Numeric Basics

- Representing and processing numeric data is a common requirement
  - unsigned integers
  - signed integers
  - fixed-point real numbers
  - floating-point real numbers
  - complex numbers
Unsigned Integers

- Non-negative numbers (including 0)
  - Represent real-world data
    - e.g., temperature, position, time, ...
  - Also used in controlling operation of a digital system
    - e.g., counting iterations, table indices
- Coded using unsigned binary (base 2) representation
  - analogous to decimal representation
Binary Representation

- **Decimal: base 10**
  - \(124_{10} = 1 \times 10^2 + 2 \times 10^1 + 4 \times 10^0\)

- **Binary: base 2**
  - \(124_{10} = 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0\)
  - \(= 1111100_2\)

- In general, a number \(x\) is represented using \(n\) bits as \(x_{n-1}, x_{n-2}, \ldots, x_0\), where

\[
x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_02^0
\]
Binary Representation

- Unsigned binary is a code for numbers
  - $n$ bits: represent numbers from 0 to $2^n - 1$
    - 0: 0000...00; $2^n - 1$: 1111...11
  - To represent $x$: $0 \leq x \leq N - 1$, need $\lceil \log_2 N \rceil$ bits

- Computers use
  - 8-bit bytes: 0, ..., 255
  - 32-bit words: 0, ..., ~4 billion
  - (rather recently, 64-bit words)

- Digital circuits can use whatever size is appropriate
Unsigned Integers in Verilog

- Use vectors as the representation
- Can apply arithmetic operations

```verilog
module multiplexer_6bit_4_to_1
  ( output reg [5:0] z,
    input [5:0] a0, a1, a2, a3,
    input [1:0] sel );

  always @*
    case (sel)
      2'b00: z = a0;
      2'b01: z = a1;
      2'b10: z = a2;
      2'b11: z = a3;
    endcase

endmodule
```
Octal and Hexadecimal

- Short-hand notations for vectors of bits
- Octal (base 8)
  - Each group of 3 bits represented by a digit
  - 0: 000, 1:001, 2: 010, ..., 7: 111
  - \(253_8 = 010\ 101\ 011_2\)
  - \(11001011_2 \Rightarrow 11\ 001\ 011_2 = 313_8\)
- Hex (base 16)
  - Each group of 4 bits represented by a digit
  - 0: 0000, ..., 9: 1001, A: 1010, ..., F: 1111
  - \(3CE_{16} = 0011\ 1100\ 1110_2\)
  - \(11001011_2 \Rightarrow 1100\ 1011_2 = CB_{16}\)
Extending Unsigned Numbers

- To extend an \( n \)-bit number to \( m \) bits
  - Add leading 0 bits
  - e.g., \( 72_{10} = 1001000 = 000001001000 \)

Verilog code:

```verilog
wire [3:0] x;
wire [7:0] y;
assign y = {4'b0000, x};
assign y = {4'b0, x};
assign y = x;
```
Truncating Unsigned Numbers

- To truncate from \( m \) bits to \( n \) bits
  - Discard leftmost bits
  - Value is preserved if discarded bits are 0
  - Result is \( x \mod 2^n \)

```
assign x = y[3:0];
```
Unsigned Addition

- Performed in the same way as decimal

```
  0 0 1 1 1 1 0 0 0 0
  1 0 1 0 1 1 1 1 0 0
  0 0 1 1 0 1 0 0 1 0
  0 0 1 1 1 1 0 0 0 0

  1 1 1 0 0 1
  0 1 0 0 1
  1 1 1 0 1
  1 1 0 0 1

  1 1 1 0 0 0 1 1 1 0
  1 0 0 1 1 0
```
Addition Circuits

- **Half adder**
  - for least-significant bits
    \[ s_0 = x_0 \oplus y_0 \]
    \[ c_1 = x_0 \cdot y_0 \]

- **Full adder**
  - for remaining bits
    \[ s_i = (x_i \oplus y_i) \oplus c_i \]
    \[ c_{i+1} = x_i \cdot y_i + (x_i \oplus y_i) \cdot c_i \]

