ECE 2300
Digital Logic & Computer Organization
Spring 2022

More Timing Analysis
Binary Arithmetic
Announcements

• Prelab 3A due tomorrow

• First batch of (raw) quiz grades released on CMS

• Supplementary notes on timing analysis posted on course web
Review: Important Timing Parameters

- $t_{setup}$
- $t_{hold}$
- $t_{comb}$
- $t_{ffpd}$
- $t_{clk}$
- $t_H$
- $t_L$
To achieve a higher clock frequency (i.e., smaller cycle time), would you prefer

1) A smaller or larger **combinational delay**?
2) A smaller or larger **setup time**?
3) A smaller or larger **hold time**?
4) A negative or positive **clock skew**?
   • A smaller or larger skew?
Review: Negative Clock Skew

Sending FF receives clock later than receiving FF

\[ t_{ffpd}(\text{max}) + t_{comb}(\text{max}) + t_{setup} \leq t_{clk} - t_{skew(\text{max})} \]

Harmful skew for meeting setup time constraint
Review: Positive Clock Skew

Receiving FF receives clock later than sending FF

$$t_{ffpd(max)} + t_{comb(max)} + t_{setup} \leq t_{clk} + t_{skew(min)}$$

Beneficial skew for meeting setup time constraint
Timing Analysis Discussion (2)

• To avoid hold time violation, would you prefer
  1) A smaller or larger combinational delay? 
  2) A smaller or larger setup time? 
  3) A smaller or larger hold time? 
  4) A negative or positive clock skew? 
    • A smaller or larger skew?
Hold Time With Positive Clock Skew

Receiving FF receives clock later than sending FF

$$t_{ffpd}(\text{min}) + t_{\text{comb}(\text{min})} \geq t_{\text{hold}} + t_{\text{skew}(\text{max})}$$

Harmful skew for meeting hold time constraint
Hold Time With Negative Clock Skew

What if sending FF receives clock later than receiving FF?

\[ t_{\text{ffpd(min)}} + t_{\text{comb(min)}} \geq t_{\text{hold}} - t_{\text{skew(min)}} \]

Beneficial skew for meeting hold time constraint
Example: Hold Time Analysis with Clock Skew

Clock may arrive at FF2 up to 2ns later than FF1

<table>
<thead>
<tr>
<th></th>
<th>Prop Delay (ns)</th>
<th>Setup Time (ns)</th>
<th>Hold Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>max</td>
<td></td>
</tr>
<tr>
<td>FF</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Comb</td>
<td>3</td>
<td>7</td>
<td>-</td>
</tr>
</tbody>
</table>

• Hold time at FF2 met?
Example: Hold Time Analysis with Clock Skew

Clock may arrive at FF2 up to 2ns later than FF1

<table>
<thead>
<tr>
<th></th>
<th>Prop Delay (ns)</th>
<th>Setup Time (ns)</th>
<th>Hold Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>max</td>
<td></td>
</tr>
<tr>
<td>FF</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Comb</td>
<td>3</td>
<td>7</td>
<td>-</td>
</tr>
</tbody>
</table>

- Hold time at FF2 met?
  \[ t_{\text{ffpd(min)}} + t_{\text{comb(min)}} \geq t_{\text{hold}} + t_{\text{skew(max)}} \]
  \[ 1 + 3 \geq 2 + 2 \]

The hold time constraint is met
Course Content

- Binary numbers and logic gates
- Boolean algebra and combinational logic
- Sequential logic and state machines
- Clocking and timing analysis
- Binary arithmetic
- Memories
- Instruction set architecture
- Processor organization
- Caches and virtual memory
- Input/output
Unsigned Binary Integers

- An \( n \)-bit unsigned number represents \( 2^n \) integer values
  - From 0 to \( 2^n-1 \)

<table>
<thead>
<tr>
<th>( 2^2 )</th>
<th>( 2^1 )</th>
<th>( 2^0 )</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>
Unsigned Binary Fractions

- For the binary number $b_{n-1}b_{n-2}...b_1b_0.b_1b_2...b_m$
  the decimal number is

  $$D = \sum_{i=-m}^{n-1} b_i \cdot 2^i$$

- Examples
  
  $101.001_2 = ?$
Unsigned Binary Addition

• **Just like base-10**
  
  – Add from right to left, propagating carry

\[
\begin{array}{cccc}
10010_{(18)} & + & 01001_{(9)} & = & \text{carry} \\
10010_{(18)} & + & 01011_{(11)} & = & 01111_{(15)} \\
(27) & + & (29) & = & 10010_{(18)}
\end{array}
\]
Signed Magnitude Representation

- **Most significant bit is used as a sign bit**
  - Sign bit of 0 for positive \((001 = 1)\)
  - Sign bit of 1 for negative \((101 = -1)\)

- **Range** is from \(-\left(2^{n-1}-1\right)\) to \(\left(2^{n-1}-1\right)\) for an \(n\)-bit number

- **Two representations for zero** (+0 and -0)

- **Does ordinary binary addition still work?**

  \[
  \begin{array}{c}
  001 \hspace{1cm} (1) \\
  +101 \hspace{1cm} (-1) \\
  \hline \\
  110 \hspace{1cm} \text{(not 0)}
  \end{array}
  \]
Another Way to Encode Signed Binary Numbers

Diagram showing the representation of signed binary numbers on a circle.
Two’s Complement Representation

- MSB has weight \(-2^{n-1}\)
- Range of an n-bit number: \(-2^{n-1}\) through \(2^{n-1}-1\)
  - Most negative number \((-2^{n-1})\) has no positive counterpart

