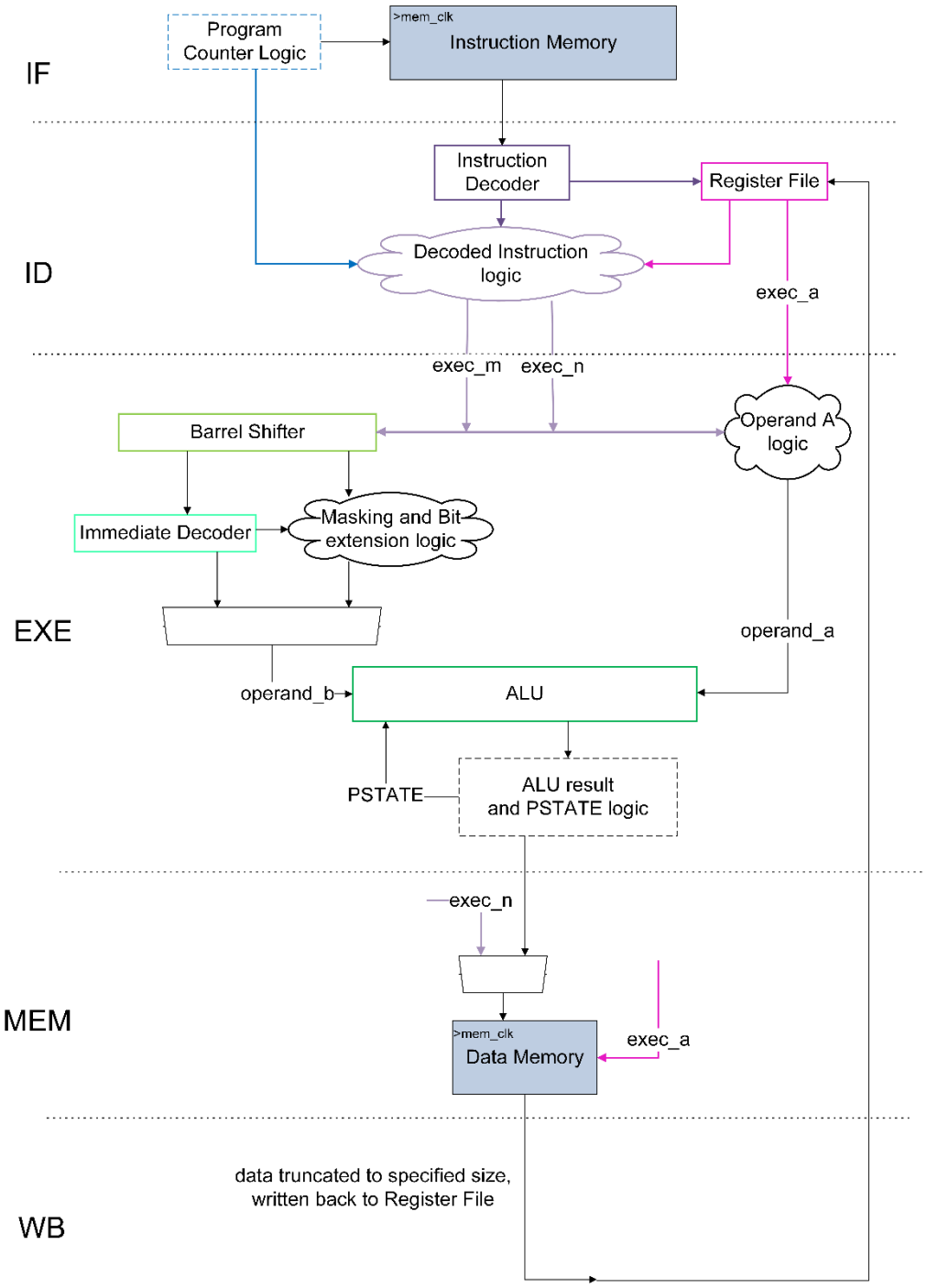


Figure 1: Simplified datapath of single-cycle Arm Education Core

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

4 Task: 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** folder.

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:

