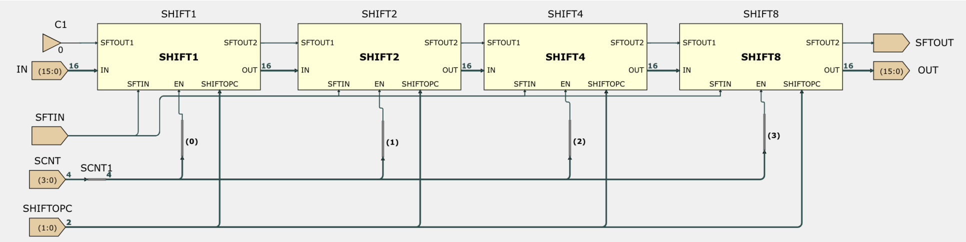
⭐DECA Challenge

Method 1: introducing 16-bit*4-bit multiplier

- *File: “Lab2 Challenge - updated3.14”*
- I got the inspiration from the way the SHIFT block is implemented



// ///
////////////////////////////// //

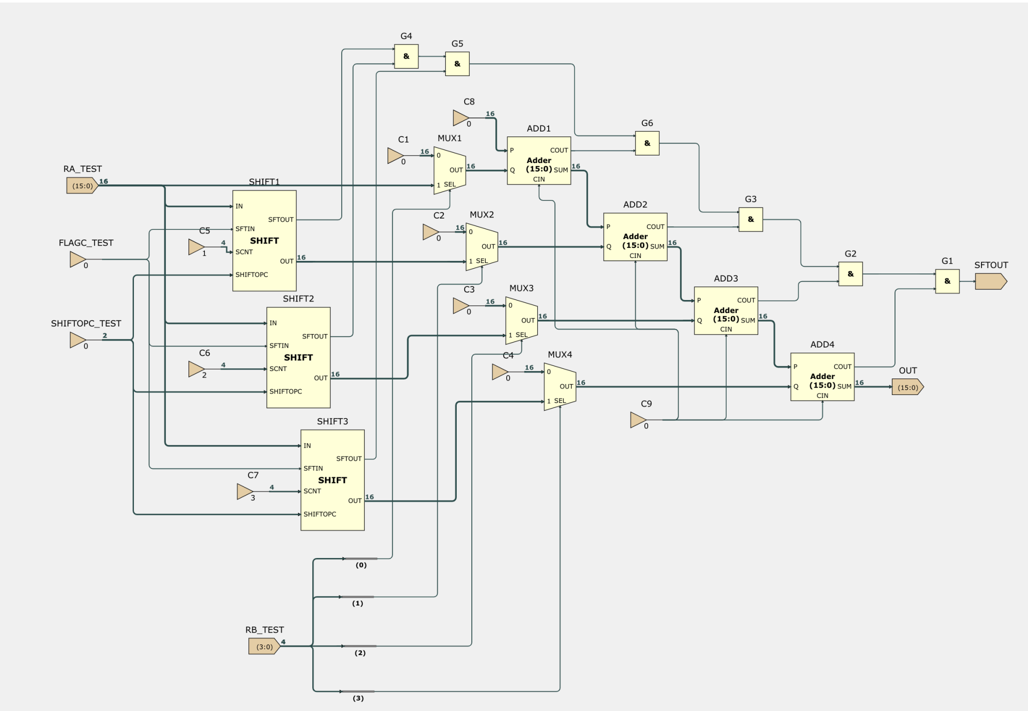
Basic Mechanism:

- When multiplying a 16-bit number to another 16-bit number, we can divide the multiplication to:

1. first do four 16-bit * 4-bit calculation
2. then add them together

New hardware design:

- In order to achieve 16-bit * 4-bit multiplication, I designed a new hardware called “bit4_mult”



