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 `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

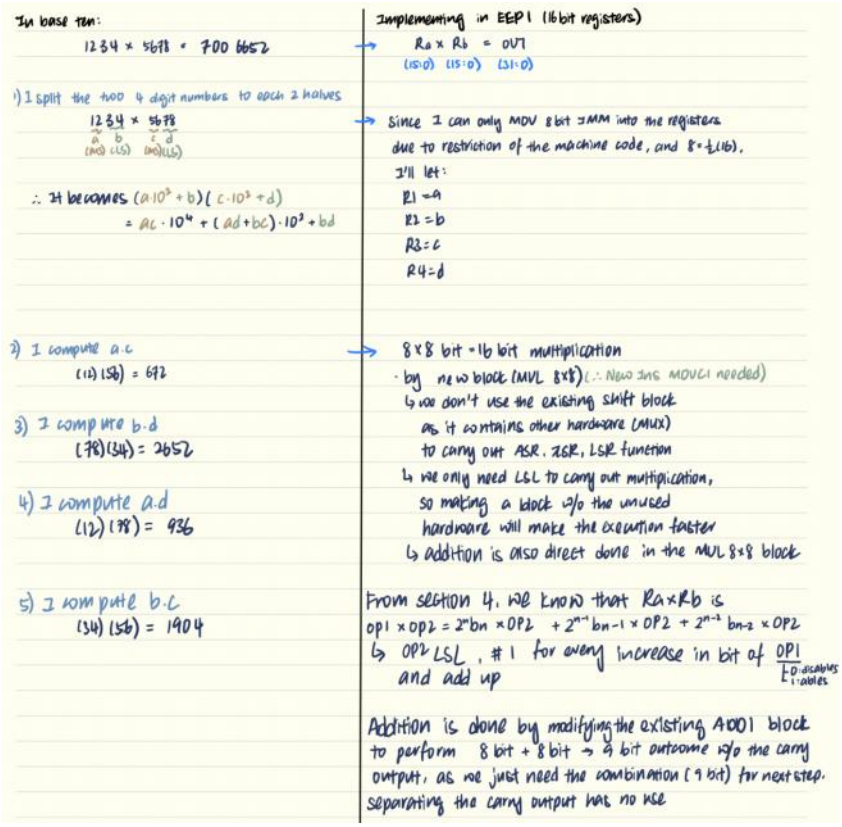
Try to work out optimal sequences of instructions. Consider whether slightly different hardware could speed up the software.

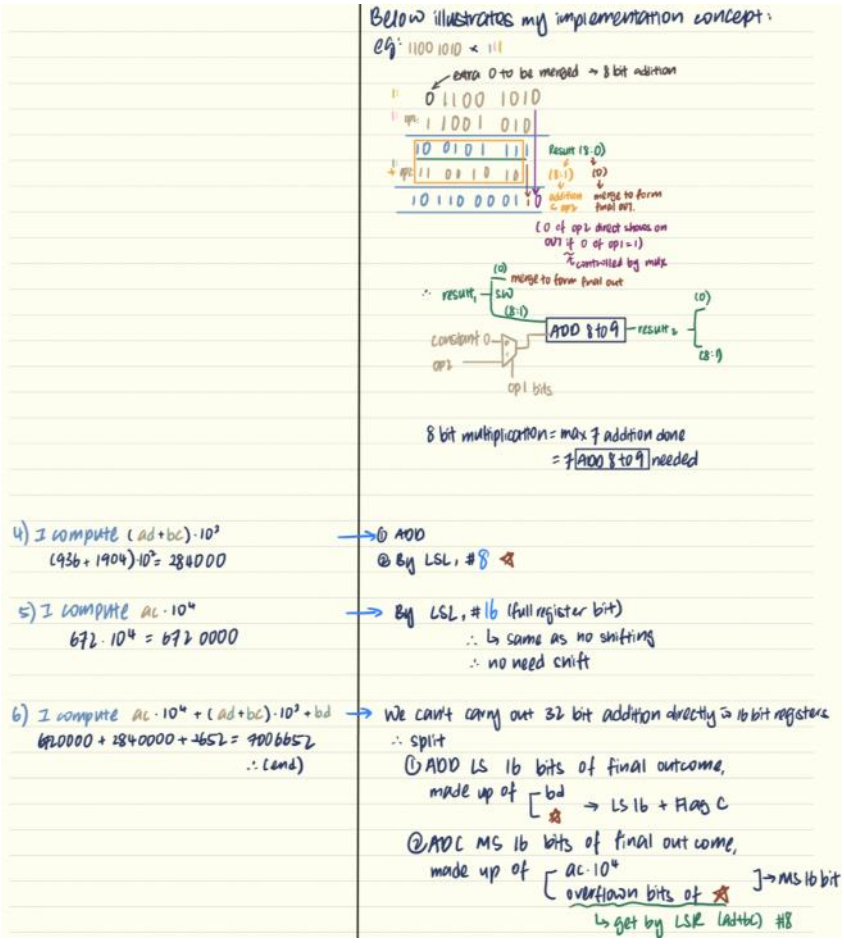
1. Goal:
- To implement additional software/ instructions to speed up the EEP1 multiplication

