CS 251: Computer Organization and Design

Charles Shen

Spring 2016, University of Waterloo

Notes written from Safaa Bedawi's lectures.

Feel free to email feedback to me at echen902@gmail.com.

Contents

1 From Zero To One, Abstractions **5**
1.1 The Three -Y's 5
1.2 Hexadecimal Number System 6
1.3 Bytes, Nibbles, and All That Jazz 6
1.4 Logic Gates 7
1.5 Beneath the Digital Abstraction 7
1.5.1 Supply Voltage 8
1.5.2 Logic Levels 8
1.5.3 Noise Margins 8
1.5.4 DC Transfer Characteristics 8

2 CMOS Transistors **8**
2.1 Semiconductors 9
2.2 Diodes 9
2.3 Capacitors 10
2.4 nMOS and pMOS Transistors 10
2.5 Transmission Gates 11
2.6 Pseudo-nMOS Logic 12

3 Summary So Far **12**

4 Introduction to Combinational Logic Design **13**

5 Boolean Equations **15**
5.1 Terminology 16
5.2 Sum-of-Products Form 16

6 Boolean Algebra **17**
6.1 Axioms 17
6.2 Theorems of One Variable 17
6.3 Theorems of Several Variables 17

7 From Logic to Gates **18**

8 Multilevel Combinational Logic **18**
8.1 Hardware Reduction 18
8.2 Bubble Pushing 19

9 Combinational Building Blocks **19**
9.1 Multiplexers 19
9.1.1 2:1 Multiplexer 19
9.1.2 Wider Multiplexers 20
9.1.3 Multiplexer Logic 20
9.2 Decoder 20
9.2.1 Decoder Logic 21

10 Summary So Far **21**

11 Introduction to Sequential Logic Design **22**

12 Latches and Flip-Flops **23**
12.1 SR Latch 24
12.2 D Latch 25
12.3 D Flip-Flop 26
12.4 Register 26
12.4.1 Reading from a Register File 27
12.4.2 Writing to a Register File 27

13 Finite State Machines **27**
13.1 State Transition Diagram 28
13.2 State Transition Table 28
13.3 State Encoding 29
13.4 FSM Review 30

14 Summary So Far **30**

15 Memory Technologies **31**
15.1 SRAM Technology 32
15.2 DRAM Technology 32

16 Multiplication **33**

17 Floating Point **33**
17.1 Floating-Point Representation 34
17.2 Floating-Point Addition 36
17.3 Floating-Point Multiplication 36

18 Introduction to the Processor **36**

19 Building a Data-path **38**

19.1 R-Format ALU Operations 39

19.2 Memory-Reference Operations 40

19.3 Branch Operations 41

1 From Zero To One, Abstractions

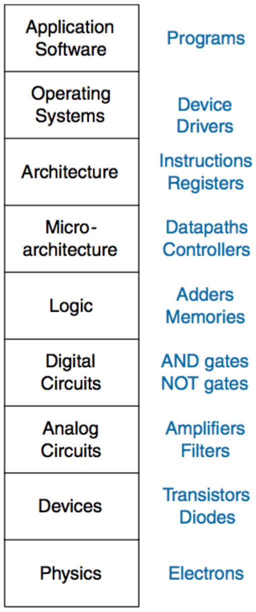


Figure 1: Levels of abstraction for electronic computing system

1.1 The Three -Y's

Hierarchy involves dividing a system into modules, then further subdividing each of these modules until the pieces are easy to understand

Modularity states that the modules have well-defined functions and interfaces, so that they connect together easily without unanticipated side effects

Regularity seeks uniformity among the modules. Common modules are reused many times, reducing the number of distinct modules that must be designed

1.2 Hexadecimal Number System

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

1.3 Bytes, Nibbles, and All That Jazz

A group of eight bits is called a byte. Represents $256 = 2^8$ possibilities.
A group of four bits is called a nibble. Represents $16 = 2^4$ possibilities.
A microprocessor is a processor built on a single chip.
Microprocessors handle data in chunks called words. The size of a word depends on the architecture of the microprocessor.
Within a group of bits, the bit in the 1's column is called the least significant bit (lsb), and the bit at the other end is called the most significant bit (msb).
Within a word, the bytes are identified as least significant byte (LSB) through most significant byte (MSB).

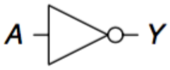
When adding operands with different signs, overflow cannot occur because the sum must be no larger than one of the operands.

1.4 Logic Gates

A circuit element that performs a basic logic function

Single Input

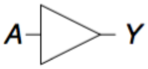
NOT



$Y=\overline{A}$

A	Y
0	1
1	0

BUF

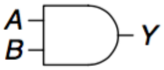


$Y=A$

A	Y
0	0
1	1

Double Output

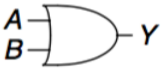
AND



$Y=AB$

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

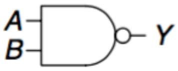
OR



$Y=A+B$

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

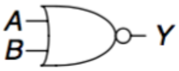
NAND



$Y=\overline{AB}$

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

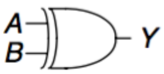
NOR



$Y=\overline{A+B}$

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

XOR



$Y=A\oplus B$

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

1.5 Beneath the Digital Abstraction

A digital system uses discrete-valued variables.
Variables are represented by continuous physical quantities such as the voltage on a wire, the position of a gear, or the level of fluid in a cylinder.

For example:
Consider representing a binary signal A with a voltage on a wire.
Let 0 volts (V) indicate A = 0 and 5V indicate A = 1.
Any real system must tolerate some noise, so 4.97V probably ought to be interpreted as A = 1 as well.
But what about 4.3V? Or 2.8V? Or 2.5000000V?

1.5.1 Supply Voltage

Suppose the lowest voltage in the system is 0V, also called ground or GND
The highest voltage in the system comes from the power supply and is usually called V_{DD}

1.5.2 Logic Levels

Mapping of a continuous variable onto a discrete binary variable is done by defining logic levels. First gate is called the driver and the second gate is called the receiver. Output of the driver is connected to the input of the receiver.
The driver produces a LOW (0) output in the range of 0 to V_{OL} or a HIGH (1) output in the range of V_{OH} to V_{DD}

1.5.3 Noise Margins

If the output of the driver is to be correctly interpreted at the input of the receiver, we must choose $V_{OL} < V_{IL}$ and $V_{OH} > V_{IH}$
Even if the output of the driver is contaminated by some noise, the input of the receiver will still detect the correct logic level. Noise margin is the amount of noise that could be added to a worst-case output such that the signal can still be interpreted as a valid input.

1.5.4 DC Transfer Characteristics

The DC transfer characteristics of a gate describe the output voltage as a function of the input voltage when the input is changed slowly enough that the output can keep up.
Called transfer characteristics because they describe the relationship between input and output voltages.

2 CMOS Transistors

Transistors are electrically controlled switches that turn ON or OFF when a voltage or current is applied to a control terminal.
Two main types of transistors are bipolar transistors and metal-oxide-semiconductor field effect transistors (MOSFETs or MOS transistors).

2.1 Semiconductors

MOS transistors are built from silicon.

Silicon has four electrons in its valence shell and forms bonds with four adjacent atoms, resulting in a crystalline lattice.

By itself, silicon is a poor conductor due to all the electrons are tied up in covalent bonds.

Becomes a better conductor when small amounts of impurities, called dopant atoms, are added.

The conductivity of silicon changes over many orders and of magnitude depending on the concentration of dopants, silicon is called a semiconductor.

Add Arsenic

If arsenic (As), a group V dopant, is added, the dopant atoms have an extra electron that is not involved in the bonds.

Electron can easily move about the lattice, leaving an ionized dopant atom (As+) behind.

Electron carries a negative charge, so arsenic is an n-type dopant.

Add Boron

If boron (B), a group III dopant, is added, the dopant atoms are missing an electron.

The missing electron is called a hole.

An electron from a neighboring silicon atom may move over to fill the missing bond, forming an ionized dopant atom (B-) and leaving a hole at the neighboring silicon atom.

The hole can migrate around the lattice.

Hole is a lack of negative charge, so it acts like a positively charged particle.

So boron is a p-type dopant.

2.2 Diodes

Junction between p-type and n-type silicon is called a diode. The p-type region is called the anode and the n-type region is called the cathode.

When the voltage on the anode rises above the voltage on the cathode, the diode is forward biased, and current flows through the diode from the anode to the cathode.

When the anode voltage is lower than the voltage on the cathode, the diode is reverse biased, and no current flows.

The diode symbol intuitively shows that current only flows in one direction.

