The university is interested in installing a **Solar Power Generating Facility** on the roof of Small Hall.

Project not official at university level yet, but SPS + Dept. will kickstart project:

- Meeting **Tuesday next week** → watch for SPS posters.
- 1st design goal: **Determine which solar technology to use.**
  → How much electrical power can we expect to get.
- Dept. of Physics will provide **modest research funds** to SPS for project.
- Final budget for project is not finalized.
  → Paid for from **Green Fees** (i.e. your money).
- Installation will occur during or immediately after **renovation of Small Hall**.
Outline:

1. **Timing noise**
   - Signal races, glitches
   - FPGA example ("assign" → bad)

2. **Synchronous circuits and memory**
   - Logic gate example

3. **Flip-Flop memory**
   - RS-latch example

4. **D and JK flip-flops**
   - Flip-flops in FPGAs

5. **Synchronous circuit design with FPGAs**
   - FPGA example ("always" → good).
   - Parallel circuit design with FPGAs.
Amplitude Noise

A digital circuit is very immune to amplitude noise, since it can only have two values (Low or High, True or False, 0 or 1). Digital electronics circuits typically have error rates smaller than 1 part in $10^9$ (no error correction).

Timing Noise

Just like an analog circuit, a digital circuit can experience timing noise. Fortunately, good clocks are cheap and easily available, and a good design will eliminate the effects of timing noise.

Timing issues/errors can easily produce amplitude noise (bit errors).
The timing delays produced by wires and logic gates can produce unwanted (illogical) outputs.

Example: 3-input NAND gate
Signal Race

The timing delays produced by wires and logic gates can produce unwanted (illogical) outputs.

Example: 3-input NAND gate

If gate delays are too long output pulse could disappear.
Signal Race

The timing delays produced by wires and logic gates can produce unwanted (illogical) outputs.

Example: 3-input NAND gate

![Diagram of a 3-input NAND gate with signals A, B, C, AB, and Y, showing pulse delay and shorter output pulse.]

Pulse is shorter than expected and delayed.
Signal Race with Glitch

XOR

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>L</td>
</tr>
</tbody>
</table>

[diagram courtesy of Altera Inc.]

Inverter delay
Inverter delay + component differences

resulting $\overline{A}B$
resulting $\overline{AB}$
resulting $\overline{B}$
resulting $Y$

[Figure adapted from *Principles of Electronics: Analog & Digital* by L. R. Fortney]
Signal Race with Glitch

[diagram courtesy of Altera Inc.]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>L</td>
</tr>
</tbody>
</table>

[Figure adapted from Principles of Electronics: Analog & Digital by L. R. Fortney]
Glitches with FPGAs

Quartus II will simulate glitches

```verilog
module adder_assign(input1, input2, result);

input [3:0] input1;
input [3:0] input2;
output [4:0] result;

assign result = input1 + input2;
endmodule
```
Asynchronous design requires very careful attention to signal delays to avoid producing glitches and other spurious signals.

Glitches will produce false data and can produce very wrong results. e.g. a glitch on the most-significant-bit will produce a factor of 2 error.

Asynchronous design can produce very fast digital circuits, but is generally avoided due to more difficult design.
The use of **memory** and a **clock** can eliminate signal races and glitches.

**Basic flip-flop operation**

The flip-flop will record and output the value at the input if the **clock** is HIGH. If the **clock** goes LOW, then the flip-flop does not change its value or output.

Glitches are eliminated if

1. The clock HIGH and LOW times are longer than any gate delays.
2. The inputs are synchronized to the clock.
Synchronous Timing

Flip-flop AB

Flip-flop C

resulting Y

Guaranteed minimum signal pulse
D-type Edge-Triggered Flip-Flop

Generally, the flip-flop changes state on a clock signal “edge”, not the level. The flip-flop takes the value *just before* the clock “edge”.

![Flip-Flop Diagram]

Input: D, S or PRE
Output: Q, R or CLR

For 74LS74: minimum $t_s = 20 \text{ ns}$
minimum $t_h = 5 \text{ ns}$

Note: A flip-flop saves information (i.e. 1 bit); it does not modify it.

[Texas Instruments 74LS74 flip-flop datasheet]
Generally, the flip-flop changes state on a clock signal “edge”, not the level. The flip-flop takes the value *just before* the clock “edge”.

For 74LS74: minimum $t_s = 20 \text{ ns}$
minimum $t_h = 5 \text{ ns}$

Note: A flip-flop saves information (i.e. 1 bit); it does not modify it.
Synchronous Timing (revisited)

A

B

C

Flip-flop AB

resulting Y

Flip-flop C

Time

clock

A

B

C

Flip-flop AB

resulting Y

Flip-flop C

Time
How does a flip-flop work?

