Law/Theorem	Law of Addition	Law of Multiplication
Identity Law	$x + 0 = x$	$x \cdot 1 = x$
Complement Law	$x + x' = 1$	$x \cdot x' = 0$
Idempotent Law	$x + x = x$	$x \cdot x = x$
Dominant Law	$x + 1 = 1$	$x \cdot 0 = 0$
Involution Law	$(x')' = x$	
Commutative Law	$x + y = y + x$	$x \cdot y = y \cdot x$
Associative Law	$x+(y+z) = (x+y)+z$	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$
Distributive Law	$x \cdot (y+z) = x \cdot y+x \cdot z$	$x+y \cdot z = (x+y) \cdot (x+z)$
Demorgan's Law	$(x+y)' = x' \cdot y'$	$(x \cdot y)' = x' + y'$
Absorption Law	$x + (x \cdot y) = x$	$x \cdot (x + y) = x$

Simply the logical Expression function

$F = AB + A(B+C) + B(B+C)$

$F = AB + AB + AC + BB + BC$ (Distributive Law)

$F = AB + AC + BB + BC$ (Idempotent Law)

$F = AB + AC + B + BC$ (Idempotent Law)

$F = AB + AC + B + BC$

$F = AB + AC + B$ (Absorption Law)

$F = AB + B + AC$

$F = B + AC$

Chapter 1

Introduction

In this Chapter

- Digital systems
- Number systems
- Digital circuits
- Computer organization
- Computer architecture
- Computer security
  - Security through hardware
- Other chapters

Digital Systems

- Computers, iPad, cell phones, digital cameras, etc. created digital revolution, changing ways we:
  - Communicate, work, are entertained, shop
- Digital systems are in everything we see and use
  - Cars, grocery checkout equipment, utility meters, set-top, boxes, emergency equipment, etc.
- Thus, more data is created, processed, stored, transmitted, and accessed
  - Results in demands for more powerful computers
    - Personal computers
    - Large computers used in
      - E-commerce, banking, search engines, research
  - Create more chances for unauthorized access to data and information

Impact of Technologies (1)

1. Impact of innovations in integrated chip (IC) technologies
   - Moore's law:
     - 35% increase in transistor density per year
     - 40% to 50% increase in transistor count per year
     - Has been used as a guide to design each next generation of microprocessors that revolutionized personal computers
     - Also resulted in increased power use and heat dissipation

Impact of Technologies (2)

2. Impact of innovations in application developments
   - Revolutionizing the way digital systems are designed
   - Digital circuits are described in HDL
   - CAD tools simulate (validate) HDL descriptions
   - CAD tools synthesize (translate) HDL descriptions to circuits

Number Systems

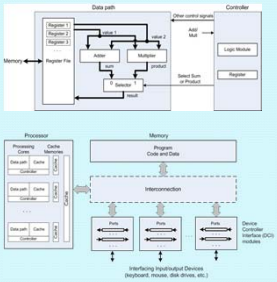
- Digital circuits make logical decisions as True or False logic values
- A voltage range defines each logic value.
  - E.g., 5-volt power source
    - 2.4 to 5V as True
    - 0 to 0.8V as False
  - Lower voltage sources help save power in battery powered devices
- True and False values as 1 and 0 form binary numbers to represent
  - Characters
    - 8-bit, or 256 ASCII codes
    - 16-bit numbers, or over 65,000 Unicodes
  - Pixels to create images
  - Audio and video data
  - Integer and real numbers used in computations

Digital Systems as Von Neumann machines

- Computer system that contains
  - One or processors
    - Each consisting of one or more *processing cores* (CPUs)
  - Memory
  - I/O devices
  - OS and application programs
- Embedded system that is
  - A complete system as circuit board or ASIC or FPGA , known as SoC
    - Contain CPU(s) and memory
    - Dedicated software known as firmware
  - May include transmitter/receiver modules
  - May include signal conversion modules (converters)
    - Analog-to-Digital (A/D)
    - Digital-to-Analog (D/A)
  - Applications:
    - Cell phones, digital camcorder, etc.
    - Host device controller interface
      - E.g., USB

Von Neumann Computer

- Processing core (CPU) consists of
  - Data path that includes
    - Digital circuit modules perform computations
    - Storage modules store computed data
  - Controller that orders data path operations
- As microcomputer that includes
  - Multicore processor (s)
  - Memory
  - Interconnection medium
  - I/O devices
  - I/O device controller and interface
  - Potential for bottleneck between faster processor and slower memory



Digital Logic Design (1)

1. Requires logic expressions
   - Example: $f = ((\text{NOT } a) \text{ AND } b) \text{ OR } c$
     - NOT, AND, and OR indicate Boolean logic operators
   - Evaluation
     - Suppose a = 0, b = 1, and c = 0

```
f  = (NOT 0) AND 1) OR 0
   = (1 AND 1) OR 0
   = 1 OR 0
   = 1
```

What function does f perform?

f is 1 when a, b, c forming a 3-bit number is prime

   - Requirement: Minimal expressions to reduce hardware

Digital Logic Design (2)

2. Requires logic gates
   - Gates perform logic operations
   - Modern gates are built as CMOS circuits
     - Complementing MOS transistors reduce power consumption and heat dissipation
   - CMOS chips can be fan-cooled when hot
     - Thus, enabled personal computers
     - Exclusively used today in all types of digital systems

Digital Logic Design (3)

3. Require logic circuits
   - Logic circuits implement logic expressions
   - All NAND or all NOR gates show implementation details
   - Two types of digital circuits:
     1. Combinational: Outputs are generated concurrently
        - Outputs depend on current inputs only
        - E.g., Adders and selectors
     2. Sequential: Outputs are generated in sequence (in steps)
        - Output depends on current inputs as well as previous inputs
        - Uses combinational circuits to generate outputs
        - E.g., registers, counters, and control units

Effect of Increased Power Consumption and Heat Dissipation

- Examples,
  - 2 watts for Intel 80386 processor
  - 130 watts for 3.3 GHz (Giga Hertz) Intel Core i7 processor
    - 65 times more watts
- Problem: Harder to make processors any faster
  - Affects computer organization, programming model, and OS
  - Current Solutions:
    - Divide tasks into subtasks using multithreaded programming
    - OS assigns processing cores to perform subtasks
      - Creates thread-level parallelism
    - Use multiprocessor systems to perform many independent and dependent tasks faster
      - Modern Supercomputers for scientific computations
      - Warehouse-scale computers for
        - Interactive applications (Facebook, Google, etc.)
        - Large-scale storage and computing (e.g., cloud computing)

Computer Organization

Specifies implementation details:

- Circuit and their physical relationship that makeup
  - Processing core
    - data path organization
    - Example: 32-bit Intel vs. AMD processors
      - Two different data paths but same instruction set
  - processor,
  - memory,
  - I/O device controller and interface
  - Interconnection of a computer components
- Memory organization
  - Cache, SDRAM, multi-channel, etc.

Computer Architecture (1)

Specifies design concepts:

- Pipelining
  - Concept of an assembly line

Stage 3: Install wheels			Car1	Car2	Car3	. . .
Stage 2: Install doors		Car1	Car2	Car3	. . .	. . .
Stage 1: Install Engine	Car1	Car2	Car3	. . .	. . .	. . .
Time slots (e.g., 10-minutes slots)	1	2	3	4	5	. . .

If each stage is 2 minutes, how many cars built in a year (assume perfect scenario)?

260,000+

  - E.G., pipelining CPU data path

Execute			Load r1, B	Load r2, C	Add r3, r1, r2	Store A, r3
Decode		Load r1, B	Load r2, C	Add r3, r1, r2	Store A, r3	. . .
Fetch	Load r1, B	Load r2, C	Add r3, r1, r2	Store A, r3	. . .	. . .
Time (T)	1	2	3	4	5	6

What can go wrong?

Computer Architecture (2)

- Parallelism
  - Single Instruction Multiple Data (SIMD)
    - E.g., Intel's Streaming SIMD Extension (SSE) instruction set
    - E.g., AMD's 3DNow instruction set
    - Also in GPUs
  - Instruction level parallelism (ILP)
    - Also called Superscalar processor (i.e., processing core)
    - E.g., processing cores in Intel Core i7
  - Multiple Instructions Multiple Data (MIMD)
    - Multicore Processors
      - E.g., Intel Core i7
    - Multiprocessor Systems
      1. *Shared memory*: Processors communicate using memory
      2. *Message passing*: Processors communicate by sending/receiving messages

Computer Security

- Protecting digital assets (programs and data) from malware attacks
- Protecting digital assets from unauthorized access by employees
- Affects all
  - individuals, government, business organizations
- Potable devices also subject to *physical attacks*
- Application examples
  - Secure data storage
  - Secure communication
  - Secure e-commerce
  - Etc.
- How hardware can help?
  - Hardware more secure than software and disk storage
  - Computer security through hardware
    - Secure co-processor
      - E.g., Crypto-processor
    - Secure processor
      - Also, supports secure execution
      - Also, guards against physical attacks

Remaining Chapters (1)

- Combinational circuits
  - Design methodology for small circuits (Ch2)
    - Circuits with fewer (e.g., ≤ 4) inputs
  - Design methodology for large circuits (Ch3)
    - Circuits with many inputs (e.g., 32-bit Adder)
- Sequential circuits
  - Basic core modules (Ch4)
    - Basic storage elements
  - Design methodology for small circuits (Ch5)
    - Registers, counters, etc.
  - Design methodology for large circuits (Ch6)
    - Data paths and control units

Remaining Chapters (2)

- Memory (Ch7)
  - Memory organization
  - Memory timing
- Processing core (CPU) design, a very complex sequential circuit (Ch8)
  - CPU data path and control
- Microcomputer organization, history and modern designs (Ch9)
  - CPU, memory, I/O device interconnections
  - Device communication
- Memory system (Ch10)
  - Cache memory organization
  - Main memory as physical memory
  - Disk space as virtual memory
- Computer security for computer architects (Ch11), an introduction
  - Threat models
  - HW and SW security models, policies, and mechanisms
  - Trusted computing base as secure co-processor or secure processor

Chapter 2

COMBINATIONAL CIRCUITS

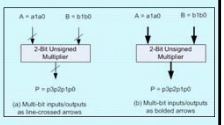
SMALL DESIGNS

In this Chapter

- Small Combinational Circuits
  - Fewer inputs (e.g., ≤ 4 inputs)
  - Circuits modeled as Truth Tables
  - Circuit minimization techniques
- Circuit implementation options
  - NANDs only
  - NORs only
- Timing diagram
  - Signal propagation delay
  - Understating signal hazards ("glitches")
- Other types of logic gates
- Design examples
- Introduction to design with HDL

Small Combinational Circuits

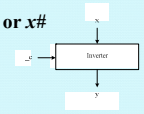
- Example: 2-bit unsigned multiplier, P = A * B
- Block diagram and truth table
  - Labeling of input and output signals
- Implementation options
  - LUT
    - Easier, slower, configurable
  - Logic circuit
    - Faster, less hardware



Inputs				Outputs			
a1	a0	b1	b0	p3	p2	p1	p0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

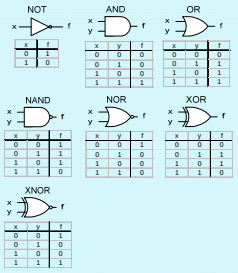
Signal Naming Standards

- Active-high signal polarity
  - 1 represents signal is active, asserted, enabled
  - 0, otherwise
  - E.g., signal labeled as *x* without a pre- or post symbol
- Active-low signal polarity
  - 0 represents signal is active, asserted, enabled
  - 1, otherwise
  - E.g., signal labeled as *_x, x', /x,* or *x#*
    - With a pre- or post-symbol



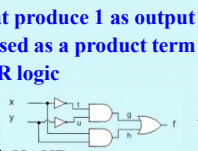
_c	x	y
0	0	1
0	1	0
1	0	0
1	1	1

Primitive Logic Gates with Truth Tables



SOP Expressions

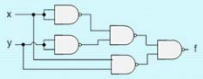
- Based on input values that produce 1 as output
- Each such input is expressed as a product term
- Circuit performs AND-OR logic



- Can be implemented with NAND gates
- DeMargan's theorems convert AND-OR circuit into NAND-only circuit

Theorem 1: $\overline{xy} = \bar{x} + \bar{y}$

Theorem 2: $\overline{x + y} = \bar{x}\,\bar{y}$



POS Expressions

- Based on input values that produce 0 for f (an output)
  - Same input values produce 1 for $\bar{f}$
- Find expression for f by complementing $\bar{f}$
- Each such input is expressed as a sum term

  $f = (x + \bar{y})(\bar{x} + y)$
- Circuit performs OR-AND logic

  SOP of $\bar{f} = \bar{x}\,y + x\,\bar{y}$

  $f = \bar{x}\,y + x\,\bar{y}$

  $f = (\bar{x}\,\bar{y})(x\,\bar{y})$
  - Can be implemented with NOR gates
- DeMargan's theorems convert OR-AND circuit into NOR-only circuit
- Also can use signal negation with Dual principle

  Dual of $\bar{f} = (\bar{x} + y)(x + \bar{y})$

Why minimize logic expressions

- Eliminates redundancies
- Requires fewer gates
- Fewer inputs per gates
- Less wire
- Less power usage
- Reduces circuit delay

How many gates and types for SOP?

Canonical SOP: 3 NOTs, four 3-input ANDs, one 4-input OR.

Minimal SOP: One NOT, two 2-input AND, one 2-input OR

Implement with NAND gates

Canonical SOP: $f = \bar{x}\,\bar{y}z + x\,\bar{y}z + xy\bar{z} + xyz$

Minimal SOP: $f = \bar{x}z + xy$

Implement with NOR gates

Canonical POS: $f = (x + y + z)(x + \bar{y} + z)(\bar{x} + y + z)(\bar{x} + y + \bar{z})$

Minimal POS: $f = (x + z)(\bar{x} + y)$

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Show mathematically

- NAND implementation

  $f = \bar{\bar{f}} \Rightarrow f = \overline{\bar{x}\,\bar{y} + x\,y}$ (Theorem 2: $\overline{x + y} = \bar{x}\,\bar{y}$) $\Rightarrow f = \overline{(\bar{x}\,\bar{y})(x\,y)}$

  NAND NAND

  NAND, again
- NOR implementation

  $f = (x + \bar{y})(\bar{x} + y) = \overline{\overline{(x + \bar{y})(\bar{x} + y)}} = \overline{(x + \bar{y}) + (\bar{x} + y)}$

  NOR NOR

  NOR, again

Karnaugh map (K-Map) Layouts

yz:	00	01	11	10
x: 0	0	1	3	2
1	4	5	7	6

2 × 4

4 × 2

z:	0	1
xy: 00	0	1
01	2	3
11	6	7
10	4	5

4 × 4

yz:	00	01	11	10
wx: 00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

Canonical Expression

$g = x\bar{y} + \bar{x}z + xyz$ Non-Canonical

$g = \bar{x}\bar{y}\bar{z} + \bar{x}\bar{y}z + xy\bar{z} + xyz$ Canonical, every term has all the variable names

Min Terms vs. Canonical expression

For example, $g(x, y, z) = \sum(0, 1, 6, 7)$

$g(x, y, z) = \sum((000)2, (001)2, (110)2, (111)2)$

$g = \bar{x}\bar{y}\bar{z} + \bar{x}\bar{y}z + xy\bar{z} + xyz$

	Inputs				Outputs			
	a1	a0	b1	b0	p3	p2	p1	p0
	0	0	0	0	0	0	0	0
	0	0	0	1	0	0	0	0
	0	0	1	0	0	0	0	0
	0	0	1	1	0	0	0	0
	0	1	0	0	0	0	0	0
	0	1	0	1	0	0	0	1
	0	1	1	0	0	0	1	0
	0	1	1	1	0	0	1	1
	1	0	0	0	0	0	0	0
	1	0	0	1	0	0	1	0
10	1	0	1	0	0	1	0	0
11	1	0	1	1	0	1	1	0
	1	1	0	0	0	0	0	0
	1	1	0	1	0	0	1	1
14	1	1	1	0	0	1	1	0
	1	1	1	1	1	0	0	1

$p2(a1, a0, b1, b0) = \sum(10, 11, 14)$

SOP and POS K-maps

yz:	00	01	11	10
x: 0				1
1			1	1

$g(x, y, z) = \sum(2, 6, 7)$

SOP

yz:	00	01	11	10
x: 0	0	0	0	
1	0	0		

$g(x, y, z) = \Pi(0, 1, 3, 4, 5)$

POS

Minimizing SOP Expressions

$\sum(2,3,6,7) = \bar{x}y\bar{z} + \bar{x}yz + xy\bar{z} + xyz$

$= \bar{x}y(\bar{z} + z) + xy(\bar{z} + z)$ Factor out smaller terms and simplify

$= \bar{x}y + xy$ Factor out y and simplify

$= y(\bar{x} + x)$ Simplify

$= y$

yz:	00	01	11	10
x: 0			1	1
1			1	1

Each pair of adjacent terms reduces to a simplified expression with one less variable.

K-Map Minimization Rules

1) **Min/max terms that differ in only one bit are adjacent (an Implicant). A K-map is assumed to wrap around on both sides.**
2) **A set of adjacent min/max terms may be combined to form a large group (a Prime Implicant). The number of terms in each group must be powers of 2**
   e.g., 2, 4, 8, or 16 terms.
3) **Each group of min/max terms must contain at least a single term that doesn't belong to any other group (no redundant groups), an Essential Prime Implicant**
4) **All terms must be grouped.**

More K-map Examples (no slides)

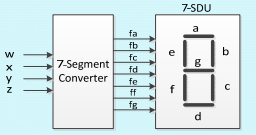
Don't-Care Signal values

- **Example: Displaying BCD numbers**

Inputs				Outputs						
w	x	y	z	fa	fb	fc	fd	fe	ff	fg
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	0	1	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	1	0	1
0	1	0	1	1	0	1	1	1	0	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	1	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	0	1	1	1
1	0	1	0							
1	0	1	1							
1	1	0	0							
1	1	0	1							
1	1	1	0							
1	1	1	1							

?

A 7-segment display unit and converter



Assuming that w, x, y, and z will not exceed 9, what should we enter in the table for inputs 10 through 15?

K-Map with Don't Cares

$f(w, x, y, z) = \Sigma(1, 9, 14) + \Sigma_d(3, 7, 11)$

yz: / wx:	00	01	11	10
00		1	d	
01			d	
11			①	
10		1	d	

$f(w, x, y, z) = \bar{x}z + wxy\bar{z}$

Logic Minimization Algorithm

- **Based on K-Map minimization technique**
  1. **Compare neighboring min/max terms two at a time (e.g., 0000 with 0001) to produce all Implicants**
  2. **Write the Implicant with a dash (e.g., 000-) for the bit that changes**
  3. **Repeat steps 1 and 2 for neighboring terms with matching dashes (e.g., 000- with 100- to get -00-)**
  4. **Prime implicants: Repeat step 3 until all prime implicants are identified**
  5. **Essential prime implicants: Choose a minimum set among the prime implicants**

Minimization Software

$f(w, x, y, z) = \Sigma\ (0, 2, 3, 4, 5, 6, 7, 8, 10, 12, 13)$

Input File

```
#Inputs: 4, Outputs: 1
.i 4
.o 1
#Input labels
.ilb w x y z
#output bit label
.ob f
#list of min-terms separated by space and a single output bit separated
by a tab
0 0 0 0        1
0 0 1 0        1
0 0 1 1        1
0 1 0 0        1
0 1 0 1        1
0 1 1 0        1
0 1 1 1        1
1 0 0 0        1
1 0 1 0        1
1 1 0 0        1
1 1 0 1        1
#end of list
.e
```

Output

```
#Inputs: 4, Outputs: 1
#Input signal labels
#output bit label
#list of min-terms and output
#end of list
.i 4
.o 1
.ilb w x y z
.ob f
.p 3
-10- 1
-0-0 1
0-1- 1
.e
```

$\bar{x}\bar{z} + \bar{w}y + x\bar{y}$

- Can be used with don't care inputs too

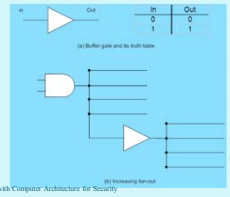
Circuit Timing Diagram

1. Circuits have gate and signal wire delays
2. Gates may have different output signal *rise* and *fall* times
3. Circuits have different signal paths from inputs to outputs

- These may result in signals reaching each gate at different times
- Can cause unwanted signal change (glitch) at some outputs
- Must wait for the longest signal propagation delay before the output(s) of a circuit can be used (e.g., stored in a register)

Circuit Fan-In and Fan-Out

- Fan-in: Number of gate inputs
- Fan-out: Number of places a gate output can connect to
- Buffer gate to increase a gate's fan-out



Other Gates

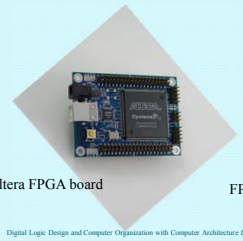
- Open collector (o.c.) buffer
  - Application: Wired-logic with a large fan-in
    - E.g., wired-AND or wired-OR logic
  - Many application areas
- Tri-state buffer
  - Used to create a bus for multiple modules to transmit data
  - Modules not outputting to the bus should be electrically isolated

Small combinational design examples

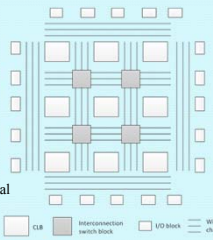
- Full-adder circuit
- Multiplexer circuit
  - Selects data one from 2 or more inputs
- Decoder circuit
  - Translates an input value to a corresponding signal
- Encoder circuit
  - Translates an active input signal to a corresponding signal number

Logic Implementation

- ASIC (application specific integrated chip)
- FPGA (Field Programmable Gate Arrays)



Altera FPGA board



FPGA internal

Design Flow

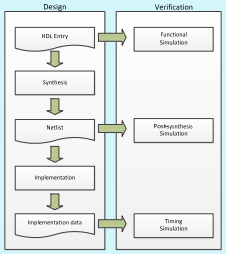
E.g., Verilog

Translate

Expressions

Map expressions to on-chip resources

Estimate signal delays



Verify circuit description

Verify netlist

Verify timing requirements

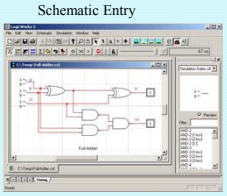
Behavioral Model

```
module full_adder
(
        input a, b, cin,
        output reg s, cout
);

always@(a or b or cin)
begin
        case ({a, b, cin})
        3'b000:     begin s = 0; cout = 0; end
        3'b001:     begin s = 1; cout = 0; end
        3'b010:     begin s = 1; cout = 0; end
        3'b011:     begin s = 0; cout = 1; end
        3'b100:     begin s = 1; cout = 0; end
        3'b101:     begin s = 0; cout = 1; end
        3'b110:     begin s = 0; cout = 1; end
        3'b111:     begin s = 1; cout = 1; end
        default:begin s = 0; cout = 0; end
        endcase
end
endmodule
```


Design Entry

- **Schematic entry**
- **Hardware Description Language**
  - **Structural Model**
    - Use gates
    - Use predefined modules
  - **Behavior model**
    - Use Boolean expressions, if-else, case (switch), for-loop, operators "+", "-", etc.
    - Not all behavioral models are synthesizable
    - Applications of non-synthesizable models
      - Generate test vectors for synthesizable models
      - Investigate computer architecture design ideas
  - **Hybrid**
    - Use both structural and behavioral models

Schematic Entry

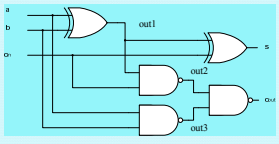


A summary of Verilog HDL operators

Precedence	Operator Type	Symbol	Example
Highest	Unary	+, -, !, ~	+a, -a (negate a), !a (logical not), ~a (bidwise not)
	Exponential	**	a ** 3 (a cubed)
	Arithmetic 1	*, /, %	a * b (multiply), a / b (divide), a % b (mod)
	Arithmetic 2	+, -	a + b (add), a - b (subtract)
	Shift:		
	Logical	<<, >>	a << 2 (shift left twice)
	Arithmetic	<<<, >>>	a >>> 3 (shift right 3 times extending the sign bit)
	Relational	<, <=, >, >=	a >= b (a greater or equal to b)
	Equality		
	Logical	==, !=	a == b if a is identical to b excluding x and z
	Case	===, !==	a === b is identical to b including x and z
	Bit-wise		
	Basics	&, |, ~, ^	a & b (and), a | b (or), ~a (not), a ^ b (xor)
	Combined	&~, |~, ~^, ^~	a &~ b (nand), a |~ b (nor), a ~^ b (xnor), a ^~ b (xnor)
	Logical	&&, ||, !	a &&b (and), a || b (or), !a (not)
Lowest	Conditional	?:	(a >= b) ? a - b : b - a

Structural Model Using Primitive Gates

```
module full_adder
(
        input a, b, cin,
        output s, cout
); //defines a module's name and its interface signals

wire out1, out2, out3;  //defines local signal names

xor     x1(out1, a, b);
xor     x2(s, out1, cin);
nand    n1(out2, out1, cin);
nand    n2(out3, a, b);
nand    n3(cout, out2, out3);

endmodule
```



Chapter 3

COMBINATIONAL CIRCUITS

LARGE DESIGNS

In this Chapter

- Design methodologies for large combinational circuits
  - Bit-parallel
  - Bit-serial
- Integer arithmetic as examples
  - Add, subtract, multiply, and divide as four basic arithmetic operations
- IEEE floating-point number standards
  - Floating-point Data Space
- Floating-point arithmetic
- Floating-point unit (FPU)

Top-Down Design Methodology

- Bit-parallel
  - Partition n-bit design problem into smaller n-bit design problems
  - E.g., 8-bit ALU designed using 8-bit adder/subtractor and 8-bit bit-wise logic
- Bit-serial
  - Partition n-bit design problem into a fewer-bit design problem (called slice)
  - E.g., 8-bit ALU designed using eight 1-bit ALU modules
- Hybrid
  - Design uses bit-parallel and bit-serial modules

Carry Propagate Adder (A bit-serial adder)

- Use FA slices
- Carry bits generated sequentially, one at a time
- Propagation delay proportional to number of carry bits
- Assuming SOP expressions for sum and carry bits and 0.1 ns delay for NANDs determine:
  - ΔCPA(8)
  - ΔCPA(32)
- CPA is the slowest

ΔCPA(8) = 1.7 ns

ΔCPA(32) = 6.5 ns, too slow

Carry Look-Ahead (CLA) Adder

- Goal: Generate carry bits in parallel
- Let's examine FA expressions
  - Easy to generate p and g bits in parallel
  - Carry bits are dependent, but can substitute carry expressions to break dependency
  - Once carry bits are known, easy to generate sum bits in parallel
  - ΔCLA(8) = ?

    0.8 *ns*

FA expression from Ch2:

$$s_i = a_i \oplus b_i \oplus c_{i-1}$$

$$c_i = (a_i \oplus b_i)c_{i-1} + a_i b_i$$

Let,

$$p_i = a_i \oplus b_i$$

$$g_i = a_i\, b_i$$

$$s_i = p_i \oplus c_{i-1}$$

$$c_i = g_i + p_i c_{i-1}$$

Observations

- If keep substituting previous carry expression in next carry expression will run into Fan-in and fan-out problems
- Solution: Generate some carry bits sequentially and some in parallel (next slide)

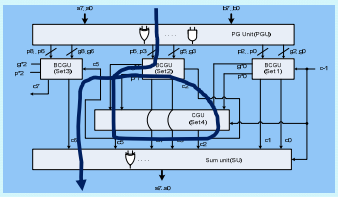
Large CLA Adder

1. Group carry bits into equal sized sets with sets resulting in no fan-in or fan-out problem
2. Generate carry bits in two steps

- What is the longest signal path from inputs to outputs?
- Determine ΔCLA(32)

ΔCLA(32) = 1.2 ns

Example: For simplicity assume n = 8



Subtractor

- Similar expressions as FA:

$$d_i = x_i \oplus y_i \oplus b_{i-1}$$

$$b_i = (\overline{x_i \oplus y_i})b_{i-1} + \bar{x}_i y_i$$

- Can use adder to do subtraction if both are needed
  - 2's complement adder/subtractor

Twos Complement Adder/Subtractor

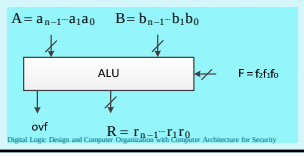
- $A - B$ can be viewed as adding A with $-B$
- Circuits for $A + B$ and $A + (-B)$ are similar except second input is negated when subtracting
- What do we known about converting negative numbers to 2's complement representation?
  1. Flip bits
     - Can be done with NOT gates
  2. Add 1
- Addition:
  - Do not flip B bits
  - Set carry-in to 0
  - Carry-out not part of the result
- Subtraction:
  - Flip B bits
  - Set carry-in to 1
  - Carry-out not part of the result
- How to combine into one circuit? Use a control bit m for mode.
  - Add when $m = 0$
  - Subtract when $m = 1$
  - Need an inverter circuit controlled by m
- Potential problem?
  - Result can overflow and become incorrect
  - Need overflow detection logic

Arithmetic Overflow

- When sum of two positive numbers is negative
  - I.e., Sign of result becomes 1
  - Applies to subtraction too
    - A - B when A > 0 and B < 0
- When sum of two negative numbers is positive
  - Sign of result becomes 0
  - Applies to subtraction too
    - A – B when A < 0 and B > 0
- A simple rule to detect overflow
  - Overflow when carry-in to sign bit position ≠ carry-out from sign bit position

Arithmetic Logic Unit (ALU)

- Performs arithmetic or bit-wise logic functions
  - A function code specifies which operation to perform
  - A complex combinational circuit
- Need to use bit-parallel or bit-serial design methodology
- Overflow flag (OVF) can only be active when performing arithmetic operations
  - Must be masked otherwise

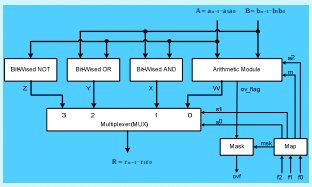


Example

f2	f1	f0	Function
0	0	0	Add
0	0	1	Sub
0	1	0	Increment
0	1	1	Decrement
1	0	0	Bitwise AND
1	0	1	Bitwise OR
1	1	0	Bitwise NOT
1	1	1	Not Defined

ALU Bit-Parallel Design

- Identify different types of operations
  - n-bit Arithmetic
    - Add, subtract, increment, decrement
    - Can combine into one 2's complement adder/subtrcator
  - n-bit Bit-wise operators
    - NOT, AND, OR
- Assume you have these modules draw a data path
  - Use MUX to select only one output
  - Include other necessary circuits
    - Circuit to convert input F into internal data path signals
    - Circuit to mask OVF during bit-wise operations
- Design modules and assemble



Example ALU Modules

- Design Arithmetic module
  - Use n-bit 2's complement adder/subtractor
  - Left input always A
  - Right input either B or 1
    - Need a circuit that outputs B if add/sub or 1 if inc/dec.
    - Use known modules when possible
- Design n-bit 4-to-1 MUX
  - Bit-parallel: Design using n-bit 2-to-1 MUXs (bit-parallel)
  - Bit-serial: Design using 1-bit 4-to-1 MUX slices
- Design Map and Mask circuits
  - Create truth tables
  - Find minimal SOP/POS expressions

ALU Bit-Serial Design

- Consider n copies of 1-bit ALU slices
  - Create truth table for 1-bit ALU slice
  - Use Espresso
- May use larger slices
  - 2-bit or 4-bit, for example
  - Larger slices may be designed bit-parallel
- Can be slow for large n

f2	f1	f0	a	b	ci	co	r	Function
0	0	0	0	0	0	0	0	Add
			0	0	1	0	1	
			0	1	0	0	1	
			0	1	1	1	0	
			1	0	0	0	1	
			1	0	1	1	0	
			1	1	0	1	0	
			1	1	1	1	1	
0	0	1	0	0	0	0	0	Sub
			0	0	1	1	1	
			0	1	0	1	1	
			0	1	1	1	0	
			1	0	0	0	1	
			1	0	1	0	0	
			1	1	0	0	0	
			1	1	1	1	1	
0	1	0	0	d	1	0	1	Increment
			1	d	1	1	0	
0	1	1	0	d	1	1	1	Decrement
			1	d	0	0	1	
1	0	0	1	1	d	d	1	Bit-wised AND
1	0	1	0	1	d	d	1	Bit-wised OR
			1	0	d	d	1	
			1	1	d	d	1	
1	1	0	0	d	d	d	1	Bit-wised NOT
			1	d	d	d	0	
1	1	1	d	d	d	d	d	Not Defined

