CS

247

CHUN

FALL

2018

AS PRINT SHOP

Associated Students Print Shop

San Jose State University

- Interesting Facts:
  - The computer is fast, yet slow
    - Solves problem at very high speed
    - But takes a long time to set up the problem for solution
  - The computer is costly, yet inexpensive
    - Mainframe may cost $500 per hour
    - But problems can be solved so quickly that cost per problem is low
  - No matter how much computer power people have, they always want more
    - Computer capacity exhibits a highly "elastic" demand
    - Self-reinforcing cycle:
      - More powerful computers make new applications possible
      - New applications require even more powerful computers
    - *=> The Demand for computational power will always exceed the supply <=*
      - More Accuracy (wider data paths)
      - More Data (larger, faster memory; more address bits)
      - More Kinds of Data (multimedia, A/V input and output)
      - More Speed (real-time processing)
      - More Interconnectivity (shared workstations, LAN, Internet)

- Computers can be made more powerful by using a good Computer Architecture

  Computer: A Programmable Electronic Device which Stores and Processes Data

  Architecture: The Art or Science of Designing and Building

  Computer Architecture:
  - As a process:
    - The design and optimization of computers
    - Providing maximum performance at the cheapest price
    - Getting hardware subsystems and software to cooperate harmoniously
  - As a description of a machine:
    - The attributes of a computer as seen by a programmer (i.e. the user)
      - Instruction set and format, op codes, addressing modes, registers
    - Instruction set architecture:
      - The actual programmer-visible instruction set
      - Serves as the boundary between the software and hardware

- Traditional digital computers operate as Sequential von Neumann Machines

  Higher Performance machines are typically Parallel in nature

  Advanced Optimization Techniques are usually employed

Digital Computer System Design has 4 major engineering activities:

1) System Design
   - Specification for computer system formulated with regard to:
     - Speed, Cost, Size, Reliability, Maintainability
   - Overall Design Philosophy is established including tradeoffs involving:
     - Arithmetic and logic functions available
     - Serial vs. Parallel
     - LSI Logic family
2) Logic Design
   - High-Level architectural issues
     - Computer word length
     - Organization of machine-code instruction set
   - Low-Level
     - Logic diagrams of gates
3) Circuit Design
   - Transistor-level representation of logic diagrams
4) Production and Marketing

- Technology has changed digital systems design

  Before: Designs were implemented at basic gate level
  MSI and LSI building blocks were used (hundreds of transistors)
  Optimization was done at the logic level with Karnaugh Maps, etc.

  Now: Fully Integrated VLSI chips (millions of transistors)
  Optimization occurs in how the blocks are put together
  Also, in how they are used algorithmically

  Therefore:
  Emphasis in computer engineering has shifted:
  Away from logic and circuit design
  More towards computer architecture and software systems.

- To create high-performance architectures at a cost-effective price,
  Knowledge of the design alternatives and tradeoffs is important

- Every computer scientist should know something about machine organization
  - Can help you program ("Drive") more efficiently
  - Can help you design hardware more efficiently
  - Good Designers:
    - Optimize the Price/Performance of the System (Hardware *and* Software)
    - Factor into account Reliability, Testability, and Fault-Tolerance

Major topics

- Computer-Aided-Design (CAD) and Simulation
  - VHDL
- Number Representation
  - Floating Point
  - Phantom 1, Excess Exponent
  - Guard and Sticky Bits
- High Speed Arithmetic Algorithms
  - Carry Select Adder
  - Booth's Algorithm
  - Non-Restoring Division
- Memory
  - RAID Disk Architectures
- Communication and Recording Protocols
  - NRZ, Manchester
- Low Power Design and Thermal Dissipation Techniques
- Advanced Architectures (SIMD, MIMD)
  - "Non-von" Machines
    - DataFlow
  - Pipelined Processors
    - Data and Branch Hazard Minimization
  - Multi-Processing
    - Array, Vector Computers
    - Multi-Cache Coherency
    - Languages, Algorithms, and Compilers for Parallel Processing
- Fault-Tolerance
  - Terminology
  - Modeling
    - Stuck-At Fault Model
  - Implementation Techniques for Hardware and Software
    - Design For Fault-Tolerance
      - TMR, N-Version Programming
    - Detection and Recovery
      - Design For Testability
      - Controllability and Observability
      - LSSD, BIST, JTAG, Recovery Blocks
  - Fault-Tolerant Systems and Architectures
- Massively Parallel Processors
  - Neural Networks

- Design Optimization
  - Computer Architect must design a computer to meet:
    - Functional specs
    - Price goals
    - Performance goals
  - Hardware is "general purpose"
    - Manufacturer supplies machine, basic utility programs (e.g. OS, compilers)
    - Customer writes specific application programs using general purpose machine
    - => No ONE architecture best for ALL applications
    - => Computer designer must optimize for "typical" applications
  - Speed of a computer determined by:
    - \- Physical parameters:
      - The electrical time to switch a device (e.g. transistor)
      - Propagation of signals along conductors (about 2/3 speed of light)
      - The mechanical time to rotate a disk or to move the seek head
    - \- Logical parameters:
      - How the physical devices are organized to build circuits
  - Speed of a computer can be improved by:
    - Using faster components
      - But fast devices are costly
      - Fast devices also consume higher power and run hotter
      - Physical limits of speed of light and atomic dimensions reached by 2000
    - Using a suitable computer structure (e.g. parallel vs. serial)
      - But more hardware increases cost

=> We discuss ways to increase Logical Speed by using Optimized Structures

- Design "trade-offs" must occur between the most important parameters.
  - No hard and fast rules for tradeoffs
  - Prioritized list of criteria needs to be established
  - Designers with different goals use different architectures
    - e.g.) Increase in speed might be "paid for by" a decrease in reliability
  - Some common pitfalls:
    - Ignoring Cost
    - Ignoring Marketplace
    - Ignoring Technology Trends
    - Performing Local Optimization
    - Overlooking Design Complexity
    - Using Faulty Performance Measures

- It is important to focus on optimizing machine cost / performance
  - A cheap machine may not be very useful
  - A very powerful machine may not be affordable
  - It is costly to spend time to optimize
    - Cost / performance curve can shift very fast
    - Need to predict cost / performance factor when machine is completed
    - May be more profitable to hit market sooner with 'weaker' machine
  - Computer designs will always be measured by cost / performance
    - Finding the best balance will always be an art

- Marketplace strongly desires backwards compatibility
  - Changing architectures often outdates existing application software
  - Billions have been previously invested in software
    - Cost to rewrite application S/W for different architecture or OS is enormous
  - From a market standpoint, compatibility with existing software is imperative
    - Interim solution during transition: Software emulation (e.g., Power Mac)

- A successful architecture must also last through trends
  - Must survive changes in H/W and S/W technologies, and application needs
  - Rate of change and impact on design of computer is tremendous compared to:
    - Light Bulb
    - Automobile
    - Airplane
  - Trends in the computer industry:
    - Cheaper
    - Faster
    - Longer Words (12, 16, 32 bits)
      - Average program's memory reqmts grow 1.5X-2X per year (1 addr bit / yr)
      - ex.) IBM 360 architecture became extinct because of lack of address bits
    - Larger primary memory
    - More capable (operating systems, languages, applications, etc.)
    - Diverse I/O capabilities
    - Diverse secondary memories (disks, etc.)
  - Trends in consumer applications of computers:
    - A number of problems require enormous computational power
      - Meteorology
      - Ballistic Missile Defense
      - PDA Handwriting, voice recognition
      - Interactive TV, real-time virtual reality, multimedia

Trends in integrated circuit technology:

- Manufacturing process Learning curve makes cost inversely $\propto$ to volume
  - Cost of module type must be spread among the copies
  - To minimize costs, need to use more copies of the same module
- Before: Designer had to minimize component count
- Now: Designer must also minimize number of different module types
- Ever-increasing density and integrated capabilities
  - Each generation of LSI has caused design methods to change
  - Changes may not be continuous; discrete steps (8, 16, 32 bits)

=> Designer must be aware of trends in technology and computer usage
  - ex.) Distributed workstations and PCs vs. IBM/DEC Mainframes

• Need Global Optimization of Total Costs of Both Hardware and Software

- Hardware: The tangible, visible devices of a computing system
- Software: Computer programs, compilers, editors, operating systems

=> Hardware and Software are logically equivalent
  - Any operation performed by software can be built directly into hardware
  - Any instruction executed by hardware can also be simulated in software

Major advantages of a software implementation:
  - Lower cost of errors
  - Easier Design
  - Simpler upgrading

Major advantages of a hardware implementation:
  - Performance

Ideally, Hardware and Software design should proceed together

Factor impact of design decisions on design time for *both* H/W *and* S/W

=> Balancing hardware and software will lead to the best machine
  - There is no single correct solution
  - Technology trends constantly change the H/W S/W boundary

• Design complexity must be factored into account

- Complex designs take longer, prolonging time to market
- Complex designs may not be manufacturable (especially for VLSI)
  - Low Fabrication Yields
  - Lack of Regularity
  - Lack of Testability
  - Lack of Reusability
- Generally easier to deal with complexity in software than hardware
  - Easier to debug and change software

- Problems with standard metrics of computer performance
  - MIPS: Million Instructions Per Second
    - MIPS is dependent on the instruction set (machine dependent)
    - MIPS varies between programs on the same machine
    - MIPS can vary inversely to performance
      - e.g.) Floating point instructions take more clocks than integer ops
        - Machine with floating point hardware may execute faster,
          - but will have a lower MIPS rating
      - e.g.) Compiler optimized code can be faster, but have a lower MIPS
  - MFLOPS: Million Floating-Point Operations per second
    - Machine and program dependent
    - Set of floating point operations not consistent across machines
  - No such thing as a 'typical' synthetic benchmark program
    - Control flow on DLX can vary 5% to 23%
  - Optimizing Compilers can take out loops in standard benchmark programs
    - e.g.) 25% of Dhrystone loops removed
  - Peak performance vs. average (and realistic day-to-day) performance
    - Gap between peak and observed performance can be factor of 10+
    - Peak: "Guaranteed not to exceed"

- Definition of a "Good" Computer Architecture
  - Some Measures:
    - The efficiency with which it can be represented in real of circuits (H/W Dev)
    - The ease with which a compiler can be developed on it (S/W Dev)
    - Amount of memory space needed to represent programs (Space Efficiency)
    - Amount of data transfers b/w CPU, memory, and registers (Time Efficient)
  - Ideal Characteristics:
    - Only one instruction should be executed for a HLL operator
    - There should be only one memory reference for each operand
    - Use explicit addressing only for operands whose location cannot be inferred
    - Addresses should be short
  - So an "ideal" machine will execute a minimum number of instructions, each of minimum size, and transfer only the minimum data bits required per op

- Speed *AND* Low Cost can be achieved by:
  - Blending high with low speed components or techniques (e.g. memory hierarchy)
  - Replicating function units (e.g. concurrency in space)
  - Overlapping operations (e.g. concurrency in time)
  - Instruction Branch Prediction (e.g. speculative computation)

- Computer-Aided-Design (CAD) and Simulation:
  - Models behavior of a real hardware system using an approximation in software
  - Mathematical formulas are used to emulate and predict physical phenomena
  - Can provide time-based information
  - Primary method used for evaluation of systems before manufacture
    - Must check final design for meeting all functional specifications
    - Building hardware prototypes is impractical for large systems
      - Takes too long and costs too much
    - Especially important for custom fabricated VLSI chips
      - Requires photomask, fab line setup, and yield curve ramp-up
  - Enables:
    - Design verification through 'soft' Rapid Prototyping
      - Allows designer to detect conceptual errors as early as possible
    - Effects of changes in the design to be analyzed quickly
      - Especially those changes that arrive late (i.e., 'just-in-time')
    - Performance Evaluation
    - Identification and tuning of critical components (optimization)
    - Comparison of [possibly experimental] architectures
      - Trade-off Evaluation of different designs
    - Hardware / Software Partitioning
    - Parallel design of hardware and software
    - Integration and Testing
  - Advantages:
    - Time and money are saved by removing faults before manufacture
    - Simulator description of design can serve as documentation
    - When number of inputs is small, exhaustive testing is possible
      - Software driven test vector generation
    - Can be even better than building a hardware prototype
      - Enables internal functions to be observed; not possible in pin-limited ICs
      - Slow-down or accelerate playback in virtual time
  - Limitations:
    - "Approximation" must be accurate "enough", yet computationally efficient
      - e.g.) Circuit-level simulation for entire complex systems not possible
    - Behavior at level boundaries accurate only if certain restrictions hold
      - e.g.) Rise, Fall, and Transient voltages of circuit level must map into 1s/0s
    - Hierarchical structured simulator is required
      - Multi-level simulation using same input description language
    - Must allow a 'top-down' design approach to be used

- Simulation can be performed at a number of different levels for digital systems:
  - Behavioral level, Functional Level, or Systems Level:
    - Emulates stimulus / response behavior of subsystem components (e.g. ALU)
    - No attempt made to replicate the internal mechanism by which this is achieved
      - Device is considered a "black box"
    - Most efficient method of simulation; least accurate
    - Designer is more concerned with *what* tasks the system needs to perform
  - Register-Transfer Level
    - Data flow involving components that handle groups of bits (e.g. register, mux)
    - Signals at this level might be integers
  - Gate Level or Logic Level
    - Components are gates (e.g. NAND gates)
    - Signals at this level correspond to individual bits
    - For VHDL, this is the most computation intensive and most accurate mode
    - Designer is concerned with *how* the system will perform its tasks
  - Circuit-Level
    - Models individual transistors
    - Generally analog in nature for detailed timing analysis (e.g. SPICE)
    - VHDL probably not used at this level
    - More accurate than gate-level; more computationally intensive too
  - Layout Level
    - Definition of the hardware in silicon structures
    - Can model on-chip parasitics, inductance, electron migration
    - Generally only used by device physicists
    - Uses computation-intensive differential equations for utmost accuracy

- Structured Machine Design:
  - Multiple levels of hierarchy are used to manage system complexity
  - Each higher level is an abstraction of the level below it
  - User (programmer or hardware designer) works at the highest level possible
    - No need for user to be concerned with details of lower levels
  - e.g.) A programmer need not be aware of how the level he is using is implemented
    - Each level is a virtual machine
    - User thinks of it as a real physical machine
    - However, it does not really exist.
    - It is implemented by a lower level (which could be another virtual machine)
  - Enables machine (or software) to be built layer-by-layer
  - Tremendously simplifies the production of complex (virtual) machines

- VHSIC: Very High Speed Integrated Circuit
  - DoD program to advance the state-of-the-art in chip design & fabrication

- VHDL: VHSIC Hardware Description Language
  - A language and simulation environment for digital devices (esp. VHSIC)
  - A 1987 standardization effort by the Department of Defense (DoD)
  - Definition involved strong industry participation
  - Based on Ada, another DoD standard (superset of Ada; 81 vs. 63 rsrvd words)
  - Consists of the Language and the Support Environment
  - Support Environment:
    - Analyzer: "compiler" which checks VHDL source syntax and static semantics
    - Library: stores intermediate format generated by analyzer
    - Simulator: verifies (through simulation) the dynamic semantics

- VHDL provides:
  - Abstractions of digital hardware in a single cohesive language
    - Based on: generalized model of stimulus / response behavior
      - Behavior is described using computer language-like code
      - A functional component reacts to activity on its input connections.
      - It responds through its output connections.
    - Can describe digital hardware ranging from logic gates to entire systems
    - Includes: Behavioral, and Structural
  - Documentation
    - Before:
      - Typical delivery of hardware to govt. included 1,000s pgs. of documents
        - Needed during acceptance / testing / maintenance of component
      - When component needed replacement, large effort was required
        - Intended behavior of part had to be reconstructed from document
    - Now:
      - Deliver VHDL with part
        - VHDL is human readable; can serve as documentation
        - VHDL is machine executable; can be used for simulation
      - VHDL serves as basis for documentation and reprocurement
  - Design Information Interchange
    - Models developed at one location will run at other locations
  - Large-Scale Design
    - Enables design decomposition
    - Supports multiperson / multicompany design teams

- A DoD and Industry (IEEE-1076-1987) Standard
    - Public Availability
    - Enables easy communication of designs among participants
    - Before:
        - Each CAD tool vendor had their own proprietary description language
        - Disparate tools for each level of simulation
    - Now:
        - ONE language EVERYONE can use, for ALL levels of simulation.
        - VHDL is accepted by a number of CAD tools.
- A Technology and Process Independent Modeling Language
    - Can survive new technology (CMOS, GaAs) and fab methods
- Wide range of descriptive capability
    - Can model from a top-level behavioral view down to detailed logic timing view
- Flexible design methodologies (e.g. top-down, bottom-up, or mixed)
- Mixing of multiple level models in one simulation
    - Designer can efficiently simulate large complex digital systems
    - Use detailed level only for portions of hardware of interest
    - Use less-accurate, more efficient levels for hardware already debugged
- Hierarchical abstractions to control scale-up problems of large systems
    - Can decompose a large, complex problem into simpler sub-problems
- Schematic entry
    - Graphical interface hides user from the language
    - Enables user to place and interconnect boxes (entities) with wires (signals)
    - User merely draws schematic
        - Added benefit: Schematic diagram documentation
- Reusability
    - Store entities in library for future use
- Input to automatic logic synthesis tools / Silicon Compilers
    - Give the silicon compiler a high-level behavioral description
    - Compiler automatically generates netlist of gates needed to get behavior
- Input to automatic test pattern generation (in the future)
    - Given a netlist, ATPG generates test vectors to check manufactured IC
    - Presently limited to exhaustive or stuck-at fault model
- Formal proof using logical calculus (in the future)
    - e.g.) Predicate calculus could prove that 8 bit wide register cannot overflow
- An amalgamation of: sequential, concurrent, net-list, timing, and waveform langs.
- A man-to-tool, man-to-man, and tool-to-tool communication medium

• We will be using VHDL primarily as a gate-level logic simulator (<u>Signals</u> of type <u>bit</u>)

- Logic Simulation
  - Logic (gate-level) diagram for design is described in topological form (i.e. netlist)
  - Each primitive element's behavior is coded and its input/output specified.
  - Propagation delays can be assigned to each gate to do timing analysis
  - Test vector input stimulus (supplied by user) is applied to model of system
  - Binary output of each logic element is calculated at each simulation time step.
    - Simulation time continues to advance until a steady state is reached

- The VHDL Language is:
  - Similar to other programming languages in that:
    - VHDL is "Ada-like"
    - Design units are read by a compiler and checked for proper syntax
    - Object modules are placed in a VHDL library
    - Objects are loaded (i.e. linked) into a simulator and executed
  - Different from other programming languages in that:
    - It has some unique constructs for the H/W designer
      - Can build a structural model of interconnected functional units
    - It offers a notation for signal delays to model gate propagation time
    - It can execute statements concurrently
      - Most algorithms are sequential
        - A program executes one instruction after another
      - However, H/W consists of concurrently active components
        - So, VHDL enables concurrent simulation of statements
  - Strongly typed language
    - Enables errors to be caught early at compile time
      - e.g.) Cannot connect an 8 bit part to a 4 bit part
    - We will use mainly type bit ('0', '1')
  - Insensitive to case
  - Comments marked by --
  - Highly powerful and verbose
    - We will study a small core subset of the language

- Symbolic names:
  - Must begin with an alphabetic letter (a-z) followed by a letter, underscore, or digit
  - Must not be a reserved word (e.g. in, out, signal, port, bit, etc.)
    - *Suggestion: Append a numeric suffix to all user chosen names*
      - *Avoids any conflict since no reserved words have numbers*

- VHDL uses signals to define a data pathway between two functional units

  Although VHDL offers both variables and signals, we will use mainly signals

  -Variables:

  Data objects that can be assigned a current value

  Assignment statements execute sequentially, 'in-line' with the code

  Variable values are not scheduled

  Target values update immediately upon execution of assignment stmt.

  Do not correlate well with H/W constructs which execute concurrently

  -Signals:

  Data objects that can be assigned a time series of values

  i.e., can specify (value, time) pairs for the data object

  Assignment statements simulate concurrent execution

  Not necessarily 'In-Line' execution order

  Signal values are scheduled

  Target values are updated only after a wait time lapses (if specified)

  Correlates to the data on physical hardware wires

  Serves as a communication path from one component to another

  Signals connect components at their ports.

- Signal Assignment Statement

  General form is:

  signal-name <= *value* after *time* ;

  where: *value* is a bit type or a computed logical Boolean expression

  Bit Type is '0' or '1' *(note: single quotes are required)*

  Logical operators in VHDL include: and, or, nand, nor, xor, not

  time is an integer number of time units in VHDL (we will use ns)

  Used to specify the delay before updating the target signal

  Models propagation time of logic gates or other digital circuitry

  (note: units are required; also, ns needs a leading blank space)

  Examples:

```
X <= '0' ;
Y <= '1' ;
Sum <= '1' after 5 ns ;
Carry <= '0' after 8 ns ;
```

  If no *after time* specified, it defaults to *after 0 ns*

  i.e., it executes and updates immediately at the current simulation time

  If an *after time* is specified, it is relative to the current simulation time

  So, if executed at time = 7 ns and delay is 5 ns, signal changes at 12 ns

- Current Simulation Time
  - The virtual time 'pointer' of the simulator clock
  - System state at current simulation time gives a snapshot of it in the simulation

- Simulation Time:
  - Based on the concept of discrete events and event-driven simulation
  - Discrete-event simulation:
    - State variables change only at a countable number of points in time
    - These points in time are the ones at which an event occurs
    - Event: An instantaneous signal change which can alter the state of the system
    - Simulation time does not advance in normal 'analog' or 'digital' fashion
      - Simulator clock does not use constant, fixed increments
      - Skips periods of no activity; Jumps to next most imminent event time
  - A statement can 'go' if any of its RHS signals change value at the current sim time
    - Its target value can only change if any of the signals it depends upon changes
  - A statement with no right-hand-side (RHS) signals will 'go' at simulation time 0
    - It does not depend on any other signals; 'Goes' once at simulation startup
    - Time specification in this case can be construed as absolute simulation time
  - When a signal assignment statement 'goes', it posts an event for the target signal

- Event Queue
  - Used by the VHDL simulator to keep track of scheduled signal changes (events)
  - Stores a sorted list of events in time ascending order
  - There can be multiple events scheduled for the same time
  - So, the above 4 signal assignment statements produce the following events

Time	0	5	8
	X: ('0', 0)	Sum: ('1', 5)	Carry: ('0', 8)
	Y: ('1', 0)		

- A target signal's value can depend on other signals specified on the RHS
  - A signal assignment statement is activated (can 'go') upon a RHS event
    - i.e., whenever a RHS signal changes in value

  Example:

```
Q <= '1' after 2 ns ;
QBAR <= not Q after 5 ns ;
```

Time	2	7
	Q: ('1', 2)	QBAR: ('0', 7)

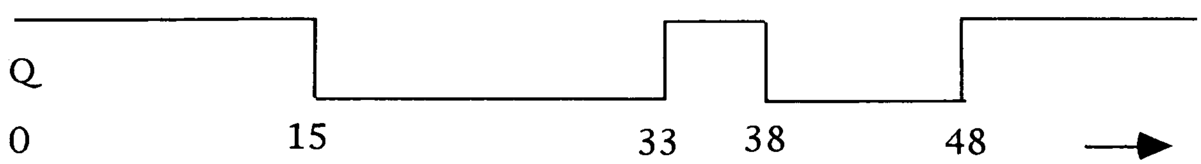
- Multiple (value, time) pairs can be assigned in a single statement
  - Enables any arbitrarily shaped waveform to be generated
  - General form is:

```
signal-name  <=  value after time ,
                 value after time ,
                 ...
                 value after time ;
```

Example:

```
Q <= '1' after 0 ns,
     '0' after 15 ns,
     '1' after 33 ns,
     '0' after 38 ns,
     '1' after 48 ns;
```

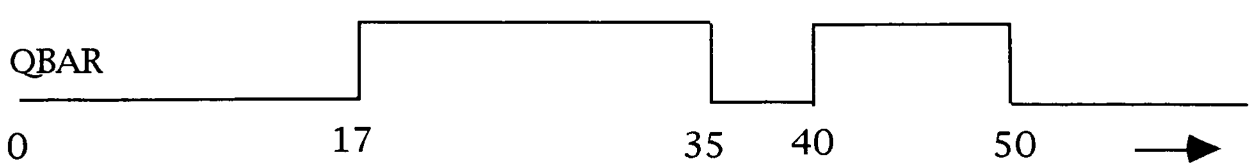
Since this assignment statement does not depend on any signals on RHS,
it is activated only once at startup (i.e., at simulation time = 0)
Entire series of events is posted to event queue when simulation run begins
Last value specified is steady state value ('1' for all time after 48 ns)
Creates the waveform below.