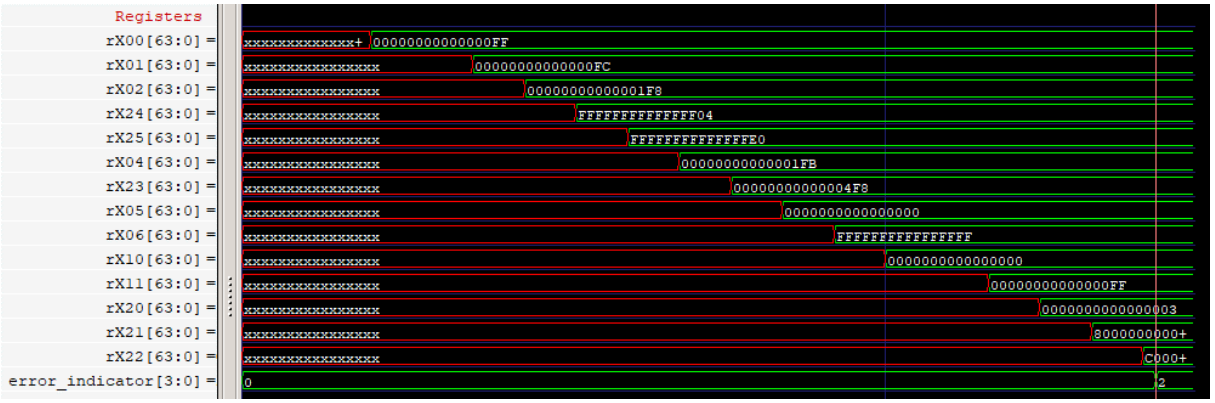
```
[EDUCORE LOG]: Test case: ../test_Lab3.mem
LXT2 info: dumpfile dump.lx2 opened for output.
[EDUCORE LOG]: Apollo has landed
```

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 your simulated waveforms values are as expected, as shown in:

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.



5 Arm Education Core top-level module

Educore.v is the top-level file of Arm Education Core. The following modules are instantiated in **Educore.v**:

- Instruction Decoder
- Register File
- Barrel Shifter
- Immediate Decoder
- Arithmetic Logic Unit
- All other logic around the modules as shown in Figure 1.

5.1 Exercise: Familiarizing with Educore.v

Open the **Educore-SingleCycle/src/Educore.v** file and familiarize with it (if needed, you may use the simulated waveforms in Task: Simulating Lab 3 instructions to help you understand better). Then, answer the following questions:

1. The following table shows the I/O signal description of the top-level module **Educore**. Complete the following table.

Signal type	Signal width (bit)	Description	I/O Name
Input	1	Global core clock	
Input	1	Clock enable	
Input	1	Active-LOW reset	
Input	32	Value read from Instruction Memory (the instruction itself)	

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

Input	64	Value read from Data Memory	
Output	4	Error indicator to the testbench. We use this to control when to stop simulation or display on command-line if there is an error.	error_indicator
Output	2	Data memory read/write size. This depends on the instruction type—e.g., 8 bits, 16 bits, 32 or 64 bits.	data_memory_s
Output	1	Control signal for reading Data Memory	data_memory_read
Output	1	Control signal for writing Data Memory	data_memory_write
Output	64	Instruction memory address	
Output	1	Instruction memory read enable	
Output	64	Data memory address	
Output	64	Data memory write value	

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

2. The Instruction Memory and Data Memory are not in the **Educore** module itself. State where these memories are located when running simulation in Task: Simulating Lab 3 instructions.

3. The following table lists the modules that are instantiated in **Educore.v**. State whether the modules receive a clock input.

Module	Clock input? Yes/No
InstructionDecoder	
RegisterFile	
BarrelShifter	
ImmediateDecoder	
ArithmeticLogicUnit	

4. The **npc_stop** active-LOW signal is a master enable internal signal for the **Educore** module. This signal stops/halts **Educore** when it is set to 0. In **Educore.v**, what causes **npc_stop** to be assigned to 0?

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

6 Instruction Fetch stage

A key component in fetching instruction is the Program Counter (PC), which is a register that holds the address of the instruction that is to be fetched from Instruction Memory. This happens in the Instruction Fetch (IF) stage.

The *Program Counter Logic* handles what the values in the Program Counter should be, depending on the program flow of sequential or branching execution. The Instruction Decode stage, following the Instruction Fetch, provides key signals to the Program Counter Logic to determine the next proper value. There are some instructions where the PC will depend on the computed values from the Execution stage (EXE), such as the B.NE (Branch Not Equal) instruction. We will take a closer look at the ID and EXE signals later on in this lab.

