

ECE 271: Chapter 5 Reading Report

Phi Luu

November 14th, 2018

1 Chapter Outline

1.1 Introduction

Chapter 2 covers combinational logic; chapter 3 covers sequential logic; and chapter 4 covers hardware description language. This chapter puts combinational logic and sequential logic together to form digital building blocks, making larger hardware. This chapter is the transition from the lower level—logic—to the higher level—architecture.

This chapter will go through basic digital building blocks, including arithmetic circuits, counters, shift registers, memory arrays, and logic arrays. These building blocks are used to build a microprocessor.

1.2 Arithmetic Circuits

Arithmetic circuits are the central building blocks of computers. They enable computers and digital logic perform arithmetic functions, such as addition, subtraction, multiplication, division, comparisons, and shifts.

1. Addition

Addition is one of the most common operations in digital systems. The *half adder* and the *full adder* mentioned in earlier chapters are two of the digital building blocks that perform addition.

(a) Half Adder

The most basic example of a half adder is a 1-bit half adder. The half adder takes in two inputs—two binary bit—and add them together to output the sum and the carry out, as demonstrated by Figure 1.2.1 below:

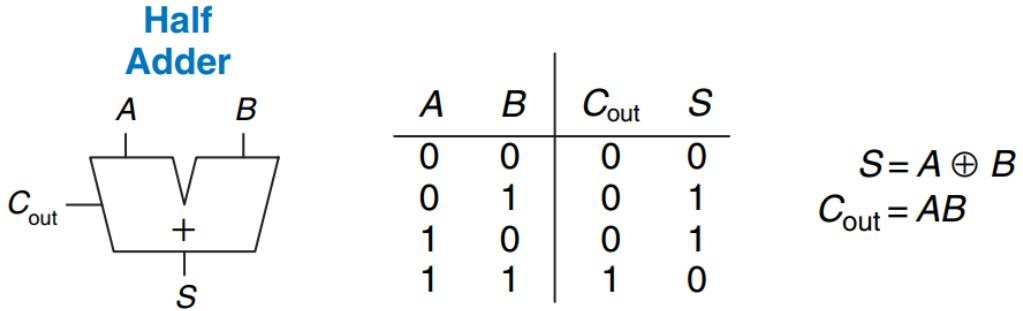


Figure 1.2.1: A 1-bit half adder's block schematic, truth table, and Boolean equations (respectively)

Similar to decimal addition, binary addition "carries" a 1 to the next column (next half adder) when the result overflows—in this case, the result of $1 + 1$. Using the truth table provided in Figure 1.2.1, the Boolean equation of S and C_{out} can be solved as $S = A \oplus B$ (XOR logic) and $C_{\text{out}} = AB$ (AND logic).

By putting multiple half adders together, a multi-adder system is formed. The C_{out} (carry out) of the less significant bit is the carry in of the more significant bit—in other words, from right to left—just like decimal addition. However, the half adder lacks the C_{in} (carry in) input. The *full adder* solves this problem.

(b) Full Adder

As described in the half adder section, the *full adder* takes in three inputs—two binary bits A and B and one carry-in bit C_{in} —and produces two outputs, one carry-out bit C_{out} and one sum bit S .

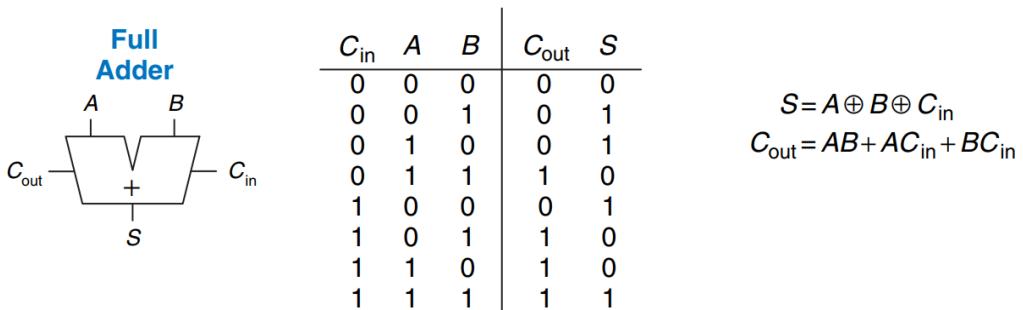


Figure 1.2.2: A 1-bit full adder's block schematic, truth table, and Boolean equations (respectively)

The addition of the full adder is exactly like that of the half adder, but the full adder takes in an extra carry-in bit and adds three 1-bit binary numbers A , B , and C_{in} . Using Boolean simplification methods (K-maps or pure Boolean algebra), the Boolean equations of the sum S and the carry out C_{out} can be expressed by Figure 1.2.2. Knowing the Boolean equations, the designer can easily implement the half adder and the full adder using combinational logic.

(c) Carry Propagate Adder

An N -bit adder sums two N -bit inputs, A and B , and a carry in C_{in} to produce an N -bit result S and a carry out C_{out} . The *carry propagate adder* propagates the carry out of one bit to the next. The block schematic of the carry propagate adder is similar to that

of the full adder, but the carry propagate adder takes in N -bit inputs instead of 1-bit, as illustrated by Figure 1.2.3 below:

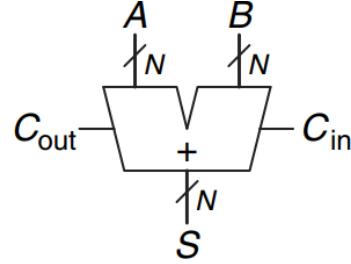


Figure 1.2.3: A carry propagate adder's block schematic

(d) Ripple-Carry Adder

The simplest way to build an N -bit carry propagate adder is to chain together N full adders. As stated in the full adder section, the C_{out} of the less significant bit is the C_{in} of the more significant bit (carrying from right to left). This is called the *ripple-carry adder*. The advantage of this system is its simplicity—a large system can be built by chaining multiple identical modules. However, its disadvantage is its speed—as C_N depends on C_{N-1} , which depends on C_{N-2} , which depends on C_{N-3} , and so on until C_0 . The block schematic of a 32-bit ripple-carry adder is as illustrated by Figure 1.2.4 below:

