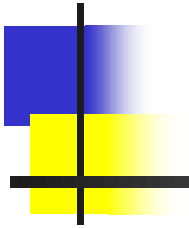


2A : Introduction to Digital System Design

CG3207 Computer Architecture

Rajesh Panicker, ECE, NUS

Note :

- Slides not covered in detail in the class are left as a self-learning exercise

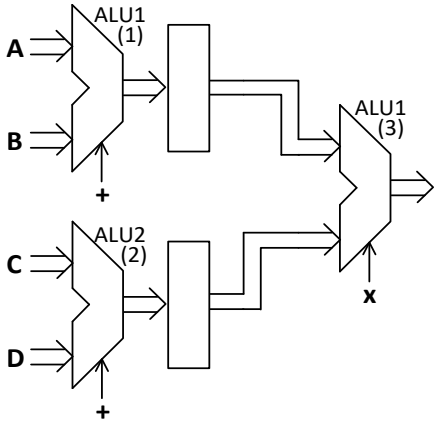
Acknowledgement :

- Some ideas from EE4218 by Dr. Ha Yajun and EE4415 notes by Prof. Lian Yong

System Design Considerations

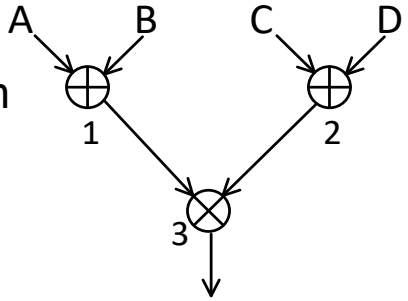
- System : sensor -> processor -> actuator
- Considerations
  - Time-to-market
  - Technology
  - Performance
  - Power consumption
  - Volume of production
  - Upgradability / ease of maintenance
  - Reliability
  - Testability
  - Availability of CAD and software tools
  - IP's, hardware and software libraries
  - Cost, chip area
  - Legal and certification requirements, client specifications
  - …..

Digital Circuits : Levels of Abstraction

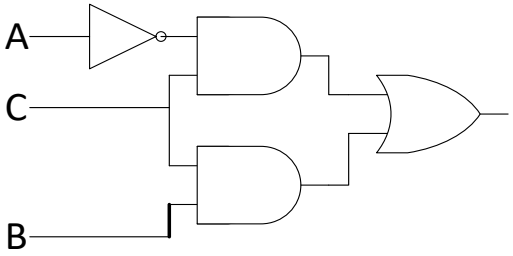


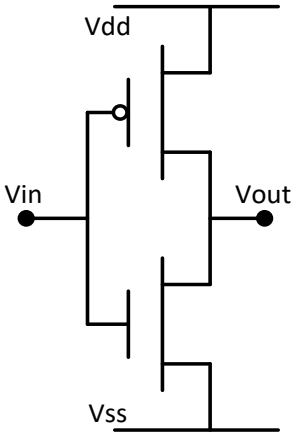
Register
Transfer Level
(RTL, timed)

Algorithm / System
Level (untimed)



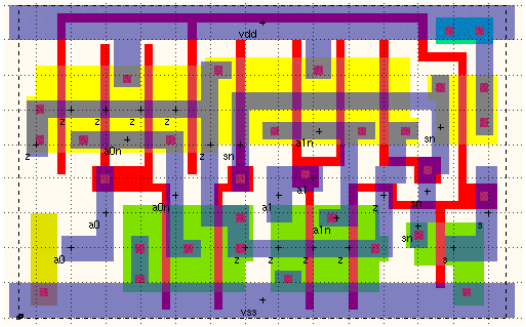
Gate
Level





Circuit
Level

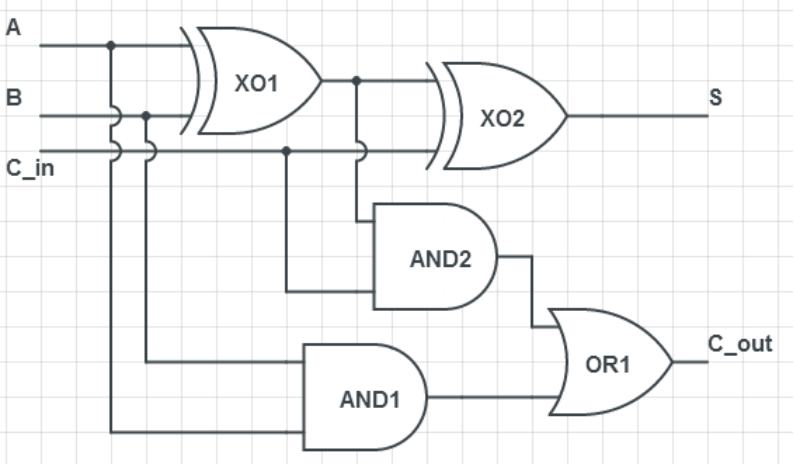
Layout
Level



Why Automation?