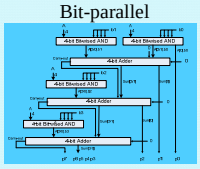
Truth Table for 1-bit ALU slice

Other design examples (unsigned multiplier)

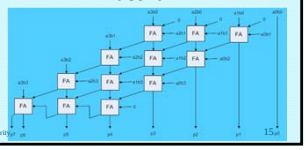
Algorithm

```
    1001   A
  * 1011   B
  ------
    1001
   0000
  1001
+ 1001
---------
  1100011   P = B * A
```

Bit-parallel



Bit-serial

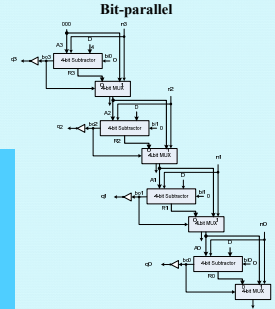


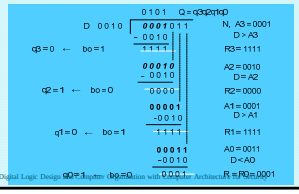
- Bit-parallel
  - Addends are added one at a time after adding 1st two
  - Less concurrency in the data path
  - Slower, longer propagation delay
- Bit-serial
  - Addend bits are added vertically, the way numbers are added by hand
  - More concurrency in the data path
  - Faster, shorter propagation delay

Unsigned Divider (restoring)

- Similar to how we divide by hand
  - In each step, remainder can be + or –
  - If remainder positive, use in the next step
  - Else, restore
  - Concatenate next numerator bit and repeat
- Bit-parallel
  - Requires subtractor and MUX modules
- Bit-serial
  - Can use 1-bit combined subtractor/MUX slices
  - See Exercise section

Bit-parallel





Real Number Arithmetic

IEEE 754 FP number Standards

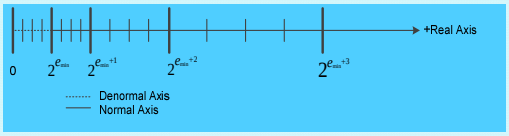
- Single, 32 bits
  - 1-bit sign, 8-bit biased exponent (bias = 127), 23-bit fraction
  - Stored as a 32-bit number in memory
- double, 64 bits
  - 1-bit sign, 11-bit biased exponent (bias = 1023), 52-bit fraction
  - Stored as a 64-bit number in memory
- Extended, 80 bits
  - 1-bit sign, 15-bit bias exponent (bias = 16383), 64-bit fraction
  - Stored in 80-bit registers only (no memory representation)

FP number Data Space (assume 32-bit FP numbers)

- Normal
  - $1 \le$ Biased exponent ≤ 254
- Denormal
  - Biased exponent = 0 and fraction $\neq 0$
- Zero
  - Biased exponent = 0 and fraction = 0
- Infinity
  - Biased exponent = 255 and fraction = 0
  - E.g., $\frac{1}{0}$
- Not-a-number (Nan)
  - Biased exponent = 255 and fraction $\neq 0$
  - E.g., $\sqrt{-1}$

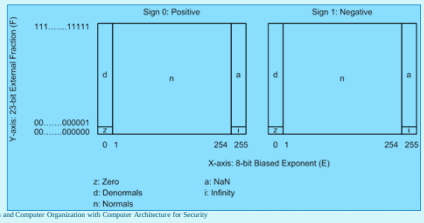
Data Space Illustration (1-Dimensional)

- Bold and thin lines indicate real numbers stored as FP numbers in computer
- More fraction bits implies more thin lines
- More exponent bits implies more bold lines



Two-Dimensional Illustration

- Easier to identify data space regions
- Easier to mark specific FP numbers or domain or range of a function
  - Eg. The largest FP number
  - E.g., for test generation purposes
    - region identified by (-1, 1) or [-1, 1], for example



FP Arithmetic

- Requires integer arithmetic
  - Operates on exponent and fraction numbers independently
  - Typically combinational arithmetic circuits
- Requires shift operations
  - Typically combinational shifter circuits
  - Used to line up implicit decimal points
    - E.g., during FP add
  - Used for normalizing results
    - Result converted to standard format
  - Used for rounding results
    - 64-bit fraction in register is converted to 23 or 52 bits format for storage
      - "float" data type: 23-bit fraction
      - "double" data type: 52-bit fraction
    - The resultant fraction is rounded
      - Based on the value of the bits lost
      - May require another normalization step

FP Add (e.g., S = A + B)

1. Switch operands (if necessary)
   - For S = A + B, |A| must be ≥ |B|
2. Align decimal points and compute result R.*F* = A.*F* + B.*F*
3. Normalize R.*F*
4. Round R.*F* to produce S.*F*

- Example

FP subtract, multiply, divide

- Subtraction
  - Lineup decimal points
  - Compute A - B if A.s = B.s or A + B if A.s ≠ B.s.
- Multiplication
  - Integer multiply fractions
  - Add exponents
  - XOR the sign bits
- Division
  - Integer divide fractions
  - Subtract exponents
  - XOR the sign bits
- The rounding and normalization steps are the same as in FP add

Chapter 4

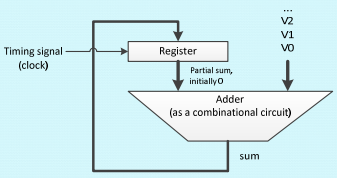
SEQUENTIAL CIRCUITS:
CORE MODULES

In this Chapter

- Basic modules for saving sequential circuit's state
  - Latches and flip flops
- Timing constraints
- Other flip-flops
- HDL models

Sequential Circuit Example

- Register retains state of circuit as *sum*
- Adder computes next *sum* (circuit's next state)
- Timing signal must guarantee *sum* is computed before register loading

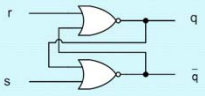


SR latch core circuit

- Three basic functions:
  - retain previous value (keeps *q* as is)
  - store 0 (reset *q*)
  - store 1 (set *q*)
- However, four possible cases, only three used
  1. *s* = 0 and *r* = 0, retain q
  2. *s* = 0 and *r* = 1, reset
  3. *s* = 1 and *r* = 1, set
  4. *s* = 1 and *r* = 1, not used
     - makes $q = \overline{q}$, invalid
     - causes *q* and $\overline{q}$ oscillate when both *s* and *r* instantly become 0 next

s	r	q^{t+1}
0	0	q^t
0	1	0
1	0	1
1	1	–

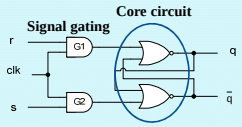
(a)



(b)

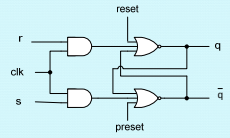
SR Latch (added clock signal)

- Goal: Prevent Case 4 (*s* = 1 and *r* = 1 affecting core circuit)
- Includes a signal gating logic to time sampling of inputs (*s* and *r*)
  - Only when both *s* and *r* are both stable (not changing) should affect core circuit
- Naming convention
  - positive-level, inputs are sampled when *clk* = 1
  - negative-level, inputs are sampled when *clk* = 0
- Assume positive-level (figure below)
  - *clk* = 0, core retains *q* value
  - *clk* = 1, *s* and *r* gated (sampled), enter the core
  - Timing constraint: *s* and *r* must stabilize <u>at</u> or <u>before</u> *clk* changes to 1 (sampling level)
    - *s* and *r* called synchronous inputs

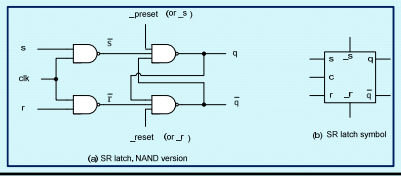


Asynchronous Inputs

- Function: Changes *q* and $\overline{q}$ independent of *clk*
- Used for circuit initialization during startup

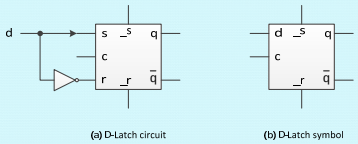


SR latch, NAND Version And Symbol



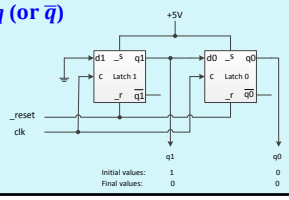
D Latch

- Requires one (synchronous) input *d* instead of two in SR latch
- Reduces number of signal routing wires, saves space
- Timing constraint: *d* must stabilize before *clk* changes to 1



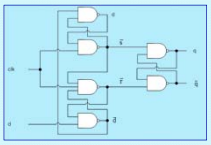
Disadvantage of Latches

- Signal chasing problem
  - Input *d1* can change values of both *q1* and *q0* during the same *clk* cycle
  - *d1* should only affect *q1*
- Limited application: Cannot be used to design circuits when *d* dependents on a *q* (or $\bar{q}$)
  - E.g., shift register

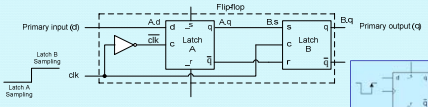


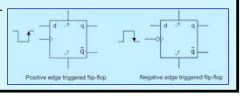
D Flip-Flop

- No signal chasing problem
- Can use two latches, one positive-level and one negative-level
  - The two latches act like "double-door entry"
- Called edge triggered, *d* sampled at *clk* edge
  - Positive edge when *clk* makes 0-1 transition
  - Negative edge when *clk* makes 1-0 transition
- Timing constraint: *d* must remain stable before and after the sampling *clk* edge



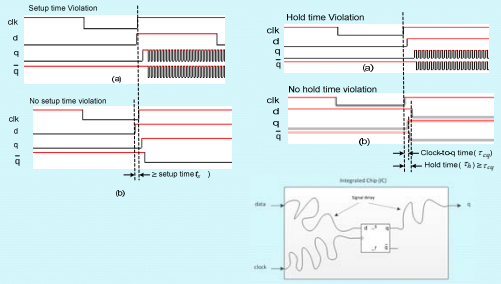
Three latches version, requires fewer total gates





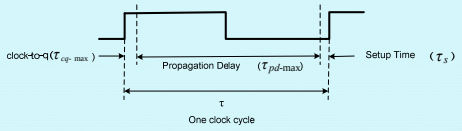
Setup and Hold times

- Set-up time: Minimum time *d* must be stable before *clk* sampling edge
- Hold time: Minimum time *d* must remain stable after *clk* sampling edge
  - May be negative due to difference in *d* and *clk* wire delays



Clock Frequency Estimation – Without Clock Skew

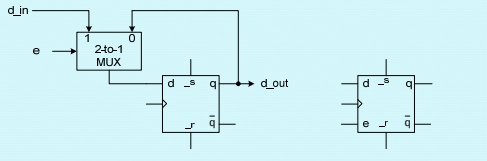
Minimum clock frequency, not including wire delays



$\tau \geq \tau_{cq-max} + \tau_{pd-max} + \tau_{st}$ $\quad f \leq \frac{1}{\tau}$ cycles/sec or Hertz

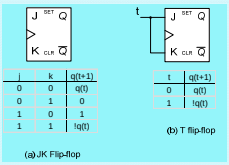
D Flip-Flop with Enable

- Retains *q* when *e* = 0
- Used when one or more flip-flops selected among many



Other flip-flops

- JK flip-flop
  - Uses two inputs called *j* and *k*
  - *j* = *k* = 1 will not cause oscillation
  - Better when wires require less space than gates
    - Not necessarily true today
- T (toggle) flip-flop
  - *q* alternates every clock cycle when *t* = 1
  - E.g., consider when *t* = 1, *q* controls a 2-to-1 MUX
- Both can be designed to operate edge triggered



j	k	q(t+1)
0	0	q(t)
0	1	0
1	0	1
1	1	!q(t)

(a) JK Flip-flop

t	q(t+1)
0	q(t)
1	!q(t)

(b) T flip-flop

D Latch HDL Model (Behavioral)

```
module d_latch
(
        input clock, _reset, _preset, d,
        output reg q,
        output  nq
);

assign nq = ~q;

always@(clock or !_reset or !_preset or d)
begin
        if(!_reset)
                q <= 0;
        else if(!_preset)
                q <= 1;
        else if(clock)
                q <= d;
end
endmodule
```

Designed like a combinational circuit.

Except, note the missing else here (creates a D-latch)

"<=" called non-blocking or concurrent assignment

D flip-flop HDL Model (Behavioral)

```
module dff
(
        input clock, _reset, _preset, d, e,
        output reg q,
        output nq
);

assign nq = ~q;

always@(posedge clock, _reset, _preset)
begin
        if(!_reset)
                q <= 0;
        else if(!_preset)
                q <= 1;
        else if(e)
                q <= d;
end

endmodule
```

Note the missing synchronous inputs *d* and *e* here

Note also the missing else here (creates a D flip-flop)

Chapter 5

SEQUENTIAL CIRCUITS:
Small Designs

In this Chapter

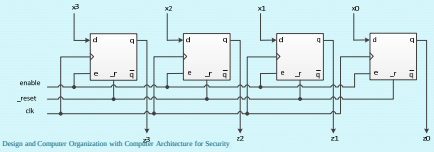
- Sequential circuit design models
- Design examples:
  - registers, counters, sequence recognizer
- Sequential circuit timing
- Interfacing sequential circuits

Sequential Circuit as a Finite State Machine (FSM)

- Requires flip-flop(s) to save circuit state
- Requires combinational circuit(s) to generate next circuit state and output(s)
- Two types of combinational circuits:
  - Functions of the circuits cannot be determined in advance
    - FSM design is modeled with a finite state diagram (FSD)
  - Functions of the circuits can be determined in advance
    - E.g., MUX, adder, ALU
    - No need for an FSD

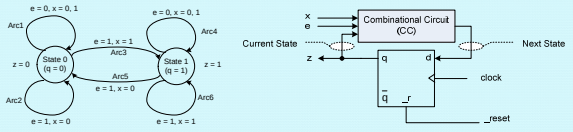
A simple design Example (Parallel-load register with enable)

- Assume unknown combinational circuit
  1. Design 1-bit register 1st
     - We have seen D flip-flop with enable in Ch4
     - Here, the flip-flop formally modeled as FSD (next slide)
  2. Combine register slices to create 4-bit register below
     - *enable*, *clk*, *_reset* connect to all



FSM design steps (1-bit parallel-load register example)

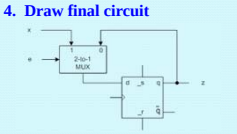
1. Draw FSD and detailed block diagram of 1-bit register slice



2. Convert FSD to truth table (also called transition table)

Current State	External Inputs		Next State
q	*e*	*x*	*d*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

4. Draw final circuit



3. Find minimal SOP or POS expression(s) for the output(s)
   $d = \bar{e}q + ex$ Ah, a MUX!

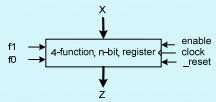
Circuit shown 1st in Chap. 4.

Some design rules

- If cannot determine function(s) of combinational circuit(s) in advance:
  1. Model FSM as FSD
     - May need to design bit-slice 1st
  2. Determine number of flip flops
  3. Convert the FSD to truth table
  4. Find minimal expressions for next state variable(s) and output(s)
  5. Draw the complete circuit with flip-flops
- Otherwise
  - Use bit-serial design with known modules
  - Or, bit-parallel design with known modules

Multi-Function Register

- Performs one of many functions
- Can be designed bit-serial or bit-parallel
- If cannot determine function(s) of combinational circuit(s) in advance
  - Model a bit-slice (e.g., 1-bit) as FSD
- Otherwise: Oh, I can use a MUX
  - Bit-serial: Use 1-bit 4-to-1 MUXs, for example
  - Bit-parallel: Use n-bit 4-to-1 MUX

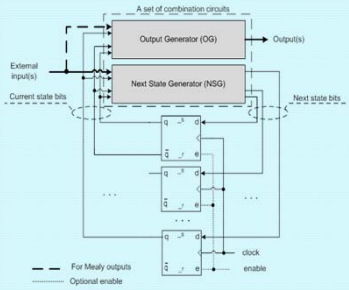


f1	f0	Action
0	0	Clear (synchronous reset)
0	1	Load X
1	0	Arithmetic shift (shifts right repeating the sign bit)
1	1	Right shift (shifts right entering 0 from left)

FSM (A formal view)

Three types of FSMs:

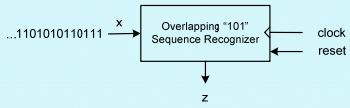
- Moore
  - External inputs do not synchronously affect outputs
  - Outputs called Moore
- Mealy
  - External inputs synchronously affect outputs
  - Outputs called Mealy
- Hybrid
  - Generates both Moore and Mealy outputs



Sequence Recognizer

Suppose FSM recognizes overlapping input sequence "101":

- Inputs processed one bit at a time (one per clock cycle)
- Not possible to determine the functions of the combinational circuits in advance
  - Model FSM as FSD
- Reset initializes the machine to known state
- Can be designed either as Moore or Mealy machine



Moore Sequence Recognizer ('101")

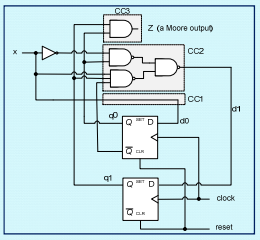
- FSD has 4 states (4 bobbles)
  - E.g., labeled A to D
    - E.g., A being the initial state
    - D being the acceptance state where output *z* becomes 1
- *z* is called Moore output
  - *I.e.*, synchronized to *clk*, can only change on *clk* edge

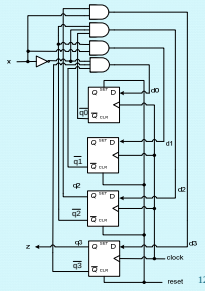
Ways to encode FSM states

- Binary encoded states
  - Assign unique binary numbers to states (each bobble in FSD)
  - Advantage: Minimum number of flip-flops
  - Disadvantage: More complex combinational circuits
- One hot design
  - Use one flip-flop per state
    - Only one flip-flop is set during each clock cycle
  - Disadvantage: Requires maximum number of flip-flops
  - Advantage: Less complex combinational circuits
  - Better with PLDs (e.g., FPGAs)

Example

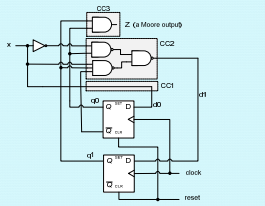
- Binary Encode State Values
  - Fewer flip-flops
  - Relatively more complex CCs
- One-hot State Values
  - More flip-flops
  - Simpler CCs



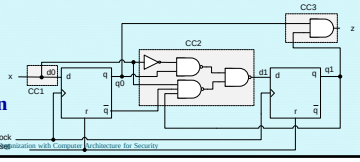


Circuit Layouts

Layout matches the formal view



Typical implementation layout:



How to deal with unknown states (if any)

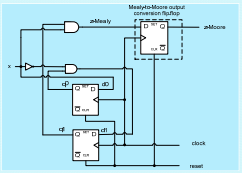
1. Unknown FSM states are ignored in the design
   - Write don't cares in the NSG and OG truth tables
     - Help further minimize combinational circuits
     - Machine must be reset if enters unknown states
2. Unknown states are transitioned to a known state
   - Machine recovers after one clock cycle
   - Some inputs may be lost
3. Fault tolerant FSM
   - Machine can detect error and correct itself
     - I.e., continues to operate normally

Mealy Sequence Recognizer ("101")

- Requires 3 states
- E.g., labeled A to C
  - Binary encoded states results in one unknown state D
- Output *z* assigned to arcs
  - *z* called Mealy output synchronized to both *clk* and input *x*
    - If *x* changes, *z* can change before *clk* edge

Converting Mealy outputs to Moore outputs

- Some designs are easier as Mealy
- Sometimes Moore outputs are preferred
- Use flip-flops to convert Mealy outputs to Moore outputs

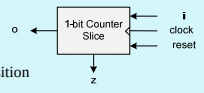


Counters

- Generate sequence of numbers, one per clock cycle
  - Binary counter, also called mod-k counter
    - E.g., Mod-4 counter counts 0, 1, 2, 3, and repeats
  - BCD counter
    - Counts 0, 1, 2, …, 9, and repeats
  - Gray code counter
    - 1-bit difference between consecutive numbers
    - Application example: Two FSM accessing a shared FIFO buffer
      - See Exercise section
  - Other sequences with fixed patterns
    - E.g., 0, 3, 6, 9, etc.
  - An arbitrary sequence
- Can count up or down
- Designed as bit-serial, bit-parallel, or hybrid

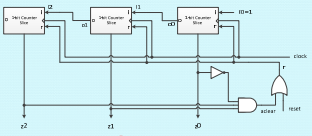
Mod-8 Up-Counter

- Counts 0, 1, 2, 3, 4, 5 ,6, 7, and repeats
- Design options
  - Bit-parallel
    - Requires a 3-bit adder
  - Bit-serial
    - E.g., for 1-bit slice, each slice is mod-2 counter
      - 1st slice counts every clock cycle
      - 2ns slice counts every other clock cycle,
      - 3rd slice counts every 4th clock cycle
    - Designed as hybrid FSM
      - *i* decides slice counts up or retains
      - *z*-Moore is counter bit
      - *o*-Mealy is input *i* for the next slice
        - *o* asserted each time z makes 1-0 transition

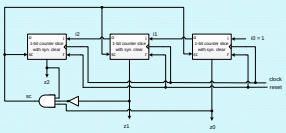


Mod-6 Up-Counter

- Counts 0, 1, 2, 3, 4, 5, and repeats
- Partially mod-8 counter
- Wrong way to design:
  - Reset counter asynchronous when counter reaches 6
- Right way to design:
  - Reset counter synchronously when count becomes 5
    - Modelled in FSD



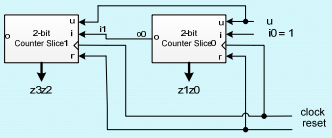
The wrong way!
(Using asynchronous reset)



The right way!
(using synchronous reset)

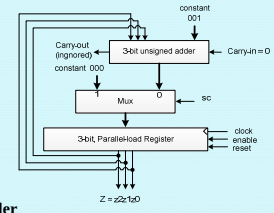
8-bit up/down counter (bit-serial)

- Assume 2-bit slices
- *i* decides next higher slice shall count up or down depending on *u*
- Design a 2-bit counter FSD



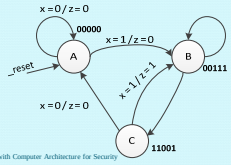
Mod-8 Up Counter (bit-parallel)

- Assume implements synchronous reset (clear)
- Required modules can be determined in advanced:
  - 3-bit adder
    - One input set to 1
    - Initial carry = 0
  - 3-bit 2-to-1 MUX
    - One input set to 0
  - 3-bit parallel-load register
- Less hardware in final circuit
  - Adder and MUX each with one fixed input will simplify to simpler expressions



Fault Tolerant FSM

- Flip-flop outputs may change due to environmental hazards (e.g., static electricity)
- Use Hamming codes as binary encoded states
  - Codes are separated with Hamming distance of 3 or more
- A 1-bit change in a state number can be detected and corrected
  - 1-bit change has 1 Hamming distance from valid state
  - 1-bit change has 2 or more Hamming distances away from other states
- Design options:
  1. Design as you would an FSM but enter single bit faults in Truth Table
  2. Design as you would non-fault tolerant FSM but then add Hamming code generation, error detection, and correction circuits



Hamming Coding Scheme and Error Detection

- Example: Consider 7-bit Hamming codes with 4-bit data 1001
  - Use set of XOR gates to generate three parity bits

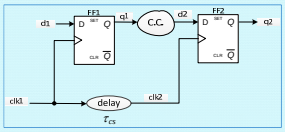
$p_1 = d_3 \oplus d_5 \oplus d_7$
$p_2 = d_3 \oplus d_6 \oplus d_7$
$p_4 = d_5 \oplus d_6 \oplus d_7$

7-bit Code: $d_7 d_6 d_5 p_4 d_3 p_2 p_1$ = 1001100, $p4p2p1$ = 100

- Faults can happen when 7-bit code is stored in memory or FFs or transmitted
- Fault detection and correction
  - E.g., 7-bit code loaded in FFs as $d'_7 d'_6 d'_5 p'_4 d'_3 p'_2 p'_1 = (1\underline{1}101100)_2$
    - Note, bit 6 has changed due to 1-bit fault
  - Calculate parity bits (p") from d' bits
  - Compare p" bits (100) with p' bits (010)
    - Use bit-wise XOR
    - E = e2e1e0 = 100 $\oplus$ 010 = 110 , indicates bit 6 is in error
  - Can use a 3-to-8 decoder and XOR gates to correct the error bit

Sequential Circuit Timing (1)

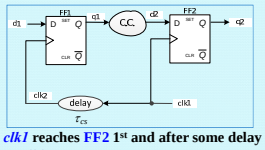
- Clock skew due to delay (τ_{CS}) in clock signal
- Clock skew can create circuit timing problems
  1. If Δcc < τ_{CS}, FF2 will load $d2^{New}$ and not $d2^{Current}$
     - FSM results in functional error
  2. If Δcc ~= τ_{CS}, *d2* may be still changing when *clk2* arrives
     - FSM malfunction due to setup or hold time violation at FF2
- Δcc > τ_{CS}, FF2 will load $d2^{Current}$ as it should
  - Circuit will function correctly



clk1 reaches FF1 1st and after some delay as *clk2* to FF2

Sequential Circuit Timing (2)

- Possible problem: Next *clk1* reaches FF2 before previous *clk2* reaches FF1
  - If Δcc ~= $\tau - \tau_{cs}$, *d2* may be still changing when *clk1* arrives
    - FSM malfunction due to setup or hold time violation at FF2
- Normal operation
  - When Δcc < $\tau - \tau_{cs}$



clk1 reaches FF2 1st and after some delay as *clk2* to FF1

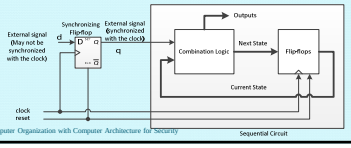
Clock Frequency Estimation – With Clock Skew

Add τ_{cs} to the clock period determined in Ch. 4

$$\tau \geq \tau_{cq-max} + \tau_{pd-max} + \tau_{st} + \tau_{cs}$$

Interfacing Sequential Circuits

- How to handle asynchronous inputs
  - Asynchronous inputs may change at any time with respect to *clock* signal
    - They can cause setup or hold time violation if directly enter FSM
- Solution: Use synchronization flip-flop(s)
  - Asynchronous input *d* may violate setup or hold time of synchronizing FF, not FFs of FSM
    - *q* of synchronizing FF will eventually stabilize if setup or hold time violated
  - May use two synchronizing FFs if *q* slow to stabilize



FSM HDL Model (All Behavioral)

- E.g., consider modeling sequence "101" recognizer
- Single module with three "always" blocks:

1. Combinational circuit NSG
   - Use "case" statement to list states
     - E.g., Moore: four states A, B, C, and D (the *q* bits)
   - Use "if-else" statements to model FSD arcs
     - Uses current state A, B, C, or D and input *x* to model next state (the *d* bits)
2. Combinational circuit OG
   - Use "case" and/or "if-else" to model outputs
     - Moore: if current state is D then *z* = 1 else *z* = 0
     - Mealy: if current state is D and *x* = 1 then *z* = 1 else *z* = 0
3. Flip-flops
   - Model FFs with either asynchronous or synchronous reset and enable (if necessary)

Chapter 6

SEQUENTIAL CIRCUITS: *LARGE DESIGNS*

In this chapter

- Recall from Ch1:
  - Complex sequential circuit = data path + control unit
- Data path types
- Control unit types
- Performance parameters
- Design examples
  - Unsigned sequential multiplier
  - 2's complement sequential multiplier
  - A simple GPU

Complex Data Paths

- Include combinational circuit modules
  - ALU, MUXs, decoders, wired-logic, etc.
  - Sometimes custom combinational circuits
- Include small sequential modules
  - Registers and counters
  - Sometimes custom sequential circuits
- Can include buses
  - With buffers and tristate buffers
- Include wires for interconnecting the modules creating multiple paths for data
  - Each path identifies a unique data path operation
    - Computation, memory access, etc.

Control Units

1. Monitor events as input signals
   - External event signals
     - E.g., an "start" signal that starts a task performed on the data path
   - Data path event signals
     - Arithmetic overflow flag signal
     - A counter reaching a target value
     - Etc.
2. Generate control as output signals
   - Signals to control data path modules
     - E.g., ALU function signals, MUXs' selection signals, register enable signals, etc.
   - Signals to other interfacing modules (if any)
     - E.g., A "done" signal when done completing a task

Register Transfer Notation (RTN)

- Formally describes a data path operation
- May use an arbitrary or an HDL syntax
- Examples:
  - CNTR ← CNTR + 1 //incrementing counter
  - CNTR <= CNTR + 1; //Verilog HDL
  - R ← R[7]//R[7:1] //Arithmetic right shift
  - R <= R >>> 1; //arithmetic right shift (Verilog)
  - R <= {R[7], R[7:1]}; //arithmetic right shift (Verilog)
  - M[x] ← R; //memory transfer (write)
  - R ← M[x]; //memory transfer (read)
  - Etc.

Data Path Types

1. Single-cycle
   - Performs an operation specified by one or more RTNs during a single clock cycle
   - Uses more hardware but simple controller
   - Can be the lowest (e.g., execute a program)
2. Multi-cycle
   - Performs an operation specified by one or more RTNs using multiple clock cycles
   - Uses less hardware but more complex controller
     - Operations share hardware modules
   - Faster than single-cycle
3. Pipelined
   - Operates like an assembly line
   - Efficient when performing stream of operations
     - Stream of instructions
     - Stream of FLOPs
     - Etc.
   - Multiple clock cycles per operation
   - Concurrent processing
   - More hardware like single-cycle
   - High performance

Design Example (Comparing different data paths)

- Problem: Consider two complex RTNs:
  - R ← A + B + C + D and R ← A + B + C - D
- Each called fused for involving three or more operands
  - 3-operand fused instructions common in some modern processors
    - E.g., Intel's multiply-add instruction
      - R ← A + B * C
      - Usefull for matrix operations (e.g., matrix multiplication)
    - One instruction instead of two
      - FMA vs. FMUL and FADD
- Advantage of fused RTN:
  - Reduces arithmetic rounding errors (refer to FP arithmetic)
  - Reduces number of instructions
- Complex fused RTNs application
  - Consider reconfigurable CPU

1. Single-Cycle Data Path

1. One clock cycle to perform R ← A + B + C + D or R ← A + B + C - D
2. Hardware required
   - two adders
   - One adder/subtractor
   - One register
3. Controller required
   - A *mode* signal deciding A + B + C + D or A + B + C − D

- Estimated data path clock period = ?

$$\tau \geq 2\Delta_{ADD} + \Delta_{ADD/SUB} + \tau_{st} + \tau_{cq} + \tau_{cs}$$

2. Multi-Cycle Data Path

- One or more clock cycles per RTN
  - Divide complex RTNs into simple ones
  - Multiple clock cycles to perform R ← A + B + C + D or R ← A + B + C - D
- More complex controller
  - Operations are done in steps
- Less hardware
  - One adder/subtractor, two MUXs, and a register
  - Data path modules used multiple times
    - E.g., adder/subtrcator module used 3 times
- Estimated data path clock period = ?

