

# Lab 2 - Challenge

Sunday, 16 March, 2025 11:53 AM

## 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 MOV<sub>Cn</sub> 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 MOV<sub>C1</sub> Ra, Rb could be used to implement  $Ra := Ra \text{ op } Rb$ , 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 MOV<sub>Cn</sub> 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.

1. Goal:
  - To implement additional software/ instructions to speed up the EEP1 multiplication
2. Defining the problem of the current multiplication software:
  - Slow due to the repeated shifting and adding
3. Factors to evaluate:
  - Number of clock cycles needed
  - Number of adders (out of 64) needed
4. Planned steps to execute:
  - i. Design Description
    - My design is based on the Karatsuba multiplication algorithm. whereby:

<p>In base ten:</p> $1234 \times 5678 = 7006652$ <p>1) I split the two 4 digit numbers to each 2 halves</p> $\begin{array}{cc} 12 & 34 \\ a & b \\ (2^0)(15) & (2^0)(15) \end{array}$ $\begin{array}{cc} 56 & 78 \\ c & d \\ (2^0)(15) & (2^0)(15) \end{array}$ <p><math>\therefore</math> It becomes <math>(a \cdot 10^3 + b)(c \cdot 10^3 + d)</math></p> $= ac \cdot 10^6 + (ad + bc) \cdot 10^3 + bd$ <p>2) I compute <math>a \cdot c</math></p> $(12)(56) = 672$ <p>3) I compute <math>b \cdot d</math></p> $(34)(78) = 2652$ <p>4) I compute <math>a \cdot d</math></p> $(12)(78) = 936$ <p>5) I compute <math>b \cdot c</math></p> $(34)(56) = 1904$	<p>Implementing in EEP1 (16 bit registers)</p> <p><math>Ra \times Rb = OUT</math> (15:0) (15:0) (31:0)</p> <p>Since I can only MOV 8 bit <math>\rightarrow</math> MM into the registers due to restriction of the machine code, and <math>8 = \frac{1}{2}(16)</math>, I'll let:</p> <p><math>R1 = a</math> <math>R2 = b</math> <math>R3 = c</math> <math>R4 = d</math></p> <p>8x8 bit = 16 bit multiplication by new block (MUL 8x8) (New 32b MOVCL needed) we don't use the existing shift block as it contains other hardware (MUX) to carry out ASR, XSR, LSR function we only need LSL to carry out multiplication, so making a block w/o the unused hardware will make the execution faster addition is also direct done in the MUL 8x8 block</p> <p>From section 4, we know that <math>Ra \times Rb</math> is  <math>OP1 \times OP2 = 2^{n-1} b_{n-1} \times OP2 + 2^{n-2} b_{n-2} \times OP2 + \dots + 2^0 b_0 \times OP2</math>  <math>\hookrightarrow OP2 \text{ LSL } \#1 \text{ for every increase in bit of } OP1 \text{ and add up}</math> <div style="text-align: right; font-size: small;"> <math>\begin{matrix} 0 \text{ - carries} \\ 1 \text{ - adds} \end{matrix}</math> </div> <p>Addition is done by modifying the existing ADD1 block to perform 8 bit + 8 bit <math>\rightarrow</math> 9 bit outcome w/o the carry output, as we just need the combination (9 bit) for next step. Separating the carry output has no use</p> </p>
---	--

