

Embedded Systems Design: A Unified Hardware/Software Introduction

Chapter 11: Design Technology



Outline

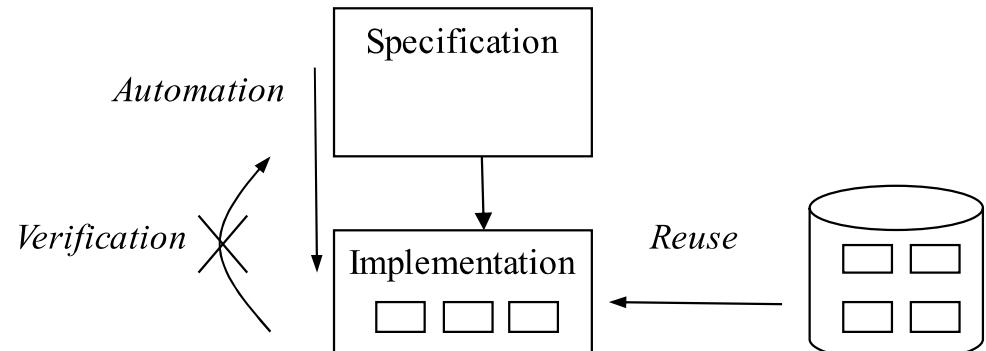
- Automation: synthesis
- Verification: hardware/software co-simulation
- Reuse: intellectual property cores
- Design process models

Introduction

- Design task
 - Define system functionality
 - Convert functionality to physical implementation while
 - Satisfying constrained metrics
 - Optimizing other design metrics
- Designing embedded systems is hard
 - Complex functionality
 - Millions of possible environment scenarios
 - Competing, tightly constrained metrics
 - Productivity gap
 - As low as 10 lines of code or 100 transistors produced per day

Improving productivity

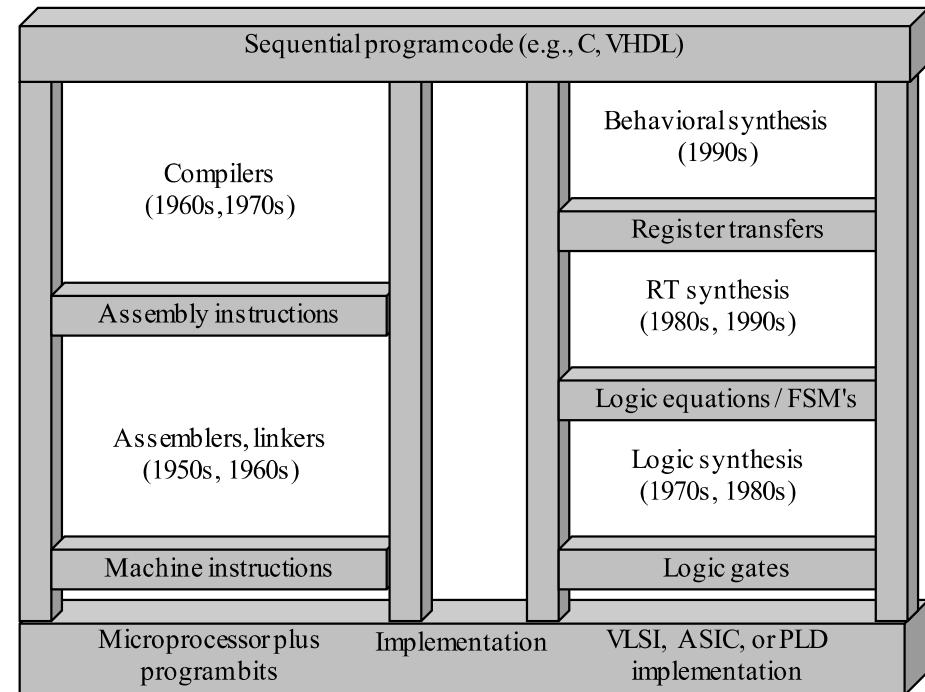
- Design technologies developed to improve productivity
- We focus on technologies advancing hardware/software unified view
 - Automation
 - Program replaces manual design
 - Synthesis
 - Reuse
 - Predesigned components
 - Cores
 - General-purpose and single-purpose processors on single IC
 - Verification
 - Ensuring correctness/completeness of each design step
 - Hardware/software co-simulation



Automation: synthesis

- Early design mostly hardware
- Software complexity increased with advent of general-purpose processor
- Different techniques for software design and hardware design
 - Caused division of the two fields
- Design tools evolve for higher levels of abstraction
 - Different rate in each field
- Hardware/software design fields rejoicing
 - Both can start from behavioral description in sequential program model
 - 30 years longer for hardware design to reach this step in the ladder
 - Many more design dimensions
 - Optimization critical

