Arithmetic - Integer Multiplication

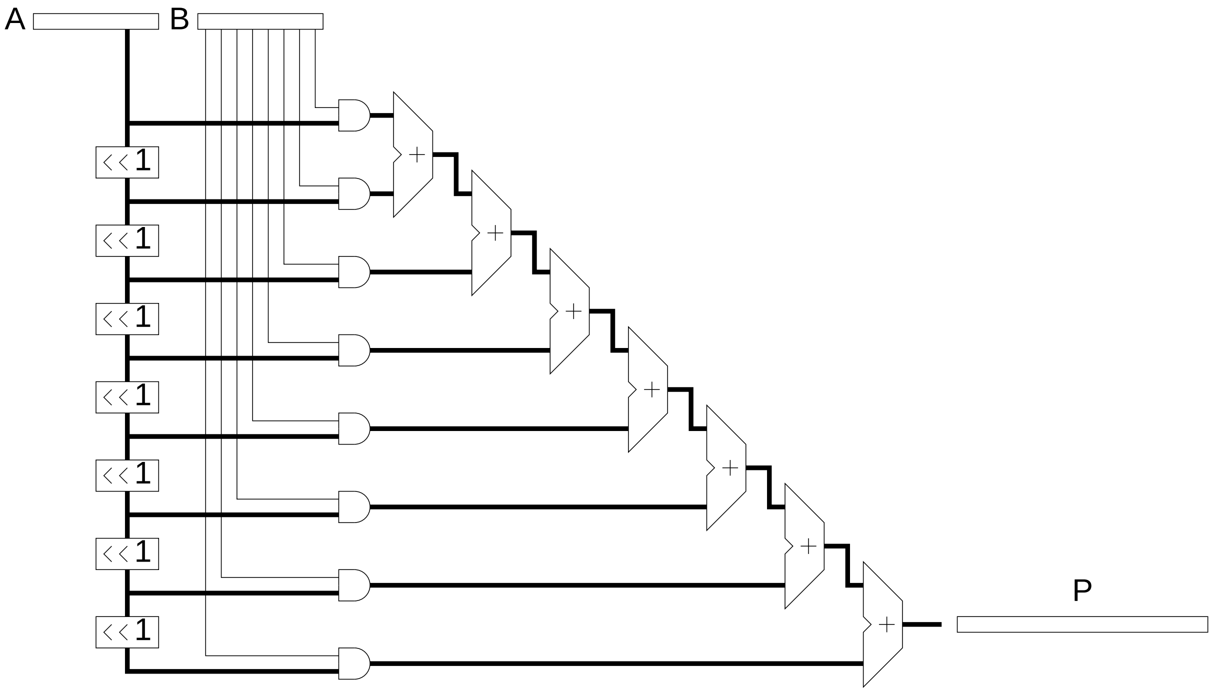
- Simple multiplication algorithm.

```
        01010111×        87×
        00100101         37
        --------         --
        01010111        609
       00000000        261
      01010111         ----
     00000000          3219
    00000000
   01010111
  00000000
 00000000
----------------
0000110010010011
```

  – two n-bit numbers will produce a $2n$-bit result.