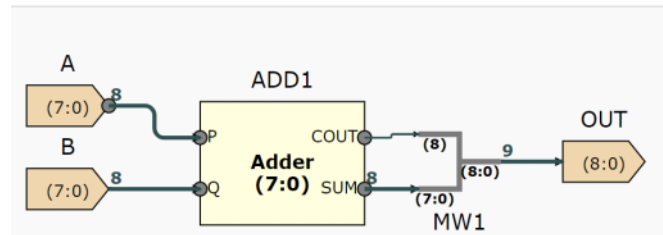
	<p>Below illustrates my implementation concept:</p> <p>eg: <math>11001010 \times 111</math></p> <p>extra 0 to be merged <math>\rightarrow</math> 8 bit addition</p> <p>8 bit multiplication = max 7 addition done = 7 ADD8to9 needed</p>
<p>4) I compute <math>(ad + bc) \cdot 10^3</math></p> $(936 + 1904) \cdot 10^3 = 284000$	<p><math>\rightarrow</math> 6 ADD @ 8b LSL, #8</p>
<p>5) I compute <math>ac \cdot 10^6</math></p> $672 \cdot 10^6 = 6720000$	<p><math>\rightarrow</math> 8b LSL, #16 (full register bit) <math>\therefore</math> same as no shifting <math>\therefore</math> no need shift</p>
<p>6) I compute <math>ac \cdot 10^6 + (ad + bc) \cdot 10^3 + bd</math></p> $6720000 + 2840000 + 2652 = 9006652$ <p><math>\therefore</math> (end)</p>	<p><math>\rightarrow</math> We can't carry out 32 bit addition directly in 16 bit registers <math>\therefore</math> split</p> <p>① ADD LS 16 bits of final outcome, made up of <math>bd \rightarrow</math> LS 16 + Flag C</p> <p>② ADD MS 16 bits of final outcome, made up of <math>ac \cdot 10^6</math> <span style="border: 1px solid black; padding: 2px;">overflow bits of <math>\star</math></span> <math>\rightarrow</math> MS 16 bit <math>\hookrightarrow</math> get by LSR (addtc) #8</p>

## ii. Implementation

### 1. First for statement:

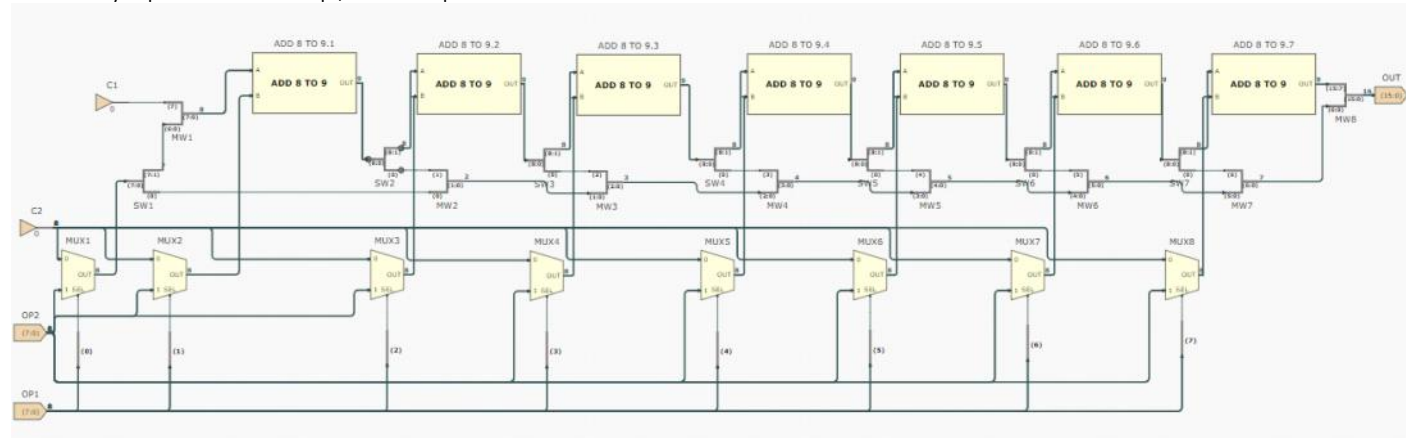
Addition is done by modifying the existing ADD1 block to perform 8 bit + 8 bit  $\rightarrow$  9 bit outcome w/o the carry output, as we just need the combination (9 bit) for next step. Separating the carry output has no use

- I've created the modified adder (ADD8to9):