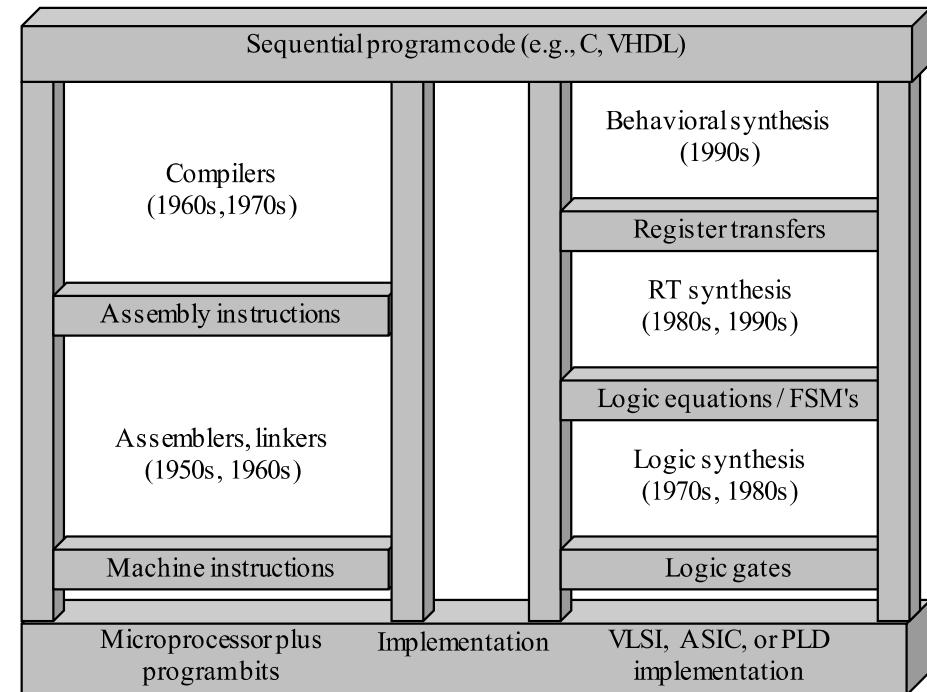
The codesign ladder



Hardware/software parallel evolution

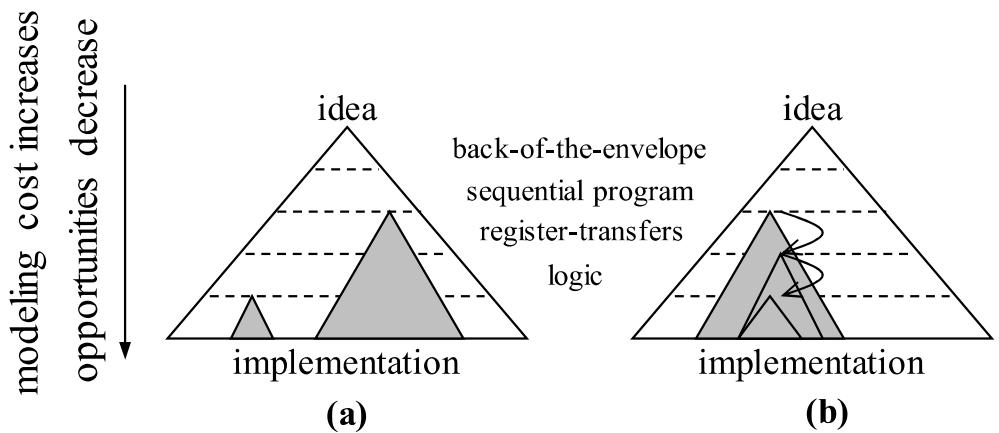
- Software design evolution
 - Machine instructions
 - Assemblers
 - convert assembly programs into machine instructions
 - Compilers
 - translate sequential programs into assembly
- Hardware design evolution
 - Interconnected logic gates
 - Logic synthesis
 - converts logic equations or FSMs into gates
 - Register-transfer (RT) synthesis
 - converts FSMDs into FSMs, logic equations, predesigned RT components (registers, adders, etc.)
 - Behavioral synthesis
 - converts sequential programs into FSMDs

The codesign ladder



Increasing abstraction level

- Higher abstraction level focus of hardware/software design evolution
 - Description smaller/easier to capture
 - E.g., Line of sequential program code can translate to 1000 gates
 - Many more possible implementations available
 - (a) Like flashlight, the higher above the ground, the more ground illuminated
 - Sequential program designs may differ in performance/transistor count by orders of magnitude
 - Logic-level designs may differ by only power of 2
 - (b) Design process proceeds to lower abstraction level, narrowing in on single implementation

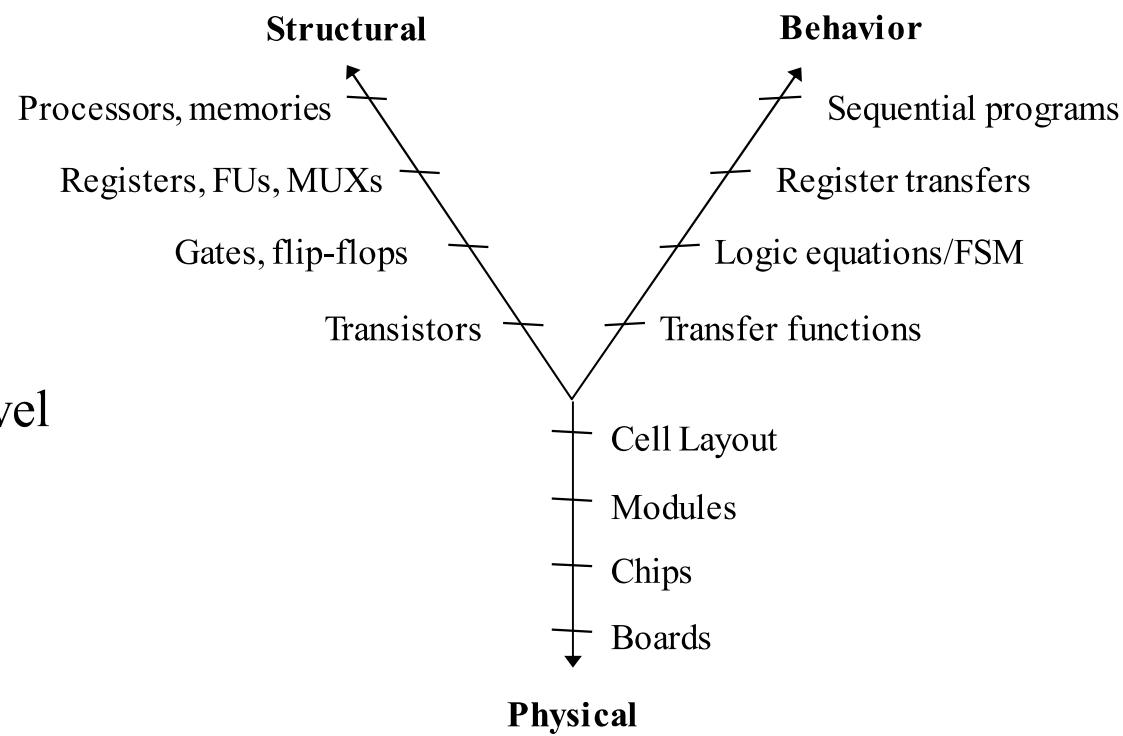


Synthesis

- Automatically converting system's behavioral description to a structural implementation
 - Complex whole formed by parts
 - Structural implementation must optimize design metrics
- More expensive, complex than compilers
 - Cost = \$100s to \$10,000s
 - User controls 100s of synthesis options
 - Optimization critical
 - Otherwise could use software
 - Optimizations different for each user
 - Run time = hours, days

