Branches

- Conditional control transfers

Four basic conditions:

N -- negative
Z -- zero
V -- overflow
C -- carry

Sixteen combinations of the basic four conditions:

1.	Always	Unconditional	`AlwaysBranch`
2.	Never	NOP	`Never Branch`
3.	Not Equal	~Z	`BNZ: Branch Not Zero`
4.	Equal	Z	`BZ: Branch Zero`
5.	Greater	~[Z + (N ⊕ V)]	
6.	Less or Equal	Z + (N ⊕ V)	
7.	Greater or Equal	~(N ⊕ V)	
8.	Less	N ⊕ V	
9.	Greater Unsigned	~(C + Z)	
10.	Less or Equal Unsigned	C + Z	
11.	Carry Clear	~C	`BNC: Branch Not Carry`
12.	Carry Set	C	`BC: Branch Carry`
13.	Positive	~N	`BP: Branch Plus`
14.	Negative	N	`BM: Branch Minus`
15.	Overflow Clear	~V	`BNV: Branch Not Overflow`
16.	Overflow Set	V	`BV: Branch Overflow`

1.	Always	Unconditional
2.	Never	NOP
3.	Not Equal	~Z
4.	Equal	Z
5.	Greater	~[Z + (N ⊕ V)]
6.	Less or Equal	Z + (N ⊕ V)
7.	Greater or Equal	~(N ⊕ V)
8.	Less	N ⊕ V
9.	Greater Unsigned	~(C + Z)
10.	Less or Equal Unsigned	C + Z
11.	Carry Clear	~C
12.	Carry Set	C
13.	Positive	~N
14.	Negative	N
15.	Overflow Clear	~V
16.	Overflow Set	V

```
BG:   Branch Greater
BLE:  Branch LT or EQ
BGE:  Branch GR or EQ
BL:   Branch Less Than
 (for signed numbers)
```

1. Always	Unconditional	(check for unsigned numbers, after executing (A - B))
2. Never	NOP	
3. Not Equal	~Z	BNE: Branch Not Equal
4. Equal	Z	BE: Branch Equal
5. Greater	~[Z + (N ⊕ V)]	
6. Less or Equal	Z + (N ⊕ V)	
7. Greater or Equal	~(N ⊕ V)	
8. Less	N ⊕ V	
9. Greater Unsigned	~(C + Z)	BH: Branch Higher Than (C=0)
10. Less or Equal Unsigned	C + Z	BLE: Branch LT or EQ
11. Carry Clear	~C	BHE: Branch HT or EQ
12. Carry Set	C	BL: Branch Less Than (Borrow bit C=1)
13. Positive	~N	
14. Negative	N	
15. Overflow Clear	~V	
16. Overflow Set	V	

Motivation for Booth's Algorithm

- Example 2 x 6 = 0010 x 0110:

```
              0010
      x       0110
    ---------------
      +       0000     shift (0 in multiplier)
      +        0010    add (1 in multiplier)
      +       0010     add (1 in multiplier)
      +      0000      shift (0 in multiplier)
    ---------------
         00001100
```

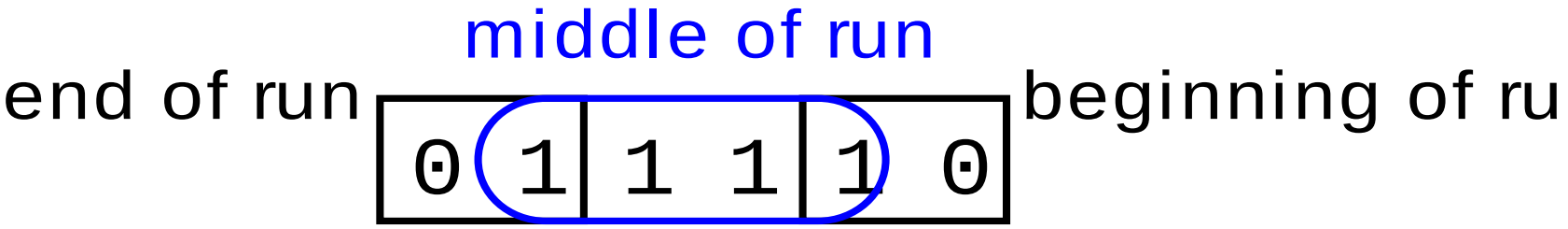
- ALU with add or subtract gets same result in more than one way:

```
6      = -2 + 8      , or
0110   = -00010 + 01000 (= 11110 + 01000)
```