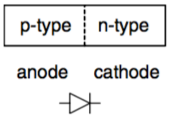
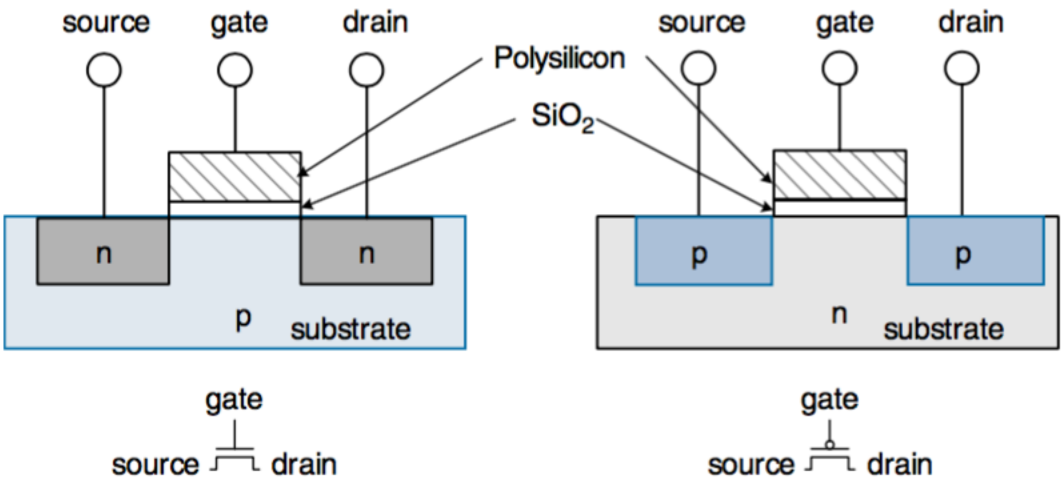


Figure 1.27 The p-n junction diode structure and symbol

2.3 Capacitors

A *capacitor* consists of two conductors separated by an insulator.
When a voltage V is applied to one of the conductors, the conductor accumulates electric *charge* Q and the other conductor accumulates the opposite charge $-Q$.
The *capacitance* C of the capacitor is the ratio of charge to voltage: $C = \frac{Q}{V}$.
The capacitance is proportional to the size of the conductors and inversely proportional the distance between them.
Capacitance is important because charing or discharging a conductor takes time and energy. More capacitance means that a circuit will be slower and require more energy to operate.

2.4 nMOS and pMOS Transistors



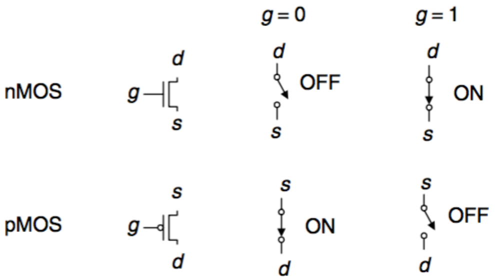
(a) nMOS

(b) pMOS

Figure 1.29 nMOS and pMOS transistors

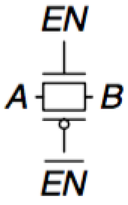
A MOSFET is a sandwich of several layers of conducting and insulating materials.
There are two flavors of MOSFETs: nMOS and pMOS.
The n-type transistors, called *nMOS*, have regions of n-type dopants adjacent tot he gate called the *source* and the *drain* and are built on a p-type semiconductor substrate.
The *pMOS* transistors are just the opposite, consisting of p-type source and

drain regions in an n-type *substrate*.
A MOSFET behaves as a voltage-controlled switch in which the gate voltage creates an electric field that turns ON or OFF a connection between the source and drain.
nMOS transistors pass 0's well but passes 1's poorly. Similarly, pMOS transistors pass 1's well but 0's poorly.
nMOS transistors need a p-type substrate, and pMOS transistors need an n-type substrate. To build both flavors of transistors on the same chip, manufacturing processes typically start with a p-type wafer, then implant n-type region called wells where the pMOS transistors should go. These processes that provide both flavors of transistors are called Complementary MOS or CMOS.
CMOS processes are used to build the vast majority of all transistors fabricated today.
CMOS processes provide two types of electrically controlled switches. nMOS transistors are OFF when the gate is 0 and ON when the gate is 1. pMOS transistors are just the opposite: ON when the gate is 0 and OFF when the gate is 1.



2.5 Transmission Gates

At times, designers find it convenient to use an ideal switch that can pass both 0 and 1 well.
nMOS transistors are good at passing 0 and pMOS transistors are good at passing 1, so the parallel combination of the two passes both values well.
Image to the right shows such a circuit, called *transmission gate* or *pass gate*.



The two sides of the switch are called A and B because a switch is bidirectional and has no preferred input or output side. The control signals are called *enables*, EN and $\overline{EN}$.
When $EN = 0$ and $\overline{EN} = 1$, both transistors are OFF. Hence, the transmission gate is OFF or disabled, so A and B are not connected.
When $EN = 1$ and $\overline{EN} = 0$, the transmission is ON or enabled, and any logic value can flow between A and B.

2.6 Pseudo-nMOS Logic

An N-input CMOS NOR gate uses N nMOS transistors in parallel and N pMOS transistors in series. Transistors in series are slower than transistors in parallel, just as resistors in series have more resistance than resistors in parallel.
Moreover, pMOS transistors are slower than nMOS transistors because holes cannot move around the silicon lattice as fast as electrons. Therefore parallel nMOS transistors are fast and the series pMOS transistors are slow, especially when many are in series.

Pseudo-nMOS logic replaces the slow stack of pMOS transistors with a single weak pMOS transistor that is always ON. This pMOS transistors is often called a *weak pull-up*. The physical dimensions of the pMOS transistor are selected so that the pMOS transistor will pull the output, Y, HIGH weakly—that is, only if none of the nMOS transistors are ON. But if any nMOS transistor is ON, it overpowers the weak pull-up and pulls Y down close enough to GND to produce a logic 0.

The advantage of pseudo-nMOS logic is that it can be used to build fast NOR gates with many inputs.
The disadvantage is that a short circuit exists between V_{DD} and GND when the output is LOW; the weak pMOS and nMOS transistors are both ON. The short circuit draws continuous power, so pseudo-nMOS logic must be used sparingly.

3 Summary So Far

There are 10 kinds of people in this world: those who can count in binary and those who can't.
The real world is analog, though digital designers discipline themselves to use a discrete subset of possible signals. In particular, binary variables have just two states: 0 and 1, also called FALSE and TRUE or LOW and HIGH.

Logic gates compute a binary output from one or more binary inputs. Some of the common logic gates are:

NOT: TRUE when all input is FALSE

AND: TRUE when all input are TRUE

OR: TRUE when any inputs are TRUE

XOR: TRUE when an odd number of inputs are TRUE

Logic gates are commonly built from CMOS transistors, which behave as electrically controlled switches.
nMOS transistors turn ON when the gate is 1.
pMOS transistors turn ON when the gate is 0.

4 Introduction to Combinational Logic Design

In digital electronics, a *circuit* is a network that processes discrete-valued variables.
A circuit can be viewed as a black box, with

- one or more discrete-valued *input terminals*
- one or more discrete-valued *output terminals*
- a *functional specification* describing the relationship between inputs and outputs
- a *timing specification* describing the delay between inputs changing and output responding

Peeing inside the black box, circuits are composed of nodes and elements.
An *element* is itself a circuit with inputs, outputs, and a specification.
A *node* is a wire, whose voltage coveys a discrete-valued variable. Nodes are classified as *input, output,* or *internal.*
Input receive values from the external world.
Output deliver values to the external world.
Wires that are not inputs or outputs are called internal nodes.

Digital circuits are classified as *combinational* or *sequential.*
A combinational circuit's outputs depend only on the current values of the inputs; in other words, it combines the current input values to compute the output. For example, a logic gate is a combinational circuit.
A sequential circuit's outputs depend on both current and previous values of the inputs; in other words, it depends on the input sequence.
A combinational circuit is *memoryless*, but a sequential circuit has *memory.*

The functional specification of a combination circuit expresses the output values in terms of the current input values.
The timing specification of a combinational circuit consists of lower and upper bounds on the delay from input to put.

To simply drawings, a single line with a slash through it and a number next to it is often used to indicate a *bus*, a bundle of of multiple signals.

