DECA Lab Spring Term

Spring Part 1: EEP1 Datapath & ALU instructions

Department of Electrical and Electronic Engineering

Imperial College London

v2.0.2

Spring 2026

Contents

Glossary of Key Terms

This glossary summarises the main hardware and notation used in the DECA Spring labs. You are not expected to memorise it; use it for reference while working through the tasks.

Datapath The part of the CPU where data moves and is transformed: it contains the register file, ALU, shifter, and the buses between them.

Control path The part of the CPU that decides *what* the datapath should do each cycle. It reads the instruction word and generates control signals (for example, which register to read, which ALU operation to use).

Instruction word The 16-bit binary value fetched from instruction memory each cycle. Different groups of bits (called *fields*) encode the operation (opcode), registers, and immediate values.

Fields a, b, c In the EEP1 instruction word, some bits are grouped and named a, b, and c. These fields usually specify registers. In assembly they appear as `Ra`, `Rb`, `Rc`.

Immediate value (#IMM) A constant value written directly in the instruction word. In EEP1 assembly, an immediate is written as `#3`, `#-7`, etc. The hardware sign-extends this 8-bit value to 16 bits before using it.

Register file A small, fast memory that holds the CPU registers (for EEP1, registers R0–R7). It usually has multiple read ports (to output register data) and at least one write port (to update a register).

Waveform viewer An ISSIE tool that displays how signal values (such as register contents, control signals, and the instruction word) change over time, cycle by cycle.

dpdecode A block in the datapath that decodes the 16-bit instruction word and produces control signals for the register file, ALU, shifter, and other datapath components.

ALU The Arithmetic Logic Unit. It performs arithmetic (ADD, SUB, ADC, SBC, CMP) and logical operations (AND, etc.) on its input operands and outputs a result and flags.

ALUOPC The field in the instruction word that encodes which ALU operation to perform (ADD, SUB, AND, etc.). Inside the ALU, this is further decoded by the `ALU.DECODE` block.

FLAGCIN A control input to the ALU that allows the current carry flag to be used as the carry-in for some instructions (for example ADC and SBC).

op2sel A 1-bit control signal (from `dpdecode`) that selects whether the ALU's second operand comes from a register or from the immediate value.

ad1sel A 1-bit control signal (from `dpdecode`) that selects which instruction field (a or c) is used as the write address for some arithmetic instructions.

WEN1 A write-enable control signal for the register file. When `WEN1 = 1`, the result from the datapath is written into a register. When `WEN1 = 0`, no register is written in that cycle.

INS(8) Bit 8 of the instruction word, used in EEP1 to distinguish variants of some instructions (for example register-to-register vs. immediate forms).

Barrel shifter A hardware block that can shift a word by 0,1,2,4,8 bits in a single cycle. EEP1 uses four stages (SHIFT1, SHIFT2, SHIFT4, SHIFT8) controlled so that the overall shift matches the amount encoded in the instruction.

Sign-extension A method of converting a smaller signed integer (for example 8 bits) to a larger one (16 bits) by copying its sign bit into the new upper bits. This preserves negative values when moving immediates into the datapath.

Introduction to DECA Spring Term labs

These labs are a central part of the computer architecture module. Lectures introduce the concepts of instruction sets, registers, and datapaths in the abstract. The labs are where you see how these concepts are realised in a real CPU design and learn how to reason about hardware behaviour.

In this term's labs you will:

- explore the internal structure of a simple CPU (EEP1), focusing on its datapath and control path;
- learn how to use tools (in ISSIE) to trace signal values, inspect instruction words, and understand how data moves each clock cycle;
- practice explaining how specific pieces of hardware implement particular instruction behaviours;
- begin to design and modify small parts of the CPU yourself.

Many of the questions ask you to identify and explain hardware designed for particular tasks. Answers should be detailed and show clear evidence that you have:

- Located the relevant hardware blocks and signals;
- Used simulations (waveforms, truth tables, etc.) to observe how they behave;
- Connected those observations back to the instruction set and the intended function.

In Section 1 (Register-register MOV instructions & the register file) we provide an example "thought process" and a worked approach. You are expected to use similar approaches throughout the term, and to record your exploration in your lab book. These notes will support your learning and will be useful when answering assessment questions.

Extension questions & challenge design tasks:
Optional extension questions and design tasks appear throughout the labs for students who want to explore beyond the examinable content. End-of-term assessments will not directly test these specific optional tasks. However, marks above 75% will require you to answer more challenging questions that probe deeper understanding, application, or design skills. Tackling extension material, or explaining your own design experiments, can make it easier to answer such questions confidently, but it is not required for a pass or for good marks in the main range.

Spring Part 1: introduction

Important: Before starting this lab, you must:

- have read the module introduction and the environment setup guide;
- have completed the introductory problem sheet.

