DECA Lab Spring Term

Part 2: EEP1 Control Path and Jump Instructions

Department of Electrical and Electronic Engineering

Imperial College London

v2.00

Spring 2025

Contents

1 Introduction

Before the lab

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

Figure 2 shows all of the EEP1 jump instructions, with their machine code encoding. A jump instruction directly modifies the program counter to execute a different part of the program, either unconditionally, or conditional on some boolean condition on the EEP1 flags. In Section 2 you will focus on how the EEP1 control path in Figure 3 implements these instructions. In section 3 you will add hardware to the Lab2 design to implement the required jump conditions. Finally in sections 4 and 5 you will use jump instructions to implement multiply routines.

2 Control Path Operation

Look at the schematic of the `controlpath` block in Figure 3 and note that this block contains the EEP1 Program Counter (PC). As you can see from this sheet, PC.Q is output to the instruction memory address MEMADDR and determines the memory location of the current instruction word (MEMDATA=INS(15:0)). PC.D therefore must be the address of the **next** instruction and it is output from NEXT.PCNEXT. The NEXT block therefore determines if and where the EEP1 will jump.

The control path contains 4 1-bit flag flip-flops: FFN, FFZ, FFC, FFV. The Q outputs of these are input to the NEXT block as the 4 Flags: FlagN, FlagZ, FlagC, FlagV. In addition FlagC is output to the datapath so it can be used as an adder input in ALU operations. The D inputs of these flip-flops come from the datapath and are set according to the flag conditions. The Q outputs of these flipflops are the *flag bits* N, Z, C, V: boolean values used to determine whether a jump will occur, e.g. a JEQ instruction will only jump if Z=1.

The flag flipflops introduce (between D and Q) a one cycle delay which allows information about data in one instruction to be used in a conditional jump in the next instruction. The `controldecode` block has outputs NZEN and CVEN that determine, based on the current instruction, whether FlagN and FlagZ, or FlagC and FlagV, should be overwritten. The ALU instruction specification states which instructions write which flags: for example, MOV will write only FlagN and FlagZ, whereas ADD will write all of the flags.

From this overview you can see that the operation of the control path depends on NEXT; this has inputs from the flags and (via `controldecode`) the instruction word. It has just one output: PCNEXT.

2.1 `Next` block

Figure 4 shows the `next` sheet. The logic that controls PCNEXT contains MUX1 and MUX2 controlled by COND.RET, COND.JUMP, and NEXT.JMP. Before doing Task 1, draw up by hand from the given hardware a truth table for how these 3 signals, and 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 truth table - they should be the same. You may assume that NEXT.JMP is 1 only for jump instructions. What is PCNEXT when NEXT.JMP = 0? Compare the truth-table with the EEP1 jump instructions (Figure 2) to determine how inputs `COND.JUMP` and `COND.RET` affect the control path operation.*

3 Jump Conditions

Open the `cond` sheet from your Lab1 design in Issie. Note that JMPCOND(3:0) is equal to JMPOPC(3:0) from the current instruction word. Check that the conditions for JMPOPC(3:1)=0,1,2,3,7 are the same as those required by Figure 2 in logic already on the `cond` sheet. You may conveniently do this using an Issie Truth-table selecting MUX1 and all of the logic connected to its inputs, and then setting the Flag inputs to be algebraic.

□ **Task 2.** *In the `cond` sheet replace the existing constant '1' connection at MUX1 inputs by appropriate logic to implement the jump conditions for JMPOPC = 4,5,6. Show that your logic is correct by using an Issie algebraic truth-table.*

4 Multiply Software

The EEP1 does not have a multiply instruction, and therefore multiply operations must be implemented in software using combinations of shift and add instructions, with the correct jumps to implement control flow. Lecture 3 provides context, showing how shifts and addition can implement multiplication.

Figure 1 shows an algorithm for multiplication written in C++.

```
// implement sum := op1 * op2 (LS 16 bits of)
// assume int = 16 bits (16 bit version of C).
unsigned int op1, op2, op2_shifted, sum; // variables
sum = 0; // initisalise
op2_shifted = op2; //initialise
while (op1 != 0) {
    if (op1 & 1) { // & bitwise AND operation
        sum = sum + op2_shifted;
    }
    op2shifted = op2shifted << 1; // left shift by 1
    op1 = op1 >> 1; // right shift by 1.
}
```

Figure 1: While loop implementation of multiply

☐ **Task 3.** *Walk through the code in Figure 1 by hand with op1=12, op2 = 5 to check that it works correctly and generates 60 as an answer. Compare its operation with the shift and add multiply method explained in Lecture 3 to show how it works.*

