Digital Logic Design
234262
Alex Bronstein
Computers Around Us
Course Goals

- Learn how to design logic systems
- Learn the design of a simple CPU (MIPS)
- Stop worrying and learn to love digital systems 😊
Short bio

In the Academia:

- B.Sc. and M.Sc. in EE, Technion (1998-2005)
- Ph.D. in CS, Technion (2005-2007)
- Prof., EE@Tel Aviv University (2010-2016)
- Prof., CS@Technion (from 2016)
- Research interests: computer vision, machine learning, computational geometry, imaging and image processing

In the Industry:

- Cofounder and VP Technology, Novafora Inc. (2006-2009)
- Cofounder, Invision Ltd. (2009-2012)
- Research Scientist & Principal Engineer, Intel Corporation (from 2012)
- Cofounder and Chief Scientist, Videocites Ltd. (from 2014)
Course Staff

- **Alex Bronstein** (lecturer in charge)
  Taub 719   bron@cs.technion.ac.il

- **Chaim Baskin** (TA in charge)
  Taub 327   chaimbaskin@cs.technion.ac.il

- **Dolev Adas** (TA)
  Taub 223   sdolevfe@cs.technion.ac.il

- **Moshe Liechtenstein** (TA)
  Taub 313   smosesli@cs.technion.ac.il

- **Shafik Nassar** (TA)
  Taub 320   shafiknassar@campus.technion.ac.il
Course Formalities

- **Grade:**
  - 30% home assignments
  - 70% final exam

- **Assignments:**
  - 2 “dry” exercises (2.5%)
  - 2 “wet” Verilog coding exercises (12.5% each)
  - Passing grade (>54) mandatory in each of the “wet” exercises

- Different course format in this semester!
Hardware
Hardware Design Flow (ASIC)

Frontend
- Design specs
- Architecture
- RTL design
- Functional verification
- Logic synthesis

Technology library

Backend
- Tapeout
- Physical verification
- Static timing analysis
- Placement & routing
- Floor planning

Foundry
- Mask tooling
- Fabrication
- Packaging
Hardware Design Flow (FPGA)

1. Design specs
2. Architecture
3. RTL design
4. Functional verification
5. Logic synthesis
6. Placement & routing
Software Design Flow

1. Design specs
2. Architecture
3. High-level code
4. Compilation
# Software vs. Hardware

<table>
<thead>
<tr>
<th></th>
<th>Software</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Design cost</strong></td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td><strong>Design cycle</strong></td>
<td>Fast</td>
<td>Slow</td>
</tr>
<tr>
<td><strong>Language</strong></td>
<td>High-level languages (Java, C#, C++, C)</td>
<td>HDL (Verilog, VHDL)</td>
</tr>
<tr>
<td><strong>Code undergoes</strong></td>
<td>Compilation</td>
<td>Synthesis</td>
</tr>
<tr>
<td><strong>Parallelism</strong></td>
<td>Usually sequential (SIMD, threading, IPC)</td>
<td>Inherently parallel</td>
</tr>
<tr>
<td><strong>Target platforms</strong></td>
<td>CPU (x86, ARM, GPU, DSP)</td>
<td>FPGA or ASIC (lots of processes)</td>
</tr>
</tbody>
</table>
Good Engineering Practices

Modularity
- Design basic building blocks (modules)
- Connect modules together to form another module
- A module can be replaced by a new version without affecting the rest of the system

Standardization
- Using a library with a small amount of standard modules
- Clearly defined functionality and interfaces
- Don’t reinvent the wheel

Abstraction
- Decomposition of design process into a hierarchy of levels
- Hide unnecessary lower-level details
Abstraction

- Simplifies design and understanding of complex systems
- A higher level of abstraction is based on the level beneath
- A higher level of abstraction hides details of lower levels
- Always try to solve problems at the highest possible level of abstraction
- Example of abstraction hierarchy:
  - High-level language
  - Machine language
  - Register-transfer level (RTL)
  - Gate level, analog electronics
  - Physics
Abstraction vs. Implementation

- **Abstraction**: low level $\rightarrow$ higher level
- **Implementation**: high level $\rightarrow$ lower level
- Abstraction requires adhering to a certain set of rules to be valid

- High-level language: variable
- Register-transfer level: register
- Gate level: NAND gates
- Physics: p-type and n-type MOSFETs
Abstraction of Digital Systems

- **Input:** $V_{\text{in}} > V_{\text{IH}}$
- **Output:** $V_{\text{out}} > V_{\text{OH}}$
- **Input:** $V_{\text{in}} < V_{\text{IL}}$
- **Output:** $V_{\text{out}} < V_{\text{OL}}$
- $V_{\text{IL}} < V < V_{\text{IH}}$ undefined

