DECA Lab Spring Term

Part 2: EEP1 Control Path and Jump Instructions

Department of Electrical and Electronic Engineering

Imperial College London

v2.01

Spring 2026

Contents

1 Introduction

This lab investigates how the EEP1 *control path* determines which instruction to execute next. In Lab 1 you examined how the datapath executes ALU instructions; here you will see how the program counter (PC), flags, and jump logic interact to support conditional and unconditional control flow.

Before the lab

- Download the `eep1lab2` design from the `lab2-2025` directory.
- Read Spring Lecture 4.

Figure 2 summarises all of the EEP1 jump instructions with their machine-code encodings. A jump instruction directly updates the program counter (PC) so that execution continues at a different address. Depending on the instruction, this update may be:

- *unconditional*, or
- *conditional* on one or more of the EEP1 flags.

In Section 2 you will analyse how the EEP1 control path (Figure 3) implements these jump instructions in hardware. In Section 3 you will add missing hardware to the Lab 2 design to support all required jump conditions. Finally, in Sections 4 and 5 you will use jump instructions to implement software multiply routines and explore possible optimisations.

2 Control Path Operation

Objectives

In this section you will:

- examine how the program counter (PC) selects the next instruction,
- understand how flags are stored and used for conditional jumps,
- relate control-path signals to the NEXT block behaviour.

Program Counter

Figure 3 shows the schematic of the controlpath block, which contains the EEP1 Program Counter (PC). The output PC.Q is connected to the instruction memory address MEMADDR, and therefore selects which instruction word MEMDATA = INS(15:0) is currently being executed.

The input PC.D must carry the address of the *next* instruction. This input is driven by NEXT.PCNEXT, meaning that the NEXT block decides whether the control path performs a normal sequential increment or a jump.

Flags

The control path also contains four 1-bit flag flip-flops: FFN, FFZ, FFC, FFV. Their Q outputs form the four EEP1 flags FlagN, FlagZ, FlagC, FlagV, which are inputs to the NEXT block. In addition, FlagC is forwarded to the datapath where it may be used as a carry-in for ALU operations.

The D inputs of the flag flip-flops come from the datapath and are updated according to the flag results of the current instruction. The Q outputs represent the stored flag bits N, Z, C, V, which are Boolean values used by conditional jump instructions (for example, JEQ will only jump if Z = 1).

The flag flip-flops introduce a one-cycle delay between their D and Q signals. This delay allows information computed by one instruction to be used by a conditional jump in the following instruction.

The block controldecode produces the signals NZEN and CVEN, which determine, for the current instruction, whether (FlagN,FlagZ) or (FlagC,FlagV) should be updated. The ALU instruction specification defines which instructions update which flags (for example, MOV updates only FlagN and FlagZ, whereas ADD updates all four flags).

From this overview, note that the overall behaviour of the control path depends critically on the NEXT block. It receives inputs from the flags and from the current instruction (via controldecode), and produces a single output PCNEXT that determines the next program counter value.

2.1 NEXT block

Figure 4 shows the NEXT sheet. The logic that controls PCNEXT includes multiplexers MUX1 and MUX2, driven by the control signals COND.RET, COND.JUMP, and NEXT.JMP. Before attempting Task 1, draw by hand an algebraic truth table showing how these three signals, together with the algebraic inputs NEXT.PC, NEXT.OFFSET, and NEXT.RA, determine the value of PCNEXT.

□ **Task 1.** *Create an 8-row ISSIE algebraic truth table, selecting components as shown in Figure 5, to examine how the NEXT sheet drives NEXT.PCNEXT. Compare the truth table with your hand-generated version; they should match. You may assume that NEXT.JMP = 1 only for jump instructions. What is the value of PCNEXT when NEXT.JMP = 0? Compare your results with the EEP1 jump instructions in Figure 2 and explain how the signals COND.JUMP and COND.RET influence the control path behaviour.*

3 Jump Conditions

Objectives

In this section you will:

- examine how the `cond` sheet evaluates jump conditions from the EEP1 flags,
- verify that existing jump conditions match the EEP1 specification,
- implement the remaining missing jump conditions in hardware.