<table>
<thead>
<tr>
<th>-2^2</th>
<th>2^1</th>
<th>2^0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
</tr>
</tbody>
</table>
Two’s Complement Addition

- Procedure for addition is the same as unsigned addition regardless of the signs of the numbers

\[
\begin{array}{c}
001 \\
+ 111 \\
000
\end{array} \quad (1) \quad (-1) \quad (0)
\]
Two’s Complement Representation

• Positive numbers and zero are same as unsigned binary representation

• To get two’s complement negative notation of an integer
  – Flip every bit first
  – Then add one

\[
\begin{align*}
001 & \quad (1) \\
110 & \quad (1’s \ comp) \\
\downarrow + 1 & \quad \downarrow + 1 \\
111 & \quad (-1) \\
01001 & \quad (9) \\
10110 & \quad (1’s \ comp) \\
\downarrow + 1 & \quad \downarrow + 1 \\
10111 & \quad (-9)
\end{align*}
\]
Two’s Complement (2’s C) Shortcut

- **To get -X**
  - Copy bits from right to left up to and including the first “1”
  - Flip remaining bits to the left

\[
\begin{array}{cccc}
011010000 & 011010000 \\
100101111 & \text{(1's comp)} \\
100110000 & \text{(flip)} \\
\end{array}
\]

\[
\begin{array}{cccc}
011010000 \\
100110000 & \text{(copy)} \\
\end{array}
\]
Converting Binary (2’s C) to Decimal

1. If MSB = 1, take two’s complement to get a positive number
2. Add powers of 2 for bit positions that have a “1”
3. If original number was negative, add a minus sign

\[
X = 11100110_{\text{two}} \\
-X = 00011010 \\
= 2^4 + 2^3 + 2^1 = 16 + 8 + 2 \\
= 26_{\text{ten}} \\
X = -26_{\text{ten}}
\]

Assuming 8-bit 2’s complement numbers
Converting Decimal to Binary (2's C)

First Method: \textit{Division}

1. Change to nonnegative decimal number
2. Divide by two – remainder is least significant bit
3. Keep dividing by two until answer is zero, recording remainders from right (LSB) to left
4. \textbf{Append a zero as the MSB}; if original number $X$ was negative, return $X'$+1

\[ X = 104_{\text{ten}} \]

\[
\begin{align*}
104/2 &= 52 \text{ r0} & \text{bit 0} &= 0 \\
52/2 &= 26 \text{ r0} & \text{bit 1} &= 0 \\
26/2 &= 13 \text{ r0} & \text{bit 2} &= 0 \\
13/2 &= 6 \text{ r1} & \text{bit 3} &= 1 \\
6/2 &= 3 \text{ r0} & \text{bit 4} &= 0 \\
3/2 &= 1 \text{ r1} & \text{bit 5} &= 1 \\
1/2 &= 0 \text{ r1} & \text{bit 6} &= 1
\end{align*}
\]

\[ X = 01101000_{\text{two}} \]
Converting Decimal to Binary (2’s C)

Second Method: Subtract Powers of Two
1. Change to nonnegative decimal number
2. Subtract largest power of two less than or equal to number
3. Put a one in the corresponding bit position
4. Keep subtracting until result is zero
5. Append a zero as MSB; if original was X negative, return X’+1

<table>
<thead>
<tr>
<th>n</th>
<th>2^n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
</tr>
</tbody>
</table>

\[
X = 104_{\text{ten}}
\]

104 - 64 = 40 \hspace{1cm} \text{bit 6} = 1
40 - 32 = 8 \hspace{1cm} \text{bit 5} = 1
8 - 8 = 0 \hspace{1cm} \text{bit 3} = 1

\[
X = 01101000_{\text{two}}
\]
Sign Extension

• Replicate the MSB (sign bit)

<table>
<thead>
<tr>
<th>4-bit</th>
<th>8-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>00000100</td>
</tr>
<tr>
<td>1100</td>
<td>11111100</td>
</tr>
</tbody>
</table>

• Necessary for adding a two’s complement numbers of different lengths
Fixed Size Representation

• Microprocessors usually represent numbers as fixed size n-bit values

• Result of adding two n-bit integers is stored as n bits

• Integers are typically 32 or 64 bits (words)
  – 4 or 8 bytes
  – byte = 8 bits
Fixed Size Addition

• Examples with n = 4

\[
\begin{array}{ccc}
2 & 0010 & 2 & 0010 & -2 & 1110 \\
+ & 3 & 0011 & + & -3 & 1101 & + & 6 & 0110 \\
\hline
5 & 0101 & -1 & 1111 & 4 & 0100 \\
\end{array}
\]

\[
\begin{array}{ccc}
-2 & 1110 & 7 & 0111 & -7 & 1001 \\
+ & -6 & 1010 & + & 6 & 0110 & + & -4 & 1100 \\
\hline
-8 & 1000 & -3 & 1101 & 5 & 0101 \\
\end{array}
\]

Something went wrong!
Overflow

• If operands are too big, sum cannot be represented as $n$-bit 2’s complement number

\[
\begin{align*}
01000 & \quad (8) \\
+ 01001 & \quad (9) \\
10001 & \quad (-15)
\end{align*}
\]

\[
\begin{align*}
11000 & \quad (-8) \\
+ 10111 & \quad (-9) \\
01111 & \quad (+15)
\end{align*}
\]

• Overflow occurs if
  – Signs of both operands are the same, and
  – Sign of sum is different

• Another test (easy to do in hardware)
  – Carry into MSB does not equal carry out
Before Next Class

• H&H 5.5

Next Time

More Binary Arithmetic
ALU