All of these are available in the Problem Sheets.

Time expectation. You are expected to complete Lab 1 (this handout) in two weeks.

What you will do in Lab 1.

i. Learn and practise tools and approaches in ISSIE for analysing and understanding complex hardware designs (*waveform viewer, wire tracing, and truth tables*).

ii. Use these tools to dissect the `eep1lab1` CPU design and see how EEP1 instructions are executed in the datapath and in the ALU.

The design `eep1lab1` is a pre-built implementation of the EEP1 CPU datapath. You will use it throughout Lab 1.

Jump instructions. Some of the jump-related hardware in `eep1lab1` is not yet complete. You will extend this hardware in a later lab that focuses on the control path. All of Lab 1 can be completed *without* modifying the hardware in `eep1lab1`.

1 Register-register MOV instructions & the register file

Relevant Lecture Material: Lecture 2 - Register Files.

Objectives

In this section you will learn:

1. how to use the waveform simulator to analyse how data moves in hardware;
2. how fields within the instruction word control reading from and writing to registers;
3. which pieces of hardware implement this control in the EEP1 datapath.

Tasks

□ **Task 1.** *Using the environment setup guide for support, convert the assembly code in Figure 1 into machine code and load it into `eep1lab1`. For all of Lab 1 you will use `eep1lab1` as your EEP1 design.*

For this section we will focus on the second instruction `MOV R7, R6`. For the first instruction `MOV R6, #3`, all you need to know is that it loads the value 3 into register R6.

```
MOV R6, #3
MOV R7, R6
```

Figure 1: Assembly language program of two EEP1 MOV instructions

1. **Write address control.** Identify the hardware that controls *which register* is written by an instruction, and the specific field(s) of the instruction word that drive it. Explain in detail how this hardware works.

   Your answers to Question 5 of the introductory problem sheet will be useful here.

   A good answer will show your reasoning steps as you explore the design, not just the final conclusion. One possible approach is:

   i. Run the simulation and use the waveform viewer (see the environment setup guide) to check that register values change as expected. Use `codemem.dout` to see the instruction word being executed each cycle.

      For these MOV instructions, do the registers change in the *same* cycle as the instruction that changes their values, or in the *following* cycle?

      You should see in the waveform viewer that the register values change one clock cycle after the instruction that updates them. You should also see that the first instruction is executed during clock cycle 0.

   ii. Using the EEP1 documentation (ALU Instructions : detail), confirm that for register-register MOV instructions (where `INS(8) = 1`) the field *a* (written as `Ra` in assembly) determines which register is written in the register file. Then, using the *ALU Instruction Encoding* sheet, state exactly which bits of the 16-bit instruction word form field *a*.

      In the waveform viewer, display the instruction words (`codemem.dout`) in binary and read the write address from this field. Check that this matches the assembly instructions and your observation of which register values change.

   iii. Open the `regfile` sheet. Examine the names and widths of its input and output ports. From the port names (and the hardware they drive), deduce which input port controls the address of the register that is written. You should find that this port is `AD1`.

iv. Trace the wire connected to `AD1` back into the `dpdecode` block. The purpose of `dpdecode` is to interpret the 16-bit instruction word each cycle and to generate the appropriate control signals for the CPU. By inspecting the hardware, determine which bits of the 16-bit instruction word drive the signal connected to `AD1`, and compare this with your earlier answer about which bits encode field *a*.

(For this investigation you may assume that `AD1SELC = 0`.)

v. Return to the register file and trace what hardware the `AD1` input connects to internally. Through which port does the data being written enter the register file?

Explain how this hardware is arranged so that the value on `AD1` selects *which* register the incoming data is written to.

vi. Using the waveform viewer, track the value of `AD1` in each cycle. Does it correspond to the register that is written in the *next* cycle, as you expect?

2. **Read address control.** Identify the hardware that controls *which register the data is read from* in a register-to-register MOV instruction such as in Figure 1. Again, explain how it works in detail.

Use a similar approach to Question 1. A suggested (shorter) method is:

i. Using ALU Instructions : detail, confirm that for a MOV instruction with `INS(8) = 0` the field *b* (written as `Rb`) selects the register from which data is read in the register file. Using the *ALU Instruction Encoding* sheet, identify exactly which bits of the instruction form field *b*.

In the waveform viewer, inspect the second instruction word (the one in cycle `x01`) in binary. Read the read address from this field and check that it matches the source register in the assembly instruction.

ii. Open `dpdecode` and trace the wire that carries the bits of field *b*. Note which port of the register file it connects to. This port is the one that controls the read address for these register-to-register MOV instructions.

iii. Open the register file and follow the hardware connected to that read-address port. Explain how the hardware uses the input signal to select which register's data is placed on the register file's output bus, and state which of the two data output ports this data leaves through.

