**Week 1 - Number representation, logic gates, boolean algebra and adders**

**Number representation**

There are many ways to represent a number. The most common representation is **decimal** or **base-10**. In this system, each digit in a number can be one of 10 **numerals** (0 to 9). In a decimal number, a digit at position n represents the product of that digit with . The number in question is the sum of all these products. For example: 12345 can be understood as .

In computer science, it is useful to represent numbers not in base-10 but **base-2 (binary)**. In this system, each digit can be either 0 or 1, corresponding to the two possible states of an electric signal (on and off). Each digit of a binary number represents the product of that digit with a power of 2 instead of 10. Binary numbers are denoted with the subscript 2 (01012).

Another useful representation is **base-16 (hexadecimal)**. In this system, each digit takes one of 16 numerals (0-9, A = 10, B = 11, C = 12, D = 13, E = 14, F = 15). Each digit of a hexadecimal number represents the product of that digit with a power of 16. Hexadecimal numbers are denoted with the subscript 16 (1C16) or with the prefix 0x (0x1C).

In general, a base-n number is defined and denoted in a similar way.

Multiplying a base-n number by n:

Observation

Multiplying a binary number by 210 (or 102) just involves adding a zero to the end of it.  
Multiplying a hexadecimal number with 1610 (or 1016) just involves adding an extra zero to the end of it.

Law

***Multiplying a base-n number with n just involves adding an extra zero to the end of it.***

Intuition

Perform the multiplication in the same way as with decimal numbers. In this example, a binary number is multiplied by 2 in binary.

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

Proof

Let be a base-n number with k digits. Denote ik the kth digit from left to right.  
By definition:  
  
Multiplying both sides by n yields:  
  
Where is an additional term that does not affect the sum.  
By definition, is a base-n number with digits. The first n digits correspond to the digits of while the last digit is 0. In other words, is just with an extra zero at the end.

**Conversion between bases**

In order to convert from **binary or hexadecimal to decimal**, we rely on the definition of binary and hexadecimal numbers. We calculate the decimal result by summing the products of every digit with their corresponding powers of 2 or 16.

In order to convert from **decimal to binary**, we repeatedly divide the decimal number by 2, collecting the remainders until the quotient reaches 0. The digits of the binary result are the reversed list of remainders.

Example: Convert 1671 to binary.

In order to convert from **decimal to hexadecimal**, we perform the same operation as in converting from decimal to binary, however this time dividing by 16 instead of 2. The remainders should be converted to their corresponding numerals in base-16.

Example: Convert 1671 to hexadecimal.

In order to convert from **binary to hexadecimal**, we go from right to left, converting each non-overlapping set of 4 binary digits into its corresponding hexadecimal numeral. To convert from **hexadecimal into binary**, we do the reverse, converting each base-16 digit into four binary digits.

Example: Convert 11101011011112 to hexadecimal.

| Binary  ↕  Hexadecimal | 0001 | 1101 | 0110 | 1111 |
| --- | --- | --- | --- | --- |
| 1 | D | 6 | F |

**Information in computers, logic gates**

In computers, information is stored in the form of **bits**, which can be either zero or one. By itself, each bit is not very useful. However, bits can be manipulated and combined in different operations to produce useful results.

**Logic gates** are a physical mechanism that manipulates bits in the computer. There are three basic logic gates, each with its own **truth table**, which lists the sets of inputs and outputs that gate can take.

**The NOT gate**

Takes one input and inverts its value. Also called an inverter.

| **Input** | **Output** |
| --- | --- |
| 0 | 1 |
| 1 | 0 |

**The AND gate**

Takes two inputs and outputs 1 only if both inputs are 1. Otherwise, outputs 0.

| **Input 1** | **Input 2** | **Output** |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**The OR gate**

Takes two inputs and outputs 0 only if both inputs are 0. Otherwise, outputs 1.

| **Input 1** | **Input 2** | **Output** |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

From these three basic gates, we can compose other gates.