☐ **Task 4.** *Using the method of Lecture 4 for* `if-then` *and* `while` *constructs, and the EEP1 instructions for addition, shifting, logical AND, translate the program in Figure 1 into assembly language, using registers R0, R1, R2, R3 for the 4 required variables. Write your program in a* `mul1.txt` *file and use eepassembler to turn this into a* `mul1.ram` *file containing machine code. Run the file using your own EEP1 design now fully working and check that it multiplies.*

☐ **Task 5.** *Write a similar multiply program so that* `op2shifted` *and* `sum` *are both 32 bits, made up of each of two registers. Replace the 16 bit add and shift operations by a corresponding 32 bit add and shift operation.*

5 Challenge

In Spring Term DECA successful Challenge work may help increase the mark awarded in lab orals and will be required for high A grades; however, it will be possible to obtain an A grade from outstanding understanding and logbook presentation of the labs without attempting any challenges: thus Challenge work is optional. Labs 2 and 3 will have Challenges; credit can be obtained from either of these.

Challenge work has no ”right” answer and consists of an open design and evaluation problem: You may tackle this in any form and try more or less of the suggested solution. Credit will be given for design innovation and the ability to evaluate the merits of your design.

5.1 Aims

The challenge is to implement additional instruction(s) and/or hardware that can speed up EEP1 register multiplication. You are limited to no using more than 64 full adders: for example 8 Issie adder blocks of 8 bits each. Note that if you add multiple copies of a design sheet containing new adders to your design each copy counts separately.

Credit can be obtained from any of the following:

1. The solution itself
2. Comparing the speed of your solution in clock cycles, and its cost in hardware adders, with the ”pure software” implementations in Lab2.
3. Knowing how your solution can be used in different contexts: for example, to implement signed and unsigned multiplication.

5.2 Design Notes

Speeding up EEP1 multiplication requires the definition of **additional ALU instructions in the ISA**. This is possible because some combinations of bits in the machine work are not currently used. Specifically the `MOV` instruction does not use register C, the three bits specifying this are set to 0 for a normal `MOV`.

5.2.1 Implementing unused instructions

There are 7 *additional* MOV instructions, using machine code as for `MOV` with with nonzero in the `c` register field `INS(4:2`, can be used. The assembler recognises `MOVCn Ra Rb` where $n = 1..7$ and generates the correct machine code. You may use any of these instructions in Lab 2 to interface with additional hardware. For example `MOVC1 Ra, Rb` could be used to implement Ra := Ra op Rb, where `op` is some new operation you have defined.

5.2.2 Adding to the datapath hardware

You can implement additional logic in and outputs from `DPDECODE` to control additional MUXes in the datapath. These MUXes can (only for the `MOVCn instructions you decide to implement` select the output of your hardware block(s) instead of the ALU. You can put your hardware on a separate sheet which you add as a component to the datapath sheet.

Use hierarchy to simplify your hardware design (this will also speed up design and testing!). As an example, look at the `shift` block you are given as part of EEP1. Be creative: you do not have to follow the examples in the notes exactly. Reuse hardware whenever possible when two different and related operations need to be implemented.

5.2.3 Using your new hardware

Try to work out optimal sequences of instructions. Consider whether slightly different hardware could speed up the software.