Example

```
Cycle 1: R ← A
Cycle 2: R ← R + B
Cycle 3: R ← R + C
Cycle 4: if mode == 0 then
           R ← R + D
         else
           R ← R – D
```

$$\tau \geq \Delta_{MUX1} + \Delta_{ADD/SUB} + \Delta_{MUX2} + \tau_{st} + \tau_{cq} + \tau_{cs}$$ Smaller than $\tau_{single-cycle}$

3. Pipelined Data Path

- Three clock cycles per RTN
  - Cycle 1: $X = A + B$
  - Cycle 2: $Y = X + C$
  - Cycle 3: $R = Y \pm D$
- Hardware similar to single-cycle except organized into stages
- Efficient when performing $A_i + B_i + C_i \pm D_i$ for $i = 0$ to $N-1$ for large N
- Requires less total time to complete a job
- Estimated clock period = ?

$$\tau \geq \Delta_{ADD/SUB} + \tau_{st} + \tau_{cq} + \tau_{cs}$$ Smaller than $\tau_{single-cycle}$ and $\tau_{multi-cycle}$

Charting Pipeline Behavior

Final Stage →

Stage 3			R0 ← Y0 ± D0	R1 ← Y1 ± D1	…
Stage 2		Y0 ← X0 + C0	Y1 ← X1 + C1	Y2 ← X2 + C2	…
State 1	X0 ← A0 + B0	X1 ← A1 + B1	X2 ← A2 + B2	X3 ← A3 + B3	…
Cycle	1	2	3	4	…

Horizontal Organization

Final Stage ↓

Cycle	Stage 1	Stage 2	Stage 3
1	X0 ← A0 + B0		
2	X1 ← A1 + B1	Y0 ← X0 + C0	
3	X2 ← A2 + B2	Y1 ← X1 + C1	R0 ← Y0 ± D0
4	X3 ← A3 + B3	Y2 ← X2 + C2	R1 ← Y1 ± D1
5	X4 ← A4 + B4	Y3 ← X3 + C3	R2 ← Y2 ± D2
…	…	…	…

Vertical Organization

Performance Parameters

- Processing time (T)
  - Time required to perform a job (e.g., *N* tasks)
- Speedup
  - Calculated as a ratio
    - $\frac{T_{slow}}{T_{fast}}$, T of slower system divided by T of faster system
  - E.g., System A is 1.5 times faster than system B
- Efficiency
  - Calculated as a ratio
    - $\frac{S}{S_{ideal}}$, speedup divided by ideal speedup
  - E.g., 100% efficiency when all hardware modules utilized all the time
- Throughput
  - Calculated as ratio
    - $\frac{N}{T}$, Number of tasks performed divided by T
  - E.g., number of tasks performed per second

Speedup (Pipelining vs. Single-cycle)

- *N* tasks: $A_i + B_i + C_i \pm D_i$ for $i = 0$ to N-1

$$Speedup = \frac{T_{single-cycle}}{T_{pipeline}}$$

Assume $\tau_{\text{single-cycle}} \simeq K\,\tau_{\text{pipeline}}$

- For $N = 3$ and $k = 3$, speedup = ? 1.8
- For N = 1000, k = 3, speedup = ? 2.99
- For very large N? (e.g., as $N \to \infty$) Speedup approaches k, the number of stages

Efficiency of Pipelining

- *N* tasks: $A_i + B_i + C_i \pm D_i$ for $i = 0$ to N-1

$$Efficiency = \frac{Speedup}{K}$$

- For $N = 3$ and $k = 3$, Efficiency = ? 60%
- For $N = 1000$ and $k = 3$, Efficiency = ? 99.8%
- For very large N? (e.g., as $N \to \infty$) 100%

Throughput of Pipelining

- *N* tasks: $A_i + B_i + C_i \pm D_i$ for $i = 0$ to N-1

$$Throughput = \frac{N}{T_{pipeline}}$$

- For N = 3 and k = 3, the throughput is about? $0.6\,\tau^{-1}$
- For N = 1000 and k = 3, the throughput is about? $0.998\,\tau^{-1}$
- Throughput as $N \to \infty$? τ^{-1} or f
- Standards
  - MIPS
    - E.g., 100 MIPS CPU
  - FLOPS
    - E.g., 1T FLOPS system
  - Benchmarks, more typical
    - SPEC CPU2006 for measuring performance of computer systems
    - SPECviewperf for measuring performance of computer-graphic systems

Types of Control Units

1. Modeled as FSD for multi-cycle data paths
2. Micro-programmed Control (programmable)
   - Easy to modify after implementation
   - Also used if FSD would be very large
3. Pipeline control for pipelined data paths

1. FSD-based Control Unit

- Design steps:

1. Draw FSD for the controller
   - Assume CU triggered by external signal *start*
   - 4 clock cycles to perform $A + B + C \pm D$
   - Specify data path operations with RTNs
2. Complete design (assuming all structural)
   1. Determine specific data path control signals
   2. Draw detailed block diagram the controller
   3. Construct truth tables
   4. Find minimal expressions
   5. Combine with data path to complete design

```
Cycle 1: R ← A
Cycle 2: R ← R + B
Cycle 3: R ← R + C
Cycle 4: if mode == 0 then
             R ← R + D
         else
             R ← R – D
```


2. Microprogrammed Control

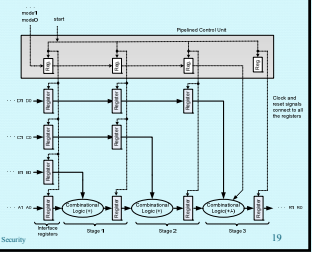
- Described as a program
  - Conditional statements use signal values
  - RTNs specifies data path operations
  - Branching changes program flow
- Tree pieces of hardware:
  - A 2-function counter, called microprogram counter (MPC)
  - A memory called control memory (CM) stores micro-instructions representation in binary
  - A MUX controls the functions of MPC

Example

```
0:if start == 0 then go to 0;
1: R ← A;
2: R ← R + B;
3: R ← R + C;
4: if mode == 0 then  R ← R + D,  go to 0;
5: R ← R - D,  go to 0;
```

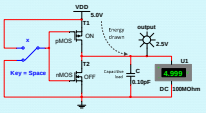
3. Pipeline Control

- As oppose to other two controllers, signals are generated at same time but applied to data path at different times
- Example, signal *mode* enters pipeline but not used until stage 3



Circuit Energy Consumption

- Energy is a function of capacitance and voltage
- Consider a CMOS NOT gate
- 0-1 transition at output draws energy from power source
  - ½ stored in capacitance
  - ½ dissipated as heat
- 1-0 transition doesn't draw energy from power source but ½ charge in capacitance dissipates as heat



$$E^{0-1}_{Dynamic} = \int_{v=0}^{v=V_{DD}} C\, v\, dv = \frac{1}{2} C\, {V_{DD}}^2 \quad \text{Joules}$$

$$E^{1-0}_{Dynamic} = \frac{1}{2} C\, {V_{DD}}^2 \quad \text{Joules}$$

Circuit Power Consumption

- Power is energy consumed over time
  - 1 Watts = Amounts of Joules consumed in one second
- During each clock period some signals make 1-0 and some 0-1 transitions
- Energy consumed for all 0-1 transitions during one clock period:

$$E_{Dynamic} = \frac{1}{2} C_{Total}\, {V_{DD}}^2 \text{ Joules}$$

- Power consumed during one clock period

$$P_{Dynamic} = \frac{E_{Dynamic}}{\tau} = \frac{1}{2} C_{Total}\, {V_{DD}}^2 f \quad \text{watts}$$

- Signals not changing during clock period consume only static power

$$P_{Static} = V_{DD}\, I_{DD}$$

I_{DD}, DC or leakage current

Some Ways to Reduce Dynamic Power Consumption

- Reduce total capacitance, C_{Total}
- Reduce supply voltage, V_{DD}
- Reduce clock frequency, f.
- Reduce glitches

$$P_{Dynamic} = \frac{1}{2} C_{Total}\, {V_{DD}}^2 f$$

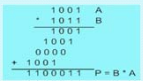
Power vs. Energy

- Energy is independent of clock frequency
  - Increasing clock frequency increases power, not energy consumption
- Consider two CPUs A and B
  - Suppose A consumes 20% more power but executes a program quicker; requires %40 less time than B
- Which CPU is more energy efficient?

CPU A uses 72% of energy used by CPU B

Unsigned Sequential Multiplier

- Hardware:
- Three registers A, B, and P
  - A and B are n-bits, P is n+1 bits
- A Mod n+1 counter (CNTR)
- An n-bit adder
- Basic algorithm:
  1. Initialization
     - Load values and clear register P and CNTR
  2. P ← P + A if next multiplier bit (b0) is 1
     - Skipping over 0 multiplier bits
  3. Right shift P //B, increment CNTR
  4. If not done go to 2
  5. Else, final product is in P//B



Multiplier Algorithm

A ← A_value; B ← B_value; P ← 0; CNTR ← 0;
Do
    if(B[0] == 1)
        P ← P + A;
    {P, B} ← {P, B} >> 1;
    CNTR ← CNTR + 1;
While CNTR < n

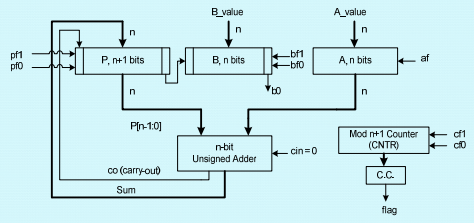
Design Entry Options
(Data Path + FSD-based Controller)

Option I: All structural
- Design all modules at gate level
- Recommended for schematic-only design tools

Option II: Hybrid
- Behavioral models for modules
- Combine modules to create structural model

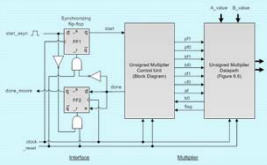
Option III: All behavioral
- Behavioral model of NSG
- Behavioral model of OG
  - Describes data path
  - OG not combinational circuit in this case
- Behavioral model of FFs

Multiplier Data Path (Structural)



Multiplier Controller (FSD)

- Assumes external input signal *start_asyn*
  - Uses one synchronizing FF (Ch5)
- Assumes external Moore output *done_moore*
  - indicates product is ready when asserted
- Draw controller FSD
  - E.g., Mealy



Multiplier Model (All Behavioral)

- One module with five "always" blocks:
  1. Model of synchronization FF
  2. Model of *done*-Moore FF
  3. Combination circuit model of controller NSG
  4. Model of OG as data path (sequential circuit)
  5. Model of controller FFs

Synchronizing FF

- Resets when *_reset* or *done* asserted
- Otherwise, loads asynchronous input *start_asyn*.

```
always@(posedge clock or negedge _reset or posedge done)
begin
        if(_reset == 0 || done == 1)
                start <= 1'b0;
        else
                start <= start_asyn;
end
```

Converting done-Mealy to done-Moore

- Asynchronously de-asserts *done*-Moore when *_reset* or *start_asyn* asserted
- Synchronously asserts *done*-Moore when *done*-Mealy asserted

```
always@(posedge clock or negedge _reset or posedge start_asyn)
begin
    if(_reset == 0  || start_asyn == 1)
                done_moore = 1'b0;
    else
                done_moore <= done;  //done = 1 kept for one clock cycle
end
endmodule
```

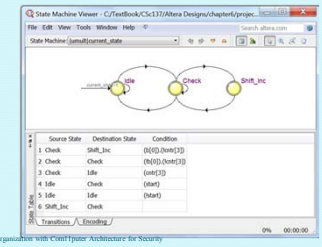

OG defines data path

```
module umult(
    ....

assign result = {p[7:0], b};
//--------------- The states ----------------------------
------

parameter       Start =             2'b00,
                Check =             2'b01,
                Shift_Inc =         2'b10;

//----------------------- Output Generator (OG)
always@(posedge clock or _reset)
begin
    if(!_reset)
    begin

                p <= 0;
                cntr <= 0;
    end
    else
```

```
case(current_state)
    Start: if(start == 1)
                begin // initialize
                        a <= a_value;
                        b <= b_value;
                        p <= 0;
                        cntr <= 0;
            end
    Check:  begin
                if (cntr < 8)
                    if(b[0] == 1)
                        p <= p[7:0] + a;
                else
                begin
                    {p,  b}  <=  {p,  b}  >>  1;
                    cntr <= cntr + 1;
                end
            end
    Shift_Inc:  begin
                {p, b} <= {p, b} >> 1;
                cntr <= cntr + 1;
                end
    endcase
end
```

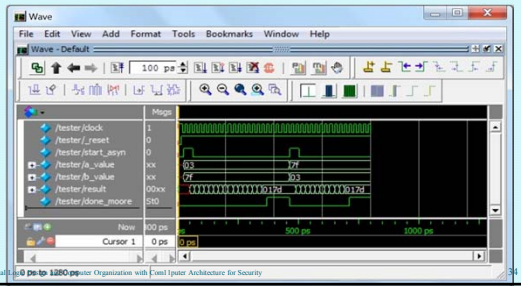

FSD Verification

- Was the FSD description correct?
  - Tools can generates FSD from the HDL description for verification purposes



Simulation Results

- 0x03 * 0x7f = 0x017d, longer multiplication time
- 0x7f * 0x03 = 0x017d, shorter multiplication time
- Optionally, can implement an initial switching circuit to switch A with B if A < B.



2's Complement Multiplier

- Multiplier B and multiplicand A are 2's complement numbers resulting in 2's complement product
- Basic data path operation is P ← P + A or P ← P – A
- Examines B bits two bits at a time overlapping
- Hardware:
  - Similar to unsigned multiplier except for adder/subtractor
- Algorithm:

A ← A_value[n-1]//A_value; B ← B_value//0; P ← 0; CNTR ← 0;
Do
  if(B[1] ⊕ B[0] == 1) //overlapping two bits
    P ← P ± A; //where m = B[1]
  {P, B} ← {P, B} >> 1;
  CNTR ← CNTR + 1;
While CNTR < n

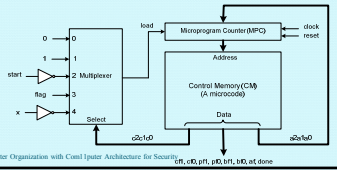
- E.g., if B_value = $(1111)_{2s}$ = -1, then P = A * -1 = -A for arbitrary A_value

2's Complement Multiplier Microprogram Controller

- Multiplier microprogram

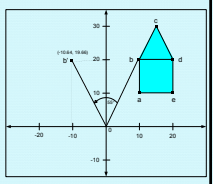
0	If *start* == 0 go to 0;
1	P ← 0, A ← *A_value*, B ← *B_value*, CNTR ← 0;
2	If CNTR == n go to 6;
3	If B[1] ⊕ *B*[0]== 0 go to 5
4	P ← P ± A;
5	{P, B} >>> 1, CNTR ← CNTR + 1, go to 2;
6	*done* = 1, go to 0;

- Controller



2D Virtual Object Rotation

- Rotating object by β degrees requires rotating each vector by β degrees
  - $X' = \cos\beta * X - \sin\beta * Y$
  - $Y' = \sin\beta * X + \cos\beta * Y$
- Implemention requires cosine and sin functions
  - More hardware
- CORDIC algorithm simpler
  - Can use integer arithmetic
    - Addition, subtraction, and left shift for multiplication
  - Rotation done in steps
    - Factor out cos β
      - $X' = \cos\beta * (X - \tan\beta * Y)$
      - $Y' = \cos\beta * (\tan\beta * X + Y)$
    - E.g., 55° rotation is done in 4 stps:
      - 45°, 7°, 2°, 1°
    - Note, $\tan 45° = 1$, $\tan 7° = 1/8$, $\tan 2° = 1/32$, $\tan 1° = 1/64$
  - Done in steps results in a gain
    - E.g., for 55°, gain is $\frac{1}{\cos 47° * \cos 7° * \cos 2° * \cos 1°}$
    - Rotated object looks bigger



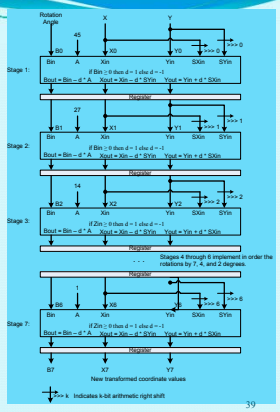
CORDIC Rotation Algorithm

- Fix number of steps for $|\beta| \le 90°$
  - Rotation can be done in 7 steps
    - E.g., for 55°, steps are 45°, 27°, -14°, -7°, 4°, 2°, 1°
  - tan $\beta * X$ and tan $\beta * Y$ are computed by shifting X and Y by 0 bits in step 1, 2 bits in step 2, …, and 6 bits in step 7
- An initial rotation of 90° or 180° if $|\beta| > 90°$
- Replace β with β Mod 360° if $|\beta| > 360°$,
- Algorithm for $|\beta| \le 90°$:

```
If (β >= 0)
        d = 1;
else
        d = -1;
for (i = 0; i < 7, i++)
        x = x – d * (y >>> i);      //arithmetic right shift (integer division)
        y = d * (x >>> i) + y
        β = β – d * tan^-1 2^-i     //tan^-1 2^-i read from table or
                                    //hard coded as 45, 27, 14, 7, 4, 2, 1
endfor
```

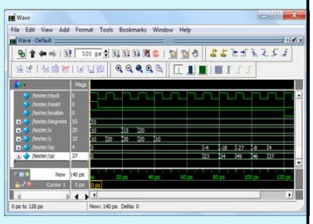

Pipelined CORDIC Rotation

- Loop unrolling: 7 stages for 7 iterations of the for-loop
- Hard coded rotation angles
- Wired shifts implement integer division
- Simpler controller
  - All stages perform the same computations but operate on different data



Simulation Result

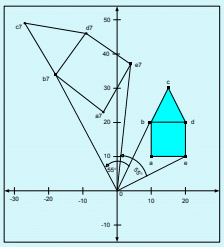
- Illustrates pipeline vector transformation
- At 1 GHz clock, one billion peak transformations per second
- Typically million vectors per virtual object



Rotated Virtual Object

- Shows a gain of 1.653 = 1/0.6048
  - CPU must multiply each new coordinate point by 0.6048 to get the actual size

    $0.6048 = \cos 47° * \cos 27° * \cos 14° * \cos 7° * \cos 2° * \cos 1°$
- New coordinate points include small error due to integer division used in the CORDIC algorithm



Chapter 7
Memory

In this chapter

- Memory technologies
- Memory design
  - Memory cell
  - Memory chip internal organization
- Memory communication protocols
- Data storage schemes
- UMA vs. NUMA system architectures

Logical organization

- Number of addresses by word size
  - E.g., 1K × 8
  - E.g., 512 × 16
- Total size in bytes
  - 1KB
  - 1MB
  - 1GB
  - Etc.

Address Decimal	Address Binary (10 bits)	Data 8-bit Content
0	0000000000	00010001
1	0000000001	10000111
2	0000000010	00111100
3	0000000011	11000000
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
1023	1111111111	10000001

(a) 1K × 8 Memory

Address Decimal	Address Binary (9 bits)	Data 16-bit Content
0	000000000	0001000100010001
1	000000001	1000011111100001
2	000000010	0011110000001100
3	000000011	1100000011010100
.	.	.
.	.	.
.	.	.
511	111111111	1000000111001011

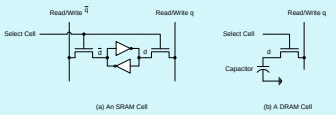
(b) 512 × 16 Memory

Memory Technologies

- Non-Volatile
  - Each memory cell retains 0 or 1 indefinitely
  - Word accessible
    - ROM
    - PROM
    - EEPROM
      - Can be written limited number (~ 100,000) of times
    - Older technologies EPROM
    - Applications: boot loader, LUT, firmware
  - Block accessible (as secondary memory storage)
    - Magnetic disk
    - Flash memory (EEPROM based)
- Volatile
  - Each memory cell retains 0 or 1 as long as powered
  - Word accessible only
    - SRAM
    - DRAM
      - SDRAM (DDR, DDR2, DRR3, etc.) as modern DRAM

RAM cells

- SRAM
  - Hardware
    - 6 transistors
  - Retains data while powered
  - fast
- DRAM
  - Hardware
    - One transistor
    - One small capacitor
  - Much smaller than SRAM cell
  - Cheaper per bit
  - Slow



(a) An SRAM Cell (b) A DRAM Cell

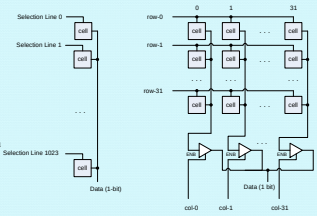
s	Action
0	a and b are disconnected (electrically isolated)
1	a and b are connected

(c) NMOS Transistor

Organization and Access

- 2D Organization (cell array)
  - Rectangular as the die
  - Requires fewer total number of wires
- Read/Write Operation
  1. Frist select a row
     - Also called row activation
  2. Then select one or more cells from activated row to either read or write
- Burst access
  - Access multiple cells in specific order typically from a single row
  - Cells form a block of data (e.g., 32B)
- Page access
  - Access many cells from one or more rows
  - Cells form a large block of data (e.g., 4KB)

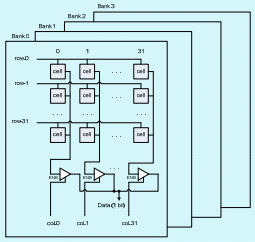
32 X 32 X 1



(a) One-dimensional organization (b) Two-dimensional organization (shown for read)

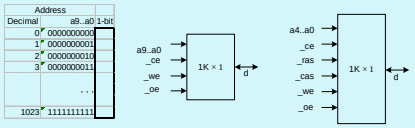
Multi-bank

- Allows seamless access
  - Cells read/written may belong to different banks
- Can overlap operations
  - Activating a row in one bank while read/writing cells from already activated row in another bank



Memory Interface

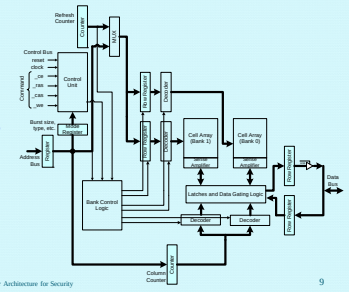
- Requires address lines (address bus)
  - Address for DRAM is provided in two cycles
- Requires control lines (control bus)
  - Indicating enabling, reading, and writing
- Requires data lines (data bus)
  - Bi-directional data bus
  - Separate input and output data lines



(a) Logical view (b) An SRAM block diagram (c) A DRAM block diagram

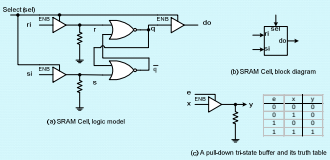
SDRAM (Synchronous DRAM)

- Interface signals form memory command
- Synchronous operation makes design of computers easier, cheaper
- Today SDRAM technologies are used for main memory



SRAM Cell Model

- Real RAM cells cannot be simulated with logic simulation tools
- It can be modeled with SR latch and tri-state buffers to mimic similar behavior
- Resister converts Hi-Z output to 0



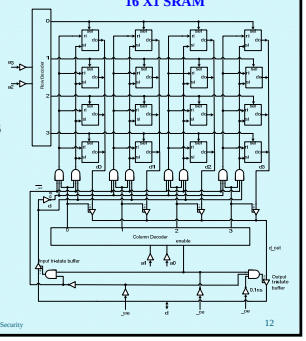
Memory Design

- Memory chip
  - Internal organization
    - Single or multi-banked
    - Bi-directional data bus
  - Access protocol defines signal timing
- Memory module
  - Wider data bus than memory chip
- Memory unit
  - Wider address bus than memory module

Memory Chip

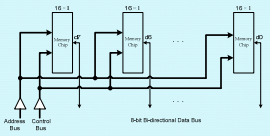
16 X1 SRAM

- Requires two decoders
  - Row decoder activates a row
  - Column decoder selects one or more cells
- Input and output tri-stated buffers to implement bi-directional data bus



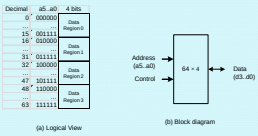
Memory Module

- Also called memory card
- 32- or 64-bit data bus
  - Wider if ECC
- For building memory unit(s) as main memory



Memory Unit

- Maps logical memory space to physical memory space
- Different mapping options
  - High-order interleaving
  - Low-order interleaving (later)
  - Hybrid
    - E.g., NUMA architectures

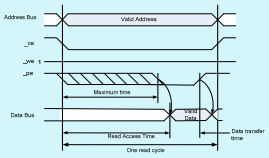


High-order Interleaving Example

Memory Access

- Follows specific communication protocol and signal timing
- Memory Cycle
  1. Starts when address decoding begins
  2. Waits to activate a row and select cell(s)
  3. Completes read or write operation
  4. Ends cycle
- Timing parameters
  - Access time
    - Read: From start until data appears on data bus
    - Write: From start until data is written to memory cells
  - Transfer time
    - Time to transfer data to/from memory
- Memory latency
  - Access time + transfer time

SRAM



SDRAM

- Concurrent memory operations
- Read Protocol:
  1. Issue burst size
  2. Issue row address
  3. Wait for row to activate (fixed number of clock cycles)
  4. Issue column address
  5. Repeat step 4 as needed
     - Timing depends on burst size
  6. Data placed on data bus, one per clock cycle, seamlessly

DDR SDRAM

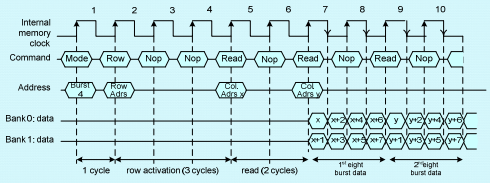
- Operation similar to SDRAM
- Data placed on data bus on rising as well as falling clock edges
  - Two data items per clock cycle
  - Doubling the bandwidth of SDRAM
    - Doubling number of data bytes per second

Data Interleaving

- High-Order Interleaving
  - Data for consecutive memory addresses are stored in the same memory module/unit
  - Advantage:
    - Divides memory space into two or more disjoint sub-spaces
    - Each sub-space may be accessed by a separate processor
- Low-Order (fine) Interleaving
  - Data for consecutive memory address are stored in different memory modules/units
  - Advantage:
    - Increases memory bandwidth

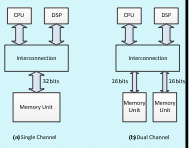
DDR2 SDRAM

- Read/write from two banks at the same time
  - Fine interleaving memory banks
- Doubling bandwidth of DDR SDRAM
  - Requires higher data transfer rate



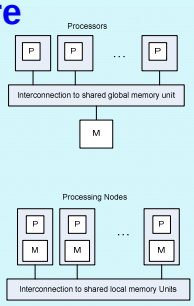
Multi-Channel

- Organize data bus into two or more independent channels
  - Separate burst access in each channel
- Larger bursts to deliver same amount of data
  - More efficient channels
  - More continuous delivery of data
  - Better performance
    - E.g., for real-time processing
  - Application
    - Better performing embedded systems



Multi-Processor Memory Architecture

- Uniform memory access (UMA)
  - Memory latency about the same (uniform)
  - Good for small systems
    - E.g., multi-core processor system
- Non-uniform memory access (NUMA)
  - Memory latencies vary (non-uniform)
    - Small when accessing local memory
    - Long when accessing remote memory
  - Average latency < UMA
  - Better for multithreaded programs
    - Each threads mostly accesses its local memory
    - Only shared data (if any) accessed remotely
      - E.g., consider producer-consumer application
  - Nodes can be multi-core



Chapter 8

INSTRUCTION SET ARCHITECTURE

Chapter 8: Instruction Set architecture

- Introduction
- Type of Instructions
- High level language program to execution
- Instruction cycle
- Instruction Set architecture (ISA)
- Addressing Modes and Machine code
- Types of ISA
- Design Examples ACC ISA
- Sparc and Pentium Assembly program architecture
- Performance Parameters

Chapter 8: Introduction

- The preceding chapters covered digital design concepts and , Datapath and Memory organization
- Modern CPUs implement pipelining and instruction-level parallelism (ILP) to increase performance
- Data path fetches an instruction from memory and decodes the instruction

Chapter 8: Introduction (cont'd)

- An instruction set architecture (ISA) refers to Data path that executes a program.
  - Single cycle
  - Multicycle
  - Pipeline
- Data-dependent instructions go through the pipeline stages
  - Can lead to stall in the pipeline
  - May reduce pipeline efficiency
- Branch instructions change execution flow
  - jne (Jump if not equal to)
  - Jge (Jump if greater than or equal to)
  - Jmp (Will jump to specified address)

Type of Instructions

- A processor is designed for general-purpose programming
  - Graphics processing Units (GPU)
  - Digital signal processing (DSP)
- Special purpose instructions
  - SIMD
  - Computer security related instructions
- Data-manipulation instructions
- Data-movement instructions
- Program-flow control instructions

High level language program to execution

As shown in Fig. 8.1, a software program is typically written in a high-level language, such as C/C++ or Java, and is translated by a compiler into assembly

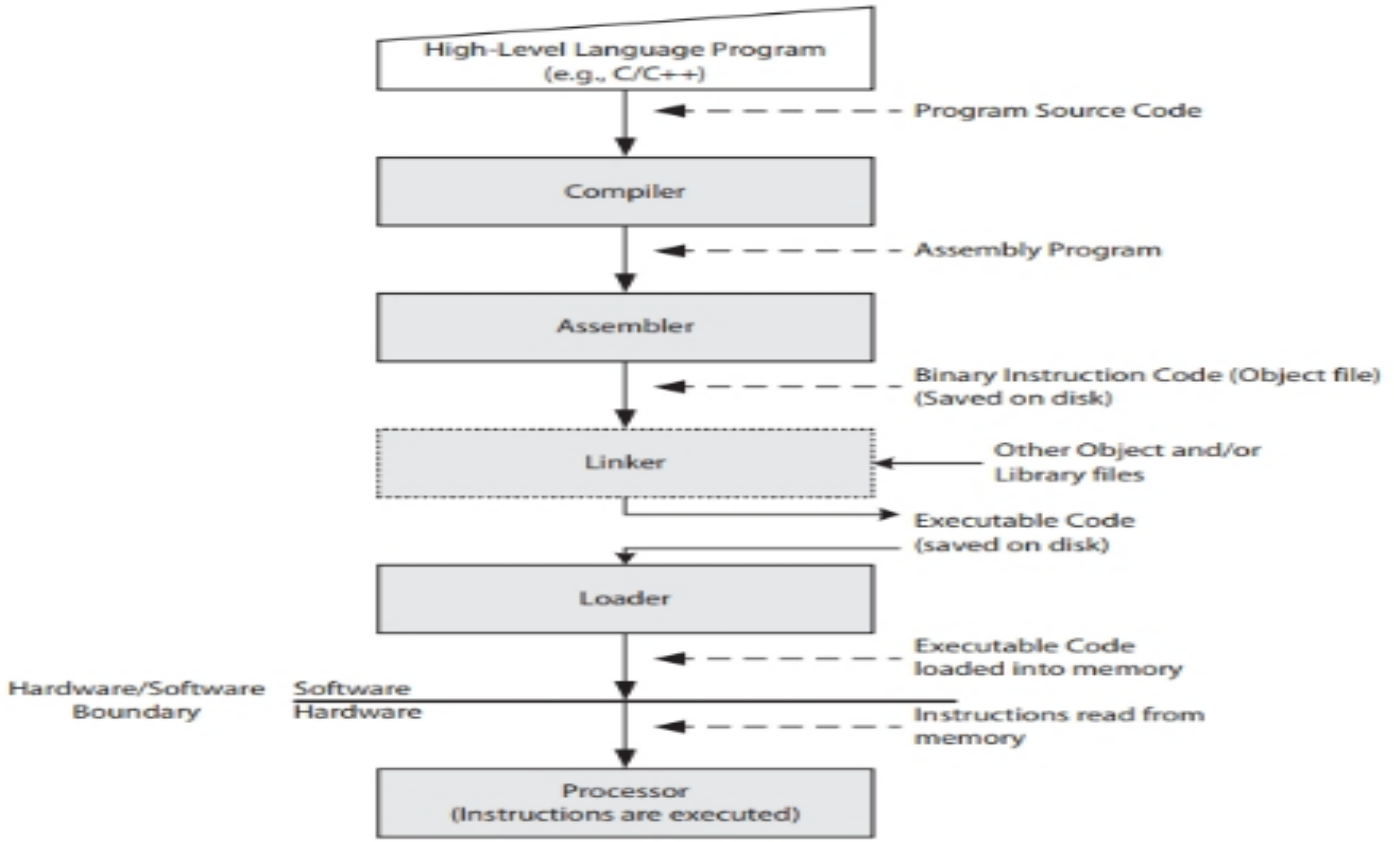


FIGURE 8.1 Basic program translation and execution process.

