5 Register multiplication

Saturday, 15 February 2025 22:27

Implement additional instructions and/or hardware that can speed up EEP1 register multiplication.
Limited to using no more than 64 full adders. (8 Issie adder blocks of 8 bits each)

Credit gained from:

- Solution
- Speed of solution in clock cycles, and its cost in hardware adders, with the "pure software" implementation in Lab 2
- Knowing how your solution can be used in different contexts: for example, to implement signed and unsigned multiplication

To speed up EEP1 multiplication, this requires the definition of **additional ALU instruction in the ISA.** This is possible because some combination of bits in the machine work are not currently used. Specifically the MOV instruction does not use register C, the 3 bits specifying this are set to 0 for a normal MOV.

There are 7 **additional** MOV instructions, using machine code as for MOV 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 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.

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

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)

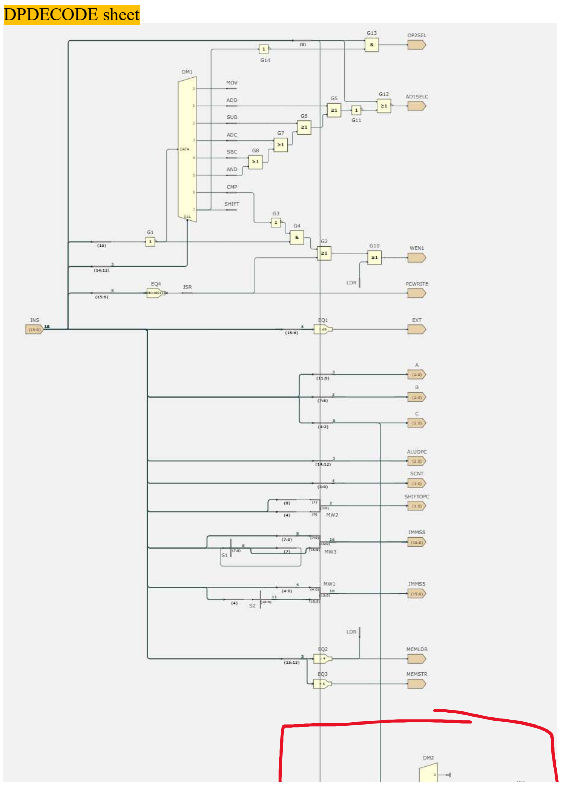
Legend	Meaning
Imm4	4 bit unsigned immediate
Imms8	8 bit signed immediate
(0)	Must be 0 for current instructions, non-zero values are reserved for expansion

Conditions to implement the extra instruction:
8th bit of the machine code must be equal to 0 to be able to use register C field or else we would overlap with immediate operand

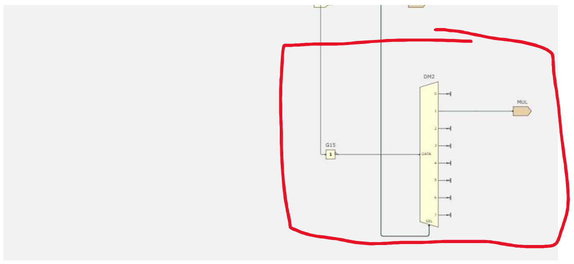
EEP1 machine code	INS bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	ALU	0	ALUOPC=7			a			Shift-opc(1)	b			Shift-opc(0)	Imm4 (SCNT)			
			ALUOPC=0..6			a			0	b			c			(0)	
									1	Imms8 (IMM)							

★

DPDECODE sheet



New logic:
The NOT gate is connected to the 8th bit bus selector at the top



New logic:
The NOT gate is connected to the 8th bit bus selector at the top
Therefore when bit 8 = 0, we assert 0 into the DEMUX and use INS(4:2) as our field C to select the MOVCn operation

In this case I have selected MOVC1 to be the operation to perform our multiplication