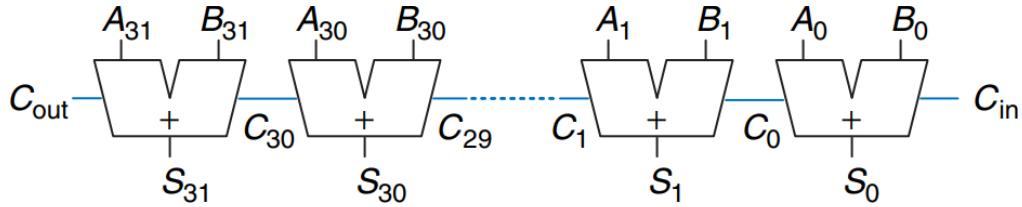


Figure 1.2.4: A ripple-carry adder's block schematic

The delay of the adder, t_{ripple} , grows linearly with the number of bits, as given in Equation 1.2.1, where t_{FA} is the delay of a full adder:

$$t_{ripple} = N t_{FA} \quad (1.2.1)$$

(e) Carry-Lookahead Adder

The *carry-lookahead adder* solves the ripple-carry adder by dividing the adder into *blocks* and providing circuitry to quickly determine the carry out of a block as soon as the carry in is known. It *looks ahead* across the blocks rather than waiting to ripple through all the full adders inside a block. For example, a 32-bit adder can be divided into eight 4-bit blocks.

Carry-lookahead adders use *generate* (G) and *propagate* (P) signals that describe how a column or block determines the carry out.

The i th column of an adder *generates* a carry if it produces a carry out regardless of the value of the carry in. According to the truth table in Figure 1.2.2, $C_{out} = 1$ regardless

of the value of C_{in} if and only if $A = B = 1$. Thus, $G_i = A_i B_i$ in the i th column of the adder.

The i th column is said to *propagate* a carry if it produces a carry out whenever there is a carry in. Using Figure 1.2.2 once again, $C_{out} = 1$ when $C_{in} = 1$ if and only if $A = 1$ or $B = 1$. Thus, $P_i = A_i + B_i$.

Putting G_i and P_i together, the i th column of an adder will generate a carry out C_i if either generates a carry, G_i , or propagates a carry in, $P_i C_{i-1}$. In equation form,

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1} \quad (1.2.2)$$

Using the block generate and propagate signals, the carry out of the block, C_i can be computed by the carry in to the block, C_j ,

$$C_i = G_{i:j} + P_{i:j} C_j \quad (1.2.3)$$

The delay of an N -bit adder divided into k -bit block is as follows:

$$t_{CLA} = t_{pg} + t_{pg_block} + \left(\frac{N}{k} - 1 \right) t_{AND_OR} + k t_{FA} \quad (1.2.4)$$

(f) Prefix Adder

Prefix adders extend the generate and propagate logic of the carry-lookahead adder to perform addition even faster. The strategy of the prefix adder is to compute the carry in C_{i-1} for each column i as fast as possible, then to compute the sum using

$$S_i = (A_i \oplus B_i) \oplus C_{i-1} \quad (1.2.5)$$

The delay of an N -bit prefix adder is

$$t_{PA} = t_{pg} + \log_2 N (t_{pg_prefix}) + t_{XOR} \quad (1.2.6)$$

2. Subtraction

Subtraction is just a form of addition where a number adds to a negative number. Mathematically, in decimal system, $A - B = A + (-B)$.

In binary system, to compute $Y = A - B$, first create the two's complement of B : invert the bits of B to obtain \bar{B} and add 1 to get $-B = \bar{B} + 1$. Hence, $Y = A - B = A + \bar{B} + 1$. This sum can be performed with a single carry propagate adder adding A and \bar{B} with $C_{in} = 1$.

3. Comparators

A *comparator* determines whether a binary number is equal, greater than, or less than another binary number. A comparator receives two N -bit binary numbers A and B . There are two common types of comparators:

- (a) An *equality comparator* produces a single output indicating whether A is equal to B . A 4-bit equality comparator is illustrated by Figure 1.2.5 below.

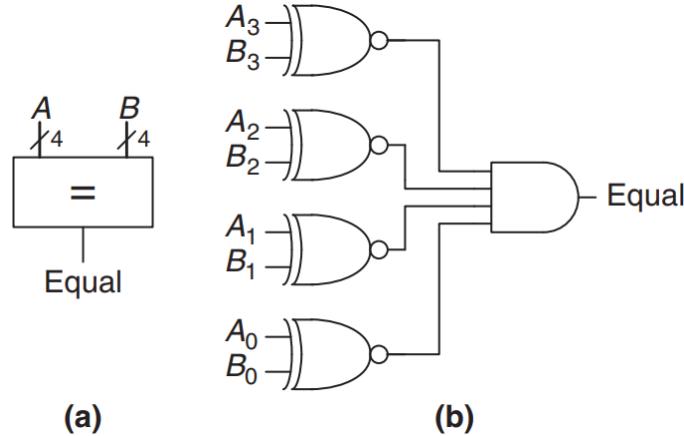


Figure 1.2.5: A 4-bit equality comparator's (a) symbol and (b) schematic

- (b) A *magnitude comparator* produces one or more outputs indicating the relative values of A and B . It checks whether the corresponding bits in each column of A and B are equal using XOR gates. The numbers are equal if all of the columns are equal. Figure 1.2.6 shows the symbol of an N -bit magnitude comparator.

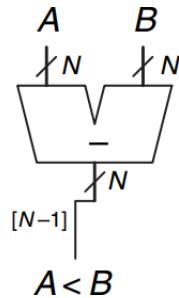
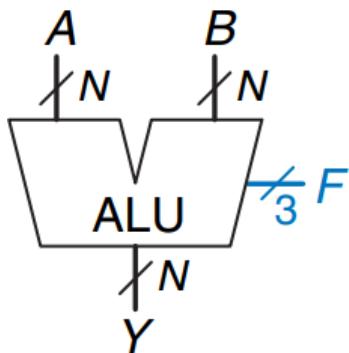


Figure 1.2.6: An N -bit magnitude comparator

4. ALU

Arithmetic/Logical Unit (ALU) combines a variety of mathematical and logical operations into a single unit. For example, a typical ALU might perform addition, subtraction, magnitude comparison, AND, and OR operations. The ALU forms the heart of most computer systems.

Figure 1.2.7 shows the symbol for an N -bit ALU with N -bit inputs and outputs. The ALU also receives a control signal F that specifies which function to perform.



