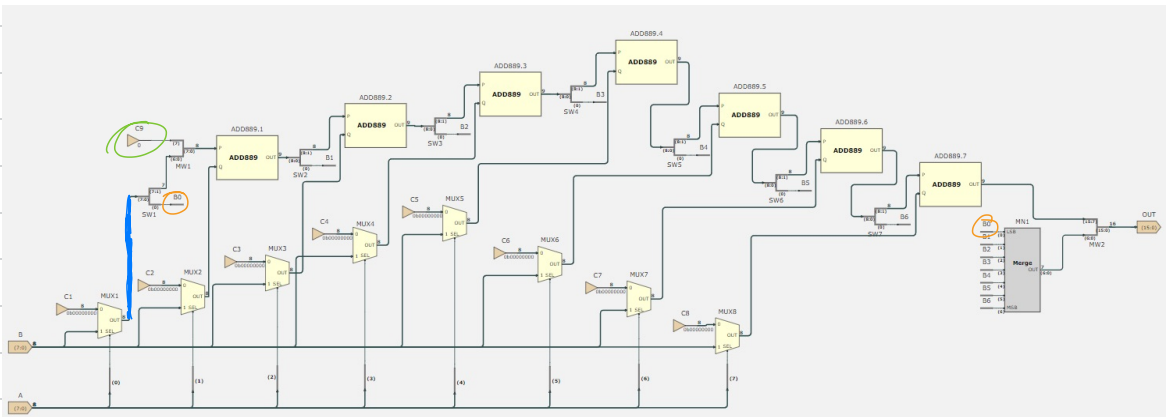
DECA CHALLENGE LAB 2

- For this challenge, I first implemented a 8bit X 8bit multiplier using the design in lecture 3.
- Extending it from 3x3 → 8x8.



This basically does the following:

takes two inputs A, B (8 bits):

suppose A = 01101110
B = 00111010

```
  0 0 1 1 1 0 1 0
x 0 1 1 0 1 1 1 0
-----------------
  0 0 0 0 0 0 0 0
+ 0 0 1 1 1 0 1 0   ← LSB of result
-----------------
```

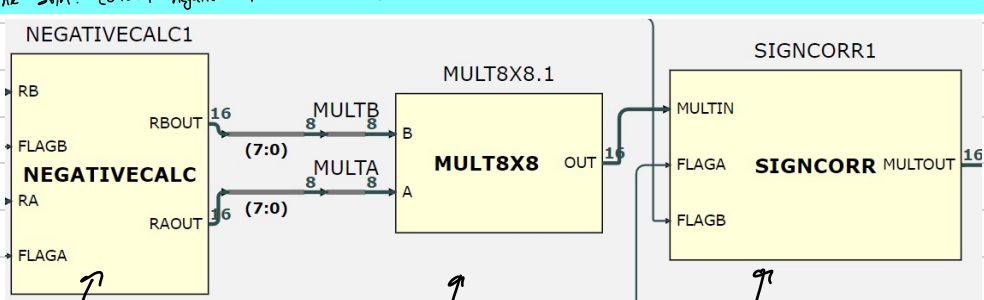
(first stage of the multiplier)

it checks every bit of A (using a 2-mux) to see if it is 0 or 1. If it is 0, then 0 is added to the sum. If it is 1, it will be added to the sum, shifted by its position in A.

this works fine for positive operands ONLY

for negative operands, we need to find an alternate soln.

- One soln: convert negative operand into a positive number → do multiplication → work out the sign of the result.



converting all operands to positive values.

performing 8x8 positive multiplication

restoring correct sign to result.

- In order to implement this, I defined two new FLAGS. (FLAGA, FLAGB).