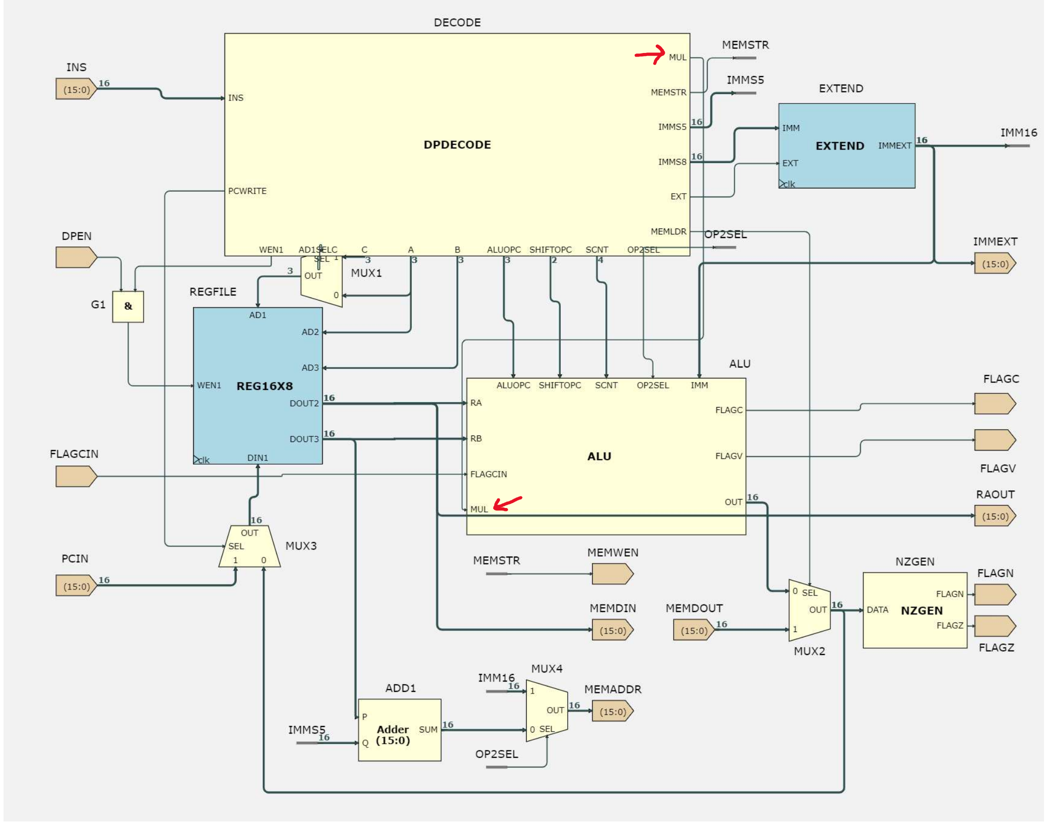
5.2.2 Adding to the datapath hardware

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.

DATAPATH sheet

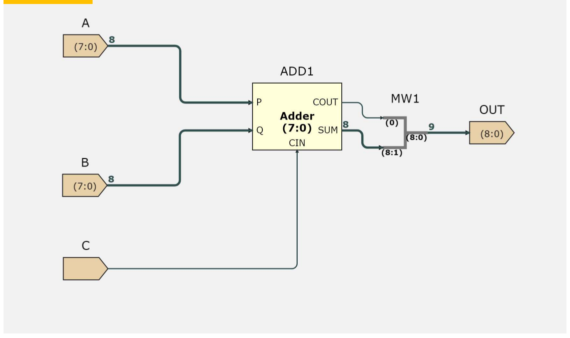


New:
Connection between MUL output and input of DPDECODE block and ALU block

5.2.3 Using your new hardware

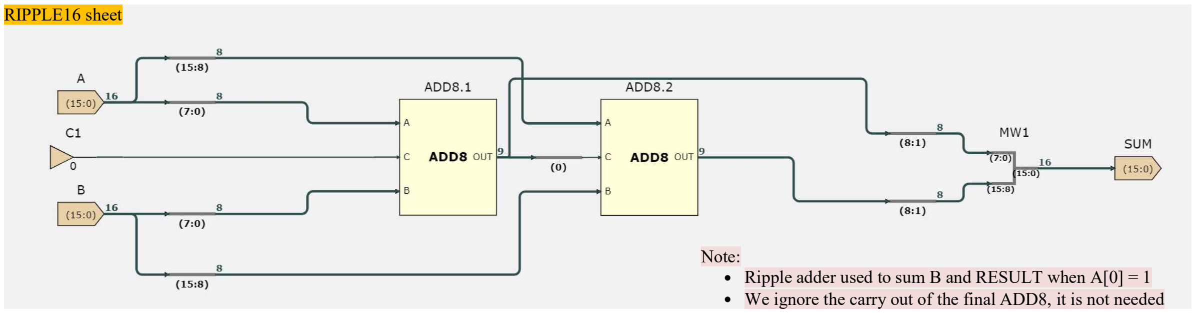
Try to work out optimal sequences of instructions. Consider whether slightly different hardware could speed up the software.