Basic flip-flop: the SR latch

Logic table

<table>
<thead>
<tr>
<th>$S$</th>
<th>$\bar{R}$</th>
<th>$Q$</th>
<th>$\bar{Q}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$Q_0$</td>
<td>$\bar{Q}_0$</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>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$Q_0 =$ value before S&R changes

$\bar{R} = 0$ & $\bar{S} = 0$:

- $\bar{S} = 0$ & assume $\bar{Q} = 0 \rightarrow Q = 1$.
- $\bar{S} = 0$ & assume $\bar{Q} = 1 \rightarrow Q = 1$.
- $\bar{R} = 0$ & assume $Q = 0 \rightarrow \bar{Q} = 1$.
- $\bar{R} = 0$ & assume $Q = 1 \rightarrow \bar{Q} = 1$. 

How does a flip-flop work?

Basic flip-flop: the SR latch

![SR latch diagram](image)

**Logic table**

<table>
<thead>
<tr>
<th>$\bar{S}$</th>
<th>$\bar{R}$</th>
<th>$Q$</th>
<th>$\bar{Q}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$Q_0$</td>
<td>$\bar{Q}_0$</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>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$Q_0 =$ value before S&R changes

- $\bar{R} = 0$ & $\bar{S} = 0$:
  - $\bar{S} = 0$ & assume $\bar{Q} = 0 \rightarrow Q = 1$.
  - $\bar{S} = 0$ & assume $\bar{Q} = 1 \rightarrow Q = 1$.
  - $\bar{R} = 0$ & assume $Q = 0 \rightarrow \bar{Q} = 1$.
  - $\bar{R} = 0$ & assume $Q = 1 \rightarrow \bar{Q} = 1$.

consistent → $\bar{R} = 0$ & $\bar{S} = 0 \rightarrow Q = 1$ & $\bar{Q} = 1$
How does a flip-flop work?

Basic flip-flop: the SR latch

Logic table

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>(\overline{Q})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>(Q_0)</td>
<td>(\overline{Q}_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>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\(Q_0 = \) value before S&R changes

\(\overline{R} = 0 \& \overline{S} = 1: \)

- \(\overline{S} = 1 \& \) assume \(\overline{Q} = 0 \rightarrow Q = 1.\)
- \(\overline{S} = 1 \& \) assume \(\overline{Q} = 1 \rightarrow Q = 0.\)

- \(\overline{R} = 0 \& \) assume \(Q = 0 \rightarrow \overline{Q} = 1.\)
- \(\overline{R} = 0 \& \) assume \(Q = 1 \rightarrow \overline{Q} = 1.\)
How does a flip-flop work?

Basic flip-flop: the SR latch

![SR latch diagram]

**Logic table**

<table>
<thead>
<tr>
<th>$S$</th>
<th>$\overline{R}$</th>
<th>$Q$</th>
<th>$\overline{Q}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$Q_0$</td>
<td>$\overline{Q}_0$</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>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$Q_0 = \text{value before S&R changes}$

$\overline{R} = 0 & \overline{S} = 1$:  
- $\overline{S} = 1 & \text{assume $\overline{Q} = 0 \rightarrow Q = 1$.}$
- $\overline{S} = 1 & \text{assume $\overline{Q} = 1 \rightarrow Q = 0$.}$
- $\overline{R} = 0 & \text{assume $Q = 0 \rightarrow \overline{Q} = 1$.}$
- $\overline{R} = 0 & \text{assume $Q = 1 \rightarrow \overline{Q} = 1$.}$

consistent $\overline{R}=0 & \overline{S}=1 \rightarrow Q=0 & \overline{Q}=1$
How does a flip-flop work?

Basic flip-flop: the SR latch

\[\overline{S} \quad Q\]
\[\overline{R} \quad \overline{Q}\]

- \(\overline{R} = 1 \land \overline{S} = 0:\)
  - The opposite of \(\overline{R} = 0 \land \overline{S} = 1\) by symmetry.

Logic table:

<table>
<thead>
<tr>
<th>(\overline{S})</th>
<th>(\overline{R})</th>
<th>(Q)</th>
<th>(\overline{Q})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>(Q_0)</td>
<td>(\overline{Q}_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>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\(Q_0\) = value before S&R changes
How does a flip-flop work?