- **Noise margins** increase robustness to noise added to signal passing from the output of one component to the input of another.
- Allows modularity!
Timing

- **Contamination delay** \( t_{CD} \) – minimum time following input change during which validity of previous output is guaranteed

- **Propagation delay** \( t_{PD} \) – maximum time following input change after which the validity of new output is guaranteed
Combinational Logic (CL)

- Output = Boolean function of the input

- Building blocks:
  - Logic gates
  - Wires
  - Combinational logic sub-systems

- Building rules:
  - No feedback
  - No floating wires (except inputs and outputs)
  - Number of inputs each output drives does not exceed **fanout** limitations
Static Timing

- If inputs to CL system are valid and stable, after settling time, outputs are also valid and stable.

\[
\begin{array}{c|cc}
\text{INPUT} & \text{OLD} & \text{NEW, Stable} \\
\hline
\text{OUTPUT}_1 & \text{OLD} & \text{NEW, Stable} \\
\text{OUTPUT}_2 & \text{OLD} & \text{NEW, Stable} \\
\end{array}
\]

- Allows resilience to inaccurate analog components.
- Prevents noise accumulation during signal propagation.
- Allows cheap mass production of components.
Static Timing

Example: Inverter (NOT gate)

- $V_{\text{in}} < V_{\text{IL}} \implies V_{\text{out}} > V_{\text{OH}}$
- $V_{\text{in}} > V_{\text{IH}} \implies V_{\text{out}} < V_{\text{OL}}$

Actual components have different analog characteristics due to:

- Fabrication tolerances
- Temperature changes
- Supply voltage fluctuations
Static Timing

- **Contamination delay** (t<sub>CD</sub>) – minimum time following input change during which validity of *all* previous outputs is guaranteed
  - Components in parallel: min of t<sub>CD</sub>
  - Components in series: sum of t<sub>CD</sub>

- **Propagation delay** (t<sub>PD</sub>) – maximum time following input change after which the validity of *all* new output is guaranteed
  - Components in parallel: max of t<sub>PD</sub>
  - Components in series: sum of t<sub>PD</sub>
Synchronous Static Memory

- Positive edge-triggered D flip-flop (DFF)

- Logical characteristic:
  Output $Q$ assumes the value of $D$ after $CLK$ rise

- Timing characteristic:
  If input is held constant $t_{SU}$ before and $t_{H}$ after clock rise then old output will stay valid at least $t_{cCQ}$ and new output will become valid at most $t_{pCQ}$ after clock rise
Synchronous Static Memory
Synchronous Sequential Logic

- State machines

- Building blocks:
  - Combinational logic modules
  - Clocked memory modules
  - Wires

- Building rules:
  - No CL loops unless through at least one memory module
  - Clock period long enough to satisfy timing requirements
  - All inputs remain stable enough time to satisfy timing requirements
Finite State Machines (FSM)

- A finite set of states $X \in \{0, \ldots, K-1\}$
  - $k$ bits represent $2^k$ states
- $n$-bit input $U$
- $m$-bit output $Y$

- **State update function:**
  
  \[ X[t+1] = \lambda(X[t], U[t]) \]

- **Output function:**
  - Mealy FSM: $Y[t] = \Omega(X[t], U[t])$
  - Moore FSM: $Y[t] = \Omega(X[t])$
FSM Timing

- Clock period
  \[ T > t_{PCQ} + t_{PD} + t_{SU} \]
**FSM Timing**

- **Clock period**
  \[ T > t_{PCQ} + t_{PD} + t_{SU} \]

- **Input setup time**
  \[ t_{SU}(U) > t_{PD}(CL) + t_{SU} \]
FSM Timing

- **Clock period**
  \[ T > t_{PCQ} + t_{PD} + t_{SU} \]

- **Input setup time**
  \[ t_{SU}(U) > t_{PD}(CL) + t_{SU} \]

- **Input hold time**
  \[ t_{H}(U) > t_{H}(DFF) + t_{CD}(CL) \]
Additional Timing Considerations

- **Clock skew**: clock signal arrives with different delay to different memories
- **Clock jitter**: clock is not exactly periodic
- **Asynchronous inputs** may violate hold requirement and cause *metastability*
Latency and Throughput

- **Latency**: amount of time passing between the beginning of a calculation and its end (input-to-output delay)
- **Throughput**: rate at which calculations are performed (=calculations per unit of time)
Latency and Throughput

- Example: sum of N numbers

**Sequential implementation**
- Latency: \( N \times t_0 \)
- Throughput: \( \frac{1}{N \times t_0} \)

**Combinational implementation**
- Latency: \( \log_2 N \times t_0 \)
- Throughput: \( \frac{1}{(\log_2 N \times t_0)} \)