Extension questions

1. In a register-register MOV, what happens if $a = b$ (for example `MOV R7, R7`)? Does this cause any problems?

Add the instruction

```
MOV R7, R7
```

to the code in Figure 1, simulate the updated program, and use the waveform viewer to investigate and explain what happens.

Review

You should now understand, and be able to answer questions on, the following concepts:

i. using the waveform simulator and wire tracing to understand hardware behaviour;

ii. how read and write addresses for register-to-register MOV instructions are encoded in the 16-bit instruction word;

iii. hardware that controls which register data is written to and read from in simple register-to-register MOV instructions.

2 Immediate value MOV instructions

*Relevant Lecture Material:*Lecture 2 - Datapath Design with Register Files.

Objectives

In this section you will learn:

1. how immediate values are encoded inside the 16-bit instruction word of MOV instructions;
2. what additional hardware is needed in the datapath to support immediate values.

Tasks

☐ **Task 2.** *Convert the assembly code in Figure* **??** *into machine code and load it into* ***eep1lab1****.*

```
MOV R0, #3
MOV R1, R0
ADD R1, #1
ADD R3, R0, R1
```

Figure 2: Assembly language program with MOV and ADD instructions.

Remember: in EEP1 assembly, `#N` denotes an *immediate* constant value. For these MOV instructions the immediate is stored in the instruction word as an 8-bit signed integer and then sign-extended to 16 bits when used in the datapath.

1. The write address of immediate MOV instructions (like those in Figure **??**) is still controlled by field *a*. However, for `MOV` ***Ra*****, #IMM** instructions, the data that enters the register file through the data input port comes from the immediate value encoded in the instruction word, not from a register.

   Run a simulation and use the waveform viewer to confirm that, for both instructions, the correct register is written and the value written is the immediate encoded in the instruction, after sign-extension to 16 bits.

2. Using ALU Instructions : detail, and ALU Instruction Encoding sheet, determine which bit(s) in the 16-bit instruction word control:
   i. whether a MOV instruction reads from an immediate field (#IMM) or from a register;
   ii. which bits form the 8-bit immediate value.

3. For this question you may assume that, for these MOV instructions, the ALU simply passes the value on its `IMM` input to the register file's data input port. You may also ignore the `EXT` block for now; in this lab it behaves as a direct connection with no delay.

   Open `dpdecode` and, using your answer to Question ii., investigate through which output port of `dpdecode` the immediate (#IMM) value is sent in order to reach the ALU's `imm` port.

4. Investigate the hardware that converts the 8-bit immediate into the 16-bit value that travels through the ALU and into the register file. How does this hardware sign-extend the 8-bit field to 16 bits so that negative numbers (for example `#-3`) are preserved when loaded from the instruction word?

   A good way to answer this is to simulate the instructions in Figure **??** and compare the 8-bit encoded immediate with the 16-bit value observed on the datapath.

Review

You should now understand, and be able to answer questions on, the following concepts:

i. using wire tracing as a technique to understand hardware;

ii. how immediate values are encoded in instruction words and sign-extended to the datapath width;

iii. which hardware routes immediate values from the instruction word into the datapath.

3 Data flow from register file to ALU

Relevant Lecture Material: Lecture 2 - Datapath Design with Register Files.

Objectives

In this section you will learn:

1. how to use waveform viewers together with truth tables to understand hardware logic;
2. how fields a, b, and c control register reads and writes in more detail, and how their roles differ for different instruction types;
3. how data flows differently for different instructions (MOV and ADD) and for different forms (register vs. immediate);
4. the purpose of the `op2sel` control signal in the context of immediate (#IMM) operands;
5. the purpose of the `ad1sel` control signal in the context of write addresses for arithmetic operations with three operands;
6. how data flows between the ALU and the register file (see Figure 3).

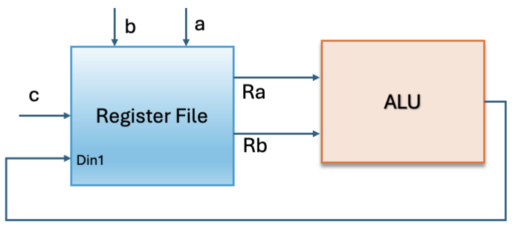


Figure 3: EEP1 datapath simplified

Tasks

☐ **Task 3.** *The assembly program in Figure 4 consists of four instructions that exercise the datapath in four different ways:*

- *a MOV with an immediate operand,*
- *a MOV with a register operand,*
- *an ADD with an immediate operand,*
- *an ADD with two register operands.*

These four patterns are the basic cases you will study in this section.

Write these four lines in a text file and load the corresponding machine code into `eep1lab1`*.*

```
MOV R0, #3
MOV R1, R0
ADD R1, #1
ADD R3, R0, R1
```

Figure 4: Assembly language program with MOV and ADD instructions.