- Billions of gates in a year. How else?
  - Turn around time can be minimized
- Target hardware independence
  - The same design can target different hardware platforms with little or no modifications
- Reusability, modularity
  - No need to reinvent the wheel each time
- Allows focusing on higher level issues
  - Lower level optimizations can be done efficiently using electronic design automation (EDA) tools

Schematic Entry



- Pros
  - Easy, intuitive for small projects
  - Instant visualization,
    closer to WYSIWYG philosophy
- Cons
  - Difficult to manage when complexity grows (millions of gates!)
  - Difficult to port projects to different tools and technologies
  - Limited automation – the optimization tool is designer's mind
- Desirable to have a higher level of abstraction
  - Something like a program?

Hardware Description Languages (HDLs)

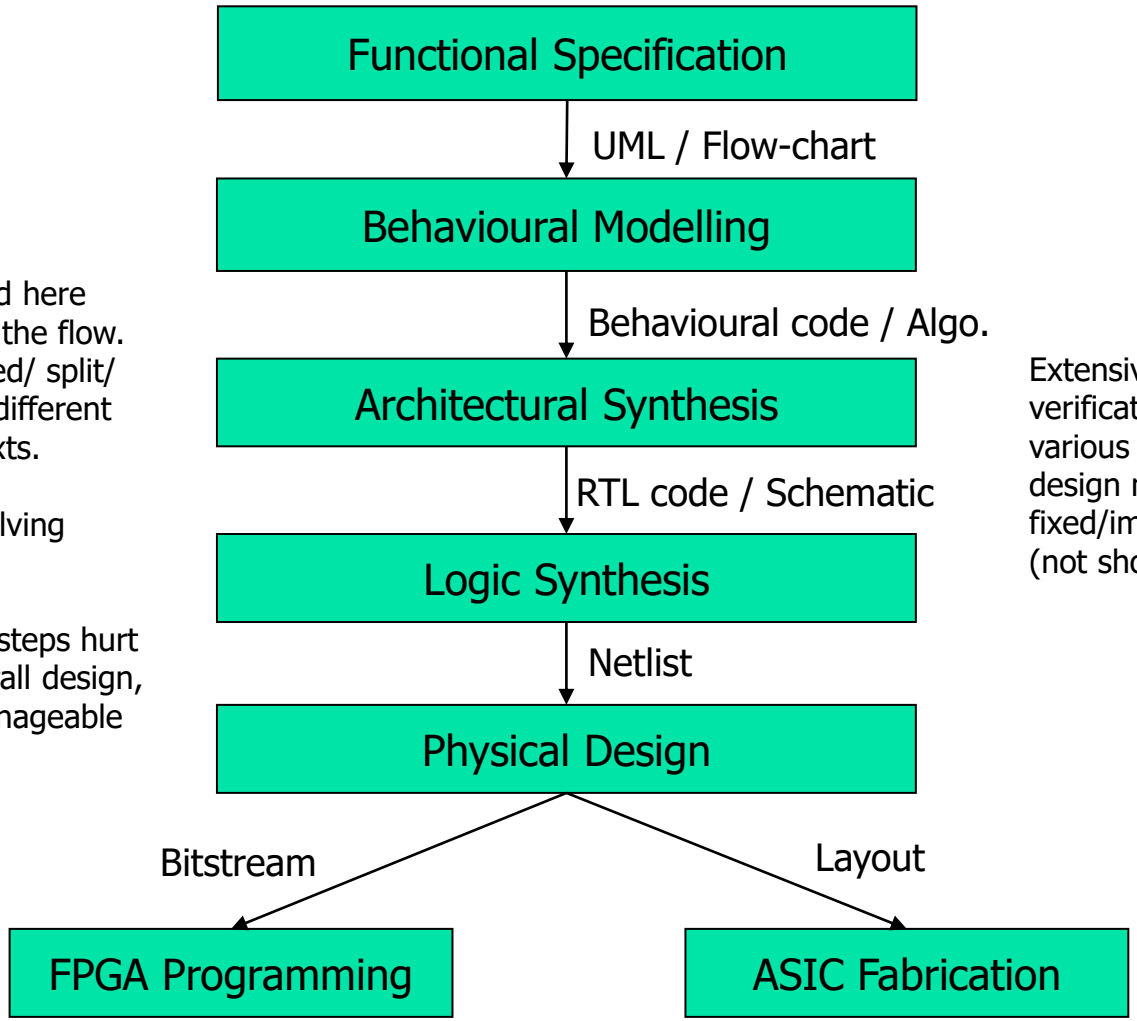
- Programming languages are sequential
  - Operations / instructions executed in a sequential manner - order matters
  - Similar to how humans think
  - No well-defined concept of timing or real concurrency

- HDLs (VHDL, Verilog etc.)
  - Are event based
  - Allow concurrency, timing, delays
  - Facilitate modelling by connection of parts, as with a real circuit
  - Coding, testing, etc. can all be done using the same language
  - Self-documenting code?

Hardware Description Languages (HDLs)

- Verification is as important as design
  - Conventional HDLs are weak in this regard
  - Higher level constructs (classes etc.) and abstractions required to perform verification efficiently
- A number of HDLs have come up, as extensions of popular high-level languages
  - SystemVerilog (Verilog)
  - SystemC (C++)
  - Bluespec (Haskell)
  - Chisel (Scala)
  - MyHDL (Python)
  - SpinalHDL (Scala)
  - CλaSH (Haskell)
- Most of these allow the use of features and libraries of the high-level language for testbench (verification), and generates Verilog/VHDL for eventual hardware synthesis

Simplified FPGA / ASIC Design Flow



The stages mentioned here are only indicative of the flow. Stages may be merged/ split/ named differently in different tools / books / contexts.