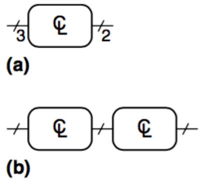


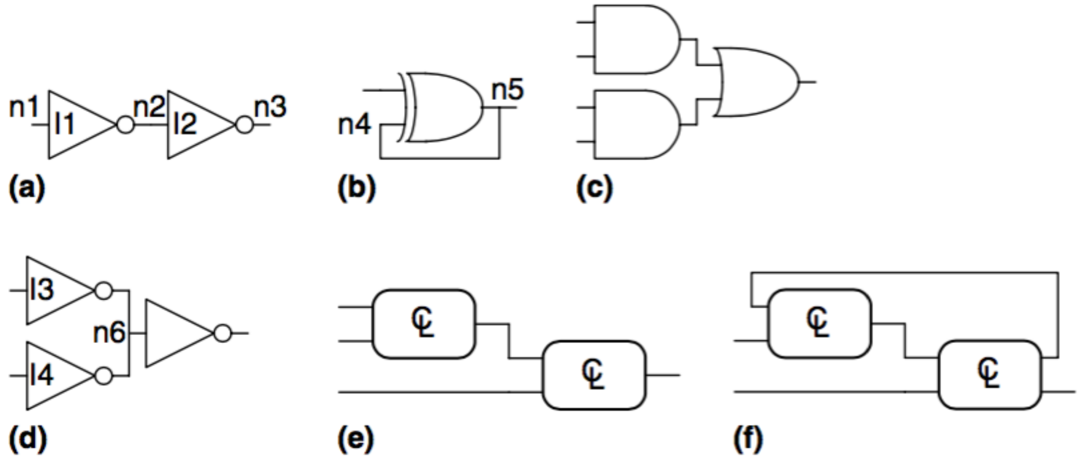
Figure 2.6 Slash notation for multiple signals

In Figure 2.6(a), it represents a block of combination logic with three inputs and output outputs.
If the number of bits is unimportant or obvious from the context, the slash may be shown without a number.
Figure 2.6(b) indicates two blocks of combinational logic with an arbitrary number of outputs from one block serving as inputs to the second block.

The rules of *combinational composition* tell us how we can build a large combinational circuit from smaller combinational circuit elements.
A circuit is combinational if it consists of interconnected circuit elements such that:

- Every circuit element is itself combinational
- Every node of the circuit is either designated as an input to the circuit or connects to exactly one output terminal of a circuit element
- The circuit contains no cyclic paths: every path through the circuit visits each circuit node at most once

Examples:



(a) is combinational. It is constructed from two combinational circuit elements (inverters $I1$ and $I2$). It has three nodes: $n1$, $n2$, and $n3$. $n1$ is an input to the circuit and to $I1$; $n2$ is an internal node, which is the output of $I1$ and the input to $I2$; $n3$ is the output of the circuit and of $I2$.
(b) is *not* combinational, because there is a cyclic path: the output of the XOR feeds back to one of its input. Hence, a cyclic path starting at $n4$ passes through the XOR to $n5$, which returns to $n4$.
(c) is combinational.
(d) is not combinational, because node $n6$ connects to the output terminals of both $I3$ and $I4$.
(e) is combinational, illustrating two combinational circuits connecting to from a larger combinational circuit.
(f) does not obey the rules of combinational composition because it has a cyclic path through the two elements. Depending on the functions of the elements, it may or may not be a combinational circuit.

The functional specification of a combinational circuit is usually expressed as a truth table or a Boolean equation.

5 Boolean Equations

Boolean equations deal with variables that are either TRUE or FALSE, so they are perfect for describing digital logic.

5.1 Terminology

The *complement* of a variable, A, is its inverse, $\overline{A}$.
The variables or its complement is called a *literal.* For example, A, $\overline{A}$, B, and $\overline{B}$ are literals.
A is called the *true form* of the variable and $\overline{A}$ the complementary form; "true form" does not mean that A is TRUE, but merely that A does not have a line over it.

The AND of one or more literals is called a *product* or an *implicant.*
$\overline{A}B$, $A\overline{BC}$, and B are all implicants for a function of three variables.
A *minterm* is a product involving all of the inputs to the function. $A\overline{BC}$ is a minterm for a function of the three variables A, B, and C, but $\overline{A}B$ is not, because it does not involve C.

Similarly, the OR of one or more literals is called a *sum.* A *maxterm* is a sum involving all of the inputs to the function. $A + \overline{B} + C$ is a maxterm for a function of the three variables A, B, and C.

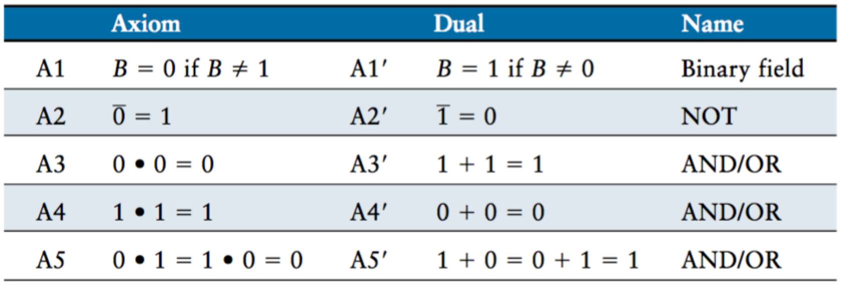
5.2 Sum-of-Products Form

A truth table of N inputs contains 2^N rows, one for each possible value of the inputs.
Each row in a truth value is associated with a minterm that is TRUE for that row.
A Boolean equation can be written for any truth table by summing each of the minterms for which the output, Y, is TRUE. This is called the *sum-of-products canonical form* of a function because it is the sum (OR) of products (ANDs forming minterms).

The sum-of-products form provides a Boolean equation for any truth table with any number of variables. Unfortunately, it does not necessarily generate the simplest equation.

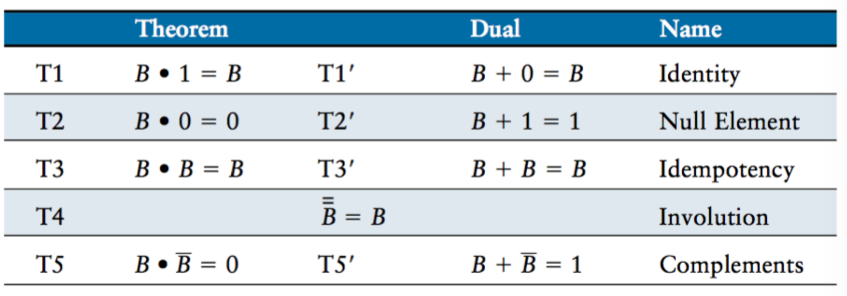
6 Boolean Algebra

6.1 Axioms



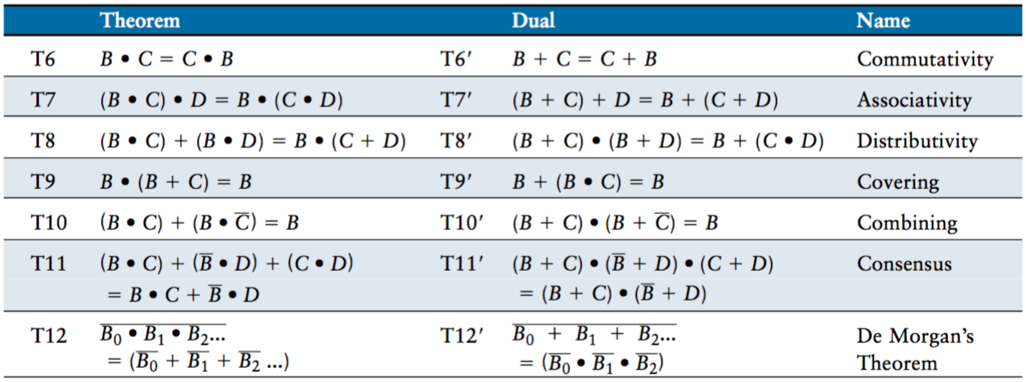
	Axiom		Dual	Name
A1	$B = 0$ if $B \neq 1$	A1′	$B = 1$ if $B \neq 0$	Binary field
A2	$\overline{0} = 1$	A2′	$\overline{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	A3′	$1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	A4′	$0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	A5′	$1 + 0 = 0 + 1 = 1$	AND/OR

6.2 Theorems of One Variable



	Theorem		Dual	Name
T1	$B \bullet 1 = B$	T1′	$B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	T2′	$B + 1 = 1$	Null Element
T3	$B \bullet B = B$	T3′	$B + B = B$	Idempotency
T4		$\overline{\overline{B}} = B$		Involution
T5	$B \bullet \overline{B} = 0$	T5′	$B + \overline{B} = 1$	Complements

6.3 Theorems of Several Variables



	Theorem		Dual	Name
T6	$B \bullet C = C \bullet B$	T6′	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	T7′	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \bullet C) + (B \bullet D) = B \bullet (C + D)$	T8′	$(B + C) \bullet (B + D) = B + (C \bullet D)$	Distributivity
T9	$B \bullet (B + C) = B$	T9′	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \overline{C}) = B$	T10′	$(B + C) \bullet (B + \overline{C}) = B$	Combining
T11	$(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D)$ $= B \bullet C + \overline{B} \bullet D$	T11′	$(B + C) \bullet (\overline{B} + D) \bullet (C + D)$ $= (B + C) \bullet (\overline{B} + D)$	Consensus
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \ldots}$ $= (\overline{B_0} + \overline{B_1} + \overline{B_2} \ldots)$	T12′	$\overline{B_0 + B_1 + B_2 \ldots}$ $= (\overline{B_0} \bullet \overline{B_1} \bullet \overline{B_2})$	De Morgan's Theorem

7 From Logic to Gates

A *schematic* is a diagram of a digital circuit showing the elements and the wires that connect them together.