Gajski's Y-chart

- Each axis represents type of description
 - Behavioral
 - Defines outputs as function of inputs
 - Algorithms but no implementation
 - Structural
 - Implements behavior by connecting components with known behavior
 - Physical
 - Gives size/locations of components and wires on chip/board
- Synthesis converts behavior at given level to structure at same level or lower
 - E.g.,
 - FSM → gates, flip-flops (same level)
 - FSM → transistors (lower level)
 - FSM X registers, FUs (higher level)
 - FSM X processors, memories (higher level)



Logic synthesis

- Logic-level behavior to structural implementation
 - Logic equations and/or FSM to connected gates
- Combinational logic synthesis
 - Two-level minimization (Sum of products/product of sums)
 - Best possible performance
 - Longest path = 2 gates (AND gate + OR gate/OR gate + AND gate)
 - Minimize size
 - Minimum cover
 - Minimum cover that is prime
 - Heuristics
 - Multilevel minimization
 - Trade performance for size
 - Pareto-optimal solution
 - Heuristics
- FSM synthesis
 - State minimization
 - State encoding

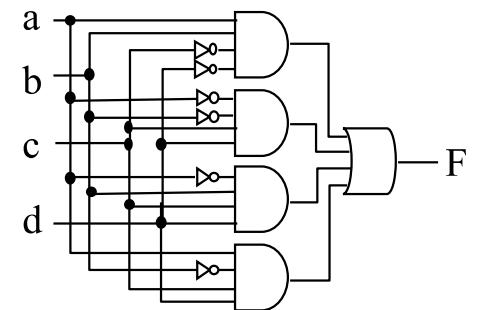
Two-level minimization

- Represent logic function as sum of products (or product of sums)
 - AND gate for each product
 - OR gate for each sum
- Gives best possible performance
 - At most 2 gate delay
- Goal: minimize size
 - Minimum cover
 - Minimum # of AND gates (sum of products)
 - Minimum cover that is prime
 - Minimum # of inputs to each AND gate (sum of products)

Sum of products

$$F = abc'd' + a'b'cd + a'bcd + ab'cd$$

Direct implementation



4 4-input AND gates and
1 4-input OR gate
→ 40 transistors

Minimum cover

- Minimum # of AND gates (sum of products)
 - **Literal:** variable or its complement
 - a or a' , b or b' , etc.
 - **Minterm:** product of literals
 - Each literal appears exactly once
 - $abc'd'$, $ab'cd$, $a'bcd$, etc.
 - **Implicant:** product of literals
 - Each literal appears no more than once
 - $abc'd'$, $a'cd$, etc.
 - Covers 1 or more minterms
 - $a'cd$ covers $a'bcd$ and $a'b'cd$
 - **Cover:** set of implicants that covers all minterms of function
 - **Minimum cover:** cover with minimum # of implicants
-

Minimum cover: K-map approach

- Karnaugh map (K-map)
 - 1 represents minterm
 - Circle represents implicant
- Minimum cover
 - Covering all 1's with min # of circles
 - Example: direct vs. min cover
 - Less gates
 - 4 vs. 5
 - Less transistors
 - 28 vs. 40

K-map: sum of products

| ab \ cd | 00 | 01 | 11 | 10 |
|---------|----|----|----|----|
| 00 | 0 | 0 | 1 | 0 |
| 01 | 0 | 0 | 1 | 0 |
| 11 | 1 | 0 | 0 | 0 |
| 10 | 0 | 0 | 1 | 0 |

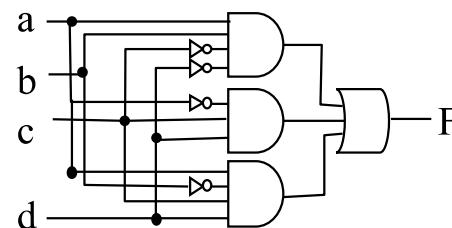
K-map: minimum cover

| ab \ cd | 00 | 01 | 11 | 10 |
|---------|----|----|----|----|
| 00 | 0 | 0 | 1 | 0 |
| 01 | 0 | 0 | 1 | 0 |
| 11 | 1 | 0 | 0 | 0 |
| 10 | 0 | 0 | 1 | 0 |

Minimum cover

$$F = abc'd' + a'cd + ab'cd$$

Minimum cover implementation



2 4-input AND gate
1 3-input AND gates
1 4 input OR gate
→ 28 transistors

Minimum cover that is prime

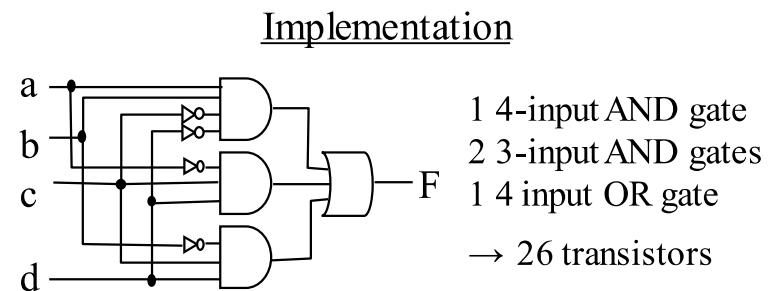
- Minimum # of inputs to AND gates
- Prime implicant
 - Implicant not covered by any other implicant
 - Max-sized circle in K-map
- Minimum cover that is prime
 - Covering with min # of prime implicants
 - Min # of max-sized circles
 - Example: prime cover vs. min cover
 - Same # of gates
 - 4 vs. 4
 - Less transistors
 - 26 vs. 28

K-map: minimum cover that is prime

| ab \ cd | 00 | 01 | 11 | 10 |
|---------|-----|----|-----|----|
| 00 | 0 | 0 | (1) | 0 |
| 01 | 0 | 0 | 1 | 0 |
| 11 | (1) | 0 | 0 | 0 |
| 10 | 0 | 0 | (1) | 0 |