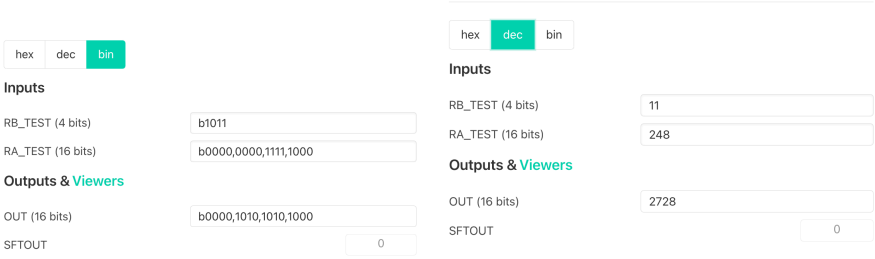
- Requrement: ≤64 full-adders
- This sheet uses four 16-bit adder, which is exactly 16*4=64 full-adders

Input:

- Ra: 16-bit
- Rb: 4-bit
- ShiftOPC: set to 0, indicating a LSL
- FlagC: not used, set to 0 in this case

Output:

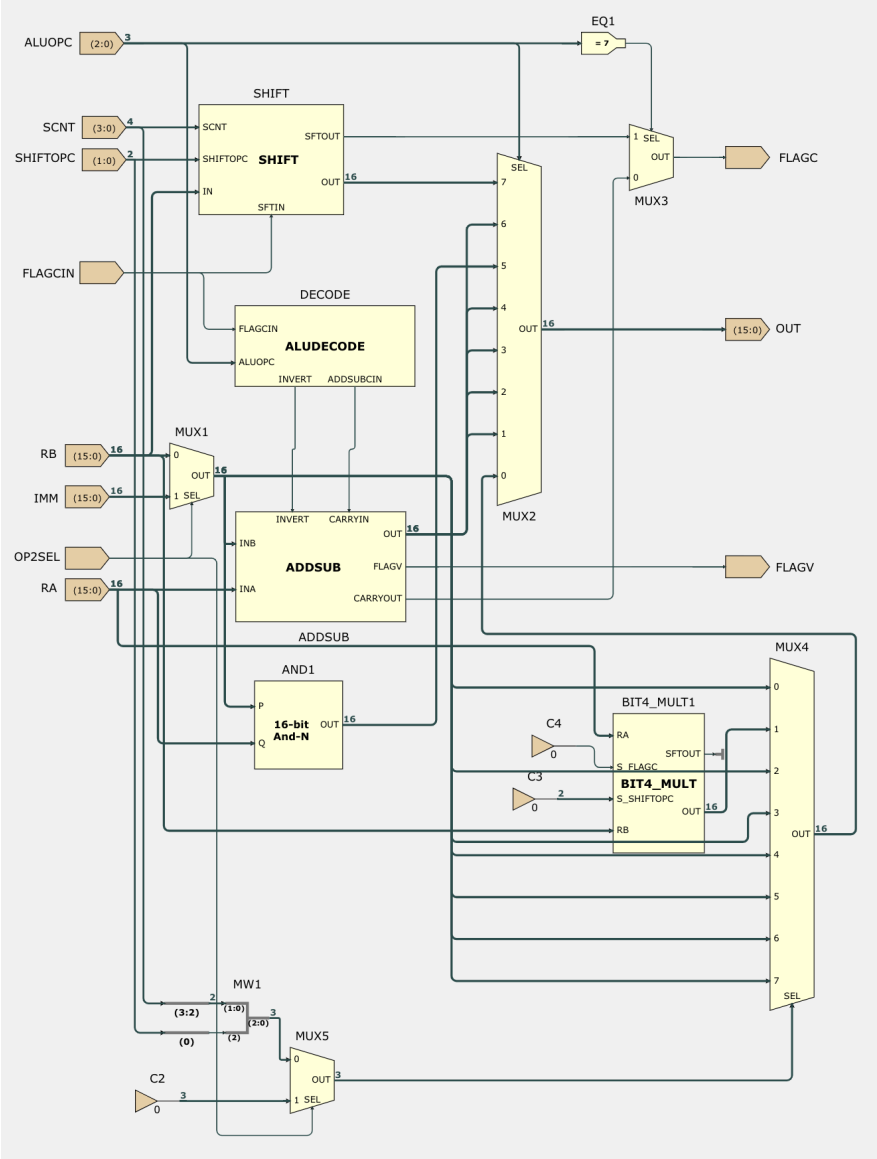
- OUT: the 16-bit result from the multiplication of Ra(15:0) and Rb(3:0)
- FlagC: indicate any overflow in either the SHIFTs and ADDs progress