Good Style in Circuit Drawing:

- Assume all literals (variables and their negations) are available
- Rectilinear wires, dots when wires split
- Do not draw spaghetti wires for inputs; instead, write each literal as needed

8 Multilevel Combinational Logic

Logic in sum-of-products form is called *two-level logic* because it consists of literals connected to a level of AND gates connected to a level of OR gates.
Designers often build circuits with more than two levels of logic gates. These multilevel combinational circuits may use less hardware than their two-level counterparts.
Bubble pushing is especially helpful in analyzing and designing multilevel circuits.

8.1 Hardware Reduction

Some logic functions require an enormous amount of hardware when built using two-level logic. A notable example is the XOR function of multiple variables. A three-input XOR can be built out of a cascade of two-input XORs.
Similarly, an eight-input XOR would require 128 eight-input AND gates and one 128-input OR gate for a two level-sum-of-products implementation. A much better option is to use a tree of two-input XOR gates.

"Best" has many meanings: fewest gates, fastest, shortest design time, least cost, least power consumption.
"Best" circuit in one technology is not necessarily the best in another. ANDs and ORs are used often, but in CMOS, NANDs and NORs are more efficient.

8.2 Bubble Pushing

CMOS circuits prefer NANDs and NORs over ANDs and ORs. However, reading the equation by inspection can be difficult.
Bubble pushing is a helpful way to redraw these circuits so that the bubbles cancel out and the function can be more easily determined.
Guidelines for bubble pushing:

- Begin at the output of the circuit and work toward the inputs.
- Push any bubbles on the final output back toward the inputs so that you can read an equation in terms of the output.
- Working backward, draw each gate in a form so that bubbles cancel. If the current gate has an input bubble, draw the preceding gate with an output bubble. If the current gate does not have an input bubble, draw the preceding gate without an output bubble.

9 Combinational Building Blocks

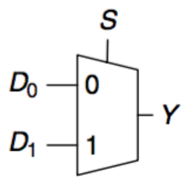
Combinational logic is often grouped into larger building blocks to build more complex systems. This is an application of the principle of abstraction, hiding the unnecessary gate-level details to emphasize the function of the building block.

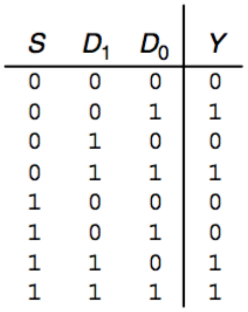
9.1 Multiplexers

Multiplexers, also known as *mux*, are among the most commonly used combinational circuits. They choose an output from among several possible inputs based on the value of a *select* signal.

9.1.1 2:1 Multiplexer

Image shows the schematic and truth table for a 2:1 multiplexer with two data inputs, D_0 and D_1, as select input, S, and one output, Y.
The multiplexer chooses between the two data inputs based on the select: if $S = 0$, $Y = D_0$, and if $S = 1$, $Y = D_1$.
S is also called a control signal because it controls what





S	D_1	D_0	Y
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

the multiplexer does.

Multiplexers can be built from tristate buffers.
The tristate enables are arranged such that, at all times, exactly one tristate buffer is active.
When $S = 0$, tristate T_0 is enabled, allowing D_0 to flow to Y.
When $S = 1$, tristate T_1 is enabled, allowing D_1 to flow to Y.

9.1.2 Wider Multiplexers

A 4:1 multiplexer has four data inputs and one output. Two select signals are needed to choose among the four data inputs. The 4:1 mux can be built using sum-of-products logic, tristates, or multiple 2:1 mux.

The product terms enabling the tristates can be formed using AND gates and inverters. They can also be formed using a decoder.

In general, an $N : 1$ multiplexer needs $log_2 N$ select lines.

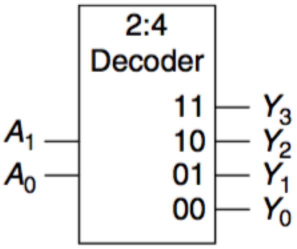
9.1.3 Multiplexer Logic

Multiplexers can be used as *lookup tables* to perform logic functions.
In general, a 2^N-input multiplexer can be programmed to perform any N-input logic functions by applying 0's and 1's to the appropriate data inputs.
By changing the data inputs, the multiplexer can be reprogrammed to perform a different function.

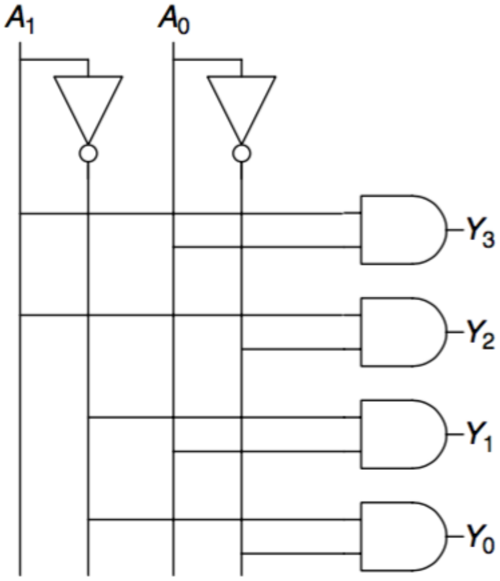
9.2 Decoder

A decoder has N inputs and 2^N outputs.
It asserts exactly one of its outputs depending on the input combination. The outputs are called *one-hot*, because exactly one is "hot" (HIGH) at a given time.

Example of 2:4 decoder and its implementation:



A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



9.2.1 Decoder Logic

Decoders can be combined with OR gates to build logic functions.
When using decoders to build logic, it is easiest to express functions as a truth table or in canonical sum-of-products form. A N-input function with M 1's in the truth table can be built with an $N : 2^N$ decoder and an M-input or gate attached to all of the minterms containing 1's in the truth table.

10 Summary So Far

A digital circuit is a module with discrete-valued inputs and outputs and a specification describing the function and timing of the module.

The function of a combinational circuit can be given by a truth table or a Boolean equation. The Boolean equation for any truth table can be obtained systematically using sum-of-products or product-of-sums form.
In sum-of-products form, the function is written as the sum (OR) of one or more implicants. Implicants are the product (AND) of literals. Literals are the true or complementary forms of the input variables.

Boolean equation can be simplified using the rules of Boolean algebra. In particular, they can be simplified into minimal sum-of-products form by combining implicants that differ only in the true and complementary forms of one of the literals: $PA + P\overline{A} = P$.

Logic gates are connected to create combinational circuits that perform the desired function.
Any functions in sum-of-products form can be built using two-level logic with the literals as inputs: NOT gates form the complementary literals, AND gates form the products, and OR gates form the sum.
CMOS circuits favor NAND and NOR gates because these gates can be built directly from CMOS transistors without requiring extra NOT gates.
When using NAND and NOR gates, bubble pushing is helpful to keep track of the inversions.

Logic gates are combined to produce larger circuits such as multiplexers, decoders, and priority circuits.
A multiplexer chooses one of the data inputs based on the select input.
A decoder sets one of the outputs HIGH according to the input.
A priority circuit produces an output indicating the highest priority input.

The timing specification of a combinational circuit consists of the propagation and contamination delays through the circuit.
These indicate the longest and shortest times between an input change and the consequent output change.
Calculating the propagation delay of a circuit involves identifying the critical path through the circuit, then adding up the propagation delays of each element along that path.

11 Introduction to Sequential Logic Design

The output of sequential logic depend on both current and prior input values. Hence, sequential logic has memory. Sequential logic might explicitly remember certain previous inputs, or it might distill the prior inputs into a smaller amount of information called the *state* of the system.
The state of a digital sequential circuit is a set of bits called *state variables* that contain all the information about the past necessary to explain the future behavior of the circuit.