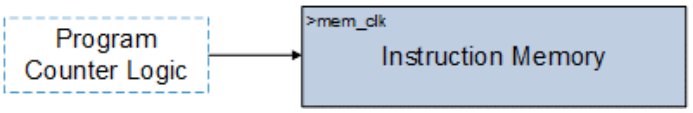


Figure 2: Instruction fetch

The **Program Counter Logic** is in **Educore.v** in the code section that starts with the following header:

```
// ------------------------- Program Counter Logic ------------------------- //
```

The following diagram shows a simplified view of the Program Counter Logic (branch signals not shown to remove complexity).

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

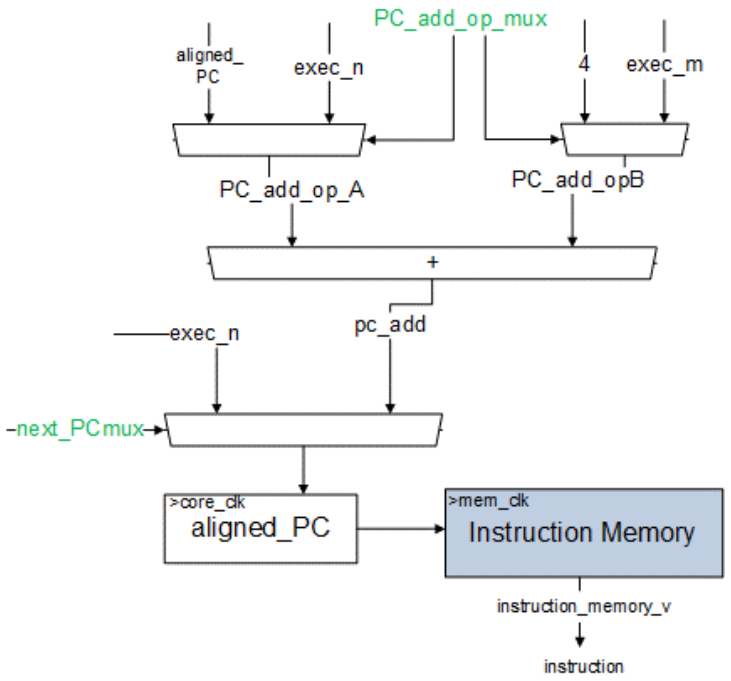


Figure 3: Simplified view of Program Counter Logic (branch signals not shown)

exec_n and **exec_m** in Figure 3 are signals that are determined by decode signals from the ID stage. They correspond to the values of the operands in the instructions, which we will look at further in this lab. The Program Counter is updated by its current value incremented by 4, or by the summation of the PC (held in **exec_n** in the diagram) and the immediate offset value of a branch instruction (held in **exec_m** in the diagram). The Program Counter should also be updated by a register value (held in **exec_n** in the diagram), as in branch register instructions like BR and BLR, although these instructions are beyond the scope of this lab. The signals in green (**PC_add_op_mux, next_PCmux**) in Figure 3 are control signals for the multiplexers. These control signals come from the Instruction Decode module.

6.1 Exercise: Program Counter Logic

Based on the Program Counter Logic Verilog code, answer the following questions:

1. Signal **PC_add_op_mux** comes from the Instruction Decode stage. If **PC_add_op_mux is** 0 (`**PC_OP_NEXT** defined in **Educore-SingleCycle/Head/Educore.vh**), then

   ```
   pc_add = aligned_pc + 4;
   ```

   Why is the Program Counter incremented by the value 4?

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

2. What is the value of the Program Counter at reset?

3. What is the value of the Program Counter when **npc_stop = 0** and **nreset_sync=1** (no reset)?

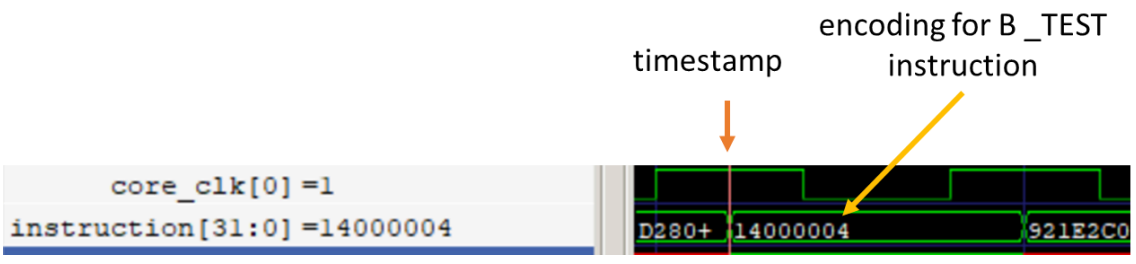
4. What is the difference between the signal **PC** and the signal **aligned_pc**?