- These work similar to FLAGN, except it is only written in combination of two instructions

MOVC$_2$ R_a, R_b
MOV R_a, R_b/#IMM
(to write FLAGA)

OR

MOVC$_3$ R_a, R_b
MOV R_a, R_b/#IMM
(To write FLAGB)

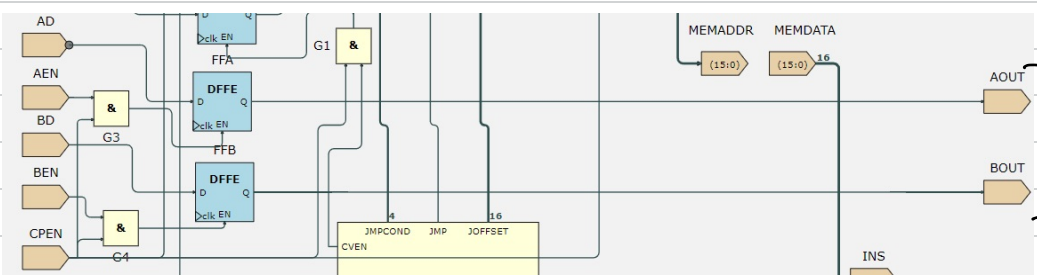
→ MOVC$_2$ initates writing to FLAGA, whereas MOVC$_3$ initiates writing to FLAGB.

→ FLAG A Stores the SIGN of the first Operand

→ FLAG B Stores the SIGN of the second operand.

FLAGA / FLAGB

FLAGA and FLAGB are stored in the controlpath (like FLAGS NZCV)

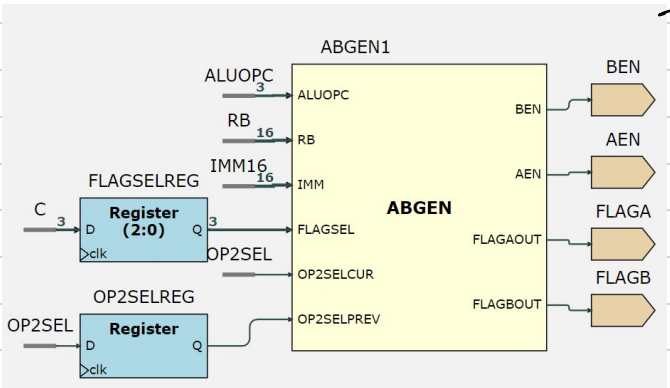


AEN, BEN enable writing to FFA and FFB, if CPEN is HIGH (const. = 1)
These are set in the clockcycle following MOVC$_2$ or MOVC$_3$ (Enabling writing)

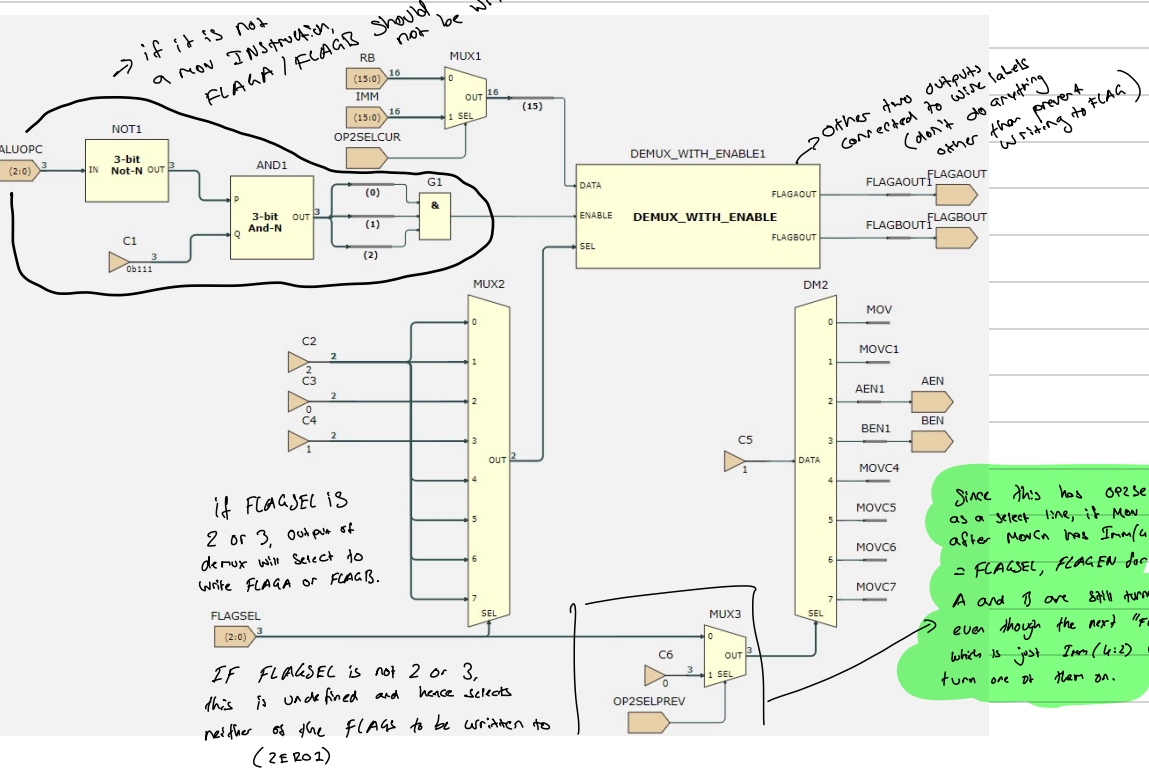
What Value FLAGA / FLAGB are written to is determined by Block: ABGEN