Arithmetic - Integer Multiplication

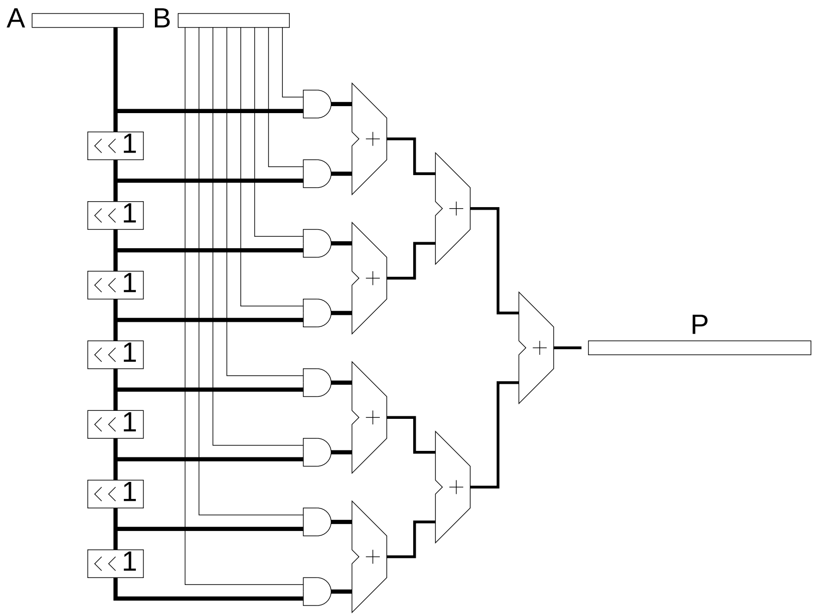
- Implementation



  – rather slow for single cycle operation

Arithmetic - Integer Multiplication

- Parallel Implementation

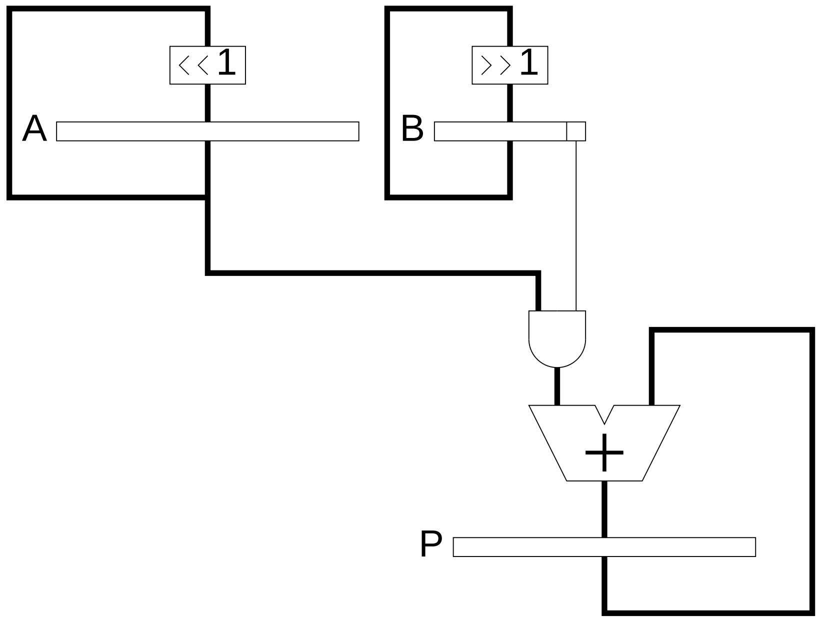


  - should be faster[1]
  - faster implementations exist using more complex building blocks

[1] note that the overall delay is less than three adder delays due to the skew of rippling carries