**The NAND gate**

Is the combination of one NOT and one AND gate. Takes two inputs and outputs 0 only if both inputs are 1. Otherwise, outputs 1. Any logic gate can be recreated with a combination of NAND gates.

| **Input 1** | **Input 2** | **Output** |
| --- | --- | --- |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**The NOR gate**

Is the combination of one NOT and one OR gate. Takes two inputs and outputs 1 only if both inputs are 0. Otherwise, outputs 0.

| **Input 1** | **Input 2** | **Output** |
| --- | --- | --- |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

**The XOR gate (Exclusive-OR)**

Can be constructed from various combinations of the three basic logic gates, the most popular of which includes two NOT gates, two AND gates and one OR gate. Takes two inputs and outputs 1 if either input is 1 but not both. Otherwise, outputs 0.

| **Input 1** | **Input 2** | **Output** |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Boolean algebra**

**Boolean algebra** is a branch of algebra in which the values of variables are true and false (denoted 1 and 0 respectively). The main operations in boolean algebra are **conjunction** (and), **disjunction** (or) and **negation** (not), rather than addition and multiplication in elementary algebra.

In digital circuit design, boolean algebra can be used to express the relationship between the different inputs and outputs of a circuit with mathematical notation. The laws of boolean algebra also help to simplify circuit expressions, allowing designers to more easily design their circuits and find equivalents.

The three main operations in boolean algebra correspond to the three basic logic gates, and their notation is as follows:

Conjunction (AND): or   
Disjunction (OR): or   
Negation (NOT): or   
Additionally, the notation for exclusive-or (XOR):

The latter notation for each operation is preferred in circuit design.

Based on the truth tables of the logic gates, the following laws can be inferred:

Identity of AND, OR:

Annihilation of AND, OR:

Associativity of AND, OR:

Commutativity of AND, OR:

Idempotence of AND, OR:

Distributivity of AND over OR:

Distributivity of OR over AND:

Absorption:

Complementation:

Double negation (NOT NOT):

De Morgan’s laws:  
 (NAND decomposition)  
 (NOR decomposition)

Additionally:  
 (XOR decomposition)

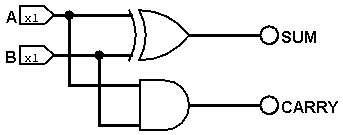
**Half adder**

Given that all decimal numbers can be represented in binary format, it might be useful to implement a circuit that adds numbers together. The basic building block of such a circuit is the **half adder**, which adds two bits together to produce a 1-bit **sum** output and a 1-bit **carry** output.

| **Input A** | **Input B** | **Sum** | **Carry** |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

*Half adder truth table*

It’s clear from the truth table that the SUM output column corresponds to the output column of the XOR gate while the CARRY output column corresponds to that of the AND gate. The half adder can be implemented as follows:



*A half adder*

**Full adder**

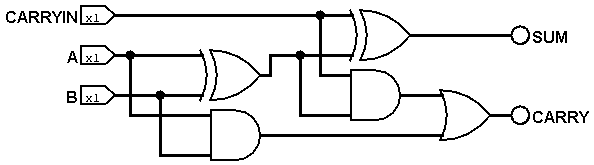
When adding corresponding digits of two binary numbers together, one might have to include the carry from the previous addition. To cater for this extra bit, a **full adder** is needed, which adds three digits together and produces a 1-bit sum output and 1-bit carry output.

The basic idea of a full adder is to add two input binary digits with a half adder to produce a sum and carry. This sum is then added to the input carry with another half adder to produce the final sum and another carry. Since both output carries cannot be simultaneously 0, as proven below, we take whatever is 1 with the OR gate.

*Proof that both carries of the full adder cannot be 1:*

Assume that the carry from the addition of A and B is 1. By the truth table of the half adder, the sum of A and B must be 0.

If the carry from the addition of the input carry and the sum of A and B is 1, by the truth table of the half adder, the sum of A and B must be 1 along with the input carry. This contradicts the assumption.