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:



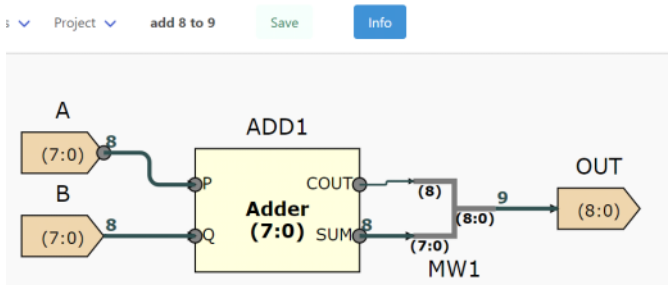


ii. Implementation

1. First for statement:

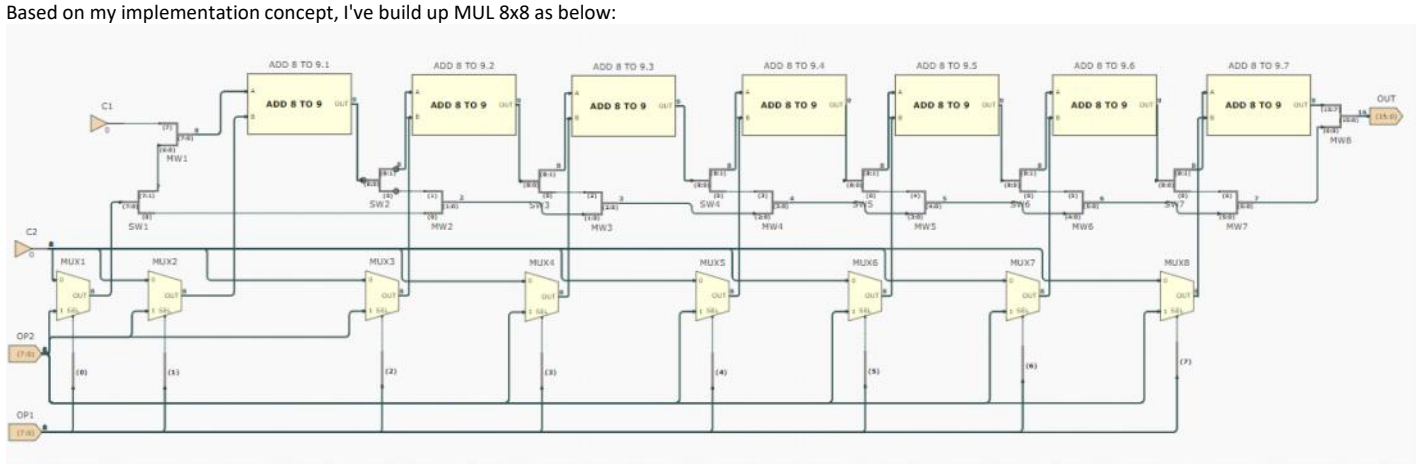
Addition is done by modifying the existing ADD1 block to perform 8 bit + 8 bit → 9 bit outcome w/o the carry output, as we just need the combination (9 bit) for next step. Separating the carry output has no use