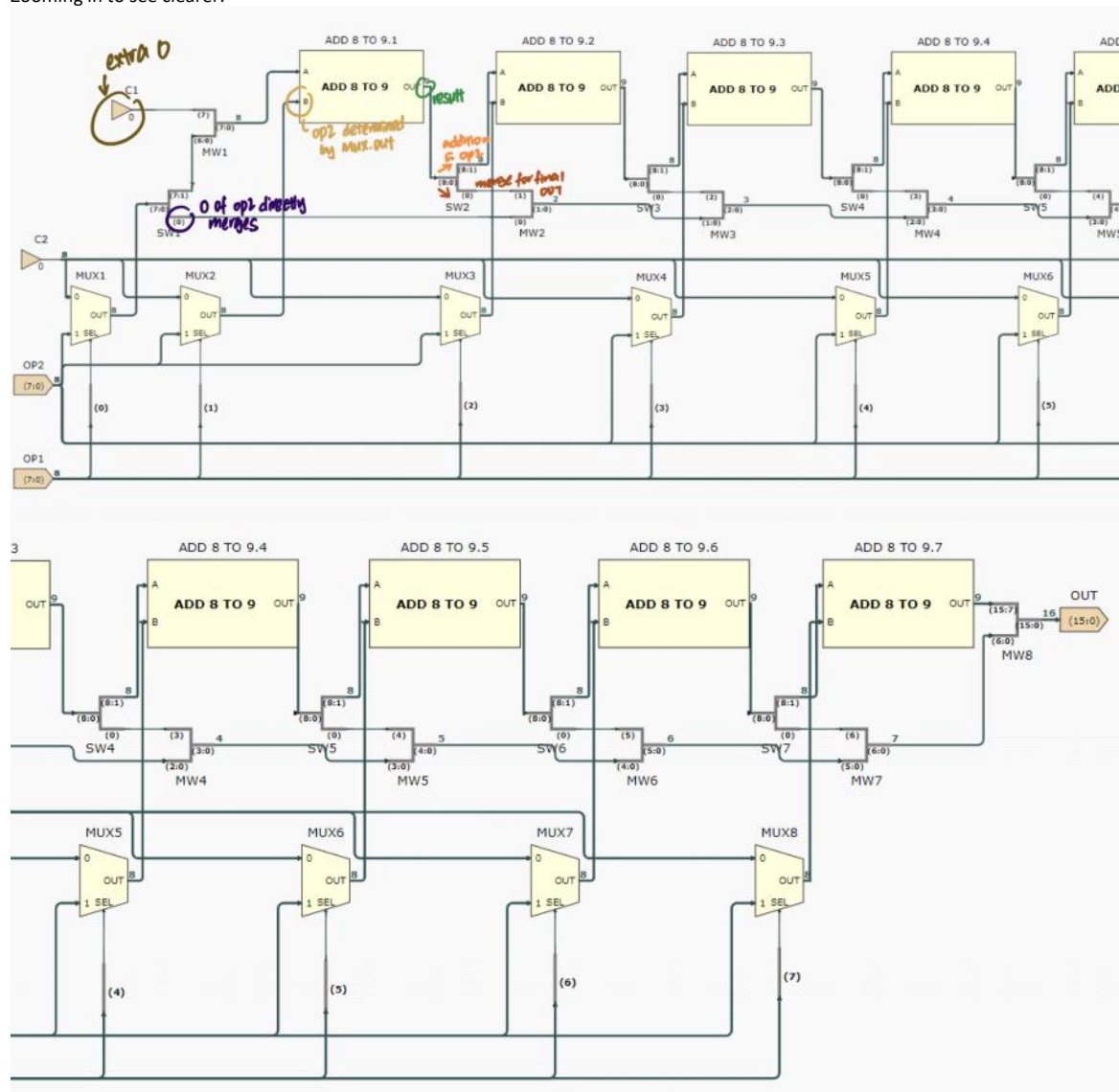


- Note that CIN was removed as it is not used

2. Based on my implementation concept, I've build up MUL 8x8 as below:



- Zooming in to see clearer:



3. I need a new instruction to tell the system that I want to use the results from my MUL 8x8

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

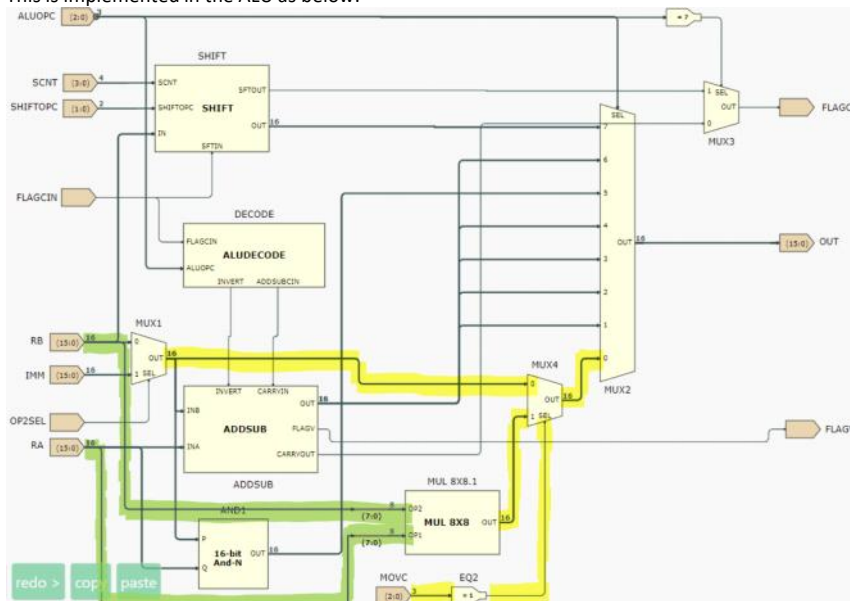
ALUOPC	INS(8)=0	INS(8)=1
0	MOV <sup>1</sup> Ra, Rb	MOV Ra, #IMM
1	ADD Rc, Ra, Rb	ADD Ra, #IMM
2	SUB Rc, Ra, Rb	SUB Ra, #IMM
3	ADC Rc, Ra, Rb	ADC Ra, #IMM
4	SBC Rc, Ra, Rb	SBC Ra, #IMM
5	AND Rc, Ra, Rb	AND Ra, #IMM
6	CMP <sup>1</sup> Ra, Rb	CMP Ra, #IMM
7	Shift Ra, Rb, #SCNT (see SHIFTOPC for Shift)	

<sup>1</sup>Instruction does not use Rb, this field is 0)

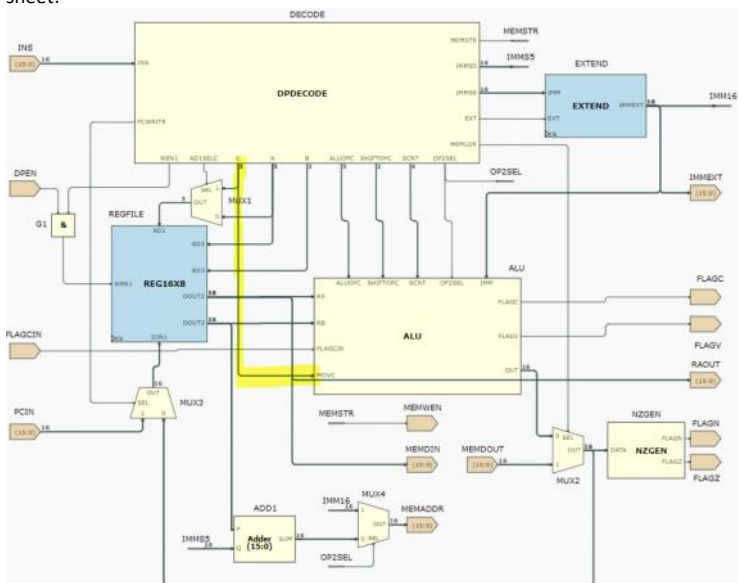
Legend	Meaning
Imm4	4 bit unsigned immediate
Imms8	8 bit signed immediate
(0)	Must be 0 for current instructions, non-zero values are reserved for expansion

- We can see that the c field for MOV instruction is not used for both  $INS(8)=0/1$ , hence, by making the c field of an MOV instruction,  $INS(4:2) \neq 0$  (I'll set it as 001), I can tell the system to use the MUL 8x8
- To use the c field from the existing hardware of MOV for MOVC, a and b field is already present, so I can't get rid of them, a and b are both 3 bits only, meaning that I can't directly fit 8 bit IMM in the instructions to implement IMM\*IMM
- Hence, I can only MOV the 8 bit IMM into registers first and call these registers out by their address (3 bit) of a and b and perform multiplication on their contents (that are passed to inputs of MUL 8x8)
- Thus I define when  $MOVC1 = 001 = MOVC\ Ra, Rb = Ra := Ra * Rb$
- This is implemented in the ALU as below:



- When ALUOPC=0, MUX4 will pass on the results of the normal MOV instruction if field c (MOVc) = 0, and MULx8.OUT if MOVc=1
- I need bus selects to select the LS 8 bits from both Ra and Rb that are needed for the multiplication, the remaining MS are just due to sign extension