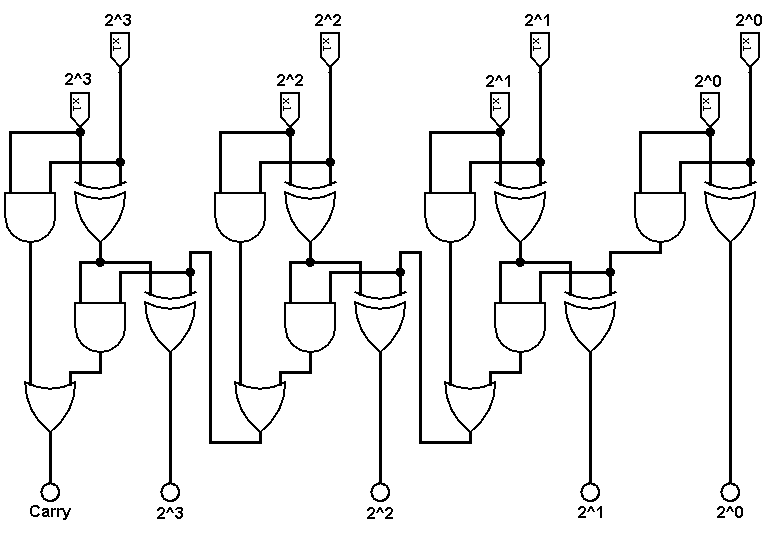
*A full adder*

| **Input A** | **Input B** | **Carry In** | **Sum Out** | **Carry Out** |
| --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

*Full adder truth table*

**4-bit adder**

Wiring up one half adder and three full adders together, feeding the carry output of the previous adder to the next, we get a **4-bit adder**, which can add two 4-bit numbers. For the least significant digits, only a half adder is needed because this is the first addition performed, therefore there is no carry input from the previous addition.



*A 4-bit adder*

**Week 2 - Programmable gates, latches and flip-flops**

**Programmable gates**

By itself, a truth table is just a list of all possible sets of inputs and outputs a logic gate can take. It is up to the circuit designer to interpret the table and find meaning from the numbers.

For instance, the AND gate can be interpreted as a **controlled buffer**, if one treats A as a control (condition) bit and B as a data bit. B can pass through only if A is 1 (true)

| **Input A (control)** | **Input B** | **AND Output** | **Action** |
| --- | --- | --- | --- |
| 0 | 0 | 0 | Block B |
| 0 | 1 | 0 |
| 1 | 0 | 0 | Output B |
| 1 | 1 | 1 |

As another example, the XOR gate can be interpreted as a **controlled inverter**, if one treats A as a control bit and B as the data bit. The value of B will be inverted if A is 1 (true).

| **Input A (control)** | **Input B** | **XOR Output** | **Action** |
| --- | --- | --- | --- |
| 0 | 0 | 0 | Keep B |
| 0 | 1 | 1 |
| 1 | 0 | 1 | Flip B |
| 1 | 1 | 0 |

The above interpretations of the AND and XOR gates illustrate the idea that logic gates offer the ability to program and control data flow with a control bit.

**Asynchronous circuits, synchronous circuits and clocks**

Sometimes, input signals to a circuit can arrive at different times, sending it into an illegal state or creating race conditions.

In order to synchronize a circuit’s activity, a **clock** is used. A clock is a device that generates a clock signal, which oscillates between a high and low state, often in the form of a square wave. Circuits that utilize a clock for synchronization are called **synchronous circuits**. Often, a synchronous circuit becomes active only at the rising or falling edge of the clock.

By contrast, **asynchronous circuits** are not regulated by a clock signal and their state changes immediately when the inputs change. Asynchronous circuits are generally faster than their synchronous counterparts but are much more unpredictable, error-prone and difficult to analyze.

**Latches and flip-flops**

**Latches** and **flip-flops** are circuits that can take one of two states (0 or 1). They are used to store one bit of data, forming the basic building blocks of memory. They can have one or more control inputs and produce one or two outputs.