$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND \bar{B}
101	A OR \bar{B}
110	A - B
111	SLT

Figure 1.2.7: An ALU's symbol and operations

5. Shifters and Rotators

Shifters and *rotators* move bits and multiply or divide by powers of 2. Shifters shift a binary number left or right by a specified number of steps. *Logical shifter* and *Arithmetic shifter* are two common used shifters:

- (a) Logical shifter shifts the number to the left (LSL) or right (LSR) and fills empty spots with 0's. For example, 11001 LSR 2 = 00110; 11001 LSL 2 = 00100.
- (b) Arithmetic shifter works similarly to the logical shifter, but **on right shifts fills the most significant bits with the copy of the old most significant bit (MSB)**. This is useful for multiplying and dividing signed numbers. **Arithmetic shift left is the same as logical shift left**. For example, 11001 ASR 2 = 11110; 11001 ASL 2 = 00100.

Rotators rotate number in circle such that empty spots are filled with bits shifted off the other end. For example, 11001 ROR 2 = 01110; 11001 ROL 2 = 00111.

The operators $<<$, $>>$, and $>>>$ indicate shift left, logical shift right, and arithmetic shift right, respectively. An N -bit shifter can be built from N $N:1$ multiplexer.

A left shift is a special case of multiplication: **a left shift by N bits multiplies the number by 2^N** . For example, $000011_2 << 4 = 110000_2$ is equivalent to $3_{10} \cdot 2^4 = 48_{10}$.

An arithmetic right shift is a special case of division: **an arithmetic right shift by N bits divides the number by 2^N** . For example, $11100_2 >>> 2 = 1111_2$ is equivalent to $-4_{10} \div 2^2 = -1_{10}$.

6. Multiplication

Multiplication of binary numbers is similar to decimal multiplication but involves only 0's and 1's. The *partial products* are formed by multiplying a single digit of the multiplier with the entire multiplicand, as illustrated by Figure 1.2.8.

$ \begin{array}{r} 230 \\ \times 42 \\ \hline 460 \\ + 920 \\ \hline 9660 \end{array} $	<p style="margin: 0;">multiplicand multiplier partial products result</p>	$ \begin{array}{r} 0101 \\ \times 0111 \\ \hline 0101 \\ 0101 \\ 0101 \\ + 0000 \\ \hline 0100011 \end{array} $
---	--	---

$$230 \times 42 = 9660$$

$$5 \times 7 = 35$$

Figure 1.2.8: Decimal and binary multiplications are performed in the same way

7. Division

Binary division can be performed using the following algorithm for N -bit unsigned numbers in range $[0, 2^{N-1}]$:

```

R' = 0
for i = N-1 to 0
    R = {R' << 1, Ai}
    D = R - B
    if D < 0 then   Qi = 0, R' = R    // R < B
    else           Qi = 1, R' = D    // R ≥ B
    R = R'

```

The *partial remainder* R is initialized to 0. The most significant bit of the dividend A then becomes the least significant bit of R . The divisor B is repeatedly subtracted from this partial remainder to determine whether it fits. If the difference D is negative (i.e. the sign bit of D is 1), then the quotient bit Q_i is 0 and the difference is discarded. Otherwise, Q_i is 1, and the partial remainder is updated to be the difference. In any event, the partial remainder is then doubled (left-shifted by one column), the next most significant bit of A becomes the least significant bit of R , and the process repeats. The result satisfies $\frac{A}{B} = Q + \frac{R}{B}$.

1.3 Number Systems

Computers operate not only on unsigned and signed integers but also on fixed- and floating-point numbers that represent non-integers.

1. Fixed-Point Number Systems

Fixed-point notation has an implied *binary point* between the integer and fraction bits, analogous to the decimal point between the integer and fraction digits of an ordinary decimal number. The implied binary point usually divides the binary number into two equal chunks of bits. The more significant chunk (the chunk on the left) can be converted to decimal using powers of 2 with positive exponents. Conversely, the less significant chunk (the chunk on the right) can be converted to decimal using powers of 2 with negative exponents. Figure 1.3.1 shows a conversion process of an 8-bit fixed-point binary number to a decimal real number.

$$\begin{aligned}
 & 01101100_2 \\
 = & 0110.1100_2 \\
 = & 2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75_{10}
 \end{aligned}$$

Figure 1.3.1: Fixed-point notation of 6.75 with four integer bits and four fraction bits

Signed fixed-point numbers works for both two's complement or sign/magnitude notation. In sign/magnitude form, the most significant bit is used to indicate the sign. The two's complement representation is formed by inverting the bits of the absolute value and adding a 1 to the least significant (rightmost) bit. In the example in Figure 1.3.1, the least significant bit position is in the 2^{-4} column.

2. Floating-Point Number Systems

Floating-point numbers are analogous to scientific notation. They overcome the limitation of having a constant number of integer and fractional bits, allowing the representation of very large and very small numbers. Like scientific notation, floating-point numbers have a *sign*, *mantissa* (M), *base* (B), and *exponent* (E). For example, the number 4.1×10^3 is the decimal scientific notation for 4100. It has a mantissa of 4.1, a base of 10, and an exponent of 3. 32 bits can be used to represent 1 sign bit, 8 exponent bits, and 23 mantissa bits.

The IEEE floating-point standard has special cases to represent 0 (zero), $\pm\infty$ (positive and negative infinity), and NaN (Not a Number), as showed in Table 1.3.1

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000000000000
∞	0	11111111	00000000000000000000000000000000
$-\infty$	1	11111111	00000000000000000000000000000000
NaN	X	11111111	Non-zero

Table 1.3.1: IEEE 754 floating-point notations for 0, $\pm\infty$, and NaN

A typical 32-bit floating-point number has 1 sign bit, 8 exponent bits, and 23 fraction bits. This is called the single-precision format. There is also another format, called the *double-precision floating-point format*, (or *doubles*) which effectively provides a greater precision and greater range:

Format	Total Bits	Sign Bits	Exponent Bits	Fraction Bits
single	32	1	8	23
double	64	1	11	52

Table 1.3.2: Single- and double-precision floating-point formats

To add two floating-point numbers, follow these steps:

- (a) Extract exponent and fraction bits
- (b) Prepend leading 1 to form the mantissa
- (c) Compare exponent
- (d) Shift smaller mantissa if necessary
- (e) Add mantissas
- (f) Normalize mantissa and adjust exponent if necessary
- (g) Round result
- (h) Assemble exponent and fraction back into floating-point number

1.4 Sequential Building Blocks

1. Counters

An N -bit *binary counter*, shown in Figure 1.4.1, is a sequential arithmetic circuit with clock and reset inputs and an N -bit output Q .