ADD8 sheet



Note:
LSB of the 9 bit output is the carry out bit from the 8 bit adder

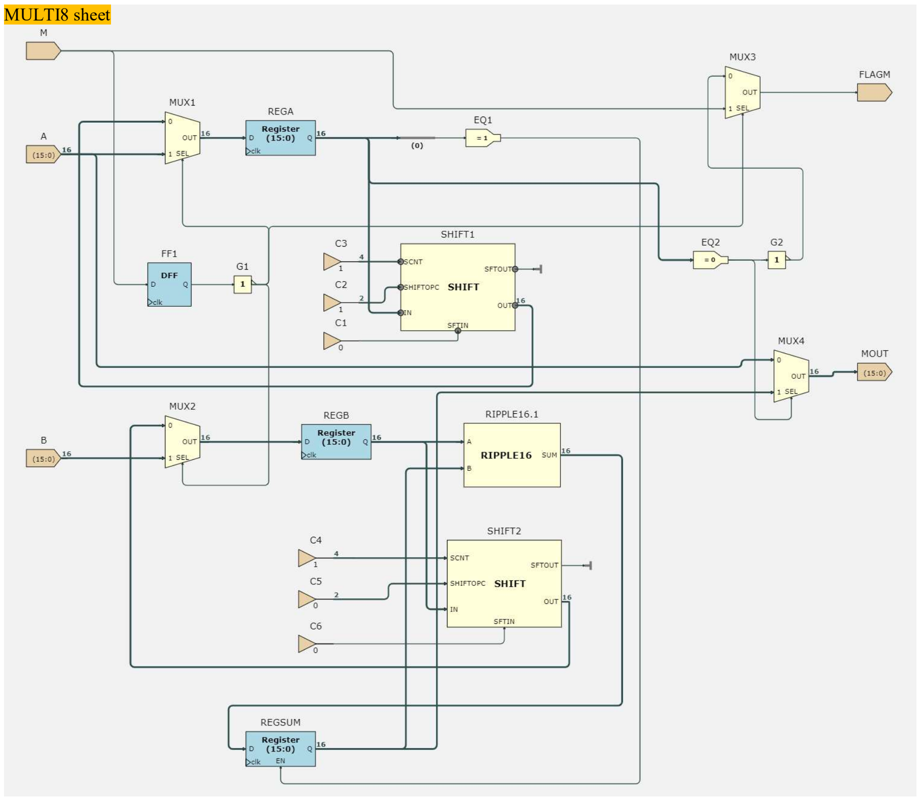
RIPPLE16 sheet



Note:

- Ripple adder used to sum B and RESULT when A[0] = 1
- We ignore the carry out of the final ADD8, it is not needed

MULTI8 sheet



Note:
Implemented "Russian Peasant Multiplication" method via hardware. This is for 8 bit multiplication.

MUX1 and MUX2 help with initialising the register at the beginning, then we pass through the output of the register after the first clock

SHIFT1 = LSR
SHIFT2 = LSL

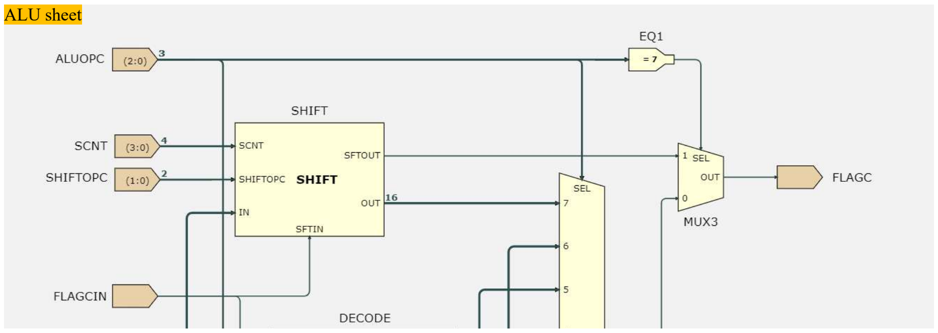
Bus comparator EQ1 acts as an enable for writing to the SUM register. Only when value A is odd will we write into the SUM register the sum of B and SUM

