Advanced Logic Design

angeloperotti7

February 2025

Contents

0.1 Introduction

- exam:
  - written exam 50
  - project: worth the other 50
  - oral: optional, +-3
- material
  - slides on moodle
  - CMOS VLSI Design, A circuits and Systems Perspective, 4th ed, Addison-Wesley
  - Computer Architecture: a quantitative approach, Morgan Kaufmann

Chapter 1

Design Methodologies

1.1 Abstraction in IC Design

1.1.1 Transistor Level

- Design: devices litterally draw on the IC
- Transistor-Level layout
- Analysis: Transistor-level device simulation

1.1.2 Transistor schematics level

- Design: schematics drawn
- Synthesis: actual layout automatically generated through transistor-level layout compaction
- Analysis: Transistor-level device simulation
- can handle from 10's to few thousand transistors

1.1.3 Logic gates level

- Design: Logic schematics drawn using a gate-level layout editor
- synthesis: transistor schematics and layout generated
- automated place and route and compaction
- analysis: Gate-level simulator, and device level simulator

1.1.4 Standard cells level

- Draw schematics from pre-defined cells
- synthesis: layout generated using cell-based place and route
- analysis: Logic-level and gate-level simulation
- analysis: Static Timing Analysis

1.1.5 Register-transfer lebel (RTL)

- Design: provide a description of the state (registers) and how it changes during operations
- synthesis: standard cell synthesized and layout automatically generated
- analysis: behavioral and gate-level simulation
- analysis: static timing analysis using abstracted gate level models

1.1.6 Model-based design

- Design: cenceive a mathematical model of the system
- synthesis: derive algorithms that implement the mathematical model
- analysis: formal mathematical analysis of the model
- analysis: model simulation
- can handle milions of gates

Chapter 2

Computational Complexity

2.1 computational complexity

2.1.1 what is computational complexity?

the computational complexity is the rate at which the number of operations increases as the size of the system grows

2.1.2 how to measure it

- we are interested in the *asymptotic behavior*
- we want to compare the complexity of different methods
- for analysis we use the big O notation (o grande)
  - let n be the number of elemets
  - $f(n) = O(g(n))$ if there exists a $c > 0$ and $n_0 > 0$ such that $f(n) \leq c(g(n)) \quad \forall n \geq n_0$

2.1.3 how to compute it?

we count the number of operations as a function of the system size, by going through the algorithm that implements it

2.1.4 types of complexity

Time Complexity

- number of elementary computational steps
- the rate at which computation time increases as the size of the problem increases

Space Complexity

- required number of memory locations
- the rate at which memory requirement increases as the size of the problem increases

2.1.5 functions to compare

Polynomial Complexity

$O(n^k)$, where n is the size of the input and k is a constant

- 52 a constant (k=0)
- $n^{1/2}$ sublinear
- n linear
- n^2 quadratic
- n^3 cubic

logarithmic complexity

- $\log_n$ logarithmic
- $n \log_n$ loglinear

non polynomial functions

- 2^n exponential
- $n!$ factorial

2.1.6 Operating with notation

- when adding big-O's the highest order wins
  - $O(n) + O(n^2) = O(n^2)$
- when multiplying big-O's we take the product of their orders
  - $mO(n) = O(mn)$
  - $O(m)O(n) = O(mn)$

2.1.7 Terminology

- A problem has complexity in **P** if it can be solved in polynomial time in size of the input
- A problem has complexity in **NP** if it can be solved in polynomial time in the size of the input by a non deterministic machine

Chapter 3

FPGAs

3.1 Programmable Logic Devices

3.1.1 pre-connected programmable matrices

- Connections are prefabricated, but functions can be programmed
  - programming is done by designer
  - useful for prototyping
- classification:
  - Programming technique
    - ROM
    - Fuse
    - EPROM
    - RAM
  - logic style
    - array
    - tables
  - interconnection style
    - channels
    - matrix

3.1.2 logic function as memory

- easy to use a memory to realize a logic function
  - memory address lines correspond to the independent variables
  - the memory contains the truth table of the desired function
  - by changing the memory content one can modify the implemented function
- practical for a few inputs
  - it does not take advantage of logic minimization
  - the size increases exponentially with the number of inputs

3.1.3 Programming

- using a ROM the device can only be programmed during fabbrication
  - once the presence of the transistor is decided it can no longer be changed
  - structure is very compact
- to make the device run-time programmable we could use a RAM
  - Dynamic RAM is small but kinda slow

