CSE140: Components and Design Techniques for Digital Systems

Adders, subtractors comparators, multipliers and other ALU elements

Instructor: Mohsen Imani
UC San Diego

Slides from:
Prof.Tajana Simunic Rosing & Dr.Pietro Mercati

Sources: TSR, Katz, Boriello & Vahid
Adders
Circuit Delay

- Transistors have intrinsic resistance and capacitance
- Signals take time to propagate from the input to the output of a gate
- Sometimes delays are labeled as @<delay_value> in circuit drawings
1-Bit & Multi-bit Adders

**Half Adder**

\[
\begin{align*}
S &= A \oplus B \\
C_{out} &= AB
\end{align*}
\]

**Full Adder**

\[
\begin{align*}
S &= A \oplus B \oplus C_{in} \\
C_{out} &= AB + AC_{in} + BC_{in}
\end{align*}
\]

**Types of multi-bit adders**
- Ripple-carry (slow)
- Carry-lookahead (faster)
- Two-level logic adder (even faster)

**Symbol**
Ripple-Carry Adder

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

Ripple-carry adder delay

\[ t_{\text{ripple}} = N t_{\text{FA}} \]

where \( t_{\text{FA}} \) is the delay of a full adder

Sources: TSR, Katz, Boriello & Vahid
Two-level Logic Adder

- No matter how many inputs you have,
  - look at the truth table,
  - convert to Kmap,
  - apply the algorithm for two-level logic minimization
- Very fast adder, but.....
- Beyond 8 inputs, a shockingly large amount of gates!
  - Number of gates increases exponentially

Ripple carry adder  Carry-lookahead adder (next slide)  Two-level logic adder

Sources: TSR, Katz, Boriello & Vahid
# Carry-lookahead adders

From the very beginning I can “look ahead” into the value of carries.