12 Latches and Flip-Flops

The fundamental building block of memory is a *bistable* element, an element with two stable states.
An element with N stable states conveys $\log_2 N$ bits of information, so a bistable element stores one bit.
The state of the cross-coupled inverters is contained in one binary state variable, Q. The value of Q tells us everything about the past that is necessary to explain the future behavior of the circuit.
Specifically, if $Q = 0$, it will remain 0 forever, and if $Q = 1$, it will remain 1 forever.
The circuit does have another node, $\overline{Q}$, but $\overline{Q}$ does not contain any additional information because if Q is known, $\overline{Q}$ is also known. $\overline{Q}$ is also an acceptable choice for the state variable.

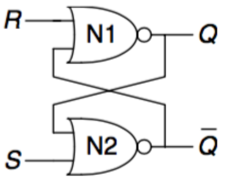
When power is first applied to a sequential circuit, the initial state is unknown and usually unpredictable. It may differ each time the circuit is turned on.

Although the cross-coupled inverters can store a bit of information, they are not practical because the user has no inputs to control the states.
However, other bistable elements, such as *latches* and *flip-flops*, provide inputs to control the value of the state variable.

Just a Y is commonly used for the output of combinational logic, Q is commonly used for the output of sequential logic.

12.1 SR Latch

SR latch is one of the simplest sequential circuit as it is composed of two cross-coupled NOR gates.
The latch has two inputs, S and R, and two outputs, Q and $\overline{Q}$.
The SR latch is similar to the cross-coupled inverters, but its state can be controlled through the S and R inputs, which *set* and *reset* the output Q.



Consider the four possible combinations of R and S:

Case I: $R = 1, S = 0$
N1 sees at least one TRUE input, R, so it produces a FALSE output on Q. N2 sees both Q and S FALSE, so it produces a TRUE output on $\overline{Q}$.

Case II: $R = 0, S = 1$
N1 receives inputs of 0 and $\overline{Q}$. Because we don't yet know $\overline{Q}$, we can't determine the output Q. N2 receives at least one TRUE input, S, so it produces a FALSE output on $\overline{Q}$. Now we can revisit N1, knowing that both inputs are FALSE, so the output Q is TRUE.

Case III: $R = 1, S = 1$
N1 and N2 both set at least one TRUE input (R or S), so each produces a FALSE output. Hence Q and $\overline{Q}$ are both FALSE.

Case IV: $R = 0, S = 0$
N1 receives inputs of 0 and $\overline{Q}$. Because we don't yet know $\overline{Q}$, we can't determine the output. N2 receives inputs of 0 and Q. Because we don't yet know Q, we can't determine the output. Now we are stuck. This is reminiscent of the cross-coupled inverters. But we know that Q must either be 0 or 1. So we can solve the problem by checking what happens in each of these sub-cases.

Case IVa: $Q = 0$
Because S and Q are FALSE, N2 produces a TRUE output on $\overline{Q}$. Now N1 receives one TRUE input, $\overline{Q}$, so its output, Q, is FALSE.

Case IVb: $Q = 1$
Because Q is TRUE, N2 produces a FALSE output on $\overline{Q}$. Now N1 receives two FALSE inputs, R and $\overline{Q}$, so its output, Q, is TRUE.

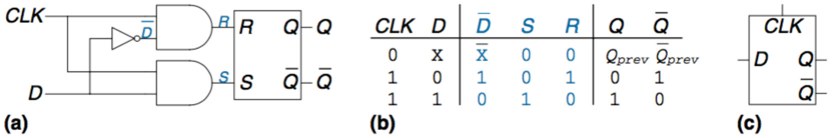
Suppose Q has some known prior value, Q_{prev}, before we enter Case IV. Q_{prev} is either 0 or 1, and represents the state of the system. When R and S are 0, Q will remember this old value, Q_{prev}, and $\overline{Q}$ will be its complement, $\overline{Q}_{prev}$. This circuit has memory.

Like the cross-coupled inverters, the SR latch is a bistable element with one bit of state stored in Q. However, the state can be controlled through the S and R inputs.
When R is asserted, the state is reset to 0.
When S is asserted, the state is set to 1.
When neither is asserted, the state retains its old value.

12.2 D Latch

The SR latch is awkward because it behaves strangely when both S and R are simultaneously asserted. Moreover, the S and R inputs conflate the issues of *what* and *when*. Asserting one of the inputs determines not only *what* the state should be but also *when* it should change.

The D latch has two inputs. The *data* input, D, controls what the next state should be. The *clock* input, CLK, controls when the state should change.



CLK	D	$\overline{D}$	S	R	Q	$\overline{Q}$
0	X	$\overline{X}$	0	0	Q_{prev}	$\overline{Q}_{prev}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0

Figure 3.7 D latch: (a) schematic, (b) truth table, (c) symbol

The clock controls when data flows through the latch.
When $CLK = 1$, the latch is *transparent*. The data at D flows through to Q as if the latch were just a buffer.
When $CLK = 0$, the latch is *opaque*. It blocks the new data from flowing through to Q, and Q retains the old value.
The D latch updates its state continuously $CLK = 1$.

12.3 D Flip-Flop

A *D flip-flop* can be built from two back-to-back D latches controlled by complementary clocks.
The first latch, $L1$, is called the *master*.
The second latch, $L2$, is called the *slave*.
The node between them is named $N1$.
A symbol for the D flip-flop is given in the figure (b).
When the $\overline{Q}$ output is not needed, the symbol can be condensed as in figure (c).

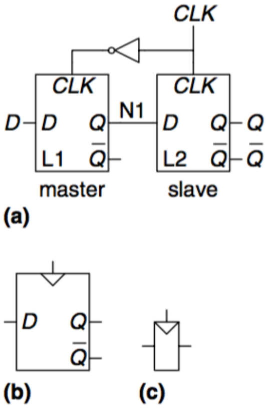


Figure 3.8 D flip-flop: (a) schematic, (b) symbol, (c) condensed symbol

When $CLK = 0$, the master latch is transparent and the slave is opaque. The value at D propagates through to $N1$.
When $CLK = 1$, the master goes opaque and the slave becomes transparent. The value at $N1$ propagates through to Q, but $N1$ is cut off from D. Hence, the value at D immediately before the clock rises from 0 too 1 gets copied to Q immediately after the clock rises.
At other times, Q retains its old value, because there is always an opaque latch blocking the path between D and Q.

A *D flip-flop* copies D to Q on the rising edge of the clock, and remembers its state at all other times.

A D flip-flop is also known as a *master-slave flip-flop*, an *edge-triggered flip-flop*, or a *positive edge-triggered flip-flop*.

12.4 Register

A N-bit register is a bank of N flip-flops that share a common CLK input, so that all bits of the register are updated at the same time.
Registers are the key building block of most sequential circuits.

For a 32-bit architectures, a register would typically have 32 flip-flops in parallel (i.e. one for each bit).
A *register file* is a way of organizing registers.

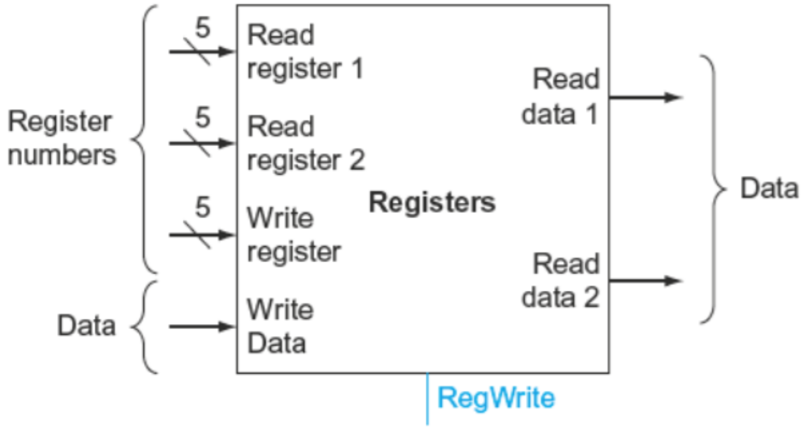


Figure 2: A register file.

12.4.1 Reading from a Register File

The values of all 32 registers go through two multiplexers.
The first multiplexer determines which register value will be routed to the 1st output.
The second multiplexer determines which register value will be routed to the 2nd output.
It will take 5 select lines to uniquely identify each source register (when there are 32 registers).