Minimum cover that is prime

$$F = abc'd' + a'cd + b'cd$$



Minimum cover: heuristics

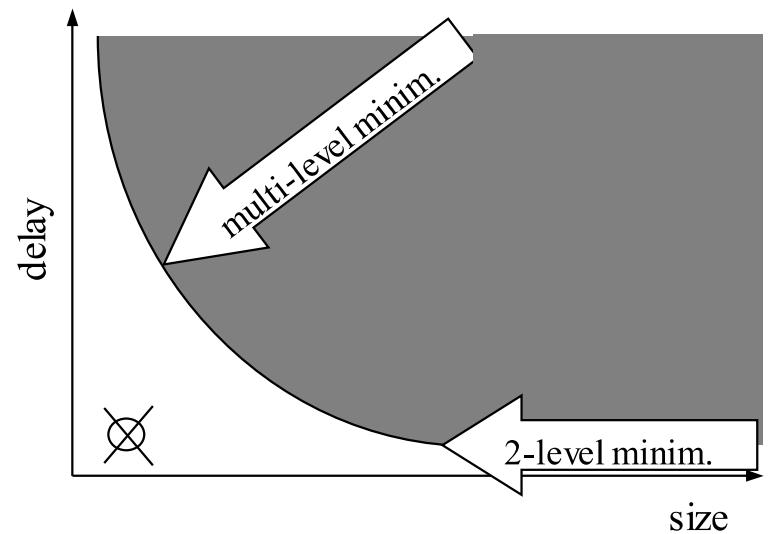
- K-maps give optimal solution every time
 - Functions with > 6 inputs too complicated
 - Use computer-based tabular method
 - Finds all prime implicants
 - Finds min cover that is prime
 - Also optimal solution every time
 - Problem: 2^n minterms for n inputs
 - 32 inputs = 4 billion minterms
 - Exponential complexity
- Heuristic
 - Solution technique where optimal solution not guaranteed
 - Hopefully comes close

Heuristics: iterative improvement

- Start with initial solution
 - i.e., original logic equation
 - Repeatedly make modifications toward better solution
 - Common modifications
 - Expand
 - Replace each nonprime implicant with a prime implicant covering it
 - Delete all implicants covered by new prime implicant
 - Reduce
 - Opposite of expand
 - Reshape
 - Expands one implicant while reducing another
 - Maintains total # of implicants
 - Irredundant
 - Selects min # of implicants that cover from existing implicants
 - Synthesis tools differ in modifications used and the order they are used
-

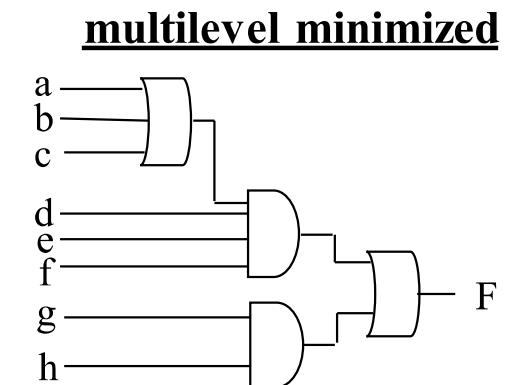
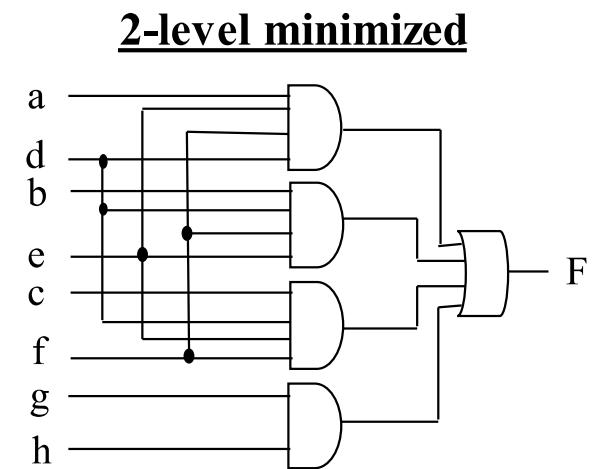
Multilevel logic minimization

- Trade performance for size
 - Increase delay for lower # of gates
 - Gray area represents all possible solutions
 - Circle with X represents ideal solution
 - Generally not possible
 - 2-level gives best performance
 - max delay = 2 gates
 - Solve for smallest size
 - Multilevel gives pareto-optimal solution
 - Minimum delay for a given size
 - Minimum size for a given delay



Example

- Minimized 2-level logic function:
 - $F = adef + bdef + cdef + gh$
 - Requires 5 gates with 18 total gate inputs
 - 4 ANDS and 1 OR
- After algebraic manipulation:
 - $F = (a + b + c)def + gh$
 - Requires only 4 gates with 11 total gate inputs
 - 2 ANDS and 2 ORs
 - Less inputs per gate
 - Assume gate inputs = 2 transistors
 - Reduced by 14 transistors
 - 36 ($18 * 2$) down to 22 ($11 * 2$)
 - Sacrifices performance for size
 - Inputs a, b, and c now have 3-gate delay
- Iterative improvement heuristic commonly used



FSM synthesis

- FSM to gates
- State minimization
 - Reduce # of states
 - Identify and merge equivalent states
 - Outputs, next states same for all possible inputs
 - Tabular method gives exact solution
 - Table of all possible state pairs
 - If n states, n^2 table entries
 - Thus, heuristics used with large # of states
 - State encoding
 - Unique bit sequence for each state
 - If n states, $\log_2(n)$ bits
 - $n!$ possible encodings
 - Thus, heuristics common

Technology mapping

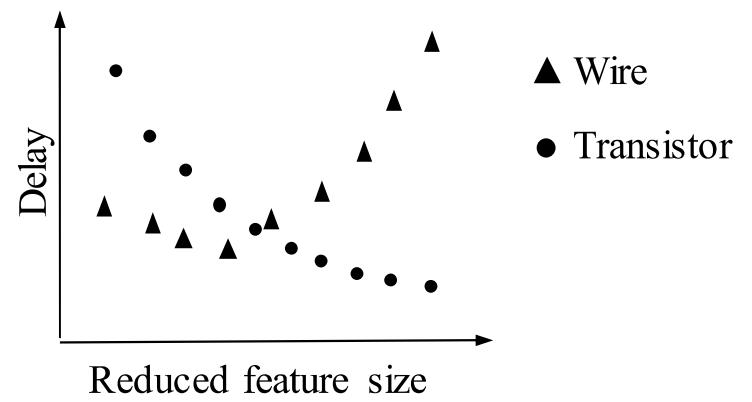
- Library of gates available for implementation
 - Simple
 - only 2-input AND,OR gates
 - Complex
 - various-input AND,OR,NAND,NOR,etc. gates
 - Efficiently implemented meta-gates (i.e., AND-OR-INVERT,MUX)
- Final structure consists of specified library's components only
- If technology mapping integrated with logic synthesis
 - More efficient circuit
 - More complex problem
 - Heuristics required