5. Signal **br_condition_mux** comes from the Instruction Decoder stage. **`BR_COND_UNCOND** and **`BR_COND_PSTATE** values are defined in the **Educore-SingleCycle/head/Educore.vh** file.

   br_condition_mux is:
   - 1 (BR_COND_UNCOND) when the instruction is an unconditional branch
   - 0 (BR_COND_PSTAGE) when it is a branch instruction that depends on the PSTATE (N,C,Z,V flags) in the EXE stage.

   Give 2 examples of branch instructions that will cause **br_taken = 1**.

6. Based on the simulated waveforms obtained in Task: Simulating Lab 3 instructions, answer the following questions based on the **timestamp** indicated in the following diagram.



Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

a. Complete the table below by filling in the respective values when the **B _TEST** instruction in **test_Lab3.s** is executed?

Entry address of label _TEST	0x14
br_taken	
exec_n	
exec_m	
pc_add	

b. The **aligned_PC** value at this time is 0x4. When does it get updated to 0x14?

7 Instruction Decode stage

The following diagram shows a simplified view of the Instruction Decode (ID) stage.

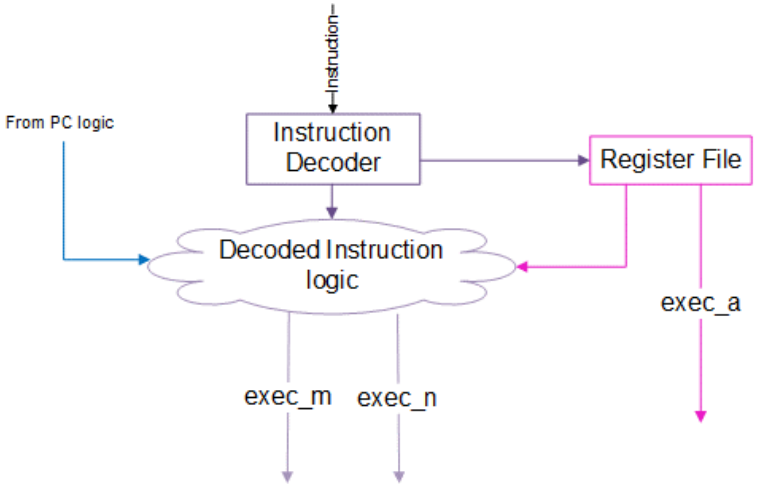


Figure 4: Instruction Decode stage

In the ID stage, the signal **instruction** from the Instruction Memory is passed to the Instruction Decoder module. The instruction is then decoded, and its output signals are passed to various parts of the **Educore** module, to various stages and logics. Arm Education Core's

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

Register File supports 3 read ports, which are the <N>, <M>, and <A> ports. This allows Arm Education Core to be able to read up to 3 registers simultaneously from the Register File, which is useful for instructions that have 3 register operands. For example:

<instruction>	<destination register (Rd)>,	<N>,	<M>
ADD	X0,	X1,	X2

<instruction>	<A>,	<N>
STUR	X0,	[X10]

<instruction>	<return register (Rt)>,	<N>
LDUR	X11,	[X10]

How about instructions that do not have 3 operands or have an immediate value? The Instruction Decoder module outputs **exec_n_mux** and **exec_m_mux** signals that the Decoded Instruction logic uses to select whether **exec_m** should be Xs or immediate value, for example.

Note that the load and store register offset-type instructions have an additional operand that is read at the <M> port. However, these instruction types are beyond the scope of this course lab.

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

7.1 Exercise: Instruction Decoder module

Arm Education Core's Instruction Decoder module has 1 input (instruction) and 38 outputs as shown in the following diagram.

