Binary arithmetic: most operations are familiar

Each place in a binary number has value 2^n

```
5    =     00000101   = 1 + 4
19   =     00010011   = 1+2+16

5 +        00000101
19         00010011
   =       00011000     24 = 16+8

19 -       00010011
5    =     00000101
           00001110     14 = 8+4+2
```

Addition often requires a 'carry'

Subtraction may require a 'borrow'

Representation of negative numbers: there are a number of ways we might do this:

1. **Use of a 'sign bit' (this is just like having a sign for the number)**

   ```
   -5   =     10000101
   ```

   Note that addition and subtraction are somewhat complex (and multiplication and division). Generally must strip the sign bit, do the operation, then figure out the sign of the result.

2. **'One's Complement': invert each bit. We won't have much to say about this.**

3. **'Two's Complement': invert each bit and add one.**

What happens if we do this operation:

```
 5          00000101
-19         00010011
   =        11110010
```

Note two things about this operation:

1. **We had to invent a 'borrow' bit from the left**
2. **What is left is the two's complement representation of -14:**

```
14    =     00001110
-14   =     11110001
                  +1
   =        11110010
```


Two's complement is consistent and reversible:

```
5   =     00000101
-5  =     11111010 +1    =  11111011
5   =                       00000100 +1    =  00000101
```

Addition and Subtraction between two's complement numbers works:

```
-5        11111011
+(-19)    11101101
=         11101000      (which is -24)
          00010111+1 =  00011000 = 16+8
```

5.5	=	00000101.1
5.0	=	00000101.0
-5.0	=	11111010.1
	+	1
	=	11111101.0

In many cases we want to extend a number: to employ more 'binary places' to represent a number. How do we do this extension?

To extend a number (represent with more places) without changing value:

If the number is:	Extend to <u>left</u>	Extend to <u>right</u>
Positive	zeros	zeros
Negative	ones	zeros

19	=	00010011
X -5	=	11111011
	=	00010011
	+	00010011
	=	00111001
	+	00010011
	=	00011010001
	+	00010011
	=	001000000001
	+	00010011
	=	0010001100001
	+	00010011
	=	00100100100001
	+	00010011
	=	001001010100001

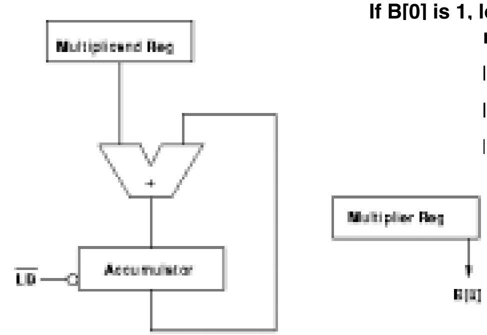
```
10100001 is the
negative of:
  01011110+1
= 01011111
= 64+16+8+4+2+1=95
```


Now consider how we might do a simple multiplication

9	=	00001001	
X 13	=	00001101	
	=	00001001	
	+	00001001	
	=	0000101101	
	+	00001001	
	=	00001110101	(117)

This involves shifting the top number repeatedly to the left and adding it to the partial sum. This works well and requires a shift register as wide as the product as well as an accumulator for the partial and final product

Here is a hardware description of a multiplier



If B[0] is 1, load (add # to
r),
licand left,
lier right,
l done

An alternative is to shift the partial product to the right

```
  9    =    00001001
X 13   =    00001101
       =    0000001001
       +    00001001
       =    00000101101
       +    00001001
       =    00001110101    (same number shifted)
```


Here is how it would work for negative numbers. We must extend the sign (put one's in as we add places to the number

```
-9     =       11110111
X 13   =       00001101
       =     1111110111    (remember sign extension)
       +     11110111
       =    11111010011
       +    11110111
       =   111110001011    (-117)
(-)        000001110101
```


'sign extension' consists of shifting ones into the MSB if it is a negative number.