- An assignment statement dependent on a RHS signal goes whenever RHS has event
  Example:

```
QBAR <= not Q after 2 ns ;
```

Creates a negated Q which lags Q by 2 ns.



- Note: It is important to initialize signal values at simulator startup
  - All signals are initialized to '0' at simulation time 0
  - Delay time of dependent signal is not 'rolled back into negative time' by simulator
  - Need to initialize value of QBAR via a signal declaration statement

- Note: Above event queue is 'simplified' inertial delay model

- **Delays**

  Controls effect of a signal assignment which is dependent on RHS signal changes

  Two types in VHDL:

  - Inertial Delay

    The default mode for VHDL simulator

    Models components using a minimum "setup and hold" time

    Value on inputs must persist for given time before output responds

    Useful in limiting 'spikes' associated with certain circuits

    e.g.) Limits transients of a flip-flop in the process of switching

  - Transport Delay

    Specified using a TRANSPORT keyword in the signal assignment stmt

    e.g.) signal-name <= transport value after time ;

    Similar to wire or transmission time delay

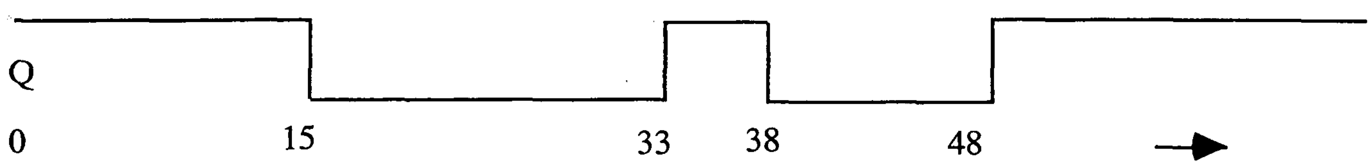
    Output always changes regardless of time duration of input signal

    Useful when more detailed examination of each simulation time step reqd

    Good for observing transient response of output

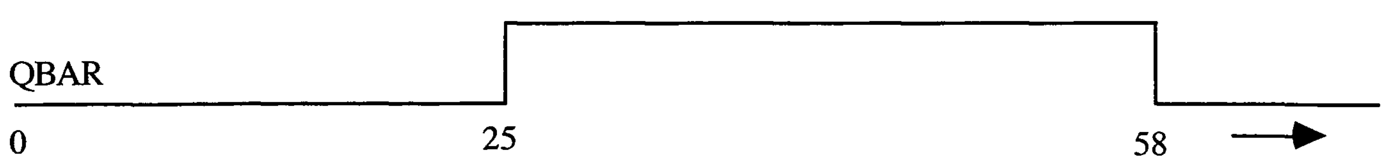
    High speed inputs without minimum steady state can be modeled

    Can provide the equivalent of macroscopic circuit simulation



- For Inertial Delay, important to ensure that RHS signals persist for at least delay time

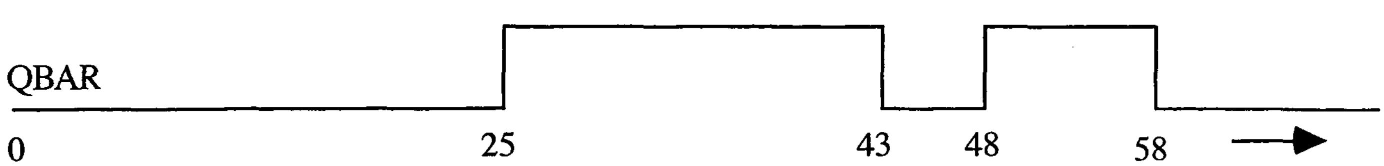
  e.g.) QBAR <= not Q after 10 ns ;



- For transport delay, important to realize that target signal's value computed at CST

  The computed value is just not assigned to the target signal until after the delay

  Contrast with: Wait for specified delay, then read RHS signals and update output



- The Design Entity
    - Represents hardware at any level of abstraction
    - The primary hardware abstraction used to model a digital device in a system
    - A component of the system, i.e. a 'part' (e.g. logic gate, chip, PC board, etc.)
    - Consists of two sections:
        - 1) Entity Declaration
        - 2) Architectural Body

- Entity Declaration
    - Defines a new component name and its input/output connections
    - Only describes a "black box" with ports
        - No information provided as to it's internal composition or it's function
        - Just like a socket spec. that constrains the type of chip that can be put into it
    - Defines the external view (i.e. the interface) of a hardware component:
        - A component's connections and means of communication to the outside world
            - e.g.) The number and names of the pins for an IC
        - Interface-list provides the means to connect the entity to other entities
            - Declares:
                - Name of the ports
                - Direction of data flow (i.e. in, out, or inout)
                    - IN: Value of input port can only be read within the entity
                    - OUT: Value of output port can only be updated within the entity
                    - INOUT: A bidirectional port which can be read and updated
                - Type of data that flows through each of the ports

General form is:

```
ENTITY identifier IS
    PORT interface-list ;
END identifier ;
```

Example:

```
ENTITY and2 IS
    PORT (in1, in2 : IN BIT ; out1 : OUT BIT);
END and2;
```

It is valid to have a top-level self-contained entity with no inputs or outputs

A testbench which contains a unit under test and a test generator is self-contained

Example:

```
ENTITY testbench IS
END;
```

- VHDL separates:
  - -The entity declaration (the I/O interface) of a design from its
  - -Architectural implementation details

  Enables entity A to be used as part of entity B even if A not completely designed

      Once entity declaration is compiled, it can be referenced as a component

  Any update to architecture body has no influence on the entity declaration

      Facilitates experimenting with alternative implementations (architectures)

      Any architecture using entity as a component is not affected either

      Enables one part of a design to change without recompiling other parts

  A design entity's declaration must be compiled before its associated architecture

- Analogy:

  Given a certain interface definition (entity declaration),

      there may be a variety of internal implementations (architectures).

  Architectural variants include vendors (Intel, AMD) and technology (TTL, CMOS).

  Entity declaration enables one to specify the socket and wiring for the

      circuit board before even having completed the chip that will go in the socket.

  To use an entity as part of a larger device, one must specify

      how to wire it into the device (via a component specification)

- Architectural Body

  Describes the functional internal implementational details of an entity

  Specifies the behavior, interconnections, and components of a design entity

  Expresses the relationships between the inputs and outputs of a design entity

General form is:

```
architecture identifier of entity-name is
  declarations
begin
  statements
end identifier ;
```

where *identifier* is by convention either *Behavioral* or *Structural*

declarations are signal and/or component declarations

statements are behav. Boolean exprs. or struct. component instantiations

VHDL has two types of architectural bodies that can be used to describe an entity

1) Behavioral description resemble algorithms of classical prog. languages

   Behavioral description defines functionalities of a device

   Described using an algorithm, i.e. a program of concurrent signal asgmts

2) Structural descriptions are essentially netlists

   Structural description defines an interconnection of components

- There is no precise dividing line between behavioral and structural
  - All VHDL components must ultimately be given behavioral descriptions
  - All lowest (leaf) level components of an entity require a behavioral model
    - Even gate-level simulations require that the behavior of the gate be specified

- Architectural Body: Behavioral Description
  - Uses a Dataflow style of modeling
    - When (new) data becomes available, an output is computed
  - Expressed using primarily concurrent signal assignment statements
  - Tells the simulator how building block reacts to all possible inputs that it sees
  - Must be provided for each primitive building block in the design
  - Structure is not explicitly specified; however, it can be implicitly deduced
    - Boolean functional relationship operators map easily to real physical gates
  - A signal assignment statement is executed only when a RHS signal changes
    - Example:

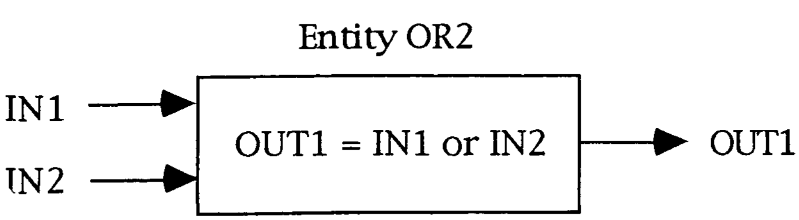
```
ENTITY or2 IS
     PORT    (in1 , in2 : IN BIT ;  out1 : OUT BIT);
END or2;

ARCHITECTURE behavioral OF or2 IS
BEGIN
     out1 <= in1 OR in2 AFTER 2 ns;
END behavioral;
```

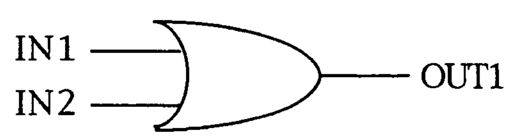
This model can be thought of as performing the following:

- Continually watches input ports (in1, in2) for any changes in value
- Whenever in1 or in2 changes, a new value xxx is <u>computed</u> & scheduled
- Port out1 <u>receives</u> the new value (experiences the event) after 2ns delay

This is the simplest recommended approach to modeling a basic gate

The above behavioral model creates the 'part' below for use in VHDL code





A Model with no signals in the declarations section has no local internal wires

Only has those signals listed in the ENTITY PORT specification

- Behavior Style Architecture Example:

  The signal TMP represents an internal wire that connects various Boolean exprs.

  - Scope of signal TMP is restricted to within the architecture body
    - Not visible outside of equiv
  - Declared with its type (we use signals only) in *declarations* section (line 2)

```
ENTITY equiv IS
    PORT (a, b : IN BIT ;  c : OUT BIT);
END equiv;

1   ARCHITECTURE behavior OF equiv IS
2   SIGNAL tmp :  BIT;
3   BEGIN
4       tmp <= a XOR b ;
5       c <= NOT tmp ;
6   END behavior;
```

Note: Order of the statements in VHDL behavioral dataflow model is not important

- Identical behavior results if lines 4 and 5 were reversed
- Simulation executes them concurrently in event driven fashion

- Architectural Body: Structural Description

  Describes what an entity's subcomponents are and how they are connected

  Provides a more direct correspondence to H/W than the behavioral description

  Uses classical "Netlist" type input common to CAD simulators (e.g. SPICE)

Example:

```
ENTITY equiv IS
    PORT (a, b : IN BIT ;  c : OUT BIT);
END equiv;

ARCHITECTURE structure OF equiv IS
SIGNAL tmp : BIT;
COMPONENT xor2 PORT (x, y: IN BIT;  z: OUT BIT); END COMPONENT;
COMPONENT inv PORT (x: IN BIT; z: OUT BIT); END COMPONENT;
BEGIN
    u0: xor2 PORT MAP (a, b, tmp);
    u1: inv PORT MAP (tmp, c);
END structure;
```

Note: The order of component instantiation statements is not important
- Identical structural specification results if lines 9 and 10 were reversed

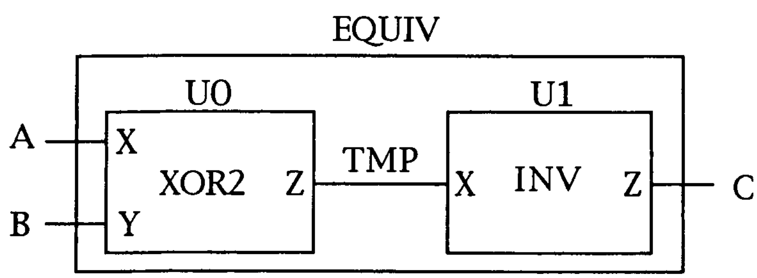
Above model has 4 actual signals visible within the part:
- All signals listed in the part's entity declaration as input/output ports (a, b, c)
- Any signals declared in a signal declaration statement (tmp)

Structure is described by declaring signals and connecting them to part ports
- PORT MAP in component instantiation statements hook parts together
    - Internal signals allow parts to communicate with each other
    - Positional association used: i^{th} signal maps (is connected) to i^{th} part port
        - e.g.) In line 9, actual signal A maps to port X of part XOR2

Structural model uses same ENTITY declaration as behavioral model of equiv
- Only one ENTITY declaration of equiv is needed
- Multiple architectures (struc or behav) can be specified for each entity (equiv)

If entity X is used as a part in entity Y, entity X is a *component* of entity Y
- Need to provide a *component declaration* for part X within architecture of Y
- Lines 6 and 7 of above model specify that parts xor2 and inv are to be used
- Specifications must correspond to a visible entity
    - Assumes that the following entities will eventually be designed/compiled

```
12  ENTITY xor2 IS PORT (x, y: IN BIT; z: OUT BIT); END xor2;
13  ENTITY inv IS PORT (x: IN BIT; z: OUT BIT); END inv;
```



- **Component Declaration vs. Entity Declaration**
    - Entity Declaration (lines 1-3, and 12, 13)
        - Is a separately compilable library unit
        - Never occurs inside another library unit
        - Declares that something really 'exists' in the design library
    - Component Declaration (lines 6, 7)
        - Never stands alone
        - Only occurs inside another entity's architecture
        - Merely declares a template that does not really 'exist' in the design library
    - Component Instantiation (lines 9, 10)
        - Creates an instance of the component in a structural architectural body
        - Instantiation references Component Declaration; not the Entity Declaration

Dual definition of external views gives designer an important flexibility

e.g.) Suppose hardware component A consists of parts B, C, and D

So, architecture body for A declares B, C, and D as components

Instantiations of B, C, and D appear in structural body of A

If instantiation referred directly to entity declaration of B, C, and D, then A could not be analyzed until entities B, C, and D were analyzed

Enables any order of analysis (i.e., compilation) to occur

Especially important when groups of designers are working together

Note: Use the same port names in comp. declrtn as those in the entity declaration

Otherwise, need to map the different names via a configuration stmt

We will not use configuration statements.

- Tester

Behavioral model which provides test pattern stimulus to the unit under test (UUT)

Generates a time-based sequence of 1/0 signal values for UUT's input ports

Input and output ports of tester should be in opposite direction from those of UUT

Example:

```
ENTITY equiv_tester IS
    PORT   (a, b : OUT BIT ;   c : IN BIT);
END equiv_tester;
ARCHITECTURE behavioral OF equiv_tester IS
BEGIN
    a <=    '0' AFTER 0 ns,
            '0' AFTER 10 ns,
            '1' AFTER 20 ns,
            '1' AFTER 30 ns;
    b <=    '0' AFTER 0 ns,
            '1' AFTER 10 ns,
            '0' AFTER 20 ns,
            '1' AFTER 30 ns;
END behavioral;
```

In this case, the data (signal events) applied can be summarized as:

Time	a	b
0 ns	0	0
10 ns	0	1
20 ns	1	0
30 ns	1	1

- **Testbench**

  A structural model that hooks together the tester and the unit under test (UUT)

  The tester and the unit under test are components of the testbench

  Need to be included in component declaration statements of the testbench

  Signals are declared for all inputs and outputs of the UUT

  Note: Its easiest to use same names for signals and all I/O ports on tester & UUT

  So, tester port x is connected to UUT port x using testbench signal x

  Tester's outputs should connect to the UUT's inputs

  Each UUT input port has a corresponding output port on the tester

  Tester's inputs should connect to the UUT's outputs

  Each UUT output port has a corresponding input port on the tester

  Testbench serves these main purposes:

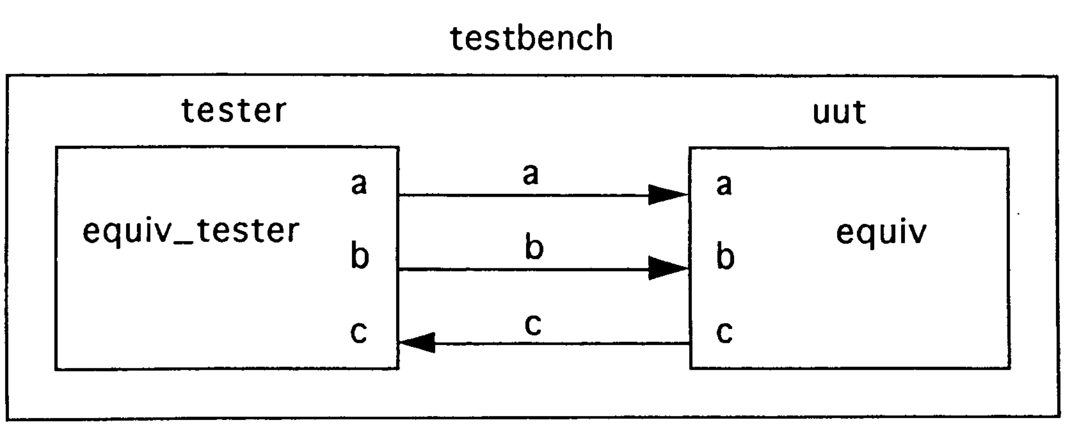
  1) Generates a stimulus for simulation (waveforms) with tester
  2) Applies stimulus to entity under test (UUT) through signal connections
  3) Enables monitoring of signals in testbench through VHDL simulator

     Testbench becomes the primary entity to be simulated

Example:

```
ENTITY testbench IS
END testbench;

ARCHITECTURE structure OF testbench IS
COMPONENT equiv_tester PORT(a, b: OUT BIT; c: IN BIT); END COMPONENT;
COMPONENT equiv PORT(a, b: IN BIT; c: OUT BIT); END COMPONENT;
SIGNAL a, b, c: BIT;
BEGIN
    tester: equiv_tester PORT MAP (a, b, c);
    UUT: equiv PORT MAP (a, b, c);
END structure;
```

Creates the following top-level self-contained testbench for checking equiv:



- A Typical VHDL Environment consists of several parts:
  1) WORK library reserved for your designs
     - WORK is the default current working library during analysis and simulation
     - A special library that does not need to be declared in order to be referenced
     - Always visible and always the current library for storing analyzed units
     - *Although different libraries can be declared, we will always use WORK*
  2) STD library containing packages STANDARD and TEXTIO
     - STANDARD package containing predefined data type declarations
       - Type Bit is ('0', '1');
       - Type Time (ns, us, ms)
     - TEXTIO package containing some utility functions
       - Ability to input and output data is limited in VHDL
  3) Predefined Behavioral Operators:
     - AND, OR, NAND, NOR, XOR, NOT
  4) Analyzer (Compiler)
     - Checks a VHDL unit for syntactic and static semantic correctness
     - Inserts the VHDL unit (if it is correct) into the WORK library
  5) Simulator
     - Executes models allowing user to verify run time and semantic correctness
     - Better simulators can draw graphical waveforms for signals of interest

  Note: PeakVHDL incorporates all of the above (included in the demo version)

- Order of Analysis
  - An Entity must always be compiled before its architecture
  - Best to compile lower level entities (e.g. gates) and their architectures first
  - Then continue compiling upwards in the design hierarchy
    - Compile top-level testbench last
  - *Note: PeakVHDL will do this automatically if options are set properly*
  - *Note: Easiest to keep each entity and architecture in one .VHD source file*
    - *This will allow you to easily reuse .VHD source file for later problems*

- VHDL allows hierarchical structured machine design to control complexity
  - Allows Efficient Simulation:
    - Only the part under investigation need be accurately simulated (gate level)
    - Remainder of system can be less accurately defined/simulated (behaviorally)
  - Allows Design Hierarchies
    - Structural description of a design can use other primitive components (etc.)
    - *Behavioral model for the device, or at least for all primitive devices are reqd.*

- **PeakVHDL:**
  - We will use a free demo version of PeakVHDL
    - All features are enabled, but has a limit on max size of design modules
    - This evaluation copy does not have a time limit on its usage period
  - Installation
    - Run the file PeakVHDL_Light.exe
    - Installer puts software into C:\PeakVHDL
    - Easiest way to launch: START/PROGRAMS/AccoladePeakVHDL/PeakVHDL
- Typical (suggested) compilation, link, and simulation scenario:
  - PeakVHDL uses projects to store modules
  - File Naming Conventions:
    - Be cognizant of the difference between .vhd File = Module = Entity
.acc File = Project
    - Projects (a collection of modules) have .acc extension
    - Modules (VHDL source files) have .vhd extension
      - Be sure to use .vhd extension so that built-in editor will recognize it
      - Editor is language sensitive and will color key words automatically
  - Each project can contain multiple modules
  - Create one project per problem on class assignments (e.g. HW1Prob2.ACC)
  - By default, all .acc and .vhd files are stored in C:\PeakVHDL
  - \- Creating a new project (.acc file)
    - Click NEW_PROJECT
    - Then, immediately save blank project via SAVE_PROJECT_AS
      - Default extension is .acc
  - \- Creating a new module (.vhd file)
    - Wizards are not recommended
      - (Module and Test Bench Wizards require major edits to customize)
    - Click NEW_MODULE
    - Click "Create Blank Module" (Don't use the Wizards)
    - Then, immediately save blank module via SAVE_MODULE_AS
      - Default extension is .vhd
    - Enter VHDL code for entity declaration and architecture body for module
    - When complete, SAVE_MODULE
  - \- Repeat above steps to enter additional modules (entities or .vhd files)
  - \- Reusing a previously written .vhd module (entity) by inserting it into a project
    - Click ADD_MODULE_TO_PROJECT
    - Select the .vhd file containing the entity to be added to the project
    - Repeat above steps to enter additional entities into the same project

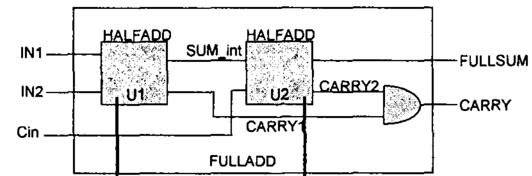
- Features of Editor and Compiler Environment
    - Color coded context sensitive editor
    - Hierarchy Browser shows relationships between design units in modules
        - Update hierarchy whenever new module added: File/Rebuild_Hierarchy
        - Can also rebuild hierarchy with one icon click
    - Compiler can JUMP_TO_ERROR
- Steps needed to perform a simulation:
    1) Compile
        - Highlight top-level module, then click toolbar icon COMPILE
        - PeakVHDL automatically compiles all lower-level modules as needed
    2) Link
        - Highlight top-level module, then click toolbar icon LINK
    3) Load (i.e. run simulator)
        - Highlight top-level module, then click toolbar icon LOAD_SIMULATOR
    - To perform all above three steps with just one click: LOAD_SIMULATOR
        - Compile and Link steps are automatically performed too
- Simulator
    - Can Select and Rearrange order of signals to be plotted
        - Upon simulator startup, user is asked which objects (signals) to display
            - Click ADD_PRIMARIES (to trace only top-level signals) or ADD_ALL
    - To run simulation click GO
        - Default time limit is 1000ns
        - A More typical value to use for the class projects will be about 100ns
        - STEP size can be changed (2ns or 5ns may be good step values to use)
    - Has Zoom capability
    - Displays VHDL source code of file executed
    - Signal values in left window augment waveform displays
- Additional Reference Resources
    - Use the On-line help within Peak VHDL
    - Read the PeakVHDL User Manual (.pdf) included on CD
        - Contains a brief introduction to VHDL
    - The link www.peakvhdl.com contains a web-based intro to VHDL
        - Click on VHDL Language Guide
    - The PEAK_FPGA folder has a help file VHDL_MUG
    - VHDL Web-based Tutorial is at: www.vhdl-online.de/~vhdl/tutorial/
    - An Interactive Tutorial can be downloaded by entering this web address
        - www.aldec.com/Free/Evita_VHDL.exe
    - Download "VHDL Entry Edition" and optional audio at
        - www.doulos.com/pacemakerframe.html

Creating Hierarchy

- VHDL supports hierarchical designs
  - A key technique to manage design complexity
- To create a hierarchical description:
  - First - compile all lower level blocks.
  - Second - in the top level description, write a *component declaration* for each lower-level block that will be included
  - Third - in the top level description, *instantiate* each instance of lower-level blocks and connect them together with *port map statements.*

1

Hierarchical Example: Full Adder




```
library ieee;
use ieee.std_logic_1164.all;

entity HALFADD is
port (A, B: in std_logic;
      SUM, CARRY: out
std_logic);
end HALFADD;

architecture FAST of HALFADD is
begin
   SUM <= A xor B;
   CARRY <= A and B;
end FAST;
```

Full Adder (cont.)

```
Library ieee;
use ieee.std_logic_1164.all;

entity FULLADD is
  port (IN1, IN2, Cin: in std_logic;
        FULLSUM, CARRY: out std_logic);
end FULLADD;

architecture STRUCTURAL of FULLADD is
  component HALFADD
    port (A, B: in std_logic;
          SUM, CARRY: out std_logic);
  end component;
  signal SUM_int, CARRY1, CARRY2: std_logic;
begin
  U1: HALFADD port map (IN1, IN2, SUM_int, CARRY1);
  U2: HALFADD port map (SUM_int, Cin, FULLSUM, CARRY2);
  CARRY <= CARRY1 or CARRY2;
end STRUCTURAL;
```

Component declaration before the begin statement

Component instantiations with port maps

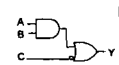
Notice the unique instance labels

3

Logical Operators: Example 1

```
Library ieee;
use ieee.std_logic_1164.all;
entity LOGIC is
port (A, B, C: in std_logic;
      Y, Z   : out std_logic);
end LOGIC;

architecture BEHAVE of LOGIC is
begin
   Y <= A and B;
   Z <= (A and B) or not C;
end BEHAVE;
```

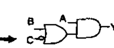
Use ( ) to prevent ambiguity. Otherwise could be:



(A and B) or not C

OR

A and (B or not C)



4

1

Logical Operators: Example 2

```
library ieee;
use ieee.std_logic_1164.all;

entity TRUTH_TABLE is
port (A, B, C: in std_logic;
      Y: out std_logic);
end TRUTH_TABLE;

architecture BEHAVE of TRUTH_TABLE is
begin
  Y <= ((not A and not B and not C)
or
          (not A and not B and C) or
          (not A and B and C) or
          (A and not B and C));
end BEHAVE;
```

A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

5

Processes

- Are concurrent statements
- May include *sequentially* executed statements
- Are executed (triggered) based on their *sensitivity list* OR *wait statement*
- Use global signals and local variables
- May not contain component instantiations
- May be combinatorial or clocked

6

Process With Sensitivity List

Process label (recommended)

Sensitivity list

```
PQR:process (A, B, C)
begin
 X <= A and B;
    if (B = '1') then
        Y <= A xor C;
    else
        Y <= (A xnor C);
    end if ;
end process PQR;
```

Statements executed sequentially

End statement matches process label

7

IF-THEN-ELSE Example

```
entity IF_MUX is
  port (C, D, E, F : in std_logic;
        S : in std_logic_vector(1 downto 0);
        POUT : out std_logic);
end IF_MUX;

architecture BEHAVE of IF_MUX is
begin
ONE: process (S, C, D, E, F)
begin
    if (S = "00") then
        POUT <= C;
      elsif (S = "01") then
        POUT <= D;
      elsif (S = "10") then
        POUT <= E;
      else  POUT <= F;
    end if ;
end process ONE;
end BEHAVE;
```

8

CASE Statement

```
case (selector) is
    when value =>
        --sequential statements
    when value1 | value2 | value3 =>
        --sequential statements
    when value1 to value2 =>
        --sequential statements
    when others =>
        --sequential statements
end case;
```

'when selector = value, then ...'