4. As the c field forms MOV<sub>C</sub>, I need to connect DP<sub>CODE</sub>.C to MOV<sub>C</sub> in the datapath sheet:



5. Final design hierarchy:



6. From the above analysis, to implement eg 101x111, I should make a code so that 101 is MOV into Ra first, and 2nd code MOV Rb, #111, then third is MOVC1 Ra, Rb  
 - To implement 0x5A4B x 0xF51F, my code is as below

```

lab2challenge.txt
File Edit View

// implementing 0x5A4B x 0xF51F = 5674B615
// 0x5A4B=ab=R1::R2
// 0xF51F=cd=R3::R4

MOV R1, #0x5A
MOV R2, #0x4B
MOV R3, #0xF5
MOV R4, #0x1F

MOV R5, R1 //MOV will change Ra's value and R1 is still needed for further steps, so must move to another register first
MOVC1 R5, R3 //R5=ac
MOV R6, R2
MOVC1 R6, R4 //R6=bd
MOVC1 R1, R4 //R1 not needed for further steps anymore, R1=ad
MOVC1 R2, R3 //R2=bc

ADD R1, R2 //R1=ad+bc
LSL R7, R1, #8
LSR R1, R1, #8 //done before ADD R6, R7 so that the FlagC for ADC will be correct

ADD R6, R7
ADC R5, R1

// The product of multiplication will be shown as R5::R6
  
```

7. I ran it with the eepAssembler to get the .ram file

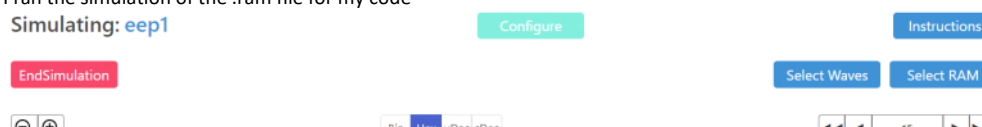
```

lab2challenge.ram
File Edit View

0x00 0x035a
0x01 0x054b
0x02 0x07f5
0x03 0x091f
0x04 0x0a20
0x05 0x0a64
0x06 0x0c40
0x07 0x0c84
0x08 0x0284
0x09 0x0464
0x0a 0x1244
0x0b 0x7e28
0x0c 0x7238
0x0d 0x1cf8
0x0e 0x3a34
  
```

iii. Simulation

- I ran the simulation of the .ram file for my code



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
CONTROLPATH.PC.Q(15:0)																			x000F
Datapath.REG0.Q(15:0)	x0000																x0000		x0000
Datapath.REG1.Q(15:0)	x005A								x005A	x0AE6	x52AD	x0052					x0052		x0052
Datapath.REG2.Q(15:0)	x0000	x004B							x004B	x47C7							x47C7		x47C7
Datapath.REG3.Q(15:0)	x0000	xFFF5															xFFF5		xFFF5
Datapath.REG4.Q(15:0)	x0000	x0000	x001F														x001F		x001F
Datapath.REG5.Q(15:0)	x0000		x0000				x5622						x5622				x5674		x5674
Datapath.REG6.Q(15:0)	x0000				x0000			x0915				x0915	xB615				xB615		xB615
Datapath.REG7.Q(15:0)	x0000									x0000	xAD00						xAD00		xAD00

- I concluded that my 16x16 -> 32 multiplier works correctly (0x5A4B x 0xF51F = 0x5674B615)
- It needs only 15 clock cycles to complete the multiplication, a reduction from the 46 cycles needed by the original eep1 multiplier

#### iv. Comparison with existing multiplier

- My multiplier runs faster, less clock cycles are needed for the multiplication
- But more hardware would be needed, thus increasing the cost to build up the system
  - o 56 Full Adders were used in MUL 8x8 (7x8 Full Adders in ADD1) - addition on old eep1 hardware
  - o 8 Full Adder were used in the ADDSUB block for ADD, ADC instructions
  - o Total of 64 Full Adders used
  - o MUL 8x8 used 8 2-MUX - addition on old eep1 hardware
  - o The ALU sheet has a new MUX4 2-MUX - addition on old eep1 hardware
  - o The new design has addition of wires, bus selects, bus compare, merge wires, split wires and constant inputs