1. **Register file to ALU connections.** Using wire tracing in ISSIE, identify through which ports register data is sent from the register file to the ALU.

   For each register file data output port (for example `DOUT2`, `DOUT3`):

  - state which address field in the instruction word (a, b, or c) controls which register is read on that port;
  - state which ALU input port this register file output is connected to.

2. **Immediate value into the ALU.** Name the ALU input port through which immediate values enter (this is the same port you used in Section 2) and confirm how it is connected to `dpdecode`.

3. **ALU output back to register file.** Name the output port of the ALU that carries the result of operations. Explain how this output is routed back to the register file so that the result can be stored in a register. In particular, identify:
   - which register file data input port it connects to;
   - how the register write address is selected in these instructions.

4. **Control signals `op2sel` and `ad1sel`.** The signals `op2sel` and `ad1sel` are 1-bit control signals from `dpdecode`. They control how data flows into the ALU and register file. In this question you will determine their purpose and for which instructions they are 0 or 1.

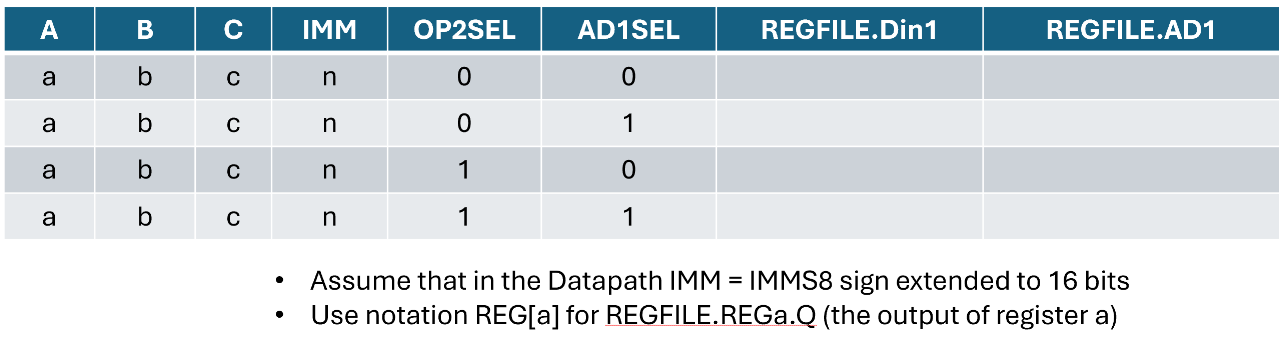
   One useful method is:

   i. Construct a 4-row truth table whose input is the assembler instruction (one of the four in Figure 4) and whose outputs are the values of `op2sel` and `ad1sel`. Run the simulation in ISSIE and use the waveform viewer to read the values of these signals for each instruction. Fill them into your truth table.

   ii. Using your truth table and the ALU Instructions : detail sheet, explain what conditions cause each signal to be 1. (There is a hint in the introduction: think about when the second operand of the ALU should come from an immediate, and when the write address should come from field a or field c.)

   iii. Trace `op2sel` and `ad1sel` to the multiplexers (muxes) they control. Using the instruction documentation, explain how the values of these signals change the data flow into the ALU and register file for the different EEP1 instructions.

   We strongly suggest that you complete an algebraic truth table using the template in Figure 5 for MOV and ADD instructions, to make your explanation of the datapath behaviour precise.



A	B	C	IMM	OP2SEL	AD1SEL	REGFILE.Din1	REGFILE.AD1
a	b	c	n	0	0		
a	b	c	n	0	1		
a	b	c	n	1	0		
a	b	c	n	1	1		

- Assume that in the Datapath IMM = IMMS8 sign extended to 16 bits
- Use notation REG[a] for REGFILE.REGa.Q (the output of register a)

Figure 5: Template for algebraic Truth-Table showing EEP1 Datapath functionality

Extension questions

1. When executing a `MOV` *`Ra`*`, #IMM` instruction, how does the register read port of the register file behave? Does it still output register data? Explain what happens to this data and which hardware arrangement ensures that the immediate value, rather than the register output, is written into the destination register.

2. Explain why having only a *single* read port from the register file to the ALU would severely limit the CPU's ability to perform useful arithmetic operations in a single cycle.

3. Write a new text file containing the four lines of assembly code in Figure 6. Add three more lines of code that add the contents of registers R0–R3 and store the result in R4, without modifying registers R5–R7. Simulate your code in `eep1lab1` and use the waveform viewer to confirm that the result in R4 is correct.

```
MOV R0, #12
MOV R1, #-7
MOV R2, #233
MOV R3, #1
```

Figure 6: Challenge assembly code

Review

You should now understand, and be able to answer questions on, the following concepts:

1. how to use the waveform viewer and truth tables together to explain the function of hardware;
2. through which ports data flows from the register file to the ALU;
3. which fields of the 16-bit instruction word control which registers are read on each register file output port;
4. why the write address can be either field *a* or field *c*, and how the hardware selects between them;
5. the role of the `op2sel` and `ad1sel` signals and the multiplexers they control;
6. how ALU results are written back into the register file.