<table>
<thead>
<tr>
<th>c4</th>
<th>c3</th>
<th>c2</th>
<th>c1</th>
<th>c0</th>
<th>Carries</th>
</tr>
</thead>
<tbody>
<tr>
<td>a3</td>
<td>a2</td>
<td>a1</td>
<td>a0</td>
<td></td>
<td>First operand</td>
</tr>
<tr>
<td>b3</td>
<td>b2</td>
<td>b1</td>
<td>b0</td>
<td></td>
<td>Second operand</td>
</tr>
<tr>
<td>s3</td>
<td>s2</td>
<td>s1</td>
<td>s0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
| 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
```

Sources: TSR, Katz, Boriello & Vahid
### Full Adder

**Ci+1 = Ai Bi + Ci (Ai \text{xor} Bi)**

- **Generate**
  - $Ci+1 = Gi + Ci Pi$

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$C_i$</th>
<th>$S$</th>
<th>$C_o$</th>
<th>Carry status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>delete</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>delete</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>propagate</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>propagate</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>propagate</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>propagate</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>generate</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>generate</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$C_{in}$</th>
<th>$G$</th>
<th>$P$</th>
<th>$\text{Sum}$</th>
<th>$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<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>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<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>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Sources: TSR, Katz, Boriello & Vahid
Carry-lookahead adders

- Adder with **propagate** (P) and **generate** (G) outputs:

\[
C_{i+1} = A_i \cdot B_i + C_i \cdot (A_i \oplus B_i)
\]

\[
C_{i+1} = G_i + C_i \cdot P_i
\]

The carry at some level is equal to 1 if either the generate signal is equal to one or if the propagate and the previous carry are both 1.
Carry-Lookahead Adder

\[ P_G = P_3 \cdot P_2 \cdot P_1 \cdot P_0 \]
\[ G_C = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 \]
\[ C_4 = G_0 + P_0 \cdot C_0 \]

Ref: Dan Earnst

Sources: TSR, Katz, Boriello & Vahid
Carry-Lookahead Adder

- 4 bit adders with internal carry look-ahead P and G logic
- Second level carry-lookahead unit creates the **GROUP** P and G signals
Carry-lookahead adders

Example: 4-bit CLA adder

\[ c_1 = G_0 + P_0 c_0 \]
\[ c_2 = G_1 + P_1 c_1 \]
\[ c_3 = G_2 + P_2 c_2 \]
\[ c_4 = G_3 + P_3 c_3 \]

\[ c_2 = G_1 + P_1 (G_0 + P_0 c_0) = G_1 + P_1 G_0 + P_1 P_0 c_0 \]

All “G” and “P” are immediately available, but “c” are not (except the c0).

So you need to make substitutions:

\[ c_1 = G_0 + P_0 c_0 \]
\[ c_2 = G_1 + P_1 (G_0 + P_0 c_0) \]
\[ c_3 = G_2 + P_2 c_2 \]
\[ c_4 = G_3 + P_3 c_3 \]

\[ c_2 = (\text{derive at home}) \]
\[ c_3 = (\text{derive at home}) \]
Carry-lookahead adders

Propagate/Generate circuit (one per each input bit)

Carry circuits (implement the equations derived in the previous slide)

Note: this approach of “looking ahead” for building multi-bit operations is not limited to adders!

Sources: TSR, Katz, Boriello & Vahid
Carry-select adder

- Redundant hardware to make carry calculation go faster
  - compute two high-order sums in parallel while waiting for carry-in
  - one assuming carry-in is 0 and another assuming carry-in is 1
  - select correct result once carry-in is finally computed
Carry-select adder

- A 16-bit carry-select adder with a uniform block size of 4
- Using three blocks and a 4-bit ripple carry adder
- Carry-in is known at the beginning of computation, a carry select block is not needed for the first four bits
- The delay of this adder will be four full adder delays, plus three MUX delays.

Sources: TSR, Katz, Boriello & Vahid
Carry Save Adder (CSA)

Making N additions independent and in parallel with no carry propagation
Propagate carry at the last stage
Approximate Adder

- Several applications accept a part of inaccuracy in their computation (e.g. multimedia)
- Inaccuracy is inherent in many applications due to its stochastic behavior (e.g. machine learning)
- Can we relax the computation in order to improve efficiency?