ABGEN : Block Explanation

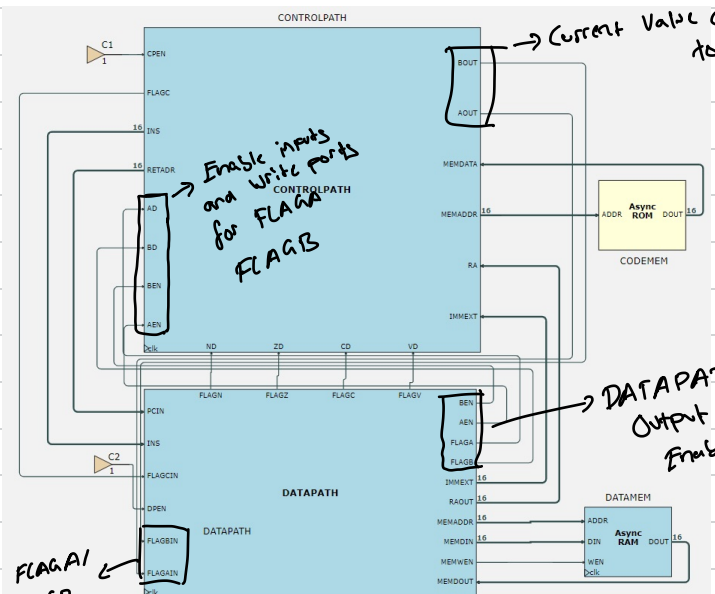
→ top level diagram of ABGEN



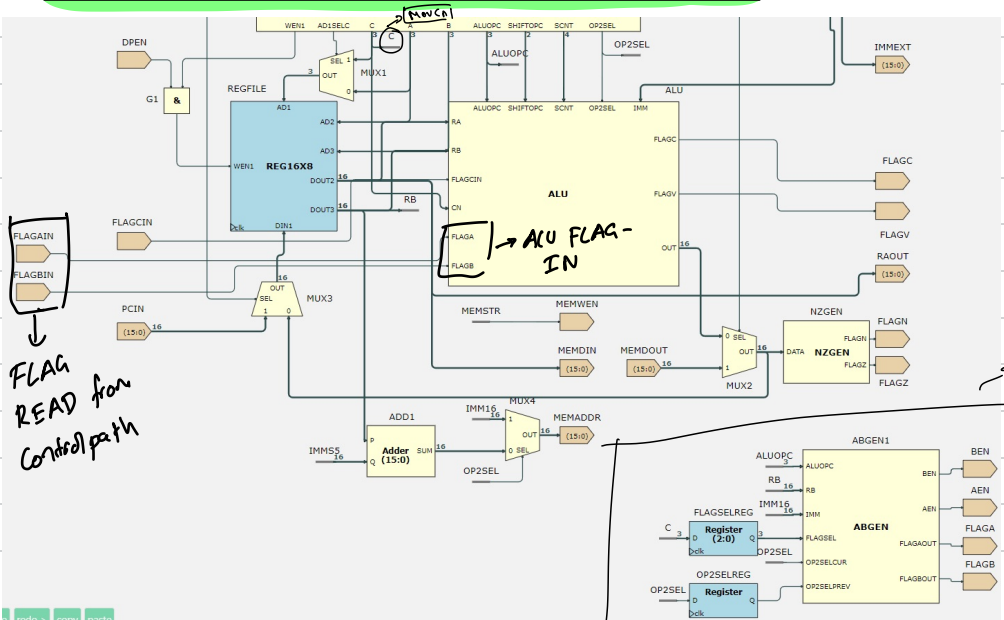
Rc (FLAGSEL) is delayed by 1 clock cycle such that when FLAGA/FLAGB is set in MOV Ra, Rb/#Imm Instruction, INS(4:2) of MOVcn clk cycle is used.