Instruction Cycle

- Data path has four main tasks
  - Fetch
  - Decode
  - Execute
    - May access data memory (another cache)
  - Writes Results

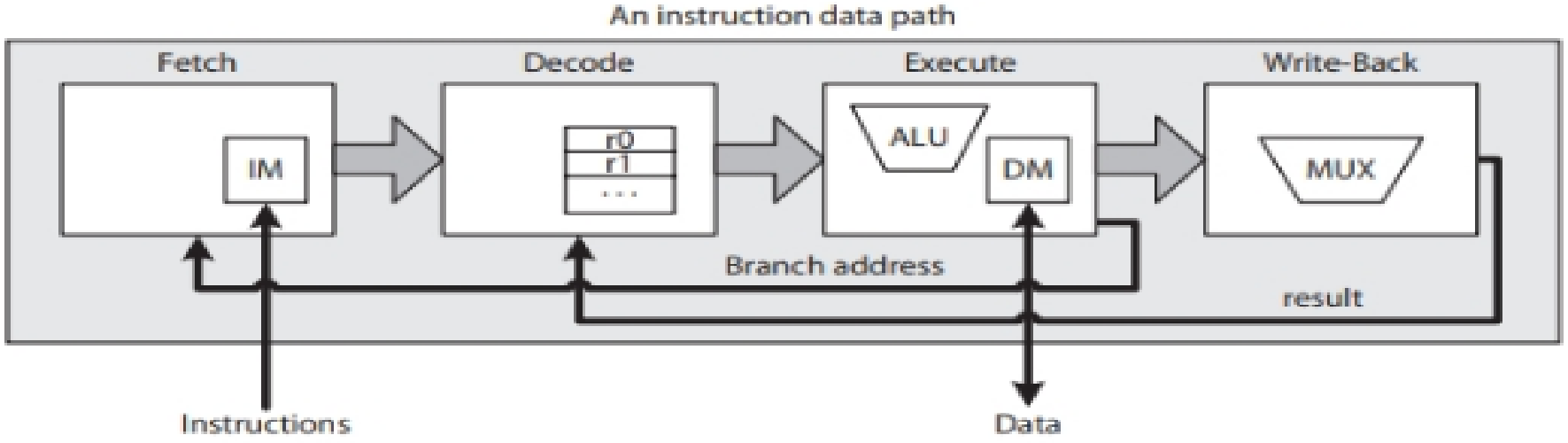


FIGURE 8.2 An instruction data path with instruction memory (IM) and data memory (DM).

Instruction Set architecture (ISA)

- **Types of Instruction set architecture:**
  - **Stack ISA**
  - **Accumulator ISA**
  - **CISC ISA (complex instruction set computer)**
  - **RISC ISA (reduced instruction set computer)**
- **An opcode is part of instruction set which tells the hardware what operation needs to be performed**
  - **Assembly language mnemonic form, an opcode is a command such as ADD, MOV, SUB or JMP**
  - **For example: MOV AL, 34h**
- **An instruction includes a set of operands that are specified explicitly, implicitly**

Addressing Modes and Syntax Examples

- Immediate
  - E.g., Add R1, 9;
- Direct
  - E.g., ADD R1, (M[9]);
- Register
  - E.g., ADD R1, R2;
- Register direct
  - E.g., ADD R1, (R2);
- Register indexed
  - E.g., ADD R1, R2, (M[9]);

Operand Notation	Addressing Mode
V	I, immediate: V is an immediate input operand, a 2's complement number.
(V)	D, direct: V is a memory address and (V) indicates the content of memory address V (i.e., M[V]).
R	R, register: Indicates an input data register source or a destination register or both
R, (V)	X, indexed: V is a memory address and R + V is the address of the next data item in memory (i.e., M[R + V]).

TABLE 8.1 Examples of Addressing Modes

Machine Instruction Format (Examples)

	Example	Instruction
Stack:	1	ADD
Accumulator:	2	ADD 9
CISC:	3	ADD R1, –9
CISC:	4	ADD R1, (9)
CISC:	5	ADD R1, R2, (9)
CISC:	6	ADD R1, R2
RISC:	7	ADD R3, R1, R2

Table 8.2

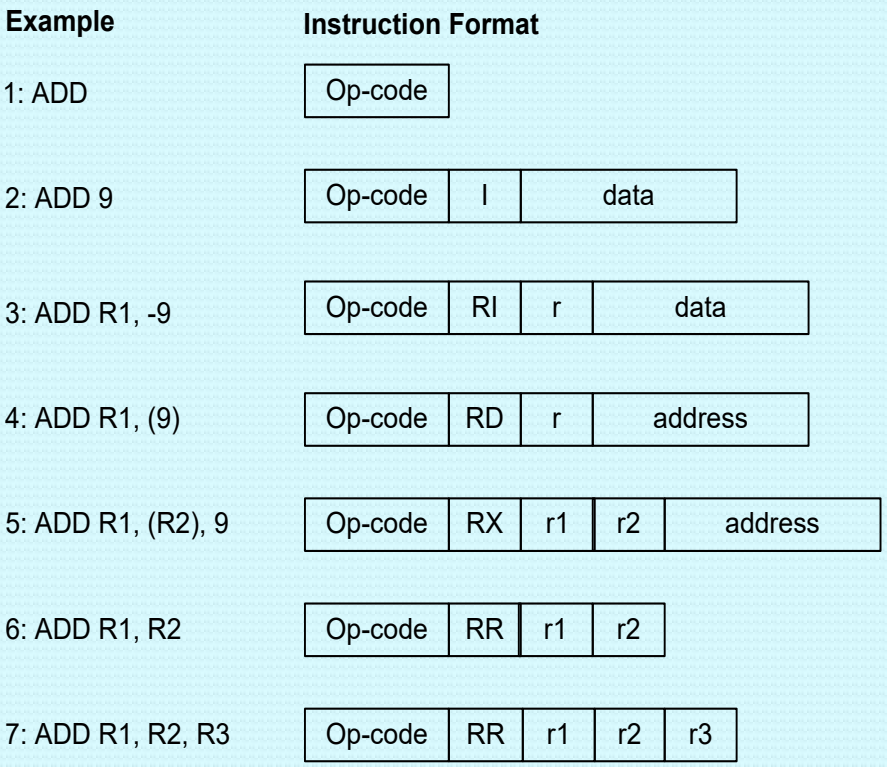


Fig 8.3

Types of ISA – Stack (1)

- Arithmetic instructions have no explicitly declared operands
- Operands are on stack inside CPU
- E.g., ADD
- Example: A = B * (C + D);
  - Requires converting statement into reverse polish notation
    - CD+B*=A
  - Reverse polish notation converted to assembly program?
- Short instructions (advantage)
- Stack as LIFO buffer (disadvantage)

Stack ISA - Example of assembly program

- **Example Program :**

```
Instruction
number
   1:      PUSH (C) //stack←M[C]
   2:      PUSH (D) //stack←M[D]
   3:      ADD      //stack←(C) + (D), values popped, added,
                    //result pushed
   4:      PUSH (B) //stack←M[B]
   5:      MUL //stack←((C) + (D)) * (B), values popped, added,
               //result pushed
   6:      POP (A)  //M[A]←(((C) + (D)) * (B)), value is popped
                    //and stored in memory
```

Types of ISA – Accumulator (2)

- One of the operands is a known register, called accumulator (Acc)
  - LD (C)
- Second operand is immediate or data from memory
- Acc always destination register
  - E.g., ADD 9 //Acc ← Acc + 9
  - E.g., ADD (C) //Acc ← Acc + M[C]
- Example Program: A = B * (C + D); ?
- Simple data path, less hardware (advantage)
- Acc bottleneck (disadvantage)
  - E.g., A = (C + D) * (E – F);

Acc ISA - Example of assembly program

- **Example Program:**

```
1: LD    (C)
2: ADD   (D)
3: MUL   (B)
4: ST    (A)
```

Types of ISA – CISC (3)

- CISC (complex instruction set computer)
- Many simple and complex instructions
- Multiple addressing modes
- Many working registers (e.g., 16)
  - Recent results kept inside CPU in registers
- Arithmetic instructions can access memory
  - E.g., ADD R1, R2 //R1 ← R1 + R2
  - E.g., ADD R1, (9) //R1 ← R1 + M[9]
- Example: A = B * (C + D);
  - Program?
- Complex instruction set (advantage)
  - Fewer instructions per program
- Complex instruction set (disadvantage)
  - Complex data path
  - Limited pipelining of instruction cycle
  - Many instructions and addressing modes seldom used

CISC-ISA: Example of assembly program

- **Example Program:**

1. **LD R1, (C)**
2. **ADD R1, (D)**
3. **MUL R1, (B)**
4. **ST (A), R1**

Types of ISA – RISC-(4)

- RISC (reduced instruction set computer)
- Arithmetic instructions cannot access memory
- Many more working registers (e.g., 32)
- Implements only most commonly used instructions
- 3-operand instructions
  - E.g., ADD R3, R1, R2 //R3 ← R1 + R2
  - E.g., ADD R2, R1, 9 //R2 ← R1 + 9
- Only LD and ST instructions access memory
- Example Program: A = B * (C + D);
- Simpler and highly pipelined data path (advantage)
- Requires compiler optimization to increase efficiency
- The architecture of all modern processing cores

RISC-ISA: Example of assembly program

- **Example Program:**

```
1. LD     R1,  (C)
2. LD     R2,  (D)
3. ADD    R3,  R1, R2
4. LD     R4,  (B)
5. MUL    R5,  R3, R4
6. ST     (A), R5
```

Design Example: Acc-ISA

- We start with example high-level language program
- Design instruction set for example program
- Generate assembly program
- Generate binary machine instructions
- Create Acc-ISA data path
- Model in HDL
- Simulation results

Example Instruction format for Acc-ISA

```
.code
        LD   0
        ST   (sum)
        ST   (i)
L1:
        CMP     7
        JGT     L2
        MVX
        LD      X(array)
        ADD     (sum)
        ST      (sum)
        LD      (i)
        ADD     1
        ST      (i)
        JMP     L1
L2:     …
.data
array:      RB  16
i:          RB  2
sum:        RB  2
```

Example code #1

```
int array[8];
int i, sum;
sum = 0;
for (i = 0; i < 8; i++)
    sum = sum + array[i];
```

Example 8.2. The listing of an Acc-ISA assembly language program for the program in Example code 1.

Example Program

Example code #1:

```
int array[8];
int i, sum;
sum = 0;
for (i = 0; i < 8; i++)
    sum = sum + array[i];
```

Acc-ISA instruction set?

- **Create a list of Acc-ISA instructions to translate the program to assembly language program**

Op-Code	Instruction	Addressing Mode	Example	Action
0	NOP		NOP	Do nothing
1	ADD	Immediate	ADD data	ACC ← ACC + data
2		Direct	ADD (address)	ACC ← ACC + M[address]
3	CMP	Immediate	CMP data	if ACC == data then GTF = 1 else GTF = 0
4	JGT	Immediate	JGT address	PP ← address if GTF = 1
5	JMP	Immediate	JMP address	PP ← address
6	LD	Immediate	LD data	ACC ← data
7		Direct	LD (address)	ACC ← M[address]
8		Indexed	LD X(address)	ACC ← M[X + address]
9	MVX	Register	MVX	X ← ACC
10	ST	Direct	ST (address)	M[address] ← ACC

ACC: Accumulator; GTF: Greater than flag; PP: Program pointer; X: Index register

TABLE 8.3 Example Acc-ISA Instruction Set That Translates a High-Level Program into an Equivalent Assembly Language Program

Acc-ISA Example Assembly Program

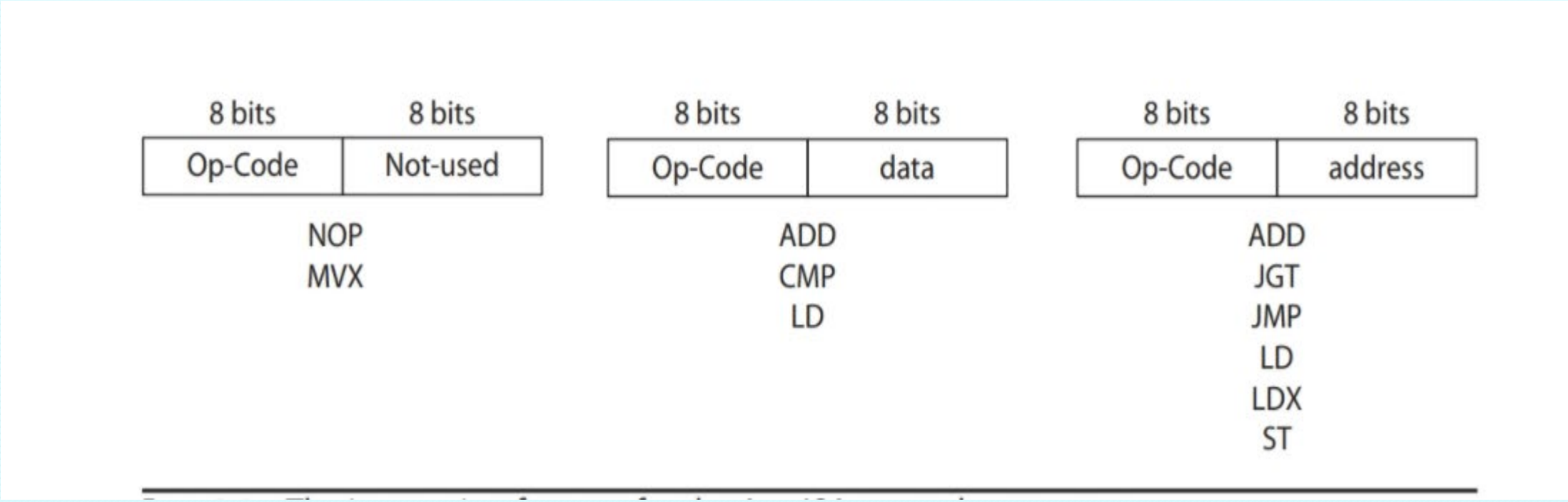


Figure 8.6 The instruction formats for the Acc-ISA example processor.

Acc-ISA Machine Instructions

Address	Instruction		Instruction in binary	Inhex
0:	LD	0	0000,0110;0000,0000	0600
2:	ST	0xEC	0000,1010;1110,1100	0AEC
4:	ST	0xEE	0000,1010;1110,1110	0AEE
6:	CMP	7	0000,0011;0000,0111	0307
8:	JGT	0x1A	0000,0100;0001,1010	041A
A:	MVX		0000,1001;0000,0000	0900
C:	LD	X(0xF0)	0000,1000;1111,0000	08F0
E:	ADD	(0xEC)	0000,0010;1110,1100	02EC
10:	ST	(0xEC)	0000,1010;1110,1100	0AEC
12:	LD	(0xEE)	0000,0111;1110,1110	07EE
14:	ADD	1	0000,0001;0000,0001	0101
16:	ST	(0xEE)	0000,1010;1110,1110	0AEE
18:	JMP	6	0000,0101;0000,0110	0506
1A:	…			

Example 8.5. The manually assembled output for the assembly program in Example 8.2.

Acc-ISA Pipelined Data path

- **Data path consists of four stages (block diagram)**

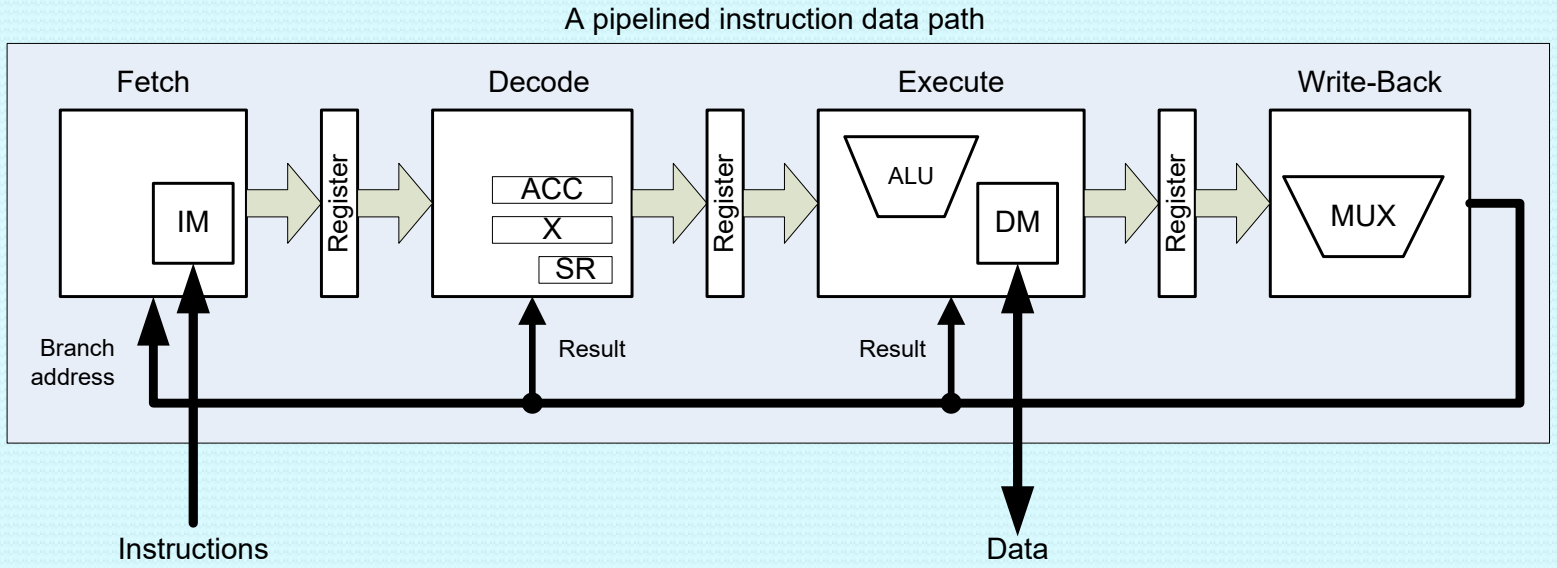


Fig 8.10

Acc-ISA example assembly program

- **Code below converted to:**
  - **Assembly program**

Example 8.1. A program code listing

```
int array[8];
int i, sum;
sum = 0;
for (i = 0; i < 8; i++)
    sum = sum + array[i]
```

Example 8.2. The listing of an Acc-ISA assembly language program for the program in Example 8.1:

```
.code   //start program code
        LD    0         //Initialize, ACC←0
        ST    (sum)     //M[sum]←ACC
        ST    (i)       //M[i]←ACC
L1:
        CMP   7         //is i > 7? (is ACC == 7?)
        JGT   L2        //exit for-loop if yes (PP←L2)
        MVX             //get next index (X←ACC)
        LD    X(array)  //get next array element (ACC←M[array
                        //+ X])
        ADD   (sum)     //and add it to the partial sum (ACC←ACC
                        //+M[sum])
        ST    (sum)     //store the partial sum in memory (M[sum]
                        //← ACC)
        LD    (i)       //do i = i + 1: get i (ACC←M[i]),
        ADD   1         //increment i (ACC←ACC + 1), and
        ST    (i)       //save i (M[i]←ACC).
        JMP   L1        //loop back
```

Single-cycle data path Acc-ISA processor

- **Fetch, decode, execute, and write-back units**
  - Decode unit contains a combinational circuit that inputs an op-code and generates all the control signals
  - Execute unit contains all the components necessary to execute an instruction
    - MUX is needed to choose either the operand when the operand is an immediate data or M[operand] when the operand is an address and indicates
    . memory content

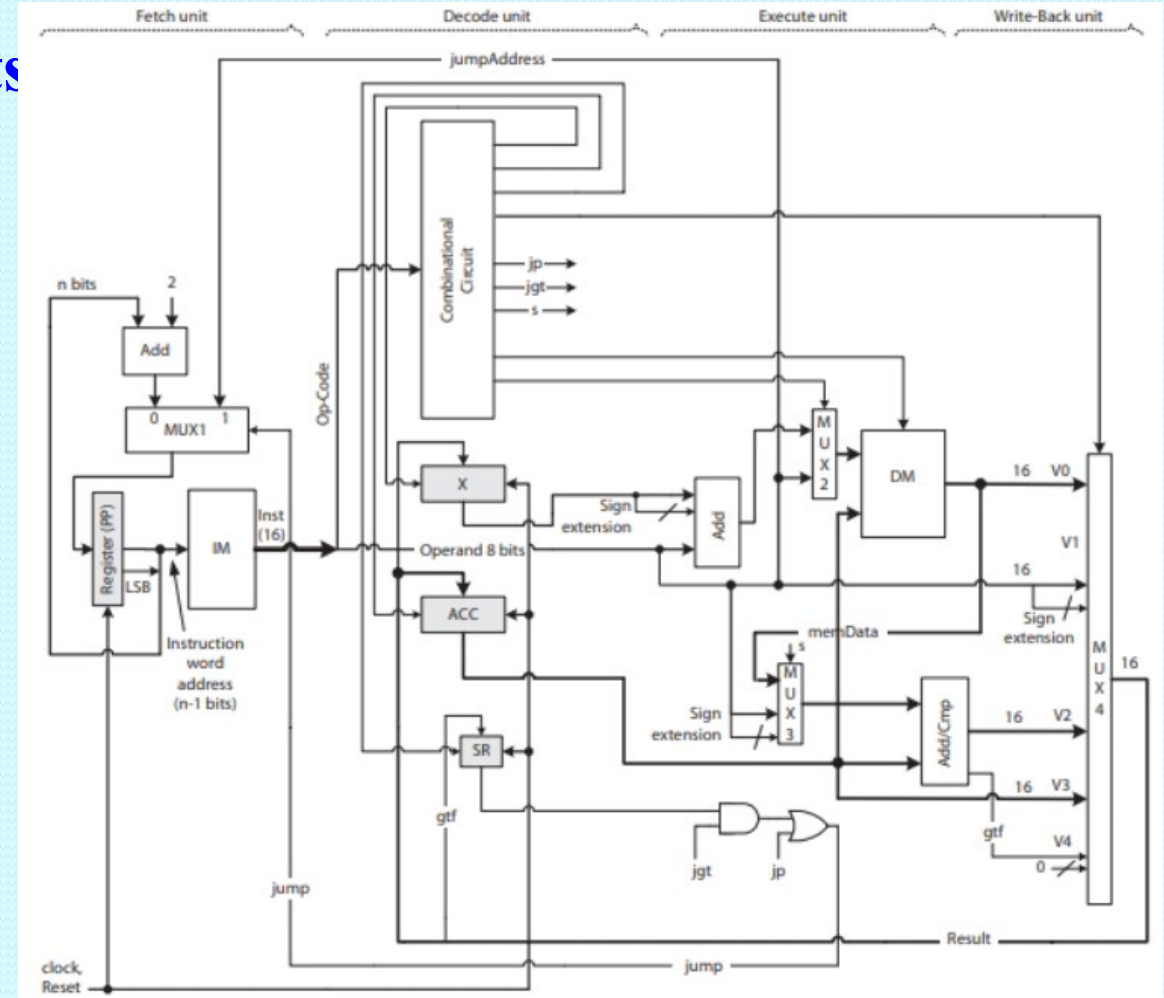
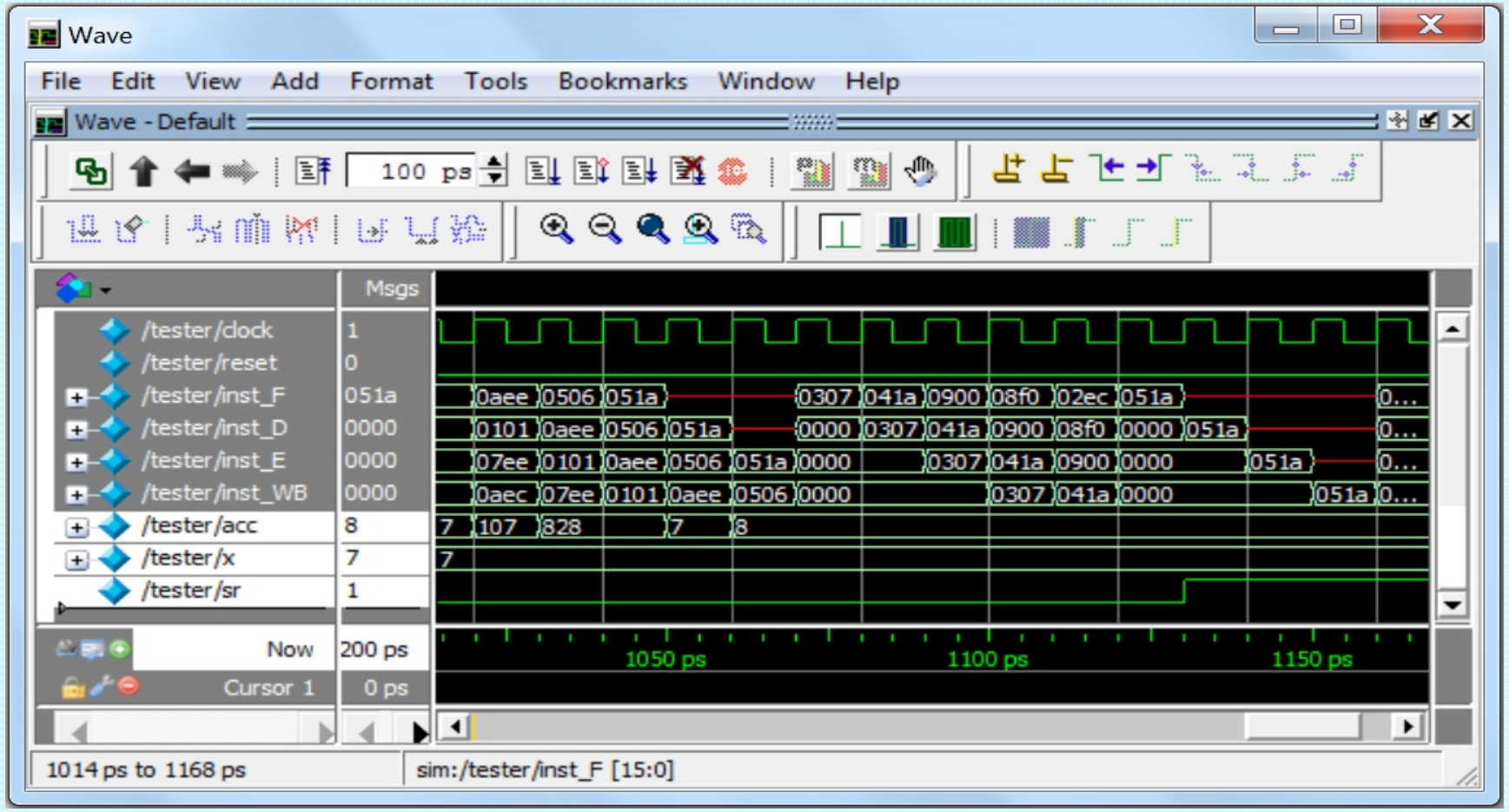


FIGURE 8.7 The Acc-ISA single-cycle data path to execute the program in Example 8.2.

Pipeline Simulation

- Illustrates pipeline flush on jumps



Sparc Example Assembly Program

- Note, arithmetic instructions access only registers
- Only “ld” and “st” instructions access memory

```
        st      %g0, [%fp-48]       //Store, Memory[fp -48] ← g0 (g0 always 0) (sum = 0)
        st      %g0, [%fp-44]       //Store, Memory[fp - 44] ← g0 (i = 0)
.LL5:
        ld      [%fp-44], %g1       //Load, g1 ← Memory[i]
        cmp     %g1, 7              //Compare, is g1 > 7?
        bg      .LL6                //Branch if greater than 7
        nop
        ld      [%fp-44], %g1       //Load, g1 ← Memory[i]
        sll     %g1, 2, %g2         //Compute ptr to array[i]: Shift Left Logical (i * 2)
        add     %fp, -8, %g1        //g1 ← fp - 8 (get memory location of array)
        add     %g2, %g1, %g1       //g1 ← g2 + g1 (array location + next i * 2)
        ld      [%fp-48], %g2       //Load sum, g2 ← Memory[fp -48]
        ld      [%g1-32], %g1       //Load array]i], g1 ← Memory[g1 - 32]
        add     %g2, %g1, %g1       //Array]i] + sum, g1 ← g2 + g1
        st      %g1, [%fp-48]       //Store sum, Memory[fp - 48] ← g1
        ld      [%fp-44], %g1       //Load i, g1 ← Memory[fp - 44]
        add     %g1, 1, %g1         //Increment, g1 ← g1 + 1
        st      %g1, [%fp-44]       //Store i, Memory[fp - 44] ← g1
        b       .LL5                //Branch to instruction ay LL5
        nop
.LL6:
```

Pentium IV Machine instructions (CISC)

- Variable size instructions
- Requires more complex data path and control unit

```
401340:       c7 45 d0 00 00 00 00   movl   $0x0,0xffffffd0(%ebp)
401347:       c7 45 d4 00 00 00 00   movl   $0x0,0xffffffd4(%ebp)
40134e:       83 7d d4 07            cmpl   $0x7,0xffffffd4(%ebp)
401352:       7f 13                  jg     401367 <_main+0x77>
401354:       8b 45 d4               mov    0xffffffd4(%ebp),%eax
401357:       8b 54 85 d8            mov    0xffffffd8(%ebp,%eax,4),%edx
40135b:       8d 45 d0               lea    0xffffffd0(%ebp),%eax
40135e:       01 10                  add    %edx,(%eax)
401360:       8d 45 d4               lea    0xffffffd4(%ebp),%eax
401363:       ff 00                  incl   (%eax)
401365:       eb e7                  jmp    40134e <_main+0x5e>
401367:       b8 02 00 00 00         ...
```

Performance Parameters

- **Cycles per instruction**
  - **Average number of clock cycles used to execute each instruction**

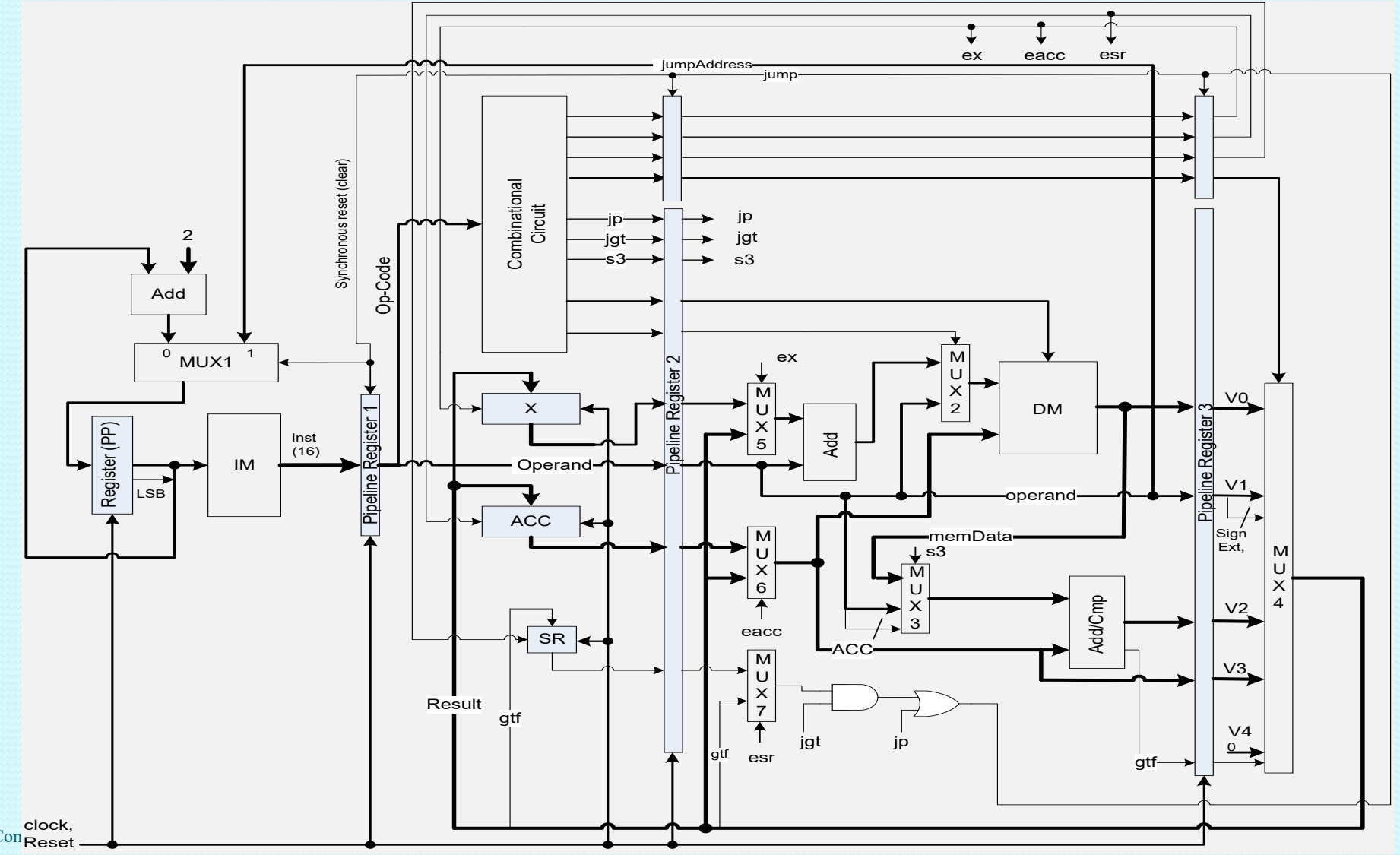
$$CPI = \frac{Number\ of\ clock\ cycles\ used\ (N)}{Number\ of\ instructions\ executed\ (n)}$$