  - refresh cycle make the function unavailable for a certain time
  - static RAM is larger in size
  - RAM is ereased each time the device is powered off
- alternative non-volatile memories have been developed
  - based on fuses and anti-fuses
  - based on floating-gate transistors

3.1.4 one time programmable devices

- PROM
  - programmable Read Only Memory
  - uses a nickel-chrome technology to make fuse contacts
  - the fuse sits between the transistor and the bit-line
- Working Principle
  - if the *fuse is present* the transistor works normally and the cell contains a *zero*
  - if the *fuse is broken* it is as if the transistor is not there and the cell contains a *one*
- programming
  - the desired fuses are blown by applying high voltage and current, and enabling the corresponding transistor
  - once blown, the contact can no longed be recovered
  - if a different programming is desired a new device is needed

3.1.5 antifuse

- just like the fuses but the connection is initially open
  - strong current induce the formation of a conductive layer
  - connection are created,not destroyed
- implementatiom through thin oxide
  - small layer of ONO, an oxide that disappears when high currents are passed and than creates a connection

3.1.6 advantages and disadvantages

  - fuses and antifuses are very small
  - programming is non volatile
  - there need special fabrication steps
  - need of a special programmer

3.1.7 non volatile writable memories

  - PROM can only be programmed once
    - not sufficiently flexible, especially when the circuit is changed often
  - need devices whose characteristics can revert back to normal
    - FAMOS transistor
    - a second isolated and floating gate is placed between the gate and the channel
    - besides an increase in oxide thickness the floating gate alters the threshold voltage

3.2 complex programmable devices

3.2.1 CPLD

- Complex Programmable Logic Devices
  - if more resources are needed several SPLDs can be integrated together
  - implementation is naturally multi level
- interconnections
  - Also the interconnections must be programmable
- CPLD characteristics
  - High Speed
  - CAn easily handle two level expressions
  - good for systems that use a lot of logic and few flip-flops
  - Ex: Graphic controllers, LAN, UART, cache controllers

3.2.2 FPGA

- Similar in structure to the CPLDs
  - they use more complex programmable blocks based on multiplexers or look-up tables
  - often have dedicated logic for arithmetic and memory
- composed of
  - CLB
  - interconnections
- requirements
  - high performance
  - high density

3.2.3 static RAM cells

- Despite the size RAM cells are commonplace
  - Extreme programming semplicity
  - can be used as a traditional memory, though a look-up table (LUT)
  - can be used to generate fixed-valued signals that drive other devices, such as multiplexers
  - often used as control signals for pass transistors and transmission gates

3.2.4 How to implement a function with MUXes

- By selecting the inputs of the mux between 0,1,X,Y and their complements you can realize all two variable functions

3.2.5 CLB with MUX

- Configurable Logic Block based on multiplexers

3.2.6 CLB with LUT

- configurable logic block based on look-up table
  - in practice its like a small RAM

3.2.7 What is the best CLB size?

- if a function is *smaller* than the CLB, part of it is wasted
  - need to size the CLB appropriately
- High granularity (fine-grain, small CLBs)
  - small and simple CLBs result in a more effiecient use of the devices, because less of them is wasted
  - there will be considerable interconnection complexity
- Low Granularity (coarse-grain, large CLBs)
  - more complex CLBs can implement complex functions in one block
  - can be more waste
  - interconnections are cheaper, simpler
- the optimal granularity depends on the application
  - studies show that LUTs with 4 to 6 inputs offer a good compromise between logic complexity and interconnections

3.2.8 Programmable interconnections

- Fuse and antifuse
  - the interconnection takes place directly through the fuse or antifuse
  - interconnections are generally sparse, many fuses need to be blown, slow programming process
  - antifuse are therefore preferable
- RAM cells
  - the interconnection takes place through pass transistors
  - when the transistor is active (RAM cell contains 1) the connection is present
  - when not active it is an open circuit

3.2.9 RAM based CLBs

- The look-up tables can be reconfigured as
  - 4-input logic function
  - 16 bit RAM memory
  - 16 bit shift register
- this way the same object can be used as combination or sequential element
  - gives higher flexibility and resource utilization

3.2.10 Logic Cell in Xilinx FPGAs

- Simplified structure
  - Combinational block
  - sequential block, configurable as a flip-flop or a latch
  - combinational or sequential output

3.2.11 Slice

- Contains two logic cells
  - Each with its own inputs
  - fast internal connections
  - Share the same flip-flop and clock (saves configuration memory)

3.2.12 Configurable Logic Block

- Made of 4 slices
  - 8 logic cells in total
  - fast interconnections between the slices