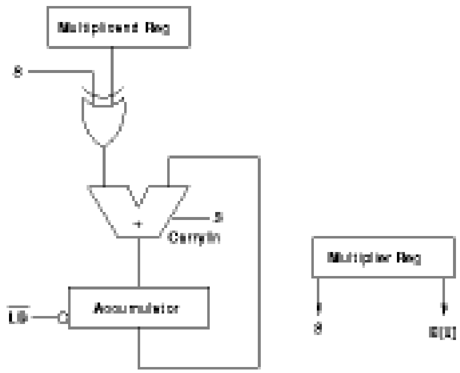
```
-9     =     11110111
X 13   =     00001101
       =     1111110111    (remember sign extension)
       +     11110111
       =     11111010011
       +     11110111
       =    111110001011    (-117)
(-)         000001110101     = 1+4+16+32+64=117
```


Multiplication of Two's Complement number by sign/magnitude number:

This is one case that works fairly well

1. Use the sign/magnitude number as the multiplier
2. If MSB is 1 (negative number), do the two's complement thing on the multiplicand

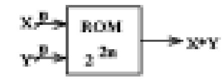
The XOR complements each bit of the multiplicand if S=1 and the Carry in adds S (1 if S is set). If the multiplier is positive, the multiplicand is not complemented and zero is carried in.



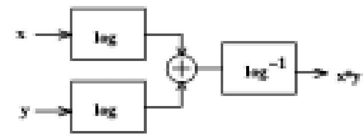
Many problems require multiplication. In fairness to what we have just said, there are

Choices:

Table Lookup
- **Each input is N bits wide**
- **ROM must store 2^N answers**
- **Fast but uses a lot of Silicon**



Log converter
- **Convert to logs**
- **Add**
- **Antilog Converter**
- **Fast, maybe uses less Silicon**
- **Precision uncertain**
- **What to do about negative num**



More Multiplication Choices

Add X input Y times
- **Load Y into downcounter**
- **Add X each time while counting down**
- **Stop when counter gets to zero**
- **Real estate cheap**
- **Time uncertain (could be very long)**
- **And what about negative numbers?**

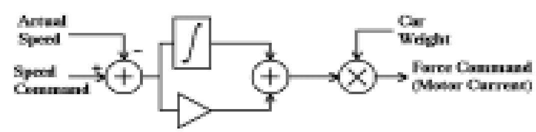
Shift and Add
- **This is the technique we have been using**
- **Technique you learned in elementary school**
- **Takes as many cycles as there are bits**
- **Uses a single register for multiplier and accumulator**
- **If care is taken with sign extension, can handle negative numbers consistently with others.**

It is Time for an example: so here is a physical system to control:

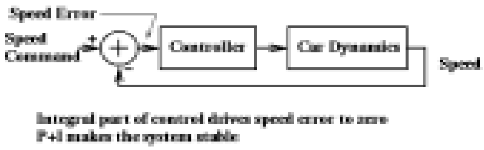
Control for a trolley car (Light Rail Vehicle)
- **Control FORCE applied to wheels**
- **Speed command by driver (the 'Go lever')**
- **Accommodates car weight: The difference between commanded speed and actual car speed should be <u>acceleration</u>**

  F = MA (required force is acceleration times car mass

- **We will also use PI control:**
  - **Integral part drives error to zero**
  - **Proportional part gives stability**



Uses feedback control

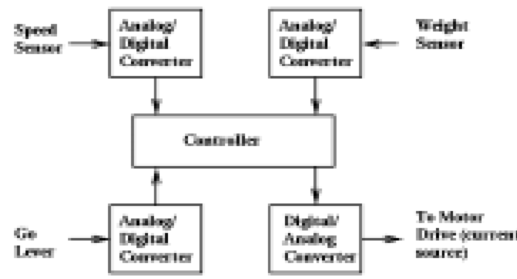


By measuring actual speed and feeding that back to the PI controller, we can drive speed error (exponentially) to zero.

So in this example we will examine how to build the controller. We assume a highly ideal drive, in which drive force is directly proportional to commanded motor current.

As with any other design, we start with a high level block diagram: here are the inputs and outputs.