Most steps involve solving NP-hard problems

Splitting into various steps hurt optimality of the overall design, but makes things manageable

Extensive validation and verification are done at various stages, and the design may be fixed/improved iteratively. (not shown here)

Behavioural Modelling

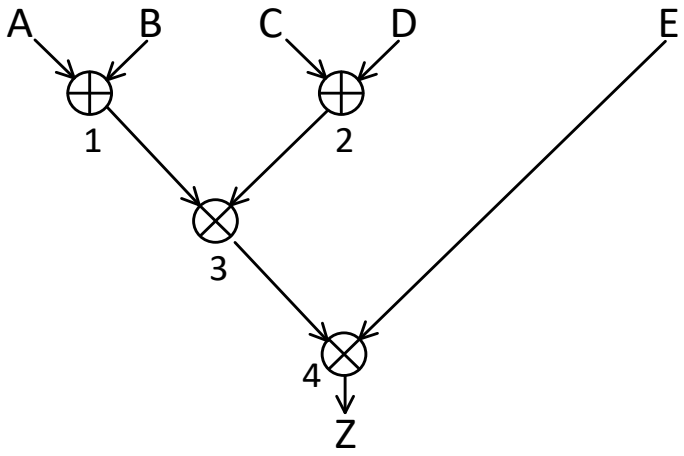
- To ensure algorithmic correctness / functionality
- Preliminary simulations using usual computer programming languages (Python, C, Java, Matlab)
- More elaborate simulations using VHDL, Verilog, SystemVerilog, SystemC etc. which allow concurrency and timing
- No cycle accuracy
- Usually not meant to be synthesized directly
  - There are more and more tools (HLS : high-level synthesis) being developed which can generate RTL code from behavioural code
    - effectiveness highly tool, domain and luck dependent
    - some manual control still needed to get closer to what you want

Architectural Synthesis

- Mostly a manual process (but getting more and more automated)
- Results in a block-level cycle accurate functional specification (RTL code)
- Operand-operations are bound in time (scheduling) and space (binding)
  - Scheduling – specifies which clock cycle is an operand-operation started in / performed at. e.g., A+B is done in, say, cycle #1
  - Binding – specifies which specific macroscopic block (adder #1, multiplier #3, etc.) performs the operation on a specific set/pair of operands. e.g., A+B is done by, say, adder #1
    - Adder #1 can perform the addition of a different pair of operands in another cycle (not cycle #1) provided there are multiplexers at the inputs
- RTL (logic) synthesis tools can infer register transfer operations, and generate a netlist provided some guidelines are followed