5.3 Collaboration: IEEE 754 half-precision (binary16) Floating Point

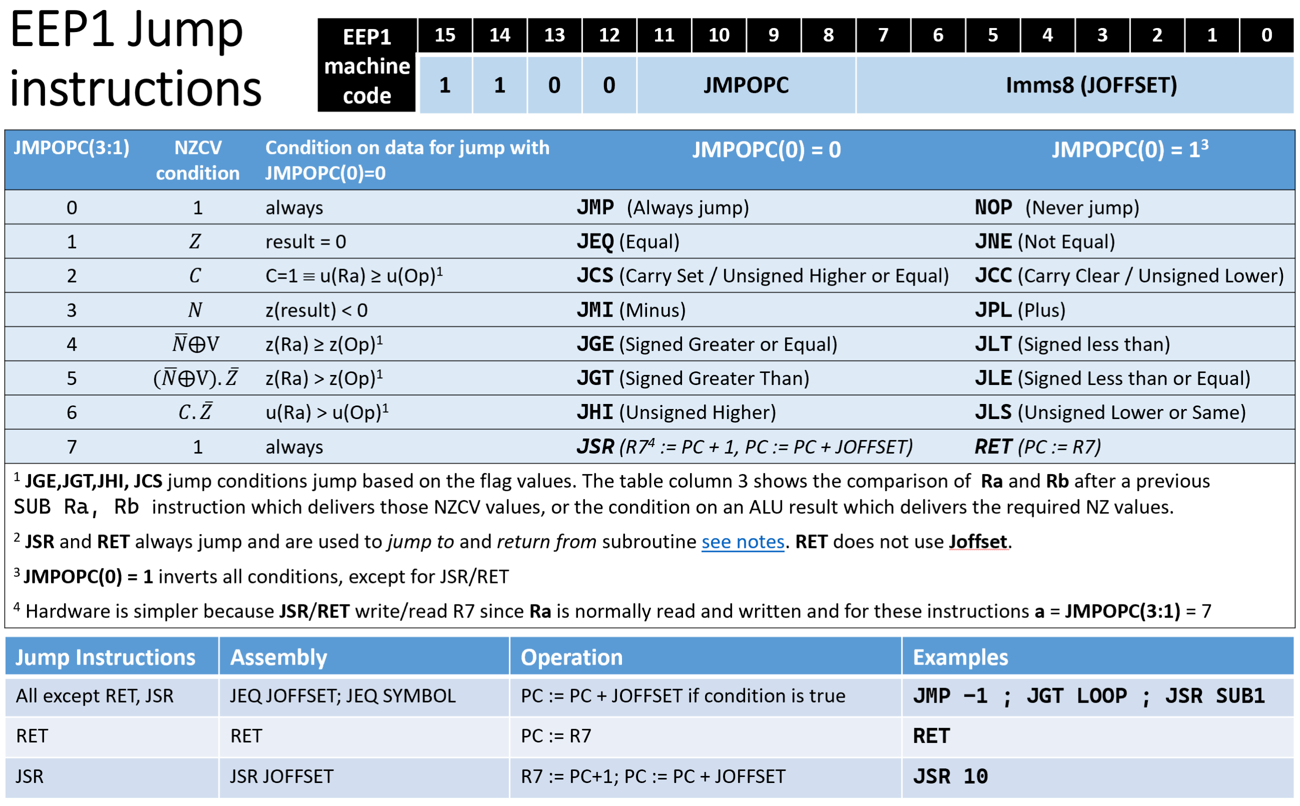
This option has never yet been attempted and is challenging but achievable by two students working as a pair: one on software and the other on hardware.

For additional credit, optionally combine hardware and software to make a fast IEEE-754 binary16 multiply routine. This is quite a challenge. To simplify the task, you may assume that *subnormal* and *special* (infinite or NaN) numbers do not occur as input or output. The standard normalised sign-magnitude-exponent coding described in the reference is thus always used together with, as a single special case, zero. IEEE-754 specifies detailed rounding requirements. For this challenge, you need not implement correct rounding as long as the result is one of the two closest possible floating-point numbers to the exact value.

For your interest, a complete floating-point library would use this multiply routine and similar routines for addition and subtraction. Division (or more accurately, reciprocal from which division can be calculated) can be implemented using multiplication and a Newton-Raphson algorithm. Such a complete library can therefore be implemented quite efficiently in software given only a fast multiply algorithm.

Challenge

Implement new instruction(s) in the EEP1 ISA to speed up multiplication.
If you have time, tackle the IEEE-754 challenge.



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	$\overline{N} \oplus V$	z(Ra) ≥ z(Op)[1]	**`JGE`** (Signed Greater or Equal)	**`JLT`** (Signed less than)
5	$(\overline{N} \oplus V).\overline{Z}$	z(Ra) > z(Op)[1]	**`JGT`** (Signed Greater Than)	**`JLE`** (Signed Less than or Equal)
6	$C.\overline{Z}$	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