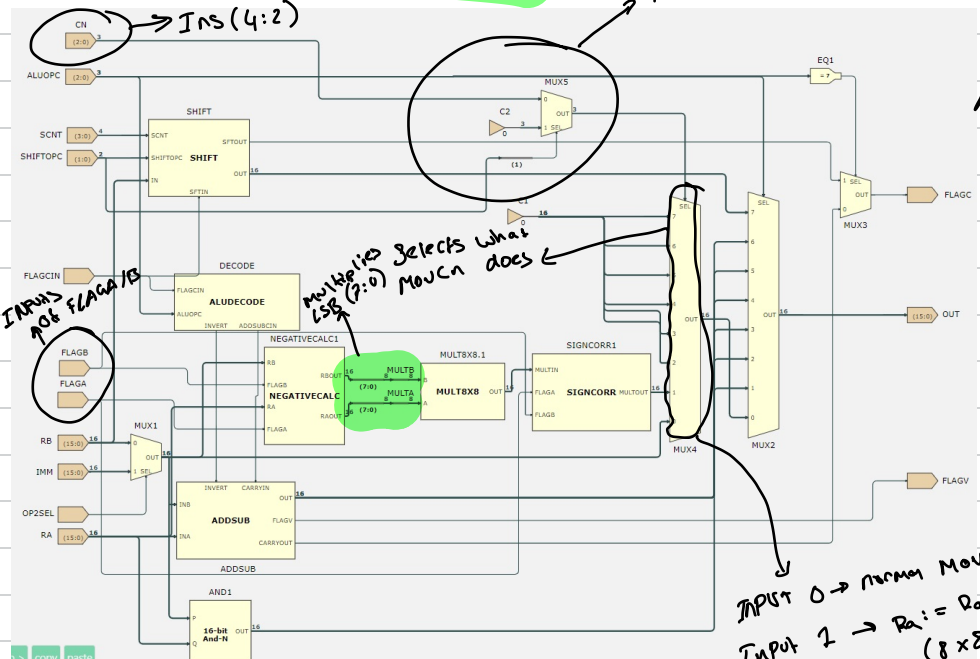
EEP1 IMPLEMENTATION



DATAPATH IMPLEMENTATION



ALU IMPLEMENTATION



This MUX checks if the MOV INS has an Immediate. If so, it automatically selects standard MOV as input to ALUOPC = $(000)_2$ output.

(or else IMM can be falsely interpreted as a MOVCN ins, and MOV INS will be wrong.

