CSE140: Components and Design Techniques for Digital Systems

Adders, multipliers and other ALU elements

Prof. Tajana Simunic Rosing
CSE140a Midterm2 Stats

Original grades:
- Max score : 99
- Min score: 13.5
- Mean score : 66

Curved grades:
- Max score : 100
- Min score (w/o 0s) : 26
- Mean score : 75

Sources: TSR, Katz, Boriello, Vahid, D & S Harris
Where we are now

• What we’ve covered so far:
  – Chap 1, 2, 3
  – We will skip chap 4 – it is on Verilog which you are learning in 140L

• Logistics:
  – HW #7 assigned
  – Added HW #8 due Monday 6/10/13 at 8pm via email or in person to Sheila/TAs/Prof; solutions posted right after 8pm
  – The lowest grade of the HWs will be dropped

• CAPEs: http://www.cape.ucsd.edu/
  – If participation rate is > 85%, 2nd lowest grade of HWs will be dropped

• Where we are going next:
  – Chap 5: components that make a CPU: adders, multipliers, ALU, counters, registers, memory
  – RTL: Chap 5 from Vahid’s textbook is uploaded on the schedule page
1-Bit Adders

**Half Adder**

\[ S = A \oplus B \]
\[ C_{out} = AB \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C_{out}</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Full Adder**

\[ S = A \oplus B \oplus C_{in} \]
\[ C_{out} = AB + AC_{in} + BC_{in} \]

<table>
<thead>
<tr>
<th>C_{in}</th>
<th>A</th>
<th>B</th>
<th>C_{out}</th>
<th>S</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>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Multibit Adders (CPAs)

• Types of carry propagate adders (CPAs):  
  – Ripple-carry  (slow)  
  – Carry-lookahead  (faster)  

• Carry-lookahead requires more hardware

Symbol
Ripple-Carry Adder

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: slow

- Ripple-carry adder delay
  \[ t_{\text{ripple}} = N t_{FA} \]
  
  where \( t_{FA} \) is the delay of a full adder
Carry-lookahead adders

- Adder with propagate (P) and generate (G) outputs:
- Evaluate Sum and Ci+1
  - Sum = Ai \text{xor} Bi \text{xor} Ci
  - Ci+1 = Ai Bi + Ai Ci + Bi Ci
    = Ai Bi + Ci (Ai \text{xor} Bi)
    = Gi + Ci Pi
Carry-Lookahead Adder

- **Example:** 4-bit blocks \((G_{3:0} \text{ and } P_{3:0})\):
  \[
  G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))
  
  P_{3:0} = P_3 P_2 P_1 P_0
  \]

- **Generally:**
  - **Step 1:** Compute \(G_i\) and \(P_i\) for all columns
  - **Step 2:** Compute \(G\) and \(P\) for \(k\)-bit blocks
  - **Step 3:** \(C_{in}\) propagates through each \(k\)-bit propagate/generate block

  \[
  G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} G_j))
  
  P_{i:j} = P_i P_{i-1} P_{i-2} P_j
  
  C_{i} = G_{i:j} + P_{i:j} C_{i-1}
  \]
For $N$-bit CLA with $k$-bit blocks:

$$t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1)t_{AND\_OR} + kt_{FA}$$

- $t_{pg}$: delay to generate all $P_i, G_i$
- $t_{pg\_block}$: delay to generate all $P_{i:j}, G_{i:j}$
- $t_{AND\_OR}$: delay from $C_{in}$ to $C_{out}$ of final AND/OR gate in $k$-bit CLA block

An $N$-bit carry-lookahead adder is generally much faster than a ripple-carry adder for $N > 16$
Subtractor

Symbol

\[ \begin{array}{cc}
A & B \\
\text{N} & \text{N} \\
\text{N} & \text{N} \\
\end{array} \]

\[- \]

\[ Y \]

Implementation

\[ \begin{array}{cc}
A & B \\
\text{N} & \text{N} \\
\text{N} & \text{N} \\
\end{array} \]

\[ \text{Logic 1} \]

\[ + \]

\[ Y \]
Adder/subtractor
Comparator: Equality

Symbol

Implementation

A\_3
B\_3
A\_2
B\_2
A\_1
B\_1
A\_0
B\_0
Equal

Equal
Comparator: Less Than

\[ A < B \]
Arithmetic Logic Unit (ALU)

<table>
<thead>
<tr>
<th>F₂:₀</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>A &amp; B</td>
</tr>
<tr>
<td>001</td>
<td>A</td>
</tr>
<tr>
<td>010</td>
<td>A + B</td>
</tr>
<tr>
<td>011</td>
<td>not used</td>
</tr>
<tr>
<td>100</td>
<td>A &amp; ~B</td>
</tr>
<tr>
<td>101</td>
<td>A</td>
</tr>
<tr>
<td>110</td>
<td>A - B</td>
</tr>
<tr>
<td>111</td>
<td>SLT</td>
</tr>
</tbody>
</table>

Sources: TSR, Katz, Boriello, Vahid, D & S Harris
Set Less Than (SLT) Example

- **SLT**: If $A < B$ then $Y = 1$, else $Y = 0$
- **Configure 32-bit ALU for SLT operation**: $A = 25$ and $B = 32$
  - $A < B$, so $Y$ should be 32-bit representation of 1 (0x00000001)
  - $F_{2:0} = 111$
    - $F_2 = 1$ (subtracting), so $25 - 32 = -7$
    - $-7$ has 1 in the MSB ($S_{31} = 1$)
    - $F_{1:0} = 11$ multiplexer selects $Y = S_{31}$ (zero extended) = 0x00000001.
**Shifters**

- **Logical shifter:** shifts value to left or right and fills empty spaces with 0’s. Can multiply/divide unsigned #s by 2.
  - Ex: $11001 >> 2 = 00110$
  - Ex: $11001 << 2 = 00100$