'when selector = value1 OR value2 OR value 3, then ..."

'when selector falls within the range from value1 to value2, then ..."

'when selector = any other value, then ...'

9

Concurrent Conditional Assignment: WHEN-ELSE and WITH-SELECT

```
Y <= IN1 when S = '0' else
     IN2 when S = '1' else
     '0';
```

```
Y <= "00" when COUNT >= 8
else
     "11";
```

"else" clause is required

"with" selection must be a signal, not an expression

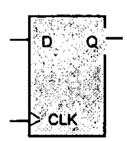
```
with DATAIN select   -- selected signal assignment
Y <= IN0 when "00",
     IN1 when "01",
     IN2 when "10",
     IN3 when others;
```

10

Describing Sequential Logic

- Use Processes and IF statements to describe sequential logic
- IF statement detects clock edge
  - Rising edge = `if (CLK'event and CLK='1')`
  - Falling edge = `if (CLK'event and CLK='0')`

```
architecture BEHAVE of DF is
begin
  INFER: process (CLK) begin
    if (CLK'event and CLK ='1') then
        Q  <= D;
    end if ;
  end process INFER;
end BEHAVE;
```

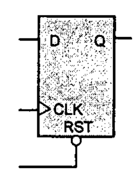


11

Sequential Logic: Example 1

```
architecture FLOP of DFALR is
begin
  INFER: process (CLK, RST)
  begin
    if (RST ='0') then
        Q <=  '0';
    elsif (CLK'event and CLK ='1') then
        Q  <= D;
    end if ;
  end process INFER;
end FLOP;
```

D flip-flop with asynchronous low reset and active high clock edge

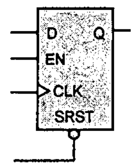


12

Sequential Logic: Example 2

```
architecture FLOP of DFSLRHE is
begin
  INFER: process (CLK)
  begin
    if (CLK'event and CLK ='1') then
      if (SRST = '0') then
        Q <= '0' ;
      elsif (EN = '1') then
        Q <= D;
      end if ;
    end if ;
  end process INFER;
end FLOP;
```

D flip-flop with synchronous low reset, active high enable and rising edge clock



Sequential Logic: Example 3

```
architecture FLOP of EN_FLOP is
begin
  INFER:process (CLK) begin
    if (CLK'event and CLK ='0') then
      if (EN = '0') then
        Q <= D;
      end if ;
    end if ;
  end process INFER;
end FLOP;
```

Will this model a positive edge or negative edge triggered flip-flop?

Is the enable synchronous or asynchronous?

Is the enable active high or active low?

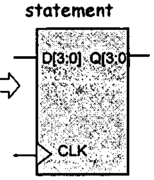
Is the reset synchronous or asynchronous?

Sequential Logic: Example 4

```
library ieee;
use ieee.std_logic_1164.all;
entity DF_4 is
port (D: in std_logic_vector(3 downto 0);
      CLK: in std_logic;
      Q: out std_logic_vector(3 downto 0));
end DF_4 ;
architecture FLOP of DF_4 is
begin
  INFER: process
  begin
    wait until (CLK'event and CLK ='1');
      Q <= D;
  end process INFER;
end FLOP;
```

Where's the sensitivity list?

4-bit register using WAIT statement



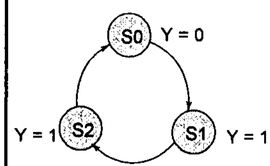
State Machines Overview

- **Typically include:**
  - At least 2 process statements (one MUST control the clocking)
  - IF-THEN-ELSE statements
  - CASE statements
  - User defined types to hold current state and next state
- **Transitions depend on current state and optionally, the inputs**
- **Outputs depend on:**
  - Current state (for **Moore machines)**
  - Current state plus inputs (for **Mealy machines)**

State Machines Template

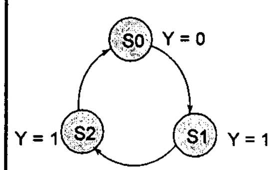
```
entity EXAMPLE is
 port( must include a clock);
end EXAMPLE;
architecture BEHAVE of EXAMPLE is
  type STATE is (S0, S1, S2);
  signal CURRENT_STATE, NEXT_STATE : STATE;
begin
Sequential: process (RST, CLK) -- First Process
  Uses IF-THEN-ELSE statements to change the state
 end process Sequential;

 NEXT_STATE_DECODE: process (RST, CLK) -- Second Process
  Uses CASE statements to define state transitions
 end process NEXT_STATE_DECODE;

 OUTPUT_DECODE: process (RST, CLK) -- Third Process(Mealy)
  Uses CASE statements to define the outputs
 end process OUTPUT_DECODE;
end BEHAVE;
```


Moore Example - First Process



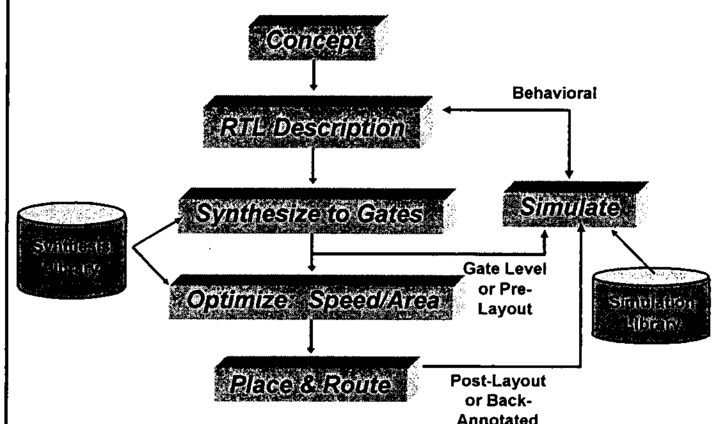

```
entity MOORE_SM is
 port( CLK, RST: in std_logic;
       Y: out std_logic);
 end MOORE_SM;

architecture BEHAVE of MOORE_SM is
  type STATE is (S0, S1, S2);
  signal CURRENT_STATE, NEXT_STATE : STATE;
begin
REG: process (RST, CLK)
 begin
  if (RST = '0') then
    CURRENT_STATE <= S0;
  elsif (CLK'event and CLK = '1') then
    CURRENT_STATE <= NEXT_STATE;
  end if;
 end process REG;
```


Moore Example - Second Process




```
COMB: process (CURRENT_STATE)
 begin
  case CURRENT_STATE is
   when S0 => NEXT_STATE <= S1;
              Y <= '0';
   when S1 => NEXT_STATE <= S2;
              Y <= '1';
   when S2 => NEXT_STATE <= S0;
              Y <= '1';
  end case;
 end process COMB;
end BEHAVE;
```

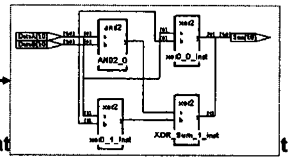

VHDL Design Flow and Levels of Simulation



What is Synthesis?

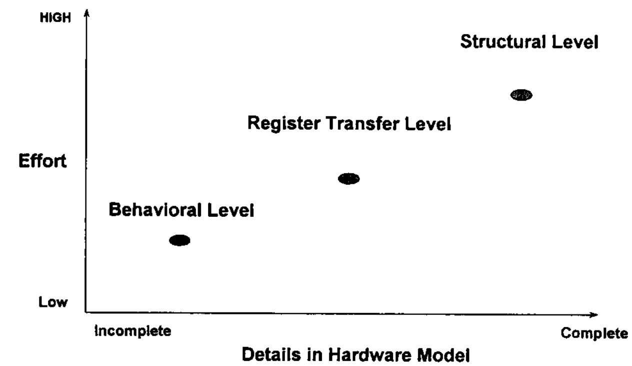
- The mapping of a behavioral description to a specific target technology

sum <= DataA + DataB; → Synthesis



- Accompanied by an optimizat t for:
  - Faster speed
  - Smaller area

Levels of Abstraction



- Data Representation

  The meaning of a set of data depends upon the interpretation of its bits

  e.g.) Data and instructions are differentiated only by their memory locations

  Data could be accidentally 'executed' with incorrect results.

- Non-Positional Number Systems

  Certain symbols represent different values

  Symbols are combined to form a certain number

  Numbers are generally hard to read and not very compact in representation

  ex) Roman numerals:

  I = 1, V = 5, X = 10, L = 50, C = 100, D= 500, M = 1000

  MDCXVII = 1617

  Difficult to formulate rules of arithmetic using non-positional number systems

- Positional Number Systems

  Most modern number systems are positional

  Position determines the magnitude of the number read

  e.g.) Decimal uses just ten symbols (0-9) to represent all numbers

  The meaning of a particular symbol is modified by the position in which it occurs

  $$N = D_{p-1} \times R^{p-1} + D_{p-2} \times R^{p-2} \ldots D_0 \times R^0 + D_{-1} \times R^{-1} + D_{-2} \times R^{-2} \ldots D_{-q} \times R^{-q}$$

  where R is the radix of the number system

  D are the digits of the number system (in which there are R allowable digits)

  p is the number of integral digits (to the left of the "point")

  q is the number of fractional digits (to the right of the "point")

  eg) $12.25_{10} = 1x10^1 + 2x10^0 + 2x10^{-1} + 5x10^{-2} = 1100.01_2 = 1x2^3 + 1x2^2 + 1x2^{-2}$

- Number Base

  Decimal system probably arose out of fingers on human hand

  Base 2 is the lowest useful base

  Base 0 does not exist; Base 1 has only one symbol, the digit 0

  Binary system seems wasteful of space

  e.g.) Takes 4 bits to represent 1 decimal digit (BCD)

  But binary can be very easily represented in electronic form in the computer

  Binary one / zero can represent TRUE and FALSE of formal logic

  Binary numbers easy to mechanize reliably (H/W tolerance, aging, noise)

  Binary arithmetic is simple

  Binary is firmly established as fundamental system for data rep. in computers

  Working with numbers in different bases requires conversion between bases

- Binary Representations and Operations
  - Fundamentally, operations are the same as in decimal
  - But the simplicity of binary numbers enable further simplifications

- Negative Numbers
  - Possible to directly subtract binary numbers in manner similar to decimal system
    - Column-by-column with borrows if necessary
    - This procedure is not used in computer designs
  - To perform a subtraction, one can change the sign of the subtrahend and add
  - Need to find the proper representation for negative numbers to facilitate this
  - Three main representation schemes for negative numbers:
    - Signed-Magnitude
    - 1's complement
    - 2's complement

- Signed-Magnitude Representation
  - A plus or minus sign precedes magnitude of number
  - Leftmost bit of the number tells whether number is positive or negative
    - Positive number = 0; Negative number = 1
  - Remaining bits give the absolute value or magnitude of the number
  - Most familiar way for dealing with negative numbers (similar to decimal)
  - If b is added to -b, then sum forms 0
    - In other words, -b is the additive inverse of b
  - \- First Problem:
    - Addition and Subtraction are separate operations
    - Need both an adder and subtractor (even to just do addition)
    - Actual operation must be determined by logical test of sign bits and operands
      - e.g.) A - B can involve the following:
        - A and B positive, $|A| > |B|$, subtract $|B|$ from $|A|$
        - A and B positive, $|A| < |B|$, subtract $|A|$ from $|B|$, set ans. sign neg
        - A positive and B negative, add $|A|$ and $|B|$, answer is positive
        - A negative and B positive, add $|A|$ and $|B|$, set answer's sign to neg
        - A and B negative, $|A| > |B|$, subtract $|B|$ from $|A|$, set answer neg
        - A and B negative, $|A| < |B|$, subtract $|A|$ from $|B|$, answer positive
    - Tolerable to use different circuitry for addition and subtraction
      - Just need to use different machine language instructions
    - But difficult to design computer that either +/- depending on signs of nums
      - Want operation to depend on instruction; not on the data itself

- Second Problem:
  - No unique 0 (e.g. 00000 and 10000 both equal 0)
    - Harder to test accumulator to see if it contains a pos. or neg. number
      - Generally, 0 is thought of as being positive
    - Also, 'Dirty' zero gets in the way when generating an address
      - ex.) Effective Address = Base Address + Relative Address
- Advantages:
  - Easy to compute negative of a number
    - Just invert MSB (sign bit)
  - Easy to read a negative sign and magnitude to find decimal equivalent
    - ex) With 4 bits total, in sign-magnitude: 6 = 0110 ; -6 = 1110
- Range:
  - With N bits, Range of sign-magnitude is $-(2^{N-1} - 1)$ to $+(2^{N-1} - 1)$

• Complement Notation

Analogy: Car odometer

If meter 'ticks' forward, it performs addition

If meter 'ticks' backward, it performs subtraction

00004	⇑ Addition
00003	
00002	
00001	1 'tick' forward from 0 (+1)
00000	
99999	1 'tick' backward from 0 (-1)
99998	
99997	
99996	⇓ Subtraction

Note that 3 'ticks' backward corresponds to 100000 - 3 = $10^5 - 3$ = 99997

So 99997 corresponds to -3 (i.e. it is the additive inverse of 3)

That is, we can 'subtract' by turning meter only forward

To perform 4 - 3, we can do the following:

$$\begin{array}{rr} & 00004 \\ + & \underline{99997} \\ 1 & 00001 \end{array}$$

This is equivalent to $4 + (10^5 - 3) = (10^5) + 4 - 3$

Therefore, ignoring the carry (the 10^5 term) effectively performs 4 - 3 = 1

99997 is called the 10's complement of 3.

Thus, subtraction can be done by representing negative numbers as comps

Two types in Binary: 1's complement
2's complement

- **Diminished Radix Complement (1's Complement)**

For a positive number, One's complement is the binary value of that number

Leading bit (MSB) is always 0

For a negative number x in N binary digits, One's complement is: $(2^N - |x|) - 1$

Leading bit (MSB) is always 1

The one's complement N' is the additive inverse of a positive binary number N

- Advantages:

Trivial conversion between positive and negative forms

Simply replace all 1's by 0's and all 0's by 1's

Only need a bit-by-bit (parallel) negation operation

e.g.) If N = 4 bits, +7 = 0111
-7 = 1000

Addition and Subtraction can be performed using the same operations

To subtract B from A, form the 1's comp of B and add to A

Add the two numbers without regard to their signs

Add the carry digit produced by the MSB to the result

Mechanics of the operation does not depend on the signs of the operands

Sign bit is included in the addition, which is not the case in sign-magnitude

End around carry needed to solve problem with addition of unlike signs

```
ex)  +3      0011
     -6      1001
             ----
             1100 (1's comp) = -0011 =  -3 (right)
```

No end-around carry needed (no carry produced out of MSB)

```
ex)  -3      1100
     +6      0110
             ----
          1  0010 (1's comp) = +0010 = +2 (wrong)
```

End-Around carry needed (carry produced out of MSB)

With end around carry, answer becomes: 0011 (1's comp) = + 3 (right)

Two stage process: Need to feed carry out of MSB into second add stage

- Problems:

Requires two additions because of end-around carry

Also, two representations of 0 exist (0000 and 1111)

- Range:

With N total bits, range of 1's comp is: $-(2^{N-1} - 1)$ to $+(2^{N-1} - 1)$

Few uses of 1's complement on modern computers

Mainly discussed to form the basis for 2's complement notation

- Radix complement (2's complement)

  For a positive number, the 2's complement is the binary value of that number

    Leading bit (MSB) is always 0

  For a negative number x, 2's complement of x in N bits is: $2^N - |x|$

    Leading bit (MSB) is always 1

    Two's comp is equivalent to (1's comp of x) + 1

  The two's complement N' is the additive inverse of a positive binary number N

ex)	-6 in 2's complement:	0110	+6
		1001	1's comp
		1010	add 1
	Check, by converting back:	1010	-6
		0101	1's comp
		0110	add 1

Addition and Subtraction can be performed using the same operations

  To subtract B from A, form the 2's comp of B and add to A

  Add the two numbers without regard to their signs

  Discard the carry digit produced by the MSB

Mechanics of the operation does not depend on the signs of the operands

Sign bit is included in the addition, which is not the case in sign-magnitude

```
+3      0011
-6      1010
        ----
        1101 (2's comp) = -(0010 + 1) = -0011 =  -3 (right)

-3      1101
+6      0110
        ----
     1  0011 (2's comp) = +0011 = +3 (right)
```

Just ignore carry out of MSB

  Connect several adders together in ripple carry manner

  Carry into LSB set to 0

- Advantages:

  Unique representation of 0

-0 in 2's complement:		0000	+0
		1111	1's comp
	1	0000	add 1

  Addition does not require the extra step of end-around carry (as in 1's comp)

- Disadvantages:
  - Takes two steps to complement a number
  - 2's complement negative numbers not easy to interpret in decimal
    - e.g.) -9 = 111110111 (2's comp)

- Range:
  - With N bits in 2's comp, range is: $-(2^{N-1})$ to $+(2^{N-1} - 1)$
  - Note: Two's comp range is not symmetrical
    - Using 4 bits, 16 numbers are representable:
      - Eight negative, one zero, and seven positive

2's Complement is widely used for arithmetic on modern computers

• **Summary**

Most computers subtract by finding the negative (complement) and adding

Reduces design complexity & cost, since separate circuitry not needed to subtract

For all three representation schemes:

- Positive numbers are identical and the same as their ordinary binary rep
- The leftmost bit (MSB) of a negative number will be 1

- From the perspective of addition:
  - Signed-Magnitude is most difficult to implement
    - Operation dependent on signs and magnitudes of operands
  - 1's complement is considerably simpler
    - Can subtract by adding but requires end-around carry addition cycle
  - 2's complement is the simplest since it lacks the end-around carry
    - The need to add 1 is known at onset, so it can be done in same add step
- From the perspective of negation:
  - Sign-Magnitude is the simplest
    - Invert sign bit (MSB) only
  - 1's complement is next
    - Invert all bits
  - 2's complement is the most expensive
    - Invert all bits and add one

• An ideal number representation system would have:

1) Only one representation for zero

2) Exactly as many positive numbers as negative numbers

But both constraints together require an odd number of representable members

For binary system, there are always an even number of representable members

- **Real Numbers**

In addition to a sign, a number may have a decimal (binary) point

Enables number to represent fractional quantities

The binary point is really an imaginary position in the accumulator

Not explicitly stored with number (as the sign bit was)

Format:

$$a_2a_1a_0.f_1f_2f_3 \quad = \quad a_2r^2a_1r^1a_0r^0.f_1r^{-1}f_2r^{-2}$$

e.g.) 1011.1001 = 8 + 2 + 1 +1/2 + 1/16 = 11 9/16

Note:

Some numbers are not 'perfectly' representable

ex) 1/3 in decimal is 0.333333333......

1/3 in binary is 0.01010101......

ex) 1/5 in decimal is 0.2

1/5 in binary is 0.0011001100110011 00.....

There are two ways of specifying the position of a binary point in a real number

Fixed position

Floating point

Fixed Point Numbers

An invisible "binary" point always occurs at some fixed place

Sign bit (if present) is always to the left of the radix point

e.g.) With 36 bit signed word, assume point is in the middle

So: sign, 17 bits to left, point, 18 bits to right

Digits to left of point represent integer part (powers of 1, 2, 4 ...)

Magnitude of a number is limited by the number of bits allocated to left part

Fixed point at extreme right would be domain of integer numbers

Digits to right of point represent fractional part (powers of 1/2: 1/2, 1/4, 1/8 ...)

Accuracy of a number is limited by the number of bits allocated to right part

- Advantages

Most efficient form of real number since point position is uniform for all nums

Not necessary to represent point position

Efficient in its use of storage and data value manipulation

- Disadvantages

Severely limits the magnitude range of numbers that can be represented

e.g.) For 36 bit signed word: Smallest num is 2^{-18}; Largest num is 2^{+17}

It is inconvenient to represent large binary numbers in fixed positional binary

Large number of bits required

Few computers use fixed point representation for real numbers

- Floating Point Numbers

  Position of point 'floats' (can move) for each num and is not uniform for all words

  Consists of two parts: mantissa (m) and exponent (e)

  Number represented is = m x R^e

  Number of bits in mantissa determines the number of significant figures

  The exponent has greatest impact in determining the range of numbers

  Equivalent to scientific notation

  Stores each number as a fraction and an exponent

  Fraction: between 0.0 and 1.0

  Exponent: Specifies the power of 2 (R^e term)

  Radix point in the mantissa m is determined by the magnitude of the exponent e

  eg) Shift mantissa k places to left (right) and decrement (increment) expo by k

- Normalized Representation:

  Numbers are stored such that mantissa is between 1/2 and 1.0 (non-inclusive)

  That is, the most significant position of mantissa must be nonzero

  - Advantages:

    There is a unique normalized representation of a real number

    Normalized representation provides the maximum precision for a FLP num

    Moves number as far as possible to the left (removes leading zeros)

    Makes room for the inclusion of as many lsb's as possible

  - Disadvantages:

    Floating point arithmetic operations require the most complex H/W implementation

    Normalized floating point addition / subtraction requires an extra step:

    The two operands must have identical exponents

    Align fractions by shifting right the one with the smaller exponent

    To minimize loss of high order significant digits when shifting mantissa,

    Algebraically smaller exponent should be increased in pos direction

    Better to drop a digit off the right end of the word instead of left end

ex)	Wrong:	x = .101101E6	Right:	y = .101010E2
		x = .010000E2		y = .000010E6

- Overhead and Truncation associated with normalized representation

  An empirical scientific study:

  D. Sweeney [1965] analyzed floating point additions for various programs

  Studied about 10 million instructions

  Only about 10% of total instructions executed were floating point additions

  Results on Pre-addition alignment and Post-addition normalization shifts:

	Shift Distance (bits)	Percent of Shifts
Alignment	0	32%
	1-4	34%
	5+	34%
Normalization	Overflow	20%
	0	60%
	1-4	15%
	5+	5%

Conclusion:

Majority of the time, exponents are quite close to each other

About 2/3 of the cases needed alignment shifts of 4 bits or less

In 1/3 of the cases, the exponents were identical

- Phantom 1

Normalized notation allows for an optional memory compaction scheme

Since the first bit of each fraction is 1, this bit is not stored internally

Saves one bit of memory

e.g.) Can obtain 9 bits of precision using only 8 bits of memory

So .10101010(1) can be stored as .01010101

Requires Phantom bit restoration or built-in 'hardwired' leading 1

Also referred to as Hidden bit scheme

- Excess Exponent

Generally, an exponent e can be either a positive or negative integer

Exponent of two FLP numbers need to be compared and equalized before add/sub

Want to simplify comparison operation of e1, e2 without involving signs of expos

Convert all exponents to positive integers by adding a positive constant to each e

Forms a biased exponent

Popular choice for bias constant is the magnitude of most negative exponent

True exponent value is = Biased Exponent value - 2^{q-1} (for a q bit exponent)

Example:

Excess 64 means that the exponent is stored as 64 larger than true value

Using 7 bits, 128 values are representable

The true exponent's representable range is from -64 to +63

So true value e = -64 is stored as biased value = 0; true +63 = biased 127

The ambiguity of +0 and -0 is replaced by an extra exponent value

Most computers use a biased exponent

- **Floating Point (FLTP) Numbers**

  Provides important conveniences to users and programmers

  - Flexibility in location of the binary point provides freedom to users
  - User need not worry about manipulating scale factors, aligning points, etc.