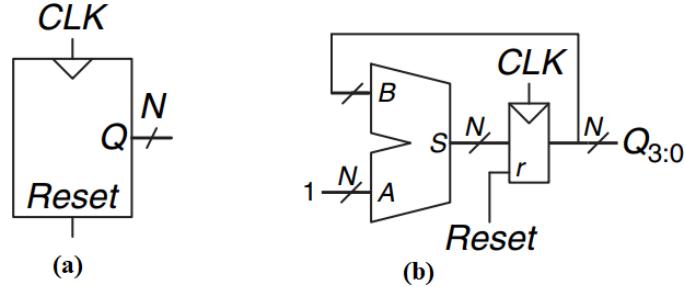


Figure 1.4.1: An N -bit counter's (a) symbol and (b) schematic

The reset initializes the output to 0. The counter then loops through all 2^N possible outputs in binary order, incrementing on every rising edge of the clock.

2. Shift Registers

A *shift register* has a clock, a serial input S_{in} , a serial output S_{out} , and N parallel outputs $Q_{N-1:0}$, as shown in Figure 1.4.2.

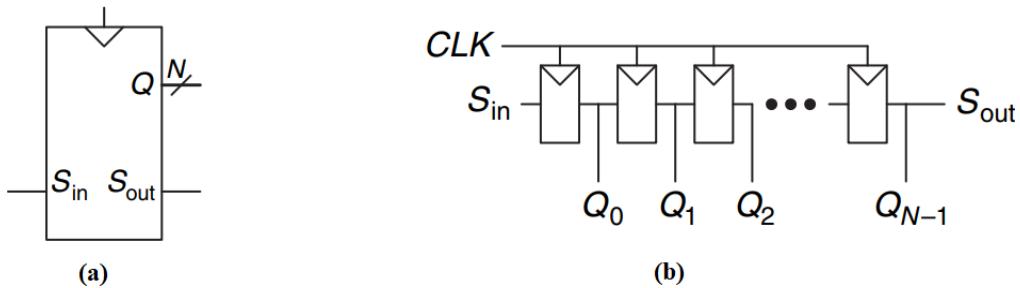


Figure 1.4.2: A shift register's (a) symbol and (b) schematic

A shift register can be constructed from N flip-flops connected in series. Some shift registers also have a reset signal to initialize the flip-flops.

1.5 Memory Arrays

The previous subsection introduced arithmetic and sequential circuits for manipulating data. Digital systems also require *memories* to store the data used and generated by such circuits. For instance, registers built from flip-flops are a kind of memory that stores small amounts of data. This subsection describes *memory arrays* that can efficiently store large amounts of information.

1. Overview

The memory is organized as a two-dimensional array of memory cells. The memory reads or writes the contents of one of the rows of the array. This row is specified by an *Address*. The value read or written is called *Data*. Each row of data is called a *word*. An array with N -bit addresses and M -bit data has 2^N rows and M columns and $2^N M$ -bit words. Figure 1.5.1 shows a generic symbol for a memory array with N address bits and M data bits.

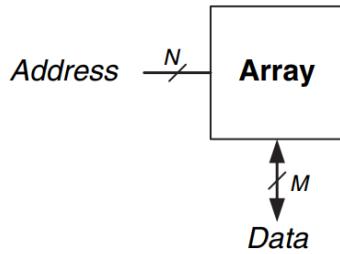


Figure 1.5.1: Generic memory array symbol

The *depth* of an array is the number of rows, and the *width* is the number of columns, also called the word size. The size of an array is given by $\text{depth} \times \text{width}$.

(a) Bit Cells

Bit cells are the fundamental building blocks of memory arrays. Each bit cell stores 1 bit of data. Figure 1.5.2 shows that each bit cell is connected to a *wordline* and a *bitline*.

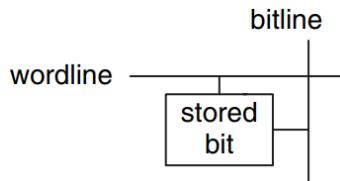


Figure 1.5.2: A bit cell

For each combination of address bits, the memory asserts a single wordline that activates the bit cells in that row. If the wordline is HIGH, the stored bit transfers to or from the bitline; otherwise, the bitline is disconnected from the bit cell.

To read a bit cell, the bitline is initially floating (Z). Then, the wordline is turned ON, allowing the stored value to drive the bitline to 0 or 1.

To write a bit cell, the bitline is strongly driven to the desired value. Then, the wordline is turned ON, connecting the bitline to the stored bit, which effectively writes the desired value to the bit cell.

(b) Organization

Figure 1.5.3 shows the symbol and the function of a 4×3 memory array. Figure 1.5.4 shows the internal organization of this memory array.



Figure 1.5.3: A 4×3 memory array's (a) symbol and (b) function

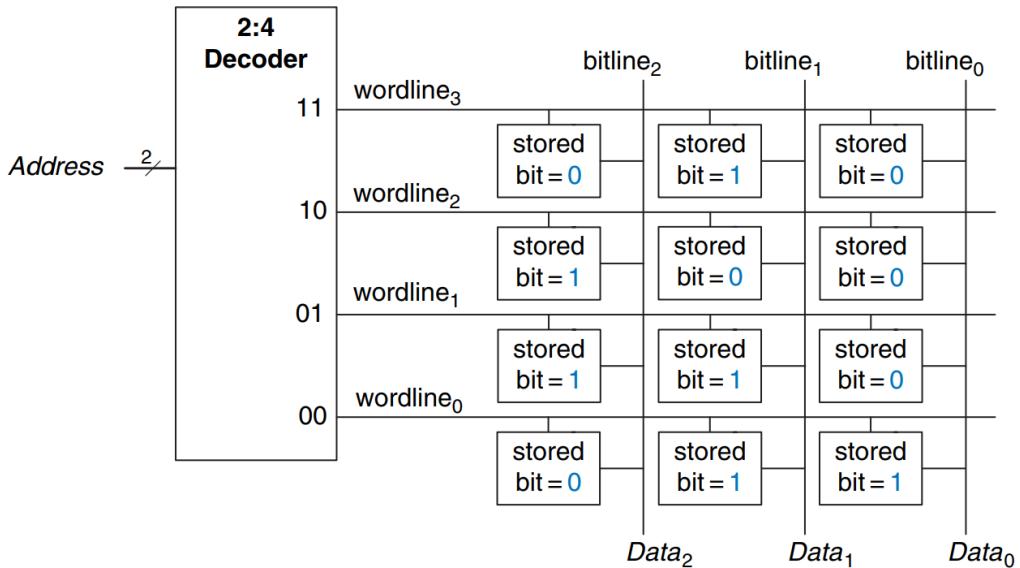


Figure 1.5.4: A 4×3 memory array's organization

The organization of memory array—in this case, a 4×3 memory array—is quite easy to understand: for each wordline from the most significant bit (top) to the least significant bit (bottom), there is an array of bit cells, each connected to a bitline from the most significant bit (left) to the least significant bit (right).