12.4.2 Writing to a Register File

The address of the register d, is routed through a decoder which determines which D flip-flop to write to.
When the d^{th} decoder line and the *RegWrite* line are both 1, they pass a 1 through the d^{th} AND gate and raise the C input of the d^{th} D flip-flop to 1.
The data will now be written to Register $\$d^{\text{th}}$.

13 Finite State Machines

Synchronous sequential circuits can be drawn in the forms called *finite state machines (FSMs)*. The name comes from a circuit with k registers can be in one of a finite number (2^k) of unique states.
A FSM has M inputs, N outputs, and k bits of state. It also receives a clock and, optionally, a reset signal.

An FSM consists of two blocks of combinational logic, *next state logic* and *output logic*, and a register that stores the state. On each clock edge, the FSM advances to the next state, which was computed based on the current state and inputs.

Finite state machines provides a systematic way to design synchronous sequential circuits given a functional specification.

Two general classes of finite state machines, characterized by their functional specifications.

In *Moore machines*, the outputs depend only on the current state of the machine.

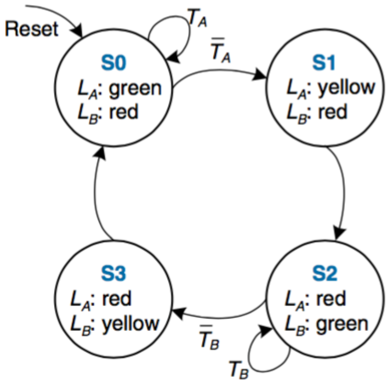
In *Mealy machines*, the outputs depend on both the current state and the current inputs.

13.1 State Transition Diagram

A *state transition diagram* indicates all the possible states of the system and the transitions between these states.

In a state transition diagram, circles represent states and arcs represent transitions betweens states.

The transitions take place on the rising edge of the clock; the clock can hidden on the diagram because it is always present in a synchronous sequential circuit. In addition, the clock only controls when the transitions should occur, whereas the diagram indicates which transitions occur.



If a state has multiple arcs leaving it, the arcs are labeled to show what input triggers each transition. In the image, when in state $S0$, the system will remain in that state if T_A is TRUE and move to $S1$ if T_A is FALSE.

If a state has a single arc leaving it, that transition always occurs regardless of the inputs. In the image, when in state $S1$, the system will always move to $S2$.

13.2 State Transition Table

A *state transition table* indicates for each state and input, what the next state, S', should be. If the next state does not depend on a particular input, the

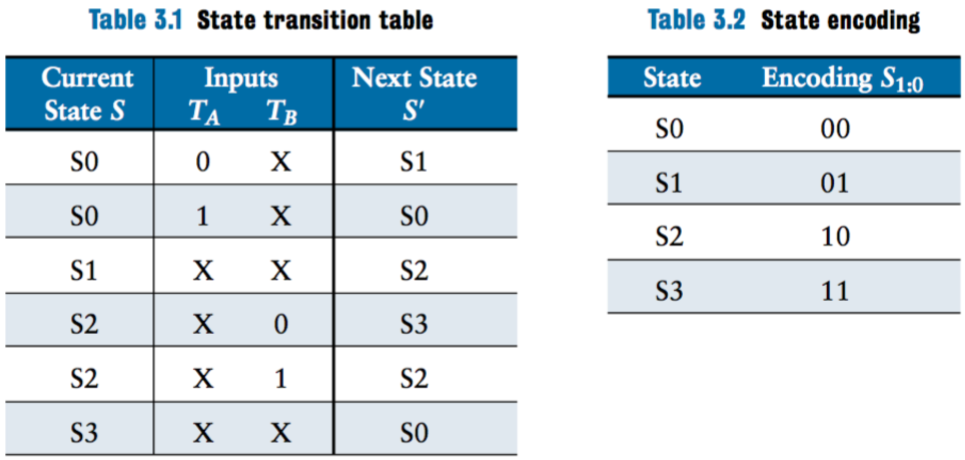


Table 3.1 State transition table

Current State S	Inputs T_A	T_B	Next State S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

Table 3.2 State encoding

State	Encoding $S_{1:0}$
S0	00
S1	01
S2	10
S3	11

Figure 3: Example Transition Table and its Encoding

don't care symbol (X) is used.
To build a real circuit, the states and outputs must be assigned *binary encodings*.

13.3 State Encoding

If the state and output encodings were selected arbitrarily, a different choice would have resulted in a different circuit.
Then how to determine the encoding that produces the circuit with the fewest logic gates or the shortest propagation delay? There is not simply method to find the best encoding.
It is possible to choose a good encoding by inspection, so that related states share bits.

An important decision in state encoding is the choice between binary encoding and one-hot encoding.
With *binary encoding*, each state is represented as a binary number. Because K binary numbers can be represented by $\log_2 K$ bits, a system with K states only needs $\log_2 K$ bits of state.

In *one-hot encoding*, a separate bit of state is used for each state. It is called one-hot because only one bit is "hot" or TRUE at any time.

Example 13.1. A one-hot encoded FSM with three states would have state encodings of 001, 010, and 100.

Each bit of state is stored in a flip-flop, so one-hot encoding requires more flip-flops than binary encoding. However, with one-hot encoding, the next-state and output logic is often simpler, so fewer gates are required.

Another encoding is the *one-cold* encoding, in which K states are represented with K bits, exactly one of which is FALSE.

13.4 FSM Review

Finite state machines are a powerful way to systematically design sequential circuits from a written specification.
Procedure to design an FSM:

> Identify the inputs and outputs.

> Sketch a state transition diagram.

> For a Moore machine:

  - Write a state transition table.
  - Write an output table.

> For a Mealy machine:

  - Write a combined state transition and output table.

> Select state encodings—selection affects the hardware design.

> Write Boolean equations for the next state and output logic.

> Sketch the circuit schematics.

14 Summary So Far

In contrast to combinational logic, whose outputs depend solely on the current inputs, sequential logic outputs depend on both current and prior inputs. Sequential logic remembers information about prior inputs. This memory is called the state of the logic.

Sequential circuits can be difficult to analyze and are easy to design incorrectly. An important element is the flip-flop, which receives a clock and an input, D, and produces an output, Q. The flip-flop copies D to Q on the rising edge of the clock and otherwise remembers the old state of Q. A group of flip-flops sharing a common clock is called a register. Flip-flops may also receive reset or enable control signals.

Finite state machines are a powerful technique for designing sequential circuits. For procedure on designing an FSM, refer to Section 13.4 (FSM Review).

Synchronous sequential circuits have a timing specification including to clock-to-Q propagation and contamination delay, t_{pcq} and t_{ccq}, and the setup and hold times, t_{setup} and t_{hold}.
For correct operation, their inputs must be stable during an aperture time that starts a setup time before the rising edge of the clock and ends a hold time after the rising edge of the clock.

The minimum cycle time, T_c, of the system is equal to the propagation delay, t_{pd}, through combinational logic plus $t_{pcq} + t_{setup}$ of the register.
For correct operation, the contamination delay through the register and combinational logic must be greater than t_{hold}. Hold time does not affect the cycle time.

Overall performance is measured in latency and throughput. The latency is the time required for a token to pass from start to end. The throughput is the number of tokens that the system can process per unit time. Parallelism improves the system throughput.

15 Memory Technologies

There are four primary technologies used in memory hierarchies.
Main memory is implemented from DRAM (dynamic random access memory), while levels closer the processor (caches) uses SRAM (static random access memory).
DRAM is less costly per bit than SRAM, although it is substantially slower. The price difference arises because DRAM uses significantly less area per bit of memory, and DRAMs thus have larger capacity for the same amount of silicon.

15.1 SRAM Technology

SRAMs are simply integrated circuits that are memory arrays with (usually) a single access port that can provide either a read or a write.
SRAMs have a fixed access time to any datum, though read and write access times may differ.
SRAMs don't need to refresh and so the access time is very close to the cycle time.
SRAMs typically use six to eight transistors per bit to prevent the information from being disturbed when read.
SRAM needs only minimal power to retain the charge in standby mode.

15.2 DRAM Technology

In a SRAM, as long as power is applied, the value can be kept indefinitely.
In a dynamic RAM, the value kept in a cell is stored as a charge in a capacitor.
A single transistor is then used to access this stored charge, either to read the value or to overwrite the charge stored there.
Because DRAMs use only a single transistors per bit of storage, they are much denser and cheaper per bit than SRAM.
As DRAMs store the charge on a capacitor, it cannot be kept indefinitely and must periodically be refreshed. Hence this memory structure is dynamic, as opposed to the static storage in an SRAM cell.