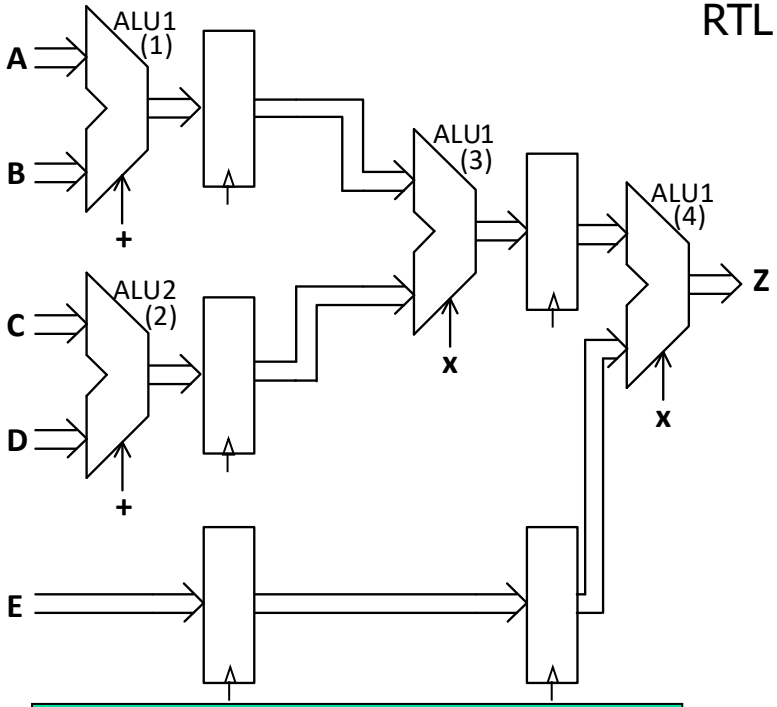
Architectural Synthesis …

Behavioural



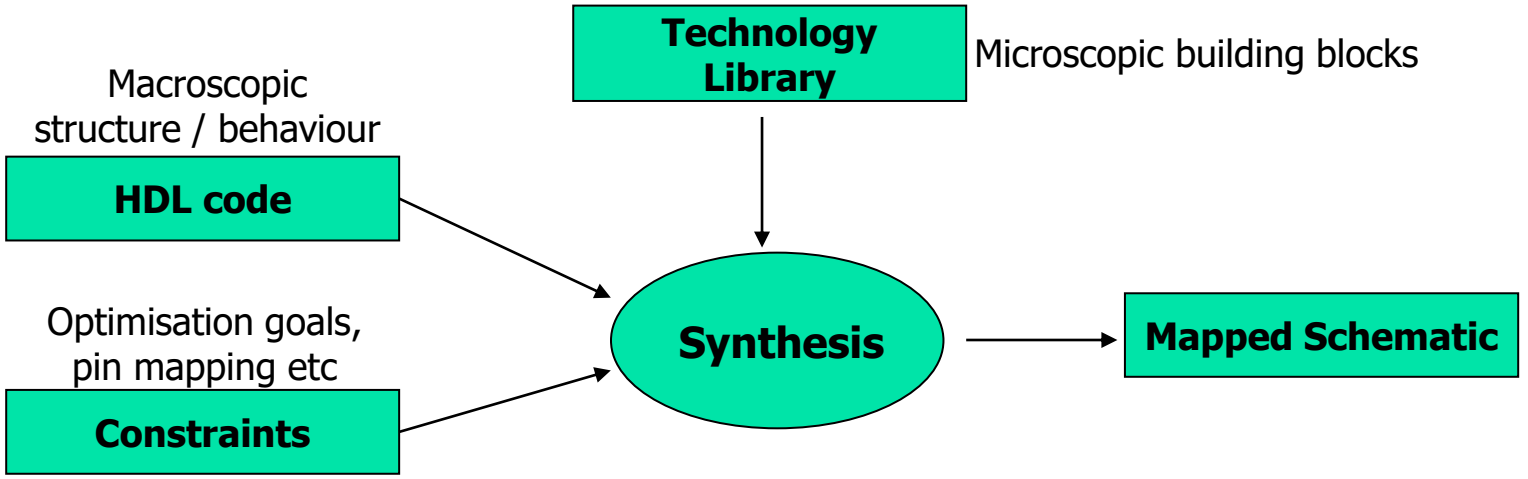

```
tmp1 = A+B;
tmp2 = C+D;
tmp3 = tmp1*tmp2;
Z = tmp3 *E;
```

RTL




```
entity ABCDE is
port(A,B,C : in …
            Z : out … );
end ABCDE;
architecture RTL of ABCDE is
signal …
begin
….
end RTL;
```

Logic Synthesis



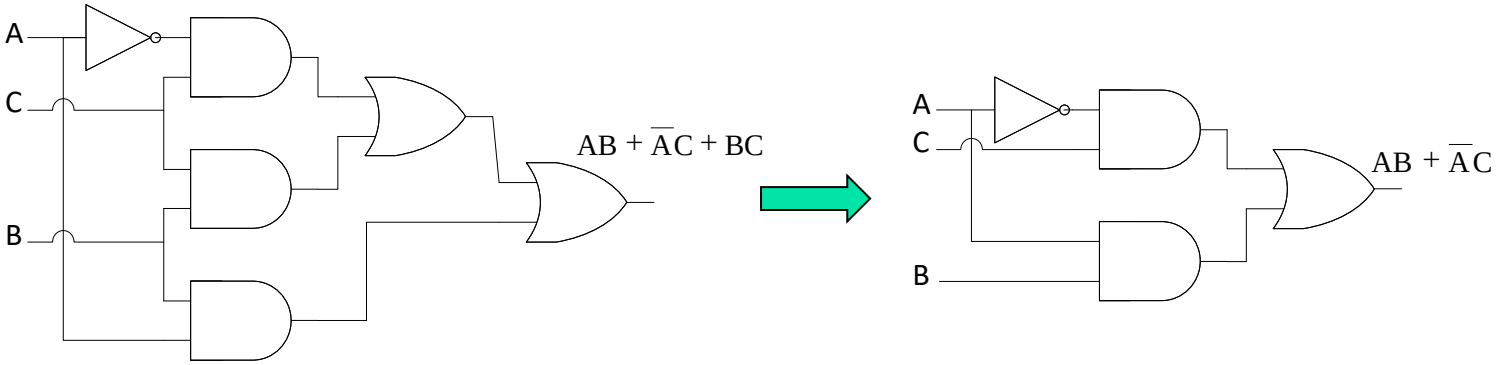
- Technology library
  - The cells / microscopic building blocks we are allowed to use
  - For ASICs, cells are usually gates or gate combinations. They are designed and characterized carefully by the foundry while respecting the physical limitations of the specific process technology
  - For FPGAs, technology library is composed of functions offered by configurable logic blocks (CLBs) – could be higher-level functions such as addition, multiplication, etc.
  - Acts as an abstraction between logical and physical realms

Logic Synthesis …

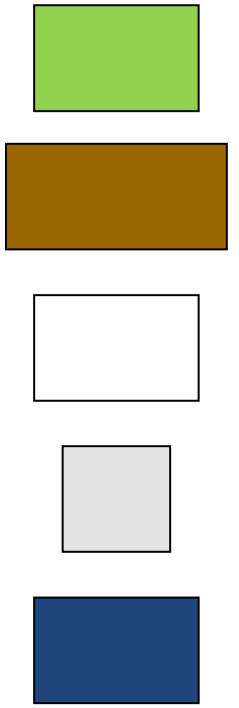
- Mapped Schematic
  - Optimized schematic realizing the HDL code, using building blocks from the technology library
  - Usually a netlist that textually describes the interconnection between cells / blocks
- Constraints
  - Location
    - Logical port to physical pin mapping etc
  - Timing specifications (optimization goals)
    - Different schematics can be obtained from the same HDL code
    - High speed required -> large area, i.e., high cost