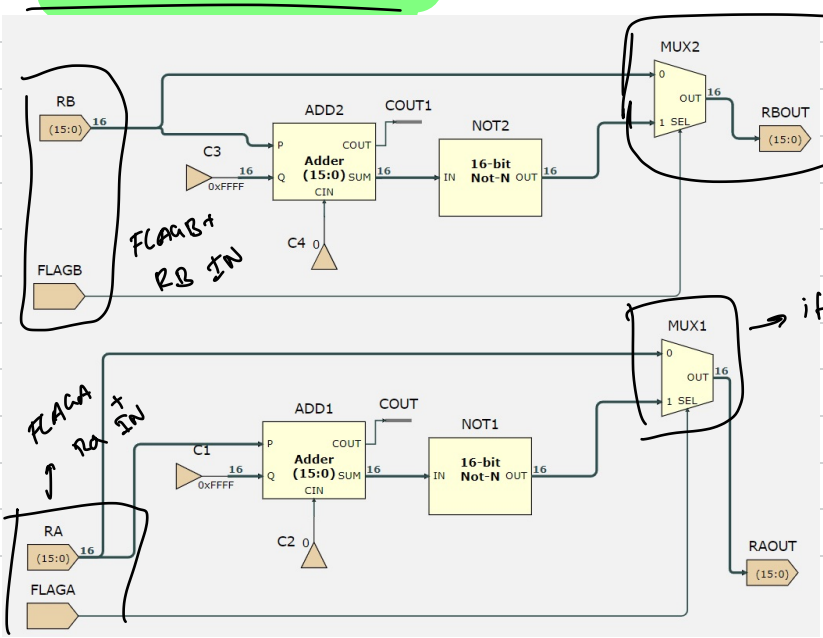
INPUT 0 → normal MOV

Input 1 → Ra := Ra × Rb (8 x 8) bit

Multiplier selects what LSB (7:0) MOVCN does

INPUT 0 OF FLAGA/B

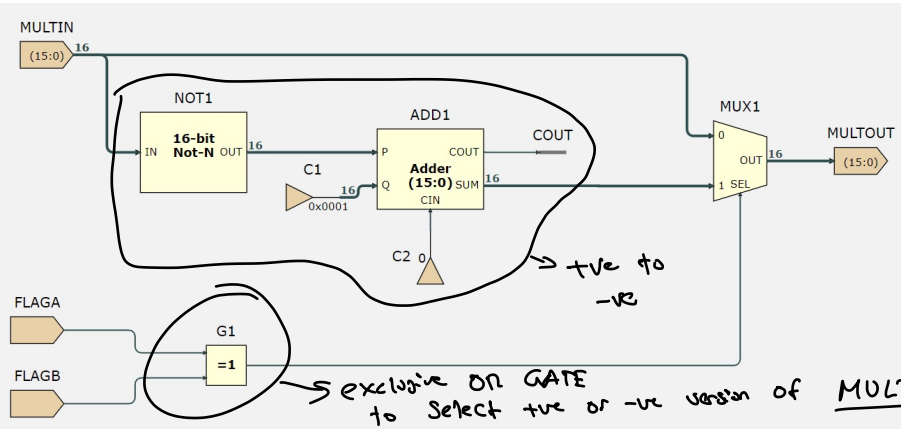
NEGATIVE CALC Block



if FLAGB = 1, SUB 1 from R_b and invert to make +ve

→ if FLAGA = 1, subtract 1 and invert to convert from -ve to +ve

SIGN CORR Block IMPLEMENTATION



Deciding whether (8x8 bit) multiplication is -ve or +ve.

-ve if : ONLY IF ONE OF THE INPUTS ARE -ve

+ve if : None of the inputs are 0, or both are 0.

Exclusive OR :

A (-ve)	B (-ve)	OUT
0	0	0
0	1	1
1	0	1
1	1	0

-ve, +ve

Example

(-ve)(-ve)

(+ve)(+ve)

(-ve)(+ve)

(+ve)(-ve)

Sign extension error/extend problem

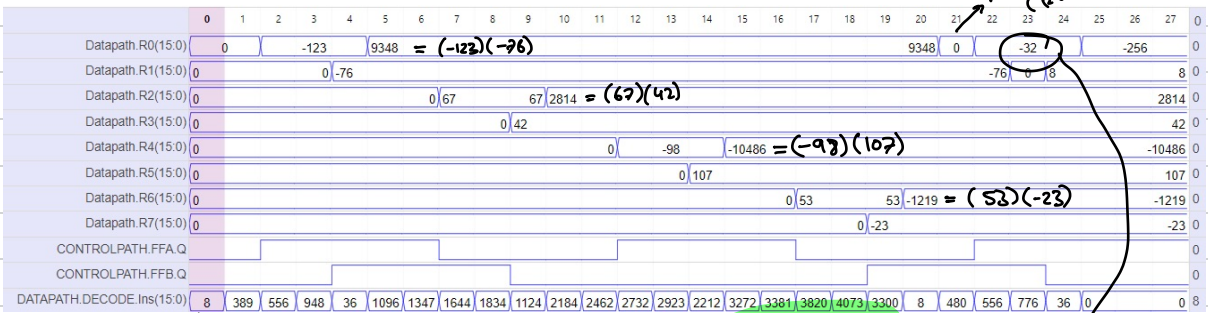
mult8x8.txt

```
1  //MULT1 (-123)x(-76) = 9348
2  MOVC2 R0, R0
3  MOV R0, #-123
4  MOVC3 R1, R1
5  MOV R1, #-76
6  MOVC1 R0, R1
7  //MULT2 (67)x(42) = 2814
8  MOVC2 R2, R2
9  MOV R2, #67
10 MOVC3 R3, R3
11 MOV R3, #42
12 MOVC1 R2, R3
13 //MULT3 (-98)x(107) = -10486
14 MOVC2 R4, R4
15 MOV R4, #-98
16 MOVC3 R5, R5
17 MOV R5, #107
18 MOVC1 R4, R5
19 //MULT4 (53)x(-23) = -1219
20 MOVC2 R6, R6
21 MOV R6, #53
22 MOVC3 R7, R7
23 MOV R7, #-23
24 MOVC1 R6, R7
25 //MULT5 (224)x(8)
26 MOVC2 R0, R0
27 MOV R0, #224
28 MOVC3 R1, R1
29 MOV R1, #8
30 MOVC1 R0, R1
```