To refresh the cell, we merely read its contents and write it back. The charge can be kept for several ms. If every bit had to be read out of the DRAM and then written back individually, the DRAM would be constantly refreshed, leaving no time to access it.
DRAMs use a two-level decoding structure, which allows the ability to refresh an entire *row* (which shares a word line) with a read cycle followed immediately by a write cycle.

The row organization that helps with refresh also helps with performance. To improve performance, DRAMs buffer rows for repeated access. The buffer acts like a SRAM; by changing the address, random bits can be accessed in the buffer until the next row access. This capability improves the access time significantly, since the access time to bits in the row is much lower. When the row is in the buffer, it can be transferred by successive addresses at whatever the width of the DRAM is, or by specifying a block transfer and the starting address within the buffer.

To further improve the interface to processors, DRAMs added clocks which is called Synchronous DRAMs or SDRAMs. The advantage is that the use of a clock eliminates the time for the memory and processor to synchronize. The speed advantage of synchronous DRAMs come from the ability to transfer the bits in the burst without having to specify additional address bits. Instead, the clock transfer the successive bits in a burst.
Double Data Rate (DDR) SDRAM is the fastest currently available. Data transfer on both the rising and falling edge of the clock, thereby getting twice as much bandwidth based on the clock rate and the data width.

Sustaining the bandwidth requires clever organization *inside* the DRAM. The DRAM can be internally organized, instead of just a faster row buffer, to read or write from multiple *banks*, which each having its own row buffer. Sending an address to several banks permits them all to read or write simultaneously. The rotating access scheme is called *address interleaving*.

16 Multiplication

The first operand is called the *multiplicand* and the second the *multiplier*. The final result is called the *product*.
Number of bits in the product is considerably larger than the number in either the multiplicand or the multiplier. If ignoring the sign bit, the length of the multiplication of an n-bit multiplicand and an m-bit multiplier is a product that is $n + m$ bits long. So $n + m$ bits are required to represent all possible products.

17 Floating Point

A number in scientific notation that has no leading 0s is called a *normalized* number, which is the usual way to write it.
Floating point is computer arithmetic that represents numbers in which the binary point is not fixed.
A standard scientific notation for reals in normalized form offers tree advantages:

1. It simplifies exchange of data that includes floating-point numbers.

2. It simplifies the floating-point arithmetic algorithms to know that numbers will always be in this form.

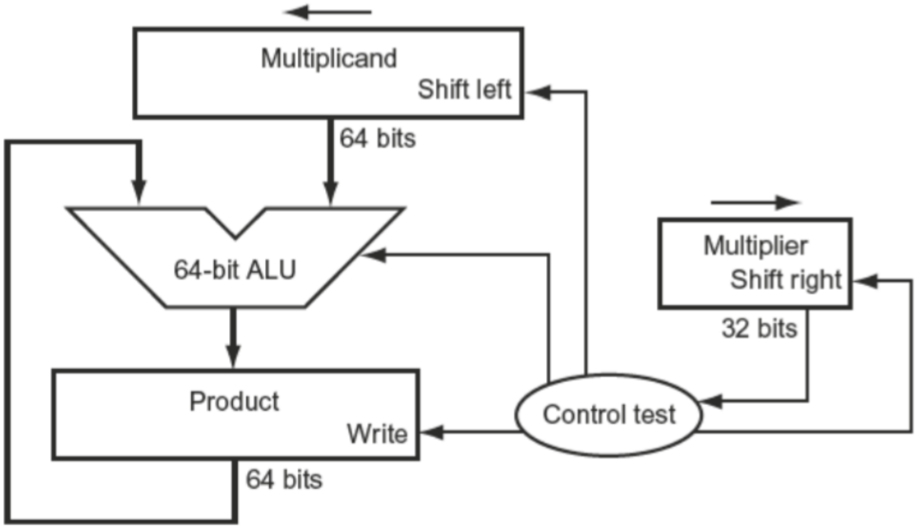
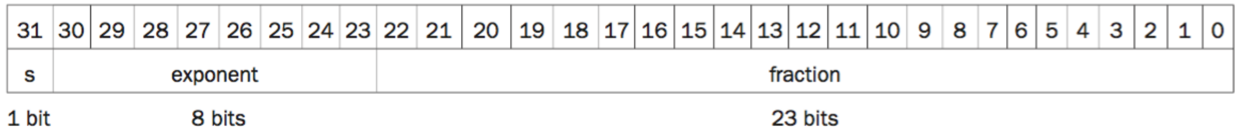


FIGURE 3.3 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. (Appendix B describes ALUs.) The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

Figure 4: Multiplication Hardware Version 1.0

3. It increases the accuracy of the numbers that can be stored in a word, since the unnecessary leading 0s are replaced by real digits to the right of the binary point.

17.1 Floating-Point Representation



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	exponent								fraction																						
1 bit	8 bits								23 bits																						

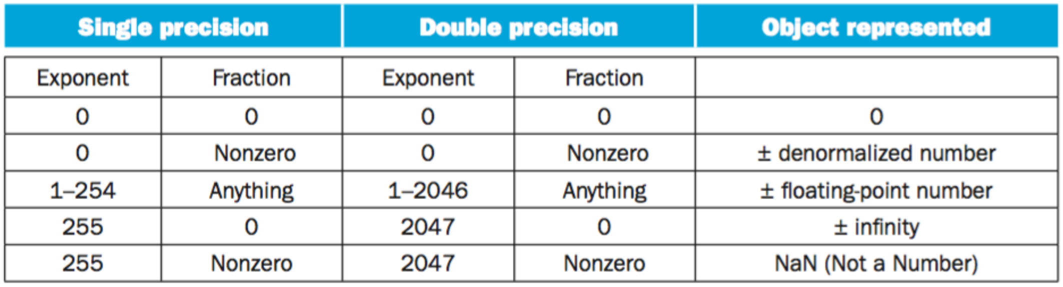
Figure 5: Representation of a MIPS *single precision* floating-point number.

s is the sign of the floating-point number (1 meaning negative). *exponent* is the value of the 8-bit exponent field (including the sign of the exponent). *fraction* is the 23-bit number.

Generally, floating-point numbers are of the form:

$$(-1)^s \times F \times 2^E$$

F involves the value in the fraction field and E involves the value in the exponent field.
A *double precision* floating-point number is represented in two 32-bit words. The exponent is the value of the 11-bit exponent field. Fraction is the 52-bit number in the fraction field.



Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Figure 6: IEEE 754 encoding of floating-point numbers.

This is also known as the *IEEE 754 floating-point standard*. This standard has greatly improved both the ease of porting floating-point programs and the quality of computer arithmetic.
In IEEE 754, the leading 1-bit of normalized binary number is implicit, so the number is actually 24 bits long in single precision.

For the fractional part, suppose we number the bits of the fraction from left to right $s1$, $s2$, $s3$, ..., then the value is

$$(-1)^S \times (1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + (s3 \times 2^{-3}) + (s4 \times 2^{-4}) + \dots) \times 2^E$$

IEEE 754 uses a bias of 127 for single precision exponent.
The exponent bias for double precision is 1023.
So the value represented by a floating-point number is really

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})}$$

overflow: positive exponent becomes too large to fit in the exponent field
underflow: negative exponent becomes too large to fit in the exponent field

17.2 Floating-Point Addition

1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent.

2. Add the significants.

3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent.

> Overflow or underflow? If yes, Exception. Otherwise, continue.

4. Round the significant to the appropriate number of bits.

> Still normalized? If no, then step 3. Otherwise, done.

17.3 Floating-Point Multiplication

1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent.

2. Multiply the significants.

3. Normalize the product if necessary, shifting it right and incrementing the exponent.

> Overflow or underflow? If yes, Exception. Otherwise, continue.

4. Round the significant to the appropriate number of bits.

> Still normalized? If no, then Step 3. Otherwise, continue.

5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ, make the sign negative.