Latches and flip-flops typically support three main operations: SET (put the state of the circuit to 1), RESET (put the state of the circuit to 0) and HOLD (keep the previous state of the circuit).

Historically, the term “flip-flop” has referred to both latches and flip-flops. However, in recent years, a distinction has been made between the two. Latches are **level-triggered**, which means that, as long as a control signal is asserted, their state updates immediately when their input signal changes. Flip-flops, on the other hand, are **edge-triggered**, which means that their state updates only on the brief rising or falling edge of a clock signal.

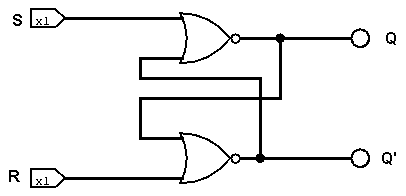
**SR Latches**

The most fundamental latch is the **SR Latch**. It takes two inputs, SET and RESET, hence the name SR, and produces two complementary outputs Q and Q’, where Q is the stored bit.

**

*SR latch/flip-flop symbol*

An SR Latch can be constructed from a pair of cross-coupled NOR gates as follows:

**

*An SR latch from NOR gates*

| **S** | **R** | **Qnext** | **Q’next** | **Action** |
| --- | --- | --- | --- | --- |
| 0 | 0 | Unchanged | Unchanged | HOLD |
| 0 | 1 | 0 | 1 | RESET |
| 1 | 0 | 1 | 0 | SET |
| 1 | 1 | ? | ? | NOT ALLOWED |

*SR latch truth table*

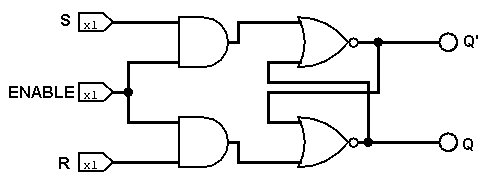
The state S=R=1 is forbidden because, due to the function of the NOR gates, both Q and Q’ would be 0, breaking the relation . Mathematically, no laws require that Q and Q’ be complements; however, in physical circuits, the issue with S=R=1 arises when both S and R are then set back to 0 at the same time (a power outage, for example). This time, Q and Q’ will race back to 1, and the winner will influence the loser, causing a race condition.

When the circuit is in its SET or RESET state, changing S or R respectively will not influence the outputs. We say that the circuit is **latched**.

Another construction of the SR latch involves two cross-coupled NAND gates. For this configuration, the state S=R=0 is the forbidden state, and S=R=1 is the HOLD state.

**Gated SR Latches and Gated D Latches**

By passing each input of the SR latch through an AND buffer along with a control bit E for ENABLE, we can choose whether the latch gets updated or not. This upgraded version of the SR latch is called the **gated SR latch**.

**

*A gated SR latch from NOR gates featuring an ENABLE pin*

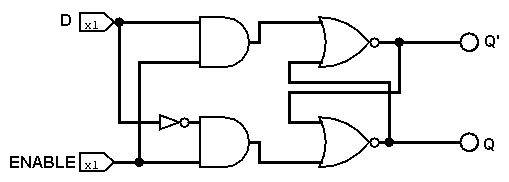
| **E (Enable)** | **Behavior** |
| --- | --- |
| 0 | Keep previous state (HOLD) |
| 1 | Like SR latch |

*Gated SR latch behavior*

The E control signal is used to control whether the inputs pass through or not. If E is low, both AND gates will produce 0, corresponding to the S=R=0 input of the HOLD state. If E is high, both AND gates will allow the values of S, R into the SR latch, and the latch behaves as normal.

E can sometimes be a clock signal (level-triggered).

The gated SR latch can be upgraded into the **gated D latch**, which takes only one input value and eliminates the forbidden state. It leverages the fact that for the two active input combinations of the latch (SET and RESET), S and R are complements. The inactive input combination of the latch (HOLD) happens as long as E is held low, regardless of S, R. Therefore, we can simply use one value (D) for S and invert it for R, preventing the S=R=1 combination.