```
\[ \begin{array}{c}
A_{31} & B_{31} & A_{30} & B_{30} & A_{1} & B_{1} & A_{0} & B_{0} \\
S_{31} & + & S_{30} & + & S_{1} & + & S_{0} & + \\
C_{out} & C_{30} & C_{29} & C_{1} & C_{0} & C_{in} \\
\end{array} \]
```
Quality Comparison

Sobel Application (Image Processing)

Robert Application (Image Processing)

Sources: TSR, Katz, Boriello & Vahid
Approximate Adder

- How about dropping the least N significant bits?
Approximate Adder

- Can we do better job than dropping values?
- Carry is more costly or sum?
- Except two cases:
  \[ C_{\text{out}} \neq \text{Sum} \]

<table>
<thead>
<tr>
<th>( C_{\text{in}} )</th>
<th>( A )</th>
<th>( B )</th>
<th>( C_{\text{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>

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

Sources: TSR, Katz, Boriello & Vahid
Subtractors
If N is a positive number, then the negative of N (its 2s complement or $N^*$) is bit-wise complement plus 1.

The most significant bit represents the sign: 0 for positive and 1 for negative.

N bit can represent $[0 \ 2^N - 1]$ integer positive numbers.

In 2s complement, you can represent the interval $[-(2^N - 1) \ (2^N - 1)]$. 

$A-B = A + 1(-B)$
2s Complement: Examples

A 8-bit example
0 0 0 0 1 0 1 1

(positive) = 11

1 1 1 1 0 1 0 0 (complement)
1 1 1 1 0 1 0 1 (add 1)

A 5-bit example
1 0 0 1 0

(negative) = -14

0 1 1 0 1 (complement)
0 1 1 1 0 (add 1)
Subtraction

If you are using 4 bit numbers, what is the result of the following equation in 2's complement: \( y = 4 - 7 \)

A. 1011 
B. 0011 
C. 1101 
D. 1100 
E. None of the above
Detecting Overflow: Method 1

- Assuming 4-bit two’s complement numbers, one can detect overflow by detecting when the two numbers’ sign bits are the same but are different from the result’s sign bit.
  - If the two numbers’ sign bits are different, overflow is impossible.
    - Adding a positive and negative can’t exceed the largest magnitude positive or negative.
- Simple circuit
  - overflow = a3'b3's3 + a3b3s3'

<table>
<thead>
<tr>
<th>Sign Bits</th>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1</td>
<td>+ 0 0 0 1</td>
<td>1 0 0 0 (overflow)</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>+ 1 0 0 0</td>
<td>0 1 1 1 (overflow)</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>+ 0 1 1 1</td>
<td>1 1 1 1 (no overflow)</td>
</tr>
</tbody>
</table>

If the numbers’ sign bits have the same value, which differs from the result’s sign bit, overflow has occurred.
Detecting Overflow: Method 2

- Detect a difference between carry-in to sign bit and carry-out from it
- Yields a simpler circuit: $\text{overflow} = c_3 \oplus c_4 = c_3 c_4' + c_3' c_4$

If the carry into the sign bit column differs from the carry out of that column, overflow has occurred.
Subtractor

A subtraction between A and B is the same as the sum between the first value and the negative of the second value:

\[(A - B) = A + (-B)\]

Represent numbers in 2s complement and use a normal adder!
Adder/subtractor

In this schematic addition occurs when Sel signal is:
A. True
B. False

\[ A + (\neg B) = (3 + 1) \]

Sources: TSR, Katz, Boriello & Vahid
More ALU Components
Comparator: Equality

Two numbers are equal if each digit at each position is equal (this is true for any base: decimal, binary, etc).
The bit-to-bit equality can be evaluated with the XNOR gate.
Comparator: Less Than

- If a number A is less than B and you consider the difference A − B, this is:
  - negative.
- So comparing numbers is equivalent to check the sign of the difference. In 2s complement representation, the sign of the result corresponds to:
  - the most significant bit

\[
A < B
\]
Shifters

• **Logical shifter:** shifts value to left or right and fills empty spaces with 0’s
  
  – 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
  
  – 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

Sources: TSR, Katz, Boriello & Vahid

General Shifter Design