- **Execution time (T) of a program**

$$T = CPI * n * \tau$$

Supplemental Slides

Acc-ISA Pipelined Data Path



Stack ISA - Example of assembly program: A = B * (C+D)

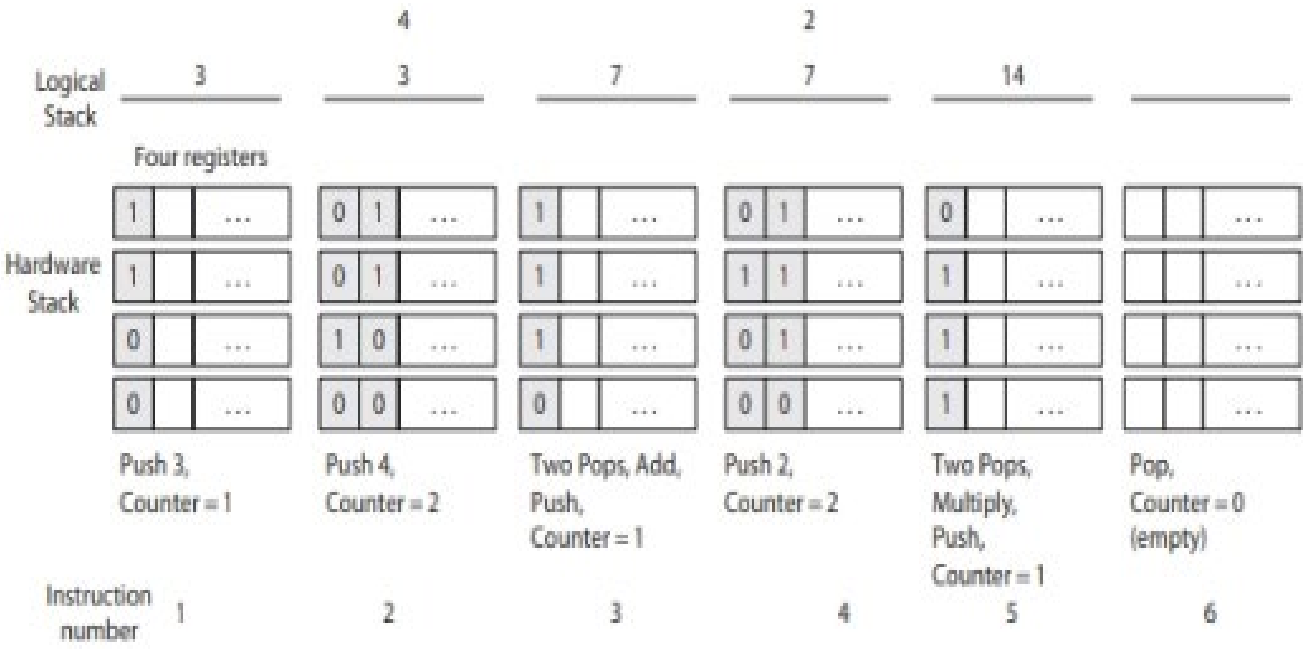


FIGURE 8.4 An illustration of stack content when computing the reverse polish notation C D + B * = A; it is assumed that (B) = 2, (C) = 3, and (D) = 4.

```
Instruction
number
  1:     PUSH (C) //stack ← M[C]
  2:     PUSH (D) //stack ← M[D]
  3:     ADD      //stack ← (C) + (D), values popped, added,
                  //result pushed
  4:     PUSH (B) //stack ← M[B]
  5:     MUL //stack ← ((C) + (D)) * (B), values popped, added,
             //result pushed
  6:     POP (A)  //M[A] ← (((C) + (D)) * (B)), value is popped
                  //and stored in memory
```

Acc ISA - Example of assembly program: A = B * (C+D)

1. LD (C) // ACC ← M[C]
2. ADD (D) // ACC ← ACC + M[D]
3. MUL (B) // ACC ← ACC * M[B]
4. ST(A) // M[A] ← ACC

CISC-ISA: Example of assembly program: A = B * (C+D)

B = 4; C = 5; D = 10

The value in R1 after execution of instruction No. 1 is 5.

The value of R1 after execution of instruction No. 3 is 60

1. LD R1, (C) // R1 ←M[C]
2. ADD R1, (D) // R1 ← R1 + M[D]
3. MUL R1, (B) // R1 ← R1 * M[B]
4. ST (A), R1 // M[A] ← R1

RISC-ISA: Example of assembly program: A = B * (C + D)

```
1.  LD   R1, (C )        //  R1 ← M[C]
2.  LD   R2, (D )        // R2 ← M[D]
3.  Add R3, R1, R2       // R3 ← R1 + R2
4.  LD   R4, (B)         // R4 ← M[B]
5. MUL R5, R3, R4        // R5 ← R3 * R4
6. ST    (A), R5         // M[A] ← R5
```

Computation is performed by a RISC ISA. A = B * (C + D). What is the value in R5 after the execution of code line # 6: (B = 5; C = 10; D = 15) ie: Code line # 6 has been completed. (20 pts)

R5 ?

Addressing Modes and Syntax Examples

- Immediate
  - E.g., Add R1, 9;
- Direct
  - E.g., ADD R1, (M[9]);
- Register
  - E.g., ADD R1, R2;
- Register direct
  - E.g., ADD R1, (R2);
- Register indexed
  - E.g., ADD R1, R2, (M[9]);

Operand Notation	Addressing Mode
V	I, immediate: V is an immediate input operand, a 2's complement number.
(V)	D, direct: V is a memory address and (V) indicates the content of memory address V (i.e., M[V]).
R	R, register: Indicates an input data register source or a destination register or both
R, (V)	X, indexed: V is a memory address and R + V is the address of the next data item in memory (i.e., M[R + V]).

TABLE 8.1 Examples of Addressing Modes

Immediate E.g., Add R1, 9;
//

Direct E.g., ADD R1, (M[9]);

Register E.g., ADD R1, R2;

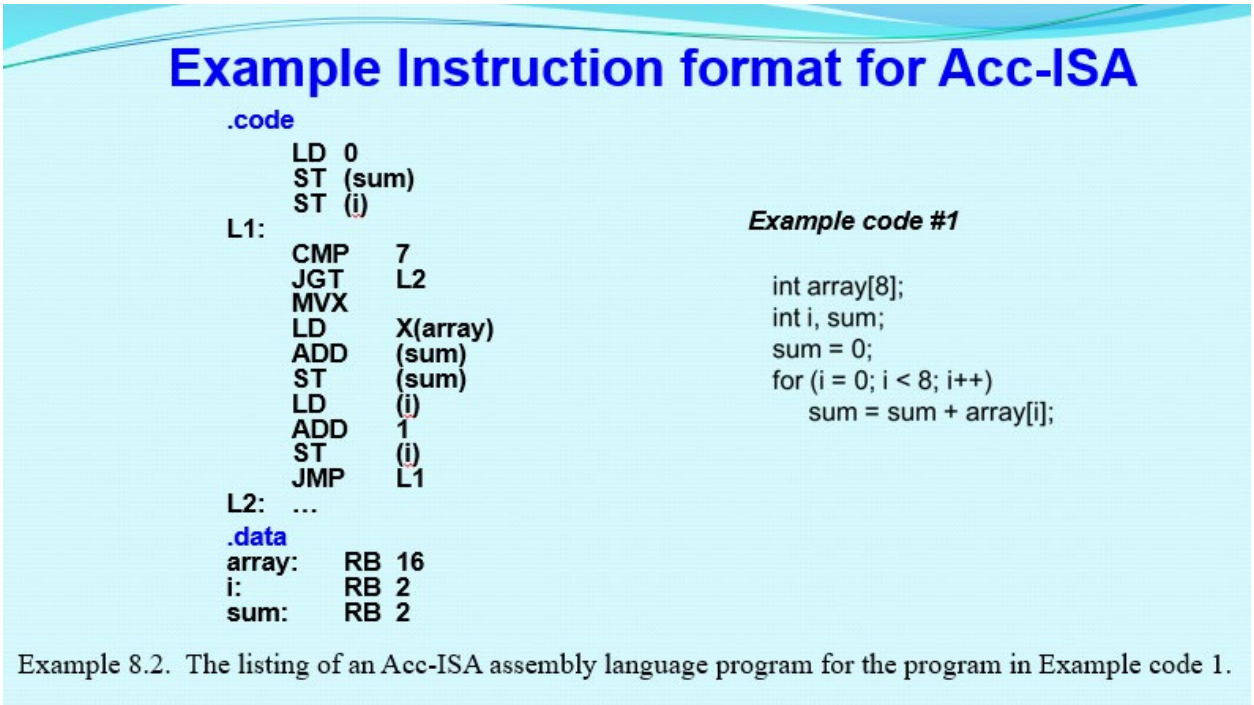
Register direct E.g., ADD R1, (R2);

Register indexed E.g., ADD R1, R2, (M[9]);

- Register indexed E.g., ADD R1, R2, (M[9]);
  - What is the value in R1 ?

RTN#1: R1 ← R1 +M[R2 + 9]

- Example 8.2 Assembly code listing of an Acc-ISA assembly language program for the high level (c) program in Example code 1



Example 8.2. The listing of an Acc-ISA assembly language program for the program in Example code 1.

.code L1: The listing of an Acc-ISA assembly language program for the program in

.code //start program code

```
        LD      0
        ST
        ST
L1:
        CMP     7
        JGT      L2
        MVX
        LD      X(array)
        ADD     (sum)
        ST      (sum)   )
        LD      (i)
```

```
ADD   1
ST    (i)
JMP   L1      //loop back   ( End of for loop)
```

Program level Translation:

Now, here is an example of a real C If-Then-Else:

```
if(x == 10)
{
    x = 0;
}
else
{
    x++;
}
```

Which gets translated into the following assembly/machine code:

X = 5; 0x 05 = 5 in decimal;

X = 0xA which is equal decimal 10.

```
Mov   eax,   $x
Cmp   eax,   0x0A   ; 0x0A = 10
```

```
Jne     else
Mov     eax,    0
Jmp     end
Else:
Inc     eax
End;
Mov     $x,     eax
```

Sequential (Seq) Circuit: Large Design

-Large Seq Circuit is made up of a data path and a control unit.

-Data path consists of seq and combinational circuit such as registers, counters, mux, decoders and ALU'S. (Arithmetic Logic Unit)

-High Clock frequency (freq) implies data path has a short maximum propagation delay.

-Max clock freq results in higher core junction temperature.

-Designer's goal is to stay within/under max Tj (Tj= temperature inside Computer Brain) to avoid signal integrity issues.

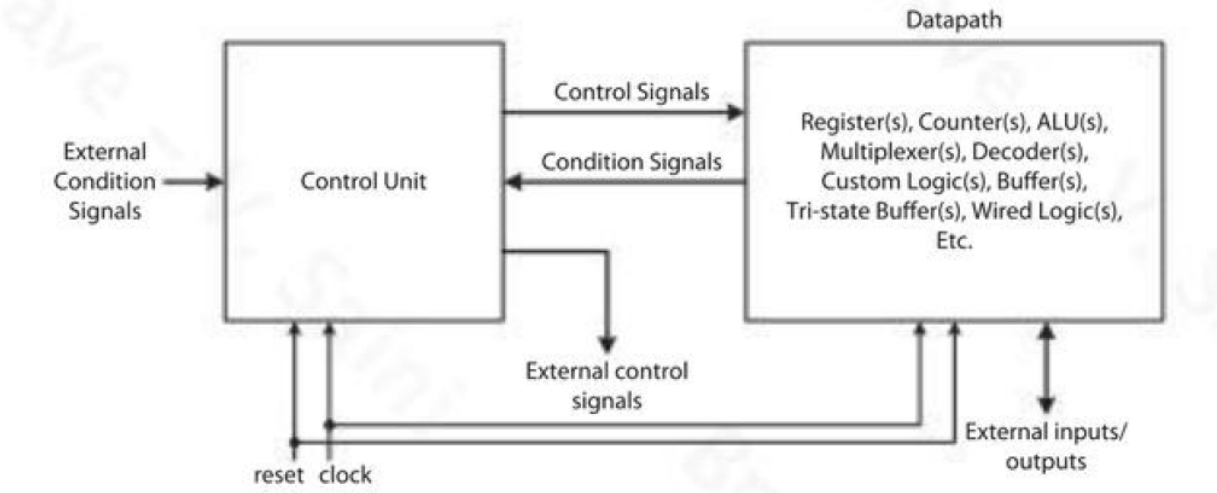
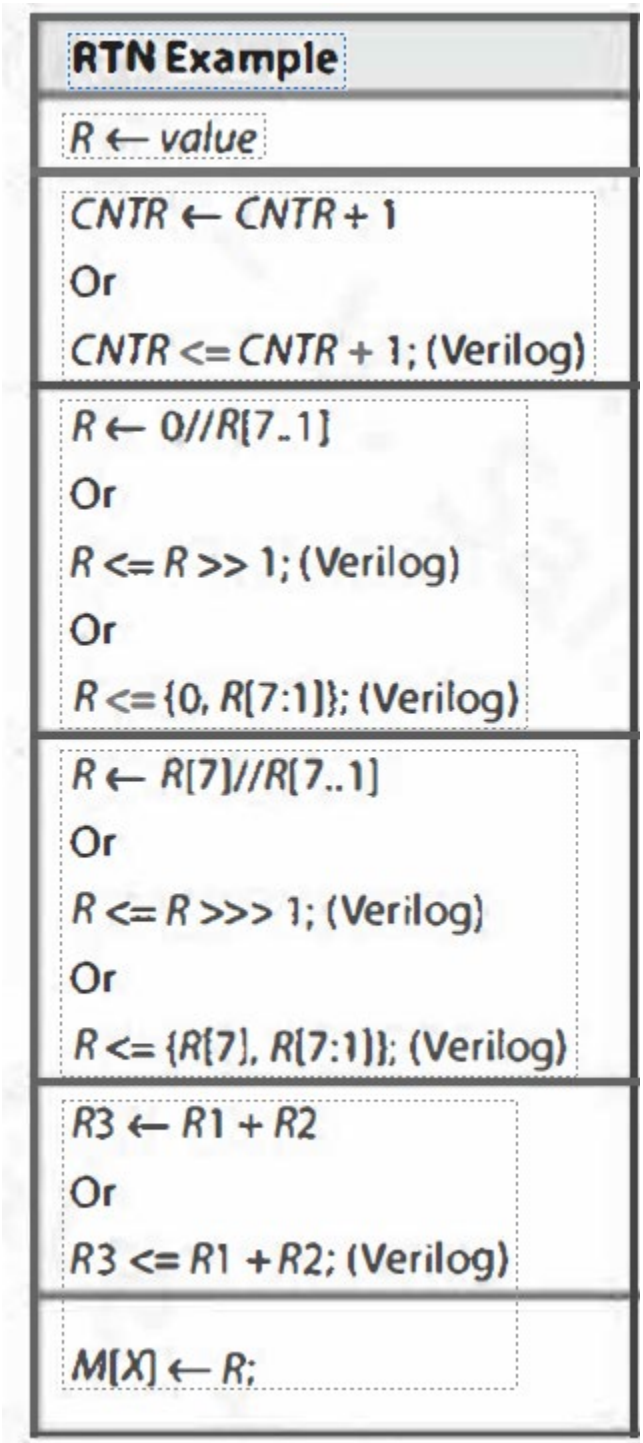


FIGURE 6.1 A diagram block of a large sequential circuit.

Register transfer notation (RTN) is used to describe an operation of a data path:

- Formally describes a data path operation
- Examples:



Data path types:

-Architecture of data path can be classified as single cycle, multiple cycles or pipelined.

-A single-cycled data path requires more hardware but a simpler control unit.

-A multicycle data path requires less hardware but generates results in several clock cycles.

-A pipelined data path also requires more hardware but can operate on multiple inputs concurrently.

-The single Data path contains two adders (+) modules and one adder/sub tractor (+/-) module.

-The single mode controls the functions of the adder/sub tractor modules.

-Time period is proportional to the propagation delay of the longest signal path that starts from the inputs of the first adder and ends at the input of the register.

--In general, a single cycle data path implements several simple and complex operations, its minimum clock would be proportional to the time required to complete most complex operations.

Sequential Circuit: Datapath (Timing)

Tcq: Clock to queue

Tst (T setup time): is defined as the minimum amount of time before the clock's active edge that the data must be stable for it to latch correctly.

Tcs: clock to skew

Thold or Th: Hold time is defined as the minimum amount of time after the clock's active edge during which data must be stable.

- Sequential circuit: single cycle data path: A + B + C + D or A + B + C – D

- Data path contains two adder modules and one adder/subtractor module
- The single mode controls the functions of the adder/subtractor modules

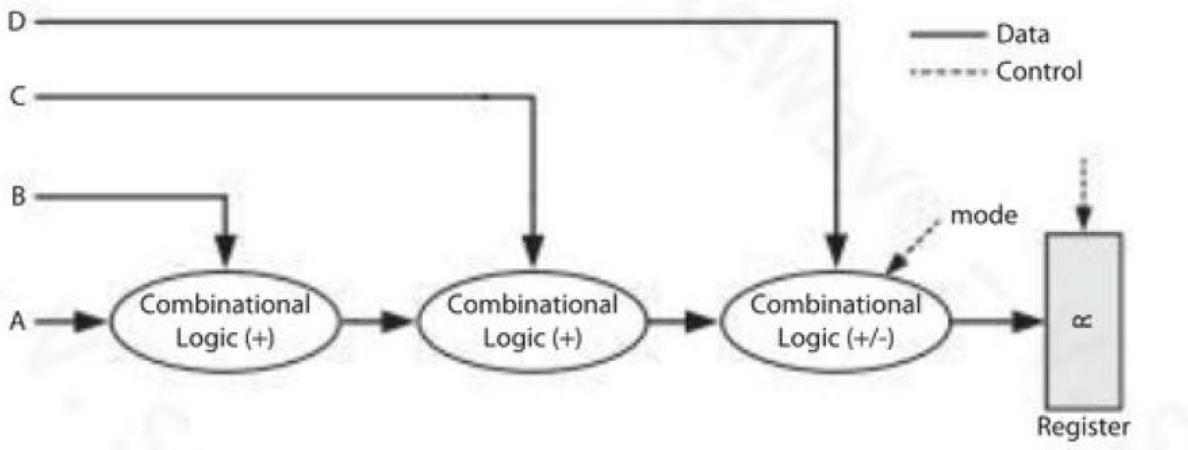


FIGURE 6.2 A single-cycle two-function data path that computes either $A+B+C+D$ or $A+B+C-D$ in one clock cycle.

Sequential circuit: Single cycle data path architecture

-Data path that computes either the quantity

- A + B + C + D or A + B + C – D

-Equation that estimates the minimum clock period (τ) required to run the data path

- Add stands for Adder; Sub stands for subtractor
- Δ is delta time delay from input to output

$$\tau_s >= 2\Delta_{add} + \Delta_{add/sub} + T_{st} + T_{cq} + T_{cs}$$

$\tau_s = \tau$-single-cycle

Sequential circuit: Multicycle data path architecture

-Data path that computes either the quantity

- A + B + C + D or A + B + C – D

-Equation that estimates the minimum clock period (τ) required to run the data path

- Add stands for Adder; Sub stands for subtractor; Mux stands for Multiplexor
- Δ is delta time delay from input to output

$$\tau_m >= \Delta_{mux1} + \Delta_{add/sub} + \Delta_{mux2} + T_{st} + T_{cq} + T_{cs}$$

$$\tau_m = \tau\text{-multicycle}$$

-A multicycle data path requires that a computation be divided and computed in steps.

-A multi cycle algorithm to implement R <---- A + B + C + D or A + B + C – D; (5 possible simple operations)

Cycle 1: R <----- A

Cycle 2: R <----- R + B

Cycle 3: R <----- R + C

Cycle 4: If mode == 0, then R <----- R + D; otherwise R <----- R – D

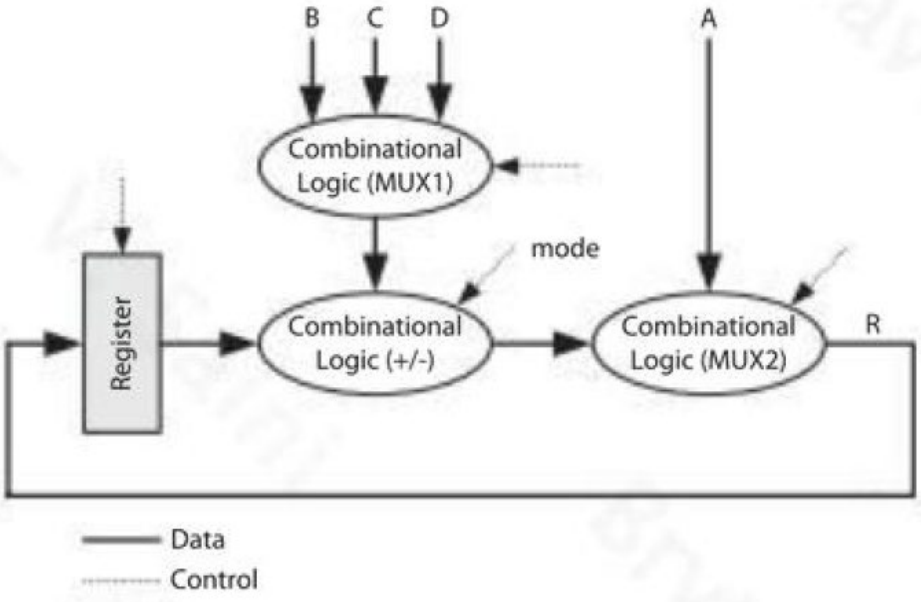


FIGURE 6.3 A multicycle data path requiring four clock cycles to compute $A+B+C+D$ or $A+B+C-D$.

Seq circuit: Pipelined Data path architecture

-Data path that computes stream of quantities Ai + bi + Ci +/- Di

- Ai + bi + Ci +/- Di
- τ = represents minimum clock period for pipeline data path architecture
- Equation that estimates the minimum clock period (τ) required to run the data path
    - Add stands for Adder; Sub stands for subtractor; Mux stands for Multiplexor
    - Δ is delta time delay from input to output

$$\tau_p >= \Delta_{add/sub} + T_{st} + T_{cq} + T_{cs}$$

$\tau_{-p} = \tau_{-pipeline}$

- Computing stream of quantities Ai + bi + Ci +/- Di.

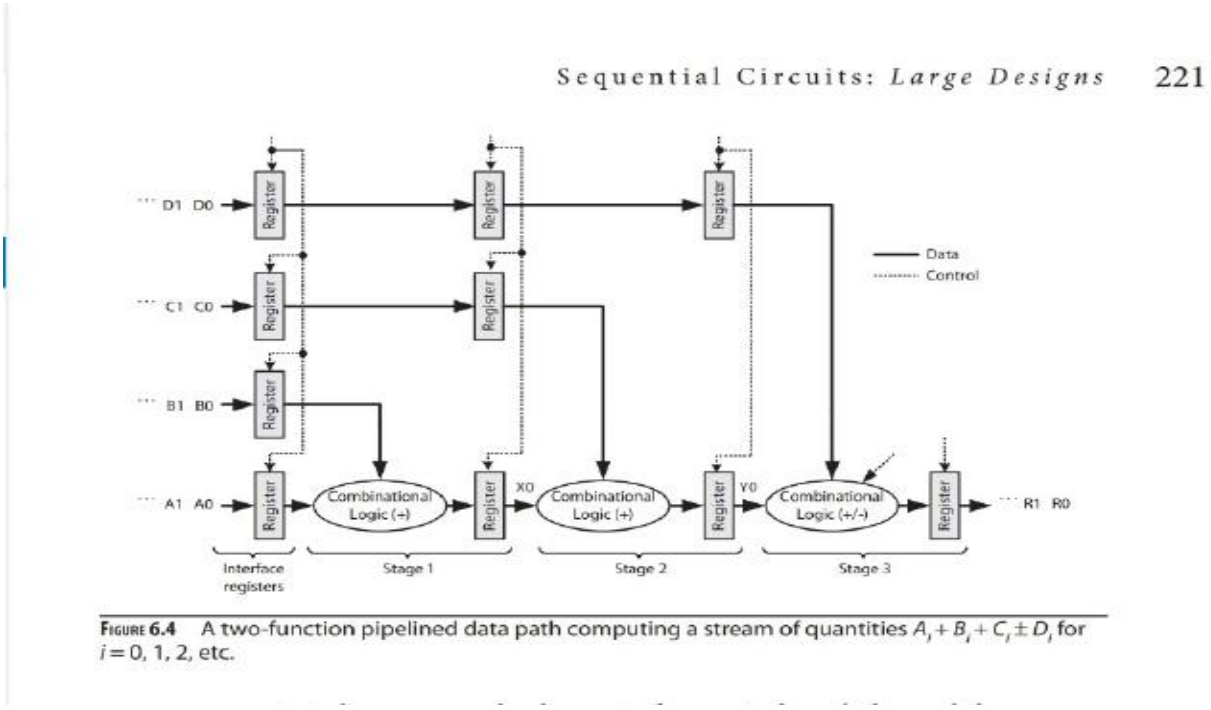
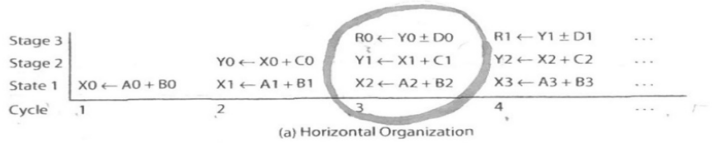


FIGURE 6.4 A two-function pipelined data path computing a stream of quantities $A_i + B_i + C_i \pm D_i$ for $i = 0, 1, 2$, etc.

- <u>Seq circuit: Pipelined Data path architecture (Cont)</u>

  Refer to Figure 6.5 -Horizontal Pipeline chart

A pipeline uses more hardware, similar to a single-cycle data path, but operates with a higher-frequency clock, similar to a multicycle data path. Furthermore, it can process a stream of data a lot faster than the other two data paths. The clock period of a pipelined data path is proportional to the propagation delay of its



(a) Horizontal Organization

Fig 6.5

Pipeline chart for 3 stage pipeline

Refer to Figure 6.5 -Horizontal Pipeline chart (Cont)

- The pipeline chart in Fig. 6.5(a) has horizontal organization with clock cycles shown on the x-axis.
  - Pipeline chart illustrates various chart, does not include 1 cycle delay caused by interfacing registers
  - Has horizontal organization with clock cycles shown on x-axis.

Sequential circuit: Pipeline performance

- In the example, R0 being the first result, requires 3 clock cycles to complete. (Ignore 1 cycle for interface registers)
- After R0, R1, R2…..Rn each output requires 1 clock cycle to produce results.
- Reduces time required to compute N final results.
- K stage (linear) pipeline requires K cycles for output.
- Equation Eq6.4 estimates total time (Tpipeline) required to process stream size of N using K stage pipeline

$$T_{pipeline} = K * \tau_p + (N-1)* \tau_p \qquad \text{(Eq 6.4)}$$

- Estimates total time- Tsingle -cycle to process data stream of size N using single cycle data path.

$$T_{Single\text{-}cycle} = N * K * \tau_p \qquad \text{(Eq 6.5)}$$

- Speed up is performance parameter that measures performance of faster system to a slower system.
- Defined as ratio of the time required by a slower system over faster system.
- Equation that defines speedup between a faster pipeline data path compared slower single cycle Datapath.

$$Speedup = T_{single\text{-}cycle}/T_{pipeline} = N * k * \tau_p / k* \tau_p + (n-1) * \tau_p \quad \text{(eq. 6.6)}$$

$$= N * K/ (K +N -1)$$

- Efficiency is performance parameter that measures how well a system's resource's are utilized.

- Overall efficiency of a system is defined as the ratio of it's speedup to its maximum possible speedup.
- Efficiency = Speedup/K = N/ (k+N-1)

- As N approaches infinity, efficiency of pipeline approaches 100%.
- Throughput is performance parameter that measures a system's rate of processing
- Indicates the # of items (N) performed per second
- Eq. 6.8 defines the throughput of a linear pipeline with k stages

Throughput = N/Tpipeline = $N/[(k\ \tau) + (N-1) * \tau]$

Calculate SpeedUp:

$$\text{Speedup} = \frac{T_{\text{single cycle}}}{T_{\text{pipeline}}} = \frac{NK\tau_p}{K\tau_p + (N-1)\tau_p}$$

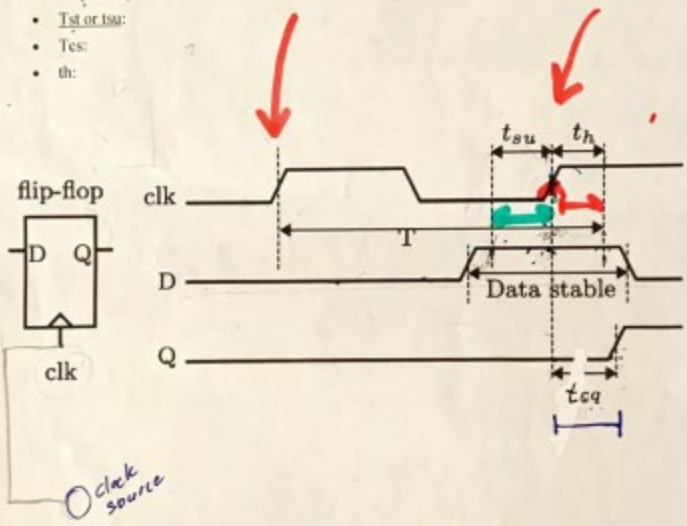
$$= \frac{NK\tau_p}{K\tau_p + N\tau_p - \tau_p}$$

$$= \frac{NK\cancel{\tau_p}}{\cancel{\tau_p}(K + N - 1)}$$

$$\text{Speedup} = \frac{NK}{(K + N - 1)}$$

Timing Diagram of D flip flop:

- Tcq or dcq:
- Tst or tsu:
- Tcs:
- th:



clock source

Chapter 7 – Memory Tech introduction

- A register is storage for single value for quick access/readily available
- Memory is designed to store the code and data of programs during execution
- Storage memory technologies require less H/W than flip-flop
- Will go over commonly used memory technologies and their applications
- Storage size is defined in terms of bytes

Unit	Reads	Approximate Size
1kb	One Kilobyte	10^3 Bytes
1MB	One Megabyte	10^6 Bytes
1 GB	One Gigabyte	10^9 bytes
1 TB	One Terabyte	10^12 Bytes

- Memory technologies are categorized as Read only memory or Random-Access memory
- ROM memory is nonvolatile and retains their content even when they are not powered
- RAM are volatile and would lose their content when not powered
- Examples of Non-Volatile memory technologies are magnetic disks, flash memory and optical disk such as CD ROM's
- The content of a ROM cell is fixed at logic 0 or 1
- An Electrically erasable PROM (EEPROM) is most commonly used today
- EEPROM cells can be programmed only a certain number of times
- Ram's functions as the Main storage for programs and data

- SRAM vs DRAM
    - SRAM is preferable for register files and L1/L2 caches
        - Faster access time
        - No Refreshes needed
        - Simpler Manufacturing.
        - Lower density
- DRAM is preferable for stand alone memory chips
    - Much higher capacity per area
    - Higher density.
    - Lower cost – i.e. cost vs performance trade off.
    - Continuously refreshed to keep the capacitors charged up.
- A memory timing diagram precisely illustrates memory communication protocols.
- Time required to select target cells and perform a read or write operation is called a memory access time
- A memory cycle includes both an access time and data transfer time.
- The access time is directly proportional to the size of the memory cell array.
- The decoders in turn determine the time required to activate a target row and access target cells

A. Peak Memory Bandwidth Exercise

Example 1: Consider a 32-bit data bus SDRAM. Given that the clock frequency of the bus is 100MHz, what is the peak memory bandwidth in megabyte per second (MBs)?