Speed sensor and 'Go lever' are actual and required speed: through A/D
Weight is measured by a load cell or air support system pressure
Output is current command to motor/drive (which we assume works well)



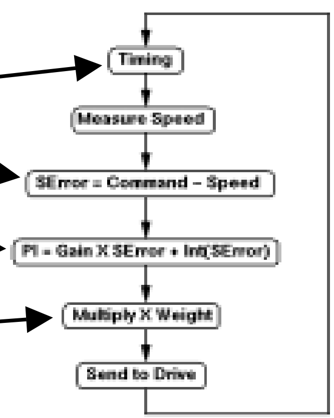
Here is a flow chart for our system:

Timing establishes a fixed interval over which our control system does its thing.

Speed error is difference between measured and commanded

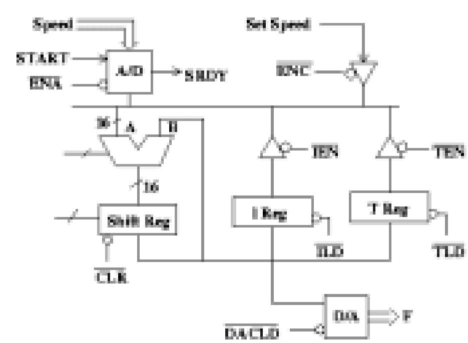
PI is just addition of the proportional and integrated signal. Integrated is added discrete signals here

Ha: multiply will be interesting



Here is a possible Data Path for our PI Controller

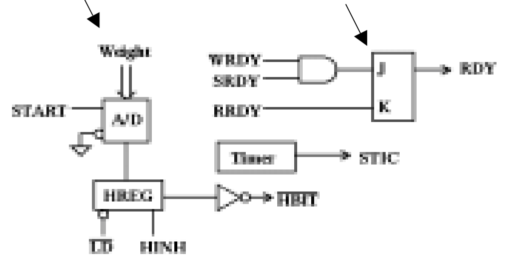
Integral approximated by a sum



PI Controller Data Path (B)

When done with the PI, we must multiply by weight.

This fragment of data path handles that multiply as part of a shift and add routine. Note there is also a synchronizer here



$$F = Weight * [(set - speed)/4 + \sum_{i=-\infty}^{now} (set - speed)/16]$$

Here is where the math gets done

ALU Controls:

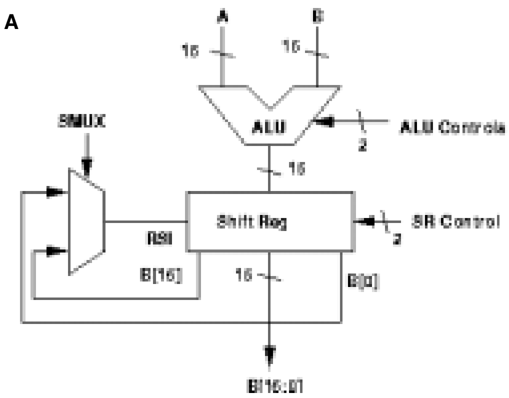
00	F = Invert A
01	F = A + 1
10	F = A + B
11	F = A - B

SR Controls

00	Hold
01	SHIFT R
10	CLEAR
11	LOAD

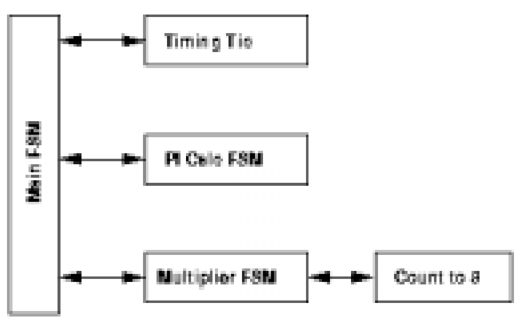
SMUX

0	Rotate
1	Sign Extend



Note: we don't use all of these!

We could control this with one large FSM, but it seems reasonable to break the control down to several smaller (more easily developed and tested) FSM's, which must then be coordinated



Here is my first guess at the 'Main' FSM that coordinates all. It starts other processes and waits for them to finish before going on. Note the TIC is produced by an FSM (a counter) that is NOT controlled by the main FSM.