3.2.13 additional components

RAM

- Several applications require substantial ammount of RAM
  - you can use CLBs(distributed RAM)
  - there are often hard RAM blocks available on the chip
  - placed on the periphery or alternating with the CLBs

MAC

- Multiplexers are slow if implemented with CLBs
  - for this reason they are often pre-fabricated
- a frequent operation is multiplication followed by result accumulation
  - Useful to compute convolutions
  - you can find dedicated multipliers with accumulator and an adder

3.2.14 Embedded Processors

- some parts of the system are better done in software
  - when performance is not so important
- some FPGAs have an on-board processor
  - Removes the need to have different components on the PCB
  - it can have RAM, I/O peripherals, and more
- sometimes they are located in the middle of the FPGA
  - there might be several kind, from low-power to high-performance, for different applications

3.2.15 Soft Core

- the CLB blocks can be configured to operate as a microprocessor
  - it's just another circuit
  - they are also called soft core, because they do not have a specific implementation
- soft cores are typically simpler and slower
  - speed can be 30 to 50 percent lower than a hard core
- you can design many variants
  - optimized for a particular application
  - with dedicated instruction sets
  - PicoBlaze, MicroBlaze, Nios, Leon 2

3.2.16 Clock manager

- the clock signal comes from outside
  - must be routed to all flip-flops
  - a dedicated devices, called clock manager, cleans up the signal from jitter and distributes it to all the rest of the FPGA

3.2.17 Programming

- How is the FPGA ocnfigured?
  - it is like a memory
  - to avoid using too many pins, all cells are connected as a very long shift register
  - it is sufficient to clock the data and insert them one by one in the device
- the configuration bitstream must be stored somewhere in an external memory
  - for example a FLASH
  - the FPGA controls the memory at power up to load the configuration
- several devices can be daisy-chained

3.2.18 technology comparison

3.2.19 TLDR

- Programmable logic devices are very common in digital design
  - For every ASIC design there are at least 100 that use an FPGA
  - Performance and integration levels today let FPGAs handle high frequency signals
- Several different architectures
  - different technologies
  - different available devices
  - a choice is not always simple
- Very useful for prototyping
  - Can implement a circuit in relative little time
  - evaluation boards are available with FPGAs as well as common interfaces and peripherals

Chapter 4

Hardware description languages

4.1 Preview

- how does it work?
  - software is translated in a circuit
  - we try to use high level concepts

4.2 VHDL

- the VHDL language
  - Very high speed integrated circuit hardware description language
    - VHSIC HDL = VHDL
  - objectives:
    - Documentation
    - simulation
    - synthesis
  - methodologies
    - Top-down
      - proceeds from a high level to a lower level decomposing a system into sub-systems
    - Bottom-up
      - creates complex systems assempbling simpler systems
    - meet-in-the-middle
      - decompose the system into sub systems, up to reaching elements of a library
  - views
    - data flow
    - sructural
    - behavioral
- Basic concepts:
  - Entity
    - The *entity* is a basic object of VHDL
      - it corresponds to a block, a module, an element
      - similar to a class in C++, or a function in C
      - It is identified by a name
      - has a number of inputs and outputs
      - *Define the interface of the block*
  - Architectures
    - every architecture corresponds to an entity
    - an entity may have several different architectures

    - *it describes its function* through equations, structural connections of other blocks (hierarchy), or through an algorithm

- description styles
  - Data flow or equations
    - describes the function through boolean equations
    - expresses the dependency of the outputs as a function of the inputs and the internal signals
    - the equation form a system
  - structural
    - describes the function as the connection of components in a hierarchy
    - the new entity can in turn be used as a component in another architecture
  - behavioral
    - describes the function through an algorithm
    - the code describes how to determine the value of the output as a function of the inputs, the internal signals and variables
  - ex: of a 2-input AND gate
    - interface
      - inputs **x** and **y**, output **z**
    - behavior
      - we use the pre-defined *AND* operator
      - the operator <= is the assignment

- Testbench
  - to perform a simulation we must provide values to the input signals
    - the block that does this is called the testbench
    - it can be implemented mixing the dataflow and the structural view
    - the testbench closes the system, therefore its entity has no input nor outputs

- operators and indentifiers
  - VHDL has all the boolean operators
    - not
    - and, or
    - nand, nor
    - xor, xnor
  - operator precedence
    - **not** has precedence over all others
    - *the others have all the same precedence*, applied from left to right
    - Careful with parenthesis
  - identifiers
    - used for names for entities, architectures and signals
    - VHDL is *case insensitive*, so **x** and **X** are the same

- Multiple expressions
  - we can use several expressions linked by internal signals
    - declare the names and types of the signals before the equations
    - the order of the equations has no importance

- delays
  - we can specify a delay for an expression
    - use the **after** operator followed by the time
    - the result is not assigned to the destination signals until after the defined simulation time
    - if no delay is specified, an arbitrary small delay, denoted δ, is assumed

4.3 VHDL in practice

Chapter 5

Static Timing Analysis

5.1 how fast does this circuit run?