- simulation shows that this can be used to successfully multiply 16-bit to 4-bit

How is it connected to the whole ALU:

- The new hardware is added as a new branch of the MOV family.
- Therefore, a MUX4 is used to select specific MOV instruction when ALUOPC = 0.
- For INS(4:2) = 0, it is the normal MOV instruction.
- For INS(4:2) = 1, it is the new MOVC1 instruction we added.
- The rest are unused, set to normal MOV for now.



Define a new instruction in ISA:

- MOVCn Ra, Rb;
- Meaning: Ra := Ra*Rb;

0 000 **aaa** 0 **bbb 001** 00

Ra Rb Cn

// ///
/////////////////////////////// //

Benefit:

1. For bits can be process in one loop, fasten speed a lot
2. Can use assembler to assemble, no need to write machine code by hand for new hardware

Drawbacks:

1. Due to the fact that MOVCn have only two operand Ra and Rb, we can't integration multiplication(16*4) and addition together

/// ///
////////////////////////////// //

Software Design(C-like language): (*op1 and op2 are inverted role in this)

```
unsigned int op1, op2, op2_remain, sum
sum = 0;
op2_remain = op2;

while (op2_shifted != 0) {
    pre_partial_result = op1; //Protect op1: Since the result of MOVCn
would be written on op1
                                    //we have to protect op1 from being
covered.

    partial_result = pre_partial_result * op2(3:0); //16-bit*4-bit,
using the new hardware
    sum = sum + partial_result;

    op2_remain = op2_remain >> 4; // Move Op2 to the next group of 4-bit
    op1 = op1 << 4 // Raise Op1 by 4 bit after every loop
}
```

/// ///
////////////////////////////// //

Test-1 (Assembly language translation)["mul-fast-16bit-1.txt"]:

Take 51 times 395 as an example

- Ra: 51 = b0000 0000 0011 0011
- Rb: 77 = b0000 0000 0100 1101
- Expected Result: 3927

```
MOV R0, #51  //op1(limited to 16-bit; for a 16-bit, just add an EXT in
the beginning)
MOV R1, #77  //op2(limited to 16-bit; for a 16-bit, just add an EXT in
the beginning)
MOV R2, R1       //op2_remain
MOV R3, #0       //sum

MOV R2, R2
JEQ 7
    MOV R4, R0

    MOVC1 R4, R2
    ADD R3, R3, R4

    LSR R2, R2, #4
    LSL R0, R0, #4
JMP -7
```

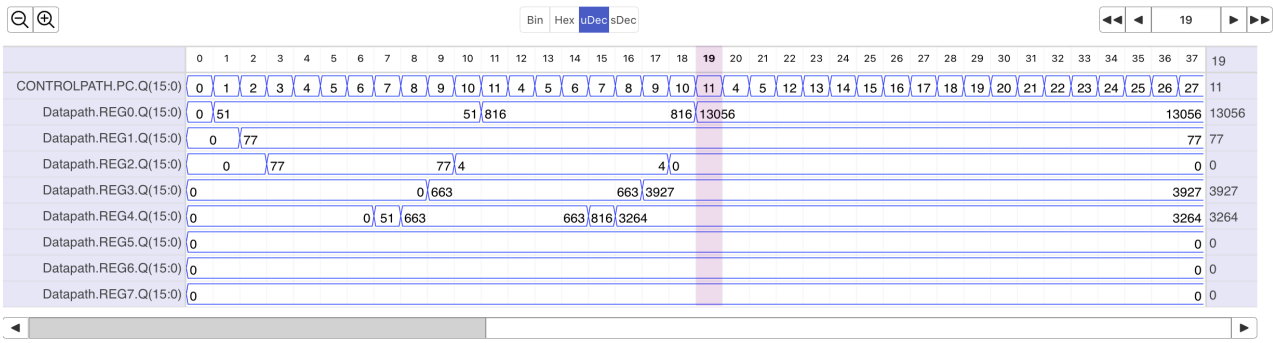
Test-1 (Hex Machine Codes):

```
0x00 0x0133
0x01 0x034d
0x02 0x0420
0x03 0x0700
0x04 0x0440
0x05 0xc207
0x06 0x0800
0x07 0x0844
0x08 0x168c
0x09 0x7454
0x0a 0x7004
0x0b 0xc0f9
```