Arithmetic - Integer Multiplication

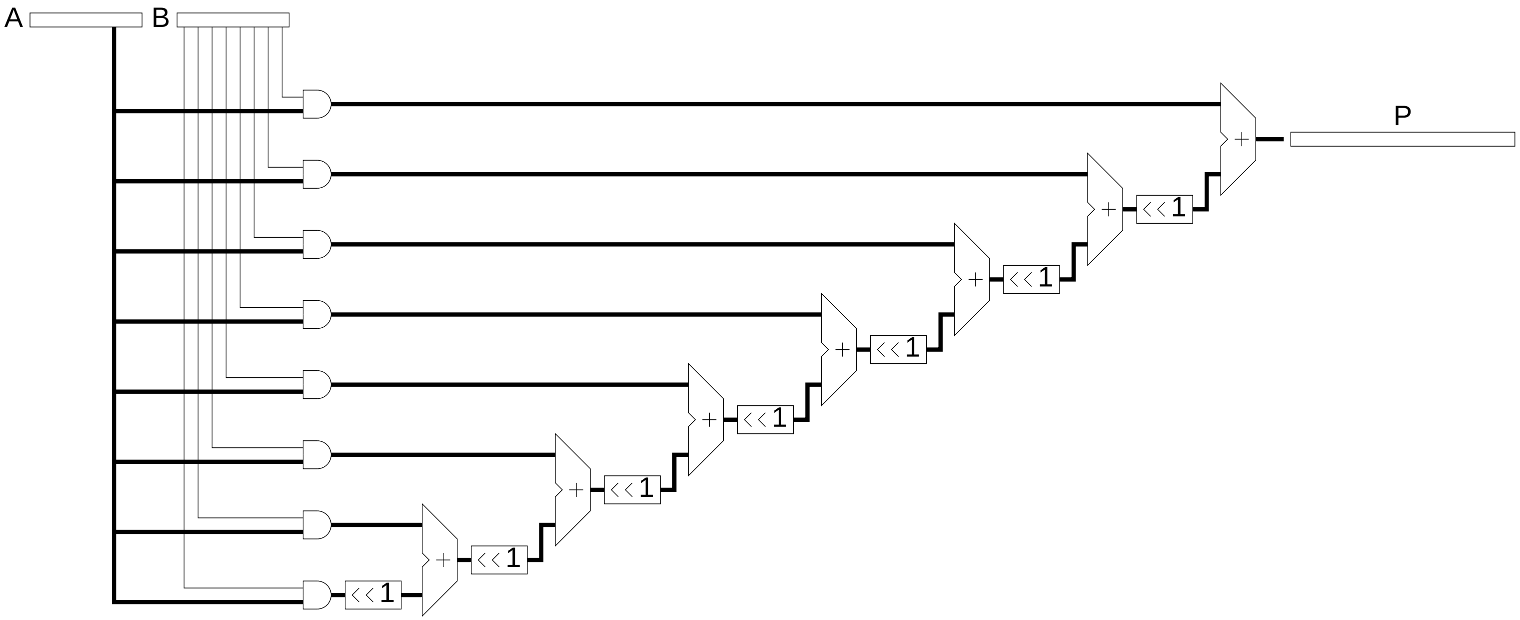
- Multi-cycle implementation



  - partial product is initially set to zero
  - calculation completes after n cycles (or when B = 0)

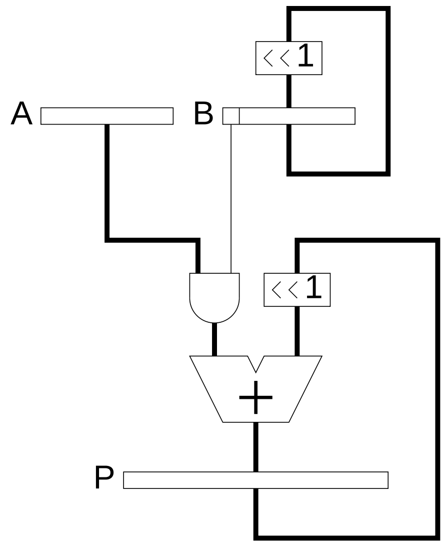
Arithmetic - Integer Multiplication

- Alternative implementation



Arithmetic - Integer Multiplication

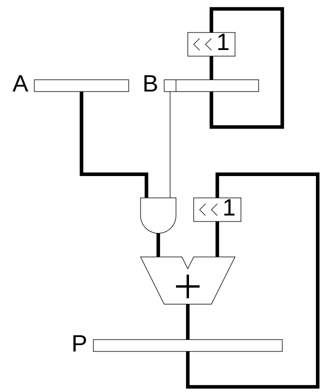
- Alternative multi-cycle implementation



  - A is not shifted giving a fixed width value on one input of the adder
  - the top n-1 bits of a dedicated adder could be half adders