This implementation works for both signed and unsigned multiplication, however it takes longer for signed multiplication as we have to wait till A has been shifted all the way to 0x0000 from 0xFFXX

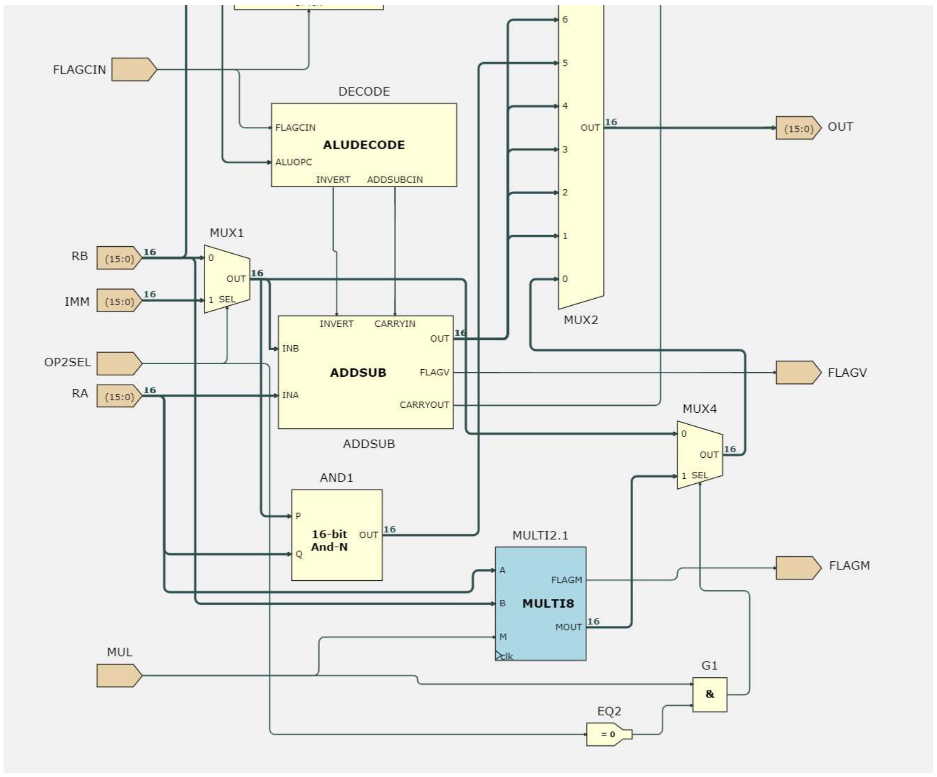
MUX3 ensures that FLAGM is not one clock cycle behind MUL so that we are able to stall PC immediately

FlagM = 1 (Multiplication is operating, therefore stall PC)
FlagM = 0 (Multiplication is not operating)

ALU sheet



New:



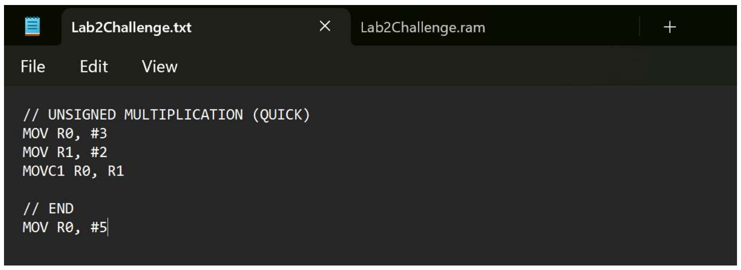
New:
An additional MUX (MUX4 in sheet)

Only the result of othe multiplication is returned to terminal 0 of MUX2, depending if MUL is asserted or not.
- MUL = 1 - Multiplication is enabled

To differentiate between using the immediate or the multiplication operation, we need the bus comparator on OP2SEL to ensure it is 00 and MUL is asserted to select our multiplication output