4 EEP1 ALU Instructions

Relevant Lecture Material: First part of lecture 3 : inside a CPU's arithmetic logic unit.

Objectives

In this section you will learn:

1. how the ALU performs operations on register data and writes new values back to the register file;
2. which different operations the ALU can perform in EEP1;
3. how the instruction word controls which ALU operation is executed.

Tasks

☐ **Task 4.** *Use ISSIE's waveform simulator to simulate the first eight clock cycles of the instructions in `lab1testmovadd.` (in the `eep1lab1` folder). Use both the assembly code and the waveform simulation to answer the questions below.*

1. Through which port does the ALU receive the information about *which* operation to perform (ADD, SUB, AND, etc.)? State which field(s) of the 16-bit instruction word are connected, directly or indirectly, to this port.

   Then, using the hardware, explain how this port connects to the `ALU.DECODE` block and other internal ALU blocks (such as `ADDSUB`). You do not need to understand the detailed internal design of these blocks yet; you only need to explain how the instruction word selects one ALU operation rather than another.

Review

You should now understand, and be able to answer questions on, the following concepts:

1. how the ALU can implement multiple arithmetic and logical operations;
2. how the instruction word controls which ALU operation is executed.

5 ADDSUB Block design in the ALU

Relevant Lecture Material: Lecture 3: design of add and subtract.

Open the ALU sheet. The block `ALU.ADDSUB` implements addition and subtraction. Its behaviour depends on the control signals `INVERT` and `ADDSUBCIN`, which are generated by the `ALU.DECODE` block. `ALU.DECODE` is itself controlled by `ALUOPC` and `FLAGCIN`.

Objectives

In this section you will:

- see how the ALU can use one adder to implement both addition and subtraction (with and without carry);
- understand the role that `ALU.DECODE` plays in generating the correct control signals for the `ADDSUB` block.

Tasks

☐ **Task 5.** *The five arithmetic instructions ADD, SUB, ADC, SBC, and CMP all use the ADDSUB and ALU.DECODE blocks.*

1. **Function of `ADDSUB`.** Explain the function of the `ADDSUB` block in terms of its inputs.

   A strong way to support your explanation is to write an algebraic truth table for `ADDSUB.OUT` as a function of the algebraic inputs `INA`, `INB`, `CARRYIN` and the binary control input `INVERT`. Express the behaviour symbolically (for example, "OUT = INA + INB + CARRYIN" when `INVERT = 0`, etc.).

2. Using your answer to the previous question and the specification of the five arithmetic instructions in Figure 8 of the handout, explain how the control signals from `ALU.DECODE` into `ADDSUB` allow all five instructions (`ADD`, `SUB`, `ADC`, `SBC`, and `CMP`) to be executed.

   The clearest way to do this is to write an algebraic truth table for the `ALU.DECODE` block, with outputs `ALUDECODE.INVERT` and `ALUDECODE.ADDSUBCIN`, and inputs `ALUDECODE.ALUOPC` and `ALUDECODE.FLAGCIN`. Use don't-cares to simplify cases where the exact value of an output does not matter (for example, when the result of `ADDSUB.OUT` will be ignored).

   For instructions other than the five considered here, `ADDSUB.OUT` is a don't-care. By looking at where the wire from `ADDSUB.OUT` goes, explain why its value does not affect the correct behaviour of the CPU for those instructions.

3. Explain how the hardware implementation of `ALU.DECODE` achieves the behaviour you have described in your truth table. Focus on how the combination of `ALUOPC` and `FLAGCIN` is mapped onto the control signals `INVERT` and `ADDSUBCIN`.

Extension question

1. What is the purpose of the `FLAGCIN` input? Which type of operation does it enable, and for which instructions is it used?

Review

You should now understand, and be able to answer questions on, the following concepts:

1. how the `ADDSUB` block works and how it combines its inputs to produce addition and subtraction;
2. how the `ALU.DECODE` block controls `ADDSUB` to implement the `ADD`, `SUB`, `ADC`, `SBC`, and `CMP` instructions.

6 CMP and AND instructions

Relevant Lecture Material: Lecture 3 - Inside a CPU Arithmetic Logic Unit.

Objectives

In this section you will learn:

1. how `CMP` differs from `SUB` at hardware level, and why;
2. how the `AND` instruction is implemented in hardware and how it relates to Boolean AND.

