Challenge

Sunday 16 March 2025

11:27

Booth's algorithm:

Here I have used Booth's multiplication algorithm to reduce the number of clock cycles in multiplication in the EEP1:

I have decided to consider 2 bits at of the multiplier at once to complete the multiplication.

00	Shift the value of the multiplicand to the left twice without altering the value of R2, R3 (equivalent of 0 x A)
01	Add multiplicand once and then shift twice (equivalent to 1 x A)
10	Shift multiplicand once, add multiplicand then shift again once (equivalent to 2 x A)
10	Shift multiplicand once, add multiplicand, shift multiplicand once again then add once more (equivalent to 4 x A)

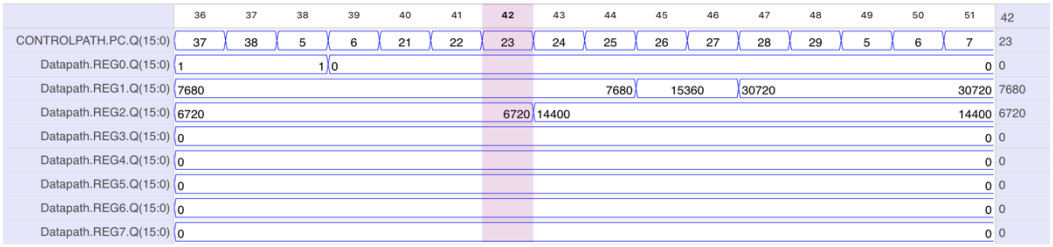
```
MOV R0 #120 //stores multiplier
MOV R1 #120 //stores multiplicand
MOV R2, #0 //least signficant bits of the answer
MOV R3, #0 //most significant bits of answer
MOV R5, #0 //stores 0 as the most significant bits of multiplicand
LSR R0, R0, #1 //check if least significant bit of 1
JCS 15 //jumps to cases 01 or 11 if LSB of R0 is 1
LSR R0, R0 #1 //check if next significant bit is 1
JCS 6 //jumps to case 10 if second LSB of R0 is 1
ADD R1, R1, R1 //case 00 shifts the values of R4 twice
ADC R5, R5, R5 //must add carry onto R5 per shift
ADD R1, R1, R1 //no alter to R2, R3 since "0" so technically not multiplying
ADC R5, R5, R5
JMP -8 //jumps back to test next least signifcant bits
ADD R1, R1, R1 //case 10 shifts left once (0 part)
ADC R5, R5, R5
ADD R2, R1, R2 //then adds the values onto R2 and R3 (1 part)
ADC R3, R5, R3
ADD R1, R1, R1 //must always shift twice since always takes account 2 bits
ADC R5, R5, R5
JMP -15 //jumps back to test next least signifcant bits
LSR R0, R0, #1 //checks if second least signifcnant bit now is a 1
JCS 8
ADD R2, R1, R2 //case 01 adds value onto R2 and R3 (1 part)
ADC R3, R5, R3
ADD R1, R1, R1 //shifts compulsary shift twice (0 part)
ADC R5, R5, R5
ADD R1, R1, R1
ADC R5, R5, R5
JMP -24 //jumps back to test next least signifcant bits
ADD R2, R1, R2 //case 11 adds values onto R2 and R3, shifts then repeats
ADC R3, R5, R3
ADD R1, R1, R1
ADC R5, R5, R5
ADD R2, R1, R2
ADC R3, R5, R3
ADD R1, R1, R1
ADC R5, R5, R5
JMP -33 //jumps back to test next least signifcant bits
```

```
0x00 0x0178
0x01 0x0378
0x02 0x0500
0x03 0x0700
0x04 0x0b00
0x05 0x7011
0x06 0xc40f
0x07 0x7011
0x08 0xc406
0x09 0x1224
0x0a 0x3ab4
0x0b 0x1224
0x0c 0x3ab4
0x0d 0xc0f8
0x0e 0x1224
0x0f 0x3ab4
0x10 0x1248
0x11 0x3a6c
0x12 0x1224
0x13 0x3ab4
0x14 0xc0f1
0x15 0x7011
0x16 0xc408
0x17 0x1248
0x18 0x3a6c
0x19 0x1224
0x1a 0x3ab4
0x1b 0x1224
0x1c 0x3ab4
0x1d 0xc0e8
0x1e 0x1248
0x1f 0x3a6c
0x20 0x1224
0x21 0x3ab4
0x22 0x1248
0x23 0x3a6c
0x24 0x1224
0x25 0x3ab4
0x26 0xc0df
```

The LSBs are checked using multiple right shifts that set the carry flag and then jump to the relevant loop. The advantage of this, is that the right shift is done with the same instruction as the compare.

I tested with 120*120=14400 as 120 contains all cases. This required 43 clock cycles to execute.

120 = 0b01111000



Waveform simulator of 120 x 120

Advantages:

1. **Reduced Clock Cycles**:
    - **Processes two bits at a time, reducing the number of iterations by half compared to the standard shift-and-add method.**
2. **Efficient for Hardware**:
    - **Requires fewer additions/subtractions**
3. **Handles Signed Numbers.**

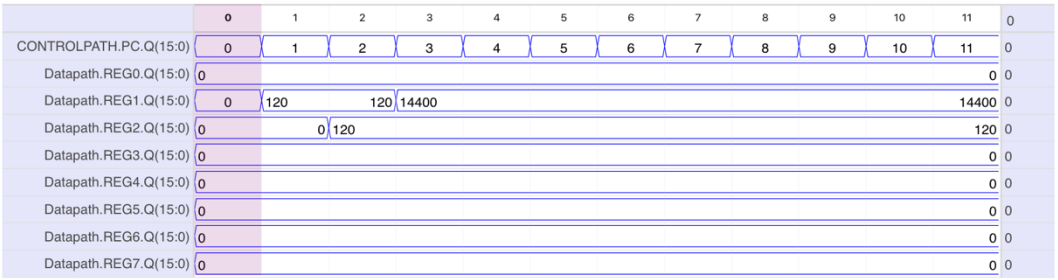
Other binary multipliers such as the Wallace tree multiplier uses partial products and adders to complete the multiplication. It is a bit like multiplying normal numbers where you would multiply by each bit and shift that outcome. E.g. A x 0 = 0 and A x 1 = A

This is faster than the purely software implementation.

However there are some downsides due the number of hardware (binary adders needed).

Wallace tree multiplier:

The Wallace tree multiplier uses partial products (e.g. each bit of the multiplier multiplied by the multiplicand to complete the multiplication) and uses a combination of half adders and full adders with a carry to complete the multiplication.



It is seen that whilst the number of clock cycles to complete the calculation has dramatically increased (from 43 using booths algorithm to 3) the amount of hardware components is very large so may become expensive.

```
MOV R1, #120
MOV R2, #120
MOVC1 R1, R2
```

.txt of my code

Using the fact that 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 in my case is used to multiply Ra by Rb and store the value in Ra.

If MOV is selected from the multiplexer and values INS(14:12) = 001 which represents C = 1 and there is no immediate value (bus select of INS8) MSEL will be 1 which is the select line for the multiplier to be inputted into the ALU.

The ALU will then select to output the output of the MULTIPLIER is MSEL is 1.