mult8x8.ram

```
1  0x00 0x0008
2  0x01 0x0185
3  0x02 0x022c
4  0x03 0x03b4
5  0x04 0x0024
6  0x05 0x0448
7  0x06 0x0543
8  0x07 0x066c
9  0x08 0x072a
10 0x09 0x0464
11 0x0a 0x0888
12 0x0b 0x099e
13 0x0c 0x0aac
14 0x0d 0x0b6b
15 0x0e 0x08a4
16 0x0f 0x0cc8
17 0x10 0x0d35
18 0x11 0x0eec
19 0x12 0x0fe9
20 0x13 0x0ce4
21 0x14 0x0008
22 0x15 0x01e0
23 0x16 0x022c
24 0x17 0x0308
25 0x18 0x0024
26
```

SIMULATING THE CODE IN EEP1



RESOLVING this issue PARTLY by Using MOVC4 ⇒ Makeshift EXT INS

EXT	CN = 06100 (4)	SEL	IMMEXT
0	0	00	IMM (15:0)
0	1	01	0(15:8) IMM(7:0)
1	0	10	IMME(15:8) IMM(7:0)
1	1	11	IMME (15:8) IMM(7:0)

SEL logic says that if MOVC4 is called and NO EXT, then IMM = 0(15:8)IMM(7:0)

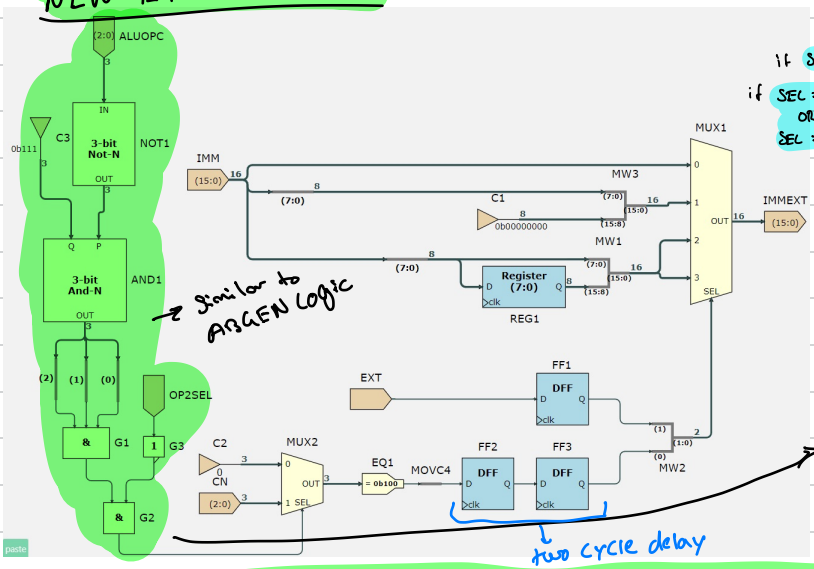
if EXT is called then IMM = IMME(15:8) IMM(7:0)

if none are called then IMM(15:8) = IMM(15:8)

if BOTH are called then IMM = IMME(15:8) IMM(7:0)