During a memory read, a wordline is asserted, and the corresponding row of bit cells drives the bitline HIGH or LOW. During a memory write, the bitlines are driven HIGH or LOW first, and then a wordline is asserted, allowing the bitline values to be stored in that row of bitcells.

Using Figure 1.5.4 for examples. To read data from *Address* 10, the bitlines are first floating; the decoder asserts wordline₂, and the data stored inside the bitcells reads out onto the corresponding *Data* bitlines: 100. To write, say, the value 001 to *Address* 11, bitline₂, bitline₁, and bitline₀ are driven to 0, 0, and 1, respectively; wordline₃ is then asserted, and the values from the bitlines are written into the corresponding bitcells.

(c) Memory Ports

Memory ports are parts of memories which gives read and/or write access to one memory address.

Multipported memories can access several addresses simultaneously.

(d) Memory Types

Memory types are specified by their size ($depth \times width$) and the number and type of ports. Memories are classified based on how they store bits in the bit cell. The following list classifies the broadest types of memory and briefly how they work.

- **Random access memory (RAM)** is *volatile*. Loses data without power.
 - *Dynamic RAM (DRAM)* stores data as a charge on a capacitor.
 - *Static RAM (SRAM)* stores data using a pair of cross-coupled inverters.
- **Read only memory (ROM)** is *nonvolatile*. Retains data indefinitely even without power.

2. Dynamic Random Access Memory (DRAM)

Dynamic RAM (DRAM) stores a bit as the presence or absence of a charge on a capacitor. Figure 1.5.5 shows a DRAM bit cell. The bit is stored on a capacitor. The nMOS transistor acts like a switch that either connects or disconnects the capacitor from the bitline. When the wordline is asserted, the nMOS transistor turns ON, and the stored bit value transfers to or from the bitline.

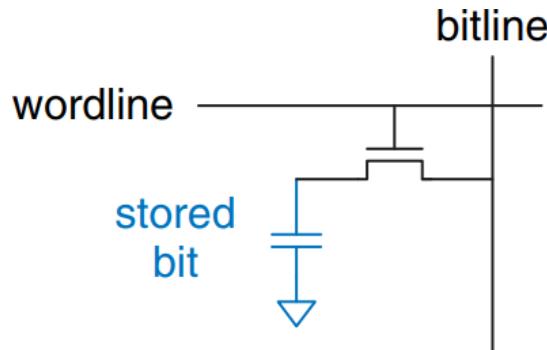


Figure 1.5.5: A DRAM bit cell

Like a generic memory array, upon a read, data values are transferred from the capacitor to the bitline; upon a write, data values are transferred from the bitline to the capacitor.

Reading destroys the bit value stored on the capacitor, so the data word must be restored (rewritten) after each read. Even when DRAM is not read, the contents must be refreshed (read and rewritten) every few milliseconds, because the charge on the capacitor gradually leaks away.

3. Static Random Access Memory (SRAM)

Static RAM (SRAM) is static because stored bits do not need to be refreshed. SRAM stores data on a cross-coupled inverters, as shown in Figure 1.5.6 below:

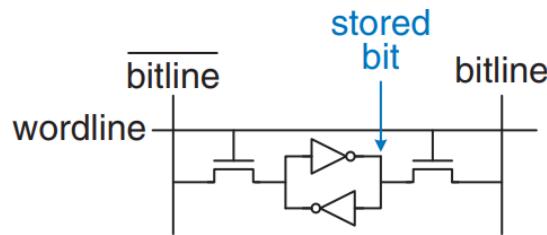


Figure 1.5.6: A SRAM bit cell

Each cell has two outputs, bitline and $\overline{\text{bitline}}$. When wordline is asserted, both nMOS transistors turn on, and the data is transferred to or from the bitlines. Unlike DRAM, the cross-coupled inverters restore the value of the stored bit.

4. Area and Delay

Flip-flops, DRAM, and SRAM are all volatile memories, but each has different area and delay characteristics. Table 1.5.1 compares these three types of memory.

Memory Type	Transistors per Bit Cell	Latency
Flip-flop	≈ 20	Fast
SRAM	6	Medium
DRAM	1	Slow

Table 1.5.1: Memory comparison between flip-flops, SRAM, and DRAM

Flip-flops is the fastest among the three, but it costs the most to build. SRAM is in the middle tier with a medium speed and a decent cost. DRAM is the cheapest but slowest of the three.

Thus, there is always a tradeoff between area and delay in memory.

5. Register Files

Register files are a group of registers that store temporary variables. They are often built as a small, multiported SRAM array.

6. Read Only Memory

Read only memory (ROM) stores a bit as the presence or absence of a transistor, as illustrated by Figure 1.5.7 below:

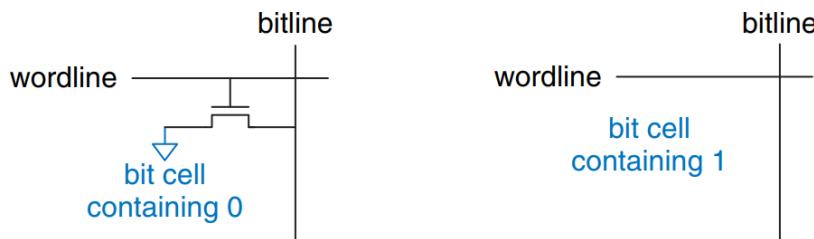


Figure 1.5.7: A ROM bit cell

To read a cell, the bitline is *weakly* pulled HIGH. Then, the wordline is turned ON. If the transistor is present, it pulls the bitline LOW; otherwise, the bitline remains HIGH. Despite their names, most modern ROM can be written.

7. Logic Using Memory Arrays

Memory arrays can perform combinational logic functions despite their primary usage for data storage. Memory arrays used to perform logic are called *lookup tables (LUTs)*. Using memory to perform logic, the user can look up the output value for a given input combination (address). Each address corresponds to a row in the truth table, and each data bit corresponds to an output value.