Logic Synthesis – Substeps

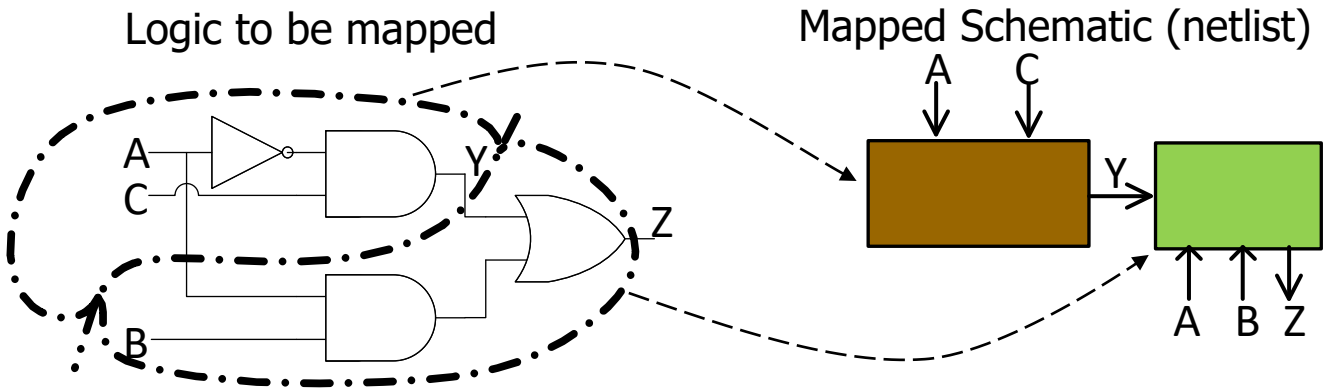
- Logic optimization – to minimize undesirable redundancies (think k-maps), and hence, the cost and complexity of the design
  - Almost entirely automated, relatively mature field and tools