  Enables:

  - Fractions and mixed numbers to be used as easily as integers
  - Very small and very large quantities to be represented
    - e.g.) Planck's constant (6.625 E-27)
      Avogadro's number (6.024 E23)

  Designer of a FLTP representation must compromise between accuracy and range

  - Allocating more bits to significand enhances accuracy
  - Increasing size of exponent field will extend the range

  Designer can choose the Format and Representation for each field of the FLTP num

  - Sign (i.e., how to represent negative numbers)
  - Mantissa (e.g., Normalized, Phantom 1)
  - Exponent and its sign (e.g., Biased or other representation scheme)

  Designer must also account for exceptional events as well

  Programmers need to know when they have computed:

  - A value too large to be represented (Infinity)
  - A fraction which is non-zero, but too small to be represented (Underflow)
  - Erroneous Computations (e.g., Divide by Zero)
  - Invalid Operations (e.g., SQRT(-Num))

  These events could result in a program giving incorrect results unless trapped

  Implementation of FLTP can involve considerable circuitry or software complexity

  Key Characteristics which Define a Floating Point Representation:

  - Precision: How Accurate a floating point value can be
    - Measure of the Resolution of the system
    - Defined as the number of bits in the significand
  - Gap: Difference between two adjacent values
    - Influenced by the value of the exponent
    - e.g.) Assume an 8-bit significand and value of $.1011\ 1010 \times 2^3$
      - Adjacent values are $.1011\ 1001 \times 2^3$ and $.1011\ 1011 \times 2^3$
      - Gap is $.0000\ 0001 \times 2^3 = 2^5$
  - Range: Span of numbers between the smallest and largest possible values

- **Precision is affected by:**
  - Number of Guard Bits Used
  - Rounding Policy

- **Guard Digits**

Operations on FLTP nums can produce results longer than original source operands.
The lower-value bits of the significand are called Excess Low-order Bits (ELBs)
These ELBs can be used: 1) To support postnormalization with guard bits
2) To support rounding with round and sticky bits
Guard Bits are Additional bits used during the intermediate mathematical steps
Consist of the low-order bits to the right of the N significant bits of mantissa
Holds the bits of the aligned operand that shifted right during exponent adjust
These digits can be used in conjunction with any rounding scheme

Permits maximum accuracy to be retained in the result
Guard digits are only needed within the arithmetic processor
Yields more precise results without having to maintain double precision throughout the entire machine.

Ex) Guard Digit Usage in Subtraction of two FLTP numbers with different exponents

- Case 1: 3-bit Arithmetic unit (No Guard digits) and 3-bit result

```
  1.10 E1                              1.10 E1  (3 Decimal)
- 1.01 E0       After Alignment =>   - 0.10 E1  (1.25 Decimal)
                                       1.00 E1  (2 Decimal)
```

- Case 2: 4-bit Arithmetic unit (1 Guard digit), 3-bit result

```
  1.10 E1                              1.100  E1  (3 Decimal)
- 1.01 E0       After Alignment =>   - 0.101  E1  (1.25 Decimal)
                                       0.111  E1
                Post Alignment =>      1.11   E0  (1.75 Decimal)
```

To Preserve Maximal Accuracy:
One may at first think that the number of Guard digits needed = bits in mantissa
But, It has been proven [YOH 73] that two guard digits are always sufficient
Then, to insure unbiased rounding, a third Sticky bit is also needed
These three bits will allow the machine to get the same results as if the intermediate results were calculated to infinite precision and then rounded

- Sticky Bit

The Third, Lowest-Order Guard Bit
Set to 1 if a 1 is ever shifted into it during fraction alignment
Useful in some rounding methods to distinguish the "tie" case
Need to know if any 1s were shifted right during alignment
ex.) In decimal .50XXXXX
IF XXXXX is 00000, S will be 0 and a tie situation exists
IF XXXXX is not all 0's, S will not be 0 and rounding should be up

- **Rounding Schemes**

For Integers:

The standard arithmetic operations are well defined

Resulting value is unambiguous (it is either representable or it is not)

Test for overflow is conceptually simple

IF mathematical result is outside the range of integers, THEN Overflow

For Floating Point Numbers:

Situation is more complex

For some valid operations, the mathematical result is non-representable

e.g.) Consider two adjacent FLTP numbers

There are no representable values between them

So computing their average will generate a non-representable result

Note that this is different from overflow

There is simply no representation for the number given the FLTP scheme being used.

Desired result is that the "closest" representable number be assigned

Also, some operations (e.g. multiply) produce results larger than register size

Computed digits may exceed total number of digits allowed by the format

Extra digits must be disposed before the final results are stored in memory

Rounding usually limits the precision of the number

One or more of the least-significant digits in a number are deleted

Then, the retained portion is adjusted in accordance with some rule

A binary fraction is shortened to create a new fraction, its approximation

The selection and formation of this approximation is referred to as Rounding

Purpose is to reduce the number of digits in the number so that it will "fit".

Regardless of the Rounding policy chosen, a small amount of error is introduced

Undesirable effects of Rounding include:

- Errors that can accumulate over time.

  For algorithms that iterate, small errors can become significant.

- Operations that yield different results if performed in a different order

  e.g.) Floating point addition may not be commutative (a + b) /= (b + a)

- Difficult exact comparison of two FLTP numbers.

  Thus, successive approximation algorithms must specify an Epsilon

  Otherwise, "convergence" may never be reached

Selection of Rounding Scheme needs to factor into account:

- Accuracy of results (numerical considerations)
- Cost and Speed of implementation (machine considerations)

- We examine Three Different Rounding Algorithms:
  - Truncation
  - Rounding (Ordinary)
  - Round to Nearest Even

- Truncation (also called Chopping)
  - Simplest Rounding Strategy: Just Discard the Excess digits (ELBs)
    - Removes the additional bits and makes no change to the remaining bits
    - e.g.) Decimal 2.99 Truncated to 1-Digit will become 2
  - Fast and Requires No additional hardware
  - May introduce a significant error
    - Could result in the loss of several bits during FLTP Add/Sub Alignment
    - Error ranges from 0 to almost 1 in the LSB of the retained bits (the ULP)
  - Ideal Rounding Line Shows the True Accurate Value of the Number being Rounded
    - By definition, Rounding implies a deviation about this Line
    - Goal is to: Minimize Variance from Ideal Rounding Line
      Maximize Symmetry around Ideal Rounding Line (IRL)
  - A Rounding Scheme may demonstrate some Positive or Negative Bias relative to IRL
  - Unbiased Approximations for Rounding are Preferred
    - Individual Errors should be symmetrically distributed over the error range
    - Given many operands and operations, positive and negative errors will cancel
    - Long, complex computations will statistically have a high probability of accuracy
  - Truncation has a Strongly Negative Bias
    - Truncation function lies entirely below the Ideal Rounding line
    - Touches ideal Rounding line only where there is no truncation error

- Rounding (Ordinary or "IRS" Rounding)
  - More complicated than Truncation: Select the nearest adjacent representable value
  - Simple, Ordinary Rounding similar to that used for decimal numbers
    - If ELDs are less than 1/2 of the result's LSD, round down; otherwise, round up
      - e.g.) 0.0 thru 0.4 would round to 0
      - e.g.) 0.5 thru 0.9 would round to 1
  - Nearly symmetric with respect to the Ideal Rounding line except for ties (.5)
    - If halfway points are always rounded up, a slight positive bias will occur
    - IRS benefits from increased taxes
  - Process involves mechanically adding 1/2 the ULP and chopping the fraction
    - e.g.) Decimal 2.9 + .5 would yield 3.49 Chopped to 3
      Decimal 2.5 + .5 would yield 3.00 Chopped to 3
      Decimal 2.4 + .5 would yield 2.90 Chopped to 2
  - In binary, a full add time is required since carry from the LSB may propagate

- Round-to-Nearest Even (or Odd)

  Same as Ordinary Rounding typically used by most people in Decimal,

  EXCEPT that the tie breaking rule is modified.

  Addresses the slight positive bias problem of Ordinary Rounding due to the tie cases

  Two variations and methods to handle ties:

  Round to Even: Choose the representation whose least-significant bit is 0

  Round to Odd: Choose the representation whose least-significant bit is 1

  By addressing ties in this manner, they will be alternately rounded up and down

  This will Cancel all Bias

  IEEE Default is Round to Nearest with ties going to the representation with LSB 0

  That is, for Tie situation, IEEE 754 chooses the nearest Even number

  Achieves the closest, unbiased approximation to the number being truncated

  Error range is approximately -1/2 to +1/2 in the LSB of the retained bits

  Most Expensive, but Most commonly used technique and generally recommended

  Creates fewer problems with systematic error than always rounding ties up

  ex) Round to Nearest Even in Decimal

  38.5 Rounds Down to 38

  39.5 Rounds Up to 40

  ex) Round to Nearest Even in Binary with 2 ELBs

	ELBs		
Significand	0x	10	11
1.1000	1.1000	1.1000	1.1001
1.0001	1.0001	1.0010	1.0010

  Side Note: Why did IEEE use Even instead of Odd ?

  Unbiased Rounding can also be obtained by Rounding to Odd instead

  ex) 1.95 Rounds to 1.9

  2.05 Rounds to 2.1

  But, generally, "Nicer" Integer numbers will result with Round to Even

  ex) 1.95 Rounds to 2

  2.05 Rounds to 2

- IEEE 754 Floating Point Standard
  - History:

    Committee Meetings were started in 1977 to draft a Floating Point Standard

    Several proposals were submitted to the IEEE Computer Society

    Most complete specification was from Kahan, Conen, Stone

    Referred to as the "KCS" or "Kahan" Proposal

- **IEEE 754 Motivation:**

  In the 1970's, Manufacturers were Widely Implementing Floating Point Operations

  However, each Manufacturer's FLTP Format was Incompatible with the others

  Every Manufacturer had a FLTP, but every Implementation was Different

  Thus, the same program could generate different results on different computers

  Numerical Software Programs and their Data were not Portable

  Main motivation for standard was to:

  Achieve Portability of numerically oriented programs across computer systems

  Encourage the development of high-quality numerical software

  Particularly important in microprocessor and small machine environments

  Individual manufacturers are not likely to develop extensive numerical routines

  Goal was to:

  - Use the same FLTP Format for all computers

    To ensure consistent computations and data storage

  - Ensure that the best possible standard be adopted for a given number of bits

    Obtain high accuracy (i.e., correct results to within 1/2 of the LSB)

- **IEEE 754 Standard defines every aspect relating to the processing of FLTP numbers**

  Specifies:

  1) Format of the Floating-point numbers
  2) Accuracy of the arithmetic computations
  3) Handling of Exception Conditions

- **IEEE 754 Format:**

  Two basic formats for Floating Point Operands: 32 bits and 64 bits

  Single precision format consists of 32 bits: (ELBs only used for intermediate results)

Sign	Exponent	1.Significand	*ELBs*
1 bit	8 bits	23 bit mantissa fraction	*G,R,S bits*

The designers of IEEE FLTP Standard chose to use a mixed number for mantissa

Format is 1.xxxx as opposed to traditional fraction format of .1xxxx

This "Significand" has one bit position to the *left* of the radix point

So 1 <= (IEEE Significand) < 2

Since a normalized significand has a 1 in the MSB, it need not be explicitly stored

Using hidden bit scheme, only the fractional part of the significand is stored

Hence, the significand is actually 24 bits long in single precision

Implied 1 and a 23-bit fraction

Exponent is stored in excess 127 representation to eliminate the exponent sign

End values of exponent (0 and 255) are reserved for special values

Therefore, usable range of exponent is 1 to 254 (-126 through +127)

- **IEEE 754 Accuracy:**

  Standard requires accurate arithmetic results

  Requires that three bits to the right of the 23 bits of the mantissa be maintained

  1) Guard bit
  2) Round bit
  3) Sticky bit

  The Guard and Round bits are just the 2 most significant bits of the ELBs

  For computations, the Guard and Round bits act as extra bits of precision

  The third, Sticky bit is the logical OR of all bits beyond the first two (G & R)

  Initialized to 0

  If a 1 is shifted through this position, it becomes a 1 and retains that value

  These three bits participate in all mathematical computations and enable:

  - Results of single operations to be computed accurate to within 1/2 of the LSB
  - Hardware support of postnormalization and various rounding schemes
  - Improved Accuracy of results without the full overhead of Double Precision

- **IEEE 754 Exception Conditions:**

  Five Exceptions are detected:

  1) Invalid Operation
  2) Division by Zero (Special case of Overflow)
  3) Overflow
  4) Underflow
  5) Inexact (Signaled if the Rounded Result of an operation is not exact)

  Standard also defines special representations for 0, Infinity, and for Not-A-Number

  - Since the number Zero has only 0s in its siginificand, it cannot be normalized

    For this reason, a Special Value is assigned to Zero

    Sign = Significand = Exponent = 0 (i.e., all 32 bits zero)

    Algorithms must explicitly check for Zero and treat it as a Special Case

  - Positive & Negative Infinity also have Special Representations and Treatment

    Sign = 0 or 1; Significand = 0; Exponent = 1111 1111

    Exponent overflow in FLTP arithmetic is the primary concern

    Exponent exceeds the most positive number or most negative number

    Propagation Rule for Infinity:

    Division by +-Infinity will generate a +-Zero

  Note: These special meanings for the extreme values of the exponent, (all 8 bits 0 for Zero; all 8 bits 1 for Infinity) decreases the FLTP range.

  Smallest number has exponent of e = -126 (E = 00000001)

  Largest number has exponent of e = +127 (E = 11111110)

- Operations with no mathematical interpretation produce Not-A-Number (NAN)
    - SQRT(-1), Infinity/Infinity, (-Infinity)+(Infinity), 0 * Infinity, Uninitialized Var
    - Not-A-Number is represented with:
        - Sign = 0 or 1; Significand non Zero; Exponent = 1111 1111
        - There are 2^{23} = 8 Million nonzero significand values
        - Information can be communicated to the user in the significand field
            - For example, the line number of the offending line of code
    - NAN is further refined into SNAN (Signal) and QNAN (Quiet)
    - SNAN: Signals an invalid operation interrupt if result is subsequently used
    - QNAN: Quiet allows result to propagate through subsequent operations
        - For example, instead of interrupting on a divide by 0, the software
            - can set the result to a bit pattern representing +-Infinity.
        - Then, when user generates report, an Infinity symbol can be printed
    - NANs allows programmers to postpone some tests until convenient
    - Propagation Rules for NAN:
        - NAN is considered a valid result of an arithmetic operation
        - So, need to specify what to do if a NAN appears as an input operand
        - Generally, NAN will simply propagate through the arithmetic operation
            - ex.) 5 + NAN = NAN, 3 * NAN = NAN, SQRT(NAN) = NAN

- **Underflow**

  Occurs when a result is too close to zero to be represented.

  - IEEE 754 Utilizes Gradual Underflow (also called Denorms or Subnormals)
    - A technique for increasing the range of representable numbers near zero
    - Gives up precision gradually
        - Effectively, the very small numbers are spread farther apart
        - Extends numbers much closer to zero rather than having a gap between 0 and the smallest normalized number.
    - Allows every last bit of precision to be obtained from a FLTP operation
    - Gradual Underflow is the most contested part of the IEEE 754 Format
        - Main objection lies in the increased implementation cost in hardware
        - Easier and Faster to simply generate a 0 on underflow
    - Accomplished by denormalizing the number
        - If E=0000 0000 (treated as -126), interpretation of the mantissa is modified
            - Normally, mantissa can never be smaller than 1.0 in IEEE 754
            - For gradual underflow, hidden bit is construed to be 0 instead of 1
                - The 23 bits of the significand represent the entire value
            - So exponent range can be extended up to 23 values -126-23 = -149
    - Generally not possible to reverse operations that resulted in a denormalization

Ex.) Smallest positive normalized value would be 1.0×2^{-126} represented as:

Exponent (-126)	Significand (1.0 with implied 1)
0000 0001	(1).000 0000 0000 0000 0000 0000

First denormalized value would be $0.1 \times 2^{-126} = 2^{-127}$ represented as:

Exponent (-126)	Significand (No implied leading 1)
0000 0000	(0).100 0000 0000 0000 0000 0000

Last denormalized single precision value would be 2^{-149} represented as:

Exponent (-126)	Significand (No implied leading 1)
0000 0000	(0).000 0000 0000 0000 0000 0001

- IEEE 754 Advantages
  - Format with exponent before the significand simplifies Sorting of FLTP numbers
    - Allows integer comparison instructions to be used
    - Numbers with bigger exponents look larger than numbers with smaller expos
  - Excess 127 Exponent Representation enables:
    - Efficient comparison of relative sizes of two floating point numbers.
    - The reciprocal of all normalized numbers to be computed without overflow.
  - IEEE FLTP Standard can be realized in Hardware, Software, or any Combination
  - The IEEE Data Type has become a successful standard and met it's main goals
    - Today, IEEE 754 is used in virtually all CPUs that have a FLTP capability
  - Implementation cost of IEEE is not much more than less-comprehensive systems
    - Smart Design Rationale led to Good Performance with Low Cost
    - Specifies features most useful in practical application programs
  - All computers conforming to this standard (almost) always compute same result
    - IEEE 754 specifies Round Toward Nearest Even as Default
      - However, user can override and use an alternate Rounding Scheme
    - Variation could be caused by different user-selectable rounding scheme

- Moral of the Story
  - Mapping b/w Mathematical Number System & Mechanized FLTP system is not perfect
  - The need to perform Rounding in FLTP operations results in Errors
    - Occurs even with the best rounding scheme
    - Accumulation of errors will depend on the particular set of operations performed
      - That is, the total error is application specific
  - Accuracy of the results obtained via FLTP is limited even if
    - the intermediate results which are calculated are accurate.
  - A significand of finite length can never exactly represent every possible number
    - Roundoff introduces some error into almost all FLTP operations
    - Programmers must remember these limits and write programs accordingly

- Summary for Floating Point Numbers

  Floating point number systems contain only a finite collection of values

  These values are not uniformly distributed on the real line.

  eg.) .1E-1 has granularity of .25 ; .1E1 has granularity of 1.0

  However, in all machine computations, such systems represent entire number line

- Range: Non-Normalized

  Largest mantissa = $(1 - 2^{-Nm})$

  Largest exponent = $(2^{Ne-1} - 1)$

  Smallest mantissa = 2^{-Nm}

  Smallest exponent = 2^{-Ne-1}

  Where Nm and Ne are number of bits in mantissa and exponent, respectively

  Example: With a 7 bit mantissa and a 7 bit exponent using excess 64 notation:

  (2^{-7}) E (2^{-6}) = .0000001E-64 to $(1 - 2^{-7})$ E $(2^{6} - 1)$ = .1111111E63

- Range: Normalized

  Same for non-normalized except for smallest mantissa

  Smallest mantissa = 2^{-1}

  Example: With a 7 bit mantissa and a 7 bit exponent using excess 64 notation:

  (2^{-1}) E (2^{-6}) = .1000000E-64 to $(1 - 2^{-7})$ E $(2^{6} - 1)$ = .1111111E63

  Note:

  Normalized form excludes some numbers and reduces range

- Finite Precision Numbers

  Usually not a problem with 'paper and pencil'; however, important in computers

  However, for any computer, there is always a finite number of 'column' devices

  The amount of memory available for storing a number is fixed at design time

  Therefore, in a computer, all numbers are of finite-precision and an approximation

  e.g.) Problems arise in the representation of irrational numbers

  It is possible for a computer in perfect working order to produce 'wrong' answers

  Error arises as a logical consequence of its finite nature

  Algebra of finite-numbers is different from normal, infinite precision algebra

  Example: Assume largest number possible is 3 decimal digits (999)

  - Associative law:

    a + (b - c) = (a + b) - c where a = 700; b = 400; c = 300

    700 + (100) = ($\underline{1100}$) - 300

  - Distributive law:

    a x (b - c) = a x b - a x c where a = 5; b = 200; c = 100

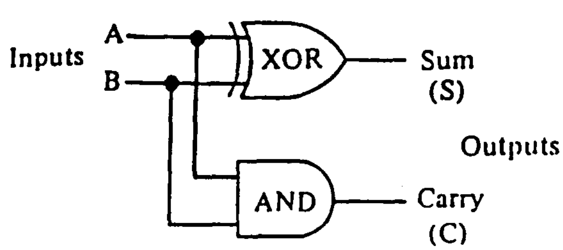
    5 x (100) = $\underline{1000}$ - 500

  Important to factor limited range/precision into account, especially in compilers

Arithmetic

- **Arithmetic**
  - Addition, subtraction, multiplication, and division are "bread and butter" operations
  - Responsible for the bulk of activity involved in processing computer data
  - Higher order numerical analysis functions are based on the four basic operations
  - Even important for "non-numeric" programs
    - Word processors must compute proportional type spacing for justified margins
    - Drawing software must compute trig functions when scaling, rotating
  - Just executing a program requires addition
    - Need to increment program counter and do address calculations
  - Arithmetic operations always tested as part of machine performance figures
  - So, speeding up these four basic operations can have tremendous impact on perf.
  - Time to perform math ops becomes more significant with larger word machines
    - Higher precision operands
    - Larger addresses for increased (virtual) memory
  - \- Operating Speed
    - Delay through a network of gates is dependent on the electronic technology
    - We assume that technology is frozen (e.g., we use the best gates available)
    - Once technology family is chosen, worst case delay is that of the longest path
  - Issue is: How do we modify architecture to reduce worst case delay
  - Optimized arithmetic serves as a low level starting point for speeding up computer
  - This section concentrates on architectures and algorithms for fast arithmetic

- **Addition**
  - Integer addition is the simplest arithmetic operation and the most important
  - Therefore, an n-bit adder with carry in and carry out is a key building block
  - Addend and Augend are added together to produce a sum (and a carry)
  - Using 2's complement notation, discussion of addition includes subtraction
  - Basic Adder Building Blocks:
    - \- Half Adder
      - Takes two inputs and produces a sum and carry output
      - $S = XY' + X'Y$
      - $C = XY$



Truth Table

Inputs		Outputs	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

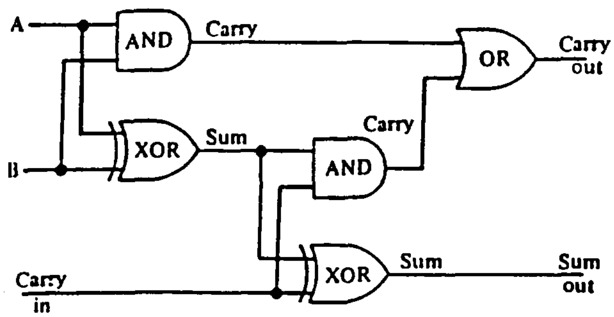
- Full Adder

  Can be constructed using half adders as building blocks

  Takes three inputs and produces a sum and carry output

  $S = X'Y'C_{in} + X'YC'_{in} + XY'C'_{in} + XYC_{in}$

  $C_{out} = XC_{in} + XY + YC_{in}$



Truth Table

Inputs			Outputs	
A	B	Carry in	Sum out	Carry out
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

- Serial Addition

  Only requires a single one bit adder for hardware

  Shifts the two numbers to be added, least significant digits first, through the adder

  During the first bit time, the LSB augend and LSB addend bits are applied

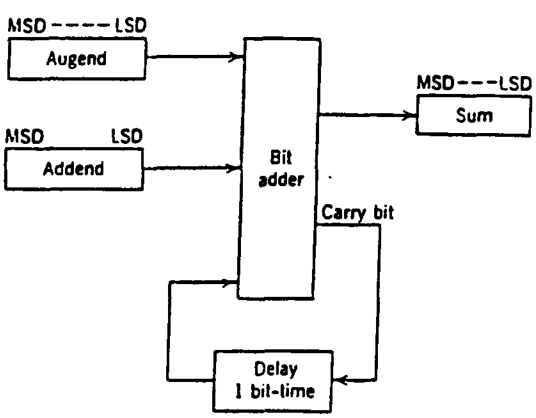
  A sum and carry are produced

  The carry is delayed back to the input side of the bit adder

  During the second bit time, the bit adder takes the second digits of augend and addend as well as the carry bit from the last time step

  Process repeats until all higher order bits are processed

  The sum appears serially at the output of the adder



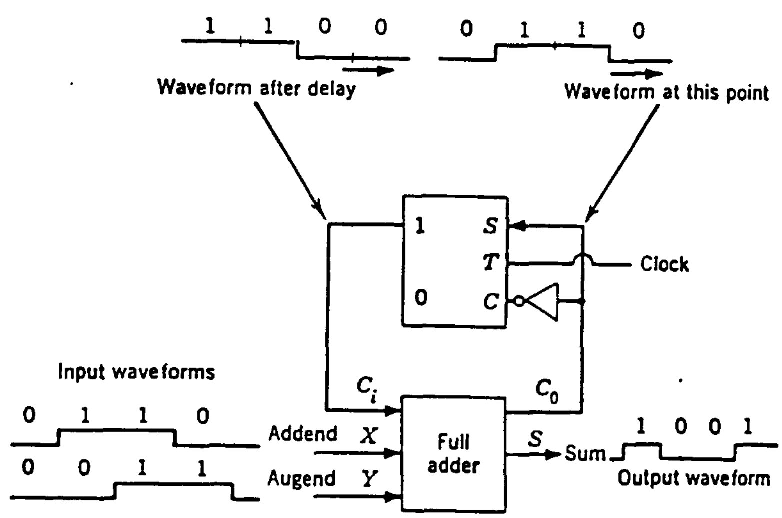


Fig. 6·7 Serial operation of full adder.