<table>
<thead>
<tr>
<th>(x_i)</th>
<th>(y_i)</th>
<th>(c_i)</th>
<th>(s_i)</th>
<th>(c_{i+1})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Ripple-Carry Adder

- Full adder for each bit, $c_0 = 0$

- Worst-case delay
  - from $x_0, y_0$ to $s_n$
  - carry must ripple through intervening stages, affecting sum bits
### Improving Adder Performance

- **Carry kill:**
  \[ k_i = \overline{x_i} \cdot \overline{y_i} \]

- **Carry propagate:**
  \[ p_i = x_i \oplus y_i \]

- **Carry generate:**
  \[ g_i = x_i \cdot y_i \]

**Adder equations**

\[ s_i = p_i \oplus c_i \]
\[ c_{i+1} = g_i + p_i \cdot c_i \]
Fast-Carry-Chain Adder

- Also called Manchester adder

Xilinx FPGAs include this structure
Carry Lookahead

\[ c_{i+1} = g_i + p_i \cdot c_i \]

\[ c_1 = g_0 + p_0 \cdot c_0 \]

\[ c_2 = g_1 + p_1 \cdot (g_0 + p_0 \cdot c_0) = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \]

\[ c_3 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \]

\[ c_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0 \]
Carry-Lookahead Adder

- Avoids chained carry circuit

- Use multilevel lookahead for wider numbers
Other Optimized Adders

- Other adders are based on other reformulations of adder equations
- Choice of adder depends on constraints
  - e.g., ripple-carry has low area, so is ok for low performance circuits
  - e.g., Manchester adder ok in FPGAs that include carry-chain circuits
Adders in Verilog

- Use arithmetic “+” operator

```verilog
wire [7:0] a, b, s;
...
assign s = a + b;
```

```verilog
wire [8:0] tmp_result;
wire c;
...
assign tmp_result = {1'b0, a} + {1'b0, b};
assign c = tmp_result[8];
assign s = tmp_result[7:0];
assign {c, s} = {1'b0, a} + {1'b0, b};
assign {c, s} = a + b;
```
Unsigned Subtraction

- As in decimal

\[
\begin{align*}
    b: & \quad 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \\
    x: & \quad 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \\
    y: & \quad - \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \\
    d: & \quad 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0
\end{align*}
\]
Subtraction Circuits

- For least-significant bits

\[ d_0 = x_0 \oplus y_0 \]

\[ b_1 = x_0 \cdot y_0 \]

- For remaining bits

\[ d_i = (x_i \oplus y_i) \oplus b_i \]

\[ b_{i+1} = \overline{x_i \cdot y_i} + (x_i \oplus y_i) \cdot b_i \]

<table>
<thead>
<tr>
<th>( x_i )</th>
<th>( y_i )</th>
<th>( b_i )</th>
<th>( s_i )</th>
<th>( b_{i+1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Adder/Subtractor Circuits

- Many systems add and subtract
  - Trick: use complemented borrows
  - HW: use boolean algebra to derive equations on the right from equations on the previous page

**Addition**

\[ s_i = (x_i \oplus y_i) \oplus c_i \]

\[ c_{i+1} = x_i \cdot y_i + (x_i \oplus y_i) \cdot c_i \]

**Subtraction**

\[ d_i = (x_i \oplus \overline{y_i}) \oplus \overline{b_i} \]

\[ \overline{b_{i+1}} = x_i \cdot \overline{y_i} + (x_i \oplus \overline{y_i}) \cdot \overline{b_i} \]

- Same hardware can perform both
  - For subtraction: complement \( y \), set \( \overline{b_0} = 1 \)
Adder/Subtractor Circuits