Library Cells

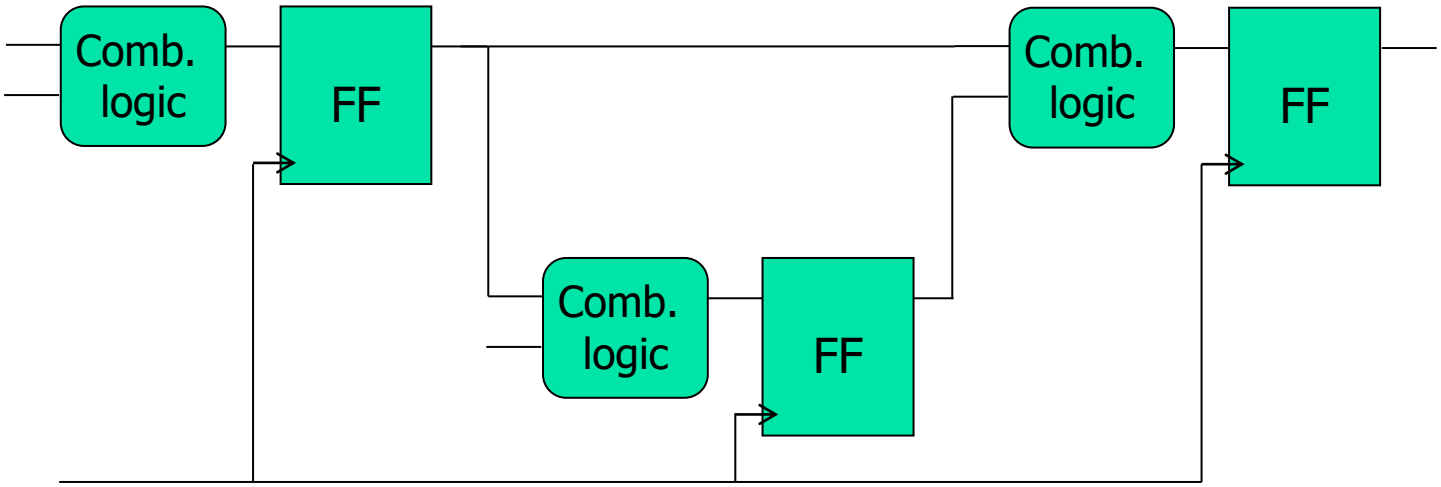


- Technology mapping – mapping logic to cells in the library



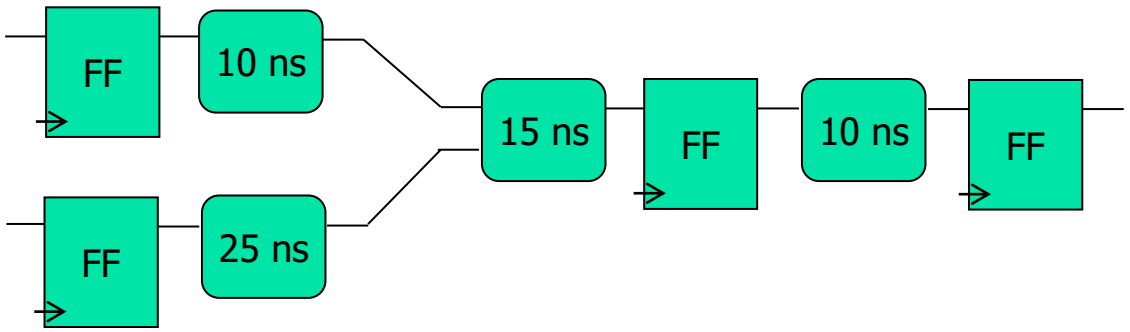
RTL and Registers

- RTL focuses on describing the flow of signals between registers - registers act as brick walls and determine the architecture of the system
- RTL synthesis tools only optimize the combinational logic between the registers (for speed or cost depending on timing constraints)
  - You should be able to write code to infer registers appropriately

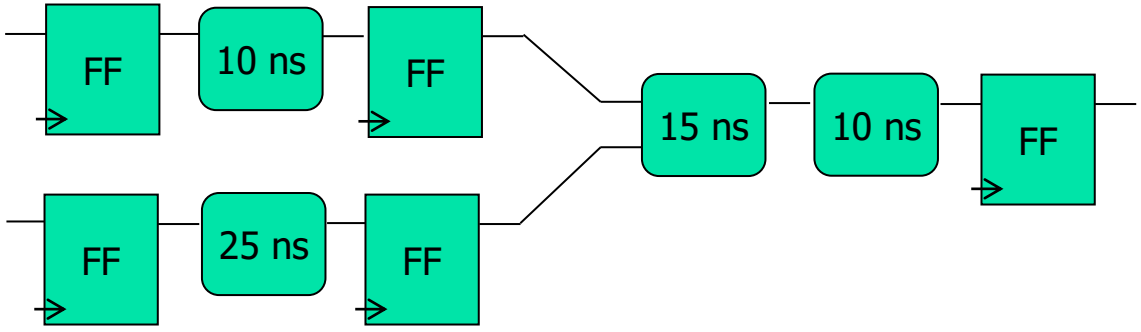


Critical Path

- Critical path = combinational path with maximum delay
- Determines the max. clock
  - assuming FFs have negligible setup time and propagation delay



Max clock
= 1/Critical path delay
= 1/40 ns = 25 MHz

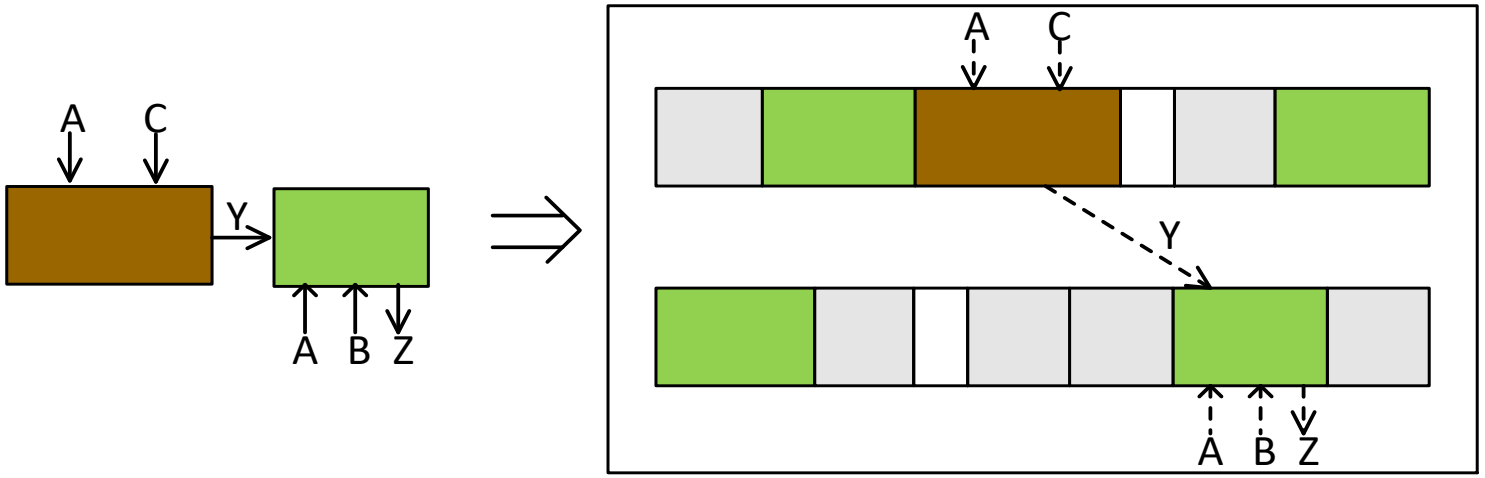


Max clock
= 1/Critical path delay
= 1/25 ns = 40 MHz

- Rearranging register positions to reduce either the critical path delay *or* the register count is called retiming. Some tools can do this for you!