Test-1 (Binary Machine Codes):

```
0x00 -> 0b0000000100110011
0x01 -> 0b0000001101001101
0x02 -> 0b0000010000100000
0x03 -> 0b0000011100000000

0x04 -> 0b0000010001000000
0x05 -> 0b1100001000000111

0x06 -> 0b0000100000000000

0x07 -> 0b 0000 1000 0100 0100
           aaa  bbbc cc00

0x08 -> 0b0001011010001100
0x09 -> 0b0111010001010100
0x0A -> 0b0111000000000100

0x0B -> 0b1100000011111001
```

Test-1 (Result):



- 3927, matched👍
- Within 19 clock cycle.

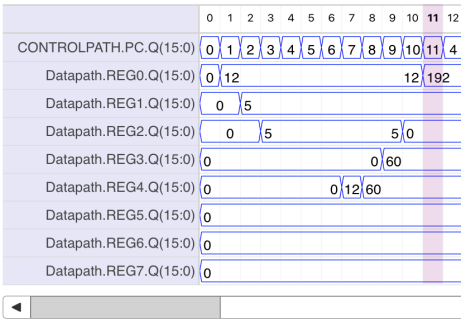
//

Test-2 (Assembly language translation)[“mul-fast-16bit-2.txt”]:

Take 12 times 5 as an example
- Ra: 12 = b0000 0000 0000 1100
- Rb: 5 = b0000 0000 0000 0101
- Expected Result: 60

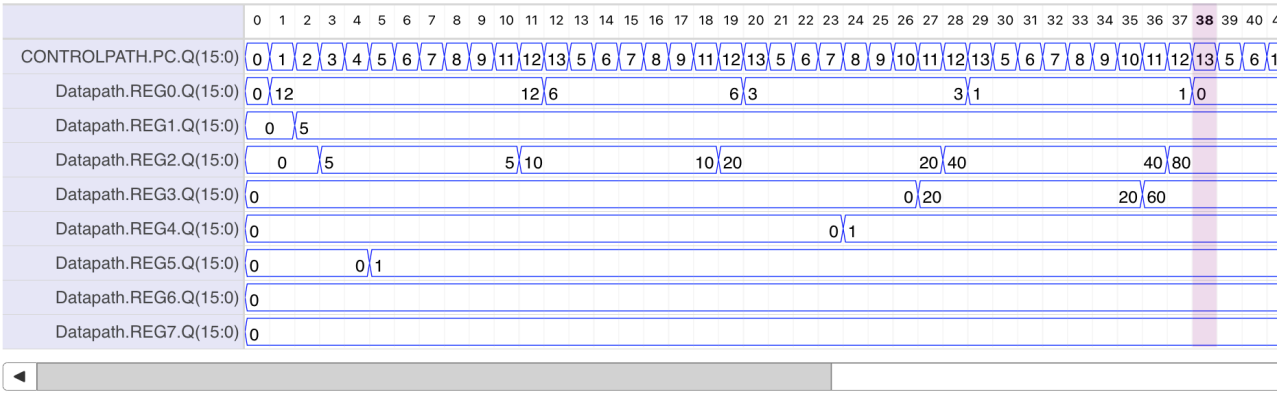
```
MOV R0, #12  //op1
MOV R1, #5        //op2
```

Test-2 (Result):



- 60, matched👍
- Within 11 clock cycle.

Compare to previous algorithm[“mul-normal-16bit.txt”]:



- It used to take 38 clock-cycle!