Tasks

☐ **Task 6.** *Using the material from Lecture 3 and Figure 8, review the behaviour of the `CMP` instruction in detail and compare it with the other ALU instructions that use `ADDSUB`.*

1. **CMP vs SUB.** The instruction `CMP` performs a subtraction internally, but its purpose is to update flags rather than to write a data result.
   i. Identify which piece of hardware is responsible for the difference in behaviour between `SUB` and `CMP`. In particular, discuss the significance of the signal `DPDECODE.WEN1`.
   ii. Inspect the logic in the `dpdecode` sheet that drives `DPDECODE.WEN1`. You may assume that, for this question, `JSR = 0` and `LDR = 0`.
   iii. Write a truth table that shows the value of `WEN1` for each of the eight ALU instructions shown in Figure 7. Explain what this table shows about when the ALU result is written back to the register file and when it is not.
2. **AND instruction.** Look at the multiplexers (`MUX2`) in the `alu` sheet and the encoding of the `AND` instruction in the instruction set.
   i. Identify the hardware that implements the `AND` instruction: which input signals feed the logic block, and through which mux selection does the ALU choose the AND result?
   ii. Explain how this hardware behaviour corresponds to the Boolean AND operation on the two operands.

Extension questions

1. If you wanted to implement additional logic functions (for example OR, XOR, etc.), which blocks would you need to add or modify? How would you change the existing hardware and the width of `ALUOPC` to support these new operations?
2. Design a new ALU sheet that extends the current design to perform additional logic functions as well as the existing ones. You will need to change the width of `ALUOPC`. This design does not need to integrate with the EEP1 datapath.

Review

You should now understand, and be able to answer questions on, the following concepts:

1. how `CMP` behaves differently from the other `ADDSUB`-based operations, and which hardware causes this difference;
2. how the `AND` instruction is implemented and how it corresponds to Boolean AND.

7 SHIFT instructions

Lecture 3 - Shift Instructions.

Objectives

In this section you will learn:

1. how shift commands (`LSL`, `LSR`, `ASR`, `XSR`) are encoded in the EEP1 instruction set;
2. how these shift commands are executed using a barrel shifter and the associated shifter hardware.

Tasks

☐ **Task 7.** *Using the LSL, LSR, ASR, and XSR shift instruction definitions from the lectures, predict the waveforms you expect from the code in `lab1testshift.txt`. Then simulate this program in `eep1lab1` and use the waveform viewer to answer the questions below.*

1. Open the `shift` sheet. Using your tracing skills and the EEP1 Instruction Encoding sheet, deduce the meaning of each signal on the ports of the `shift` block. You do *not* need to understand the internals of each individual shifter (`SHIFT1`, `SHIFT2`, etc.) in detail; you may assume that each shifter:
   - shifts its input by a fixed number of bits when its enable signal `EN` is 1;
   - passes its input through unchanged when `EN` is 0;
   - uses `SHIFTOPC` to select which of the four shift behaviours (`LSL`, `LSR`, `ASR`, `XSR`) it performs.

   Explain how the combination of `SHIFT1`, `SHIFT2`, `SHIFT4`, and `SHIFT8` allows the barrel shifter arrangement to perform the correct shift (type and amount) given the shift count (`SCNT`) encoded in the instruction.
2. Open the `SHIFT1` sheet and explain how its hardware implementation enables it to:
   - shift the input data by 1 bit in the required direction;
   - implement all four shift types (`LSL`, `LSR`, `ASR`, `XSR`) with appropriate carry-in and carry-out behaviour.

   At first glance the hardware may look complicated. To structure your exploration, you may find the following hints helpful:
   - There are two bus-splitting arrangements, one corresponding to a left shift and one to a right shift. Identify which is which first.
   - The different candidate shifted results (`LSL`, `LSR`, `ASR`, `XSR`) converge at a multiplexer (`MUX2`) that selects the final output.
   - The bit shifted into the word (the “shifted-in” bit) depends on the shift type and is selected by another multiplexer (`MUX4`). By looking at its inputs, you can see how the different shift-in behaviours are implemented for each shift type.
3. Compare the other three shifter sheets (`SHIFT2`, `SHIFT4`, `SHIFT8`) with `SHIFT1`. Describe how they differ so that they shift by 2, 4, or 8 bits respectively, while otherwise implementing the same four shift types. In particular, explain how the carry-in behaviour of `ASR` is implemented correctly when the shift amount is greater than 1, and why this is essential for its intended meaning.

Extension questions

1. Inspect the multiplexer that selects the shifted-in bit for each shift block. What limitation of the `XSR` instruction does this hardware arrangement reveal?
2. Why is the 1/2/4/8 barrel shifter more efficient than using sixteen `SHIFT1` blocks in series? Could you design a barrel shifter that way, and what would be the disadvantages?