Arithmetic - Integer Multiplication

- Software implementation



– double word result

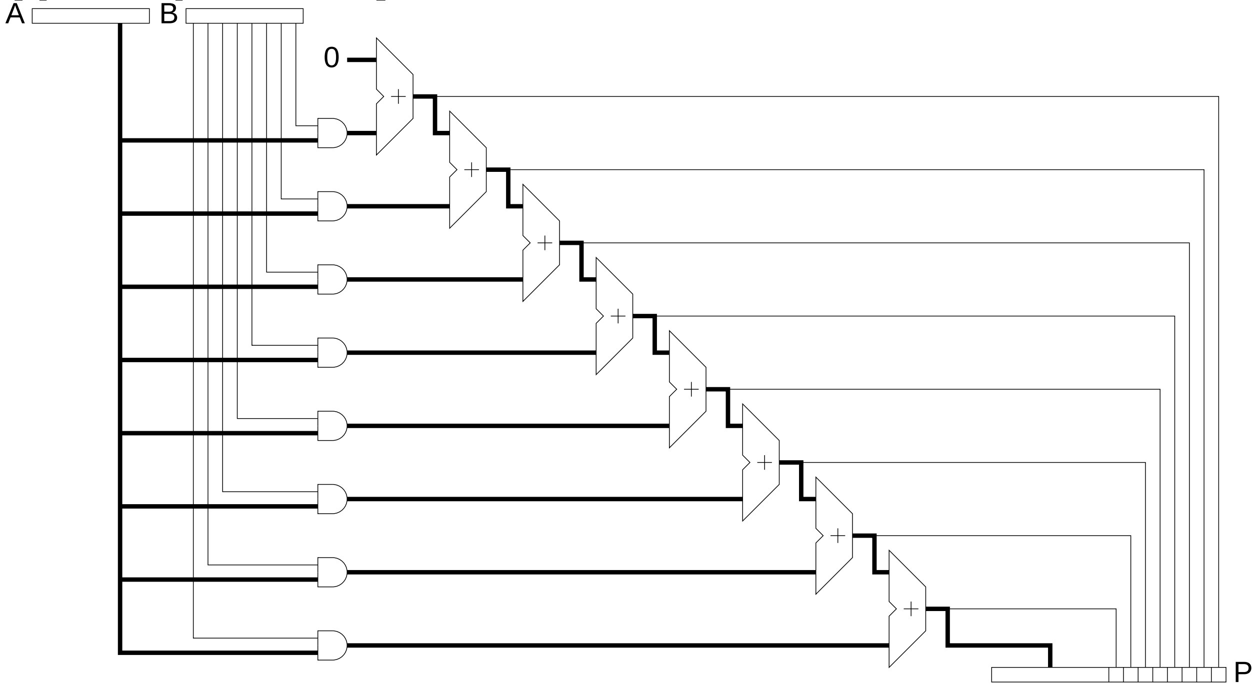
- - use multi word shift left and add

– single word result

- - check for overflow on each addition and each left shift of P

Arithmetic - Integer Multiplication

- Reduce adder widths



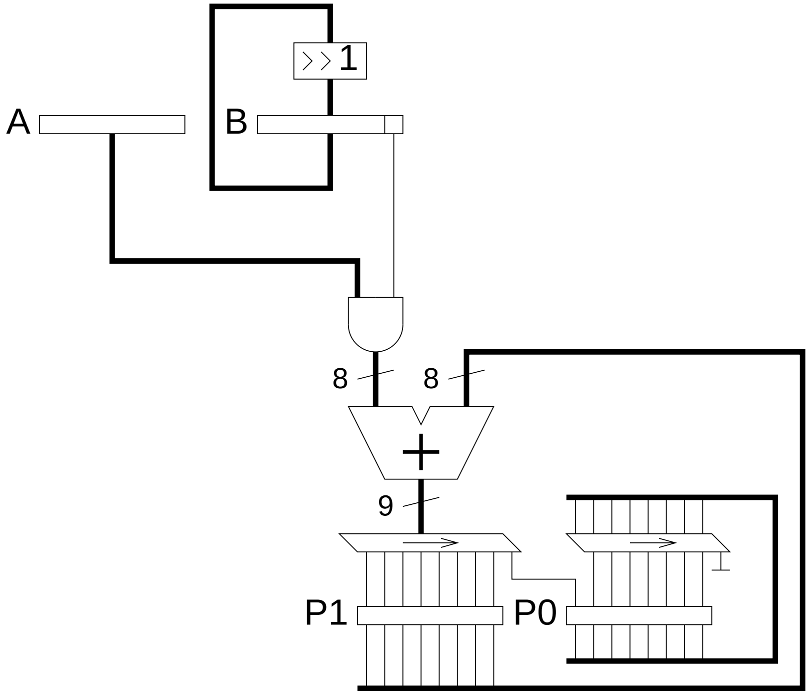
  – each adder produces an $(n+1)$-bit result