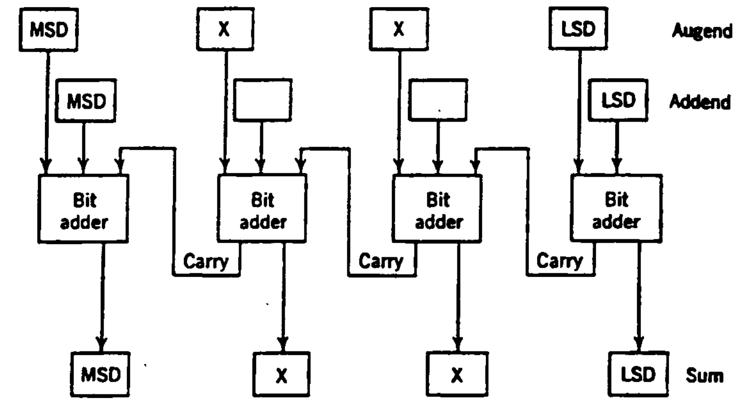
Serial adder is the cheapest and slowest form of an adder circuit

  Uses the single three input full adder over and over for successive digit pairs

  Temporary storage for carry is usually a flip-flop

- Ripple Carry Adder

  Generally, since speed is important, parallel adder sections are used; not serial
  As many bit adders as pairs of bits to be added are needed
  Each carry is applied to the following higher order column to modify its sum



Advantage: Ripple carry adder can be connected in a simple, regular way
Disadvantage: Higher hardware cost (N times as many adders as serial version)
Although it has N times as much hardware as a serial adder, its not N times as fast
Main speed limitation:

Sum digits are not correct until all carries have finished propagating
Carries may propagate over many columns, not just to the adjacent one
e.g.) It is possible that a carry from the LSB can affect the MSB

$$
\begin{array}{cccccccccc}
 & 1 & 1 & 1 & 1 & 1 & 1 & & & \\
 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & \text{or} & 91_{10} \\
 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & & +111_{10} \\
\hline
1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & & 202_{10}
\end{array}
$$

Ripple carry design must be based on the worst-case carry propagation
Longest path is from the inputs X_0, Y_0 (LSB) to the S_n, C_n outputs (MSB)
Thus, the carry digits are essentially formed sequentially, column by column
Total time delay is linearly proportional to the length N of the adder
Therefore, Ripple carry adder is expensive, but not much faster than serial adder

Serial nature of carry propagation is the most difficult problem in addition
Need to reduce the time needed to generate C_i inputs near the MSB of the adder
Logic structures for fast adder design try to speed up the formation of the carries
We study two:

Carry Lookahead
Carry Select Adder

• **Carry Lookahead**

Provide supplemental logic circuitry to form a carry signal into an adder stage if:

1) Predecessor stage generated a carry
2) Predecessor propagated a carry generated two stages previously
3) All j immediate predecessors propagated a carry formed (j-1) stages ago

Supplemental circuitry simultaneously provides carry info to all adder stages

All sums can be computed at same time since carry in to each stage is known

Avoids "ripple" effect

Carry propagation thus becomes concurrent instead of sequential

Consider bit i in an N bit adder system which adds X_i and Y_i:

X_i	Y_i	Carry Action
0	0	No carry from this stage possible (kill any incoming carry)
0	1	A carry will be propagated from this stage if one is received
1	0	A carry will be propagated from this stage if one is received
1	1	This stage will generate a carry

There are three circumstances that may arise:

1) $X_i = Y_i = 0$

There will be no carry out from this bit position even if there is a carry in

2) $X_i = Y_i = 1$

In this case, a carry out will start from bit i regardless of previous carry

Define G (Generate a carry) such that $G_i = X_i Y_i$

3) $X_i \neg= Y_i$

Carry out is dependent on the carry in, and will be equal to the carry in

Define P (Propagate a carry) such that $P_i = X_i$ .EXOR. Y_i

Note: These three actions are mutually exclusive

Only one can occur at each digit position

So, there will be a carry out of stage i (C_i) if either:

1) It is generated in this stage (G_i true) or
2) If a carry from bit i-1 is propagated through stage i

Expressed in logic, the carry out for adder stage bit i is:

$C_i = G_i + P_i C_{i-1}$

Using repeated substitutions for higher order bits:

$C_1 = G_1 + P_1 C_0 = G_1 + P_1 G_0$

$C_2 = G_2 + P_2 C_1 = G_2 + P_2 G_1 + P_2 P_1 G_0$

$C_3 = G_3 + P_3 C_2 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$

$C_4 = G_4 + P_4 C_3 = G_4 + P4G3 + P_4 P_3 G_2 + P_4 P_3 P_2 G_1 + P_4 P_3 P_2 P_1 G_0$

All Gi and Pi functions can be formed independently and in parallel

To compute C_i just need local info from stage i: X_i and Y_i

The N-bit addition process is now independent of N

- Example:

i	8	7	6	5	4	3	2	1	
X	0	1	1	0	0	1	1	0	Time 0
Y	0	0	1	0	1	0	1	0	
Pi	0	1	0	0	1	1	0	0	Time 1
Gi	0	0	1	0	0	0	1	0	
Ci	0	1	1	0	1	1	1	0	Time 2
Si	1	0	0	1	0	0	0	0	Time 3

- Example:

i	8	7	6	5	4	3	2	1	
X									Time 0
Y									
Pi									Time 1
Gi									
Ci									Time 2
Si									Time 3

Continuing expansion, can get an expression for any carry stage (in theory)

Limited in practice by the complexity of the gating structure

To implement C_i, we need:

1 OR gate with i+1 inputs and

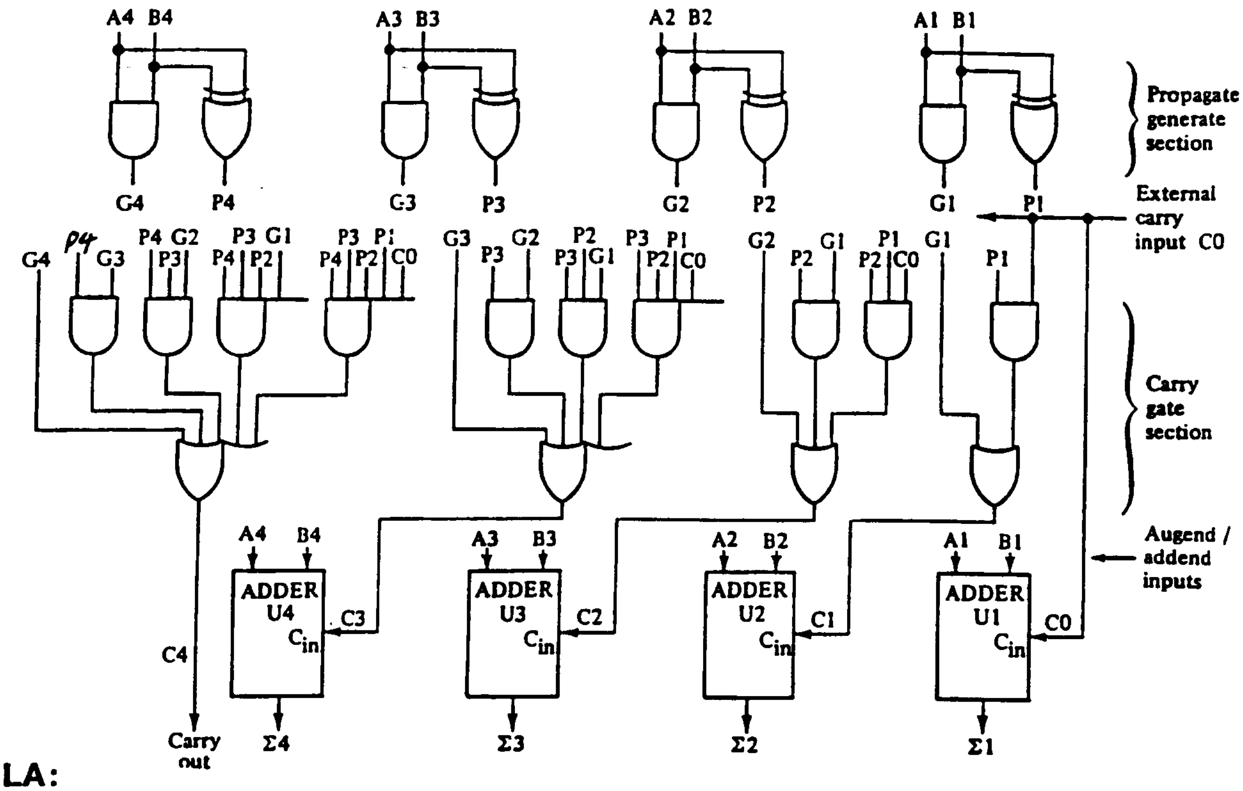
i AND gates with 2, 3, ... i+1 inputs

Requires large fan-in and fan-out

Electrical circuit implementation usually limits fan-in to eight or less

Solve the fanout limitation problem by using groups of blocks

Compromise between implementation complexity and speedup potential



- **Block CLA:**
  - **Common to construct blocks of four-stage look-ahead circuits**
    - **Each block forms carries to the individual stages within it**
    - **Each block also forms Block$_j$ Generate and Block$_j$ Propagate outputs**
      - **If first four stages generate a carry to the fifth stage, then Block$_1$ G = 1**
      - **If first four stages propagate to the fifth a carry received as input, B_1P = 1**
    - **As before, these conditions are mutually exclusive**
  - **Structured as: (Total number of bits) = (Number of blocks) x (Bits per block)**
  - **- Example:**
    - **To get 16 bit adder, use 4 blocks, each 4 bits wide**
    - **Carries inside of each block are formed by lookahead circuits**
    - **Carries between blocks will ripple**

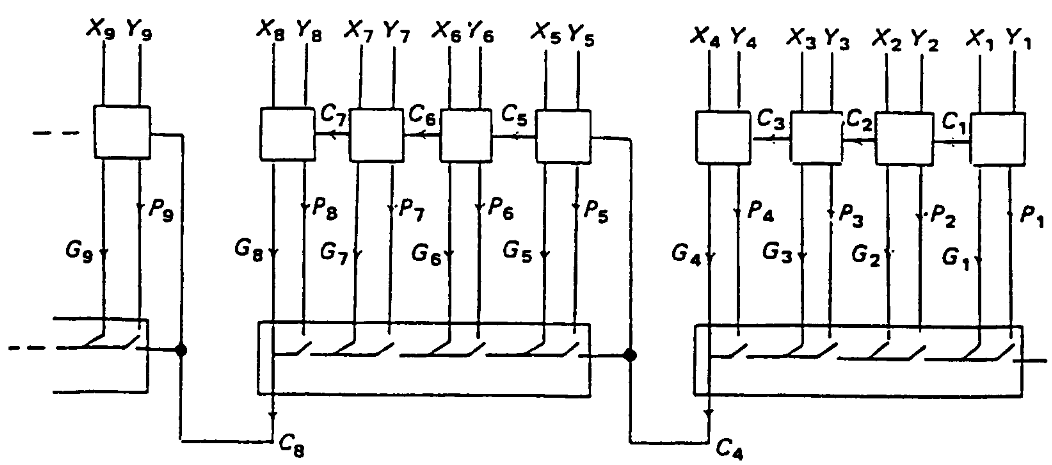


Figure 2.6 Block-carry adder

- **Carry Select Adder**

  Also called Conditional-Sum Adder

  Proposed by Sklansky in 1961

  Since it takes time for carry to reach MSB stages, pre-compute two sum values

    Then, just select the correct one when the carry in becomes known

  For each stage i, two additions are performed in parallel

    One assuming the carry in is zero

    The other assuming the carry in is one

  If we assume carry in is 0, the conditional sum and conditional carry out are:

    S^0 = X .EXOR. Y

    C^0 = X .AND. Y

  If we assume carry in is 1, the conditional sum and conditional carry out are:

    S^1 = XY .OR. X'Y' = (X .EXOR. Y)' = X .EQUIV. Y

    C^1 = X .OR. Y

  Note that there is really no "addition" in the classic sense (S, C_{out} = F (X, Y, C_{in}))

    Because we "add" assuming C_{in} = 0 or 1, we can simplify adder logic

    Equations above only depend on local X and Y; not on C_{in}

    Simultaneous "additions" performed on all stages independently, in parallel

    This will generate all provisional sums and provisional carries in 1 gate delay

    All steps after this consist of just select operations (done with multiplexers)

  The concept of Recursive Doubling is also used during selection

    Number of bits handled at each step can double from previous step

    The number of correct sum digits avail at each step grows as a power of 2

    Thus, the addition of two numbers of length 2^N requires only (N+1) steps

  The Process:

    At first time step: Form two conditional sum and carry out bits for each i

    At second time step: Conditional sum bits from step 1 are grouped in pairs

      Carry from right half of pair is used to select sum & carry for left half

    At each succeeding time step, pairs of pairs are grouped, etc.

      Each time, carry from right half is used to select sum & carry for left half

  Notation:

    Left half and Right half of a pair are marked with L/R subscripts

    Sum and Carry formed assuming 0/1 for carry in marked with 0/1 superscripts

    So, S_L^0 represents left half of a pair of sum digits assuming carry in was 0

    The right half of a carry out assuming that carry in was a 1 would be = C_R^1

Sum & Carry at step K+1 for Left half of word selected by Right half carry (C_R) of step K

S_L^0 C_L^0	S_R^0 C_R^0	These are formed assuming $C_{in} = 0$	Step K
S_L^1 C_L^1	S_R^1 C_R^1	These are formed assuming $C_{in} = 1$	
S_L^* C_L^*	S_R^0	These are formed based on C_R^0	Step K + 1
S_L^{**} C_L^{**}	S_R^1	These are formed based on C_R^1	

Where:

$(S_L^*\ C_L^*) = (S_L^0\ C_L^0)$ if $C_R^0 = 0$; $(S_L^1\ C_L^1)$ if $C_R^0 = 1$

$(S_L^{**}\ C_L^{**}) = (S_L^0\ C_L^0)$ if $C_R^1 = 0$; $(S_L^1\ C_L^1)$ if $C_R^1 = 1$

Example (assume $C_0 = 0$):

	8	7	6	5	4	3	2	1	Bit Position
X	0	1	0	1	1	0	1	1	Addend
Y	0	1	0	1	0	1	0	1	Augend
S^0	0	0	0	0	1	1	1	0	Step 1
C^0	0	1	0	1	0	0	0	1	
S^1	1	1	1	1	0	0	0		
C^1	0	1	0	1	1	1	1		
S^0	1	0	1	0	1	1	0	0	Step 2
C^0	0		0		0		1		
S^1	1	1	1	1	0	0			
C^1	0		0		1				
S^0	1	0	1	0	0	0	0	0	Step 3
)	0				1				
S^1	1	0	1	1					
C^1	0								

Example (assume $C_0 = 0$):

i	8	7	6	5	4	3	2	1	Bit Position
X									Addend
Y									Augend
S^0 C^0									Step 1
S^1 C^1									
S^0 C^0									Step 2
S^1 C^1									
) C^0									Step 3
S^1 C^1									

Each step in the carry select adder represents a new level of multiplexers

Speed advantage over ripple adder grows exponentially with larger N bit words

Logic regularity and complexity between that of CLA and Ripple adder

• Summary for Addition

Different adder schemes are not necessarily disjoint choices

Methods can be mixed and combined as building blocks

	Time	Space
Ripple	O(n)	O(n)
CLA*	O(log n)	O(n log n)
Carry Select	O(sqrt(n))	O(n)

* CLA time/space assumes full lookahead without resorting to ripple between blocks

- **Carry Select Adder**
  - **Time flows down 4 rows per step**
    - **R1 - R4 are all computed at the same time during step 1**
    - **R5 - R8 are all computed at the same time during step 2**
    - **R9 - R12 are all computed at the same time during step 3**
  - **Step 1:**

    For each column, generate S and C two ways:

    1) Assuming Cin = 0 (thereby generating S^0 and C^0)

    2) Assuming Cin = 1 (thereby generating S^1 and C^1)

C_{in}	X_i	Y_i	C^0	S^0	
0	0	0	0	0	
0	0	1	0	1	C^0 = X and Y
0	1	0	0	1	S^0 = X exor Y
0	1	1	1	0	
C_{in}	X_i	Y_i	C^1	S^1	
1	0	0	0	1	
1	0	1	1	0	C^1 = X or Y
1	1	0	1	0	S^1 = X equiv Y
1	1	1	1	1	

Assume C_0 = 0:

	i	D8	D7	D6	D5	D4	D3	D2	D1		Bit Position
	X	0	1	0	1	1	0	1	1		Addend
	Y	0	1	0	1	0	1	0	1		Augend
R1	S^0	0	0	0	0	1	1	1	0	S^0 = X exor Y	Assuming
R2	C^0	0	1	0	1	0	0	0	1	C^0 = X and Y	Carry in = 0
R3	S^1	1	1	1	1	0	0	0		S^1 = X equiv Y	Assuming
R4	C^1	0	1	0	1	1	1	1		C^1 = X or Y	Carry in = 1
R5	S^0									Groups	Assuming
R6	C^0									of	Carry in = 0
R7	S^1									two	Assuming
R8	C^1									bits	Carry in = 1
R9	S^0									Groups	Assuming
R10	C^0									of	Carry in = 0
R11	S^1									four	Assuming
R12	C^1									bits	Carry in = 1

At this point, all possible answers have been generated.

ow, just need to select the correct ones and put them together, column by column.

We use recursive doubling and put sum bit answers together two columns at a time.

The number of correct sum digits identified grows as a power of two each step after step 1.

This avoids a selection of sum bits based on a rippling carry, which would take linear time.

Key concept: Carry out of right side determines which answer of the left side to use.

- Step 2: Merging Columns of 2 wide

Combine the following digits: (D1, D2) (D3, D4) (D5, D6) (D7, D8)

These are all merged at the same time to form four resulting groups.

For each group of two digits, need to get out 2 sum digits and 1 carry digit.

Note that is not a computation, but just a selection of answers precomputed at step 1.

e.g.) R5D1 = R1D1

IF R2D1 = 0 THEN R5D2 = R1D2
R6D2 = R2D2

IF R2D1 = 1 THEN R5D2 = R3D2
R6D2 = R4D2

	i	*D8*	*D7*	*D6*	*D5*	*D4*	*D3*	*D2*	*D1*		Bit Position
	X	0	1	0	1	1	0	1	1		Addend
	Y	0	1	0	1	0	1	0	1		Augend
R1	S^0	0	0	0	0	1	1	1	0	S^0 = X exor Y	Assuming
R2	C^0	0	1	0	1	0	0	0	1	C^0 = X and Y	Carry in = 0
R3	S^1	1	1	1	1	0	0	0		S^1 = X equiv Y	Assuming
R4	C^1	0	1	0	1	1	1	1		C^1 = X or Y	Carry in = 1
R5	S^0	1	0	1	0	1	1	0	0	Groups	Assuming
R6	C^0	0		0		0		1		Of	Carry in = 0
R7	S^1	1	1	1	1	0	0			Two	Assuming
R8	C^1	0		0		1				Bits	Carry in = 1
R9	S^0									Groups	Assuming
R10	C^0									Of	Carry in = 0
R11	S^1									Four	Assuming
R12	C^1									Bits	Carry in = 1

- **Step 3: Merging Columns of 4 wide**

Combine the following digits: (D1, D2, D3, D4) (D5, D6, D7, D8)

These are all merged at the same time to form two groups.

For each group of four digits, need to get out 4 sum digits and 1 carry digit.