1.6 Logic Arrays

Like memory, gates can be organized into regular arrays called *logic arrays*. These logic arrays can perform any function without the user having to connect wires in specific ways because they are made programmable. Most logic arrays are also configurable, which means the designer can modify them programmatically without replacing the hardware.

1. Programmable Logic Array

Programmable logic arrays (PLAs) implement two-level combinational logic in sum-of-products (SOP) forms. PLAs are built from an AND array followed by an OR array, as shown in Figure 1.6.1

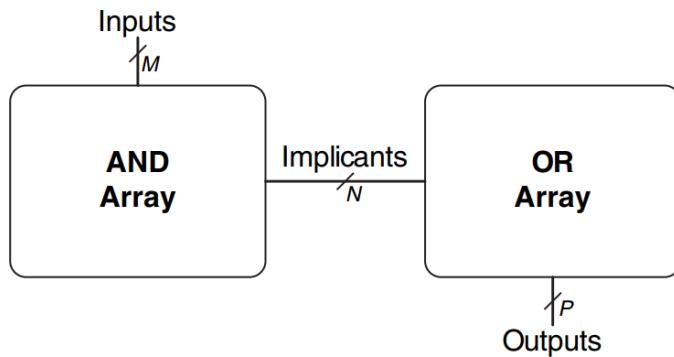
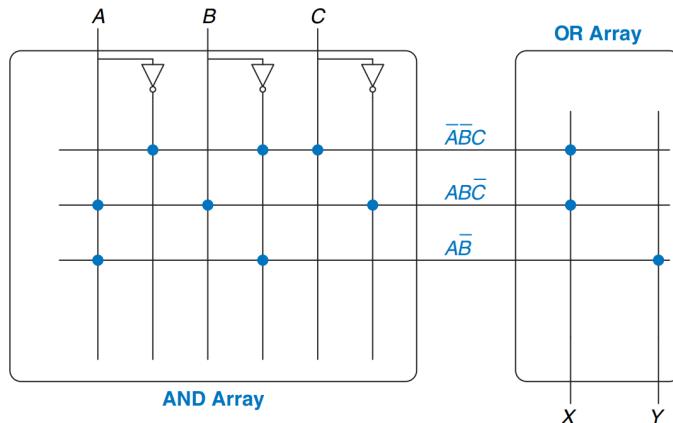


Figure 1.6.1: A $M \times N \times P$ -bit PLA

M inputs drive an AND array, which produce N implicants. The implicants then drive the OR array, which produces a P outputs. A dot notation of a friendly example of a $3 \times 3 \times 2$ -bit PLA is as follows:



In this dot notation, each row of dots is an AND gate, and the dots on the corresponding row specifies the input to that AND gate.

2. Field Programmable Gate Array

Since PLAs are the older technology which can only perform combinational logic functions, the newer technology creates *field programmable gate arrays (FPGA)* which can perform both combinational logic and sequential logic. The logic gates of an FPGA are reconfigurable through software.

The FPGAs are more powerful and flexible than PLAs for various reasons:

- Implement both combinational and sequential logic (versus only combinational logic functions for PLAs)
- Implement multilevel logic functions (versus two-level logic functions for PLAs)
- Built-in multipliers
- High-speed I/Os
- Data converters using analog-to-digital converters (ADC)
- Large RAM arrays
- Better processors.

FPGAs are built as an array of configurable *logic elements (LEs)*, also referred to as *configurable logic blocks (CLBs)*. Each LE can be configured to perform combinational and sequential logic. The LEs are surrounded by *input/output elements (IOEs)* for interfacing with the world.

3. Array Implementation

To minimize the size and cost, ROMs and PLAs usually use pseudo-nMOS or dynamic circuits instead of conventional logic gates. For example, Figure 1.6.2 compares the implementation of a ROM using the dot notation (conventional logic gate) with a pseudo-nMOS implementation:

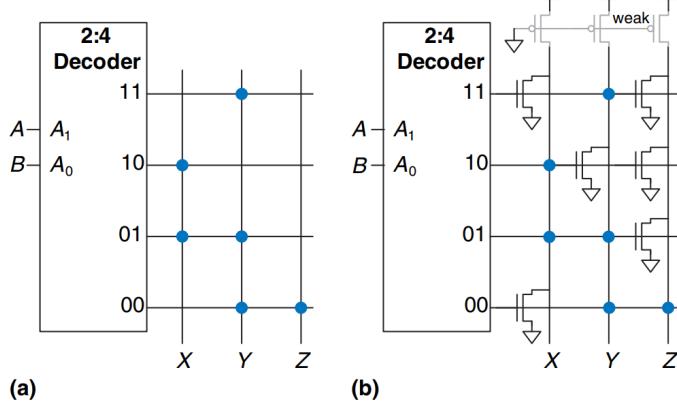


Figure 1.6.2: ROM implementations: (a) dot notation, (b) pseudo-nMOS circuit

An nMOS transistor is placed at the junction without a dot in the array. Note that in pseudo-nMOS circuits, the weak pMOS transistor pulls the output HIGH *only if* there is no path to GND through the pulldown (nMOS) network.

1.7 Summary

This chapter introduced digital building blocks used in many—if not most—digital systems. These blocks include arithmetic circuits such as adders, subtractors, comparators, multipliers, dividers, and shifters; sequential building blocks such as counters and shift registers; and arrays for memory and logic. The number systems are also covered sufficiently throughout this chapter, such as fixed- and floating-point numbers and how to represent them using binary bits.

Adders form the basis of most arithmetic circuits. A half adder adds two inputs, A and B , and produces a sum S and a carry out C_{out} . A works just like the half adder but has an additional input—the carry in C_{in} . N full adders can be chained together to form a carry propagate adder (CPA) that adds two N -bit numbers. This type of CPA is called the ripple-carry adder since the carry ripples from one adder to another. CPAs can be made faster by the lookahead and the prefix techniques.

A subtractor is just an adder where the second input is negated by an inverter, and it takes in an additional $C_{in} = 1$ due to the property of two's complement.

Fractional numbers can be represented by fixed- and floating-point forms. Fixed-point numbers are analogous to decimals, and floating-point numbers are analogous to scientific notation. Fixed-point numbers use ordinary arithmetic circuits, whereas floating-point numbers require more complicated hardware to extract and process the sign, exponent, and mantissa of the notation.