Basic flip-flop: the SR latch

\[
\begin{align*}
\overline{S} & \quad \overline{R} \\
& \quad Q
\end{align*}
\]

\[
\begin{align*}
& \quad \overline{S} \\
\overline{R} & \quad \overline{Q}
\end{align*}
\]

Logic table

<table>
<thead>
<tr>
<th>$\overline{S}$</th>
<th>$\overline{R}$</th>
<th>$Q$</th>
<th>$\overline{Q}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$Q_0$</td>
<td>$\overline{Q}_0$</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>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$Q_0 = \text{value before S\&R changes}$

$\overline{R} = 1 \text{ & } \overline{S} = 1$: 

- $\overline{S} = 1 \text{ & assume } \overline{Q} = 0 \rightarrow Q = 1$.
- $\overline{S} = 1 \text{ & assume } \overline{Q} = 1 \rightarrow Q = 0$.
- $\overline{R} = 1 \text{ & assume } Q = 0 \rightarrow \overline{Q} = 1$.
- $\overline{R} = 1 \text{ & assume } Q = 1 \rightarrow \overline{Q} = 0$. 
How does a flip-flop work?

Basic flip-flop: the SR latch

![SR latch diagram]

**Logic table**

<table>
<thead>
<tr>
<th>$\bar{S}$</th>
<th>$\bar{R}$</th>
<th>$Q$</th>
<th>$\bar{Q}$</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$Q_0 = \text{value before S&R changes}$

- $\bar{R} = 1 \& \bar{S} = 1$:
  - $\bar{S} = 1 \& \text{assume } \bar{Q} = 0 \rightarrow Q = 1$.
  - $\bar{S} = 1 \& \text{assume } \bar{Q} = 1 \rightarrow Q = 0$.
  - $\bar{R} = 1 \& \text{assume } Q = 0 \rightarrow \bar{Q} = 1$.
  - $\bar{R} = 1 \& \text{assume } Q = 1 \rightarrow \bar{Q} = 0$.

- $\bar{R} = 1 \& \bar{S} = 1 \rightarrow Q = 1 \& \bar{Q} = 0$ (consistent)
- $\bar{R} = 1 \& \bar{S} = 1 \rightarrow Q = 0 \& \bar{Q} = 1$ (consistent)
How does a flip-flop work?

Basic flip-flop: the SR latch

![Flip-flop circuit diagram]

Logic table

<table>
<thead>
<tr>
<th>$S$</th>
<th>$\bar{R}$</th>
<th>$Q$</th>
<th>$\bar{Q}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$Q_0$</td>
<td>$\bar{Q}_0$</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>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$Q_0$ = value before S&R changes

$\bar{R} = 1$ & $\bar{S} = 1$:

- $\bar{S} = 1$ & assume $\bar{Q} = 0 \rightarrow Q = 1$. consistent
- $\bar{S} = 1$ & assume $\bar{Q} = 1 \rightarrow Q = 0$. consistent
- $\bar{R} = 1$ & assume $Q = 0 \rightarrow \bar{Q} = 1$. consistent
- $\bar{R} = 1$ & assume $Q = 1 \rightarrow \bar{Q} = 0$. consistent

Two settings are possible → i.e. flip-flop keeps its state.
SR Latch Switch Debouncer

SR latch flip-flops are not used much for memory, but they are used for debouncing switches.

Switch Bounce:

When a switch is toggled it will not go smoothly from HIGH to LOW, or vice versa.
Clock Circuit Analysis:

- \( C = 1 \) & \( D = 1 \) → \( \overline{S} = 0 \) & \( \overline{R} = 1 \).
- \( C = 1 \) & \( D = 0 \) → \( \overline{S} = 1 \) & \( \overline{R} = 0 \).

- \( C = 0 \) & \( D = 1 \) → \( \overline{S} = 1 \) & \( \overline{R} = 1 \).
- \( C = 0 \) & \( D = 0 \) → \( \overline{S} = 1 \) & \( \overline{R} = 1 \).

Logic table:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( D )</td>
<td>( C )</td>
<td>( Q )</td>
<td>( Q_0 )</td>
</tr>
<tr>
<td>( X )</td>
<td>0</td>
<td>( Q_0 )</td>
<td>( Q_0 )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Clock Circuit Analysis:

- $C = 1 \land D = 1 \rightarrow \overline{S} = 0 \land \overline{R} = 1$. **Clock HIGH:** D sets the flip-flop state
- $C = 1 \land D = 0 \rightarrow \overline{S} = 1 \land \overline{R} = 0$. 
- $C = 0 \land D = 1 \rightarrow \overline{S} = 1 \land \overline{R} = 1$. **Clock LOW:** flip-flop state is locked
- $C = 0 \land D = 0 \rightarrow \overline{S} = 1 \land \overline{R} = 1$

**Logic table**

<table>
<thead>
<tr>
<th>$D$</th>
<th>$C$</th>
<th>$Q$</th>
<th>$\overline{Q}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>0</td>
<td>$\overline{Q}_0$</td>
<td>$\overline{Q}_0$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Clock Circuit Analysis:

- **Clock HIGH:** D sets the flip-flop state
  - \( C = 1 \) & \( D = 1 \) → \( S = 0 \) & \( R = 1 \).
  - \( C = 1 \) & \( D = 0 \) → \( S = 1 \) & \( R = 0 \).