- the output of the flip flops (the state) changes at che clock active edge
- their inputs (the next state must be stable before the next active edge
  - the state information must propagate from the output of the flip flops to their input within a clock period
- *the **worst case delay** of the combinational network therefore defines the minimum clock period (and the maximum frequency)*
- ex: a 1GHz clock the combinational network must compute the next state in at most 1 ns
- delay is intended between the output of each flip flop and the input of any other flip flop

5.1.1 flip flop timing

- the state of the flip flop depends at the same time on the clock and the value of the input D
  - the clock edge causes the state change
  - the circuit uses a feedback to store the information
  - this feedback needs some time to stabiize
- the input signal D must be stable some time *before* and some time *after* the clock edge
  - **setup time:** time *before* the edge in which D must be stable
  - **hold time:** time *after* the edge in which D must be stable
- if these conditions are not met, the flip flop may end up in a random state (metastability)

5.2 solutions for violations

5.2.1 Solutions for setup violations

- if there is a setup violation, then the circuit is too slow
  - the next state is not computed quickly enough
  - one may try to optimize and *speed up the logic*
  - alternatively one may *slow down the clock*
  - the maximum clock frequency depends on the largest delay between any two flip flops, plus the setup time
  - binning process

5.2.2 solutions for hold violations

- for a hold violation, ite means the logic is too fast
  - the output of a register propagates to the input of another too quickly
  - all happens on the same clock edge, changing the clock frequency does not work
  - one must slow down the logic, for instance adding delay buffers
  - no binning, if there is a hold violation the circuit is trashed

5.3 Analyzing design performance

- Basic questions
  - does the design meet a given timing requirement?
  - how fast can i run the design?
  - assume we know the delays of blocks in the network
- why not just use ordinary gate-level simulations?
  - requires way too many input patterns
    - must simulate transition between any two inputs patterns
    - exponential in the number of inputs
    - even worse if we consider sequences needed to initialize latches
  - so, what do we do instead?
    - *separate functions every time*
    - determine *when* transitions occur without worrying about *how*

5.3.1 separation of concerns

- Gate level simulation good at verifying both functionality and timing
  - but *slow* (1-10 clock cycles of 100k gate design per 1 CPU second)
  - *incomplete:* results only as good as your vector set -easy to overlook incorrect timing/behavior
- Speed up verification using cycle-based simulation
  - but it doesn't check timing
- solution: check timing separately using a time analyzer
  - *statistically* check that all combinatorial paths meet the clock cycle constraints

5.3.2 conditions for separation

- separating functionality and timing possible because of the synchronous assumption
  - only care about time of latest data arrival at the registers
  - detailed timing (glitches) irrelevant
- advantage
  - can use efficient static techniques to check timing
- asynchronous design unable to take advantage of this separation
  - timing is part of the functionality
  - must check the two simultaneously

5.4 uses of Static Timing Analysis (STA)

- timing verification
  - ensure design meets the timing costraints
- timing driven optimization
  - obtain accurate timing information quickly to drive optimization tools

5.4.1 what STA does

- forward: given the signal arrival at time at I1, I2 and I3
  - compute the arrival time at I3, I4 and I5
- Backward: Given the required time at I3, I4 and I5
  - compute the required time at I1, I2 and I3

  ęgister to register path is a path throuth the combinatorial logic between any pair of registers
  - for every pair of registers
    - find the *slowest path* (propagational through the path takes the longest time)
    - find the *fastest path* (propagation through the path takes the shortest time)
  - the slowest path between any two registers determines the highest clock speed
    - also called **critical** path
    - you can optimize other paths, but clock frequency does not improve!