Large memories can be organized into arrays of words. There are two common types of memories: RAM (volatile) and ROM (nonvolatile). RAMs are further classified into SRAM (static RAM), which use cross-coupled inverters to store the data, and DRAM (dynamic RAM), which use a charge on a capacitor to store the data. ROM stores the data by the presence or absence of the transistors in the bit cells. Despite their names, most modern ROM can be written.

Arrays can be used to not only store memory but also build logic. Two common types of logic arrays are PLA and FPGA. PLAs are composed of dedicated connections between configurable AND and OR arrays (conventional logic), and they only implement combinational logic. FPGAs,

on the other hand, are newer and more advanced. Modern FPGAs are easy to reprogram and are large and cheap enough to build highly sophisticated digital systems.

2 Grey Box Exploration

1. The first blurb is on page 255, which states *Fixed-point number systems are commonly used for banking and financial applications that require precision but not a large range.*

The maximum the money in transaction is in trillions, which is *really* rare in real life. For a typical person in banking and finance, the amount of money they can transact is even smaller: millions or hundreds of thousands is the maximum the money can realistically get. In the US dollar system, the decimal part of the number is always two-digit long (0 cents–99 cents), and the integer part is—realistically—six- to seven-digit long at maximum. Each digit varies from 0 to 9, which can be represented by a 4-bit binary number. Therefore, in the banking and finance's point of view, the transaction money can be represented by a fixed-point binary number with 24 bits of integer part and 8 bits of decimal part.

2. The second blurb is on page 270, which states *Flash memory drives with Universal Serial Bus (USB) connectors have replaced floppy disks and CDs for sharing files because Flash costs have dropped so dramatically.*

Flash memory drives use electrically erasable programmable read only memory (EEPROM) to store information. When a USB connects to a computer, the type of ROM and the data values to program are determined. Thus, to read from and write to a USB, the computer uses firmware to manipulate the charge of the capacitors inside the ROM. On the other hand, floppy disks use magnetism to store data, and CDs use optics and mechanics to hold information. Without proper handling, CDs and floppy disks can be easily damaged, causing loss of information. Due to this disadvantage and the Flash memory's continually decreasing costs, Flash memory becomes popular and gradually replaces the other two.

3 Figures

Two figures were selected from this chapter for special recognition. Figure 3.0.1 was selected because it shows that a subtractor is just a modified adder where the second input is obtained by getting the two's complement of its original value: $-B = \overline{B} + 1$. Then, setting $C_{in} = 1$ to the adder to balance the equation. The $Y = A - B$ subtractor is constructed.

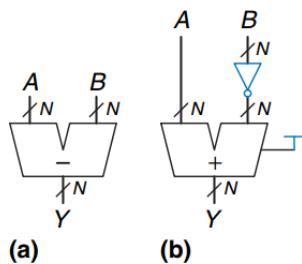


Figure 3.0.1: Subtractor: (a) symbol, (b) implementation

Figure 3.0.2 was selected because it shows the general layout of an FPGA: several logic elements (LEs) surrounded by input/output elements (IOEs). The LEs can be configured to perform combinational or sequential functions. The IOEs help the FPGA to interface with the outside world. This layout also shows that the FPGA can be flexibly programmed to construct sophisticated digital systems.

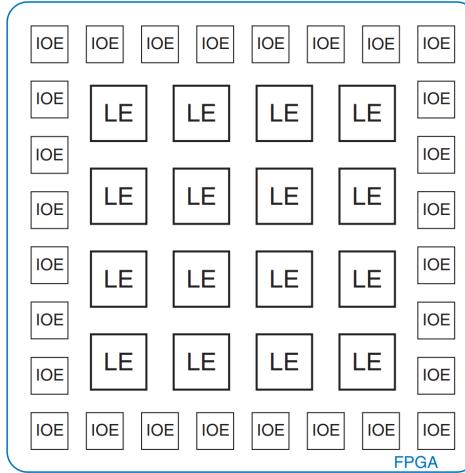


Figure 3.0.2: General FPGA layout

4 Example Problems

See the attached images on the next three pages.

EXAMPLE PROBLEMS

Example 5.1: Ripple-carry adder and carry-lookahead adder delay (page 243)

32-bit ripple-carry adder 32-bit carry-lookahead adder
with 4-bit blocks. $t_{pg} = 100\text{ ps}$, $t_{FA} = 300\text{ ps}$, $N=32$

$$t_{\text{ripple}} = N t_{FA} = 32 \cdot 300\text{ ps} = 9600\text{ ps} = 9.6\text{ ns}$$

$$\begin{aligned} t_{\text{CLA}} &= t_{pg} + t_{pg\text{-block}} + \left(\frac{N-1}{k}\right) t_{\text{AND-OR}} + k t_{FA} \\ &= 100\text{ ps} + (6 \cdot 100\text{ ps}) + \left(\frac{32}{4} - 1\right) (2 \cdot 100\text{ ps}) + (4 \cdot 300\text{ ps}) \\ &= 3.3\text{ ns} \end{aligned}$$

Thus, the carry-lookahead adder is almost 3 times faster than the ripple-carry adder.

Variation: $N=64$, $k=8$, $t_{pg}=50\text{ ps}$, $t_{FA}=200\text{ ps}$.

$$t_{\text{ripple}} = N t_{FA} = 64 \cdot 200\text{ ps} = 12.8\text{ ns}$$

$$\begin{aligned} t_{\text{CLA}} &= t_{pg} + t_{pg\text{-block}} + \left(\frac{N-1}{k}\right) t_{\text{AND-OR}} + k t_{FA} \\ &= 50\text{ ps} + (6 \cdot 50\text{ ps}) + \left(\frac{64}{8} - 1\right) (2 \cdot 50\text{ ps}) + (4 \cdot 200\text{ ps}) \\ &= 1.85\text{ ns} \end{aligned}$$

Example 5.2: Prefix adder delay.