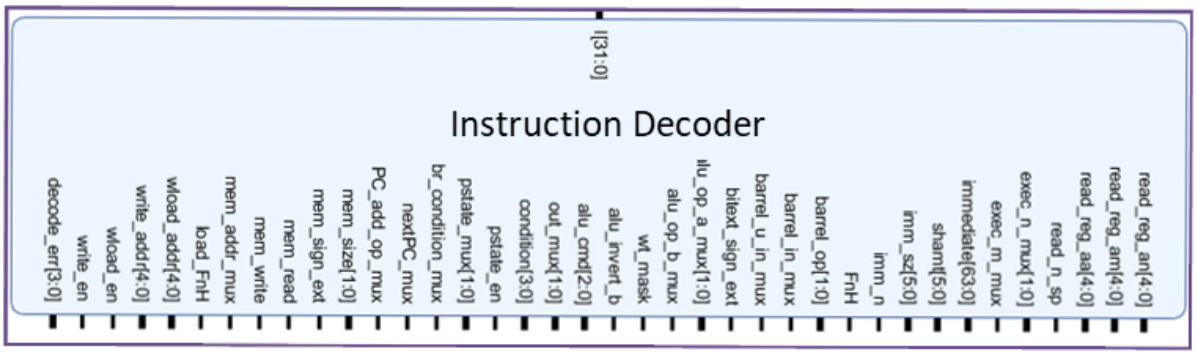


Figure 5: Input/output of Instruction Decoder module

These 38 output signals are basically decoded fields from all the instructions that Arm Education Core supports. The full listing of the I/Os descriptions is in Instruction Decoder I/Os. Some of these signals are used as control or input signals to modules/combination logic in various processing stages including Execute (EXE), Memory Access (MEM), and WriteBack (WB), and even in the Program Counter Logic. We will look at this further in Lab 3B.

Answer the following questions:

1. Based on the simulation in Task: Simulating Lab 3 instructions, view the following signals from the Instruction Decoder module and fill in the values when the following instruction is being decoded (look for the values after they have been updated by the decoder for this instruction):

```
ADD X4, X1, X0
```

Signal	Description	Value
read_reg_an	Index of register to read from Port N.	
read_reg_am	Index of register to read from Port M.	
write_reg_a	Index of register to be written to.	
exec_n_mux	Used in Decoded Instruction Logic to select sources for N datapath to proceed to EXE.	
exec_m_mux	Used in Decoded Instruction Logic to select sources for M datapath to proceed to EXE.	

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

immediate	Immediate value in instruction.	
FnH	1—64-bit instruction; 0—32-bit instruction.	
br_condition_mux	Used in PC logic to specify branch.	
nextPC_mux	Used in PC logic to specify if the next PC value is sourced from a register (1) or an adder (0).	
PC_add_op_mux	Used in PC logic to specify sources for **pc_add**.	

7.2 Exercise: Register File module

The Register File module contains flip-flops with some control logic representing programmer-visible 64-bit registers X0-X31. The following diagram shows the I/Os of the Register File module.

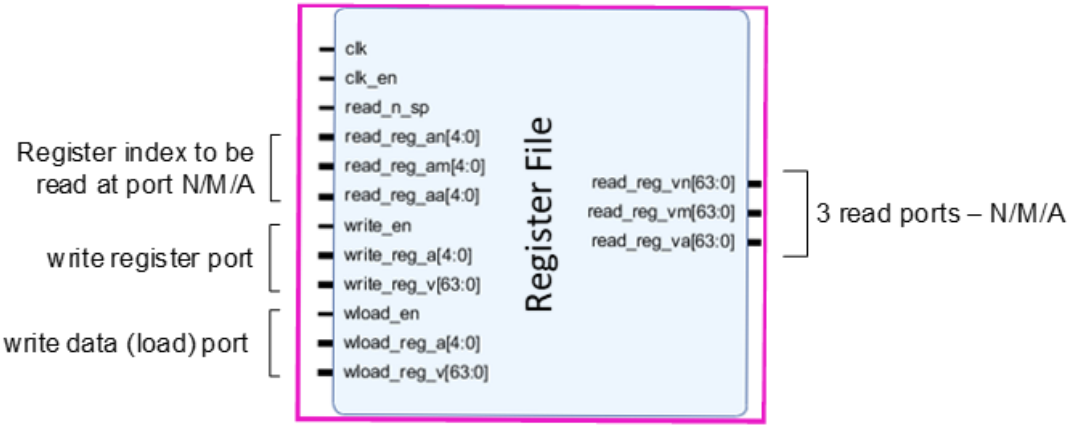


Figure 6: Register File module

The full listing of the I/Os descriptions is in Register File I/Os. In Arm Education Core's Register File, there are:

- Three read ports (N/M/A). These contain values of registers specified in **read_reg_an**, **read_reg_am**, or **read_reg_aa**.
- Two write ports. One for writing to a register once the result is ready at the end of the EXE stage and the other for loading data from memory into a register. Each write port has an enable signal (**write_en**, **wload_en**). **write_reg_a** is connected to **rd_addr**, and **wload_reg_a** is connected to **rt_addr** from the Instruction Decoder.
- **read_n_sp** indicates if register 31 read from N port is the stack pointer (some instructions support this, but this is not the focus of this course lab).

Answer the following questions:

1. Based on the Verilog code in RegisterFile.v, are the writes to registers (write reg or load) done synchronously to a clock, or asynchronously?

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

2. Are the reads for N, M, and A being done synchronously or asynchronously?

3. What processing stage drives the input signal **wload_reg_v** to the Register File?

4. Based on the simulation in Task: Simulating Lab 3 instructions, view the following signals from the Register File module and fill in the values when the following instruction is being decoded:

```
ADD X4, X1, X0
```

Signal	Description	Value
read_reg_an	Index of register to read from Port N.	
read_reg_am	Index of register to read from Port M.	
write_reg_a	Index of register to be written to.	
write_reg_v	Result computed to at the end of EXE stage.	
read_reg_vn	Value of register specified by index in read_reg_an	
read_reg_vm	Value of register specified by index in read_reg_am	

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

7.3 Exercise: Decoded Instruction Logic

The Decoded Instruction Logic contains a bunch of multiplexers that select which signals should be passed on to the Barrel Shifter module and other parts of Arm Education Core.

The diagram below gives an overview of what the Decoded Instruction Logic does.

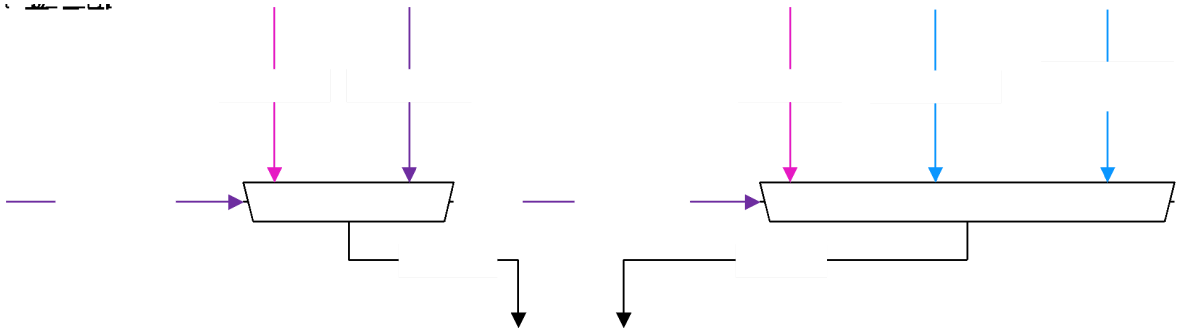


Figure 7: Schematic overview of Decoded Instruction logic

Note:

- **rn_value** and **rm_value** are from the Register File's outputs **read_reg_vn** and **read_reg_vm,** respectively.
- Signals **exec_m** and **exec_n** are routed to other parts of the **Educore** module as well, such as the Program Counter logic, the EXE stage, and the MEM stage.
- **aligned_pc** is basically the Program Counter value. {**aligned_pc[63:12],12'h000**} is the 52 MSB of the Program Counter.
- **exec_n** = "PC" is used for branches (B, BL, B.COND), as well as load literal instructions that are beyond the scope of this course lab. **exec_n** = "PC 52 MSB" is only used for the **ADRP** instruction. **ADRP** forms PC-relative address to 4KB page; this instruction is beyond the scope of this course lab.