- Adder can be any of those we have seen
  - depends on constraints

\[
\begin{array}{cccc}
  y_0 & y_1 & \cdots & y_{n-1} \\
  x_0 & x_1 & \cdots & x_{n-1} \\
  \cdots & \cdots & \cdots & \cdots \\
  x_{n-1} & x_1 & x_0 & y_{n-1} \\
  y_1 & y_0 & \cdots & y_1 \\
  y_0 & y_1 & \cdots & y_{n-1} \\
  c_0 & \text{ovf/unf} & c_n & \text{add/sub} \\
  s_0 & s_1 & \cdots & s_{n-1} \\
  s_{n-1} & s_1 & s_0 & c_0 \\
  s_{n-1}/d_{n-1} & s_1/d_1 & s_0/d_0 & \text{adder}
\end{array}
\]
Subtraction in Verilog

```verilog
module adder_subtracter ( output [11:0] s,
                          output ovf_unf,
                          input [11:0] x, y,
                          input mode );

  assign {ovf_unf, s} = !mode ? (x + y) : (x - y);
endmodule
```
Increment and Decrement

- Adding 1: set $y = 0$ and $c_0 = 1$

$$S_i = x_i \oplus c_i \quad C_{i+1} = x_i \cdot c_i$$

- These are equations for a half adder

- Similarly for decrementing: subtracting 1
Increment/Decrement in Verilog

- Just add or subtract 1

```
wire [15:0] x, s;
...

assign s = x + 1;  // increment x
assign s = x - 1;  // decrement x
```

- Note: 1 (integer), not 1'b1 (bit)
  - Automatically resized
Equality Comparison

- XNOR gate: equality of two bits
  - Apply bitwise to two unsigned numbers

- In Verilog, `x == y` gives a bit result
  - `1'b0` for false, `1'b1` for true

```
assign eq = x == y;
```
Inequality Comparison

- Magnitude comparator for \( x > y \)

\[
\begin{align*}
    x_{n-1} > y_{n-1} \\
    x_{n-1} = y_{n-1} \\
    x_{n-2} > y_{n-2} \\
    x_{n-2} = y_{n-2} \\
    \vdots \\
    x_1 > y_1 \\
    x_1 = y_1 \\
    x_0 > y_0
\end{align*}
\]
Comparison Example in Verilog

- Thermostat with target temperature
- Heater or cooler on when actual temperature is more than 5° from target

```
module thermostat ( output heater_on, cooler_on, 
                   input [7:0] target, actual );
assign heater_on = actual < target - 5;
assign cooler_on = actual > target + 5;
endmodule
```
Scaling by Power of 2

\[ x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_02^0 \]

\[ 2^k \times x = x_{n-1}2^{k+n-1} + x_{n-2}2^{k+n-2} + \cdots + x_02^k + (0)2^{k-1} + \cdots + (0)2^0 \]

- This is \( x \) shifted left \( k \) places, with \( k \) bits of 0 added on the right
  - *logical shift left* by \( k \) places
  - e.g., \( 00010110_2 \times 2^3 = 000101100000_2 \)
- Truncate if result must fit in \( n \) bits
  - overflow if any truncated bit is not 0
Scaling by Power of 2

\[ x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_02^0 \]

\[ x / 2^k = x_{n-1}2^{n-1-k} + x_{n-2}2^{n-2-k} + \cdots + x_k2^0 + x_{k-1}2^{-1} + \cdots + x_02^{-k} \]

- This is \( x \) shifted right \( k \) places, with \( k \) bits truncated on the right
  - *logical shift right* by \( k \) places
  - e.g., \( 01110110_2 / 2^3 = 01110_2 \)
- Fill on the left with \( k \) bits of 0 if result must fit in \( n \) bits
Scaling in Verilog

- **Shift-left (<<) and shift-right (>>) operations**
  - result is same size as operand

\[
\begin{align*}
\text{s} &= 00010011_2 = 19_{10} \\
\text{assign } y &= s << 2; \\
\text{y} &= 01001100_2 = 76_{10}
\end{align*}
\]

\[
\begin{align*}
\text{s} &= 00010011_2 = 19_{10} \\
\text{assign } y &= s >> 2; \\
\text{y} &= 000100_2 = 4_{10}
\end{align*}
\]
Unsigned Multiplication

\[ xy = x(y_{n-1}2^{n-1} + y_{n-2}2^{n-2} + \cdots + y_02^0) \]
\[ = y_{n-1}x2^{n-1} + y_{n-2}x2^{n-2} + \cdots + y_0x2^0 \]