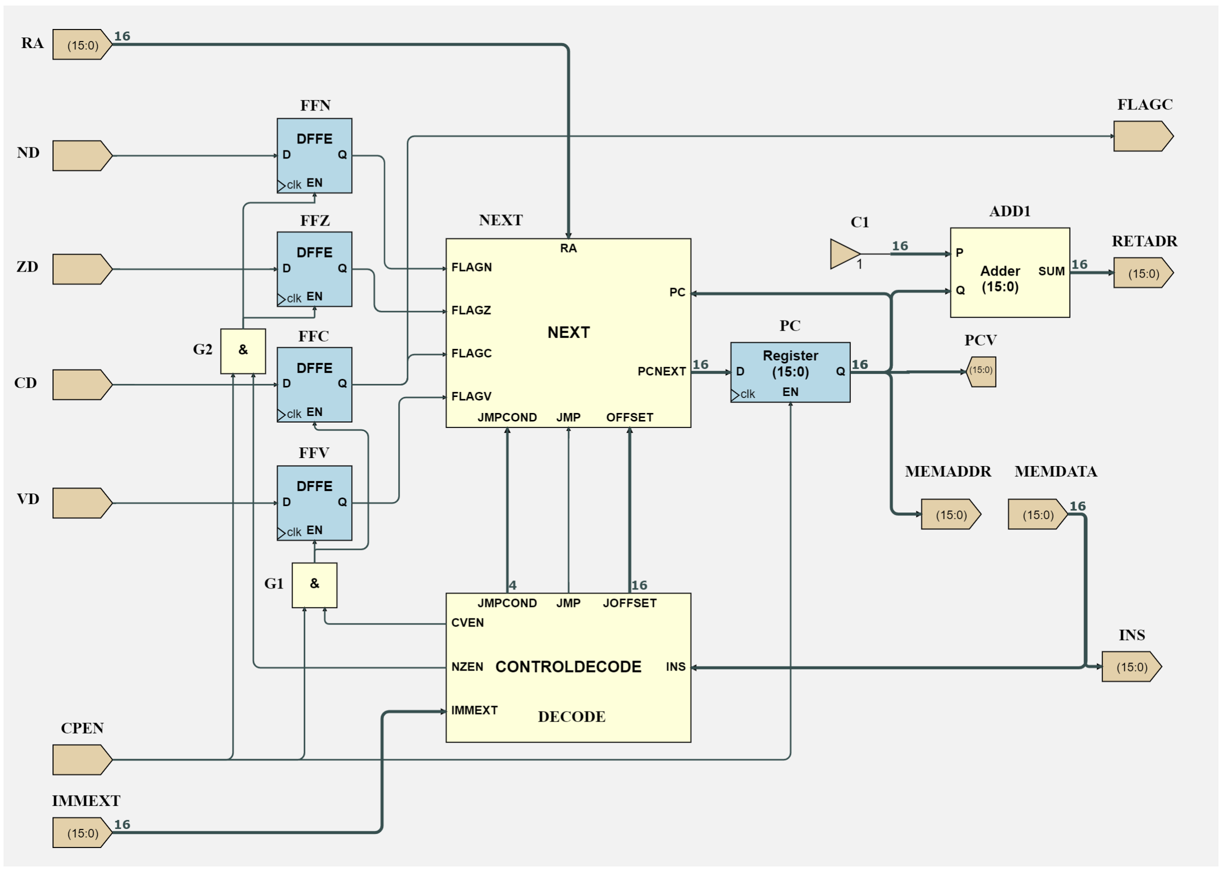


Figure 3: EEP1 Control Path

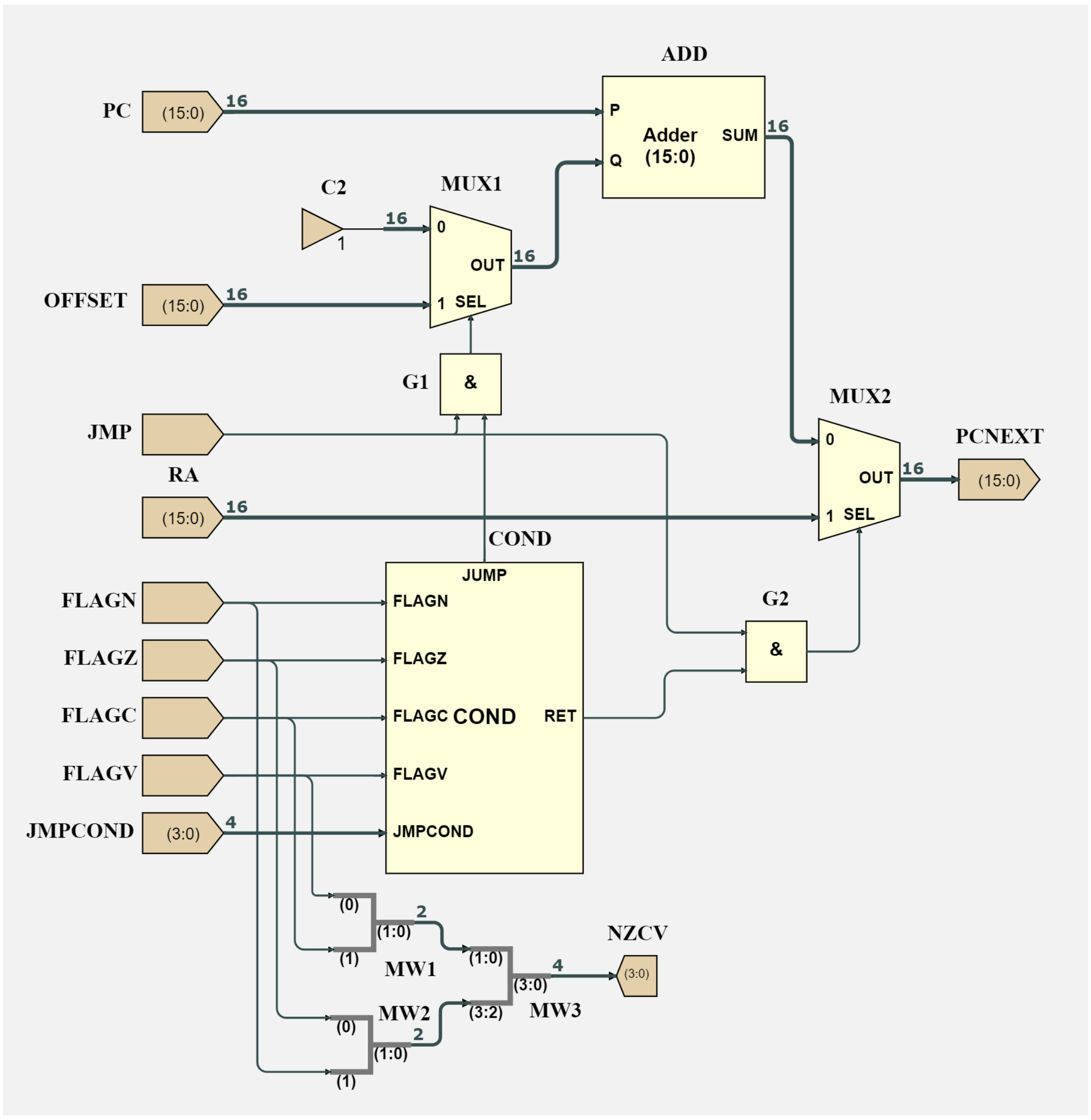


Figure 4: EEP1 Next Block