Answer the following questions by referring to the **Educore.v** code:

1. What module drives the signals with pink arrows (**rn_value**, **rm_value**) in Figure 7?

2. What module drives the signals with purple arrows (**exec_m_mux**, **exec_n_mux**) in Figure 7?

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

3. Where do the signals with blue arrows (**aligned_pc**, **aligned_pc[63:12]**) come from in Figure 7?

8 Summary

In this lab, we have looked at key components, modules, control signals, and combinational logic in the Instruction Fetch and Instruction Decode stages of Arm Education Core, as shown below:

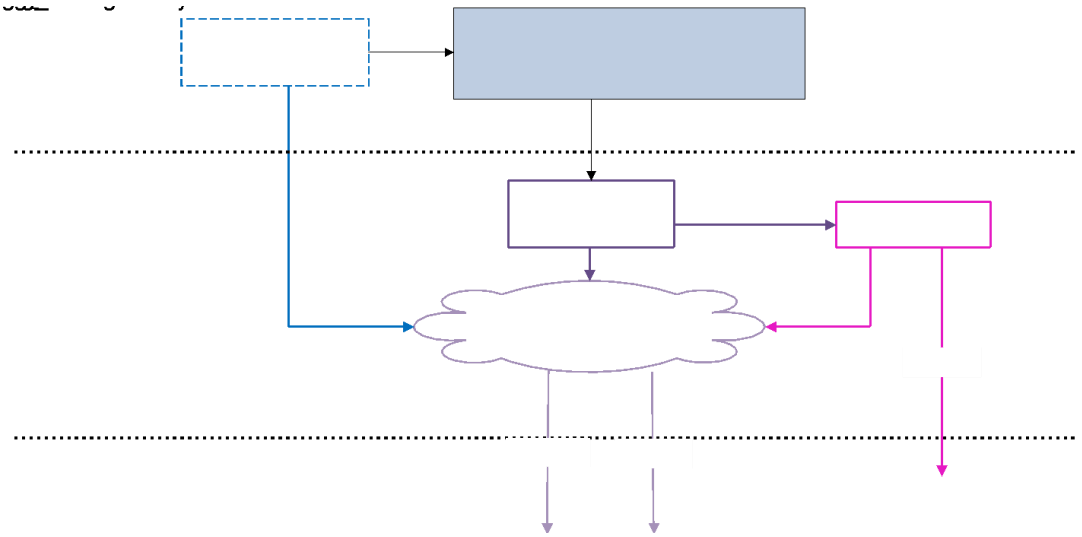


Figure 8: Overview of Instruction Fetch and Instruction Decode stages

- The Program Counter Logic selects sources for the Program Counter.
- The Program Counter points to the instruction to be fetched.
- The instruction fetched is passed to the Instruction Decoder module, which outputs values and control signals that are passed to other key components of Arm Education Core, including the Program Counter Logic and Register File.
- The Register File has 3 asynchronous read ports (N, M, A) and 2 synchronous write ports (one for output of internal computation (i.e., output of ALU) and the other for memory-loaded values).
- **exec_a**, **exec_m**, and **exec_n** correspond to values of operands in the instructions and are passed on to the EXE stage for further execution.
- The Decoded Instruction Logic selects the sources for **exec_m** and **exec_n**.
-

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

9 Appendix

9.1 Instruction Decoder I/Os

Input signal

Signal	Size (bits)	Description
instruction	32	Binary representation of the fetched instruction

Output signals

Signal	Size (bits)	Description
read_reg_an	5	Index of register to read from port N
read_reg_am	5	Index of register to read from port M
read_reg_aa	5	Index of register to read from port A
read_n_sp	1	ACTIVE HI—Indicates that read_reg_an of N port 31 is the stack pointer, not the zero register
exec_n_mux	2	Selects from the following sources to the N datapath: ● Value of the PC for this instruction ● Value of the PC most significant 52 bits (4KB PC page). This is used for the ADRP instruction. ADRP forms PC-relative address to 4KB page; this instruction is beyond the scope of this course lab. ● Value of register read from N port
exec_m_mux	1	Selects from the following sources to the M datapath: ● The immediate value obtained from the instruction (shifted & sign extended appropriately) ● Value of the register read from the M port
immediate	64	Value of the immediate obtained from the instruction after it has been subjected to shifting and/or sign extension if required.
shamt	6	Shift amount to be carried out by the barrel shifter
imm_sz	6	Immediate decoder field size; i.e., number of contiguous set bits.
imm_n	1	ACTIVE HI—Immediate decoder 64-bit field size indicator.