18 Introduction to the Processor

A subset of the core MIPS instruction set will be examined:

> The memory-reference instructions *load word* (lw) and *store word* (sw)

> The arithmetic-logical instructions add, sub, AND, OR, and slt

> The branch instructions *branch equal* (beq) and *jump* (j)

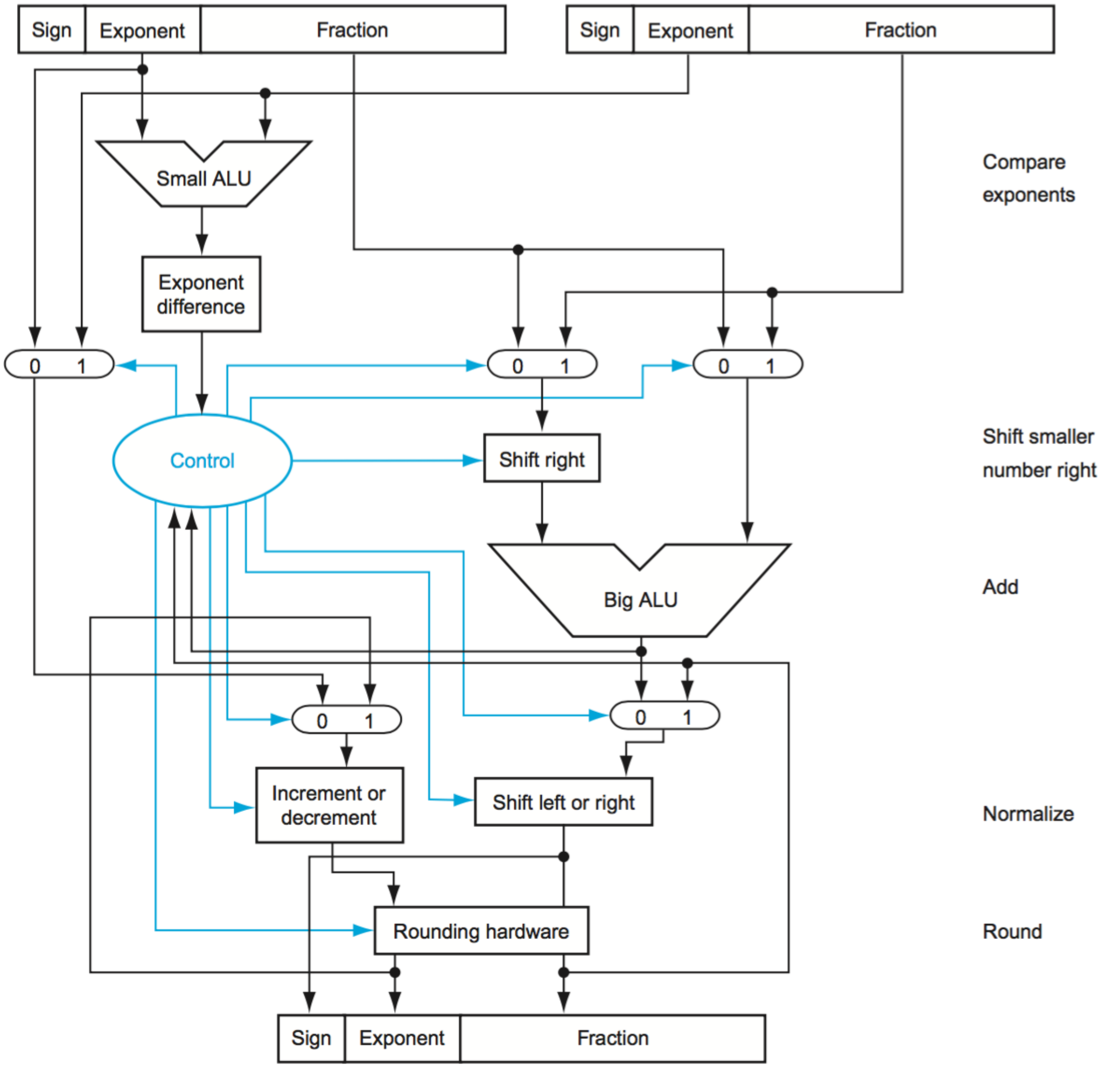


Figure 7: Arithmetic unit for floating-point addition.

To implement those instructions, the first two steps are identical:

1. Send the *program counter* (PC) to the memory that contains the code and fetch the instruction from that memory.
2. Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction, only need to read one register, but most other instructions require reading two registers.

After the two steps, the required actions depend on the instruction class. For

each of the three instructions classes (Section 18), the actions are largely the same, independent of the exact instruction.
The simplicity and regularity of the MIPS instruction set simplifies the implementation by making the execution of many of the instruction classes similar.

All instruction classes, except jump, use the arithmetic-logical unit (ALU) after reading the registers. The memory-reference instructions use the ALU for an address calculation, the arithmetic-logical instructions for the operation execution, and branches for comparison.
After using the ALU, the actions required to complete various instruction classes differ.
A memory-reference instruction will need to access the memory either to read data for a load or write data for a store.
An arithmetic-logical or load instruction must write the data from the ALU or memory back into a register.
For a branch instruction, it may be required to change the next instruction address based on the comparison; otherwise, the PC should be incremented by 4 to get the address of the next instruction.

19 Building a Data-path

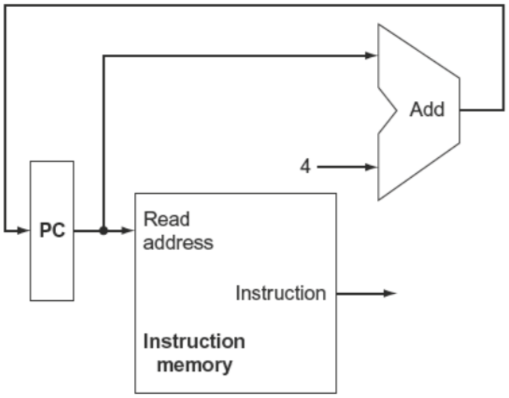


Figure 8: A portion of the data-path used for fetching instructions and incrementing the program counter.

A *data-path element* is a unit used to operate within a processor. In the MIPS implementation, the data-path element include the instruction and data memories, the register file, the ALU, and adders.

To execute any instruction, start by fetching the instruction from memory. Then increment the program counter so that it points at the next instruction, 4 bytes later.
Refer to the figure above for how the elements are combined, forming a datapath that fetches instructions and increments the PC.

A *R-type* or *R-format* is an arithmetical-logical instruction since they perform. All R-format instructions read two registers, perform an ALU operation on the contents of the register, and write the result to a register.

Through a register file, it contains the register state of the computer and the ALU needs to operate on the values read from the registers.
R-format instructions have three register operands, to read two data words from the register file and write one data word into the register file for each instruction.
For each data word to be read from the register, an input to the register file is needed that will carry the value that has been read from the registers.
To write a data word, two inputs are needed: one to specify the register number to be written and one to supply the *data* to be written into the register.
The register file always outputs the contents of whatever register numbers are on the Read register inputs. However, writes are controlled by the write control signal, which must be asserted for a write to occur at the clock edge.

Thus, we need a total of four inputs (three for register number and one for data) and two outputs (both for data). The register number inputs are 5 bits wide to specify one of 32 registers ($32 = 2^5$), whereas the data input and two data output buses are each 32 bits wide (each word is 32 bits).

19.1 R-Format ALU Operations

Two elements are needed to implement R-format ALU operations: register file (Section 12.4) and the ALU.

The register file contains all the registers and has two read ports and one write port. The register file outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed.
Note that a register write must be explicitly indicated by asserting the write control signal, $RegWrite$. Writes are edge-triggered, so that all write inputs must be valid at the clock edge.

The inputs carrying the register number to the register file are all 5 bites wide, whereas the lines carrying data values are 32 bits wide.
The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4 bits wide.

19.2 Memory-Reference Operations

Consider *lw* and *sw* instructions. They compute a memory address by adding the base register to the 16-bit signed offset field contained in the instruction.
If the instruction is a store, the value to be stored must also be read from the register file where it resides in the destination register.
If the instruction is a load, the value read from memory must be written into the register file in the specified register.
Thus, both register and the ALU are required in addition to more.

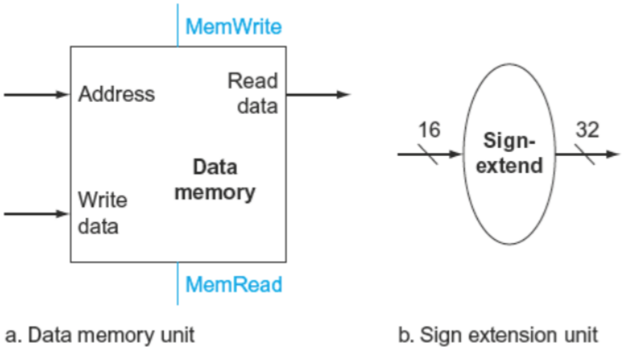


Figure 9: Two units needed for loads and stores. In addition to the register file and ALU.

We need a unit to *sign-extend* the 16-bit offset field in the instruction to a 32-bit signed value, and a data memory unit to read from or write to. A *sign-extend* is to increase the size of a data item by replicating the high-order sign bit of the original data item in the high-order bits of the larger, destination data item.
The data memory must be written on store instructions; hence, data memory has read and write control signals, an address input, and an input for the data to be written into memory.
Refer to figure above for the two units required.

19.3 Branch Operations