Review

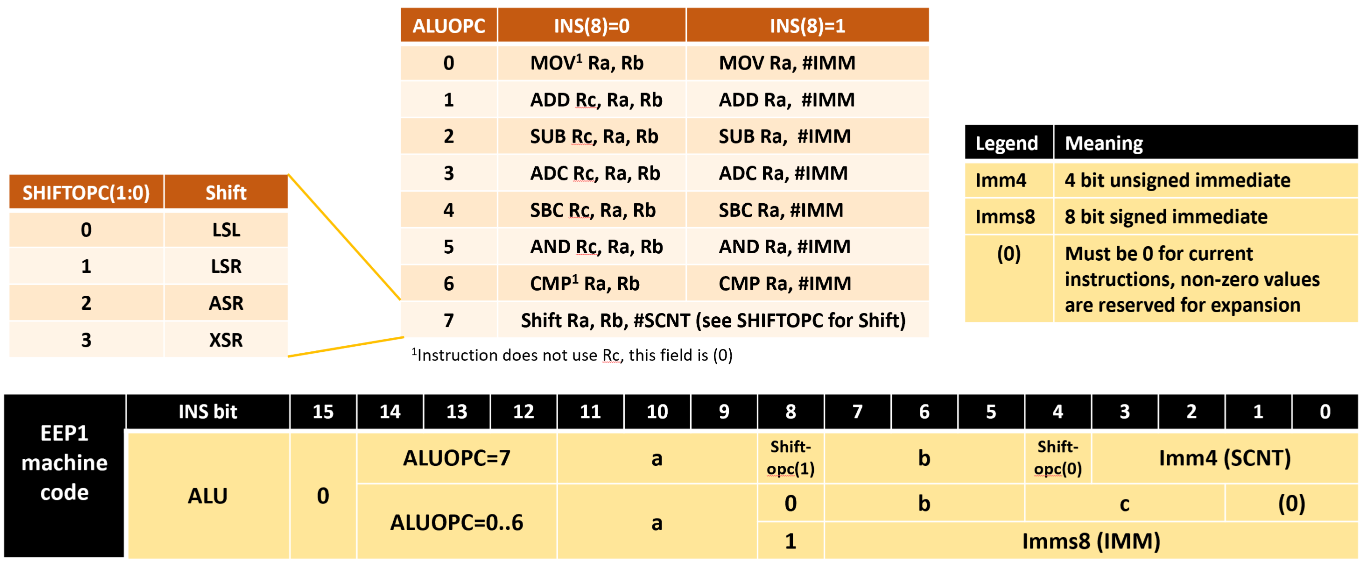
You should now understand, and be able to answer questions on, the following concepts:

1. how the different shift instructions are encoded into the 16-bit instruction word;
2. how these shift instructions are implemented using a barrel shifter and supporting hardware.

Reflections

In Lab 1 you have explored almost all of the EEP1 datapath design. You have seen how the CPU's ALU instructions are implemented in hardware using a register file, an ALU, and various control signals decoded from the instruction word.

In Lab 2 the focus will move to the control path. You will see how the EEP1 CPU decides which instruction address to fetch next and how different kinds of jumps are implemented. You will complete the EEP1 implementation of the jump instructions and investigate how the flags (`FlagN`, `FlagZ`, `FlagC`, `FlagV`) are set and used.



ALUOPC	INS(8)=0	INS(8)=1
0	MOV[1] Ra, Rb	MOV Ra, #IMM
1	ADD Rc, Ra, Rb	ADD Ra, #IMM
2	SUB Rc, Ra, Rb	SUB Ra, #IMM
3	ADC Rc, Ra, Rb	ADC Ra, #IMM
4	SBC Rc, Ra, Rb	SBC Ra, #IMM
5	AND Rc, Ra, Rb	AND Ra, #IMM
6	CMP[1] Ra, Rb	CMP Ra, #IMM
7	Shift Ra, Rb, #SCNT (see SHIFTOPC for Shift)	