- Replace a string of 1s in multiplier with an initial subtract when we first see a one and then later add for the bit after the last one. For example

```
              0010
        x     0110
    ---------------
              0000    shift (0 in multiplier)
    -          0010   sub (first 1 in multiplier)
             0000     shift (middle of string of 1s)
    +       0010      add (prior step had last 1)
    ---------------
         00001100
```

Booth's Algorithm Insight



Current Bit	Bit to the Right	Explanation	Example
1	0	Beginning of a run of 1s	0001111000
1	1	Middle of a run of 1s	0001111000
0	1	End of a run of 1s	0001111000
0	0	Middle of a run of 0s	0001111000

Originally for Speed since shift faster than add for his machine

Booth's Algorithm

1. Depending on the current and previous bits, do one of the following:

00: a. Middle of a string of 0s, so no arithmetic operations.

01: b. End of a string of 1s, so **add** the multiplicand to the left half of the product.

10: c. Beginning of a string of 1s, so **subtract** the multiplicand from the left half of the product.

11: d. Middle of a string of 1s, so no arithmetic operation.

2. As in the previous algorithm, shift the Product register right (arith) 1 bit.

Multiplicand	Product (2 x 3)	Multiplicand	Product (2 x -3)
0010	0000 0011 0	0010	0000 1101 0

Booths Example (2 x -3)

Operation	Multiplicand	Product	next?
0. initial value	0010	0000 1101 0	10 -> sub
1a. P = P - m	1110	+ 1110	
		1110 1101 0	shift P (sign ext)
1b.	0010	1111 0110 1	01 -> add
		+ 0010	
2a.		0001 0110 1	shift P
2b.	0010	0000 1011 0	10 -> sub
		+ 1110	
3a.	0010	1110 1011 0	shift
3b.	0010	1111 0101 1	11 -> nop
4a		1111 0101 1	shift
4b.	0010	1111 1010 1	done

Extra Bits

"Floating Point numbers are like piles of sand; every time you move one you lose a little sand, but you pick up a little dirt."

- How many extra bits?

IEEE: As if computed the result exactly and rounded.

- Addition:

	1.xxxxx	1.xxxxx	1.xxxxx
+	1.xxxxx	0.001xxxxx	0.01xxxxx
	1x.xxxxy	1.xxxxxyyy	1x.xxxxyyy
	post-normalization	pre-normalization	pre and post

- Guard Digits: digits to the right of the first p digits of significand to guard against loss of digits —can later be shifted left into first P places during normalization.
  - Addition: carry-out shifted in
  - Subtraction: borrow digit and guard
  - Multiplication: carry and guard, division requires guard

Rounding Digits

- Normalized result, but some non-zero digits to the right of the significand --> the number should be rounded

e.g., B = 10, p = 3:

	0 2 1.69	= 1.6900 * 10	2-bias
-	0 0 7.85	= - .0785 * 10	2-bias
	0 2 1.61	= 1.6115 * 10	2-bias

- One round digit must be carried to the right of the guard digit so that after a normalizing left shift, the result can be rounded, according to the value of the round digit

- IEEE Standard:
  four rounding modes: round to nearest (default)
  round towards plus infinity
  round towards minus infinity
  round towards 0

round to nearest:
  round digit < B/2 then truncate
  > B/2 then round up (add 1 to ULP)
  = B/2 then round to nearest even digit

it can be shown that this strategy minimizes the mean error introduced by rounding

Sticky Bit

- Additional bit to the right of the round digit to better fine tune rounding

```
  d0 . d1 d2 d3 . . . dp-1  0  0  0
+  0 .  0  0  X . . .  X   X  X  S
                          ---------
                          X  X  S
```

Sticky bit: set to 1 if any 1 bits fall off the end of the round digit

```
  d0 . d1 d2 d3 . . . dp-1  0  0  0
-  0 .  0  0  X . . .  X   X  X  0
                          ---------
                          X  X  0
```

```
  d0 . d1 d2 d3 . . . dp-1  0  0  0
-  0 .  0  0  X . . .  X   X  X  1
                          ---------
```

generates a borrow

- Rounding Summary:

Radix 2 minimizes wobble in precision

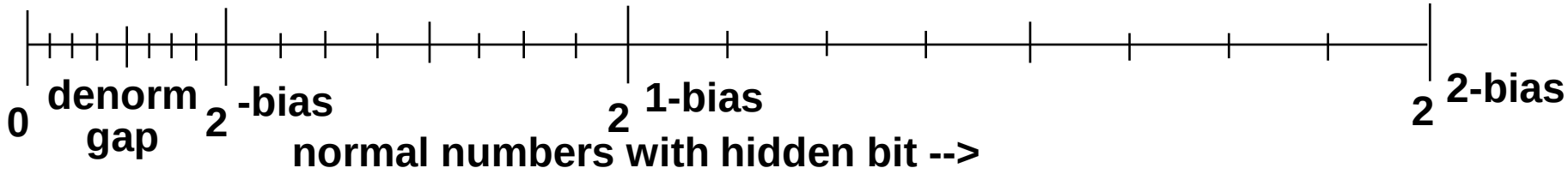
Normal operations in +, -, *, / require one carry/borrow bit + one guard digit

One round digit needed for correct rounding

Sticky bit needed when round digit is B/2 for max accuracy

Rounding to nearest has mean error = 0 if uniform distribution of digits are assumed

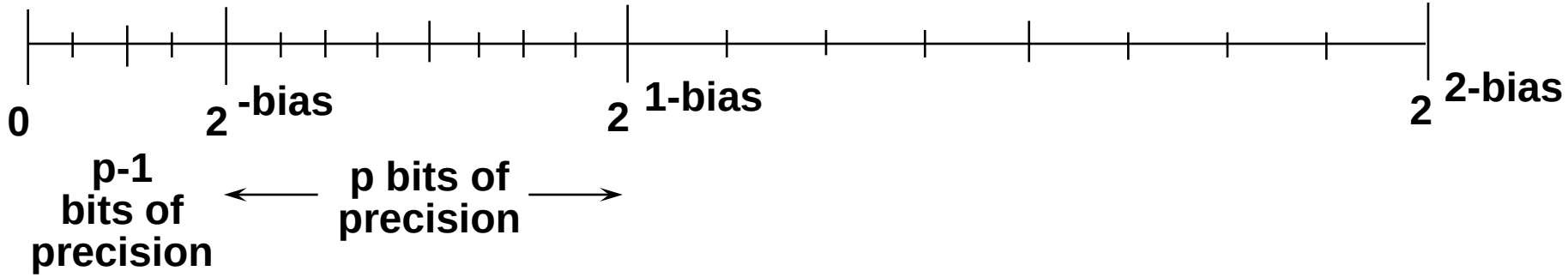
Denormalized Numbers



B = 2, p = 4

- The gap between 0 and the next representable number is much larger than the gaps between nearby representable numbers.

- IEEE standard uses denormalized numbers to fill in the gap, making the distances between numbers near 0 more alike (Gradual underflow).



same spacing, half as many values!

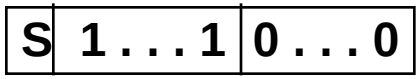
NOTE: PDP-11, VAX cannot represent subnormal numbers. These machines underflow to zero instead (Flush to zero).

Infinity and NaNs

- Result of operation *overflows*, i.e., is larger than the largest number that can be represented

overflow is not the same as divide by zero (raises a different exception)

+/- infinity



- It may make sense to do further computations with infinity
  e.g., X/0 > Y may be a valid comparison

- Not a number, but not infinity (e.q. sqrt(-4))
  invalid operation exception (unless operation is = or ≠)

NaN

HW decides what goes here

NaNs propagate: f(NaN) = NaN

Exceptions

Invalid operation:

result of operation is a NaN (except = or ≠)
inf. +/- inf.; 0 * inf; 0/0; inf./inf.; x remainder y where y = 0;
sqrt(x) where x < 0, x = +/- inf.

Overflow:

result of operation is larger than largest representable #
flushed to +/- inf. if overflow exception is not enabled

Divide by 0:

x/0 where x = 0, +/- inf.;
flushed to +/- inf. if divide by zero exception not enabled

Underflow:

subnormal result OR non-zero result underflows to 0

Inexact:

rounded result not the actual result (rounding error ≠ 0)

- **IEEE Standard --> specifies defaults and allows traps to permit user to handle the exception**

contrast with the more usual result of aborting the computation altogether!