Complexity impact on user

- As complexity grows, heuristics used
- Heuristics differ tremendously among synthesis tools
 - Computationally expensive
 - Higher quality results
 - Variable optimization effort settings
 - Long run times (hours, days)
 - Requires huge amounts of memory
 - Typically needs to run on servers, workstations
 - Fast heuristics
 - Lower quality results
 - Shorter run times (minutes, hours)
 - Smaller amount of memory required
 - Could run on PC
- Super-linear-time (i.e. n^3) heuristics usually used
 - User can partition large systems to reduce run times/size
 - $100^3 > 50^3 + 50^3$ ($1,000,000 > 250,000$)

Integrating logic design and physical design

- Past
 - Gate delay much greater than wire delay
 - Thus, performance evaluated as # of levels of gates only
- Today
 - Gate delay shrinking as feature size shrinking
 - Wire delay increasing
 - Performance evaluation needs wire length
 - Transistor placement (needed for wire length) domain of physical design
 - Thus, simultaneous logic synthesis and physical design required for efficient circuits



Register-transfer synthesis

- Converts FSMD to custom single-purpose processor
 - Datapath
 - Register units to store variables
 - Complex data types
 - Functional units
 - Arithmetic operations
 - Connection units
 - Buses, MUXs
 - FSM controller
 - Controls datapath
 - Key sub problems:
 - Allocation
 - Instantiate storage, functional, connection units
 - Binding
 - Mapping FSMD operations to specific units

Behavioral synthesis

- High-level synthesis
- Converts single sequential program to single-purpose processor
 - Does not require the program to schedule states
- Key sub problems
 - Allocation
 - Binding
 - Scheduling
 - Assign sequential program's operations to states
 - Conversion template given in Ch. 2
- Optimizations important
 - Compiler
 - Constant propagation, dead-code elimination, loop unrolling
 - Advanced techniques for allocation, binding, scheduling

System synthesis

- Convert 1 or more processes into 1 or more processors (system)
 - For complex embedded systems
 - Multiple processes may provide better performance/power
 - May be better described using concurrent sequential programs
- Tasks
 - Transformation
 - Can merge 2 exclusive processes into 1 process
 - Can break 1 large process into separate processes
 - Procedure inlining
 - Loop unrolling
 - Allocation
 - Essentially design of system architecture
 - Select processors to implement processes
 - Also select memories and busses

System synthesis

- Tasks (cont.)
 - Partitioning
 - Mapping 1 or more processes to 1 or more processors
 - Variables among memories
 - Communications among buses
 - Scheduling
 - Multiple processes on a single processor
 - Memory accesses
 - Bus communications
 - Tasks performed in variety of orders
 - Iteration among tasks common

System synthesis

- Synthesis driven by constraints
 - E.g.,
 - Meet performance requirements at minimum cost
 - Allocate as much behavior as possible to general-purpose processor
 - Low-cost/flexible implementation
 - Minimum # of SPPs used to meet performance
- System synthesis for GPP only (software)
 - Common for decades
 - Multiprocessing
 - Parallel processing
 - Real-time scheduling
- Hardware/software codesign
 - Simultaneous consideration of GPPs/SPPs during synthesis
 - Made possible by maturation of behavioral synthesis in 1990's

Temporal vs. spatial thinking

- Design thought process changed by evolution of synthesis
- Before synthesis
 - Designers worked primarily in structural domain
 - Connecting simpler components to build more complex systems
 - Connecting logic gates to build controller
 - Connecting registers, MUXs, ALUs to build datapath
 - “capture and simulate” era
 - Capture using CAD tools
 - Simulate to verify correctness before fabricating
 - Spatial thinking
 - Structural diagrams
 - Data sheets

Temporal vs. spatial thinking

- After synthesis
 - “describe-and-synthesize” era
 - Designers work primarily in behavioral domain
 - “describe and synthesize” era
 - Describe FSMDs or sequential programs
 - Synthesize into structure
 - Temporal thinking
 - States or sequential statements have relationship over time
- Strong understanding of hardware structure still important
 - Behavioral description must synthesize to efficient structural implementation

Verification

- Ensuring design is correct and complete
 - Correct
 - Implements specification accurately
 - Complete
 - Describes appropriate output to all relevant input
- Formal verification
 - Hard
 - For small designs or verifying certain key properties only
- Simulation
 - Most common verification method

Formal verification

- Analyze design to prove or disprove certain properties
- Correctness example
 - Prove ALU structural implementation equivalent to behavioral description
 - Derive Boolean equations for outputs
 - Create truth table for equations
 - Compare to truth table from original behavior
- Completeness example
 - Formally prove elevator door can never open while elevator is moving
 - Derive conditions for door being open
 - Show conditions conflict with conditions for elevator moving

Simulation

- Create computer model of design
 - Provide sample input
 - Check for acceptable output
- Correctness example
 - ALU
 - Provide all possible input combinations
 - Check outputs for correct results
- Completeness example
 - Elevator door closed when moving
 - Provide all possible input sequences
 - Check door always closed when elevator moving

Increases confidence

- Simulating all possible input sequences impossible for most systems
 - E.g., 32-bit ALU
 - $2^{32} * 2^{32} = 2^{64}$ possible input combinations
 - At 1 million combinations/sec
 - $\frac{1}{2}$ million years to simulate
 - Sequential circuits even worse
- Can only simulate tiny subset of possible inputs
 - Typical values
 - Known boundary conditions
 - E.g., 32-bit ALU
 - Both operands all 0's
 - Both operands all 1's
- Increases confidence of correctness/completeness
- Does not prove