32 -bit data bus:

1 byte = 8 bits, so 32 bits x 1 byte/8bits = 4Bytes per cycle

1 MB = 1,000,000 Bytes
1 Hz = 1 cycle/sec

1 MHz= 1 million Hz = 1,000,000 * 1 cycle/sec = 1,000, 000 cycles/sec

100 MHz = 100 Million Hz = 100 Million * 1 cycle/sec = 100,000,000 cycles/sec

100,000,000 cycle/sec * 4 Bytes/cycle = 400,000,000 Bytes/sec
400,~~000,000~~ ~~Bytes~~/sec * 1MB/1,~~000,000~~ ~~Bytes~~ = 400 MB/sec

7.5.1 SRAM

Figure 7.16 illustrates an SRAM read cycle from the memory point of view.

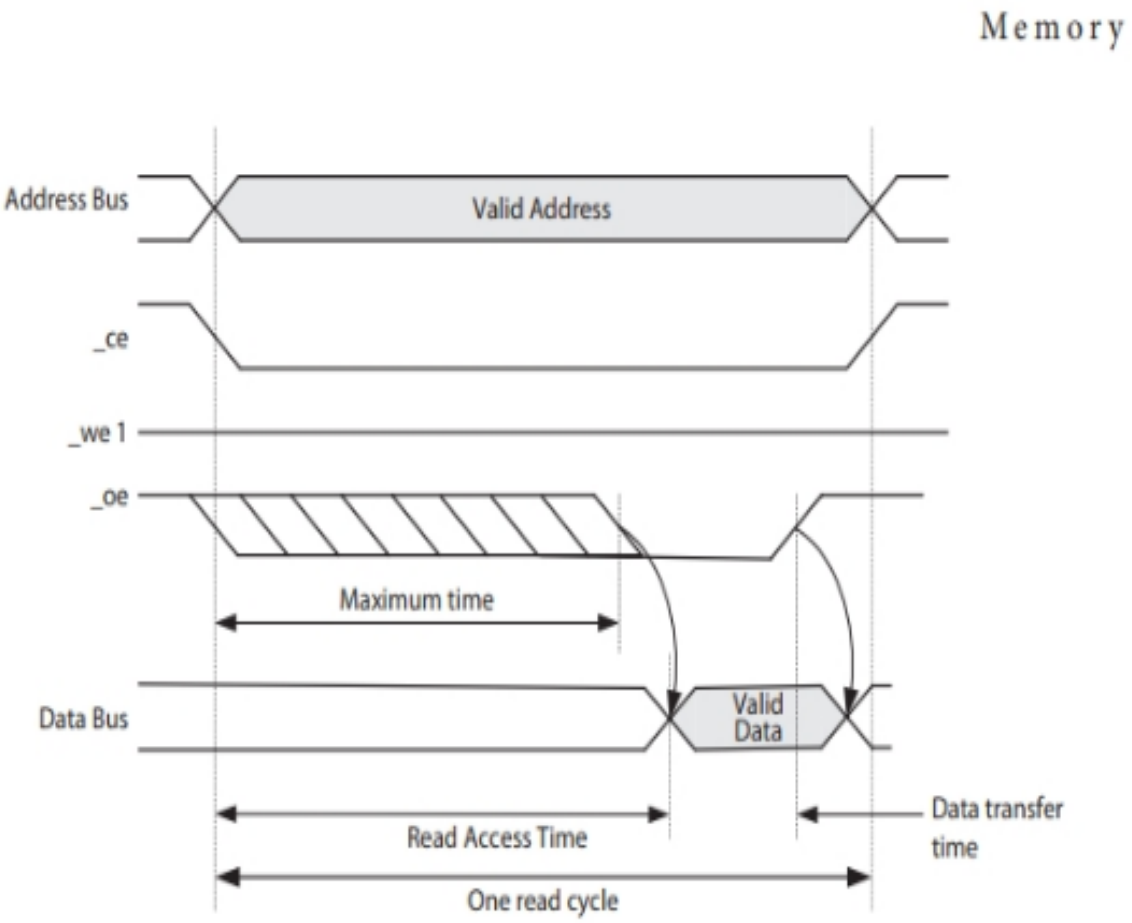


FIGURE 7.16 An SRAM read cycle from the memory point of view.

Figure 7.17 illustrates an SRAM memory write cycle.

A memory cycle is initiated by CPU and typically takes multiple CPU clock cycles to complete.

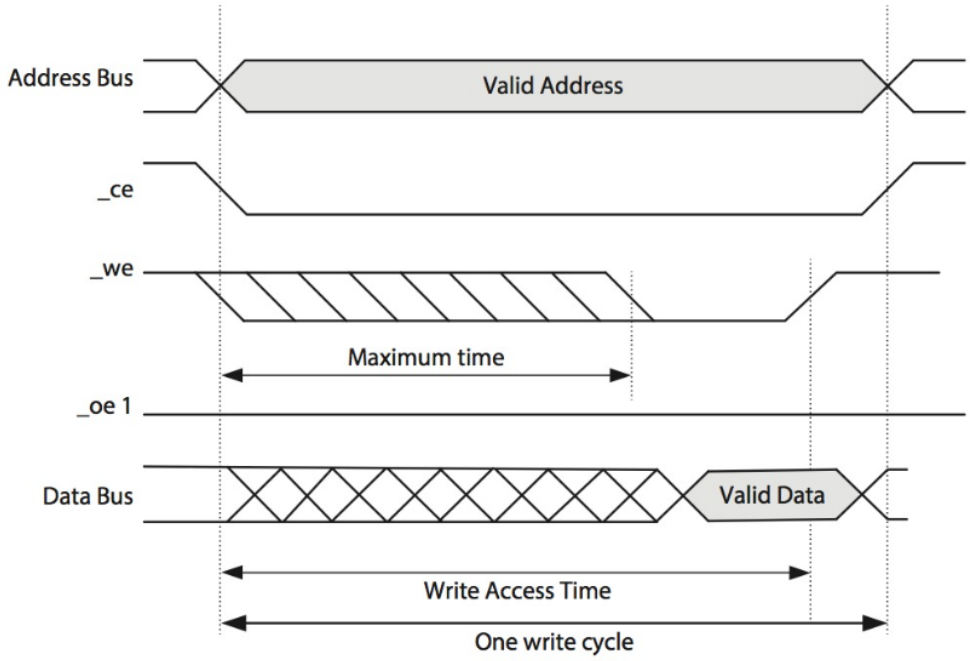


FIGURE 7.17 An SRAM write cycle from a memory point of view.

Tri-State Buffer

A tri-state buffer is similar to a buffer, but it adds an additional "enable" input that controls whether the primary input is passed to its output or not. If the "enable" inputs signal is **true**, the tri-state buffer behaves like a normal buffer. If the "enable" input signal is **false**, the tri-state buffer passes a *high impedance* (or hi-Z) signal, which effectively disconnects its output from the circuit.

Truth table for a tri-state buffer

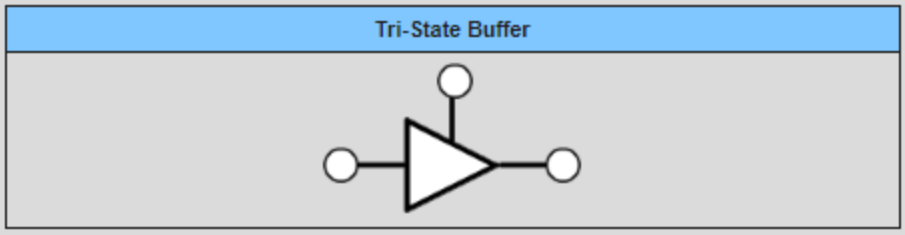
Enable Input	Input A	Output
false	false	hi-Z
false	true	hi-Z
true	false	false
true	true	true

Tri-state buffers are often connected to a *bus* which allows multiple signals to travel along the same connection.

The *truth table* for a tri-state buffer appears to the right.

Symbols

The symbol below can be used to represent a tri-state buffer.



tri-state buffer is similar to a buffer, but it adds an additional "enable" input that controls whether the primary input is passed to its output or not. If the "enable" inputs signal is **true**, the tri-state buffer behaves like a normal buffer. If the "enable" input signal is **false**, the tri-state buffer passes a *high impedance* (or hi-Z) signal, which effectively disconnects its output from the circuit.

Tri-state buffers are often connected to a *bus* which allows multiple signals to travel along the same connection.

The *truth table* for a tri-state buffer appears to the right.

System Performance Advantages of Higher Density SRAMs

By Jayasree Nayar, Applications Engineer Staff, Cypress Semiconductor Corp.

Executive Summary

With ever increasing system bandwidth requirements on the order of multigigabits/sec, SRAMs need to be optimized for higher density and performance as well as reliability, especially for look up table and packet buffering in networking applications. This article will address the different factors affecting system performance using Synchronous SRAMs.

With ever-increasing system bandwidth requirements of the order of multigigabits/sec, SRAMs need to be optimized for higher density and performance.

Today, Synchronous SRAMs are used in networking applications, such as L2/L3 caches and buffers, and SRAM performance is a significant factor that contributes to overall system performance. The need for high-density, high-performance SRAMs with high reliability is crucial for networking applications, especially for look-up table and statistics buffers.

Some of the advantages of higher density SRAMs include:

- Reduced board area which can be achieved by replacing two or more lower density parts with a higher density part having the same package footprint
- Increased density to handle more data traffic leading to better system performance
- Fewer interface signals and lower pin count leading to improved signal integrity, timing management, and skew control
- Lower cost
- Lower power
- Lower latency

This article will address the different factors affecting system performance using Synchronous SRAMs. Some of the critical factors that enhance system performance using Synchronous SRAMs are:

- Higher density
- Higher operating frequencies and wider data bus
- Lower latency
- Lower power
- Improved signal integrity

Higher density

The higher the SRAM memory performance, the better will the system performance.

Higher SRAM memory performance can be enhanced by increasing the operating frequencies and increasing the capacity of storing data.

The latter can be achieved by increasing the density of the SRAMs. This enables more data to be buffered as well as increasing the number of transactions that can be performed in a fixed time interval. The greater the transaction speed, the higher the density required for the SRAMs. Therefore, increased density of the SRAMs enables it to handle more data traffic leading to better system performance.

Another advantage of using higher density SRAMs is reduced board space. System designers strive to reduce the board area occupied by the different components for reducing overall board cost. Because SRAM vendors may not be offering the

Published in Comms Design http://www.commsdesign.com/showArticle.jhtml?articleID=212201996&pgno=2 December 2008

density required in their applications, system designers often have to use multiple SRAMs to meet the memory density requirements of the system. If higher density SRAMs are available, then 2 or more SRAMs can be replaced by a single SRAM. For example, a single 144M density SRAM can replace two 72M density SRAMs or four 36M density SRAMs mounted on the board thereby reducing board space occupied by the SRAMs (see Figures 1 and 2).

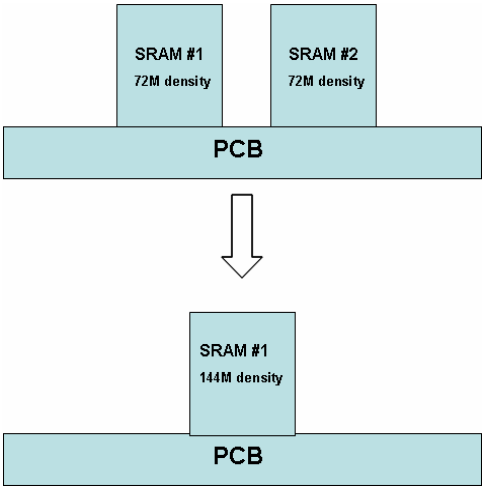


Figure 1: Two 72M density SRAMs replaced by a single 144M density SRAM

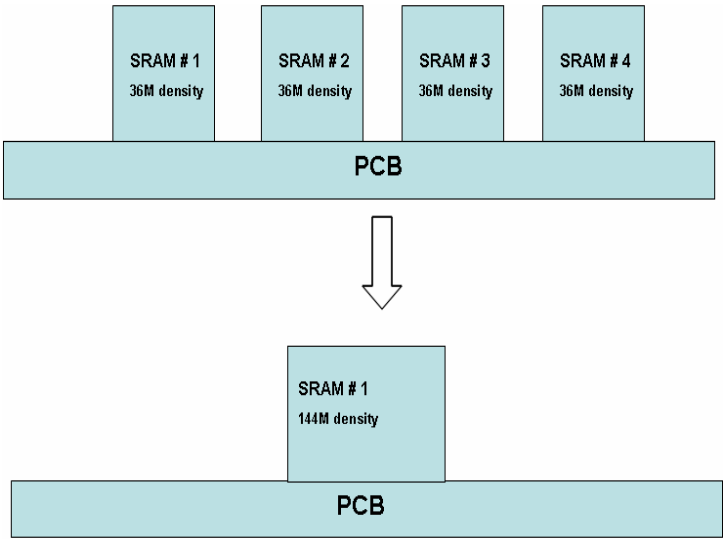


Figure 2: Four 36M density SRAMs replaced by a single 144M density SRAM

Higher Operating Frequencies and Wider Data Bus

The higher the frequency of the SRAMs and greater the bus width, the better the bandwidth. The more bandwidth, the more number of bits that can be processed per time interval, leading to better performance.

Below is the formula for calculating bandwidth in SRAMs:

For Single data rate SRAMs:

- Bandwidth = Bus width x SRAM Frequency
  - Example: for a Single data rate Synchronous SRAM operating at 250MHz with a bus width of 36:
  - Bandwidth = 36 * 250 * 10^6 = 9Gb/sec.

For Double data rate SRAMs:

- Bandwidth = Bus width x 2(DDR) x SRAM Frequency
  - Example: for a Double data rate Synchronous SRAM operating at 400MHz with a bus width of 36:
  - Bandwidth = 36 * 2 * 400 * 10^6 = 28.8Gb/sec.

Lower Latency

Lower latency results from the fact that the higher density part is capable of operating at higher frequencies. Reduced latency enables data to driven out of the bus faster, leading to increased bus efficiency. Reduced latency also enables more transactions to be performed on the memory within a certain time interval.

Lower power

One of the main challenges faced by system designers is to lower overall system power. The greater the density of the SRAMs, the fewer SRAMs required to satisfy system density requirements. Reducing the number of SRAMs used also lowers overall power consumption.

Consider the case of the following 2 density SRAMs.

144Mb density SRAM with Vdd =1.8V

Idd =1070mA

Core power = Vdd * Idd =1.926W

where Vdd is the voltage of the core power supply and Idd is the operating current

72Mb density SRAM with Vdd =1.8V

Idd =1000mA

Core power = Vdd * Idd =1.800W

where Vdd is the voltage of the core power supply and Idd is the operating current

Bear in mind that two 72M SRAMs could be replaced with a single 144M SRAM.

Core power dissipated by two 72M SRAMs = 1.8 * 2=3.6W

Core Power dissipated by a single 144M SRAM = 1.926W

Please refer to Figures 3 & 4 below:

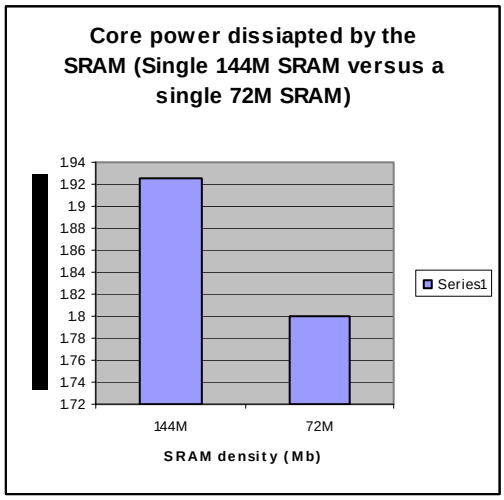


Figure 3:

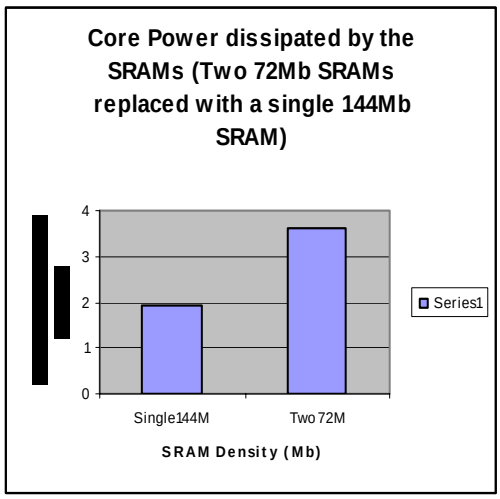


Figure 4:

Therefore, though the power consumed by a single 72M SRAM is lower than a single 144M SRAM, the power consumed by a single 144M SRAM is lower than the power consumed by two 72M SRAMs combined by about ~47%.

Improved Signal Integrity

The higher the density of the SRAMs, the fewer number of interface signals required, thus leading to improved signal integrity at high operating frequencies. This also eliminates the need to clamshell SRAMs to reduce board space, thus making the board routing less complicated and signal integrity better. Fewer traces also reduces crosstalk, thereby improving signal integrity, especially at high frequencies.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone: 408-943-2600
Fax: 408-943-4730
http://www.cypress.com

Week 3

DeMorgan's Theory Explained:

Theorem 1: Theorem states that two or more variables NAND'ed together is the same as the two terms inverted and OR'ed

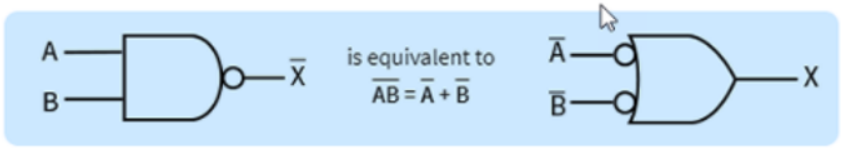
$$\overline{A \cdot B} \equiv \overline{A} + \overline{B}$$

Thereon 2: Theorem states that two or more variables NOR'ed together is the same as the two variables inverted and AND'ed

$$\overline{A + B} \equiv \overline{A} \cdot \overline{B}$$

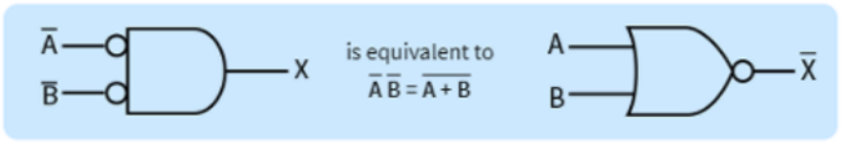
DeMorgan's Theorem illustrated in Gates (Schematic):

$$\overline{A \cdot B} \equiv \overline{A} + \overline{B}$$



A NAND gate is equivalent to an inverted input OR gate

$$\overline{A + B} \equiv \overline{A} \cdot \overline{B}$$



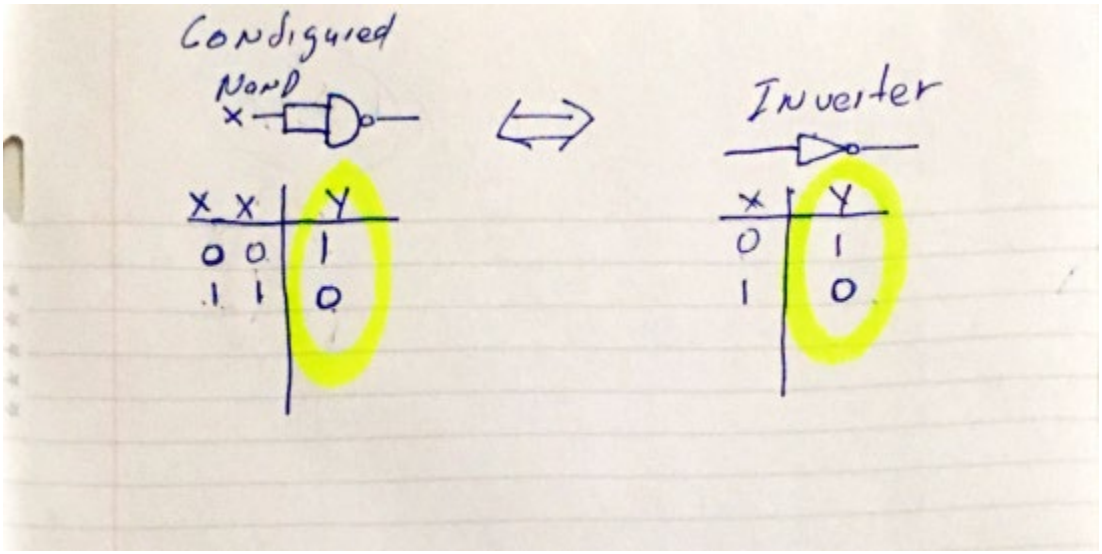
A NOR gate is equivalent to an AND gate followed by inversion

Lets to go video to work out an example: Refer to video example:

DeMorgan's Theorem

F1 = F2

Implementation of SOP expression with NAND Gates: Inverter



Karnaugh Maps (K – Map)

- A Karnaugh map is a graphical representation of a logic function.
- The map for n-input logic function is an array with 2^N cells,(One for each possible input combination or minterm.
    - 2 variables → 4 cells; 3 variables → 8 cells
- To represent a logic function o a k-map, we simply copy 1's from the truth table to the corresponding cells in the K-Map.
- Implicants are squares or rectangles made up of minterms
- Pairs of adjacent "1" cells in the K-map have minterms that differ in ONLY one variable
- The minterm pairs can be combined into a single product term using the generalization of theorem T10:
    - T10 = Term * Y + Term * Y' = (Term)
- Thus, we can use k-map to simplify the sum of the function.

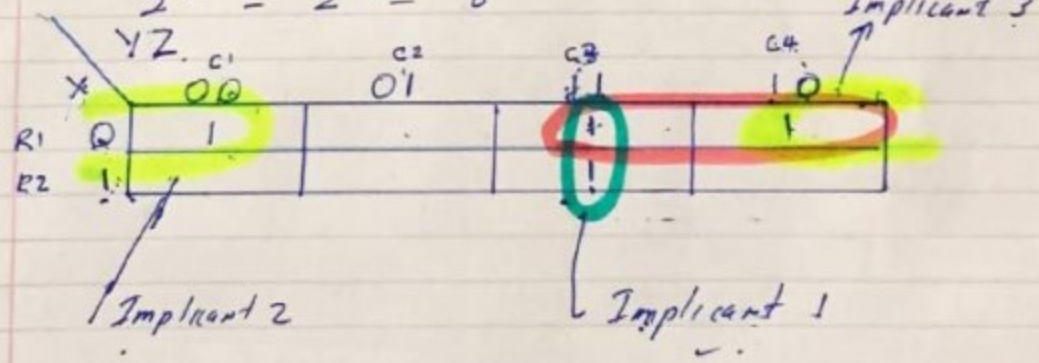
Three Variable K-Map.

Truth table

	X	Y	Z	F
R1	0	0	0	1
R2	0	0	1	0
R3	0	1	0	1
R4	0	1	1	1
R5	1	0	0	0
R6	1	0	1	0
R7	1	1	0	0
R8	1	1	1	1

$$F = \bar{X}\bar{Y}\bar{Z} + \bar{X}Y\bar{Z} + \bar{X}YZ + XYZ$$
$$= \bar{Y}\bar{Z} + \bar{X}Y + \bar{X}\bar{Z}$$

$2^N = 2^3 = 8$



Combinational circuit -Small Design Summary

Truth table:

LUT (LOOK UP TABLE)

Entire truth table ..

Advantage:

Disadvantage:.

Minimal logic circuit Implementation

Advantage:

Disadvantage:

Week 4

Design of Full adder

Carry Propagate Adder (CPA)

- Full adder is an arithmetic logic unit or circuit to add two single digit numbers with Carry-in
- It is still combinational circuit with inputs Ax and Bx
    - Sx – Sum (1bit); C-in; Cx – Carry out;
- Adder is also called Ripple Carry Adder (RCA)
- Fundamental circuit for an adder and has longest propagation delay that is proportional to the number of the carry bits.
- Equation to estimate the propagation delay of an n-bit CPA. ΔFAC and ΔFAS stand for delta time for carry bit and delta time for sum signal propagation delay of a Full Adder (FA)

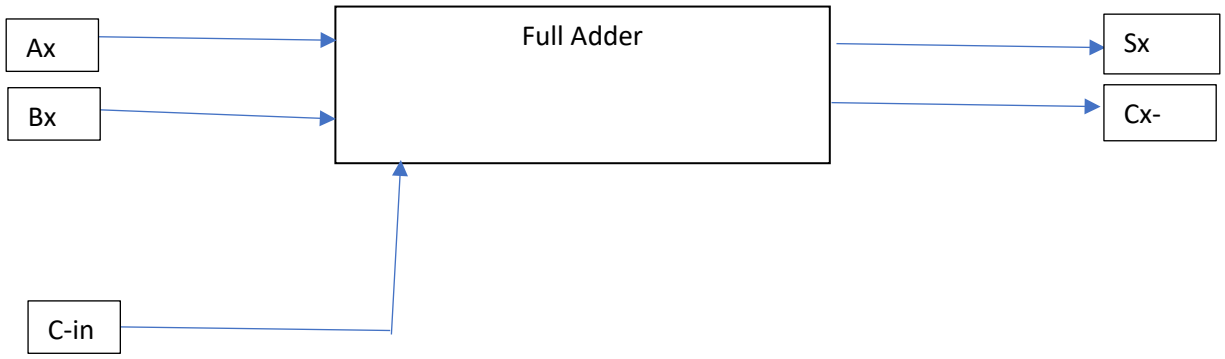
$$\Delta CPA(n) = [\ (n-1) * \Delta FA_C\ ] + \Delta FA_S \quad Eq(2.8)$$

- Using AND/OR/EOR gate, Δ for AND/OR gate is = 0.1 ns and Δ for EOR = 0.3ns, ΔFAC = 0.2ns and ΔFAS = 0.3ns, as per Eq(2.8)
    - For 8 bit CPA (n=8), the equation to calculate propagation delay is as follows:

$$\Delta CPA(n) = (n-1)(0.2ns) + 0.3ns$$
$$= (8-1)\_(0.2nx) + 0.3ns$$

-

Block Diagram of Full adder

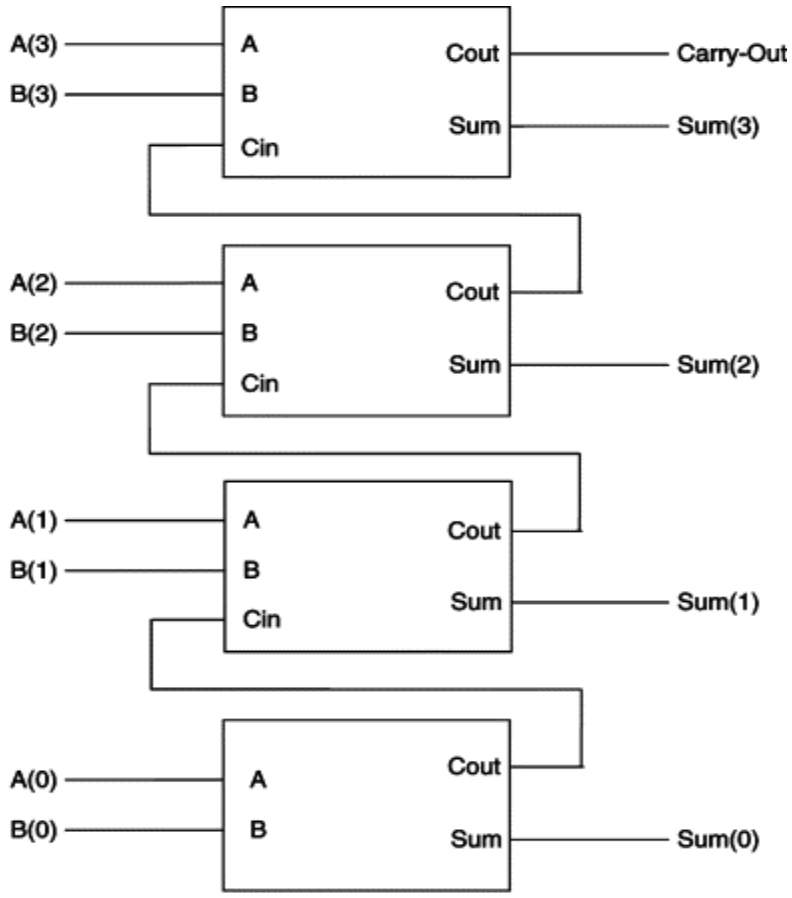


Design of Full Adder

-

Inputs			Outputs	
A	B	C – IN	Sum	C - Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Schematic of a 4 bit Cary Propagate Adder (CPA)



4-bit binary addition

```
                A(3)    A(2)    A(1)    A(0)
                B(3)    B(2)    B(1)    B(0)    +
-------------------------------------------------
 Carry-Out   Sum(3)  Sum(2)  Sum(1)  Sum(0)
-------------------------------------------------
```

4-bit binary addition

```
                A(3)    A(2)    A(1)    A(0)
                B(3)    B(2)    B(1)    B(0)    +
-------------------------------------------------
 Carry-Out  Sum(3)  Sum(2)  Sum(1)  Sum(0)
-------------------------------------------------
```

Binary Addition review:

A	B	S	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	**0**	**1**

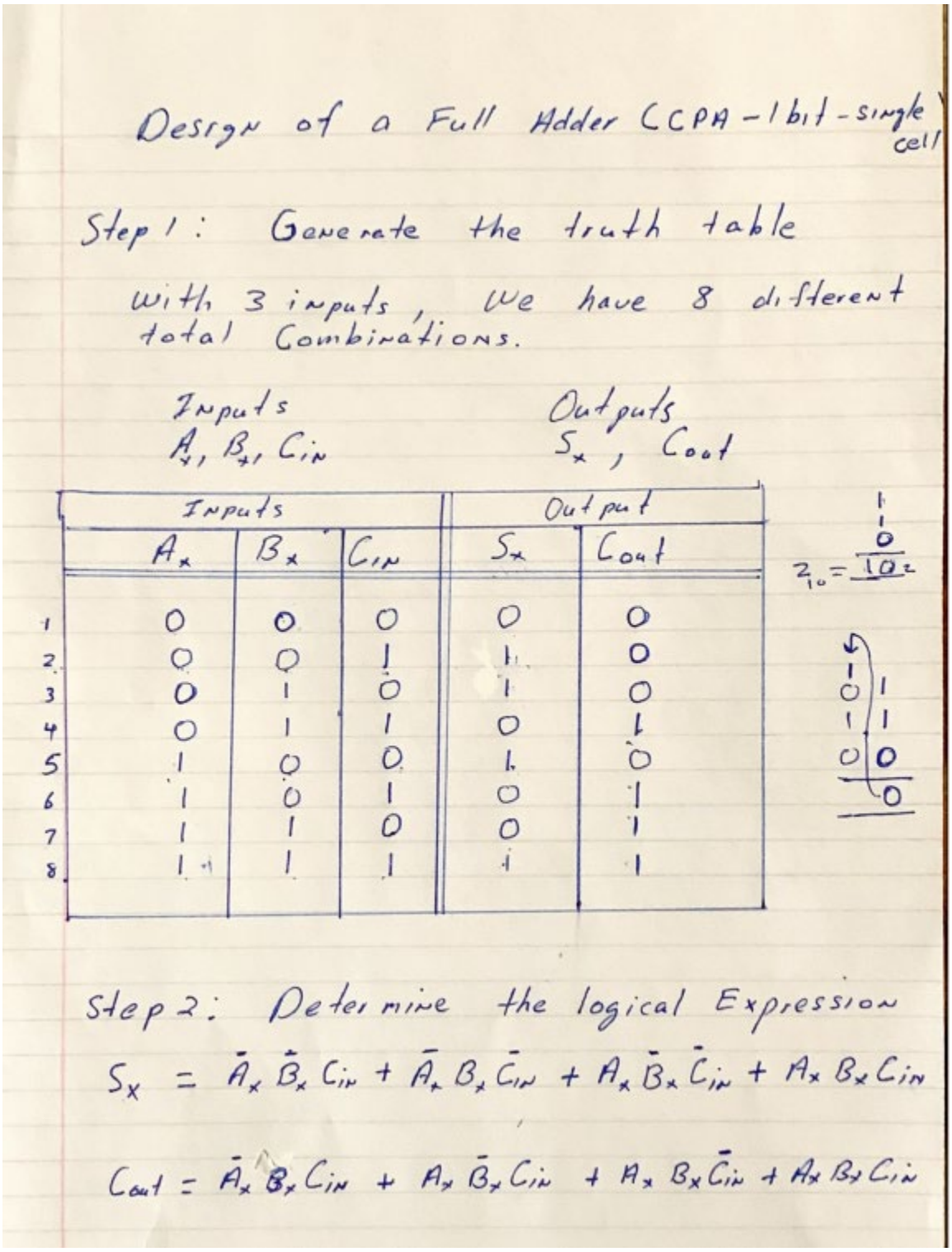
$1_2 + 1_2 = 10_2$
$1_{10} + 1 = 2_{10} = \mathbf{10}_2$
$1_2 + 1_2 + 1_2 = 11_2$

9
+ 1 =
10

A = 0111
B = 1111

Inputs			Output	
A[i}	B[i}	C-in	S	C out
1	1	0	0	1
1	1	1	1	1
1	1	1	1	1
0	1	1	0	1

Design a full adder from ground up…. (Exercise video Note)



Design of a Full Adder (CPA - 1 bit - single cell)

Step 1: Generate the truth table

With 3 inputs, we have 8 different total Combinations.