Open the `cond` sheet in your Lab 2 design. Note that the signal `JMPCOND(3:0)` is driven directly from the instruction field `JMPOPC(3:0)` in the current instruction word. Figure 2 defines the meaning of each `JMPOPC` value.

Verification of existing conditions

Before adding new hardware, first verify that the existing logic correctly implements the jump conditions for `JMPOPC(3:1) = 0,1,2,3,7` as specified in Figure 2. One convenient approach is to use an ISSIE algebraic truth-table containing `MUX1` and all logic connected to its inputs, while treating the flag inputs as algebraic values.

□ **Task 2.** *In the `cond` sheet, replace the existing constant '1' connected to the `MUX1` inputs with appropriate logic to implement the remaining jump conditions for `JMPOPC = 4,5,6`. Demonstrate that your logic is correct by constructing an ISSIE algebraic truth-table.*

4 Multiply Software

Objectives

In this section you will:

- review how multiplication can be implemented using shift-and-add,
- translate a structured control-flow algorithm into EEP1 assembly,
- extend your solution to support 32-bit partial multiplication.

The EEP1 ISA does not include a hardware multiply instruction, so multiplication must be implemented in software using combinations of shift and add instructions, with conditional jumps providing control flow. Lecture 3 introduces the shift-and-add method for multiplication and motivates why this works.

Figure 1 shows a simple C++ implementation of multiplication using this technique.

```
0  // implement sum := op1 * op2   (LS 16 bits of)
1  // assume int = 16 bits (16-bit version of C)
2  unsigned int op1, op2, op2_shifted, sum; // variables
3  sum = 0;                  // initialise
4  op2_shifted = op2;        // initialise
5  while (op1 != 0) {
6      if (op1 & 1) {        // & = bitwise AND
7          sum = sum + op2_shifted;
8      }
9      op2_shifted = op2_shifted << 1;  // left shift by 1
10     op1 = op1 >> 1;                  // right shift by 1
11 }
```

Figure 1: Shift-and-add multiplication using a while loop

□ **Task 3.** *Manually trace the code in Figure 1 with* `op1 = 12` *and* `op2 = 5` *to verify that it produces the result* `60`*. Compare its behaviour with the shift-and-add multiply method explained in Lecture 3 to confirm that the algorithm is correct.*

□ **Task 4.** *Using the method from Lecture 4 for compiling* `if-then` *and* `while` *constructs into jumps, and the EEP1 instructions for addition, shifting, and logical AND, translate the program in Figure 1 into EEP1 assembly.*

Use registers `R0`*,* `R1`*,* `R2`*,* `R3` *to hold the four variables* `op1`*,* `op2`*,* `op2_shifted`*,* `sum` *respectively.*

Write your assembly program into a `mul1.txt` *file and use* `eepassembler` *to generate a* `mul1.ram` *machine-code file. Run it on your own (now fully working) EEP1 design in Issie and check that it multiplies correctly.*

□ **Task 5.** *Extend your implementation so that* `op2_shifted` *and* `sum` *are 32-bit quantities, each stored across two registers. Replace the 16-bit add and shift operations by corresponding 32-bit operations using pairs of registers.*

5 Challenge

During Spring Term DECA, successful challenge work may help increase the mark awarded in lab orals and is typically required for high A grades. However, it is entirely possible to obtain an A grade through outstanding understanding and logbook presentation without attempting any challenges. Challenge work is therefore optional. Labs 2 and 3 contain challenge opportunities, and credit may be obtained from either.

Challenge tasks have no single "correct" answer: they are open-ended design and evaluation problems. You may approach them in any form, and pursue more or less of the suggested directions. Credit will be given for design innovation and for the ability to evaluate the merits and limitations of your solution.