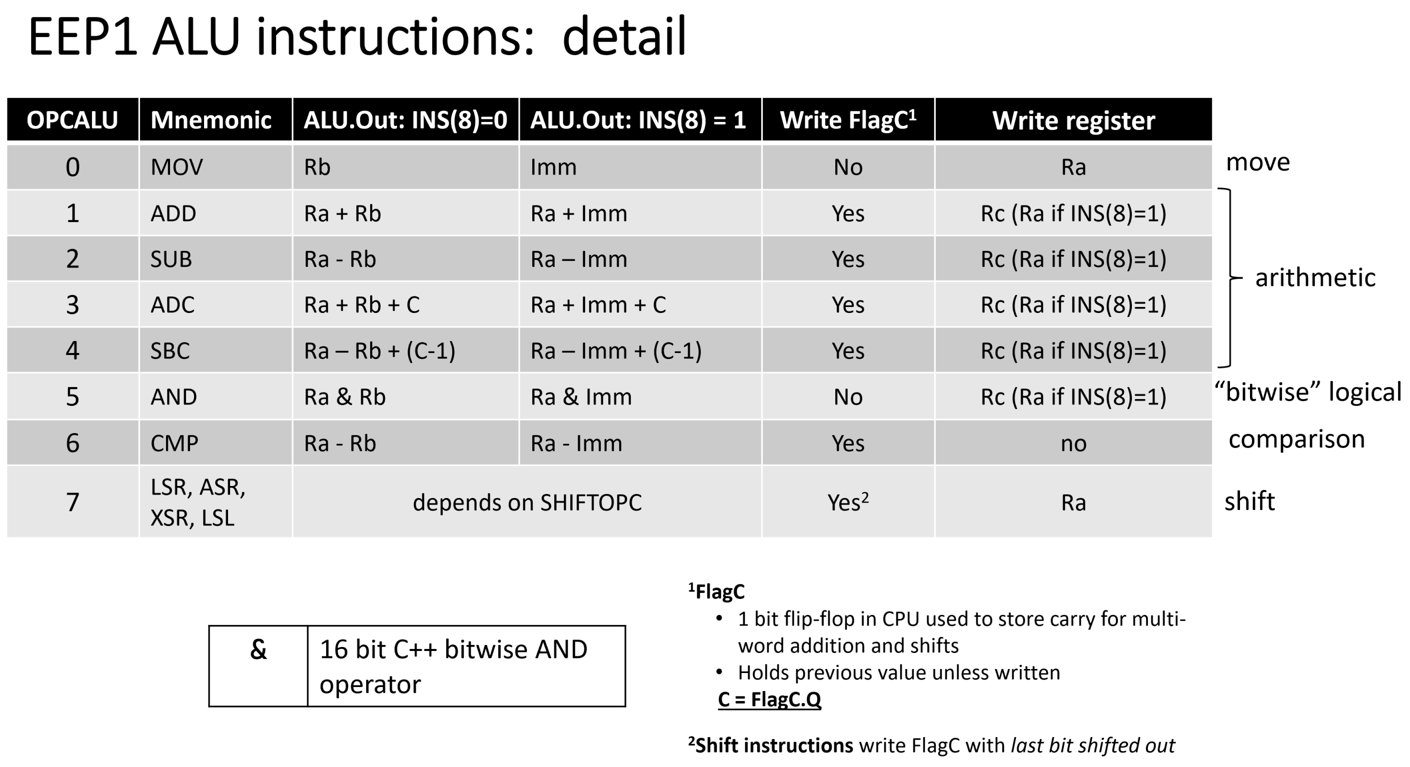
[1]Instruction does not use Rc, this field is (0)

SHIFTOPC(1:0)	Shift
0	LSL
1	LSR
2	ASR
3	XSR

Legend	Meaning
Imm4	4 bit unsigned immediate
Imms8	8 bit signed immediate
(0)	Must be 0 for current instructions, non-zero values are reserved for expansion

EEP1 machine code	INS bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	ALU	0	ALUOPC=7			a			Shift-opc(1)	b			Shift-opc(0)	Imm4 (SCNT)			
	ALU	0	ALUOPC=0..6			a			0	b			c			(0)	
	ALU	0	ALUOPC=0..6			a			1	Imms8 (IMM)							

Figure 7: EEP1 ALU Instructions and Machine Code



EEP1 ALU instructions: detail

OPCALU	Mnemonic	ALU.Out: INS(8)=0	ALU.Out: INS(8) = 1	Write FlagC[1]	Write register	
0	MOV	Rb	Imm	No	Ra	move
1	ADD	Ra + Rb	Ra + Imm	Yes	Rc (Ra if INS(8)=1)	arithmetic
2	SUB	Ra - Rb	Ra – Imm	Yes	Rc (Ra if INS(8)=1)	arithmetic
3	ADC	Ra + Rb + C	Ra + Imm + C	Yes	Rc (Ra if INS(8)=1)	arithmetic
4	SBC	Ra – Rb + (C-1)	Ra – Imm + (C-1)	Yes	Rc (Ra if INS(8)=1)	arithmetic
5	AND	Ra & Rb	Ra & Imm	No	Rc (Ra if INS(8)=1)	“bitwise” logical
6	CMP	Ra - Rb	Ra - Imm	Yes	no	comparison
7	LSR, ASR, XSR, LSL	depends on SHIFTOPC		Yes[2]	Ra	shift

&	16 bit C++ bitwise AND operator

[1]**FlagC**
- 1 bit flip-flop in CPU used to store carry for multi-word addition and shifts
- Holds previous value unless written

C = FlagC.Q

[2]**Shift instructions** write FlagC with *last bit shifted out*

Figure 8: Simulating EEP1 ALU instruction detailed operation