- \( y_i x 2^i \) is called a partial product
  - if \( y_i = 0 \), then \( y_i x 2^i = 0 \)
  - if \( y_i = 1 \), then \( y_i x 2^i \) is \( x \) shifted left by \( i \)

- Combinational array multiplier
  - AND gates form partial products
  - adders form full product
Unsigned Multiplication

- Adders can be any of those we have seen
- Optimized multipliers combine parts of adjacent adders
Product Size

- Greatest result for $n$-bit operands:

$$ (2^n - 1)(2^n - 1) = 2^{2n} - 2^n - 2^n + 1 = 2^{2n} - (2^{n+1} - 1) $$

- Requires $2n$ bits to avoid overflow
- Multiplying $n$-bit and $m$-bit operands
  - requires $n + m$ bits

```verilog
wire [ 7:0] x; wire [13:0] y; wire [21:0] p;
...

assign p = {14'b0, x} * {8'b0, y};

assign p = x * y; // implicit resizing
```
Other Unsigned Operations

- Division, remainder
  - More complicated than multiplication
  - Large circuit area, power
- Complicated operations are often performed sequentially
  - in a sequence of steps, one per clock cycle
  - cost/performance/power trade-off
Signed Integers

- Positive and negative numbers (and 0)
- $n$-bit *signed magnitude* code
  - 1 bit for sign: 0 $\Rightarrow$ +, 1 $\Rightarrow$ –
  - $n - 1$ bits for magnitude
- Signed-magnitude rarely used for integers now
  - circuits are too complex
- Use *2s-complement* binary code
2s-Complement Representation

\[ x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_02^0 \]

- Most-negative number
  - 1000...0 = \(-2^{n-1}\)
- Most-positive number
  - 0111...1 = +2^{n-1} - 1
- \(x_{n-1} = 1 \Rightarrow \text{negative, }\)
- \(x_{n-1} = 0 \Rightarrow \text{non-negative}\)
  - Since \(2^{n-2} + \cdots + 2^0 = 2^{n-1} - 1\)
2s-Complement Examples

- **00110101**
  - \(= 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^2 + 1 \times 2^0 = 53\)

- **10110101**
  - \(= -1 \times 2^7 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^2 + 1 \times 2^0\)
    - \(= -128 + 53 = -75\)

- **00000000** = 0
- **11111111** = -1
- **10000000** = -128
- **01111111** = +127
Signed Integers in Verilog

- Use signed vectors

```verilog
wire signed [7:0] a;
reg signed [13:0] b;
```

- Can convert between signed and unsigned interpretations

```verilog
wire [11:0] s1;
wire signed [11:0] s2;
...
assign s2 = $signed(s1);  // s1 is known to be less than 2**11
...
assign s1 = $unsigned(s2); // s2 is known to be nonnegative
```
Octal and Hex Signed Integers

- Don’t think of signed octal or hex
  - Just treat octal or hex as shorthand for a vector of bits
- E.g., \(844_{10}\) is \(001101001100\)
  - In hex: \(0011\ 0100\ 1100\ ⇒ \ 34C\)
- E.g., \(-42_{10}\) is \(1111010110\)
  - In octal: \(1\ 111\ 010\ 110\ ⇒ 1726\) (10 bits)
Resizing Signed Integers

- To extend a non-negative number
  - Add leading 0 bits
  - e.g., $53_{10} = 00110101 = 000000110101$

- To truncate a non-negative number
  - Discard leftmost bits, provided
    - discarded bits are all 0
    - sign bit of result is 0
  - E.g., $41_{10}$ is $00101001$
    - Truncating to 6 bits: $101001$ — error!
Resizing Signed Integers

- To extend a negative number
  - Add leading 1 bits
    - See textbook for proof
  - e.g., $-75_{10} = 10110101 = 111110110101$
- To truncate a negative number
  - Discard leftmost bits, provided
    - discarded bits are all 1
    - sign bit of result is 1
Resizing Signed Integers