5.1 Challenge Overview

The challenge in Lab 2 is to implement additional instruction(s) and/or hardware that speed up EEP1 register multiplication.

5.2 Design Constraints

You are limited to using no more than 64 full adders in total (for example, eight Issie 8-bit adder blocks). If you add multiple copies of a sheet containing new adders, each copy counts separately.

Credit can be obtained from any of the following:

1. The quality of your solution itself (instruction design, hardware structure, software interface).
2. Comparison of speed (in clock cycles) and hardware cost (number of adders) against the "pure software" implementations from Lab 2.
3. Understanding of how your solution can be used in different contexts (for example, implementing signed and unsigned multiplication).

5.3 Design Notes

Speeding up EEP1 multiplication generally requires defining one or more **additional ALU instructions in the ISA**. This is possible because some machine-word encodings are currently unused. Specifically, the standard `MOV` instruction does not use register `c`, and the three bits encoding this field are set to zero for a normal `MOV`.

5.3.1 Implementing unused instructions

There are seven additional encodings available that follow the standard `MOV` format but use a non-zero value in the `c` register field `INS(4:2)`. The assembler recognises these as `MOVCn Ra, Rb` for $n = 1..7$ and generates the appropriate machine code. You may use any of these instructions to interface with additional hardware. For example, `MOVC1 Ra, Rb` could be defined to execute `Ra := Ra op Rb`, where `op` is a new operation implemented by your design.

5.3.2 Adding to the datapath hardware

You may extend the datapath by adding logic and new output signals from `DPDECODE` to control additional multiplexers. These multiplexers can (for the `MOVCn` instructions you decide to implement) select the output of your hardware block(s) instead of the ALU. Your hardware can live on a separate sheet, which you then instantiate as a component inside the datapath.

Use hierarchy to simplify your design (and to speed up simulation and testing). The provided `shift` block is a good example. Be creative: you do not need to follow any template exactly. Reuse hardware where possible, especially when related operations can share structure.

5.3.3 Using your new hardware

Once your hardware and instructions are defined, explore how many cycles are required for multiplication using your approach. Consider instruction sequences that minimise control overhead. Also consider whether small hardware variations could further accelerate the software interface.

5.4 Optional Collaboration Challenge: IEEE 754 half-precision (binary16) Floating Point

This option has not yet been attempted and is challenging but achievable for two students working as a pair: one focusing on software and one on hardware.

For additional credit, you may combine hardware and software to produce a fast IEEE-754 binary16 multiply routine. To simplify the task, you may assume that *subnormal* and *special* (infinite or NaN) values do not occur as input or output. The standard normalised sign–exponent–mantissa representation is thus always used, with zero as the only special case.

IEEE-754 specifies detailed rounding rules. For this challenge, you do not need to implement correct rounding as long as the result is one of the two closest representable floating-point numbers to the exact product.

For your interest, a complete floating-point library would combine multiply with similar routines for addition and subtraction. Division (or more precisely reciprocal, from which division can be derived) can be implemented using multiplication and a Newton–Raphson method.

EEP1 Jump instructions

EEP1 machine code	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	0	0	JMPOPC				Imms8 (JOFFSET)							

JMPOPC(3:1)	NZCV condition	Condition on data for jump with JMPOPC(0)=0	JMPOPC(0) = 0	JMPOPC(0) = 1[3]
0	1	always	**JMP** (Always jump)	**NOP** (Never jump)
1	Z	result = 0	**JEQ** (Equal)	**JNE** (Not Equal)
2	C	C=1 ≡ u(Ra) ≥ u(Op)[1]	**JCS** (Carry Set / Unsigned Higher or Equal)	**JCC** (Carry Clear / Unsigned Lower)
3	N	z(result) < 0	**JMI** (Minus)	**JPL** (Plus)
4	$\bar{N} \oplus V$	z(Ra) ≥ z(Op)[1]	**JGE** (Signed Greater or Equal)	**JLT** (Signed less than)
5	$(\bar{N} \oplus V).\bar{Z}$	z(Ra) > z(Op)[1]	**JGT** (Signed Greater Than)	**JLE** (Signed Less than or Equal)
6	$C.\bar{Z}$	u(Ra) > u(Op)[1]	**JHI** (Unsigned Higher)	**JLS** (Unsigned Lower or Same)
7	1	always	***JSR*** *(R7[4] := PC + 1, PC := PC + JOFFSET)*	***RET*** *(PC := R7)*