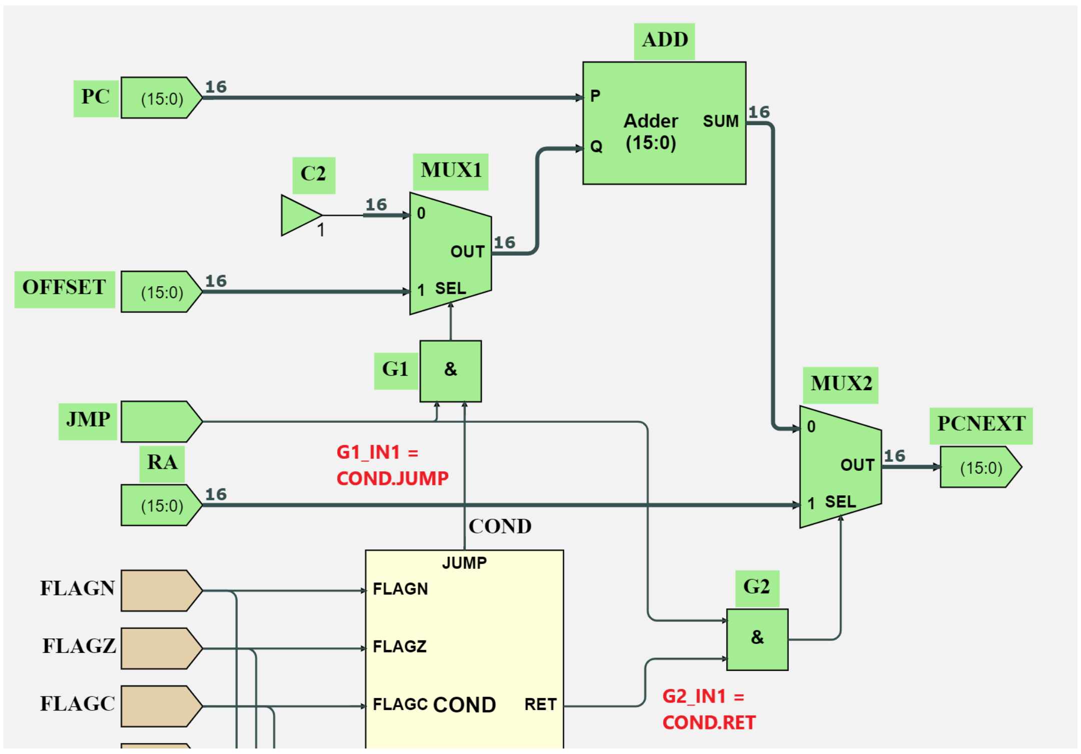


Figure 5: Component Selection to generate PCNEXT truth table: red text indicates resulting truth table column headers