    - - the least significant bit is fed straight to P

    - - the most significant n bits are fed to the next adder

Arithmetic - Integer Multiplication

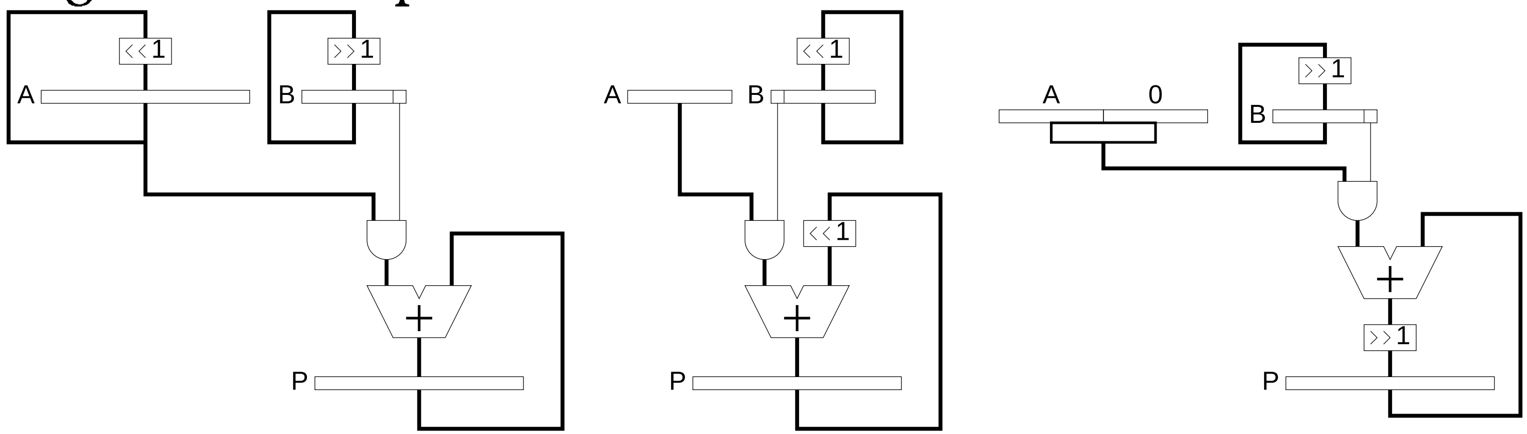
- Multi-cycle implementation



  – after each addition the least significant bit of the result is placed in P0 which is the least significant word of the partial product

Arithmetic - Integer Multiplication

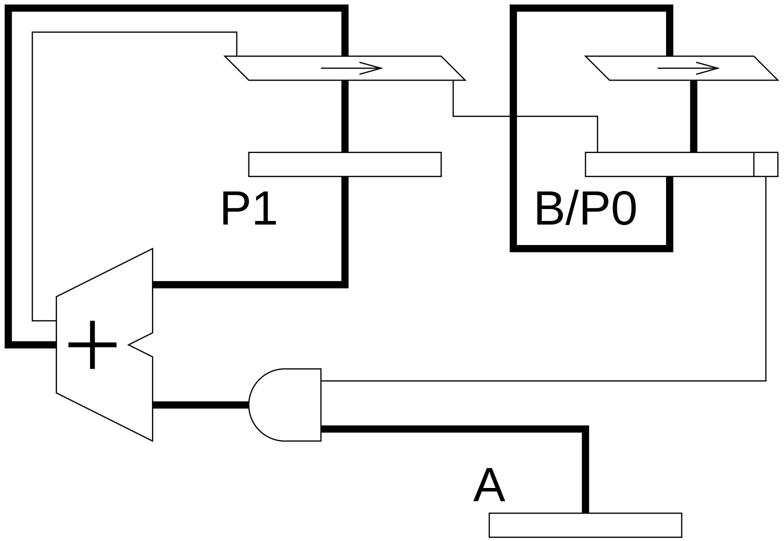
- Algorithm Comparison



  - justify each term before addition
  - don't justify terms
    - - beginning with most significant term add each term to least significant slot and shift left
    - - beginning with least significant term add each term to most significant slot and shift right

Arithmetic - Integer Multiplication

- Standard implementation



  - P0 uses the same register as B
  - standard n-bit adder with carry out

Arithmetic - Integer Multiplication

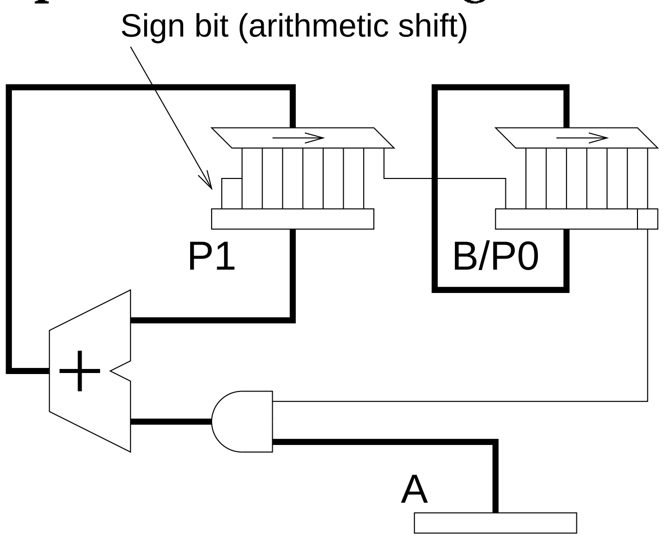
- Multiplication of 2's complement numbers.

```
        11010111×            -41×
        00100101              37
        --------             ----
11111111111010111            -287
0000000000000000            -123
111111111010111             -----
00000000000000              -1517
0000000000000
111111010111
00000000000
0000000000
-----------------
1111101000010011
```