Again, this is just a selection of the groups of two digits wide already formed in step 2.

e.g.) R9D1 = R5D1
R9D2 = R5D2

IF R6D2 = 0 THEN R9D3 = R5D3
R9D4 = R5D4
R10D4 = R6D4

IF R6D2 = 1 THEN R9D3 = R7D3
R9D4 = R7D4
R10D4 = R8D4

	i	*D8*	*D7*	*D6*	*D5*	*D4*	*D3*	*D2*	*D1*		Bit Position
	X	0	1	0	1	1	0	1	1		Addend
	Y	0	1	0	1	0	1	0	1		Augend
R1	S^0	0	0	0	0	1	1	1	0	S^0 = X exor Y	Assuming
R2	C^0	0	1	0	1	0	0	0	1	C^0 = X and Y	Carry in = 0
R3	S^1	1	1	1	1	0	0	0		S^1 = X equiv Y	Assuming
R4	C^1	0	1	0	1	1	1	1		C^1 = X or Y	Carry in = 1
R5	S^0	1	0	1	0	1	1	0	0	Groups	Assuming
R6	C^0	0		0		0		1		of	Carry in = 0
R7	S^1	1	1	1	1	0	0			two	Assuming
R8	C^1	0		0		1				bits	Carry in = 1
R9	S^0	1	0	1	0	0	0	0	0	Groups	Assuming
R10	C^0	0				1				of	Carry in = 0
R11	S^1	1	0	1	1					four	Assuming
2	C^1	0								bits	Carry in = 1

Answer: 1 0 1 1 0 0 0 0
MSB Carry out = 0

Example (assume $C_0 = 0$):

	8	*7*	*6*	*5*	*4*	*3*	*2*	*1*	**Bit Position**
X									**Addend**
Y									**Augend**
S^0 **C^0**									**Step 1**
S^1 **C^1**									
S^0 **C^0**									**Step 2**
S^1 **C^1**									
S^0 **C^0**									**Step 3**
S^1									

Example (assume $C_0 = 0$):

i	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	**Bit Position**
X									**Addend**
Y									**Augend**
S^0 **C^0**									**Step 1**
S^1 **C^1**									
S^0 **C^0**									**Step 2**
S^1 **C^1**									
S^0 **C^0**									**Step 3**
S^1 **C^1**									

- **Carry LookAhead Adder**

There will be a carry out of stage i (C_i) if either:

1) It is generated in this stage (G_i true) or

Define G (Generate a carry) such that $G_i = X_iY_i$

2) If a carry from bit i-1 is propagated through stage i

Define P (Propagate a carry) such that $P_i = X_i$ .EXOR. Y_i

Expressed in logic, the carry out for adder stage bit i is:

$C_i = G_i + P_iC_{i-1}$

Using repeated substitutions for higher order bits:

$C_1 = G_1 + P_1C_0 = G_1 + P_1G_0$

$C_2 = G_2 + P_2C_1 = G_2 + P_2G_1 + P_2P_1G_0$

$C_3 = G_3 + P_3C_2 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$

$C_4 = G_4 + P_4C_3 = G_4 + P4G3 + P_4P_3G_2 + P_4P_3P_2G_1 + P_4P_3P_2P_1G_0$

All Gi and Pi functions can be formed independently and in parallel

To compute C_i just need local info from stage i: X_i and Y_i

The N-bit addition process is now independent of N

- Example:

i	8	7	6	5	4	3	2	1	
X	0	1	1	0	0	1	1	0	Time 0
Y	0	0	1	0	1	0	1	0	
Pi	0	1	0	0	1	1	0	0	Time 1
Gi	0	0	1	0	0	0	1	0	
Ci	0	1	1	0	1	1	1	0	Time 2
Si	1	0	0	1	0	0	0	0	Time 3

- Example:

i	8	7	6	5	4	3	2	1	
X									Time 0
Y									
Pi									Time 1
Gi									
Ci									Time 2
Si									Time 3

- **Multiplication**

Addition and Subtraction are usually included in the instruction set

Multiply and divide operations are comparatively more complex than addition

  Small, low-end computers generally do not include multiplication logic

  H/W multipliers and dividers only built into in high performance computers

But mult/div can be performed in S/W given add/sub machine instructions

  Multiply: A repetitive sequence of adds and shifts

  Divide: A repetitive sequence of subtracts and shifts

The basic process:

  Identical to pencil and paper method in the decimal system

  Multiplicand is multiplied by Multiplier to produce a product

Partial Products:

  The intermediate numbers (formed by a bit-by-bit multiply) in the calculation

  Formation of partial products for binary system is easy

  Addition of partial products is more difficult, especially if carries are generated

Only two rules are needed for multiplying a single binary number by a binary digit

  1) If the multiplier digit is 1, the multiplicand is simply copied

  2) If the multiplier digit is 0, the product is zero

In binary, partial products are either a copy of the multiplicand (shifted) or all zeros

Process repeats for all multiplier digits; then the partial products are summed

```
          0 1 0 1   = multiplicand (5)
          1 1 0 1   = multiplier (13)
          -------
          0 1 0 1
        0 0 0 0     Four partial products
      0 1 0 1
    0 1 0 1
    -------------
    1 0 0 0 0 0 1   = product (65)
```

The computer only needs three operations to multiply in this manner:

  1) Ability to sense whether a multiplier bit is either a 1 or 0

  2) Ability to shift partial products

  3) Ability to add the partial products

    Note: Don't need to wait until all partial products are formed before adding

    They can be summed two at a time as they become available.

Two basic methods exist to speedup multiplication:

  Make the additions of the partial products faster (e.g., by using fast adders)

  Reduce the number of additions required

We study recoding techniques to reduce the number of partial products formed

- **Booth's Algorithm**

  Invented in 1951 by A. D. Booth

  Treats both positive and negative numbers uniformly

  Can perform signed-number multiplication in 2's complement

  Key concepts:

  - Subtraction can be done as easily and by using same hardware as addition

    So a system that generates negative partial products is OK

  - A block of 1's can be rewritten as the difference between two numbers

    e.g.) 0111 (7) is equal to 1000 (8) - 0001 (1)

  So instead of adding 3 partial products, just do one add and one subtract

  We let each bit of the multiplier be either zero, positive one, or negative one

  We introduce a new notation to show recoding of multiplier: 0, +1, -1

Recoding Table (X_i is original multiplier bit, Z_i is new multiplier bit):

X_i	X_{i-1}	Z_i
0	0	0
0	1	1
1	0	-1
1	1	0

Recoding can be done in parallel (2 digits in each set via table lookup)

Assume an imaginary trailing 0 to the right of the LSB

Examples:

X = 0 1 1 0 1 0 0 1 1 1 0

Z = 1 0 -1 1 -1 0 1 0 0 -1

ex.) 0 1 1 = 2 + 1 = 3
+1 0 -1 = 4 - 1 = 3

ex.) 0 1 1 1 = 4 + 2 + 1 = 7
+1 0 0 -1 = 8 - 1 = 7

ex.) 0 1 1 0 0 1 1 1 = 64 + 32 + 4 + 2 + 1 = 103
+1 0 -1 0 +1 0 0 -1 = 128 - 32 + 8 - 1 = 103

ex.) 0 0 1 0 1 1 0 0 1 1 1 0 1 0 1 1 0 0
0 +1 -1 +1 0 -1 0 +1 0 0 -1 +1 -1 +1 0 -1 0 0

Note: When complement numbers are added, "blanks" must be 'filled-in'

Especially important when adding partial products of shifted multiplicand

___ 0 1 0 1 Has a leading left "blank"

0 1 0 1___ Has a trailing right "blank"

In general, an N bit number multiplied by a M bit number has a NM bit product

Fill-in on the right is same as in decimal: just add trailing zeros

To fill-in on the left, since the MSB is the sign, need to perform sign extension

e.g.) In 2's complements: +3 = 011 = 0011 = 00011 = 000011 ...

-3 = 101 = 1101 = 11101 = 111101 ...

=> Therefore, left fill-in digits must be the same as the MSB (the sign digit)

e.g.) Assume partial products to be added are (in 2's comp):

	1	1	1	1	1	0	0	1	= -7	
	0	0	0	0	1	1	1	0	= +14	
	1	1	1	0	0	1	0	0	= -28	
(1)	1	1	1	0	1	0	1	1	= -21	(correct: ignore MSB C_{out})

Improper sign extension on first number would yield an incorrect sum:

0	0	0	0	1	0	0	1	= -7	(incorrect)
0	0	0	0	1	1	1	0	= +14	
0	0	1	0	0	1	0	0	= -28	(incorrect)
0	0	1	1	1	0	1	1	= +59	(incorrect)

- Without Booth's Algorithm:

				0	1	0	1	= multiplicand (5)
				0	1	1	1	= multiplier (7)
0	0	0	0	0	1	0	1	
0	0	0	0	1	0	1	0	3 non-zero partial products
0	0	0	1	0	1	0	0	
~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	
0	0	1	0	0	0	1	1	= product (35)

- Using Booth's Algorithm to recode multiplier: (2's comp of 0101 is 1011)

					0	1	0	1	= multiplicand (5)
					1	0	0	-1	= recoded multiplier (7)
	1	1	1	1	1	0	1	1	pre-computed negative of mcand
	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	
	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	2 non-zero partial products
	0	0	1	0	1	0	0	0	
(1)	0	0	1	0	0	0	1	1	= product (35) [ignore MSB C_{out}]

Booth's algorithm is also called the "skipping over 1s" technique

For each block of consecutive 1s, only one add and subtract is needed

The speed gain possible by skipping over 1s is data dependent

Booth's Algorithm does not provide a consistent performance advantage

Booth's algorithm does not guarantee that recoded number is "better" than original

Best case: A large single block of 1s (Booth decreases number of adds reqd)

ex.)	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	+1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1

However, in some cases, Booth's algorithm can be worse than not recoding

Worst case is when multiplier has alternating 1s and 0s (increases # of adds)

ex.)	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1

Note:

Recoded multiplier never has two adjacent non-zero digits of the same sign

Good, since don't have to worry about 11 = 3X or -1-1 = -3X occurring

Want multiples of 2x, which are easy to form in binary using only shifts

- Modified Booth's Algorithm

Ideally, we could reduce the number of additions by scanning many multiplier bits

That is, by going to a higher radix, the number of cycles is reduced

e.g.) Taking 4 bits at a time, the number of multiply cycles is cut by factor of 4

This would also reduce the number of partial products by same amount

However, this might yield hard to compute partial products

e.g.) The sequence 0111 would require 7x of the multiplicand be formed

Cannot be done easily in binary (not a multiple of 2x so can't shift)

Also, it would be good to "skip over a group of zeros in the multiplier"

But this would make the multiply algorithm data dependent

Not good since the concept of variable shifts is difficult to implement

Instead of considering strings of arbitrary length, we consider strings of 2 bits

Modified Booth's: Provides more consistent speedup than Booth's Algorithm

Guarantees that an N bit multiplier will generate at most N/2 partial products

Uniformly handles the signed operand case (complement numbers)

Recodes multiplier bits by pairs of digits and a third digit (1 bit to the right)

Assume an implied 0 to right of LSB

A problem with Booth's recoding is the set of pairs (-1 +1) and (+1 -1)

These occurred when an isolated zero interrupted a series of 1s

Any digit pair .... -1 +1 .... is equal to .... 0 -1 ....

and .... +1 -1 .... is equal to .... 0 +1 ....

Also note that .... +1 0 .... is equal to .... 0 +2 ....

Recoding table for Modified Booth's:

Left Multiplier bit pair: bit i+1	Multiplier bit pair: bit i	Multiplier bit on the right: bit i-1	Multiplicand selected (Z_i)
0	0	0	0 x M
0	0	1	+1 x M
0	1	0	+1 x M
0	1	1	+2 x M
1	0	0	-2 x M
1	0	1	-1 x M
1	1	0	-1 x M
1	1	1	0 x M

Recoding can be done in parallel by scanning triplets of digits

Assume that the digit to the right of the LSB is an imaginary 0

X = 0 1 0 1 1 1 1 0 = 1 x 64 + 1 x 16 + 1 x 8 + 1 x 4 + 1 x 2 = 94

1 2 0 -2 = 1 x 64 + 2 x 16 - 2 x 1 = 94

Modified Booth's is also called canonical recoding

Although pairs of digits are recoded at a time, no two adjacent 1s appear in pair

Therefore, the 3x factor never occurs in the recoded multiplier

Results in simple multiples of the multiplicand: + - x and + - 2X

ex.) 0 1 1 = 2 + 1 = 3

+1 -1 = 4 - 1 = 3

ex.) 0 1 1 1 = 1x4 + 1x2 + 1x1 = 7

+2 -1 = 2x4 - 1x1 = 7

ex.) 0 1 1 0 0 1 1 1 = 1x64 + 1x32 + 1x4 + 1x2 + 1x1 = 103

+2 -2 +2 -1 = 2x64 - 2x16 + 2x4 - 1x1 = 103

ex.) 0 0 1 0 1 1 0 0 1 1 1 0 1 0 1 1 0 0

+1 -1 -1 +1 0 -1 -1 -1 0

Each step in the multiplication cycle selects:

Multiplicand if multiplier is +1, or complement of multiplicand if multiplier is -1

Multiplicand shifted if multiplier +2, or complement of multiplicand if multiplier -2

- Example (assume 2's complement notation):

Using Modified Booth's Algorithm to multiply

```
 0 1 0 0  = 4 multiplicand
 0 0 1 1  = 3 multiplier
```

```
Recode multiplier from:   0  0  1  1
                    to:     +1    -1
```

Precompute 2's comp of multiplicand 0100 as 1100

Note: Since multiplier is being processed 2 bits at a time, shift twice each cycle

```
                0 1 0 0   = multiplicand (4)
                 +1  -1   = recoded multiplier (3)
        1 1 1 1 1 1 0 0     pre-computed negative of mcand
        0 0 0 1 0 0 0 0
    (1) 0 0 0 0 1 1 0 0   = product (12)  [ignore MSB Cout]
```

- Example (assume 2's complement notation):

Using Modified Booth's Algorithm to multiply

```
 0 1 0 1  = 5 multiplicand
 0 1 1 1  = 7 multiplier
```

```
Recode multiplier from:   0  1  1  1
                    to:     +2    -1
```

Precompute 2's comp of multiplicand 0101 as 1011

Note: Since multiplier is being processed 2 bits at a time, shift twice each cycle

So a + - 2x multiplier would require an additional shift (3 total)

```
                0 1 0 1   = multiplicand (5)
                 +2  -1   = recoded multiplier (7)
        1 1 1 1 1 0 1 1     pre-computed negative of mcand
        0 0 1 0 1 0 0 0     multiplicand shifted left since +2
    (1) 0 0 1 0 0 0 1 1   = product (35)  [ignore MSB Cout]
```

The Modified Booth's recoding yields a minimal recoding

It has the smallest number of nonzero digits

Best case: A large single block of 1s

```
ex.)  0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
        +2  0   0   0   0   0   0   0  -1
```

Modified Booth's algorithm cannot be worse than not recoding

Worst case is when multiplier has alternating 1s and 0s

```
ex.)  0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
        +1  +1  +1  +1  +1  +1  +1  +1  +1
```

- **Division**

  The most difficult and time-consuming arithmetic operation

  Fortunately, it occurs much less frequently than any of the others

  Dividend is divided by divisor to produce a quotient and remainder

  Central idea:

  Subtract divisor from dividend repeatedly until result is zero or negative

  The number of subtractions before overdrawing is the quotient

  To accelerate process, we do it digit by digit, shifting each time

- Example of paper and pencil process in decimal:

$$13\overline{)274}$$

We first 'try' 13 divided into 2 and it 'doesn't go'. Q_3=0.

So we do a shift left, effectively bringing down the next digit of dividend.

Next, we 'try' 13 divided into 27. It goes.

Now try 2 times 13 and it still goes (26 is less than 27).

Try 3 times 13 and its an overshoot (39 subtracted from 27 is negative).

So, largest successful subtraction is with $Q_2 = 2$.

Subtract 26 from 27 and get 1 remainder and bring down 4 (shift left)

Try 13 divided into 14 and it goes once.

Now try 2 times 13 and its an overshoot (26 from 14 is negative).

So, $Q_1 = 1$

No more digits in dividend

Answer is $Q_3Q_2Q_1 = 21$ with remainder 1

Note: The above division "algorithm" has a trial and error step (will it go ?)

Much more difficult to automate in logic circuitry

Instead of partial products, we have partial remainders

Partial remainder:

The quantity left after the most recent subtraction

Partial because it is not the final remainder

No simple algorithms for performing signed division

The machine cannot tell by 'looking' whether divisor will go into dividend digits

It is easy, however for the machine to detect sign inversions (MSB flips)

Machine must perform each 'try' as an actual subtraction and test sign

-If result is positive or zero, then the divisor can go into dividend digits

-If result is negative, then the divisor didn't go into dividend digits

If restoring method is used, need to add back divisor before shifting

Cannot speedup division in same way we can speedup multiplication
- Need to examine result of each subtraction before doing next cycle
- So full subtractions must be performed at each cycle
- Cannot save subtractions for end as with partial products of multiplication
- Sign of partial remainder is used to determine new operand on next cycle

General restrictions:
- Divisor must not be zero
  - Most computers even have problems when divisor is close to 0
- Repeated subtraction makes it important that the divisor be large
  - Subtraction by a small number can continue for a long time
- If we assume a floating point representation, most computers require:
  - Divisor > Dividend

There are two division algorithms used for binary:
- Restoring
- Non-Restoring

• Restoring Division

- The restoring process in decimal:
  - Position divisor with respect to dividend and perform a (trial) subtraction
  - Initial alignment is with the MSD of the dividend
  - Subtract repeatedly until answer is negative
  - Number of successful subtractions is quotient digit i
  - Restore negative partial remainder by adding back divisor
  - Shift divisor right

Binary division is simpler than decimal because quotient digits are either 0 or 1
- No need to "guess" how many times divisor will go into dividend

- The Restoring Division process in binary:
  - Position divisor with respect to dividend and perform a (trial) subtraction
  - Initial alignment is with the MSB of the dividend
  - If remainder is zero or positive, a quotient bit $Q_i = 1$ is determined
    - This means dividend can be divided by divisor
    - Remainder is extended by another bit of the dividend (shift left)
    - Divisor is repositioned and another subtraction performed
  - If remainder is negative, a quotient bit $Q_i = 0$ is determined
    - This means that the guess "overshot" and divisor cannot go into dividend
    - Dividend is restored by adding back the divisor
    - Divisor is repositioned (shifted) for another subtraction

- **Restoring Example in Decimal:**

$8 \overline{)624}$

	6	24	
Cycle 1: subtraction	- 8	00	Align divisor at Q3 digit
Result:	- 1	76	Negative result, so Q3 = 0
Cycle 2: restore add	+8	00	
Result:	6	24	
Cycle 3: subtraction	-	80	Shift Divisor to right
Result:	5	44	Positive result, Q2 = 1
Cycle 4: subtraction	-	80	
Result:	4	64	Positive result, Q2 = 2
Cycle 5: subtraction	-	80	
Result	3	84	Positive result, Q2 = 3
Cycle 6: subtraction	-	80	
Result:	3	04	Positive result, Q2 = 4
Cycle 7: subtraction	-	80	
Result:	2	24	Positive result, Q2 = 5
Cycle 8: subtraction	-	80	
Result:	1	44	Positive result, Q2 = 6
Cycle 9: subtraction	-	80	
Result:		64	Positive result, Q2 = 7
ycle 10: subtraction	-	80	
Result:	-	16	Negative result, so Q2 = 7
Cycle 11: restore add	+	80	
Result:		64	
Cycle 12: subtraction	-	08	Shift Divisor to right
Result:		56	Positive result, Q1 = 1
Cycle 13: subtraction	-	08	
Result:		48	Positive result, Q1 = 2
Cycle 14: subtraction	-	08	
Result:		40	Positive result, Q1 = 3
Cycle 15: subtraction	-	08	
Result:		32	Positive result, Q1 = 4
Cycle 16: subtraction	-	08	
Result:		24	Positive result, Q1 = 5
Cycle 17: subtraction	-	08	
Result:		16	Positive result, Q1 = 6
Cycle 18: subtraction	-	08	
Result:		08	Positive result, Q1 = 7
Cycle 19: subtraction	-	08	
Result:		00	ZERO result, so Q1 = 8

Answer = (Q3x100) + (Q2x10) + Q1 = 78

- Restoring Example in binary:

Note: This example (and the non-restoring one) uses abbreviated binary numbers

Goal is to demonstrate the general concepts of Restoring vs. Non-Restoring

Just shows + or - sign and magnitude of binary number (instead of 2's comp)

For simplicity, the math (the addition or subtraction) is computed in decimal

This makes it easier for us to add, subtract, and view result

But a shift means to shift the binary representation; not the decimal number

So each shift in divisor represents a power of two (not ten)

In a real machine, everything is worked using complement binary notation

9 divided by 3:

$11\,\overline{)1001}$

	0001	001	= 9	
First cycle: subtraction	- 0011	000	= -24	Align divisor at Q4 digit
result:	-0001	111	= -15	Negative result, so Q4 = 0
Second cycle: restore add	+ 0011	000	= + 24	
result:	0001	001	= 9	
Third cycle: subtraction	-001	100	= -12	Shift divisor
result:	-	011	= - 3	Negative result, so Q3 = 0
Fourth cycle: restore add	+001	100	= +12	
result:	0001	001	= 9	
Fifth cycle: subtraction	-00	110	= - 6	Shift divisor
result	0000	011	= + 3	Positive result, so Q2 = 1
Sixth cycle: subtraction	-0	011	= - 3	Shift divisor
result:		+0	= + 0	Positive result, so Q1 = 1

Answer is Q4 Q3 Q2 Q1 = 0 0 1 1 = 3

Restoring division is generally not used because additions "undo" subtractions

Non-Restoring division is usually favored.

- Non-Restoring Division

Avoids restoration step by allowing for negative partial remainders

Instead of always subtracting divisor, allow for adding the divisor

Rule: Whenever the signs of the partial remainder changes, toggle b/w add & sub

Number of subtractions yields a positive digit quotient

Number of additions yields a negative digit quotient

=> Make use of Booth's Representation in the quotient digits

So an "overshoot" is OK; it can be corrected on the next cycle instead of restored

- Non-Restoring Process in Decimal:
  - Subtract repeatedly until partial remainder changes sign
  - Then switch to adding until answer changes sign again and resume subtracting
  - At each switch, also shift divisor (in effect, bringing down next digit)
  - Number of subtractions done is a positive quotient digit Qi
  - Number of additions done is a negative quotient digit Qi

- Non-Restoring Example in Decimal:

$8\overline{)624}$

	6	24	
Cycle 1: subtraction	- 8	00	A subtraction was done so: Q3 = 1
Result:	- 1	76	Sign change (+624 to -176), so add
Cycle 2: addition	+	80	Shift divisor; Add was done: Q2 = -1
Result:	-	96	Same sign (-176 to -96)
Cycle 3: addition	+	80	Add was done, so: Q2 = -2
Result:	-	16	Same sign (-96 to -16)
Cycle 4: addition	+	80	Add was done, so: Q2 = -3
Result:	+	64	Sign change (-16 to +64), so subtract
Cycle 5: subtraction	-	08	Shift divisor; Sub was done: Q1 = 1
esult:		56	Same sign (+64 to +56)
Cycle 6: subtraction	-	08	Continue subtracting: Q1 = 2
Result:		48	
Cycle 7: subtraction	-	08	Q1 = 3
Result:		40	
Cycle 8: subtraction	-	08	Q1 = 4
Result:		32	
Cycle 9: subtraction	-	08	Q1 = 5
Result:		24	
Cycle 10: subtraction	-	08	Q1 = 6
Result:		16	
Cycle 11: subtraction	-	08	Q1 = 7
Result:		08	
Cycle 12: subtraction	-	08	Q1 = 8
Result:		00	ZERO result, so Q1 = 8

Answer = (Q3x100) + (Q2x10) + Q1 = 100 -30 +8 = 78

Note: This gives the same answer as the restoring method

But saves two restoring adds and (in this example) requires less total cycles

Although it is more complicated, fewer steps are required in non-restoring div

- Justification of the non-restoring division process for a binary case:
  - To illustrate how restoring and non-restoring both compute same answer,
    - Assume a partial remainder is being computed
    - So, B (shifted version of divisor) is subtracted from A (dividend or partial rem)

Restoring	Non-Restoring
First cycle: subtract B from A A - B	First cycle: subtract B from A A - B
Result is negative, so need to <u>restore</u>: (A - B) + B	Result is negative, so add on next cycle
Shift left to produce: 2(A - B + B) = 2A	Shift left to produce: 2(A - B)
Second cycle: subtract B: 2A - B	Second cycle: <u>add</u> B 2(A - B) + B = 2A - 2B + B = 2A - B

Note: Results are the same after second cycle for both restoring and non-restoring
- Therefore, Non-Restoring division:
  - Reduces the total number of steps (bypasses the restore operation)
  - Assumes subtraction and addition take same amount of time (complements)
  - Requires Booth's Decoding of the resulting quotient

- Non-Restoring Example in Binary:
  - If signs of before and after partial remainders change, toggle b/w add & subtract
  - Record a +1 for Qi when a subtraction is performed
  - Record a -1 for Qi when an addition is performed

11 ⟌ 1001

	0001	001	= 9	
First cycle: subtraction	- 0011	000	= -24	
result:	-0001	111	= -15	Q4 = +1
Second cycle: add	+ 001	100	= + 12	Sign changed (+9 to -15)
result:	-0	011	= -3	Q3 = -1
Third cycle: add	+00	110	= + 6	Sign same (-15 to -3)
result:	+0	011	= + 3	Q2 = -1
Fourth cycle: subtract	-0	011	= -3	Sign changed (-3 to +3)
esult:	0	000	= + 0	Q1 = +1

Answer (in Booth's Notation) = Q4 Q3 Q2 Q1 = +1 -1 -1 +1 = 8 - 4 - 2 + 1 = 3

• Instruction Sets

An important aspect of computer design is the specification of the instruction set

The computer architecture is affected by the current environment in terms of:

Technology (both H/W and S/W)

Market demands for cost/performance

A large part of current computing is done using High-Level Languages (HLLs)

Therefore, the "goodness" of an architecture is largely determined by:

1) How efficiently HLL programs can be compiled into machine object code

2) How efficiently the resulting code executes on the machine

• Complex Instruction Set Computer (CISC)

During the 1960's and 1970's, trend was toward language-directed architectures

The architecture should support (as closely as possible) the S/W language

More and more instructions were being implemented as part of the H/W

The instructions were also getting more and more complex

This trend was caused by the emergence of high-level programming languages

Assembly language was getting outdated due to low programmer productivity

Rich languages had many constructs (statement types)

Powerful languages also had sophisticated instructions

New machine instructions were created to support HLL programs efficiently

Goal was to reduce the "semantic gap" b/w the HLL and the machine code

Belief was that large semantic gaps made the HLL code hard to compile

This trend was also motivated by marketing for "upward compatibility"