NEW EXTEND LOGIC

if EXT = 0 and MOVC4 = 0, IMM does not change



if SEL = 01 (EXT low, MOVC4 High), IMM = 0(15:8) IMM(7:0)

if SEL = 10 OR SEL = 11 (EXT High, MOVC4 High => Normal EXT)

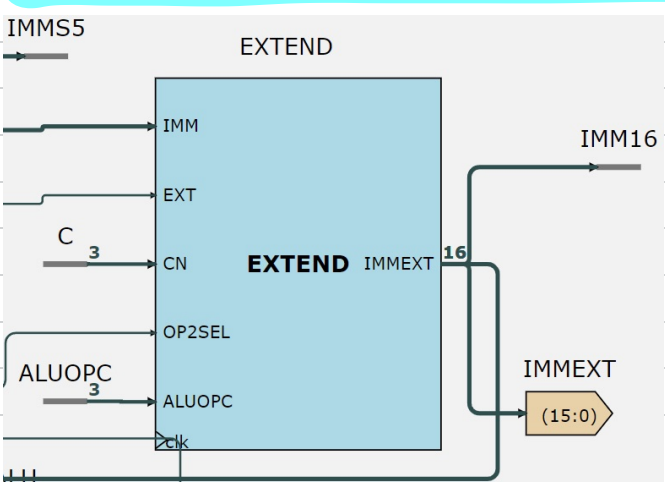
Similar to ARGEN LOGIC

This ensures that this EXT instruction can only be enabled by MOVC4 (needs to be a MOV INSTRUCTION AND OPSEL has to be 0)

two cycle delay

Since MOVCn does not allow IMM values (since this part of the INSTruction word is also Rc, we cannot put an arbitrary number in place of IMM(15:8) like with EXT. INSTEAD if we want to put a larger number than +127, we have to call MOVC4 to prevent sign extension, making the assigned value wrong.

The only solution to this would be to implement an EXT1 instruction that takes an 8 bit - IMM INPUT



New EXTEND block with INS(4:2) input [CN(2:0)], OPSEL AND ALUOPC

REDOING EXAMPLE WITH NEW Logic

mult8x8.txt

```
//MULT1 (-123)x(-76) = 9348
MOVC2 R0, R0
MOV R0, #-123
MOVC3 R1, R1
MOV R1, #-76
MOVC1 R0, R1
//MULT2 (67)x(42) = 2814
MOVC2 R2, R2
MOV R2, #67
MOVC3 R3, R3
MOV R3, #42
MOVC1 R2, R3
//MULT3 (-98)x(107) = -10486
MOVC2 R4, R4
MOV R4, #-98
MOVC3 R5, R5
MOV R5, #107
MOVC1 R4, R5
//MULT4 (53)x(-23) = -1219
MOVC2 R6, R6
MOV R6, #53
MOVC3 R7, R7
MOV R7, #-23
MOVC1 R6, R7
//MULT5 (224)x(8)
MOVC4 R0, R0 // sets IMM(15:8) of MOV R0, #224 INS to 0 -> no negative sign extension
MOVC2 R0, R0 // sets FLAGA on #224
MOV R0, #224 // R0 := 224 (0b 0000 0000 1110 0000) NOT -32 (0b 1111 1111 1110 0000)
MOVC3 R1, R1 // sets FLAGB on #8
MOV R1, #8 // R1 := 8 (0000 0000 0000 1000)
MOVC1 R0, R1 // R0 := R0 x R1 (224 x 8 = 1792)
```

Stay the same as before as no IMM > 127

224 > 127, hence before R0 was assigned to -32

mult8x8.ram

```
0x00 0x0008
0x01 0x0185
0x02 0x022c
0x03 0x03b4
0x04 0x0024
0x05 0x0448
0x06 0x0543
0x07 0x066c
0x08 0x072a
0x09 0x0464
0x0a 0x0888
0x0b 0x099e
0x0c 0x0aac
0x0d 0x0b6b
0x0e 0x08a4
0x0f 0x0cc8
0x10 0x0d35
0x11 0x0eec
0x12 0x0fe9
0x13 0x0ce4
0x14 0x0010
0x15 0x0008
0x16 0x01e0
0x17 0x022c
0x18 0x0308
0x19 0x0024
```