Based on the value of the selection input (shamt = shift amount)

The “chain” of multiplexers determines how many bits to shift

Example: if S = 01 then
Y3 = 0
Y2 = A3
Y1 = A2
Y0 = A1
Multiplication of positive binary numbers

- Generalized representation of multiplication by hand
  
  Example: in decimal,
  
  \[ 32 \times 4 = (30+2) \times 4 = 30 \times 4 + 2 \times 4 \]

  Basically: sum up the partial products (pp)

  The binary multiplier is based on the same idea:

  \[
  \begin{array}{cccccc}
  a_3 & a_2 & a_1 & a_0 \\
  x & b_3 & b_2 & b_1 & b_0 \\
  \end{array}
  \]

  \[
  \begin{array}{cccccccc}
  \text{(pp1)} & b_0a_3 & b_0a_2 & b_0a_1 & b_0a_0 \\
  \text{(pp2)} & b_1a_3 & b_1a_2 & b_1a_1 & b_1a_0 & 0 \\
  \text{(pp3)} & b_2a_3 & b_2a_2 & b_2a_1 & b_2a_0 & 0 & 0 \\
  \text{(pp4)} & b_3a_3 & b_3a_2 & b_3a_1 & b_3a_0 & 0 & 0 & 0 \\
  \end{array}
  \]

  For demo see:
  
  http://courses.cs.vt.edu/~cs1104/BuildingBlocks/multiply.010.html
Multiplier – Array Style

- Multiplier design – array of AND gates

If the multiplier has two N-bit inputs, how many bits are required for the output?
Approximate Multiplier

• How about Approximate Multiplier?
• What will happen if one of the input operands are power of two?
  – Do we need to multiply?
• Detect such cases and replace multiplication with shift operation
• E.g. 100010 * 000010  ➞ shift the first operand to right (1-bit)
• E.g. A * 000100  ➞ shift the first operand to right (2-bit)
• How about A* 01000000001?
• Can I ignore the last “1” bit and just shift the number by 9-bits?

Floating Point Multiplication

- Why we need floating point unit (FP) representation?
  - More precision, covering wide range of values!
- FP representation: \( \text{sign} \ 1.\text{fraction} \times 2^{\text{exponent}} \)
  - E.g. 1001 010 = 1.(010)'b \times 2^{9} = 1.25 \times 2^{9}

<table>
<thead>
<tr>
<th>Variable</th>
<th>sign</th>
<th>exponent</th>
<th>fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>1001</td>
<td>010</td>
</tr>
<tr>
<td>Y</td>
<td>0</td>
<td>0111</td>
<td>110</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{Operation} & \quad \text{Energy [pJ]} & \quad \text{Relative Cost} \\
32 \text{ bit int ADD} & \quad 0.1 & \quad 1 \\
32 \text{ bit float ADD} & \quad 0.9 & \quad 9 \\
32 \text{ bit Register File} & \quad 1 & \quad 10 \\
32 \text{ bit int MULT} & \quad 3.1 & \quad 31 \\
32 \text{ bit float MULT} & \quad 3.7 & \quad 37 \\
\end{align*}
\]
Division of positive binary numbers

• Repeated subtraction
  – Set quotient to 0
  – Repeat while dividend >= divisor
    • Subtract divisor from dividend
    • Add 1 to quotient
  – When dividend < divisor:
    • Reminder = dividend
    • Quotient is correct

Example:
• Dividend: 101; Divisor: 10

<table>
<thead>
<tr>
<th>Dividend</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>0 +</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11 -</td>
<td>1 +</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

For demo see:
http://courses.cs.vt.edu/~cs1104/BuildingBlocks/Binary.Divide.html

Sources: TSR, Katz, Boriello & Vahid
ALU: Arithmetic Logic Unit
Arithmetic Logic Unit – Example

Implement the ALU using as few components as possible

<table>
<thead>
<tr>
<th>$F_{2:0}$</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>$A &amp; B$</td>
</tr>
<tr>
<td>001</td>
<td>$A \mid B$</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>$A &amp; \sim B$</td>
</tr>
<tr>
<td>101</td>
<td>$A \mid \sim B$</td>
</tr>
<tr>
<td>110</td>
<td>$A - B$</td>
</tr>
<tr>
<td>111</td>
<td>Not used</td>
</tr>
</tbody>
</table>

Sources: TSR, Katz, Boriello & Vahid


Summary of what we have seen so far

- Transistors
- Boolean algebra
- Basic gates
- Logic functions and truth tables
- Canonical forms (SOP and POS)
- Two-level logic minimization
- Kmaps
- Multiplexers (behavior and how to implement logic functions with them)
- Decoders (behavior and how to implement logic functions with them)

Today:
- Adders, subtractors, and other ALU components
- SO FAR: only COMBINATIONAL logic (i.e. no “memory” elements)
CSE140: Components and Design Techniques for Digital Systems

Sequential Circuit Introduction
Latches and Flip-Flops

Tajana Simunic Rosing
What is a sequential circuit?

A circuit whose output depends on current inputs and past outputs

A circuit with **memory**

\[
y_i = f_i(S^t, X) \\
s_{i+1} = g_i(S^t, X)
\]

Sources: TSR, Katz, Boriello & Vahid
Why do we need circuits with ‘memory’?

- Circuits with memory can be used to store data
- Systems have circuits that run a sequence of tasks

Memory Hierarchy

- Registers
- Cache
- Main Memory
- Hard disk

Sources: TSR, Katz, Boriello & Vahid
Flight attendant call button

- Flight attendant call button
  - Press call: light turns on
    - Stays on after button released
  - Press cancel: light turns off
  - Logic circuit to implement this?

- SR latch implementation
  - Call=1: sets Q to 1 and keeps it at 1
  - Cancel=1: resets Q to 0

Sources: TSR, Katz, Boriello & Vahid
SR Latch Analysis

- $S = 1, R = 0$:
  
  then $Q = 1$ and $\overline{Q} = 0$

- $S = 0, R = 1$:
  
  then $Q = 1$ and $\overline{Q} = 0$

Sources: TSR, Katz, Boriello & Vahid
SR Latch Analysis

– $S = 0, \ R = 0$:  
  then $Q = Q_{prev}$

– **Memory!**

– $S = 1, \ R = 1$:  
  then $Q = 0, \ \bar{Q} = 0$

– **Invalid State**
  $\bar{Q} \neq \text{NOT} \ Q$
What if a kid presses both call and cancel & then releases them?

- If $S=1$ and $R=1$ at the same time and then released, $Q=\ ?$
  - Can also occur also due to different delays of different paths
  - $Q$ may oscillate and eventually settle to 1 or 0 due to diff. path delay

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>hold</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>not allowed</td>
</tr>
</tbody>
</table>

Sources: TSR, Katz, Boriello & Vahid
SR Latch Symbol

- SR stands for Set/Reset Latch
  - Stores one bit of state ($Q$)
- Control what value is being stored with $S$, $R$ inputs
  - **Set**: Make the output 1
    
    $(S = 1, R = 0, Q = 1)$
  - **Reset**: Make the output 0
    
    $(S = 0, R = 1, Q = 0)$
  - **Hold**: Keep data stored
    
    $(S = 0, R = 0, Q = Q_{\text{previous}})$
SR Latch Characteristic Equation

To analyze, break the feedback path

\[
\begin{align*}
Q(t+\Delta) &= S + R' Q(t) \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q(t)</th>
<th>Q(t+\Delta)</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>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

characteristic equation

\[
Q(t+\Delta) = S + R' Q(t)
\]

State Diagram

SR Latch Symbol

Sources: TSR, Katz, Boriello & Vahid
Avoiding S=R=1 Part 1: Level-Sensitive SR Latch

• Add input “C”
  – Change C to 1 only after S and R are stable
  – C is usually a clock (CLK)
**Clocks**

- **Clock** -- Pulsing signal for enabling latches; ticks like a clock
- **Synchronous** circuit: sequential circuit with a clock

- **Clock period**: time between pulse starts
  - Above signal: period = 20 ns
- **Clock cycle**: one such time interval
  - Above signal shows 3.5 clock cycles
- **Clock duty cycle**: time clock is high
  - 50% in this case
- **Clock frequency**: 1/period
  - Above : freq = 1 / 20ns = 50MHz;
Clock question

The clock shown in the waveform below has:

A. Clock period of 4ns with 250MHz frequency
B. Clock duty cycle 75%
C. Clock period of 1ns with 1GHz frequency
D. A. & B.
E. None of the above
Avoiding $S=R=1$ Part 2: Level-Sensitive D Latch

- SR latch requires careful design so $SR=11$ never occurs
- D latch helps by inserting the inverter between S & R inputs
  - Inserted inverter ensures R is always the opposite of S when $C=1$
D Latch Truth Table

<table>
<thead>
<tr>
<th>CLK</th>
<th>D</th>
<th>D</th>
<th>S</th>
<th>R</th>
<th>( Q )</th>
<th>( \overline{Q} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0_{prev}</td>
<td>1_{prev}</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>