Old programs had to work on a new machine

So, a new computer should have all the instructions of old mach. and more

Thus, each new instruction set was designed to be a superset of its predecessor

Instruction sets continued to grow for each generation of machine family

The majority of computers produced during the last decade were CISC

Some architectures were even designed for "direct" execution of HLL programs

If machine's instruction set = HLL's instruction set, no compilation needed

ex.) Burroughs B5000/B6000 series were designed for Algol-like languages

Another example: Intel APX-432 Ada chip

- CISC Characteristics
  - A large number of instructions, typically from 100 to 250 instructions
  - Instructions are variable length in terms of memory bytes and execute time
  - Instructions can have a variable number of operands
  - Instructions have variety of addressing modes (e.g. direct, indirect)
  - Instructions can directly access operands in memory (bypassing registers)
  - Some instructions perform specialized tasks and are used infrequently

- CISC Rationale
  - -Replaces expensive software with inexpensive hardware
    - Hardware is one-time cost; programming is recurring, highly labor intensive
    - OK to migrate functionality (and cost) from software into hardware
    - Since H/W cheap, OK to do this even for seldom used (but slow) instructions
      - e.g.) Matrix multiply
  - -Increases performance
    - Traditional belief is that H/W will also generally execute faster than S/W
  - -Saves memory space required to represent a HLL program
    - Incorporate variable-length instruction formats.
      - Makes instructions are only as long as they need to be
    - Allow instructions to directly access memory (vs. registers) as needed
      - No need to write to temporary scratch pad registers
      - Provides more flexibility in addressing modes
    - Don't need as much "expansion" from HLL to object code
      - Object code can map directly into the HLL source code
  - -Eases compiler writing by reducing semantic gap b/w HLL and target architecture
    - A more powerful instruction set should make it easier to optimize code **
      - ** Or, so it was thought

- CISC Problems
  - Hardware has too much complexity
    - Too many instructions to design into hardware (lots of H/W)
    - Many instructions were very complex to implement in logic (high design cost)
    - Increased design and test time for hardware development cycle
  - Variable length instructions are harder to pipeline
    - Execute phase could require a different number of clocks (multiply vs. add)
      - Especially since some did direct access to memory instead of registers
  - Extreme HLL machines were too language specific (not general purpose)
    - The 432 only good for Ada; Burroughs machines only good for Algol
  - The biggest problem: Compilers didn't even use most of the instructions

• The first evidence against CISC: Compilers
  - Empirical evidence disproved need for powerful HLL-like instructions in machine
    - Having instructions semantically close to HLL did not ease compilation step
    - In fact, compilers used only a small fraction of the available instruction set
  - Alexander (1975) measured IBM 360 code against machine instructions:
    - 10 instructions accounted for 80% of all instructions executed
    - 21 instructions accounted for 95%
    - 30 instructions accounted for 99%

Wulf (1981) explained this phenomenon:

A compiler essentially performs a large case analysis

IF HLL instruction is xx THEN object code is yy, zz

Thus, a compiler for a machine with a more complex instruction set will:

- Have more ways to realize a given HLL instruc. in machine instructions
- Need to perform a larger case analysis on more alternatives
- Require greater compilation time

The production of efficient code must be balanced with compilation speed

Longer search time needed to find the "best" machine instruction to use

If full power of a CISC were exploited, compiler would become too slow

Therefore, compiler writers generally:

Use only about one-sixth of the instruction set 99% of the time

Prefer simpler architectures with fewer choices more uniform instructions

Ease of compilation is enhanced with simpler and more uniform architectures

• Reduced Instruction Set Computer (RISC)

Patterson 1980 UC Berkeley

Even if the CISC trend was justified in the past, today's environment is different

New factors have arisen to favor simpler and more regular architectures

RISCs represent a departure from the classic evolutionary trends in computers

RISCs constitute a significant and distinct architectural style.

Basic idea: More cost-effective computers can be realized w/ simpler architectures

- RISC Characteristics:
  - A small set of instructions, typically 20-30
  - Few addressing modes
  - All operations are done to/from internal registers
  - All instructions are the same fixed length
  - All instructions consume a single processor cycle
  - Control is hardwired instead of microcoded
  - Instruction pipelining is used extensively

- RISC Rationale:
  - From the viewpoint of Compilation:

Since most machine instructions are ignored by the compiler,

select a small (or "reduced") set of CISC instructions

that are sufficiently simple to be exploited fully by the compiler.

The instr. set supports the most frequent and time-consuming HLL operations

Less time-consuming operations could be synthesized by the compiler

Let compiler quickly find many short, simple instructions for the HLL operation

- From the viewpoint of VLSI Implementation:
    - Larger and more varied instruction sets require large amounts of control store
    - Thus, CISC architectures consume substantial chip area for the control unit
        - e.g.) Chip area used for control logic is 40% for the 432; 50% for 68000
    - Also, more complicated instruction codes require more specialized circuits
        - Expends more chip area for seldom used instructions
    - Major limiting factor for VLSI is total transistor count (or chip area floorspace)
    - The greater the area, the greater the probability of a manufacturing defect
    - A given amount of chip area can be used in many different ways
    - Simple architectures require less control logic
        - Frees up area for other functions and architectural implementations
    - Instead of control logic, area could be used in better ways to improve perf.
        - e.g.) Pipelining or larger register banks
        - These techniques have a more profound and direct effect on performance
    - Therefore, VLSI RISCs can be faster than VLSI CISCs

- From the viewpoint of Design Time:
    - RISCs can be expected to be more cost effective to design
        - Their simplicity reduces time to design and test the H/W architecture
    - Reduces time from conception to market
        - Particularly important in rapidly moving technology

- Berkeley RISC-1 Chip (1982)
    - One semester graduate student team project using Berkeley CAD tools and MOSIS
    - Goal: To provide a prototype chip demonstration of RISC concepts
    - Evaluation:
        - -Design Time:

TABLE 7.3 Design Metrics for the RISC I and Some Other Microprocessors

Processor	Transistor Count (× 1000)	Design Effort (Man-months)	Layout Effort (Man-months)
RISC I	44	15	12
MC68000	68	100	70
Z8000	17.5	60	70
iAPX-432/01	110	170	90
iAPX-432/02	49	170	100

Source: Patterson and Sequin (1982); Katevenis (1985).

- -Performance:
    - Two measures:
        - High-level language execution support factor (HLLESF)
        - HLLESF = speed of program written in assembly language / speed of the same program written in a HLL
        - HLLESF = 0 implies higher penalty of using RISC as an HLL machine
        - HLLESF = 1 implies architecture is appropriate for HLL

TABLE 7.4—Comparative Performances

Measure (Average ± Standard Deviation)	RISC-I	68000	Z8002	VAX 11/780	PDP-11/70
HLLESF	0.90 ± 0.1	0.34 ± 0.3	0.46 ± 0.3	0.45 ± 0.2	0.50 ± 0.2
Performance ratio (times slower than RISC-I)	1	3.5 ± 1.8	4.1 ± 1.6	2.1 ± 1.1	2.6 ± 1.5

Source: Patterson and Piepho (1982).

- **Fault Tolerance**

  Previously, cost and performance were the primary dimensions of interest.
    Cost / Performance design space is easily understood.
  In some applications, however, reliability is another factor for consideration.
    But Reliability dimension is more abstract and harder to quantify and verify.
  Mission Critical Systems require ultra-high reliability from computers.
    In these situations, it is essential that the machine be available and reliable
    Under no circumstances should it be allowed to break down completely
      Spacecraft Computers (High reliability required for certain lifetime)
      Air Traffic Control (High reliability and availability required)
      Traffic Lights (Fail-soft capability required)
      Telephone System
  Computer designer must be able to specify and control reliability of the system
    Reliability can be enhanced through the use of certain design techniques
  Reliability:
    A measure of the capability of the machine to operate without failure
    The probability of survival of the machine over some specified time period
  Achieving reliable operation becomes more difficult with more complex designs
    More parts and interconnections increase the likelihood of faults occurring
    As more parts are involved, probability of failure is product of failure rates
      e.g.) If each part has 90% reliability, two parts have .9**2 = 81% reliability
        Three parts will have a system reliability of .729 when combined, etc.
    Increasing capacity requires increased reliability for usefulness
      While higher levels of integration generally reduce the failure rate per bit,
        the increases in the total number of bits can offset this improvement.
      e.g.) Memory: Even though failure rates per bit improved,
        the overall reliability decreased due to the increased capacity.
    => Complex systems must be designed to tolerate faults
      Error detection and correction become important
  Economic benefits can also be gained from system resiliency
    Reduces down time and maximizes useful computing cycles delivered
    Can contribute to more performance, and thus, increased cost/performance
  Speed of computer has far outstripped the speed of manual maintenance
    Human intervention is orders of magnitude too slow
    Fault detection and correction must be done automatically
  One solution suggests improvement of each of the individual components
    That is, to design a reliable system, use higher quality constituent parts
      Will obviously improve situation, but not cost effectively
    The manufacturing and testing processes would be pushed into overutilization
    Impractical to obtain mission-critical reliabilities with only local optimization

- Failure Rate measures the rate of malfunctions per unit time
  - The failure rate of a component changes during its lifespan
  - Three main regions to the "Bathtub Curve":
    1) Infant Mortality (Initial High Rate of Failure)
       - Caused by manufacturing faults that went undetected during factory testing
       - These occur early in the life cycle under moderate stress levels.
       - This passage through this phase can be accelerated by using "burn-in".
    2) Useful Life (Stable Period of Highest Reliability)
       - This is the ideal range of operational service for a device
       - Failure rate is a low, constant value, the lowest during entire lifespan
       - Only occasional random failures occur in this phase
    3) Wearout (Terminal High Failure Rate)
       - Aging causes a rapid rise in failure rate
       - Occurs because device is near the end of it's useful lifetime

- Causes of Faults:
  - Normal background level of failures during "Bathtub curve" product life
    - Due to Random, statistical failures
  - Incorrect and/or Incomplete Specifications
    - Critical design goals were unstated or misstated in formalized design docs
  - Poor Implementation
    - Incorrect Design of algorithms and/or architectures
    - Poor manufacturing, bad components, poor coding
  - Physical Limitations of Technology:
    - e.g.) Magnetic media is generally very fragile and error prone
      - Requires environment free of dust and stray magnetic fields
      - Storage life also affected by temperature and relative humidity
  - Environmental Stresses
    - Harsh operating conditions imposed by outside sources
    - Shock, vibration, thermal, electrostatic, X-Ray

- Types of Faults:
  - Permanent:
    - Continuous in persistence after onset occurs.
    - Generally Repeatable and Predictable once Detected and Understood
  - Intermittent:
    - Exists only during some intervals, but not others.
    - Seemingly Random in occurrence (and possibly random in location)
    - Caused by Internal Errors (Aging H/W, glitches, race conditions, deadlock)
  - Transient:
    - A one-time occurrence caused by a temporary external environmental factor

- Complex systems can contain residual faults despite extensive factory testing
  - Therefore, faults are to be expected during normal operation and must be *tolerated*
  - A fault-tolerant computer will have schemes to deal with the various types of faults.
    - Want to prevent system failures and lengthy service interruptions due to faults
  - We will briefly describe two common Hardware techniques:
    1) TMR
    2) Parity/Hamming

- Triplicated Modular Redundancy (TMR)
  - TMR uses a multi-processor architecture to provide fault tolerance
    - System is composed of three (or more) redundant processor units
    - Processors perform identical, duplicate work to check and correct each other
  - TMR is a form of masking redundancy
    - All redundant components are active at all times and operating independently
    - When a fault occurs, the effect of the faulty never appears at the primary output
      - The fault is completely masked by the redundant circuits in real time.
    - There are no separate steps for fault detection followed by fault correction.
    - Detection and Correction are both done inherently during normal operation
  - TMR is the most common form of passive hardware redundancy
    - An ordinary design is simply triplicated and voters are inserted between stages
  - Each module in the system is triplicated
    - The three functional modules each receive identical inputs
      - and perform identical functions using those inputs.
    - Majority voters are placed between the stages of module triplets.
  - A failure of any single module in a stage is masked by the majority voter
    - The two good modules will outvote the one bad module
    - No computational cycles are lost due to any single module failure
    - Error never becomes apparent, nor propagates past the majority voter
  - Voter's result is correct as long as no more than one module per stage is faulty

  - Hardware voters for digital data are relatively simple and easy to design
    - The time required to perform the vote is a two gate propagation delay
    - But it is critical that the timing of the 3 modules is synchronized
      - If input values to voter are not synchronized, voter results will be incorrect
        - e.g.) Space Shuttle uses TMR
          - Timing error occurred on maiden launch causing scrub

  - Who checks the checker (voter) ?
    - To verify correctness of checking circuitry, Voters can be triplicated as well
    - Any single voter failure is equivalent to a failure of the module it feeds
      - So voter failures can also be corrected and masked automatically

- N-Modular Redundancy (NMR)

  A generalization of TMR using N = 2t + 1 modules and voters in each stage

  In most cases, N is selected as an odd number so that majority voting can be used

  Using N modules instead of three allows more module faults to be tolerated.

  Up to t failures in each stage are masked using 2t + 1 input majority voters

  e.g.) A 5-MR system can still produce a correct answer even if two modules fail

  The primary tradeoff in NMR is the large amount of hardware required (i.e., cost)

  - N-Version Software Redundancy and Design Diversity can also be used

    Each program is designed and coded independently then voted upon

  - Interesting Note:

    Although the reliability of a triplicated circuit is initially higher than a single circuit, it drops faster with time and aging of components.

    That is, after a certain amount of time, a triplicated system becomes less reliable than a simplex one.

    Knowing this crossover time is a critical design parameter.

    Total Integrated Failure Probability Area under curve must be same.

    TMR just "shifts/skews" reliability curve so that its higher before crossover

    After crossover point, there are simply more parts to go bad as they age.

    Mission must be completed before crossover point is reached.

  - TMR Increases reliability, but at a very high price.

    Generally, TMR is too expensive to use in most commercial computers

    Triplicates / Multiplies by N, the hardware costs

    Even for mission critical applications, all costs must be weighed carefully

    e.g.) Requires more space, weight, power (at a premium for spacecraft)

  - Compromises are often made via the use of Duplication and Error-detecting codes

    Error-correcting codes can provide masking of individual bit errors without requiring three times the circuitry as needed by TMR.

- PARITY

  The most common error detection code used are parity bits.

  Obtained by including an extra digit with the information bits such that the decimal sum of 1's in the number is either odd or even.

  Use of the parity bit to detect errors rests upon two assumptions:

  1) That the Probability of Errors occurring is relatively small

  2) If an error does occur, it is most likely to be a 1-bit error.

  Parity assumes that the chances of two or more incorrect bits is very small

  In Simplest Form, Parity codes can detect single errors, but not multiple errors.

  Thus, undetected errors are still possible (but can be made highly improbable)

- PARITY for single bit failure detection simply requires 1 extra check bit
  - This parity bit, P, is appended to data packet upon transmission
    - e.g.) Given 11110000 as an 8-bit data packet and Odd Parity Protocol, P = 1
      - Data is then transmitted as 11110000P = 111100001
      - Receiver verifies that 9 bits received has an odd number of 1's in string

- PARITY for multiple bit failure detection and automatic single-bit correction
  - For some applications, double errors within the same data block are likely
    - Probability of a dropout affecting multiple bits in magnetic tape are significant
  - A Horizontal and Vertical Parity Check scheme can be used
    - Consider the set of bits arranged into rows and columns (i.e., as a matrix)
    - Parity is computed across each of the rows and columns
    - \- Double error detection can be accomplished
      - e.g.) A double error in any row will result in a (false) good row parity check
      - Two column parity bits will be incorrect thereby detecting the double error
      - But, It cannot be corrected since exact location of faults cannot be isolated
    - \- Single error correction can be accomplished
      - e.g.) Any single error will produce a parity error on the row and column for it
      - Incorrect parity bits provide the row/column coordinates of the bad bit.
      - The incorrect bit can be corrected by simply inverting it.

- HAMMING Code: Least-Cost PARITY for Checking / Correcting a Linear set of bits
  - Coding is the representation of information by code symbols (words)
  - Certain codes can have more bits than needed to store just information
    - This excess representation space (redundancy) can be used for error control
    - State space can be separated into valid and invalid code words
  - Error-detecting code is set up so that the most likely errors produce invalid words

- HAMMING Distance
  - Associated with each error control model, we describe the concept of distance
  - Distance between code words is the number of failures needed to change one code word into another and hence cause an undetectable error.
  - The Hamming distance between two words is the number of bits in which they differ
    - ex.) 1011 and 0110 are a distance of 3 apart from each other
  - The minimum distance of a code S is the <u>minimum</u> of the Hamming distances between <u>all possible pairs</u> of code words in S

ex.)

Distance-2 code:	Distance-3 code:
11011	1101101
10001	0010101
01110	1011010
00000	0000000

- Hamming Distance, d, defines a code's Error-Detection & Correction Capabilities.
  - To Detect e Single-bit Errors, need a minimum distance e + 1 code
  - To Correct e Single-bit Errors, need a minimum distance 2e + 1 code

d	*Capability*
1	None
2	1-error detection, 0-error correction
3	2-error detection, 1-error correction

To achieve single error correction on a linear stream of bits, where
n are information bits, and
k are the additional parity bits needed,
We need to be able to identify (point to) any of n + k bits that might be wrong (n + k single errors) plus, identify the no error case.
Therefore: 2**k >= n + k + 1
e.g.) For n = 4, we would need k = 3 parity bits for a total of 7 bits

- Arrangement of check bits dispersed within a single linear word
  - The k check bits occupy the binary power positions: 1, 2, 4, 8, 16, etc.
  - Data bits occupy all other positions: 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 17, etc.
  - Note: Numbering scheme in diagram begins from 1 and reads from left to right
  - Check bits are parity bits that check the parity of a particular subset of data bits
    - Covered if that power of 2 would be used to encode that data bit's location

Parity bit	Bits Checked
1	1, 3, 5, 7, 9, 11 ....
2	2, 3, 6, 7, 10, 11 ....
4	4, 5, 6, 7, 12 ....
8	8, 9, 10, 11, 12 ....

If a data bit is erroneous, all the parity bits that check it will give a parity error
The intersection of the bits checked for all bad parities will locate the error
But, by arranging the parity bits such that they occupy binary power positions, the parity bits, in themselves, give the binary code of the bad data bit
e.g.) If bit 6 is bad, parity bits 2 and 4 will be bad, giving error location 2 + 4

Simple method for finding the incorrect bit is first to compute all the parity bits.
If all parity bits are correct, then there is no error.
Else, add up the value of all the incorrect parity bits, where
their value is simply their position in the power of two location of the word
The resulting sum is the position of the bad bit (which needs to be inverted)

Errors in the Parity Bits themselves will also be isolated in this manner.

- Computational Grand Challenges
  - Applications continue to grow in number, size, and complexity
  - A "Grand Challenge" as defined by Wilson in 1987:
    - Is a fundamental problem in science or engineering
    - Has potentially broad economic and scientific impact
    - Could be advanced by High Performance Computing (HPC) Resources
  - Includes Problems drawn from:
    - Prediction of Weather, Climate, and Global Change
    - Superconductivity
    - Design of Drugs
    - Human Genome
    - Speech / Vision
  - Modern-Day potential applications of Infinitely Free Computational Cycles
    - Model of Car Crash Safety
    - SemiConductor Modeling of Wafer-Scale Devices at high level of accuracy
  - The 3T Goal:
    - 1 TeraFlop/second of Processor Power
    - 1 TeraByte of Main Memory
    - 1 Terabyte/second of I/O Bandwidth

- The von Neumann Bottleneck
  - Traditional Architectures use a Single Memory Interface
    - All data and control information must pass through this single interface
    - Creates a Bottleneck making High Performance Computing difficult
  - “Non-Von” (typically parallel) machines address this problem in a variety of ways

- Parallel Processing vs. Sequential Processing
  - Sequential Algorithms are advantageous in that they have:
    - Simpler Representation
      - Programs are written in a simple language and execute linearly
    - Simpler Hardware Realizations
      - Smaller number of components (1 CPU, 1 Control, 1 Memory)
      - Simple interconnection scheme and timing synchronization
  - Concurrent Algorithms are advantageous in that they offer:
    - Speed
      - Given a period when electronic logic-element speed increased only 3X, Computer speeds increased 9X via the use of parallel architectures
    - Cost Effectiveness
      - Given a technology curve, it is Cheaper to use 2 CPU's in Parallel vs. trying to push (over-utilize) single CPU's performance to 2x faster

- Parallel processing can be viewed & implemented from various levels of complexity
  - Logic Level
    - Simultaneous processing of several bits (e.g. parallel CLA adder) using H/W
  - Instruction Level
    - Microscopic view:
      - Intra-Instruction Concurrency
      - A single instruction is divided into different phases or stages
      - Try to overlap phases of consecutive instructions
      - Possible implementation: Pipelining
    - Macroscopic view:
      - Inter-Instruction Concurrency
      - Simultaneous execution of several instructions
      - The program is sequential, but many steps can be done in parallel
      - Possible implementation: Multi-Function Processors
    - The two above techniques can also be combined
  - Program Level
    - Instruction Group Concurrency
    - Simultaneous execution of several processes (instruction groups)
      - e.g.) Several subroutines can be run at same time
    - Possible implementation: Multi-Processors, Multi-Computers

- Classification of Parallel Processors
  - Flynn identified four classes based on number of instructions and data handled
    - SISD: Single Instruction, Single Data
      - One instruction applied to one piece of data
      - Single Processor, Control Unit, and Memory
      - Executes instructions sequentially
      - Internal (microscopic) parallel processing could occur with pipelining
      - The SISD Class Includes:
        - Intel 8080, Intel 8086, DEC VAX 11/780, IBM 360/91, CDC 6600
    - SIMD: Single Instruction, Multiple Data
      - The same instruction is performed on many pieces of different data
      - Single Control Unit fetches & decodes instruction, then broadcasts it
      - Memory is usually local to each Processor
        - Each processor has direct access to only its own local memory
      - Processors are synchronized, but local memories have different contents
      - e.g.) Vector computer with 100 processors can do loop in one cycle:

```
      Do 20 I = 1, 100
   20 C(I) = A(I) + B(I)
```

      - Generally, programmer assists in the identification of parallelism:
        - e.g.) Above loop is coded as: C(1:100) = A(1:100) + B(1:100)

The SIMD Class Includes:

- ILLIAC IV, MPP, STARAN, Connection Machine, BSP
- Further Subclassification can be based on following characteristics
  - Complexity of the Control Unit
  - Computational Power of the Processor
  - Addressing method used by the Processors
  - Interconnection facilities between the Processors

- MISD: Multiple Instruction, Single Data
  - Non-sensible configuration

- MIMD: Multiple Instruction, Multiple Data
  - Many different instructions being applied to many different sets of data
  - Several Processors with separate Control Units.
    - Each processor:
      - Runs its own instruction sequence
      - Works on a different part of the problem
      - Communicates data to other processors
  - Memory is usually shared
    - All processors have direct access to all of the memory
    - Sharing memory for data, OS system code, etc. reduces costs
    - But, Memory Contention can be severe when executing shared code
  - Instruction streams generally independent & processors not synchronized
    - Processors may have to wait for other processors or for access to data
  - The MIMD Class Includes Multiprocessors and Multicomputers
    - e.g.) Cm*, Cmmp, CRAY XMP, IBM 370/168, IBM 4381, IBM 3090
    - Subclassification characteristics include:
      - Degree of Coupling of Processor Units and Memories
      - Homogeneity of the Processing Units
  - Example of an MIMD: Tightly Coupled Symmetric Multiprocessor (SMP)
    - Current commercial versions have 2 to 6 identical processors
    - All processors share memory and are controlled by common OS
    - SMPs look like a "single system" but provides more computing power
      - Users may not even be aware that system has multiple processor
    - Major Features of an SMP include:
      - Two or more identical processors
        - Each has independent, identical Instruct. execution capabilit
      - All addresses are equally accessible to all processors
        - All processors see same response time to all addresses
      - Systems software is equally accessible to any Processor
        - Any processor can execute any part of the OS

- Program execution floats freely from one processor to another
  - Application may migrate unpredictably between processors
  - Processors have a common work queue
    - Any process can run on any processor
- Synchronization must occur at granularity of an instruction
  - SMP is Tightly coupled, so typically cannot be asynchronous
  - Other MIMDs are loosely coupled
    - Synchronized at higher granularity, e.g., at process level

- We will discuss two Fast Processor Techniques:
  - Pipelining: Microscopic Concurrency
    - Decompose the Instruction into a sequence of subprocesses
  - Vector / Multiple Function Unit Processors: Macroscopic Concurrency
    - Have several functional units, each performing a different instruction

- Pipelining:
  - Key implementation technique used to make fast CPUs.
  - Enables multiple instructions to be overlapped in execution.
  - Exploits parallelism among the instructions in a sequential instruction stream
  - Transparent to programmer

Analogy: Assembly line
  - Work to be done in an instruction is broken into smaller pieces
  - Each piece takes a fraction of the time needed for the entire instruction
  - A pipeline is partitioned into stages or segments
  - Each stage in the pipeline completes a part of the instruction
  - Because pipe stages are hooked together, all stages must operate in lock-step.
  - Time required per step down the pipeline is determined by the slowest pipe stage.