Advantages over physical implementation

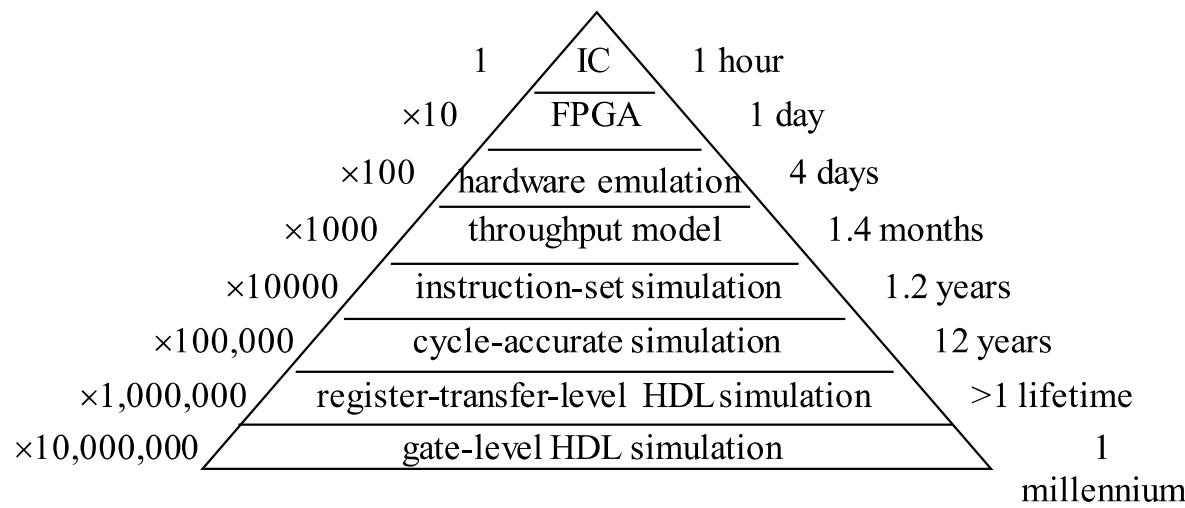
- Controllability
 - Control time
 - Stop/start simulation at any time
 - Control data values
 - Inputs or internal values
- Observability
 - Examine system/environment values at any time
- Debugging
 - Can stop simulation at any point and:
 - Observe internal values
 - Modify system/environment values before restarting
 - Can step through small intervals (i.e., 500 nanoseconds)

Disadvantages

- Simulation setup time
 - Often has complex external environments
 - Could spend more time modeling environment than system
- Models likely incomplete
 - Some environment behavior undocumented if complex environment
 - May not model behavior correctly
- Simulation speed much slower than actual execution
 - Sequentializing parallel design
 - IC: gates operate in parallel
 - Simulation: analyze inputs, generate outputs for each gate 1 at time
 - Several programs added between simulated system and real hardware
 - 1 simulated operation:
 - = 10 to 100 simulator operations
 - = 100 to 10,000 operating system operations
 - = 1,000 to 100,000 hardware operations

Simulation speed

- Relative speeds of different types of simulation/emulation
 - 1 hour actual execution of SOC
 - = 1.2 years instruction-set simulation
 - = 10,000,000 hours gate-level simulation



Overcoming long simulation time

- Reduce amount of real time simulated
 - 1 msec execution instead of 1 hour
 - $0.001\text{sec} * 10,000,000 = 10,000 \text{ sec} = 3 \text{ hours}$
 - Reduced confidence
 - 1 msec of cruise controller operation tells us little
- Faster simulator
 - Emulators
 - Special hardware for simulations
 - Less precise/accurate simulators
 - Exchange speed for observability/controllability

Reducing precision/accuracy

- Don't need gate-level analysis for all simulations
 - E.g., cruise control
 - Don't care what happens at every input/output of each logic gate
 - Simulating RT components ~10x faster
 - Cycle-based simulation ~100x faster
 - Accurate at clock boundaries only
 - No information on signal changes between boundaries
- Faster simulator often combined with reduction in real time
 - If willing to simulate for 10 hours
 - Use instruction-set simulator
 - Real execution time simulated
 - $10 \text{ hours} * 1 / 10,000$
 - = 0.001 hour
 - = 3.6 seconds

Hardware/software co-simulation

- Variety of simulation approaches exist
 - From very detailed
 - E.g., gate-level model
 - To very abstract
 - E.g., instruction-level model
- Simulation tools evolved separately for hardware/software
 - Recall separate design evolution
 - Software (GPP)
 - Typically with instruction-set simulator (ISS)
 - Hardware (SPP)
 - Typically with models in HDL environment
- Integration of GPP/SPP on single IC creating need for merging simulation tools

Integrating GPP/SPP simulations

- Simple/naïve way
 - HDL model of microprocessor
 - Runs system software
 - Much slower than ISS
 - Less observable/controllable than ISS
 - HDL models of SPPs
 - Integrate all models
- Hardware-software co-simulator
 - ISS for microprocessor
 - HDL model for SPPs
 - Create communication between simulators
 - Simulators run separately except when transferring data
 - Faster
 - Though, frequent communication between ISS and HDL model slows it down

Minimizing communication

- Memory shared between GPP and SPPs
 - Where should memory go?
 - In ISS
 - HDL simulator must stall for memory access
 - In HDL?
 - ISS must stall when fetching each instruction
- Model memory in both ISS and HDL
 - Most accesses by each model unrelated to other's accesses
 - No need to communicate these between models
 - Co-simulator ensures consistency of shared data
 - Huge speedups (100x or more) reported with this technique

Emulators

- General physical device system mapped to
 - Microprocessor emulator
 - Microprocessor IC with some monitoring, control circuitry
 - SPP emulator
 - FPGAs (10s to 100s)
 - Usually supports debugging tasks
- Created to help solve simulation disadvantages
 - Mapped relatively quickly
 - Hours, days
 - Can be placed in real environment
 - No environment setup time
 - No incomplete environment
 - Typically faster than simulation
 - Hardware implementation

Disadvantages

- Still not as fast as real implementations
 - E.g., emulated cruise-control may not respond fast enough to keep control of car
- Mapping still time consuming
 - E.g., mapping complex SOC to 10 FPGAs
 - Just partitioning into 10 parts could take weeks
- Can be very expensive
 - Top-of-the-line FPGA-based emulator: \$100,000 to \$1 mill
 - Leads to resource bottleneck
 - Can maybe only afford 1 emulator
 - Groups wait days, weeks for other group to finish using