- **Clock LOW:** flip-flop state is locked
  - \( C = 0 \) & \( D = 1 \) → \( S = 1 \) & \( \bar{R} = 1 \).
  - \( C = 0 \) & \( D = 0 \) → \( S = 1 \) & \( \bar{R} = 1 \).

### Logic table

<table>
<thead>
<tr>
<th>( D )</th>
<th>( C )</th>
<th>( Q )</th>
<th>( \bar{Q} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X )</td>
<td>0</td>
<td>( Q_0 )</td>
<td>( \bar{Q}_0 )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Master-Slave D-type Flip-Flop

Note: The flip-flop triggers on the falling edge of the clock.
74LS74 D-type edge-triggered flip-flop

Both PRE and CLR behave like S and R inputs, respectively, on the SR latch.

**IMPORTANT:** Both PRE and CLR must be high for normal D-type operation.

**Note:** The flip-flop triggers on the rising edge of the clock.
74LS74 D-type edge-triggered flip-flop

Both PRE and CLR behave like S and R inputs, respectively, on the SR latch.

**IMPORTANT:** Both PRE and CLR must be high for normal D-type operation.

**Note:** The flip-flop triggers on the rising edge of the clock.
JK-type flip-flops are used in counters.

Logic table for clock falling edge

<table>
<thead>
<tr>
<th>J</th>
<th>K</th>
<th>Q_{n+1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Q_n</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>\overline{Q_n}</td>
</tr>
</tbody>
</table>
Flip-flops in FPGAs

Architecture of a single Logic Element

- LUT
- CLOCK triggers
- Memory (a few bits)

Frequently a D-type Flip-Flop

FPGAs are already set-up for synchronous circuit designs
Flip-flops in FPGAs

Architecture of a single Logic Element

- **LUT**
- **CLOCK triggers**
- **Memory** (a few bits)

- **Inputs**
  - Clock signals

- **Output**
  - Global
  - Local

Feedback loop

Frequently a D-type Flip-Flop

FPGAs are already set-up for synchronous circuit designs
Synchronous programming in Verilog (I)

module adder_always(clock, input1, input2, result);

input [3:0] input1;  // 4-bit input, first number
input [3:0] input2;  // 4-bit input, second number

input clock;        // 1-bit clock input

output reg [4:0] result; // 5-bit output register

always @(posedge clock) // performs this section on the positive clock edge
begin
  result = input1 + input2; // Standard 4-bit addition
end

endmodule
module adder_always(clock, input1, input2, result);
input [3:0] input1; // 4-bit input, first number
input [3:0] input2; // 4-bit input, second number
input clock; // 1-bit clock input
output reg [4:0] result; // 5-bit output register
always@(posedge clock) // performs this section on the positive clock edge
begin
result = input1 + input2; // Standard 4-bit addition
end
endmodule
Synchronous programming in Verilog (I)

```verilog
module adder_always(clock, input1, input2, result);
    input [3:0] input1;    // 4-bit input, first number
    input [3:0] input2;    // 4-bit input, second number
    input clock;           // 1-bit clock input
    output reg [4:0] result;    // 5-bit output register

    always@(posedge clock)
        begin
            result = input1 + input2;    // Standard 4-bit addition
        end
endmodule
```

Read as “always at the positive clock edge do the following … ”

“always” is the core command for synchronous programming, it should be used as frequently as possible.

“assign” should be used as little as possible. It is only useful for DC-type signals (signals that don’t change).
Synchronous programming in Verilog (II)

Quartus II circuit simulation

![Simulation Diagram](image-url)
Synchronous programming in Verilog

Quartus II circuit simulation

Clock Line

No more glitches
How did the FPGA implement the circuit?
How did the FPGA implement the circuit?

Tools > Netlists > Technology Map Viewer

D-type edge-triggered flip-flops
Always use “always”

– A. Stummer, U. of Toronto.
Parallel programming in Verilog

- The “always” structure is used for exploiting the parallel processing features of the FPGA.
- Parallel processing must almost always be synchronous if several processes exchange data.

Parallel and Sequential processing examples:

```
Sequential
always@ (negedge clock)
begin
a = b;
c = a;
end
```

```
Parallel
always@ (negedge clock)
begin
a <= b;
c <= a;
end
```
The “always” structure is used for exploiting the parallel processing features of the FPGA.

Parallel processing must almost always be synchronous if several processes exchange data.

Parallel and Sequential processing examples:

Sequential
always@ (negedge clock)
begin
  a = b;
  c = a;
end

Parallel
always@ (negedge clock)
begin
  a <= b;
  c <= a;
end

\[
\begin{align*}
  &\text{c = b} \\
  &\text{a = b} \\
  &\text{c = a (previous value)}
\end{align*}
\]

executed simultaneously