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

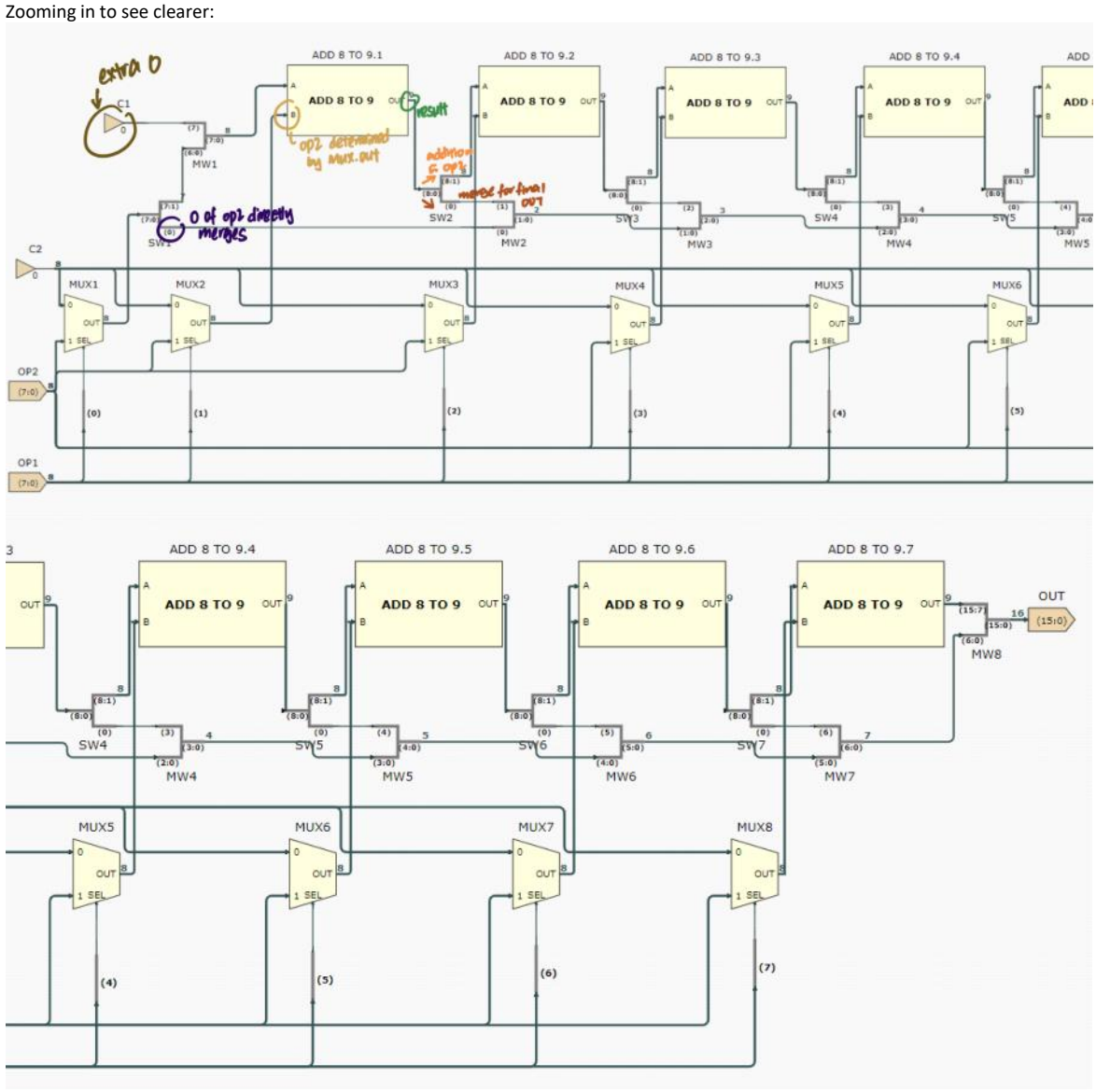
- 

- 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

- By looking at the table below:

ALUOPC	INS(8)=0	INS(8)=1
0	MOV[1] 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[1] Ra, Rb	CMP Ra, #IMM
7	Shift Ra, Rb, #SCNT (see SHIFTOPC for Shift)	

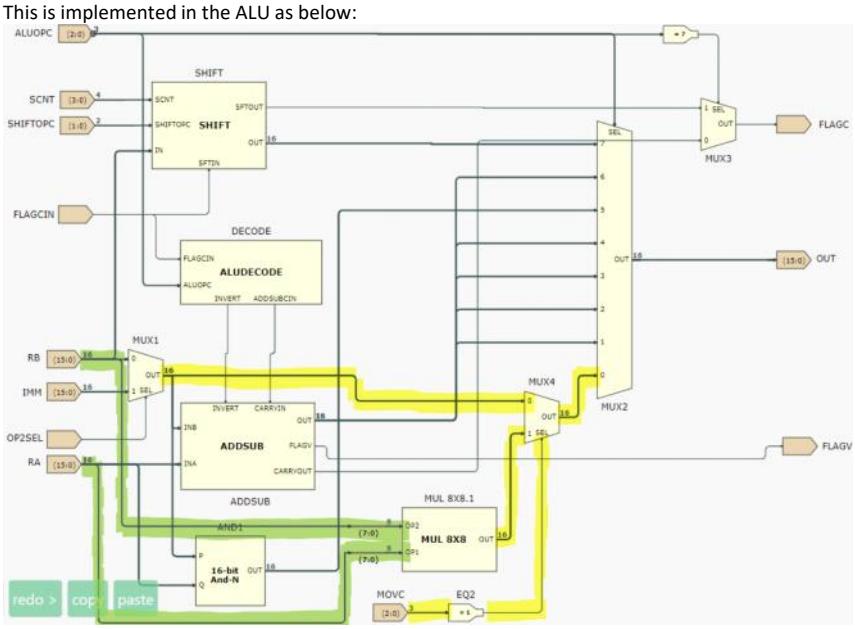
[1]Instruction does not use Rc, this field is (0)

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

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

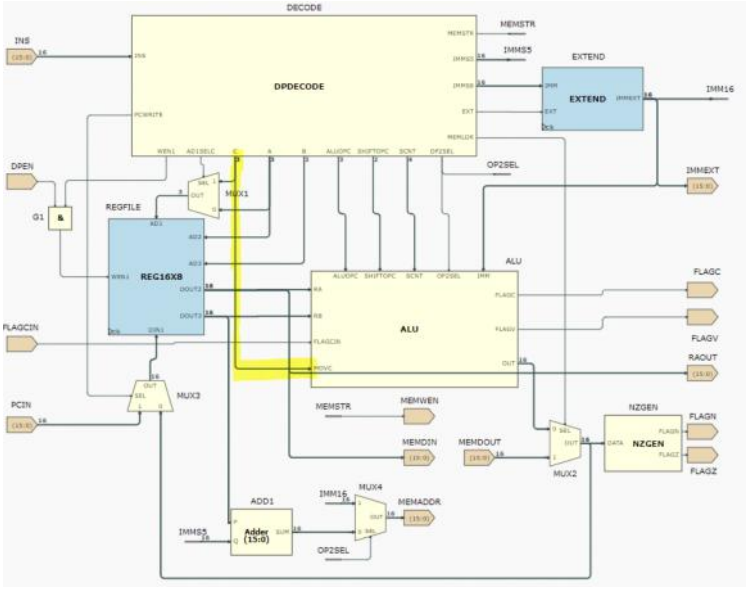
EEP1 machine code	INS bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ALU	0	ALUOPC=7				a			Shift-opc(1)	b			Shift-opc(0)	Imm4 (SCNT)			
		ALUOPC=0..6				a			0	b			c			(0)	
									1	Imms8 (IMM)							