- **Arithmetic shifter:** same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb). Shifting right by $n$ bits on a two's complement signed binary number has the effect of dividing it by $2^n$, rounding down.
  - Ex: $11001 >>> 2 = 11110$
  - Ex: $11001 <<< 2 = 00100$

- **Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
  - Ex: $11001$ ROR 2 = 01110
  - Ex: $11001$ ROL 2 = 00111
4 x 4 Multiplier Array

\[
\begin{array}{cccc}
A_3 & A_2 & A_1 & A_0 \\
\times & B_3 & B_2 & B_1 & B_0 \\
\hline
A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 \\
A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 \\
A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0 \\
\end{array}
\]

Sources: TSR, Katz, Boriello, Vahid, D & S Harris
CSE140: Components and Design Techniques for Digital Systems

Registers, Counters and Memory

Tajana Simunic Rosing
Basic Register

Sources: TSR, Katz, Boriello, Vahid, D & S Harris
### Shift Register

- Shift a new bit in on each clock edge
- Shift a bit out on each clock edge
- *Serial-to-parallel converter*: converts serial input ($S_{in}$) to parallel output ($Q_{0:N-1}$)

**Symbol:**

**Implementation:**

Sources: TSR, Katz, Boriello, Vahid, D & S Harris
Shift Register with Parallel Load

- When $Load = 1$, acts as a normal $N$-bit register
- When $Load = 0$, acts as a shift register
- Now can act as a *serial-to-parallel converter* ($S_{in}$ to $Q_{0:N-1}$) or a *parallel-to-serial converter* ($D_{0:N-1}$ to $S_{out}$)
Design of a universal shift register

<table>
<thead>
<tr>
<th>clear</th>
<th>s0</th>
<th>s1</th>
<th>new value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>output</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>output value of FF to left (shift right)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>output value of FF to right (shift left)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>input</td>
</tr>
</tbody>
</table>

Nth cell

Sources: TSR, Katz, Boriello, Vahid, D & S Harris
Pattern recognizer using a shift register
Binary Counters

- Increments on each clock edge
- Used to cycle through numbers. For example, 
  - 000, 001, 010, 011, 100, 101, 110, 111, 000, 001…
- Example uses:
  - Digital clock displays
  - Program counter: keeps track of current instruction executing

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Symbol" /></td>
<td><img src="image2.png" alt="Implementation" /></td>
</tr>
</tbody>
</table>
General binary counters

• Default operation: counts up by 1
  • QA-QD counter output
  • A-D parallel load data
  • LOAD enables parallel data load
  • RCO ripple carry out – set to 1 if count reaches max value
  • CLR clears data
  • EN enables the counter
Sequence Generator Design

• Sequences through a fixed set of patterns
**Memory: basic concepts**

- Stores large number of bits
  - $m \times n$: $m$ words of $n$ bits each
  - $k = \log_2(m)$ address input signals
  - or $m = 2^k$ words
  - e.g., 4,096 x 8 memory:
    - 32,768 bits
    - 12 address input signals
    - 8 input/output data signals

- Memory access
  - r/w: selects read or write
  - enable: read or write only when asserted
  - multiport: multiple accesses to different locations simultaneously

---

Sources: TSR, Katz, Boriello, Vahid, D & S Harris
Comparing RAM

- **Register file**
  - Fastest
  - But biggest size

- **SRAM**
  - Fast
  - More compact than register file

- **DRAM**
  - Slowest
    - And refreshing takes time
  - But very compact
  - Different technology for large caps.
• Similar internal structure as register file
  – Decoder enables appropriate word based on address inputs
  – \( rw \) controls whether cell is written or read
**Static RAM (SRAM) - writing**

- "Static" RAM cell
  - Writing this cell
    - *word enable* input comes from decoder
    - When 0, value \(d\) loops around inverters
      - That loop is where a bit stays stored
    - When 1, the *data* bit value enters the loop
      - *data* is the bit to be stored in this cell
      - *data'* enters on other side
      - Example shows a "1" being written into cell

**Sources:** TSR, Katz, Boriello, Vahid, D & S Harris
• “Static” RAM cell - reading
  – When rw set to read, the RAM logic sets both data and data’ to 1
  – The stored bit d will pull either the left line or the right bit down slightly below 1
  – “Sense amplifiers” detect which side is slightly pulled down
Dynamic RAM (DRAM)

- “Dynamic” RAM cell
  - 1 transistor (rather than 6)
  - Relies on large capacitor to store bit
    - Write: Transistor conducts, data voltage level gets stored on top plate of capacitor
    - Read: Look at the value of $d$
    - Problem: Capacitor discharges over time
      - Must “refresh” regularly, by reading $d$ and then writing it right back
Read-Only Memory – ROM

- Memory that can only be read from
  - Data lines are output only
- Advantages over RAM
  - Nonvolatile
  - Low power
  - Compact

\[ \text{Let } A = \log_2 M \]

Sources: TSR, Katz, Boriello, Vahid, D & S Harris
ROM Types

- **Erasable Programmable ROM (EPROM)**
  - Uses “floating-gate transistor” in each cell
  - Programmer uses higher-than-normal voltage so electrons *tunnel* into the gate
    - Electrons become trapped in the gate
    - Only done for cells that should store 0
    - Other cells will be 1
  - To erase, shine ultraviolet light onto chip
    - Gives trapped electrons energy to escape
    - Requires chip package to have window

- **Electronically-Erasable Programmable ROM EEPROM**
  - Programming similar to EPROM
  - Erasing one word at a time *electronically*

- **Flash memory**
  - Like EEPROM, but large blocks can be erased *simultaneously*

- **EEPROM & FLASH are in-system programmable**