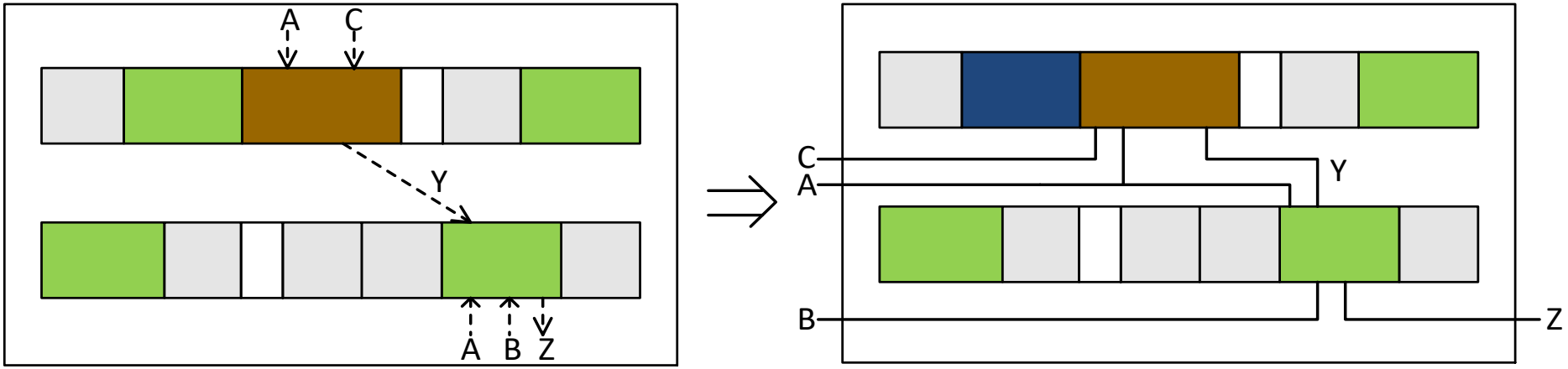
Physical Design

- Placement - mapping the shape of a cell to a physical place in the layout
- Place cells which having more interconnections closer



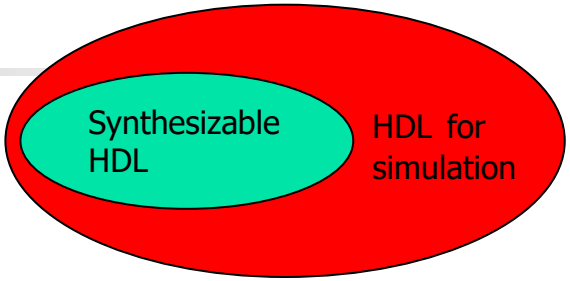
Physical Design …

- Routing – mapping of logical connection between cells to physical interconnects
- Shorter routes minimize delays



More in EE4218 Embedded Hardware System Design !!

Synthesizable HDL



- Not all HDL code can be synthesized
- Synthesizable HDL is a subset of HDL
- The exact subset that can be synthesized varies with synthesis tool
  - RTL-level synthesis tools: restrictive
  - Behavioural (high level) synthesis tools: larger subset
- Simulation and synthesis tools work very differently
  - Simulation 'executes' HDL code following the semantics of the HDL
  - Synthesis 'infers the hardware structure' described by HDL (RTL) code
  - The behaviour of the tools do not always match
- If you understand how each tool works, bear in mind that you are describing hardware and not writing a conventional 'program' (C-style code), and follow guidelines provided in the second part of this chapter, you will get the same expected behaviour from both
  - Have the topology of the circuit in mind, and write RTL code such that the tool can infer the topology from the code

If your HDL code is not synthesizable..

- Synthesis tool may not be able to create hardware
- It may or may not give warning / error message(s),
  - The messages generated may not be directly indicative of the actual problem
- It may try and give you a result which is 'not quite right'
  - The resultant hardware could have a behaviour different from that of your expectation / simulation
- Subsequent steps (technology mapping, place and route etc.) may not run into completion
  - The logic inferred by the synthesis tool could be impractically big for the target hardware and/or for the subsequent tools to handle

To Do: Refresh your memory on blocking/non-blocking assignments
To Do: Go through http://www.sunburst-design.com/papers/CummingsSNUG2000SJ_NBA_rev1_2.pdf, which is considered a classic reference on Verilog synthesis, though I don't entirely agree with some points there