- In general, for 2s-complement integers
  - Extend by replicating sign bit
    - *sign extension*
  - Truncate by discarding leading bits
    - Discarded bits must all be the same, and the same as the sign bit of the result

```verilog
wire signed [7:0] x;
wire signed [15:0] y;
...
assign y = {{8{x[7]}}, x};
assign y = x;
...
assign x = y;
```
Signed Negation

- Complement and add 1
  - Note that $\overline{x_i} = 1 - x_i$

$$\overline{x} + 1 = -(1 - x_{n-1})2^{n-1} + (1 - x_{n-2})2^{n-2} + \cdots + (1 - x_0)2^0 + 1$$

$$= -2^{n-1} + x_{n-1}2^{n-1} + 2^{n-2} - x_{n-2}2^{n-2} + \cdots + 2^0 - x_02^0 + 1$$

$$= -(x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_02^0)$$

$$- 2^{n-1} + (2^{n-2} + \cdots + 2^0) + 1$$

$$= -x - 2^{n-1} + 2^{n-1} = -x$$

- E.g., 43 is 00101011
  so $-43$ is 11010100 + 1 = 11010101
Signed Negation

- What about negating $-2^{n-1}$?
  - $1000\ldots00 \Rightarrow 0111\ldots11 + 1 = 1000\ldots00$
  - Result is $-2^{n-1}$!

- Recall range of $n$-bit numbers is not symmetric
  - Either check for overflow, extend by one bit, or ensure this case can’t arise

- In Verilog: use $-$ operator
  - E.g., `assign y = -x;`
Signed Addition

\[
x = -x_{n-1}2^{n-1} + x_{n-2}...0 \quad \quad y = -y_{n-1}2^{n-1} + y_{n-2}...0
\]

\[
x + y = -(x_{n-1} + y_{n-1})2^{n-1} + x_{n-2}...0 + y_{n-2}...0 \quad \quad \text{yields } c_{n-1}
\]

- Perform addition as for unsigned
  - Overflow if \( c_{n-1} \) differs from \( c_n \)
  - See textbook for case analysis
- Can use the same circuit for signed and unsigned addition
## Signed Addition Examples

<table>
<thead>
<tr>
<th>0 0</th>
<th>0 0 0 0 0 0 0 0 0</th>
<th>1 1</th>
<th>1 0 0 0 0 0 0 0 1</th>
<th>0 0</th>
<th>0 0 0 0 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>72:</td>
<td>0 1 0 0 1 0 0 0 0</td>
<td>−63:</td>
<td>1 1 0 0 0 0 0 0 1</td>
<td>−42:</td>
<td>1 1 0 1 0 1 1 0 0</td>
</tr>
<tr>
<td>49:</td>
<td>0 0 1 1 0 0 0 0 1</td>
<td>−32:</td>
<td>1 1 1 0 0 0 0 0 0</td>
<td>8:</td>
<td>0 0 0 0 1 0 0 0 0</td>
</tr>
<tr>
<td>121:</td>
<td>0 1 1 1 1 0 0 0 1</td>
<td>−95:</td>
<td>1 0 1 0 0 0 0 0 1</td>
<td>−34:</td>
<td>1 1 0 1 1 1 1 0 0</td>
</tr>
</tbody>
</table>

**no overflow**

<table>
<thead>
<tr>
<th>0 1</th>
<th>0 0 1 0 0 0 0 0 0</th>
<th>1 0</th>
<th>0 0 0 0 0 0 0 0 0</th>
<th>1 1</th>
<th>1 1 1 1 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>72:</td>
<td>0 1 0 0 1 0 0 0 0</td>
<td>−63:</td>
<td>1 1 0 0 0 0 0 0 1</td>
<td>42:</td>
<td>0 0 1 0 1 0 1 0 0</td>
</tr>
<tr>
<td>105:</td>
<td>0 1 1 0 1 0 0 1 1</td>
<td>−96:</td>
<td>1 0 1 0 0 0 0 0 0</td>
<td>−8:</td>
<td>1 1 1 1 1 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>1 0 1 1 0 0 0 0 1</td>
<td></td>
<td>0 1 1 0 0 0 0 0 1</td>
<td>34:</td>
<td>0 0 1 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>