[1] **JGE,JGT,JHI, JCS** jump conditions jump based on the flag values. The table column 3 shows the comparison of **Ra** and **Rb** after a previous `SUB Ra, Rb` instruction which delivers those NZCV values, or the condition on an ALU result which delivers the required NZ values.

[2] **JSR** and **RET** always jump and are used to *jump to* and *return from* subroutine see notes. **RET** does not use **Joffset**.

[3] **JMPOPC(0) = 1** inverts all conditions, except for JSR/RET

[4] Hardware is simpler because **JSR/RET** write/read R7 since **Ra** is normally read and written and for these instructions **a** = **JMPOPC(3:1)** = 7

Jump Instructions	Assembly	Operation	Examples
All except RET, JSR	JEQ JOFFSET; JEQ SYMBOL	PC := PC + JOFFSET if condition is true	`JMP -1 ; JGT LOOP ; JSR SUB1`
RET	RET	PC := R7	`RET`
JSR	JSR JOFFSET	R7 := PC+1; PC := PC + JOFFSET	`JSR 10`

Figure 2: EEP1 Jump Instructions

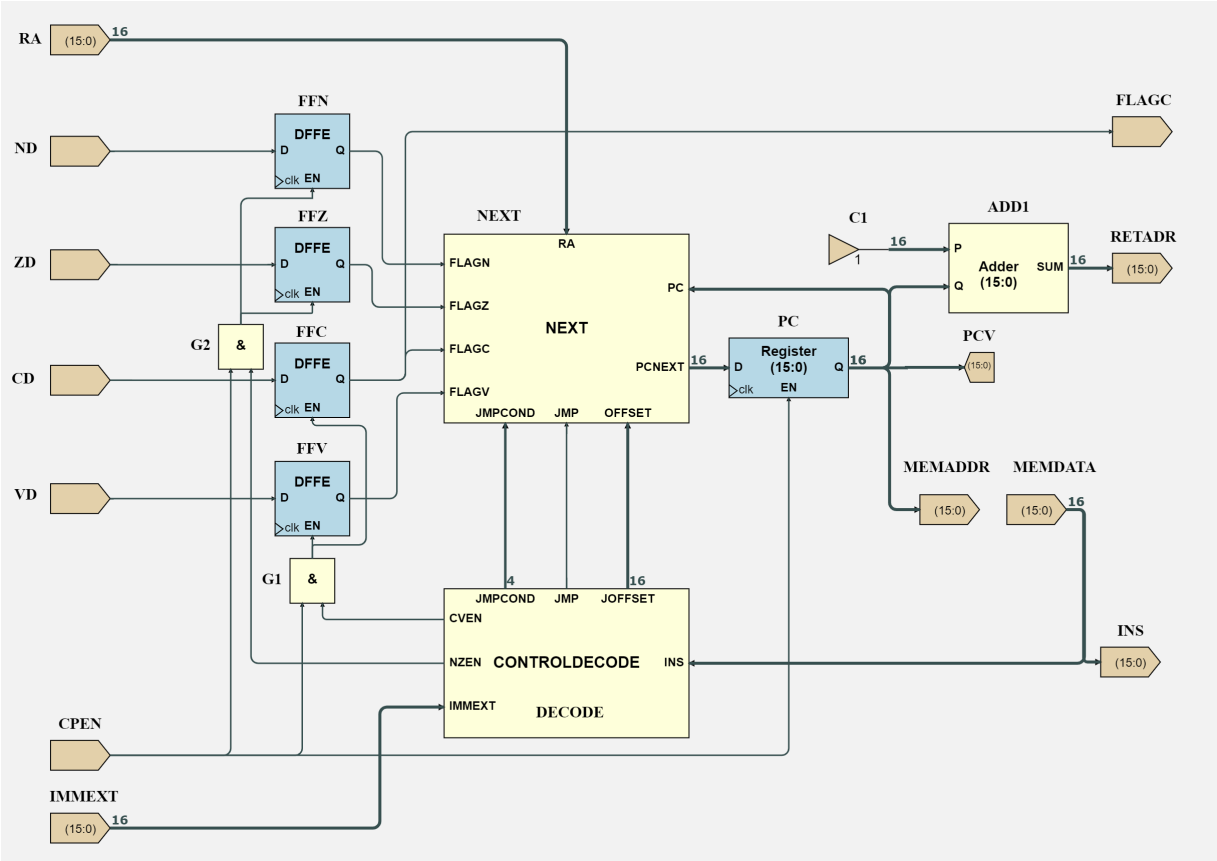

Figure 3: EEP1 Control Path

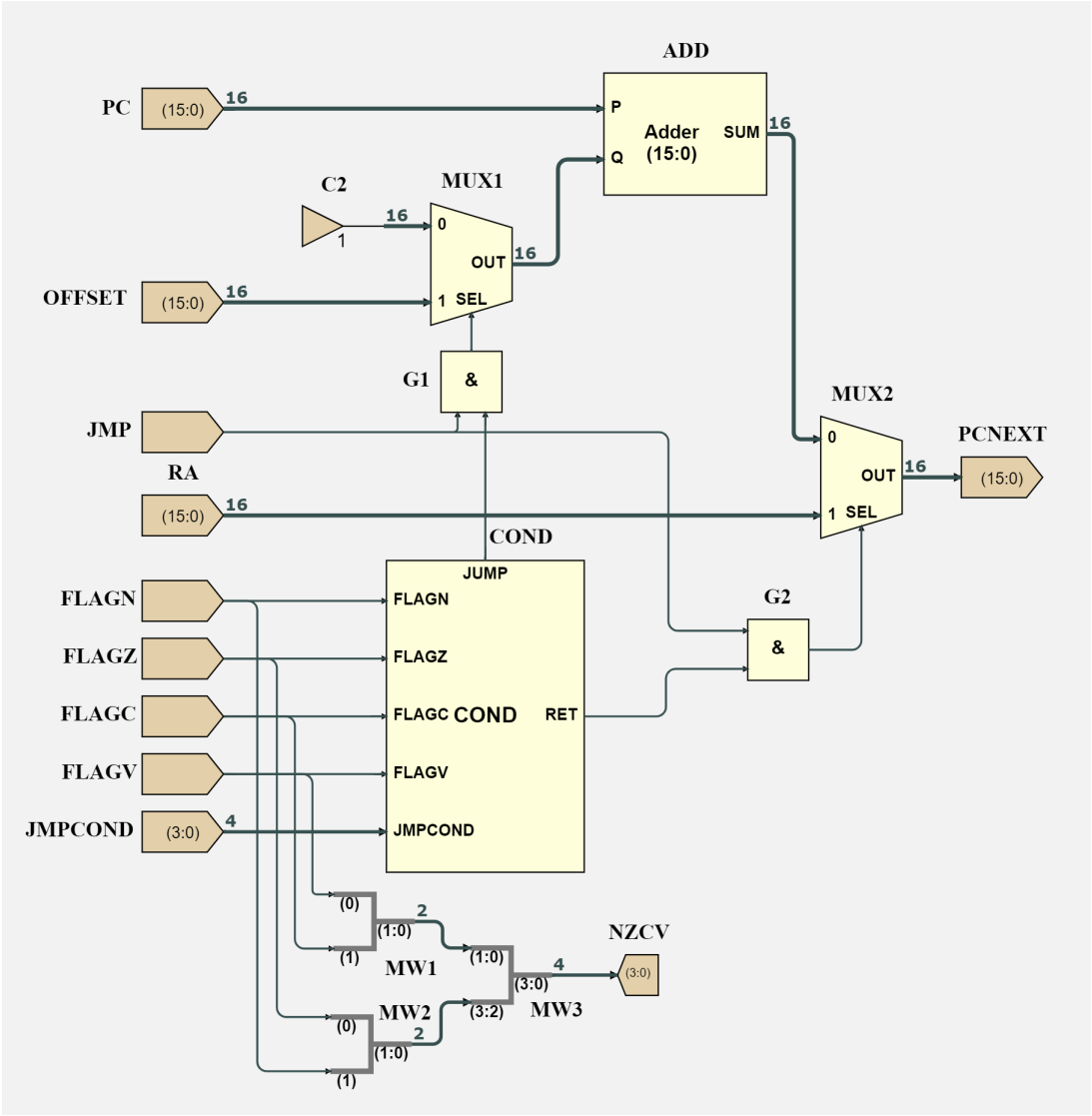

Figure 4: EEP1 Next Block

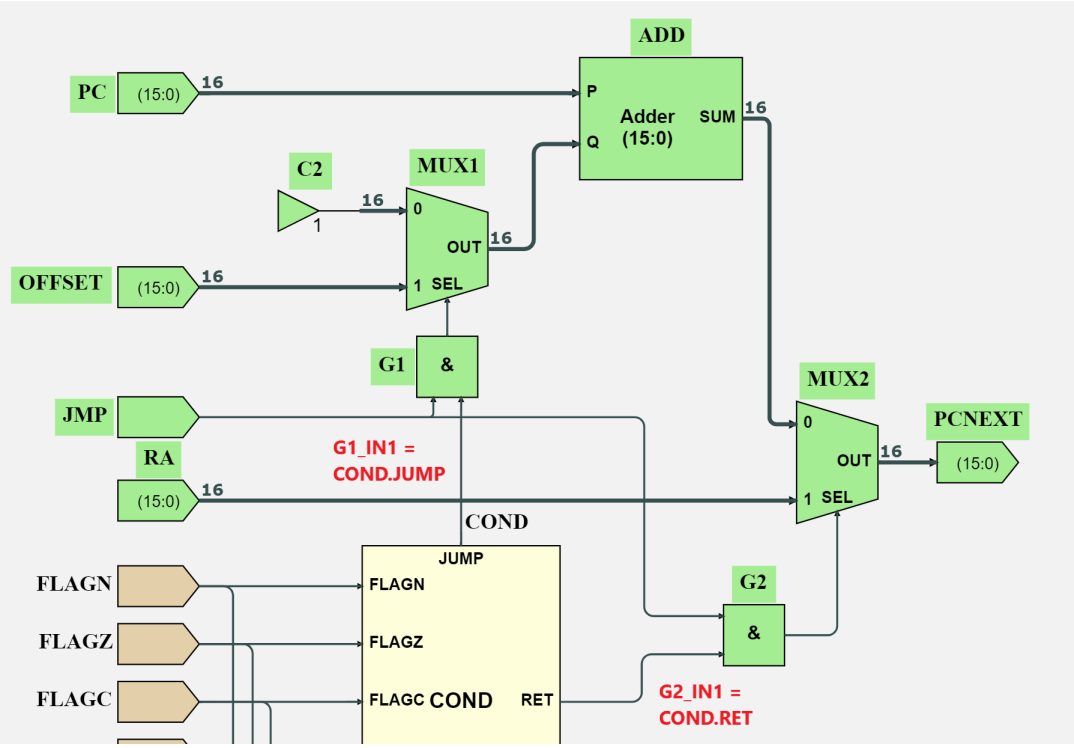

Figure 5: Component Selection to generate PCNEXT truth table: red text indicates resulting truth table column headers