HDL Coding Tips

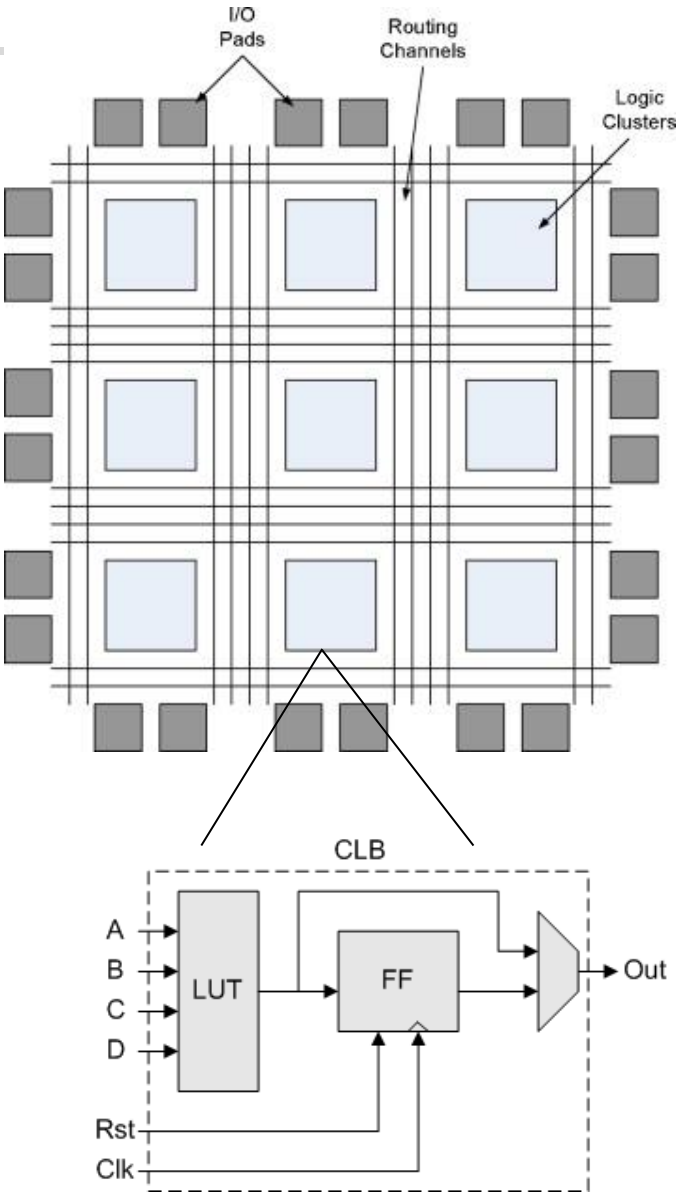
- Modularize your system into entities / modules of appropriate size, with well-defined interfaces
  - Bigger modules / entities are harder to test / debug
  - Smaller modules / entities could make your code unnecessarily long and might also limit the amount of optimization done by the synthesis tool
  - Modules / entities (components) should have logical flow of data and minimal interconnections between them, and should have at most one clock input
  - Use descriptive labels (names) for modules / entities / signals / variables / wires / regs
- Separate combinational and sequential processes / always blocks wherever possible
  - Combinational blocks and lean/simple sequential process / always blocks (output <= input at the ACT) are relatively easy to test
  - Complex sequential blocks (state machines) are very hard to test

HDL Coding Tips

- Simulate each entity/ module thoroughly using well-designed test-cases, at every level of hierarchy
- Synthesize each entity / module by setting it as the top-level module and check for errors / warnings. Check the synthesis report to see if the inferred primitives (memories, adders, multipliers, FFs etc.) are as expected
- Most warnings are NOT safe to ignore
  - Do not modify the code just so that the warning will go away. Analyze the warning very carefully and identify the exact reason
- For best results, use templates from the synthesis manual of the EDA tool you are using
- Do a post-synthesis functional simulation and if possible, even more advanced simulations (you need not create new testbenches for these!)
- Testing on hardware should be the last step, after thorough simulations and being able to synthesize without avoidable warnings
- Indent your code. Comment liberally. Can't read = can't debug

FPGAs

- Matrix of configurable logic blocks (CLBs) connected via programmable interconnects (switches)
- Usually have several blocks of high-speed static/ dynamic RAMs
- Some FPGAs have embedded processors, multiplier units etc
- Low-medium performance and density
- Cost effective for low volumes, very costly compared to ASICs for large volumes
- Low development cost and time



FPGA : Highlights

- Lego of the digital world
- Scalable, fast deployable, can adapt to emerging /and changing standards
- Apply bug fixes by just downloading a new configuration file to the device
- Standalone embedded systems, or in tandem with conventional micro-processors/controllers
  - Custom instructions
  - Custom accelerators/co-processors
  - Application specific optimizations (highly-tailored solutions)
- Massive parallelism – high performance computing, eg: in bioinformatics, cryptanalysis, high-frequency trading, machine learning, big-data, …
- Partial reconfiguration, fault tolerance

FPGA : Disadvantages

- Power
  - 14 times ASIC's power
- Area/Cost
  - 35 times larger than ASIC
- Speed
  - 3-4 times slower than ASIC
- More suited for low volume products / prototyping / verification / emulation
- Needs hardware engineers!
  - Becoming less the case

Ian Kuon, Rose, J., "Measuring the Gap Between FPGAs and ASICs", IEEE Transactions on CAD of Integrated Circuits and Systems, vol. 26, no. 2, 2007

Summary

- Hardware design flow – architectural, logic, and physical synthesis
  - EDA tools can help designers to get layout from a proper system specification using a chain of commercial synthesis tools
- Synthesizable HDL is a subset of HDL
  - For best results, use templates from the user guide of the EDA tool you are using
  - DO NOT ignore warnings
- Think Hardware - have the circuit in mind and write HDL code to describe it
  - Important to know when registers are inferred
- Testing on hardware should be the LAST step
- FPGAs are highly configurable hardware well-suited for prototyping