Inputs A_x, B_x, C_{IN} Outputs S_x, C_{out}

	Inputs			Output	
	A_x	B_x	C_{IN}	S_x	C_{out}
1	0	0	0	0	0
2	0	0	1	1	0
3	0	1	0	1	0
4	0	1	1	0	1
5	1	0	0	1	0
6	1	0	1	0	1
7	1	1	0	0	1
8	1	1	1	1	1

$2_{10} = 10_2$

Step 2: Determine the logical Expression

$$S_x = \bar{A}_x \bar{B}_x C_{IN} + \bar{A}_x B_x \bar{C}_{IN} + A_x \bar{B}_x \bar{C}_{IN} + A_x B_x C_{IN}$$

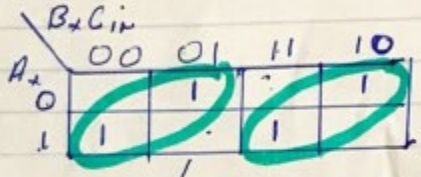
$$C_{out} = \bar{A}_x B_x C_{IN} + A_x \bar{B}_x C_{IN} + A_x B_x \bar{C}_{IN} + A_x B_x C_{IN}$$

Step 3. Simplify Equation or logical Expression:

$$S_x = \bar{A}_x \bar{B}_x C_{IN} + \bar{A}_x B_x \bar{C}_{IN} + A_x \bar{B}_x \bar{C}_{IN} + A_x B_x C_{IN}$$

$$S_x = A_x \oplus B_x \oplus C_{IN}$$

Simplify Using K-Map

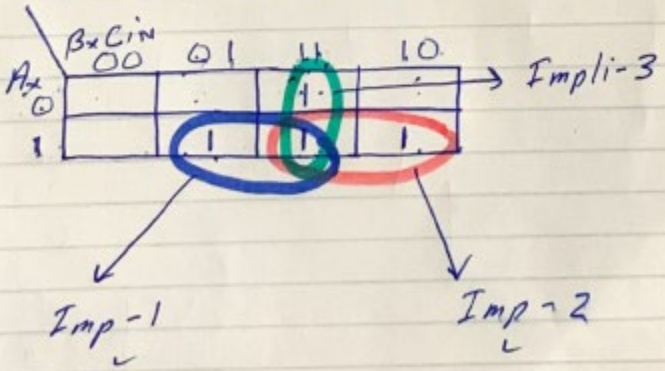


→ EOR

$$S_x = A_x \oplus B_x \oplus C_{IN}$$

$$C_{out} = \bar{A}_x B C_{IN} + A_x \bar{B} C_{IN} + A_x B_x \bar{C}_{IN} + A_x B_x C_{IN}$$

$$= A_x C_{IN} + A_x B_x + B_x C_{IN}$$

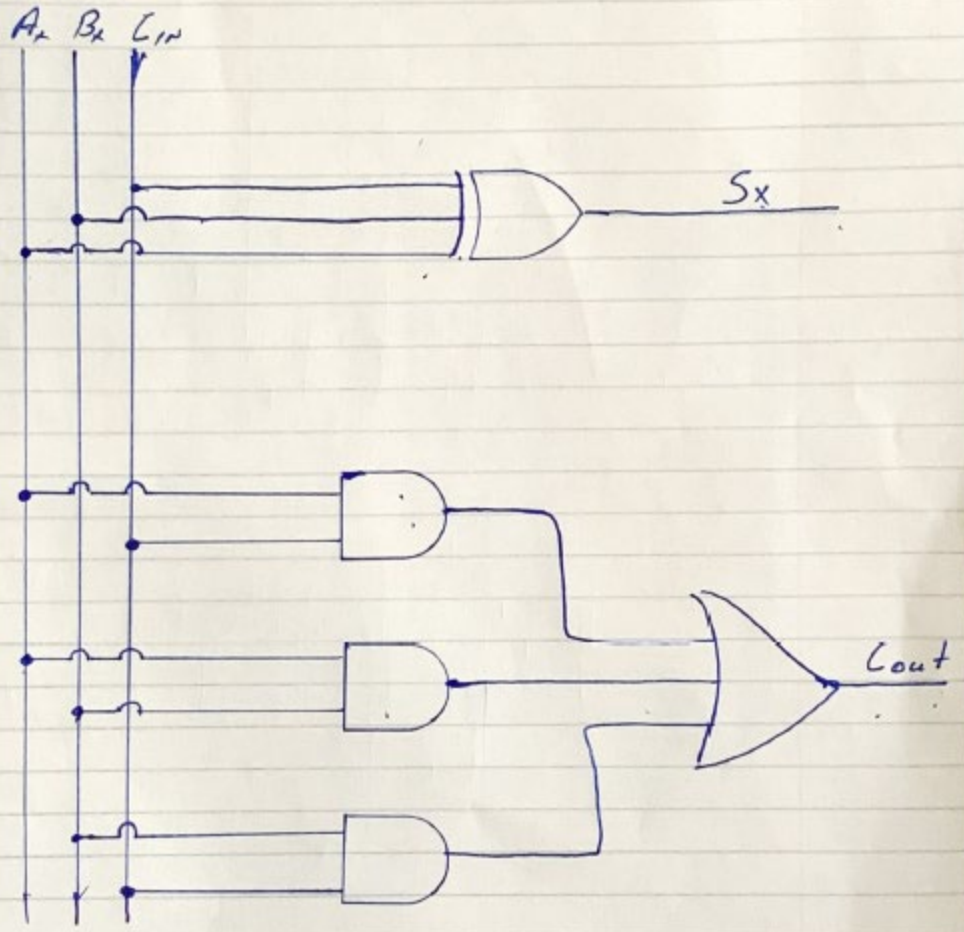


$$C_{out} = A_x C_{IN} + A_x B_x + B_x C_{IN}$$

Step 4: Design of the Circuit

$S_x = A_x \oplus B_x \oplus C_{IN}$ ✓

$C_{out} = A_x C_{IN} + A_x B_x + B_x C_{IN}$



Multiplexer

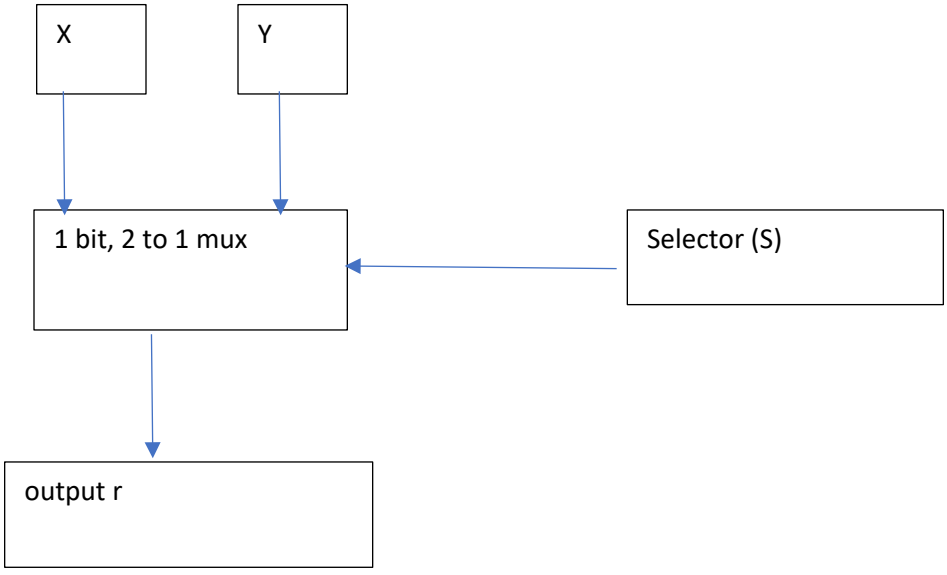
- Selecting of data or information is a critical function in a digital systems and computers
- A Multiplexer (Mux for short) is a digital switch; Mux is a circuit used to select and route any of the several inputs to an output signal
- Mux is a combinational circuit. it has the following:
    - 2^n inputs
    - n control inputs
    - one set of output
    - In summary, has the following 2^n input signals, n control inputs (selector signals), and a single set of output – Output signal.

- For a multiplexer, the value of the control inputs (selector signals) determines the data input that is selected.
- Multiplexer means many into one. a simple example of a non-electronic circuit of a mux is a single pole multi-position switch. Multi-position switches are widely used in many electronics circuit, however, circuits that operate at high speed require the multiplexer to be automatically selected. A mechanical switch cannot perform this task satisfactorily. Therefore, mux is used to perform high speed switching and are constructed for digital circuits

- Example:
    - X and Y are inputs
    - S is the selector signal
    - r is the output

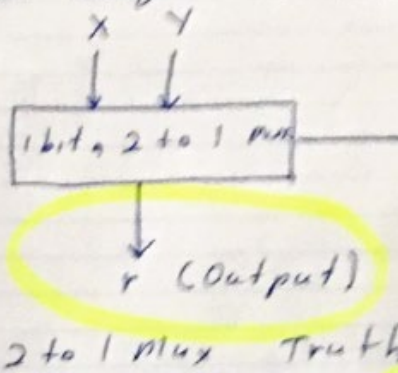
Design of a 1 bit, 2 to 1 mux

1. Block diagram of 1 bit, 2 to 1 mux



Design of 1bit, 2 to 1 Mux

1. Block diagram of 1bit, 2 to 1 Mux



2. 1 bit, 2 to 1 Mux Truth Table

	Ctr	Input Signals		Output
	S	X	Y	r
1	0	0	0	0
2	0	0	1	1
3	0	1	0	0
4	0	1	1	1
5	1	0	0	0
6	1	0	1	0
7	1	1	0	1
8	1	1	1	1

Outputs Y when S=0; Outputs X when S = 1.

$$r = \bar{S}\bar{X}Y + \bar{S}XY + S\bar{X}\bar{Y} + SXY$$

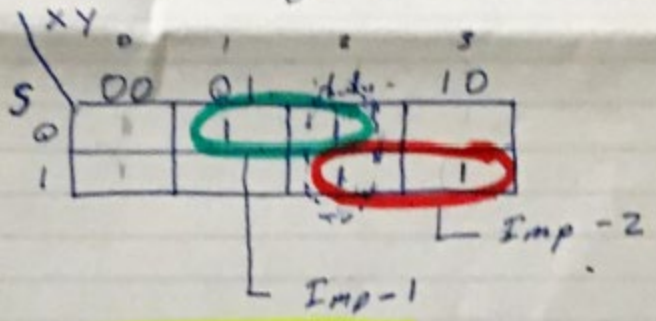
Design of 1 bit, 2 to 1 Mux

3. Simplify the logical Expression:

Minimal SOP Expression: $\bar{S}Y + SX$

$r = \bar{S}\bar{X}Y + \bar{S}XY + SX\bar{Y} + SXY$

Simplify Using K-map:



$r = \bar{S}Y + SX$

Conversion to Floating Point Representation: 5.375_{10}

step 1: Convert decimal to Binary : 5 . 375

- First convert whole number or integer: 5

$$2 \overline{)5} \quad 2 \text{ R } 1$$
$$2 \overline{)2} \quad 1 \text{ R } 0$$
$$2 \overline{)1} \quad 0 \text{ R } 1$$

$5_{10} = 101_2$

- Now we take Fraction piece : $.375_{10}$

$0.375 \times 2 = 0 + 0.75$

$0.75 \times 2 = 1 + 0.5$

$0.5 \times 2 = 1 + .0$ END

$0.375_{10} = 011_2$

$5.375_{10} = 101.011_2$

Step 1: $+5.375_{10} = 101.011_2$

Step 2: Converting Binary to Scientific Notation: $5.375_{10} = 101.011_2$

Convert 631.65_{10}

631.65_{10} → (Scientific Notation) → 6.3165×10^2

Convert 101.011_2

101.011_2 → (Scientific Notation) → 1.01011×2^2

$5.375_{10} = 1.01011_2 \times 2^{0+2}$

Step 2: 1.01011×2^{2}

Step 3: Calculate the Biased Exponent:

Biased offset : 7

Biased Expo = Unbiased Exp + Biased offset

$= 2 + 7$

$= 9_{10}$

$1.01011 \times 2^{2+7} = 1.01011 \times 2^{9}$

Biased Exponent $= 9_{10} = 1001_{2}$

Biased offset

N = # of bits used to store Biased Exponent

$N = 4$

Biased offset $= 2^{N-1} - 1$

$= 2^{4-1} - 1$

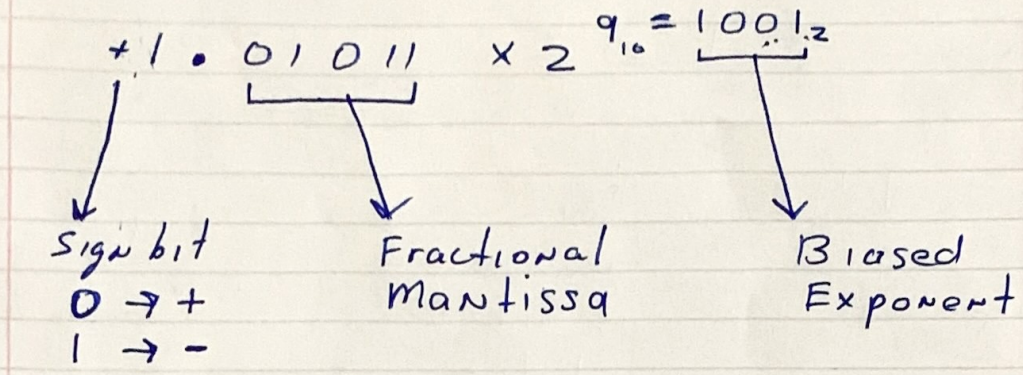
$= 2^{3} - 1$

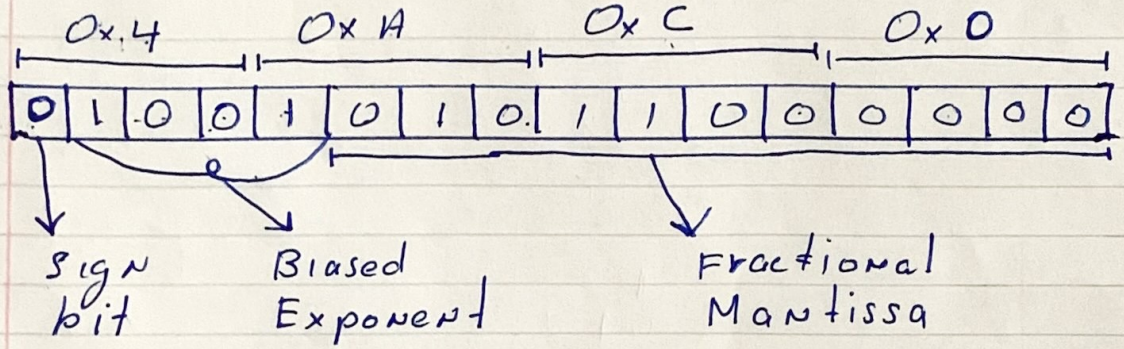
$= 8 - 1$

$= 7$

Step 3 : 1.01011×2^9

Step 4 : Fusion of Floating Point : 1.01011×2^9





Step 5 : Convert to Hex

0x 4AC0

Hexadecimal Numbers

Dec	Binary	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

$0x10 = 16$

$0x10 = 1 \times 16^1 + 0 \times 16^0$

$= 16 + 0 \times 1$

$= 16 + 0$

$= 16$

$= 18$

$= 1 \times 10^1 + 8 \times 10^0$

$= 10 + 8 \times 1$

$= 10 + 8$

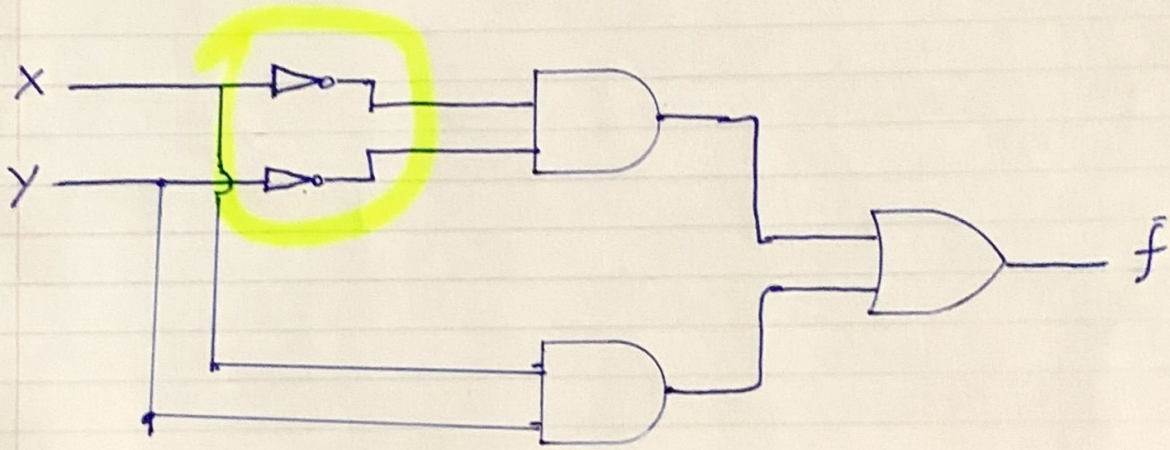
$= 18$

Implementation of SOP Expression with NAND Gates.

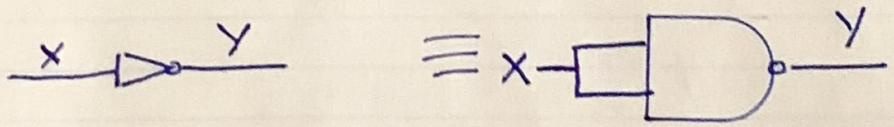
Theorem #1 : $\overline{X \cdot Y} \Leftrightarrow \bar{X} + \bar{Y}$

Theorem #2 : $\overline{X + Y} \Leftrightarrow \bar{X} \cdot \bar{Y}$

$f = \bar{X} \cdot \bar{Y} + X \cdot Y$



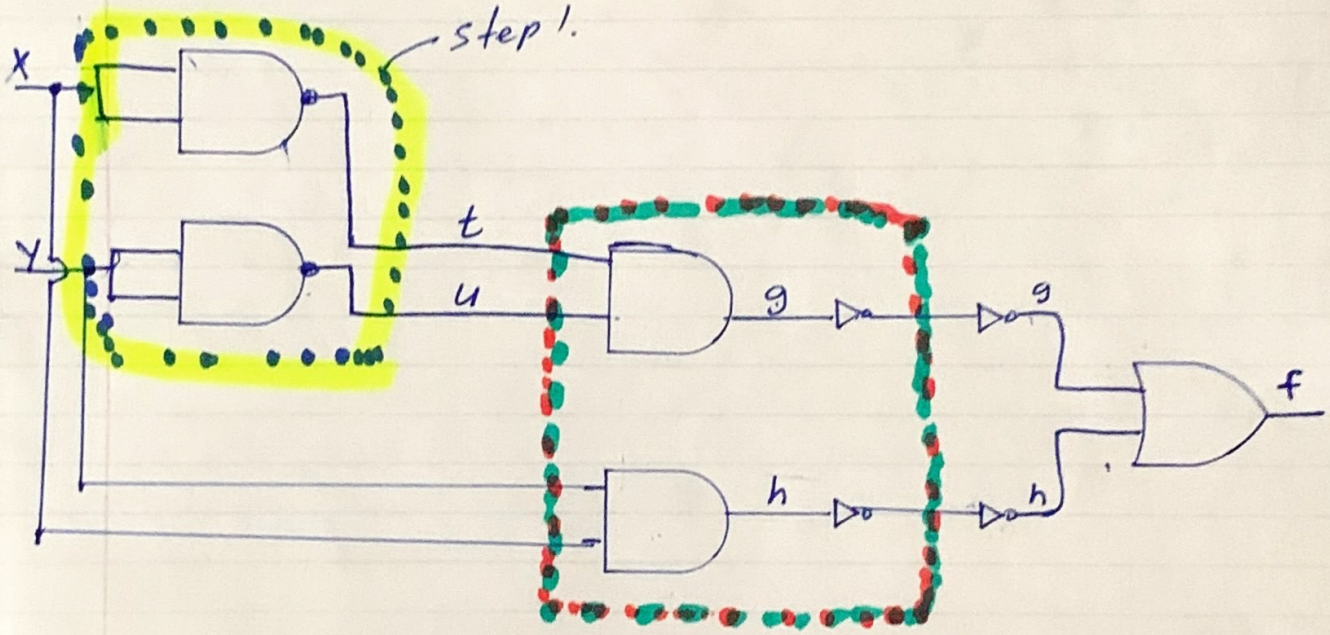
Step 1: Replace each not gate with its equivalent NAND gate by connecting inputs of a two input NAND gates to the single input of the not gate



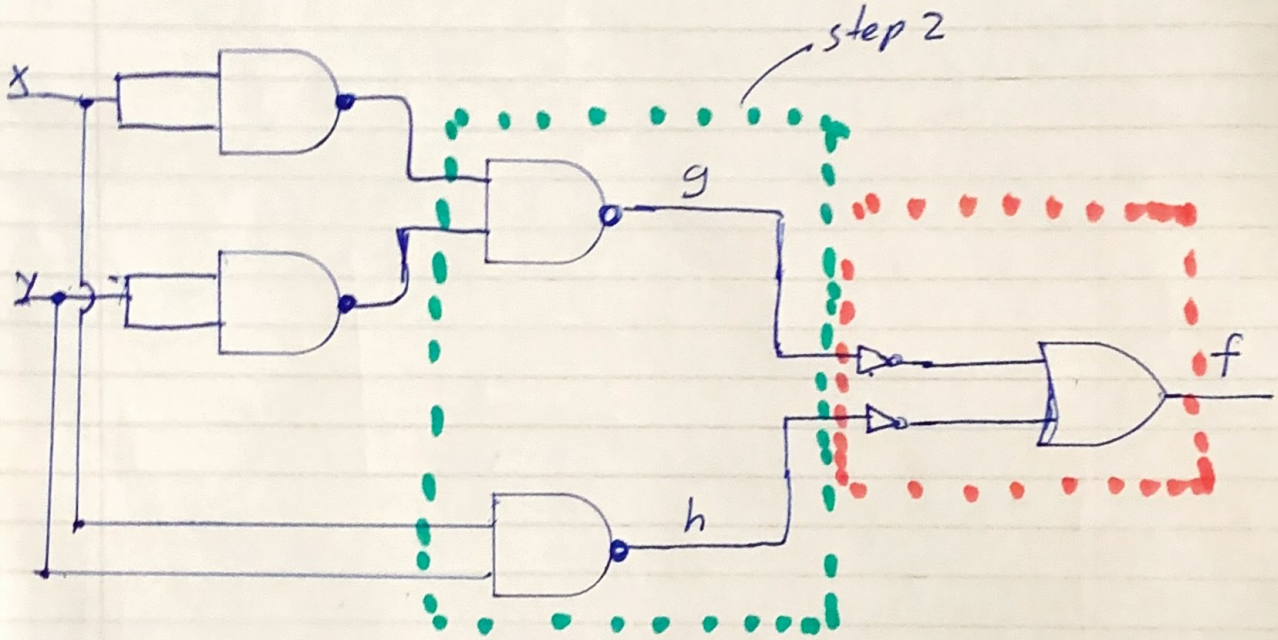
Step 2: Place two NOT gates on each end of the intermediate signals g and h. This will not change the output.

E.G. $\bar{\bar{g}} = g$

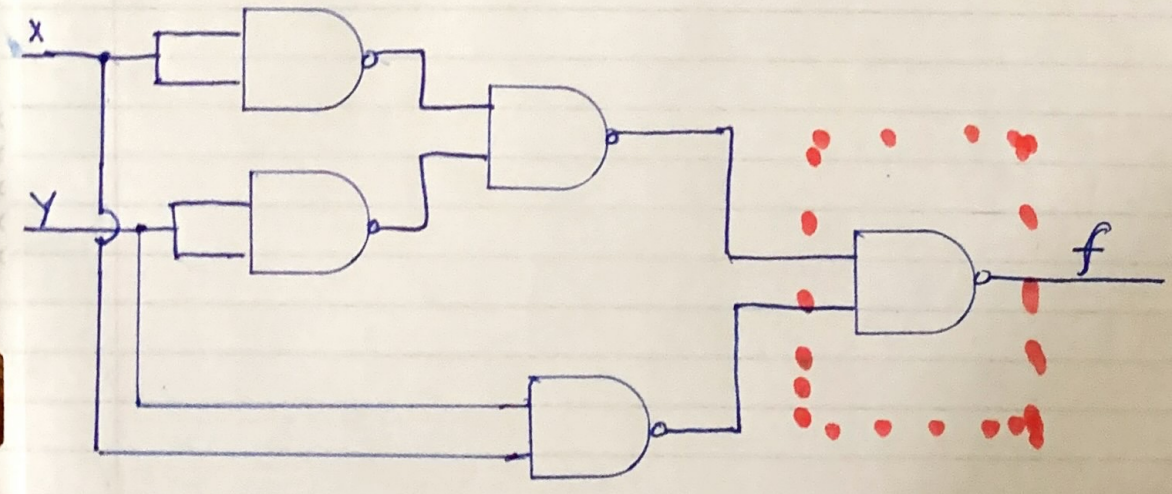
An AND-OR gate with two NOT gate replaced.



Step 3: Replace each AND-NOT gate with NAND gate.



step 4: Replace OR gate with inverted input with NAND gate.



$\bar{X} + \bar{Y} \iff \overline{X \cdot Y}$

Sequential Circuit – Core modules

S-R Latches

- Latches are basic building block of flip-flops (Basic memory Unit).
- Two types of memory elements based on the type of trigger that is suitable to operate.
  -Latches
  -Flip Flops
- Latches operate with enable signal, which is level sensitive, whereas flip-flops are edge sensitive.

D flip-flop

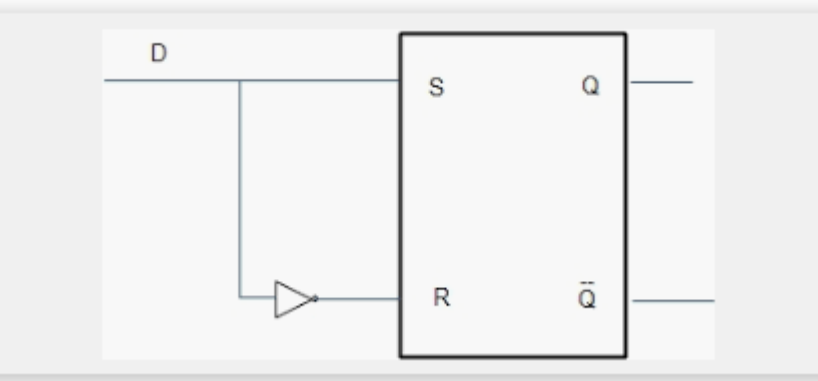
- Basic memory storage device.

-For D flip-flop, a clock signal is needed to change states.

- Similar to S-R latch, only we use set and reset function. (Tie D input to s input and not D to R input to make S_R flip flop into D flip flop.

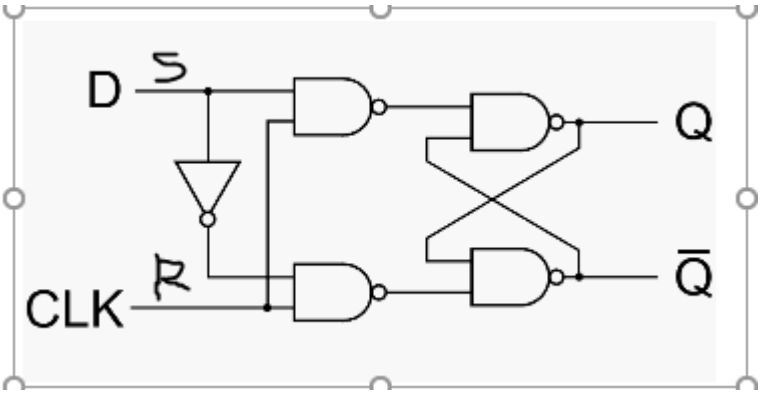
- Block Diagram of D flip flop.