- 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) !=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:
-

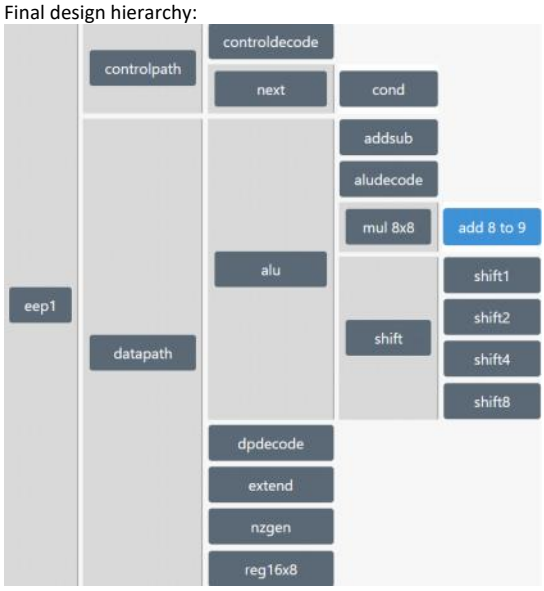


- When ALUOPC=0, MUX4 will pass on the results of the normal MOV instruction if field c (MOVC) = 0, and MUL8x8.OUT if MOVC=1
- I need bus selects to select the LS 8 bits from both Ra and Rb that are needed for the multiplication, the remaining MS are just due to sign extension

4. As the c field forms MOVC, I need to connect DPDECODE.C to MOVC 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

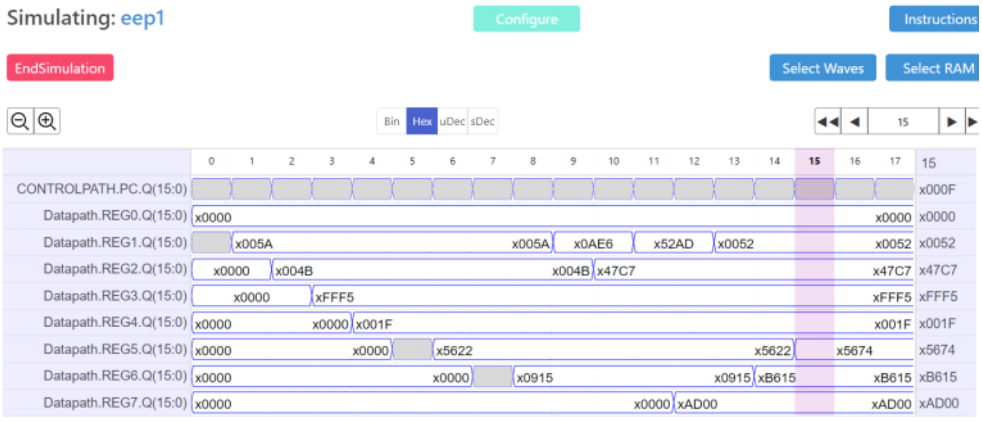
- ```
  // implementing 0x5A4B x 0xF51F = 5674B615
  // 0x5A4B=ab=R1::R2
  // 0xF51F=cd=R3::R4

  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

- ```
  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



- 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
  - 56 Full Adders were used in MUL 8x8 (7x8 Full Adders in ADD1) - addition on old eep1 hardware
  - 8 Full Adder were used in the ADDSUB block for ADD, ADC instructions
  - Total of 64 Full Adders used
  - MUL 8x8 used 8 2-MUX - addition on old eep1 hardware
  - The ALU sheet has a new MUX4 2-MUX - addition on old eep1 hardware
  - The new design has addition of wires, bus selects, bus compare, merge wires, split wires and constant inputs