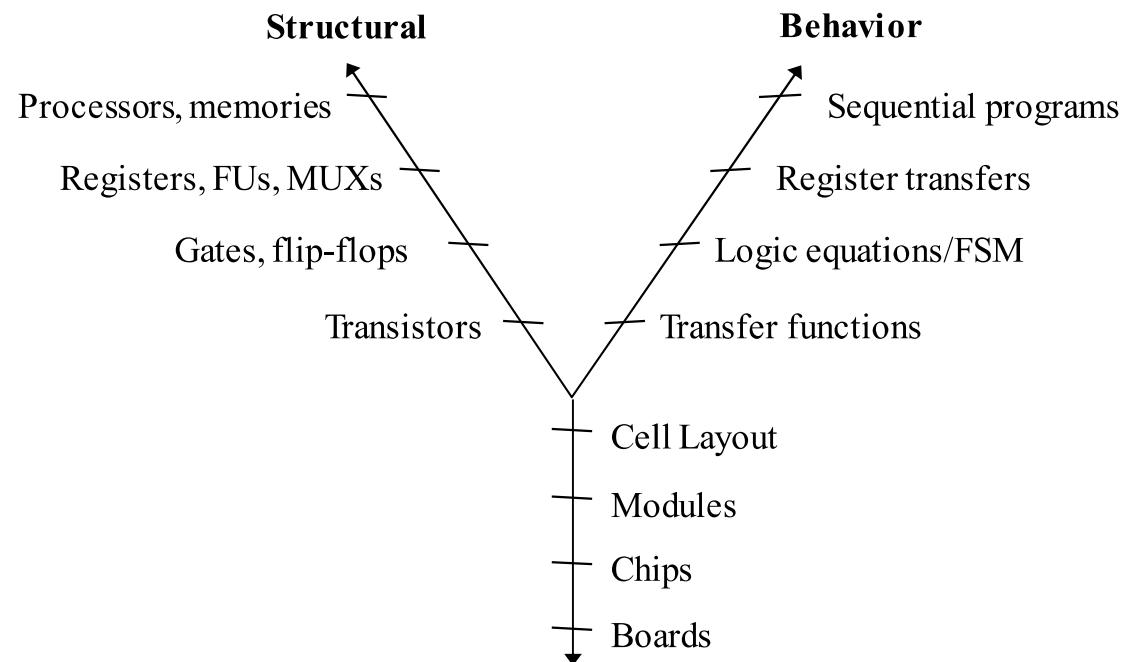
Reuse: intellectual property cores

- Commercial off-the-shelf (COTS) components
 - Predesigned, prepackaged ICs
 - Implements GPP or SPP
 - Reduces design/debug time
 - Have always been available
- System-on-a-chip (SOC)
 - All components of system implemented on single chip
 - Made possible by increasing IC capacities
 - Changing the way COTS components sold
 - As intellectual property (IP) rather than actual IC
 - Behavioral, structural, or physical descriptions
 - Processor-level components known as cores
 - SOC built by integrating multiple descriptions

Cores

- Soft core
 - Synthesizable behavioral description
 - Typically written in HDL (VHDL/Verilog)
- Firm core
 - Structural description
 - Typically provided in HDL
- Hard core
 - Physical description
 - Provided in variety of physical layout file formats

Gajski's Y-chart



Advantages/disadvantages of hard core

- Ease of use
 - Developer already designed and tested core
 - Can use right away
 - Can expect to work correctly
- Predictability
 - Size, power, performance predicted accurately
- Not easily mapped (retargeted) to different process
 - E.g., core available for vendor X's 0.25 micrometer CMOS process
 - Can't use with vendor X's 0.18 micrometer process
 - Can't use with vendor Y

Advantages/disadvantages of soft/firm cores

- Soft cores
 - Can be synthesized to nearly any technology
 - Can optimize for particular use
 - E.g., delete unused portion of core
 - Lower power, smaller designs
 - Requires more design effort
 - May not work in technology not tested for
 - Not as optimized as hard core for same processor
- Firm cores
 - Compromise between hard and soft cores
 - Some retargetability
 - Limited optimization
 - Better predictability/ease of use

New challenges to processor providers

- Cores have dramatically changed business model
 - Pricing models
 - Past
 - Vendors sold product as IC to designers
 - Designers must buy any additional copies
 - Could not (economically) copy from original
 - Today
 - Vendors can sell as IP
 - Designers can make as many copies as needed
 - Vendor can use different pricing models
 - Royalty-based model
 - Similar to old IC model
 - Designer pays for each additional model
 - Fixed price model
 - One price for IP and as many copies as needed
 - Many other models used

IP protection

- Past
 - Illegally copying IC very difficult
 - Reverse engineering required tremendous, deliberate effort
 - “Accidental” copying not possible
- Today
 - Cores sold in electronic format
 - Deliberate/accidental unauthorized copying easier
 - Safeguards greatly increased
 - Contracts to ensure no copying/distributing
 - Encryption techniques
 - limit actual exposure to IP
 - Watermarking
 - determines if particular instance of processor was copied
 - whether copy authorized

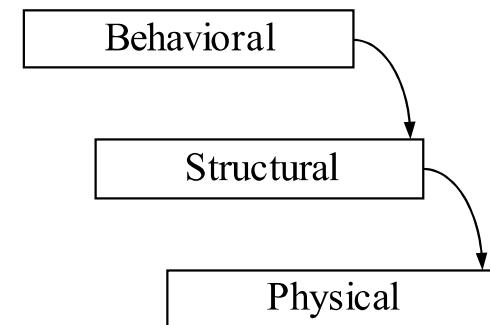
New challenges to processor users

- Licensing arrangements
 - Not as easy as purchasing IC
 - More contracts enforcing pricing model and IP protection
 - Possibly requiring legal assistance
- Extra design effort
 - Especially for soft cores
 - Must still be synthesized and tested
 - Minor differences in synthesis tools can cause problems
- Verification requirements more difficult
 - Extensive testing for synthesized soft cores and soft/firm cores mapped to particular technology
 - Ensure correct synthesis
 - Timing and power vary between implementations
 - Early verification critical
 - Cores buried within IC
 - Cannot simply replace bad core