$N=32$, $t_{pg}=100\text{ ps}$, $t_{pg\text{-prefix}}=200\text{ ps}$

$$\begin{aligned} t_{PA} &= t_{pg} + \log_2 N (t_{pg\text{-prefix}}) + t_{XOR} \\ &= 100\text{ ps} + \log_2(32) \cdot 200\text{ ps} + 100\text{ ps} = 4.2\text{ ns} \end{aligned}$$

Variation $N=64$

$$\begin{aligned} t_{PA} &= t_{pg} + \log_2 N (t_{pg\text{-prefix}}) + t_{XOR} \\ &= 100\text{ ps} + \log_2(64) \cdot 200\text{ ps} + 100\text{ ps} = 4.4\text{ ns} \end{aligned}$$

Example 5.3. Set less than.

$$N = 32, A = 25_{10}, B = 32_{10}$$

Since $A < B$, $Y_{\text{expected}} = 1$. For SLT, $F_{2:0} = 111$.

$$S = A - B = 25_{10} - 32_{10} = -7_{10} = \underbrace{1111\dots}_{32 \text{ bits}} \underbrace{1001_2}_2$$

with $F_{1:0} = 11$, the final multiplexer sets $Y = S_{31} = 1$

Variation: $N = 64, A = 96_{10}, B = 69_{10}$

Since $A > B$, $Y_{\text{expected}} = 0$. For SLT, $F_{2:0} = 111$

$$S = A - B = 96_{10} - 69_{10} = 27_{10} = \underbrace{0000\dots}_{32 \text{ bits}} \underbrace{11011_2}_2$$

with $F_{1:0} = 11$, the final multiplexer sets $Y = S_{31} = 0$

Example 5.4. Arithmetic with fixed-point numbers.

Compute $0.75 + -0.625$ using fixed-point numbers

$$0.625 \xrightarrow{-2^{-1}} 0.125 \xrightarrow{-2^{-3}} 0 \Rightarrow 0.625_{10} = 0000.1010_2 \begin{array}{r} \\ \xrightarrow{-1} \\ 1111.0101_2 \\ + \\ \hline 1111.0110_2 \end{array}$$

$$\Rightarrow -0.625_{10} = 1111.0110_2$$

$$0.75 \xrightarrow{-2^{-1}} 0.25 \xrightarrow{-2^{-2}} 0 \Rightarrow 0.75_{10} = 0000.1100_2$$

$$\text{Thus, } 0.75_{10} + -0.625_{10} = \begin{array}{r} 0000.1100_2 \\ + 1111.0110_2 \\ \hline 1111.0010_2 \end{array} = 0.125_{10}$$

Variation: $1.5 + 7.5 = ?$

$$1.5 \xrightarrow{-2^0} 0.5 \xrightarrow{-2^{-1}} 0 \Rightarrow 1.5_{10} = 0001.1000_2$$

$$7.5 \xrightarrow{-2^2} 3.5 \xrightarrow{-2^{-1}} 1.5 \xrightarrow{-2^0} 0.5 \xrightarrow{-2^{-1}} 0 \quad 7.5_{10} = 0111.0111_2$$

$$\Rightarrow 7.5_{10} = 0111.1000_2$$

$$\text{Thus, } 1.5_{10} + 7.5_{10} = \begin{array}{r} 0001.1000_2 \\ + 0111.1000_2 \\ \hline 01001.0000_2 \end{array} = 9_{10}$$

Phi Luu

ECE-271-001-F2018

11/14/2018

Example 5.5: 32-bit floating point numbers

Show the floating-point representation of the decimal number 228

$$228_{10} = 11100100_2 = 1.11001_2 \cdot 2^7$$

1 bit	8 bits	23 bits
0	0000 0111	111 0010 0000 0000 0000 0000
Sign	Exponent	Mantissa

Variation: $6969_{10} = ?$

$$6969_{10} = 1101100111001_2 = 1.10110011100100000000 \cdot 2^{12}$$

1 bit	8 bits	23 bits
0	0000 1100	11011001110010000000000
Sign	Exponent	Mantissa

5 Glossary

All definitions were found from the Google search engine, typing "define arithmetic logic unit" for the first item.

1. Arithmetic Logic Unit (ALU)

noun:

- (a) A unit in a computer that carries out arithmetic and logical operations.

2. Comparator

noun:

- (a) a device for comparing a measurable property or thing with a reference or standard.

- an electronic circuit for comparing two electrical signals.
- something used as a standard for comparison.

3. Read Only Memory

noun:

- (a) [computing] memory read at high speed but not capable of being changed by program instructions.

4. Array

noun:

- (a) an impressive display or range of a particular type of thing.

- (b) an ordered series or arrangement.

- an arrangement of troops.
- [mathematics] an arrangement of quantities or symbols in rows and columns; a matrix.
- [computing] an indexed set of related elements.

- (c) [literary] elaborate or beautiful clothing.

- (d) [law] a list of jurors empaneled.

verb:

- (a) display or arrange (things) in a particular way.

- (b) dress someone in (the clothes specified).

- (c) [law] empanel (a jury).

5. Random Access

noun:

- (a) [computing] the process of transferring information to or from memory in which every memory location can be accessed directly rather than being accessed in a fixed sequence.

6 Interview Question

See the attached image on the next page.

Ques.

INTERVIEW QUESTION

Question 5.1: What is the largest possible result of multiplying two unsigned N-bit numbers?

$$\underbrace{0111\dots111}_N \times \underbrace{0111\dots111}_N = (2^{N-2})_{10} + (2^{N-3})_{10} + \dots + (2^2)_{10} + (2^1)_{10} + (2^0)_{10}$$

$$\times \quad \underbrace{0111\dots111}_N = (2^{N-2})_{10} + (2^{N-3})_{10} + \dots + (2^2)_{10} + (2^1)_{10} + (2^0)_{10}$$

$$\Rightarrow M = \text{result} = (2^{N-2} + 2^{N-3} + \dots + 2^2 + 2^1 + 2^0)(2^{N-2} + 2^{N-3} + \dots + 2^2 + 2^1 + 2^0)$$

$$M = (2^{N-1} - 1)(2^{N-1} - 1)$$

$$M = (2^{N-1} - 1)^2$$

$$M = 2^{2(N-1)} - 2 \cdot 2^{N-1} \cdot 1 + 1^2$$

$$\boxed{M = 2^{2N-2} - 2^N + 1}$$

7 Reflection

I learn a lot of new things from this chapter. I has had a little familiarity with arithmetic circuits, but this chapter really expands my knowledge by diving deep into the circuit and logic aspects of various types of arithmetic circuits.

The content of this chapter is quite easy to grasp since I think it is closely related to mathematics and logic. What blows my mind the most is the method to organize memory into arrays and cells. The organization method seems very simple but also elegant.

I have been using an FPGA for the lab version of this course, and now I finally learn what FPGAs are made of. I also learn that we can write information onto *read only* memory despite its name.

8 Questions for Lecture

1. How can we write information onto the read only memory?
2. Are there better ways to organize memories than using arrays?
3. What storage technology do solid-state drives (SSDs) use such that it makes writing and reading information so fast?