5.4.2 cycle time

- for FFs to work correctly, input must be stable during
  - setup time (T_{setup}) before clock arrives
  - determined by delay of longest path

5.4.3 hold time

- for FFs to work correctly, input must be stable during
  - Hold time (T_{hold}) after ckicj arruves
  - determined by delay of shortest path

5.5 elements of timing verification

- to verify a circuit timing we need
  - accurate delay calculation
  - timing analysis engine
- Delay calculation
  - delay numbers for gates
  - delay numbers for wires
- timing analysis engine
  - circuit path analysis

5.6 timing models

- *propagation delay*
  - time between transition at input and at the output
  - transition: crossing of 50%
  - usually rise and fall delays are different
- rise and fall *slew rate*
  - time a signal takes to go from 10% to 90% (or viceversa) of its range

gate delay and output slew

- delay and output slew of a gate depends on
  - the output *load*
  - the *slew* of the input transitions
  - the *input* that makes the transition
  - the *kind of input transition*
- determined for several pairs of input slew and output load via circuit simulation or measurements
  - get exact data for a number of pairs
  - build a matrix (function of load and slew)
  - interpolate/extrapolate the other pairs

Delay tables for gates

- for every gate a delay table is constructed that gives
  - propagation delay
  - output slew
  - as a function of input slew and output capacitance

interconnection delay

- determined by resistance, capacitance and inductance of wires
  - wire delay becoming more and more significant
  - several models, and various levels of accuracy have been proposed
  - wire-loaded model: estimate wire delay as a function of circuit size

simplifying assumptions

- to illustrate this method we will use the following simplifications:
  - only one clock, arrives at the registers at the same time
  - gate delay is just one number, fixed
  - interconnect delay given or ignored
- real tools can handle most cases
  - multi phase clocks
  - clock skew
  - complex delay models

Mathematical Model

- to analyze timing, we use a graph model
  - well suited for schematic diagrams
- A graph G=(V,E)
  - A vertex v is a boolean function with delay
  - An edge e is an interconnection with delay
  - already seen for event-driven simulation
- must find the *longest and the shortest path* in the graph
  - general problem, can be applied to many other contexts

5.7 STA algorithm

- Remove all registers and keep only combinational logic
  - the combinational logic is acyclic (no feedback)
  - Registers outputs become inputs to logic network
  - register inputs become outputs of logic network
- Arrival time at inputs is given
  - including arrival time at output of registers
- propagate arrival times thorugh the logic gates and compute arrival times for internal nodes until outputs are reached
- Traverse the network *breadth first* (before we traverse any node we must traverse all previous nodes in the graph topology) in topological order
  - assumes no cycles in the graph
- for each node v, compute the largest arrival time $A(v)$ and the smallest arrival time $a(v)$ as a function of the input arrival time from each node w_i
  - $A(v) = max(A(w_i) + d)$ for all w_i
  - $a(v) = min(a(w_i) + d)$ for all w_i
- topological order
  - must proceed in topological order because for each gate we need to know the data from all of its previous gates (its fanin)

5.7.1 STA algorithm: complexity

- each node visited exactly once
  - if traversal is in topological order
  - topological order is easy to enforce by working backwards from the outputs
- max and min operations proportional to number of gate of inputs: O(g)
  - g independent of design size n
- overall complexity linear in circuit size
  - there are n gates
  - $O(gn)=O(n)$ because g is a constant

5.7.2 ex

5.7.3 in practice

- how do we enforce topological order?
  - solution 1
    - start from inputs, traversing the graph breadth first
    - every time a node is traversed for a number on times equal to the number of its inputs, assign it a level grater than the level of all its fanin
    - order the gates according to their level
  - solution 2
    - start from the output
    - recursively descend through the network to the inputs

5.7.4 STA

- recursion terminates because we assume the graph is acyclic
- total number of paths exponential in the number of logic gates
- but timing analysis is linear
  - does not follow every path individually
- delays through combinational network computed for all possible input patterns
  - tractable because restricted to only best and worst case

STA for design

- Largest arrival time gives worst case clock speed
- for design we want to set the clock speed and determine the conditions that the circuit must satisfy to meet the constraint
  - set the required arrival time at the outputs
  - propagate reqired time to the rest of the network

STA algorithm for required time

- Same as arrival time
  - but recursion is on the fanout nodes (reverse topological order)
- fanout in general not limited
  - complexity may be higher
  - in practice fanout is limited so complexity is again linear
- can set latest and earliest arrival time
  - latest time to satisfy setup conditions: data must arrive no later than R
  - Earliest time to satisfy hold conditions: data must arrive no earlier than r

timing slack

- For each node in the network we compute
  - the earliest and latest arrival time
  - the earliest and latest required time
- slack: difference between required and arrival time at each node
  - $S(v) = R(v) - A(v)$
  - $s(v) = a(v) - r(v)$
- the slack gives a measure of timing flexibility at the nodesdfghzxcvbz