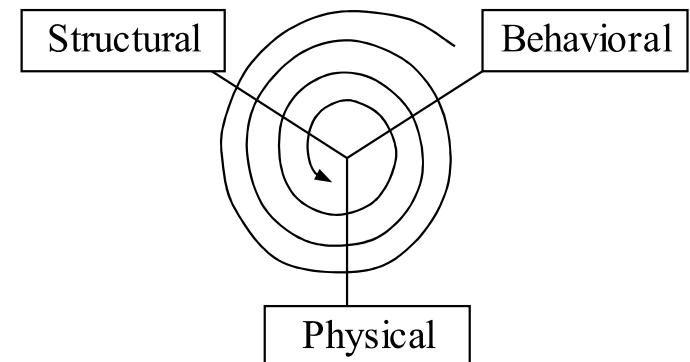
Design process model

- Describes order that design steps are processed
 - Behavior description step
 - Behavior to structure conversion step
 - Mapping structure to physical implementation step
- Waterfall model
 - Proceed to next step only after current step completed
- Spiral model
 - Proceed through 3 steps in order but with less detail
 - Repeat 3 steps gradually increasing detail
 - Keep repeating until desired system obtained
 - Becoming extremely popular (hardware & software development)

Waterfall design model



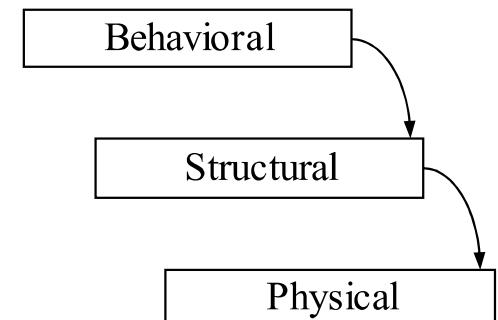
Spiral design model



Waterfall method

- Not very realistic
 - Bugs often found in later steps that must be fixed in earlier step
 - E.g., forgot to handle certain input condition
 - Prototype often needed to know complete desired behavior
 - E.g., customer adds features after product demo
 - System specifications commonly change
 - E.g., to remain competitive by reducing power, size
 - Certain features dropped
- Unexpected iterations back through 3 steps cause missed deadlines
 - Lost revenues
 - May never make it to market

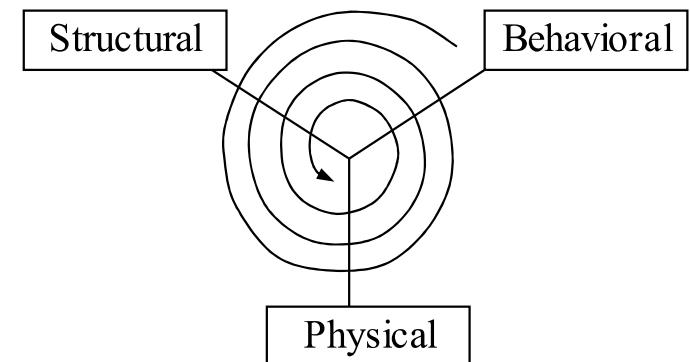
Waterfall design model



Spiral method

- First iteration of 3 steps incomplete
- Much faster, though
 - End up with prototype
 - Use to test basic functions
 - Get idea of functions to add/remove
 - Original iteration experience helps in following iterations of 3 steps
- Must come up with ways to obtain structure and physical implementations quickly
 - E.g., FPGAs for prototype
 - silicon for final product
 - May have to use more tools
 - Extra effort/cost
- Could require more time than waterfall method
 - If correct implementation first time with waterfall

Spiral design model

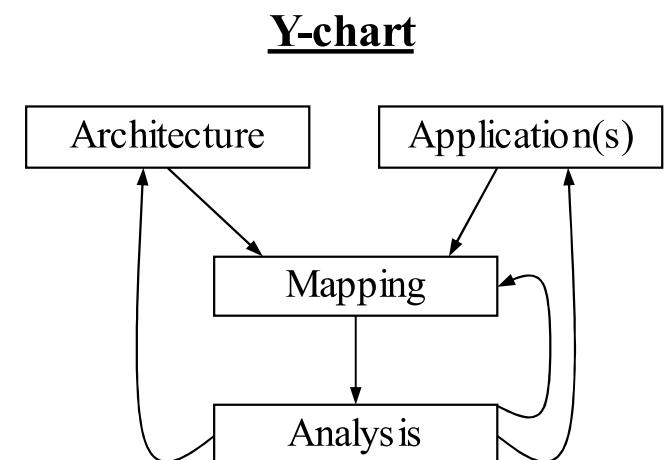


General-purpose processor design models

- Previous slides focused on SPPs
- Can apply equally to GPPs
 - Waterfall model
 - Structure developed by particular company
 - Acquired by embedded system designer
 - Designer develops software (behavior)
 - Designer maps application to architecture
 - Compilation
 - Manual design
 - Spiral-like model
 - Beginning to be applied by embedded system designers

Spiral-like model

- Designer develops or acquires architecture
- Develops application(s)
- Maps application to architecture
- Analyzes design metrics
- Now makes choice
 - Modify mapping
 - Modify application(s) to better suit architecture
 - Modify architecture to better suit application(s)
 - Not as difficult now
 - Maturation of synthesis/compilers
 - IPs can be tuned
- Continue refining to lower abstraction level until particular implementation chosen



Summary

- Design technology seeks to reduce gap between IC capacity growth and designer productivity growth
 - Synthesis has changed digital design
 - Increased IC capacity means sw/hw components coexist on one chip
 - Design paradigm shift to core-based design
 - Simulation essential but hard
 - Spiral design process is popular
-

Book Summary

- Embedded systems are common and growing
 - Such systems are very different from in the past due to increased IC capacities and automation tools
 - Indicator: National Science Foundation just created a separate program on Embedded Systems (2002).
- New view:
 - Embedded computing systems are built from a collection of *processors*, some general-purpose (sw), some single-purpose (hw)
 - Hw/sw differ in design metrics, not in some fundamental way
 - Memory and interfaces necessary to complete system
 - Days of embedded system design as assembly-level programming of one microprocessor are fading away
- Need to focus on higher-level issues
 - State machines, concurrent processes, control systems
 - IC technologies, design technologies
- There's a growing, challenging and exciting world of embedded systems design out there. There's also much more to learn. Enjoy!