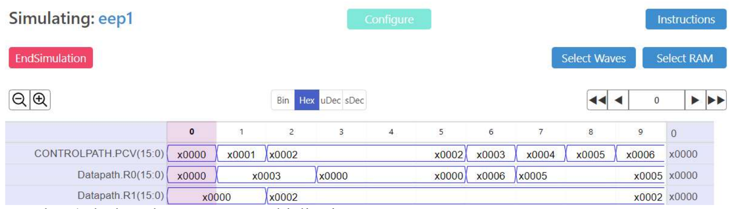
Testing of new multiplication hardware (8 bit multiplication):

Limitation of implementation:
- There is rubbish value (0x0000) while computing the multiplication
- Number of clock cycles needed for operation is highly dependent on the values used


Lab2Challenge.txt   Lab2Challenge.ram

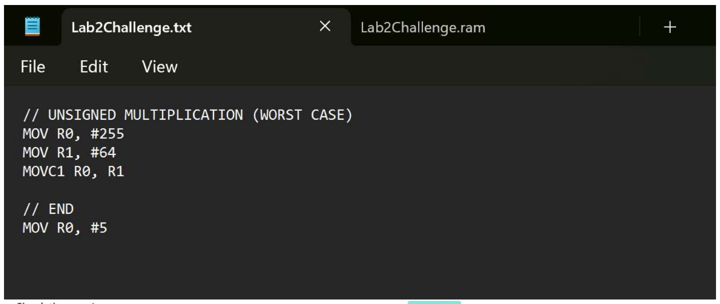
```
// UNSIGNED MULTIPLICATION (QUICK)
MOV R0, #3
MOV R1, #2
MOVC1 R0, R1

// END
MOV R0, #5
```


Simulating: eep1

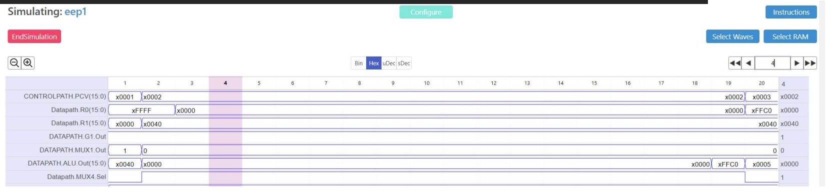
```
                      0       1       2                      5       6       7       8       9       0
CONTROLPATH.PCV(15:0) x0000   x0001   x0002                  x0002   x0003   x0004   x0005   x0006   x0000
Datapath.R0(15:0)     x0000       x0003       x0000          x0000   x0006   x0005           x0005   x0000
Datapath.R1(15:0)         x0000       x0002                                                  x0002   x0000
```

Takes 4 clock cycles to execute multiplication

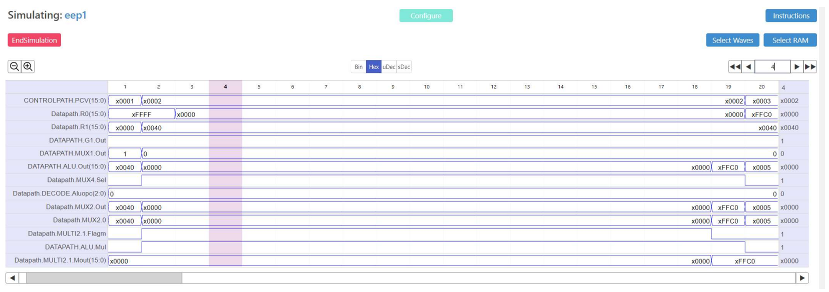

Lab2Challenge.txt   Lab2Challenge.ram

```
// UNSIGNED MULTIPLICATION (WORST CASE)
MOV R0, #255
MOV R1, #64
MOVC1 R0, R1

// END
MOV R0, #5
```


Simulating: eep1

```
                        1      2      3   4   ...   18     19     20     4
CONTROLPATH.PCV(15:0)   x0001  x0002                       x0002  x0003  x0002
Datapath.R0(15:0)          xFFFF      x0000                x0000  xFFC0  x0000
Datapath.R1(15:0)       x0000  x0040                              x0040  x0040
DATAPATH.G1.Out                                                          1
DATAPATH.MUX1.Out       1      0                                  0      0
DATAPATH.ALU.Out(15:0)  x0040  x0000                x0000  xFFC0  x0005  x0000
Datapath.MUX4.Sel                                                        1
```



9 cycles to execute multiplication in worst case (where Ra needs to shift all 8 bits to complete operation)