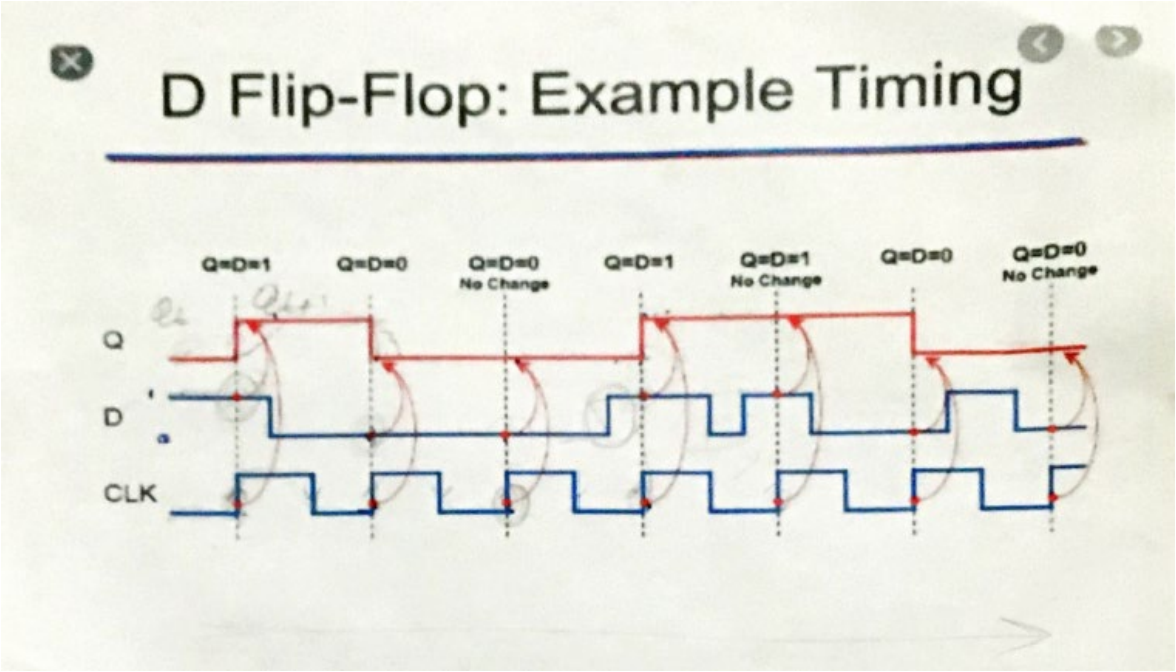
-



Internal circuit of D flip flop:

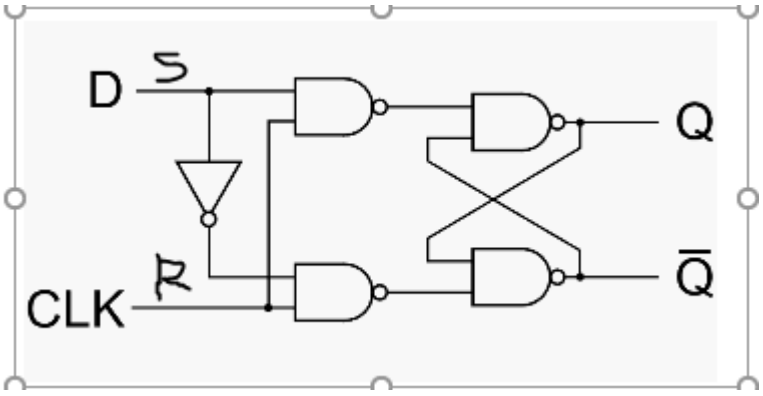
- D flip flop is designed using S-R latches.
- D input goes into S and D goes through inverter and feeds into R.



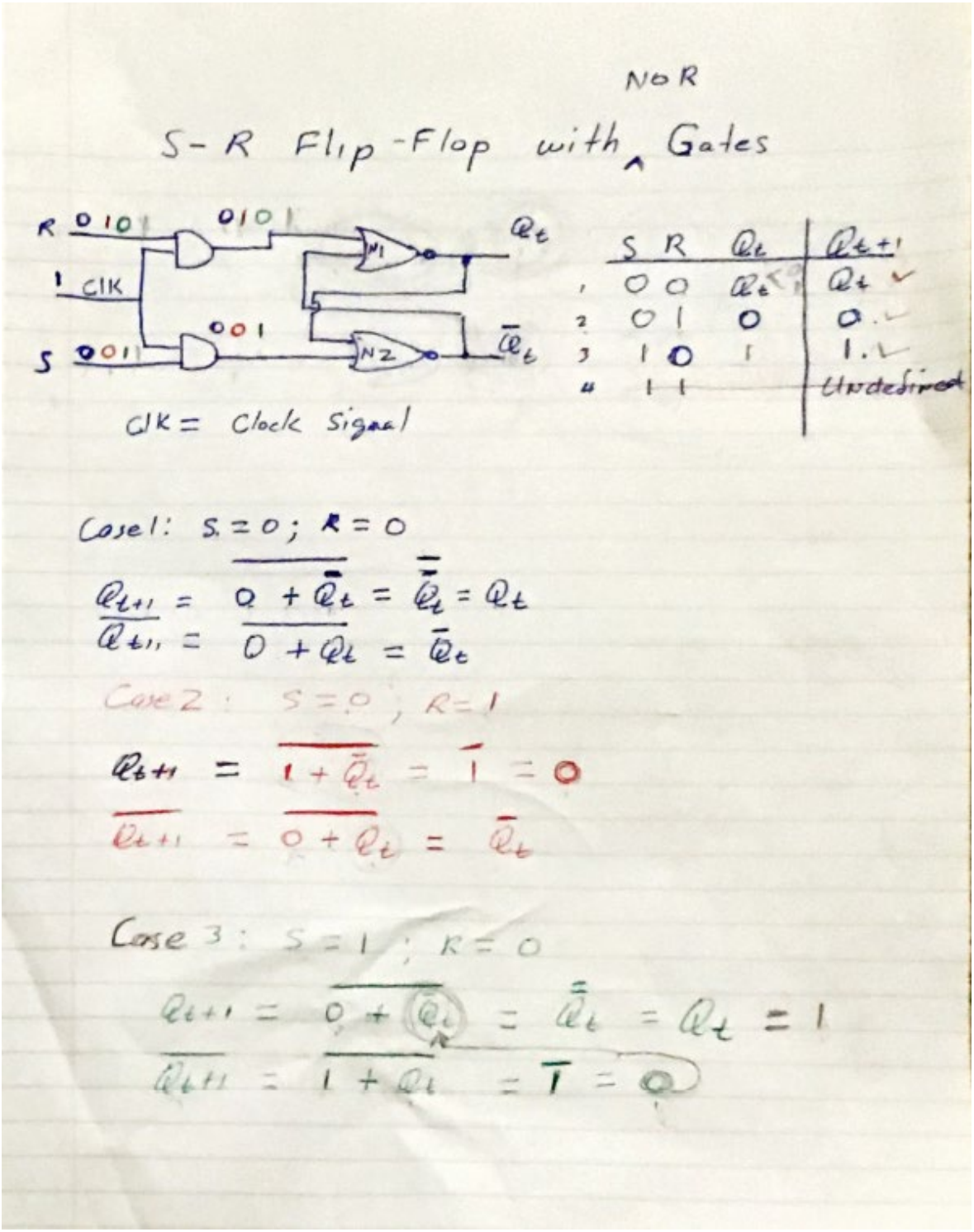


Sequential Circuit: Core Modules (Cont)

Internal circuit of D flip flop:
(D flip flop is designed using S-R latches. D input goes into S and D goes through inverter and feeds into R)



Lets take look at the design of a S-R-flip flop built with NOR gates.



S-R Flip-Flop with NOR Gates (Cont)

Case 4: $S = 1$; $R = 1$

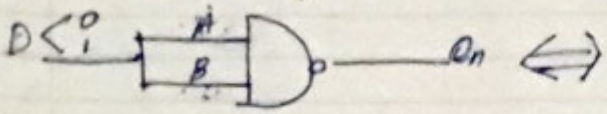
$$Q_{t+1} = \overline{1 + \bar{Q}_t} = \overline{1} = 0 \quad \emptyset$$

$$\overline{Q_{t+1}} = \overline{1 + Q_t} = \overline{1} = 0 \quad \emptyset$$

$\emptyset$ = Not Used!

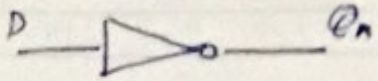
NAND As a Inverter

NAND gate



$\Leftrightarrow$

Inverter



D	A	B	Qn
0	0	0	1
1	1	1	D
			1

Truth Table

D	Qn
0	1
1	0

Sequential Circuit: Small Design Introduction

- All small and large sequential circuits are made of flip-flops and set of CC's – Combinational circuits.
- Contrary to CC's, a sequential circuit design has states and transitions from a current state to next state.
- A sequential circuit design problem is typically modeled as a finite state diagram (FSD).
- FSD consist of circles as states and arcs (arrows) as transitions, which specifies the behavior of sequential circuit.
- An FSD is systematically converted into circuit called a finite state machine. (FSM)
- Finite state machine designs are categorized into Mealy, Moore or hybrid machines.
- Mealy is a FSM whose output values are determined by its ***current state and current input***. Contrast to a Moore machine, whose output values are determined **solely by** its current inputs.
- Finite state diagram (FSD) describes the behavior of system
- A register is used as a storage module to save the output of a CC.
- FSM's are categorized into Mealy, Moore or hybrid machines.

Sequential Circuit: Small Design Introduction – Moore Vs Mealy

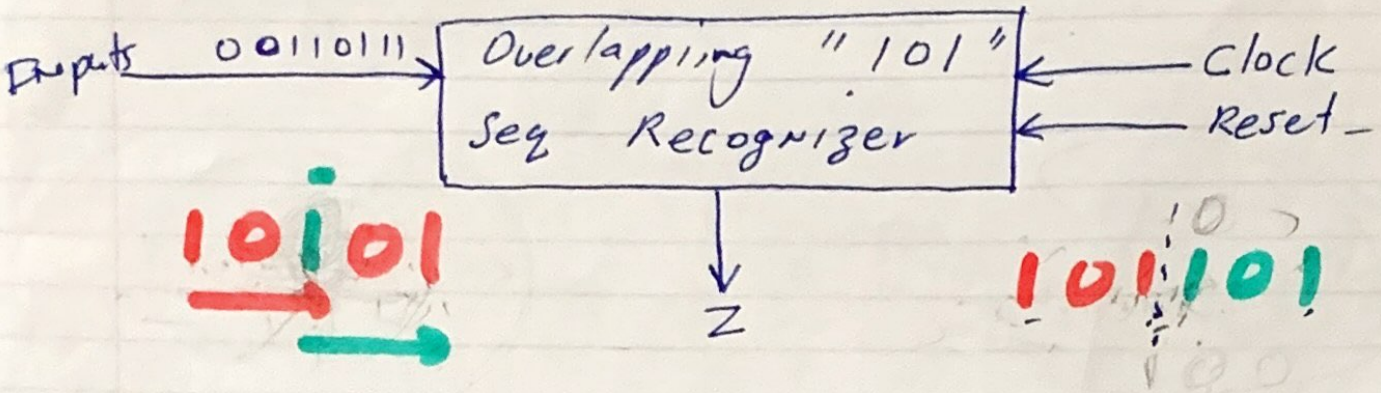
Moore Machines

- Output depends only upon present state
- If input changes, output does not change
- More number of states are required
- There is more hardware requirement
- They react slower to inputs (One clock cycle later)
- Synchronous output and state generation
- Output is placed on states
- Easy to design

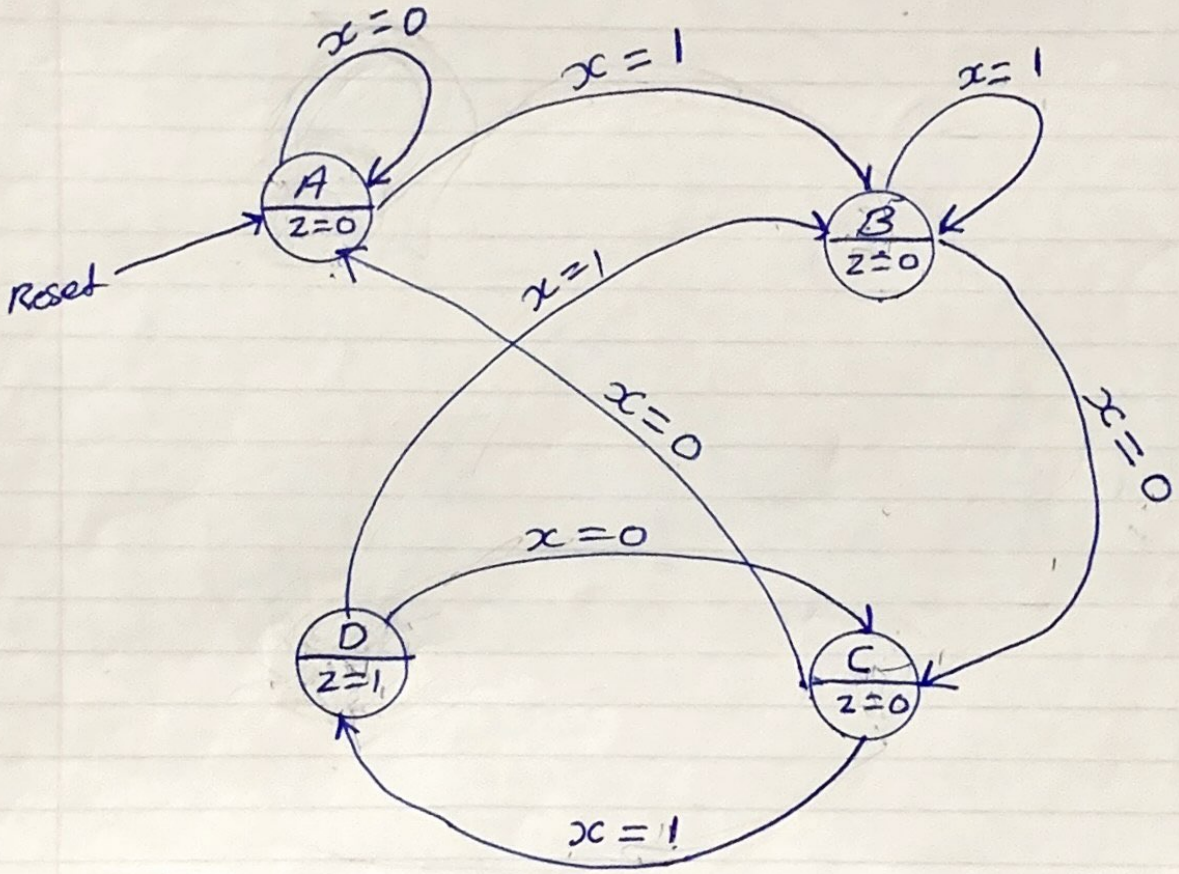
Mealy Machine

- Output depends on present state as well as present input
- If input changes, output also changes
- Less number of states are required
- There is less hardware requirement
- They react faster to inputs
- Asynchronous output generation
- Output is placed on transitions
- It is difficult to design

Example: Design of a Moore FSM that detects overlapping seq "101"



Step 1: Create a Moore FSD - Finite State Diagram



Design of Moore FSM that detects Overlapping Sequence "101" - Cont.

Step 2: Determine the Minimum number of bits required to store the states

Number of bits $= \text{Log}_2[k] = \text{Log}_2[4] = 2$

k = # of states

Step 3: From the FSD, create the Truth table for, NSG & OG

A = 00, B = 01

C = 10 D = 11

NSG

	Current state		Input X	Next state		
	q_1	q_0	X	d_1	d_0	
1	A	0	0	0	0	A
2		0	1	0	1	B
3	B	0	0	1	0	C
4		0	1	0	1	B
5	C	1	0	0	0	A
6		1	1	1	1	D
7	D	1	0	1	0	C
8		1	1	0	1	B

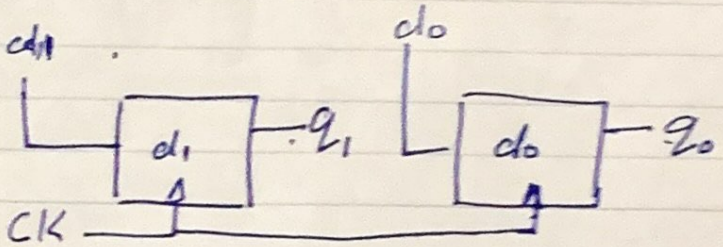
Step 3 - Cont

Design of a Moore FSM that detects Overlapping Sequence "101"

Step 3: Create Output Generator (OG)

OG

	Current state q_1	q_0		output Z
1	0	0	A	0
2	0	1	B	0
3	1	0	C	0
4	1	1	D	1



clk = Clock signal

Design of a Moore FSM that detects Overlapping Sequence "101"

Step 4: From the truth table, Determine Min SOP for each of the states - var d_1 d_0, output Z

$$d_1 = \bar{x}\,\bar{q}_1 q_0 + x\,q_1\bar{q}_0 + \bar{x}\,q_1 q_0$$

$$= \bar{x}\,\bar{q}_1 q_0 + \bar{x}\,q_1\bar{q}_0 + x\,q_1\bar{q}_0$$

$$= \bar{x}\,(\bar{q}_1 q_0 + q_1 q_0) + x\,q_1\bar{q}_0$$

$q_0 = 0$, q_0

$q_0 = 1$

q_0

Complement Law

$q_0(\bar{q}_1 + q_1)$

1

$q_0 \cdot 1$

q_0

$$= \bar{x}\,q_0 + x\,q_1\bar{q}_0$$

$$d_1 \equiv \bar{x}\,q_0 + x\,q_0\bar{q}_0$$

$d_0 = [x]$; $Z = q_1 q_0$ ✓

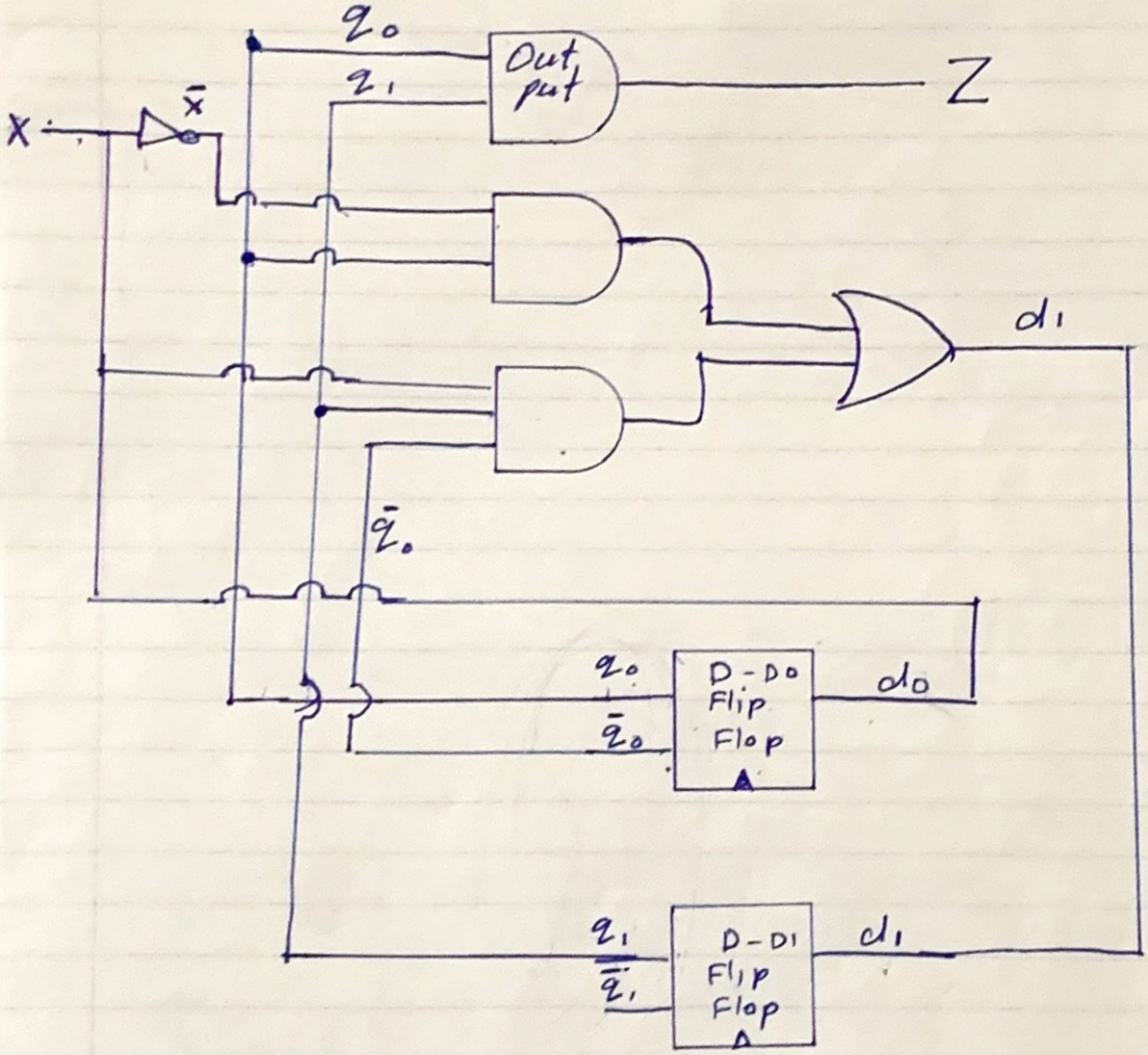
$\bar{q}_1 q_0 + q_1 q_0 \Longleftrightarrow q_0$

$q_0 = 0 \Rightarrow$ whole expression $= 0$

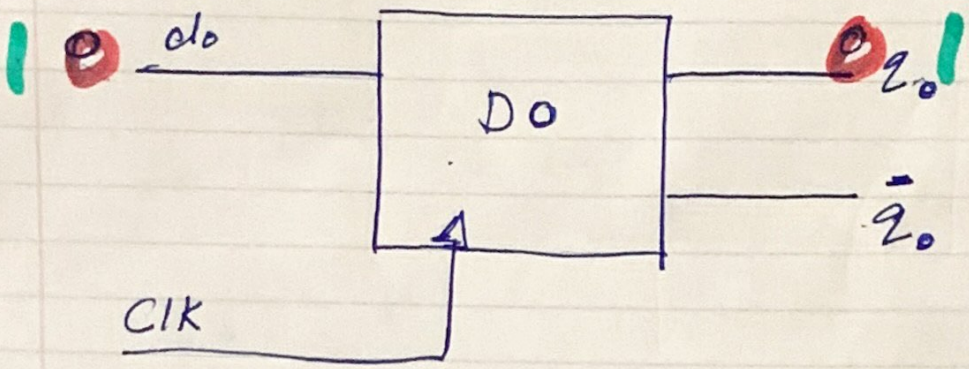
$\bar{q}_1 \cdot 0 + q_1 \cdot 0 =$

Design of a Moore FSM that Detects "101" Sequence

Step 5: Built or design the circuit that detects sequence "101"



D flip Flops



CLK = Clock

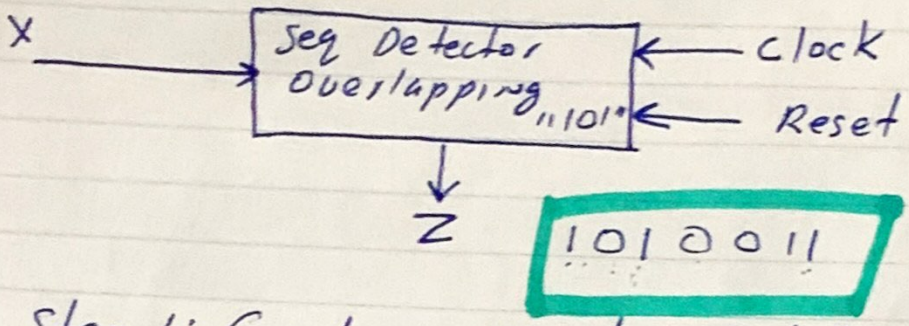
d_0 = Input (1 bit)

q_0 = Output

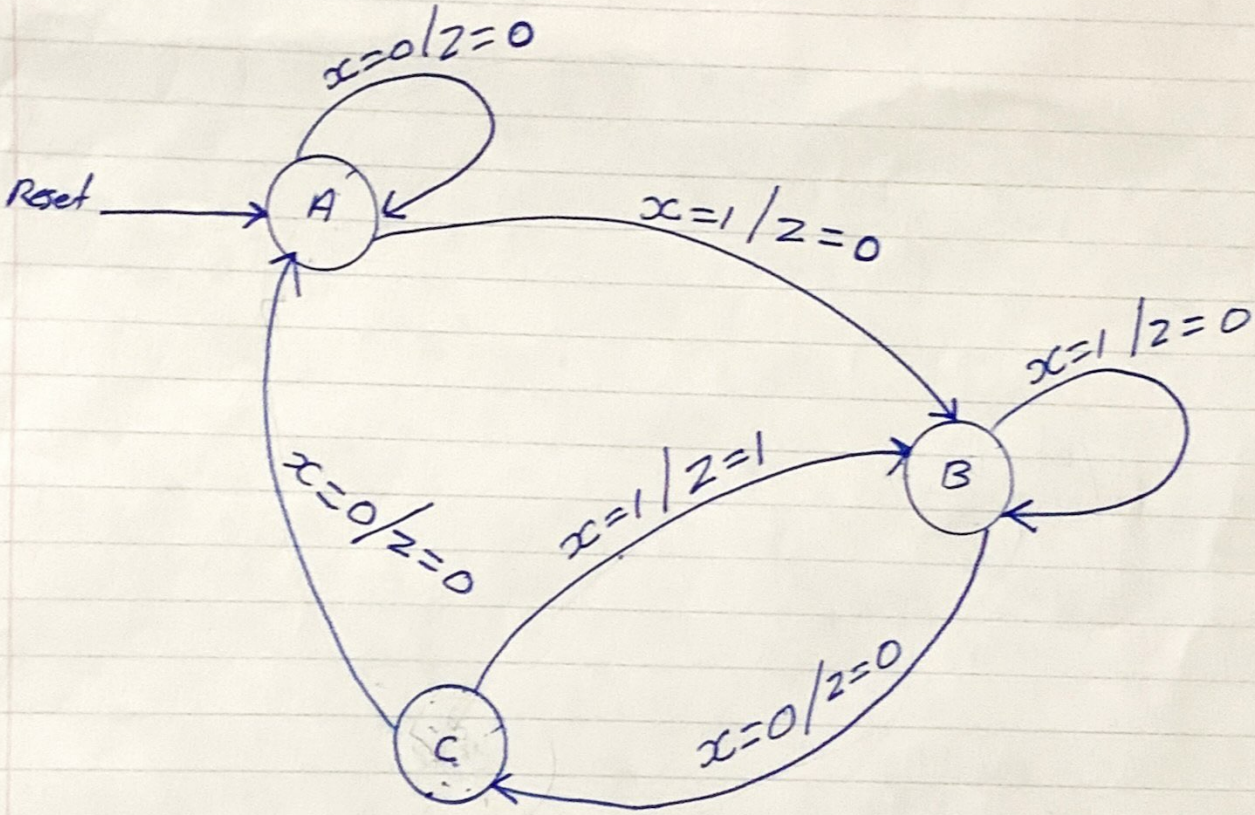
$\bar{q}_0$ = Inverted Output

Design of Mealy FSM - Overlapping
Sequence "101"

* Mealy output is assigned to the arcs and Not to the states



Step 1: Create a Mealy Finite state Diagram



Design of Mealy FSM - Overlapping Sequence "101"

Step 2: Determine the Min number of states / bits required to store the states

$$\text{Number of bits} = \log_2[K] = \log_2[3] \cong 2$$

K = Total # of states

Step 3: From FSD, Create the truth table.
Let 00 = A, 01 = B, 10 = C, 11 = D.

NSG / OG

	Current States		Input	Next state			Z
	q_1	q_0	X	d_1	d_0		
1	A	0	0	0	0	A	0
2		0	1	0	1	B	0
3	B	0	0	1	0	C	0
4		0	1	0	1	B	0
5	C	1	0	0	0	A	0
6		1	1	0	1	B	1
7	D	1	1	d	d		d
8		1	1	d	d		d

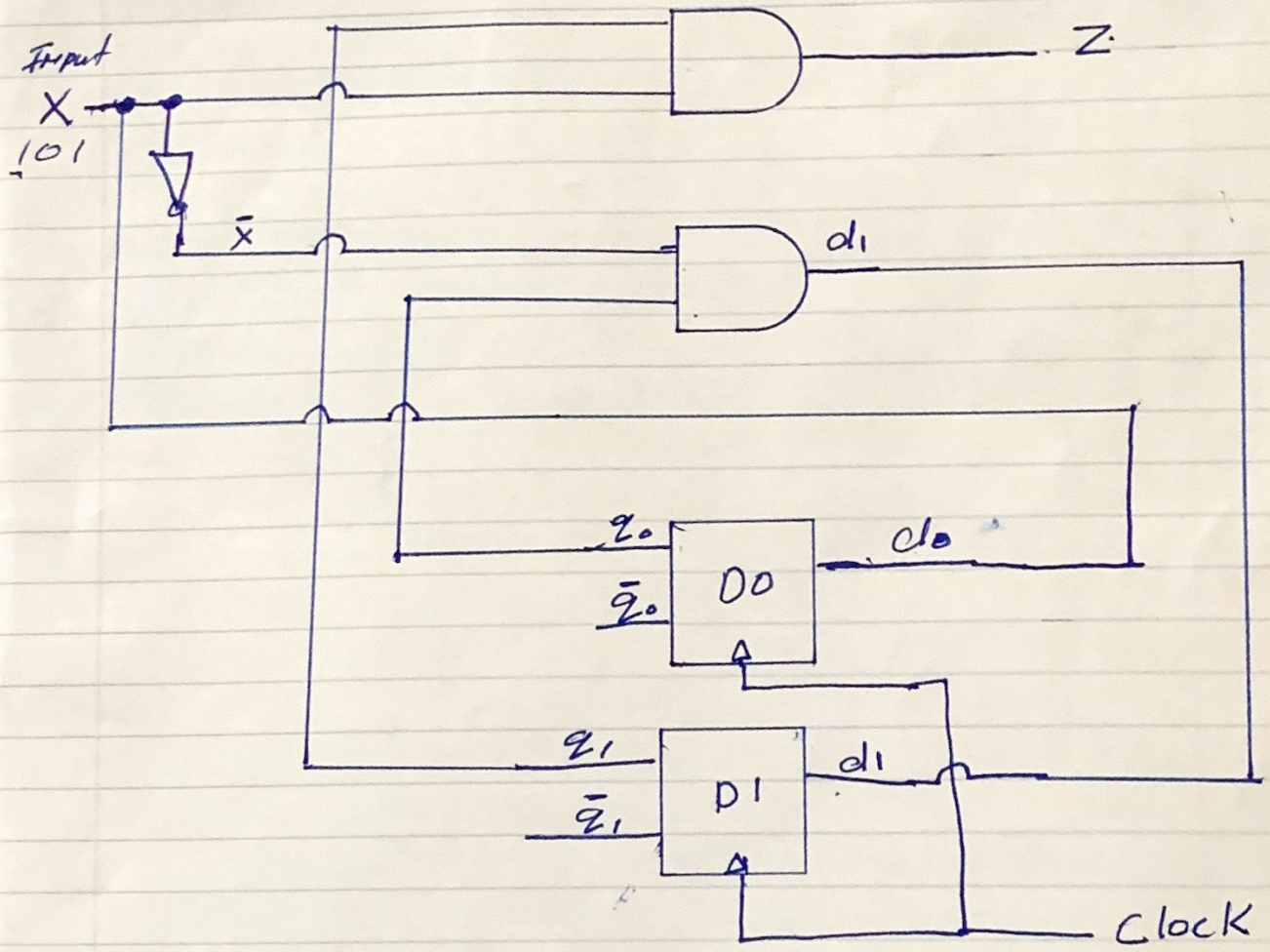
Design of Mealy FSM - Overlapping Sequence "101"

Step 4: Determine the logical Expression

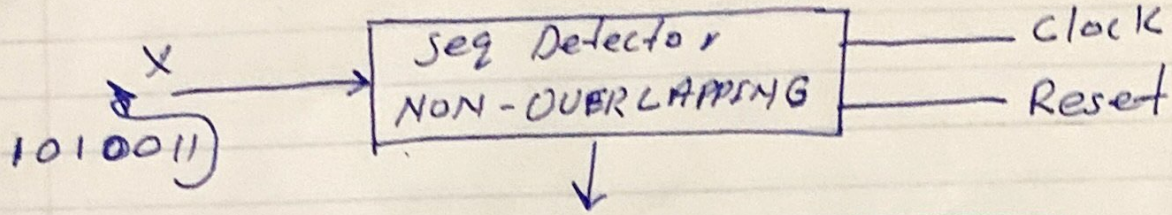
$d_1 = q_0 \bar{x}$; $d_1 = q_0 \bar{q}_1 \bar{x}$

$d_0 = x$ $z = q_1 x$

Step 5: Draw the Circuit Diagram



Example: Design of Mealy Finite State machine that detects NON-Overlapping Sequence "1011"



10101

1010011

Step 1: Create a mealy FSD - Finite state Diagram.

Let A = 00, B = 01, C = 10, D = 11