**positive overflow**

**negative overflow**

**no overflow**
Signed Addition in Verilog

- Result of + is same size as operands

```verilog
wire signed [11:0] v1, v2;
wire signed [12:0] sum;
...
assign sum = {v1[11], v1} + {v2[11], v2};
...
assign sum = v1 + v2; // implicit sign extension
```

- To check overflow, compare signs

```verilog
wire signed [7:0] x, y, z;
wire ovf;
...
assign z   = x + y;
```
Signed Subtraction

\[ x - y = x + (-y) = x + \bar{y} + 1 \]

- Use a 2's-complement adder
- Complement \( y \) and set \( c_0 = 1 \)
Other Signed Operations

- Increment, decrement
  - same as unsigned

- Comparison
  - =, same as unsigned
  - >, compare sign bits using $x_{n-1} \cdot y_{n-1}$

- Multiplication
  - Complicated by the need to sign extend partial products
  - Refer to Further Reading
Scaling Signed Integers

- Multiplying by $2^k$
  - logical left shift (as for unsigned)
  - truncate result using 2s-complement rules

- Dividing by $2^k$
  - arithmetic right shift
  - discard $k$ bits from the right, and replicate sign bit $k$ times on the left
  - e.g., $s = \text{"11110011"}$ \( \rightarrow -13 \)
    
    \[
    \text{shift\_right}(s, 2) = \text{"11111100"} \quad \rightarrow -13 / 2^2
    \]
Fixed-Point Numbers

- Many applications use non-integers
  - especially signal-processing apps
- Fixed-point numbers
  - allow for fractional parts
  - represented as integers that are implicitly scaled by a power of 2
  - can be unsigned or signed
Positional Notation

- In decimal
  \[ 10.24_{10} = 1 \times 10^1 + 0 \times 10^0 + 2 \times 10^{-1} + 4 \times 10^{-2} \]

- In binary
  \[ 101.01_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} = 5.25_{10} \]

- Represent as a bit vector: 10101
  - binary point is implicit
Unsigned Fixed-Point

- $n$-bit unsigned fixed-point
  - $m$ bits before and $f$ bits after binary point

$$x = x_{m-1}2^{m-1} + \cdots + x_02^0 + x_{-1}2^{-1} + \cdots + x_{-f}2^{-f}$$

- Range: $0$ to $2^m - 2^{-f}$
- Precision: $2^{-f}$
- $m$ may be $\leq 0$, giving fractions only
  - e.g., $m = -2$: 0.0001001101
Signed Fixed-Point

- $n$-bit signed 2s-complement fixed-point
  - $m$ bits before and $f$ bits after binary point

\[
x = -x_{m-1} 2^{m-1} + \ldots + x_0 2^0 + x_{-1} 2^{-1} + \ldots + x_{-f} 2^{-f}
\]

- Range: $-2^{m-1}$ to $2^{m-1} - 2^{-f}$
- Precision: $2^{-f}$
- E.g., 111101, signed fixed-point, $m = 2$
  - $11.1101_2 = -2 + 1 + 0.5 + 0.25 + 0.0625$
  - $= -0.1875_{10}$
Choosing Range and Precision

- Choice depends on application
- Need to understand the numerical behavior of computations performed
  - some operations can magnify quantization errors
- In DSP
  - fixed-point range affects dynamic range
  - precision affects signal-to-noise ratio
- Perform simulations to evaluate effects
Fixed-Point in Verilog

- Use vectors with implied scaling
  - Index range matches powers of weights
  - Assume binary point between indices 0 and –1

```verilog
module fixed_converter ( input [5:-7] in, output signed [7:-7] out );
    assign out = {2'b0, in};
endmodule
```
Fixed-Point Operations

- Just use integer hardware
  - e.g., addition:
    \[ x + y = (x \times 2^f + y \times 2^f) / 2^f \]

- Ensure binary points are aligned
Summary

- **Unsigned:** $x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_02^0$
- **Signed:** $x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_02^0$
- Octal and Hex short-hand
- Operations: resize, arithmetic, compare
- Arithmetic circuits trade off speed/area/power
- Fixed- and floating-point non-integers
- Gray codes for position encoding