  - allowing for a negative multiplicand requires only minor modifications (sign extension is used to retain sign information).
  - *coping with a negative multiplier is somewhat harder and will not be covered here.*

Arithmetic - Integer Multiplication

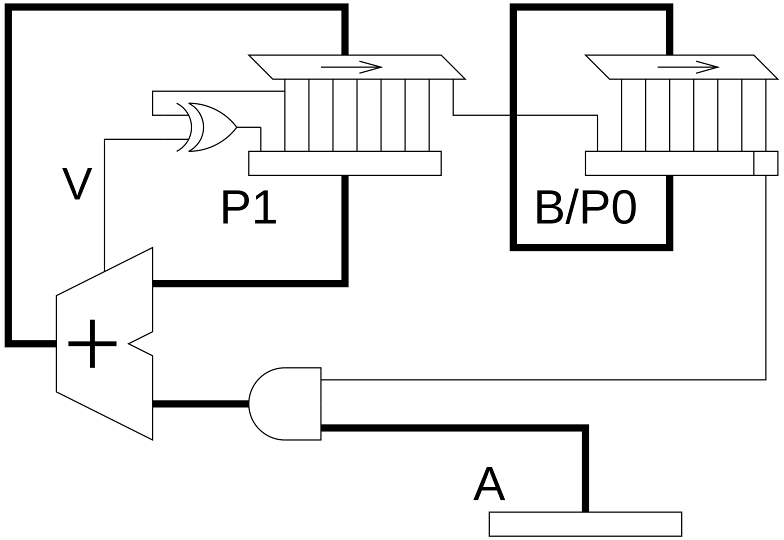
- Possible implementation?
  signed multiplicand (A) - unsigned multiplier (B)



  - we see that we need to perform an arithmetic shift on the result of the addition rather than using the carry out.
  - unfortunately this doesn't account for a possible overflow (which will give the wrong sign)

Arithmetic - Integer Multiplication

- Standard implementation
  signed multiplicand (A) - unsigned multiplier (B)



  – use overflow information to recover the correct sign information (all other bits will be correct when overflow occurs)

ALU Functionality

Where a single cycle multiply is not appropriate we may implement multiply as:

- A multi-cycle instruction
  - The 6809 `MUL` instruction performs a complete multiplication of two unsigned 8-bit numbers yielding a 16-bit result.

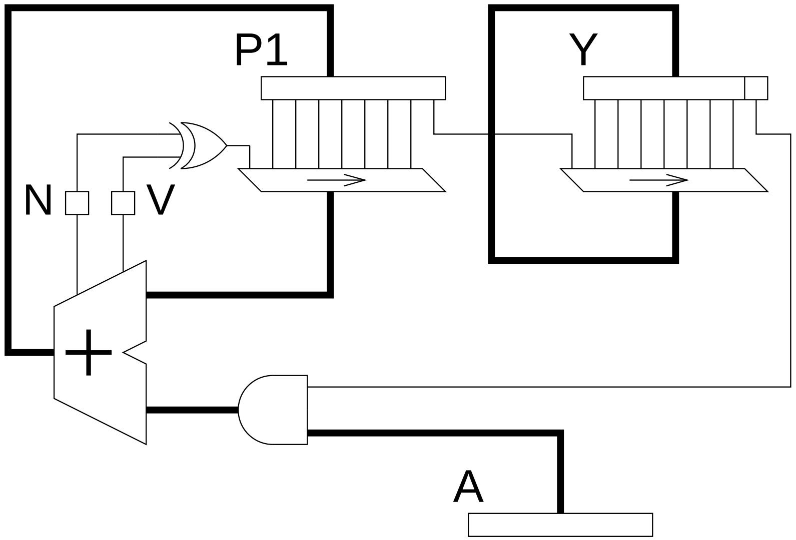
$$ACCA' : ACCB' \leftarrow ACCA \times ACCB$$

or

- A single-cycle multiply step instruction
  - The SPARC `MULScc P1,A,P1` performs a single multiply step instruction. The instruction uses a dedicated `Y` register which contains `B/P0`.

Arithmetic - Integer Multiplication

- SPARC implementation



  - takes a step longer due to arrangement of registers and shifters.
  - since neither signed nor unsigned numbers are dealt with properly in this system a few additional instructions are used to post-process (fiddle) the answer for signed or unsigned multiplication.