- Pipeline designer's goal:
  - Balance the length of each of the pipeline's stages
  - Reduce stalls (caused by hazards)
- Throughput:
  - Determined by how often an instruction exits the pipeline
  - Number of instructions output per clock cycle
  - For an instruction stream consisting of N instructions:
    - Each instruction is divided into K equal segments (stages)
    - A non-pipelined machine would need: NK time steps
    - A pipelined machine would need:
      - K time steps for the first instruction (assuming pipeline was empty)
      - One time step for each of the remaining (N-1) instructions
      - Total = K + (N - 1) time steps
    - Speedup = NK / (K + N - 1)

Speedup from pipelining (asymptotically) equals the number of pipe stages
  For N >> (K - 1), denominator approaches N
  So, Speedup = NK / N = K
If stages are perfectly balanced, and no stalls, high stride occurs (ideal conditions),
  Throughput(pipelined) = Throughput(non-pipelined) x Number of pipe stages

- Interesting Note: Although pipelining increases throughput,
  - The total time needed by each instruction remains the same.
    - e.g.) For a five stage pipeline, each instruction still takes five clock cycles
      - On each clock cycle:
        - Hardware is executing some part of five different instructions
        - An instruction is completed and exits the pipe, and another enters
  - In fact, Total time needed for each instruction may actually increase !
    - Overhead is needed to control the pipeline
    - Latches are required between pipe stages, adding setup and propagation time
  - The increase in instruction throughput means that a program runs faster,
    - even though no single instruction runs faster.

- Pipelining can be implemented in various places in the hardware:
  - Memory: Interleaved memory banks
    - Partition Memory cycle time into access + wait
    - Overlap M2 access phase with M1 wait phase
  - Arithmetic Logic Unit: Math operations are phased
    - Many alternatives are possible
      - e.g.) Floating Point Addition can be partitioned into the following steps:
        1) Compare Exponents
        2) Align Mantissas
        3) Add / Subtract Mantissas
        4) Normalize Result
  - Control Unit: Instruction fetch, decode, execute
    - Although different partitions and granularities are possible, the Basic Steps of Instruction Execution are:
      1) Fetch Instruction (FI)
         Read Program Counter and fetch instruction from memory
      2) Decode Instruction and Address (DA)
         Determine operation and effective address of operand(s)
      3) Fetch Operand (FO)
         Fetch argument(s) associated with the instruction to be performed
      4) Execute Instruction (EX)
         Perform the operation on the operand(s)

- **Hazards:**

Pipelining changes the normal sequential nature of instruction execution
Their relative timing is changed by the overlapping of their execution.
This can introduce hazards due to interaction between (uncompleted) instructions
Hazards prevent next instruction from executing during its designated clock cycle.
Hazards reduce the pipeline's performance from the ideal speedup possible

- Three types of hazards:

1) Data hazards: Instruction cannot be performed until operands are available.
This can arise when an instruction depends on the results of a previous instruction in a way that is exposed by their overlapping in the pipeline

2) Control hazards: Next instruction to be executed is determined by the previous
This arises from the pipelining of branches and other "decision-point" instructions that change the program counter.

3) Resource hazards: Instruction cannot be performed until resources are available
Arises when not enough of the right kind of hardware is available

Hazards may make it necessary to stall the pipeline, thereby breaking its stride

Stalls:

In a Pipelined Machine, There are multiple instructions under execution at once.
Typically, when an instruction is stalled:
All instructions later in the pipeline than it are also stalled
Instructions earlier than it can continue
No new instructions are fetched during the stall.

- **Data Hazards:**

Occur when the order of access to operands is changed by the pipeline
(versus the normal order encountered by sequentially executing instructions)
Two instructions can create a hazard by writing and reading the same variable

- Example:

```
R1 = MUL R2, R3
R4 = ADD R1, R5
R8 = SUB R6, R7
```

The MUL instruction has a target, R1, that is the source of the ADD instruction
It is possible that the MUL instruction does not write R1 until after ADD reads R1
ADD starts Operand Fetch (FO) before MUL completes Execute (EX) step
Unless precautions are taken, ADD will use an old value of R1
Non-deterministic behavior could also result:
e.g.) If interrupt occurs b/w MUL and ADD, then ADD <u>will</u> get the new R1

- The most common solution to Data Hazards is a Hardware Pipeline Interlock.
  - Pipeline Interlock detects a Hazard and Stalls the pipeline until hazard is cleared.
  - Pipeline is stalled beginning with the instruction that wants to use the data until the earlier sourcing instruction completes and produces it.
  - In previous example: ADD and following instructions are stalled until MUL writes R1
  - This delay cycle (pipeline stall) creates a "bubble" in the timing diagram

- Example of Data Hazard Causing Pipeline Stall

  S1: X = X + 1
  S2: Z = X + Y [S2 has a data hazard on X]
  S3: A = B + C
  S4: J = K + L

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
S1	FI	DA	FO	EX						
S2		FI	DA	--	FO	EX				
S3			FI	--	DA	FO	EX			
S4				--	FI	DA	FO	EX		

Minimizing Impact of Data Hazards
  - Pipeline stalls represent lost computing cycles (essentially a No-Op)
  - Compiler could try to schedule the pipeline to avoid these stalls
  - Code sequence is rearranged to eliminate (or at least reduce) the hazard
  - Delayed Load: A load requiring that the following Instruction not use its result
  - Example:

R1 = MUL R2, R3		R1 = MUL R2, R3
R4 = ADD R1, R5	rearranged to:	R8 = SUB R6, R7
R8 = SUB R6, R7		R4 = ADD R1, R5

- Data hazard Classifications are Named by the ordering that must be preserved
  - Consider two instructions: statement i and J, with i occurring before J.
  - The three possible data hazards are:
    - RAW (Read After Write):
      - True, Flow Dependency
      - J tries to read a source before i writes it, so J incorrectly gets old value
      - This is the most common type of hazard as seen in pipeline.

        i R1 = R2 + R3
        J R4 = R1 + R6

      - A smart compiler can perform some re-arranging to reduce stalls, but chances are, not all RAW dependencies can be eliminated
      - RAW dependencies are artifacts of the program.

- WAR (Write After Read):
  - AntiDependency: A mirror image of flow dependence
  - J writes a destination before it is read by i, so i incorrectly gets new value
    - Case 1:
      - J could be started at same time as i, but finish faster
      - ex) J is an ADD while i is a MUL instruction

        i R1 = R2 x R3
        J R3 = R3 + 1

    - Case 2:
      - J could be started sooner than i
      - A Hazard delays i, but J is hazard free and allowed to proceed
      - ex) RAW stalls usage of operand; later instruction overwrites it

        z R2 = R1 + R8
        i R9 = R4 x R2
        J R4 = R7 + R8

        RAW on R2 between i and z causes z to stall
        But, J has no dependencies on R7 and R8, so proceeds
        J may complete and write into R4 before R4 is ready by i
          Result R9 will be incorrect

  - WARs can be prevented by buffering source operands (Renaming)
    - In above example, if i is stalled, store value of R4 in Buffer
    - When i executes, it reads Buffered value of R4
      - R9 will still be correct even if J completes before i

- WAW (Write After Write):
  - Output Dependency
  - J tries to write an operand before it is written by i.
  - The writes end up being performed in the wrong order
  - Leaves the value written by i rather than the value written by J
  - Example:

    i R4 = R3 + R1
    J R4 = R7 + R8
    z R9 = R4 + R5

    Assume a dependency stalls i, but not J
      So i finishes (and writes into R4) after J
    z will then read the wrong value of R4

  - Buffering (or Variable Renaming) can be used to alleviate WAW

    i R4 = R3 + R1
    J B4 = R4 = R7 + R8
    z R9 = B4 + R5

- Control Hazards:
  - Caused primarily by conditional branch instructions
  - Can change the normal contiguous, sequential Instruction stream flow
  - Results in a delay in knowing which instruction really needs to be executed next
  - Easiest solution is to just introduce stalls

- Example of Control Hazard Causing Pipeline Stall

  Assume pipeline is simply stalled and flushed on branch.

  Note: Using this conservative approach, control hazard stall > data hazard stall

  S1: X = X + 1
  S2: IF A > 10 THEN GOTO S4
  S3: A = B + C [Control hazard on S2]
  S4: J = K + L [Control hazard on S2]

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
S1	FI	DA	FO	EX						
S2		FI	DA	FO	EX					
S3			FI	--	--	FI	DA	FO	EX	
S4				--	--		FI	DA	FO	EX

- Example of Combined Data and Control Hazards

  Assume pipeline is simply stalled and flushed on branch.

  S1: X = X + 1
  S2: IF X > 10 THEN GOTO S4 [Data Hazard on S1]
  S3: A = B + C [Control hazard on S2]
  S4: J = K + L [Control hazard on S2]

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11
S1	FI	DA	FO	EX							
S2		FI	DA	--	FO	EX					
S3			FI	--	--	--	FI	DA	FO	EX	
S4				--	--	--		FI	DA	FO	EX

- Minimizing Impact of Control Hazards

  Approximately 30% of all instructions are jumps

  Jumps can be classified into three categories:

  1) Unconditional
  2) Conditional
  3) Loop

  Loop instructions are a special case of conditional Jumps

  These are known in advance to almost always be taken

Amount of stalling due to branches can be reduced by using various techniques:

- Just assuming that the jump will not be taken and continue filling pipeline
  - Requires pipeline flush if jump really is taken
  - Also, need to undo any pre-executed effects of (wrong) instruction
- "Guessing" Statically (During Compile Time) which path will be taken
  - Provide an extra bit for each branch instruc that can be set by the compiler
    - Bit instructs processor as to the most likely direction of the branch
    - Bit (Prediction) is not modified during program execution
  - Statistically, about 60% of all Branches are taken
  - Stereotypical Behaviors can also be identified for certain branch types
    - Loop conditional jumps will almost always be taken
    - Jumps to System error routines will almost never be taken
    - Examples:

```
Loop:   X(I) = Y(I)              IF A = 0 THEN sys_error(div_0)
        I = I + 1                C = D / A
        If (I < 10) GoTo Loop
```

- "Guessing" Dynamically (During Run Time): Dynamic Branch Prediction
  - The Static branch prediction scheme is too rigid
    - Stereotypical behavior does not account for individuality of branches
  - Dynamic Hardware Predictors base guess on the behavior of each branch
    - Predictions are allowed to change for a branch during execution
    - Uses the past outcome of a branch as a predictor for its future path
  - Varying degrees of Sophistication are Possible with increasing costs
    - Branch Prediction Buffer or Branch History Table can be used
    - One Bit Prediction Scheme
      - Associative memory stores branch instruction address plus 1 more bit
      - Extra bit tells whether the branch was last taken or not
      - Problem: Even if branch is almost always taken, prediction will be wrong twice, not once, when branch is not taken.
      - e.g.) Loop-Back Branch at the end of a loop
        - Assume branch is taken 9 times in a row, then is not taken
        - Prediction scheme will be wrong for the first and last iterations
          - Last iteration is wrong because it was taken 9 times in a row
            - History says branch, but it is not taken on 10th time
            - The loop exit iteration is typically mis-predicted
          - First iteration is wrong too because of last loop's bit setting
            - Last time loop was executed, the loop exit occurred
            - Mis-prediction occurs because history says not to branch
      - Thus, prediction accuracy is 80% for a loop that is taken 90%
        - Two incorrect predictions and eight correct ones

- Two Bit Prediction Scheme
  - Prediction must be wrong twice before it is changed
  - Branch that strongly favors taken or not will be mispredicted only once
  - Uses a simple Finite State Machine (Patterson figure 6.53, pg 502)
- Amount of History recorded should be small
  - Need to keep implementation cost of hardware prediction logic low

- Prefetching on both paths of a branch
  - Requires two pipelines in the H/W for parallel execution along both paths
  - Complicates control structure of pipeline
  - Used only on highest speed, highest cost machines
    - Similar in cost and technique to Carry Select Adder
    - Half the answers computed will eventually be discarded
  - VLIW Architectures often fetch both paths speculatively (see page 82)

- Compiler (re)-scheduling of instructions (Delayed Branching)
  - Splits jump into test and action part
  - Inserts useful instructions instead of no-op stalls b/w test and next instr.
  - These instructions would have to be done regardless of branch outcome
  - Requires static analysis to identify and exploit these possibilities
  - Delayed Branch: Make successor of Branch Instructions valid and useful
    - Similar to Delayed Load
  - Location following a branch instruction is called a Branch Delay Slot
    - Instructions in the delay slots are always fetched
    - They can be designed to fully execute whether or not branch is taken
  - Objective is to place useful instructions in these Branch Delay Slots

Example:

X = Y + Z	IF B < C
IF B < C	X = Y + Z
THEN A = B + C	THEN A = B + C

If useful instruction cannot be moved into Branch Delay Slot, use a NoOP
  - Compiler can utilize a one branch delay slot in 85% of cases

- Performance of Different Control Branch Handling Schemes
  - Assume a 5 stage pipeline with maximum speedup of 5X if no Stalls

Scheduling Scheme	Pipeline Speedup over Non-Pipelined
Stall Pipeline	3.5
Predict Taken/Not Taken	4.4
Compiler rescheduling	4.6

- **Limitations of Pipelining**

    Pipeline speedup potential is limited by the number of pipeline segments

    Number of segments is limited by the total number of separate functions into which the instruction or machine operation can be broken (typically 4-8).

    Pipeline speedup also limited by stride

    - Stride is an unbroken consecutive string of instructions through pipeline
    - Once stride is broken (e.g. by a branch flush), pipeline needs to reload
    - Pipeline penalty (startup) is higher for smaller N and larger K
    - If only a small number of instructions are processed consecutively, N is small
        - So, cannot assume N >> (K - 1) and that Speedup = NK / (K + N -1) = K
        - For example, for N = 5 instructions, and a K = 8 stage pipeline:
            - Speedup = (5) (8) / 12 = 3.3
    - Probability for long consecutive strings highest in matrix math computations

- **Macroscopic Instruction Parallelism**

    Try to process several (whole) instructions concurrently vs. pieces of instructions

    Can actually change the order of instructions relative to how they appear in prog.

    More complicated than Pipelining (Microscopic Instruction Parallelism)

    - Requires the use of multiple function units (Resource Hazards)
        - At any one time, many instructions may be in their execute stage
        - Does not happen in pipelining (Instructions are Phased)
    - Requires more restricted data dependency analysis (Data Hazards)
        - An instruction later in the stream might go before an earlier instruction
        - Does not happen in pipelining (Instructions are only Overlapped)

- **Concurrent Execution of Sequential Algorithms:**

    A procedural-oriented computer language injects an "apparent" sequentiality

    - One instruction must "apparently" be completed before the next is initiated

    The "apparent" view enables some hidden concurrency to be built into architecture

    Goal: Obtain faster execution while retaining advantages of sequential represent.

    Approach: Remove any unnecessary sequentiality from the software program

    A sequential algorithm has:

    - Inherent Sequentiality
        - An ordering of operations which are an implicit part of the algorithm
        - These must be preserved as a fundamental part of the S/W program
        - Changing the order of these instructions will alter what was intended
    - Artificial Sequentiality
        - Injected by the semantics of the software specification of an algorithm
        - Most languages do not enable the programmer to specify concurrency
        - Temporary variables contribute to sequential step appearance

By eliminating artificial sequentialities, execution can be accelerated

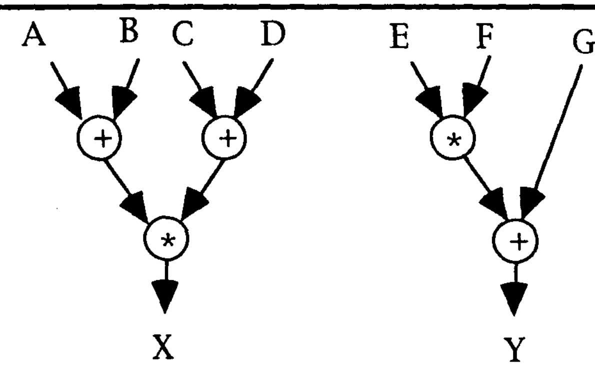
Continues to preserve the required dependencies for correct behavior

Maintains "apparent" sequentiality while transparently using parallelism

Identification of inherent sequentiality requires more detailed hazard analysis

T1 = A + B
T2 = C + D
X = T1 * T2
T1 = E * F
Y = T1 + G

Five Time Steps



Two Time Steps

- Multi-Function Units

Augmenting H/W w/ multiple functional units enables parallel proc. of instructions

Removes artificial sequentiality whenever possible (i.e. resource available)

Requires resource hazard analysis

Keep parallel processing transparent to programmer

Requires data hazard analysis to preserve inherent sequentiality

If resource is available and no data hazards are present, control unit can issue and begin executing a later instruction even before an earlier one is started

The control unit performs "lookahead" to identify instructions to process in parallel

Look-Ahead Control unit needs to perform:

- Detection: Determine which instructions can be executed concurrently
  A machine Independent Task
- Scheduling: Assigning concurrently executable instructions to FU
  Must factor into account the specific number of FUs on target machine

Degree: The number of instructions scanned ahead of the current instruction

Multiple degrees of "lookahead" are possible

Higher degrees enable more potential speedup but are more complicated

We assume a simple single instruction lookahead issuing scheme:

Control unit issues consecutive instructions until a hazard is detected.

At that point, all issuing stops until the blocked statement can execute.

An instruction can be issued if:

1) No data dependency is detected on any instruction currently executing.

AND

2) The appropriate type of resource (function unit) is available.

- **Example:**

High-Level Language Source	Compiler Generated Register Transfer Code
A = (B + C) * (D + E)	S1: R1 = B + C
F = G + H + I + J	S2: R2 = D + E
H = K * L	S3: A = R1 * R2
	S4: R3 = G + H
	S5: R4 = I + J
	S6: F = R3 + R4
	S7: H = K * L

- CASE 1:

One adder and one multiplier unit available.

Time	1	2	3	4	5
Adder	R1=B+C	R2=D+E	R3=G+H	R4=I+J	F=R3+R4
Multiplier			A=R1*R2		H=K*L
Hazard:	S2:adder	S3:R2	S5:adder	S6:R4,adder	

- CASE 2:

Two adders and one multiplier unit available

Time	1	2	3	4	5
Adder 1	R1=B+C	R3=G+H	F=R3+R4		
Adder 2	R2=D+E	R4=I+J			
Multiplier		A=R1*R2	H=K*L		
Hazard:	S3:R1, R2	S6: R3, R4			

- Note: There are practical limits to "transparent" parallel processing
  - Decreasing marginal rates of return occur as more functional units are added
    - Inherent sequentiality of source code algorithm is the ultimate bottleneck
  - High degree of lookahead needed to utilize large number of functional units
    - Need to continue issuing later instructions even if earlier one is blocked
      - Don't want to hold up Instruc(k) because Instruc(i)'s FU is busy
      - Instruc(k)'s FU may be available
    - One method that allows this is Virtual Functional Units
      - Allows examination of instructions to not be blocked due to busy FU
      - Each FU is augmented with a queue of Virtual Functional Units
      - Instruc(i) will be dispatched assuming:
        - 1) A FU or a Virtual FU is available
        - 2) There are no data dependency hazards
      - VFUs will not necessarily allow Instruc(i) to be completed earlier
        - Execution of Instruc(i) will still require a real FU eventually

- VLIW: Very Long Instruction Word
  - An alternative architecture for exploiting instruction-level parallelism
  - VLIW Architectures are characterized by:
    - A Processor that contains a large number of Function Units (FU)
    - An Instruction containing different fields with different OP codes for each FU
    - Resources that are completely & independently controlled by the VLIW Word
    - A control mechanism that exerts fine-grain control over all machine resources
  - The Instruction set of a VLIW consists of simple RISC-like instructions
  - VLIW Instruction Word is typically 256 to 1024 bits long
    - Packed into a single VLIW instruction are several primitive instructions
    - These instructions can be grouped together for independent, parallel execution
    - The entire set of instructions is dispatched to the FUs for parallel execution
  - Exploiting the Full Capability of a VLIW CPU is the Compiler's Responsibility
  - Compiler Must:
    - Be intelligent enough to decide how to build the very long words
      - Assemble many primitive operations into a single "instruction word"
    - Group together independent instructions executable in parallel
      - Guarantee no dependencies between instructions that issue at same time
    - Keep as many of the FUs busy by filling all the available operation slots
      - But also ensure that there are no resource hardware hazards
  - The VLIW's Static Scheduling vs. The SuperScalar's Dynamic Scheduling
    - Most other SuperScalar processors DO perform dynamic scheduling/reordering
      - SuperScalar architectures include Intel i860, Sun UltraSPARC
      - Since the ILP is handled by the H/W, it is more complex than a VLIW's
    - Thus, Modern CPUs have developed very complicated hardware units for:
      - 1) Rearranging Instructions at run time for effective Out of Order Execution
      - 2) Performing Branch Prediction
    - The VLIW Architecture overcomes the two above complications by:
      - 1) Having compiler pack several RISC instructions into one long word
        - Processor can then take unpack operations without further analysis
          - Processor simply gives each operation to an appropriate FU
        - These instructions are already certified to be executable in parallel
        - Processor H/W does not need to have the ability to detect and
          - schedule the parallel operations in Real Time.
      - 2) Eliminates Branch Prediction by executing all branch outcomes
        - After true outcome of branch is known, invalid results are discarded
    - All Instruction Level Parallelism (ILP) is handled completely by the compiler
      - No dynamic scheduling nor reordering of operations is performed in H/W
      - The VLIW control logic has less responsibility, and is therefore simpler
    - => VLIW CPUs have fewer gates and are better scalable than RISC CPUs

=> VLIW Architectures have been described as a natural successor to RISC
- Main Advantage is its simplicity in H/W structure and Instruction Set
- Takes RISC to its next level of simplicity
  - Removes dynamic scheduling and reordering from the control hardware
- Moves complexity from the Hardware to the Software (i.e., the compiler)
  - Eliminates the complicated instruction scheduling and parallel dispatching that occurs in the H/W of most modern superscalar microprocessors.
  - Allows even more simpler, faster processors than ordinary RISC.
- Hardware can be smaller, cheaper, and require less power to operate

Limitations of VLIW architectures:
- Binary incompatibility across implementations with varying number of FUs
- Code size bloat due to aggressive scheduling policies

• Example VLIW CPU: Transmeta Crusoe
- Crusoe performs translation (code morphing) of x86 instructions into VLIW words
  - A Software pre-processor layer assembles the VLIW words
    - The pre-processor performs the code morphing (effectively, emulation)
  - Transmeta claims a transistor count reduction of 75% over Intel Approach
- Code Morphing Software can be Upgraded Easily
  - Pre-processor S/W located in non-volatile, reprogrammable Flash memory
  - Pre-processors could be designed to emulate other processor architectures
  - Can also be upgraded to enhance performance without replacing the CPU
- Extremely Low Power Consumption (for mobile applications)

• Comparing CISC CPUs (e.g., x86) vs. VLIW CPUs (e.g., IA-64)

x86 (CISC):
1) Uses complex, variable-length instructions processed one at a time.
2) Reorders and optimizes the instruction stream at Run Time
3) Tries to Predict which way branches will fork
   - Speculatively executes instructions along the predicted path
4) Loads data from memory only when needed
   - Tries to find the data in the cache first

IA-64 (VLIW):
1) Uses simpler, fixed-length instructions bundled together in groups of 3
   - Three instructions are packed into a 128-bit Long Instruction Word
2) Reorders and Optimizes the instruction stream at Compile Time
3) Speculatively Executes instructions along Both Paths of a Branch
   - Then discards the results it doesn't need
4) Speculatively loads data before it's needed
   - Still tries to find the data in the cache first