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

FnH	1	Operand/result size of execution modules output: ● 1 → 64-bit ● 0 → 32-bit
barrel_op	2	Specifies the operation of the barrel shifter from LSL/LSR/ASR/ROR.
barrel_in_mux	1	Specifies the source to lower 64-bit input to barrel shifter. ● N datapath ● M datapath
barrel_u_in_mux	1	Specifies the source to upper 64-bit input to barrel shifter. ● Same as lower ● N datapath
bitext_sign_ext	1	Specifies the operation of bit extension logic to be: ● 0 → Zero extension ● 1 → Sign extension
alu_op_a_mux	2	Selects from the following sources to the operand A input to the ALU: ● Zero ● Datapath N ● Datapath A
alu_op_b_mux	1	Selects from the following sources to the operand B input of the ALU: ● WMASK output of the immediate decoder ● Bit extension logic output
wt_mask	1	ACTIVE HI—applies bit clearing on ALU operand A source and bit masking on operand B source before ALU operation.
alu_invert_b	1	ACTIVE HI—inverts operand B before applying ALU operation.
alu_cmd	3	Specifies ALU operation from logic operations (AND/ORR/EOR) and arithmetic operations (ADD_0/ADD_1/ADC).
out_mux	2	Selects the output to be written to the register file or used as an address in memory. ● Value of the PC of this instruction + 4 ● Output of the ALU ● Conditional output (selects Datapath N if condition is true, otherwise selects output of ALU. See CSEL/CSINC/CSINV/CSNEG). ● System register value (only one system register is implemented, namely the PSTATE).
condition	4	Specifies condition code to compare with PSTATE current state.
pstate_en	1	ACTIVE HI—Enable writes to the PSTATE register

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

pstate_mux	2	Selects the input to write into the PSTATE register. ● ALU output flags ● Conditional (If condition is true, then select ALU output otherwise overwrite with immediate value. See CCMP/CCMN). ● Datapath A value
br_condition_mux	1	Specifies whether this instruction (if branch) is conditional to PSTATE or unconditional.
nextPC_mux	1	Specifies whether the next PC value is sourced from a register or the adder.
PC_add_op_mux	1	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.
mem_size	2	Specifies the size of memory load or store to be 8/16/32/64 bytes.
mem_sign_ext	1	ACTIVE HI—Specifies whether the memory-loaded value is subjected to a sign extension.
mem_read	1	ACTIVE HI—This instruction reads from memory (load).
mem_write	1	ACTIVE HI—This instruction writes to memory (store).
mem_addr_mux	1	Selects between operand N and the output of the ALU as the memory address for the read/write.
load_FnH	1	Loaded value size: ● 1 → 64-bit ● 0 → 32-bit
wload_addr	5	Specifies register index to write with memory-loaded value.
write_addr	5	Specifies register index to write with computation (see out_mux) output.
wload_en	1	ACTIVE HI—registers write enable memory load port.
write_en	1	ACTIVE HI—registers write enable for computation (see out_mux) output port.
decode_err	4	Indicates any errors such as undefined instructions or indicates a halt by the YIELD instruction.

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

9.2 Register File I/Os

Input Signals

Signal	Size (bits)	Description
clk	1	The clock signal synchronizing the design. Reacts at posedge.
clk_en	1	ACTIVE HI—Clock enable signal.
read_n_sp	1	ACTIVE HI—Indicates that register 31 read from N port is the stack pointer (SP)
read_reg_an	5	Index of register to be read at port N
read_reg_am	5	Index of register to be read at port M
read_reg_aa	5	Index of register to be read at port A
write_en	1	ACTIVE HI—Stores the value presented at write_reg_v at register specified by write_reg_a.
write_reg_a	5	Index of register to be written with value write_reg_v
write_reg_v	64	Processor-computed value (as opposed to memory-loaded)
wload_en	1	ACTIVE HI—Stores the value presented at wload_reg_v at register specified by wload_reg_a
wload_reg_a	5	Index of register to be written with value wload_reg_v
wload_reg_v	64	Memory-loaded value to be written to one of the registers

Output Signals

Signal	Size (bits)	Description
read_reg_vn	64	Value of register specified by index in read_reg_an

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

read_reg_vm	64	Value of register specified by index in read_reg_am
read_reg_va	64	Value of register specified by index in read_reg_aa

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.