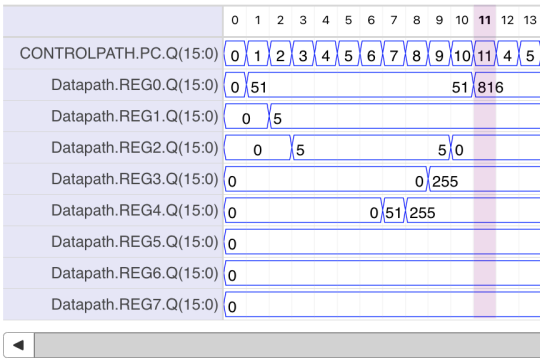
//

Test-3 (Is 51*5 faster than 5*51?):

[“mul-fast-16bit-3.txt”]

Using 51*5: (expected 1 loop)
- Ra: 51 = b0000 0000 0011 0011
- Rb: 5 = b0000 0000 0000 0101
- Expected Result: 255

```
MOV R0, #51  //op1
MOV R1, #5        //op2
```

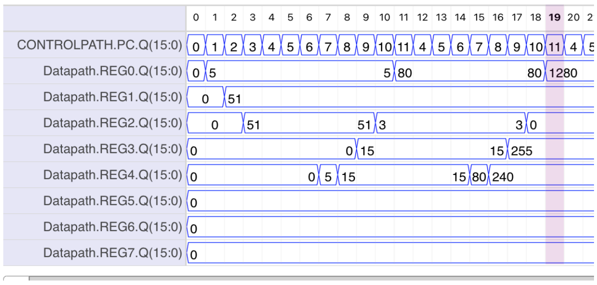


- Done in 11 loops, faster

[“mul-fast-16bit-4.txt”]

Using 51*5: (expected 2 loops)
- Ra: 5 = b0000 0000 0000 0101
- Rb: 51 = b0000 0000 0011 0011
- Expected Result: 255

```
MOV R0, #5        //op1
MOV R1, #51   //op2
```



- Done in 19 loops, slower.

This leads us to an approach to manually improve calculation speed:
- When we are multiplying two numbers with different length, placing the shorter one as Rb might fastern the speed (but not always)
- This is because, the loop number is determined by how many 4-bits Rb occupies.

// ///
/////////////////////////////// ///

Further explorations:

1. **Take signed & unsighed.**
- Just add MOVC2
- and change SHIFTOPC to ASR?
2. **Take multiply and diviion.**
- Use logical shift right
- for signed, try the other two right-shift type
3. **Extend from 16-bit to 32-bit multiplication.**
4. **Take immediate value.**

// ///
/////////////////////////////// ///

Further improvement ideas:

Using substraction to optimize (using 4-bits as a group):

Identify the Closest Power of Two
- *Find the largest power of two that is greater than or equal to the multiplier.*

N ≈ 2^m

1. *If the multiplier is just below a power of two (e.g., 15 = 16 - 1), use subtraction.*
2. *If the multiplier is just above a power of two (e.g., 9 = 8 + 1), use addition.*

- We can design another hardware that uses substraction rather than addition to implement 16*4 bit multiplication.
- We still use the idea of grouping to set 4 bits as a group.
- The problem is: how do we quickly determine weather to use addition-multiply or substraction-multiply in each 4-bit group?
- For example:
- Rb: **** / **** / **** / ****
- sub add add sub
- It should look similar to the one we have.

// // /////////////////////////////// //

Method 2: integrating MOVCn and ADD

- One big problem we have on the previous method is that MOVCn only have two operand: Ra, Rb
- this means that we can't do the `MOVC1 R4, R2` and `ADD R3, R3, R4` at the same time.
- If we are able to change the assembler to design some new instructions like "MOVCn Ra, Rb, Rc" in the ISA (or…write the machine code by ourselves)
- Then we can have three operand. Integrating MOVC1 and ADD.
- (But this is hard to achieve since we are on a 16-bit CPU😭)
- If this is achieved, we might only need 1 loop to finish everything!