**

*A gated D latch from NOR gates*

| **E** | **D** | **S** | **R** | **Qnext** | **Q’next** | **Action** |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | X | 0 | 0 | Unchanged | | HOLD |
| 1 | 1 | 1 | 0 | 1 | 0 | SET |
| 0 | 0 | 1 | 0 | 1 | RESET |

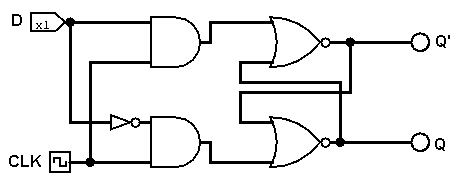
*Gated D latch truth table*

**D Flip-flop**

The **D Flip-flop** is very similar to the gated D latch but uses a clock signal instead of the control bit E. The D flip-flop is edge-triggered, meaning that it becomes active only on the rising or falling edge of the clock, depending on the configuration.

**

*D flip-flop symbol*

**

*A D flip-flop from NOR gates*

Note that the above circuit is not very accurate. At the moment, it is still level-triggered - the input changes immediately as long as the clock signal is 1, making it a gated D latch. The correct circuit should have the clock connected to a pulse detector to detect the rising/falling edge.

| **Clock** | **D** | **Qnext** | **Q’next** | **Action** |
| --- | --- | --- | --- | --- |
| Rising edge | 1 | 1 | 0 | SET |
| 0 | 0 | 1 | RESET |
| Non-rising | X | Unchanged | | HOLD |

*D flip-flop truth table, assuming positive edge-triggering*

**T Flip-flop**

The **T flip-flop** is a special kind of flip-flop which supports only two operations - HOLD and TOGGLE, which flips the current state. It takes an input T which enables or disables the toggle action and a clock signal. When the clock signal is on the rising/falling edge and T is high, the state of the flip-flop is flipped.

**

*T flip-flop symbol*

| **Clock** | **T** | **Qnext** | **Q’next** | **Action** |
| --- | --- | --- | --- | --- |
| Non-rising | X | Unchanged | | HOLD |
| Rising edge | 0 |
| 1 | Flipped | | TOGGLE |

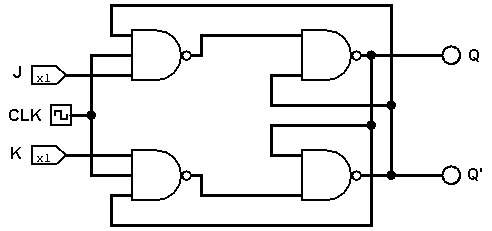
*T flip-flop truth table, assuming positive edge-triggering*

**JK Flip-flop**

The **JK flip-flop** is another upgrade to the SR latch that uses a clock. It also utilizes two inputs (J, K for S, R respectively) but adds another action for J=K=1: the TOGGLE action, which flips the current state. JK flip-flops are preferred over SR flip-flops because it eliminates the forbidden state.

**

*JK flip-flop symbol*

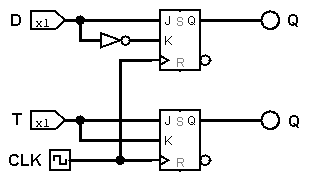
**

*A JK flip-flop constructed from NAND gates. There are many ways to wire up this circuit.*

| **Clock** | **J** | **K** | **Qnext** | **Q’next** | **Action** |
| --- | --- | --- | --- | --- | --- |
| Non-rising | X | X | Unchanged | | HOLD |
| Rising edge | 0 | 0 |
| 1 | 0 | 1 | 0 | SET |
| 0 | 1 | 0 | 1 | RESET |
| 1 | 1 | Flipped | | TOGGLE |

*JK flip-flop truth table, assuming positive edge-triggering*

The JK flip-flop is the universal flip-flop because it can be converted into an SR flip-flop, a D flip-flop and a T flip-flop.

**

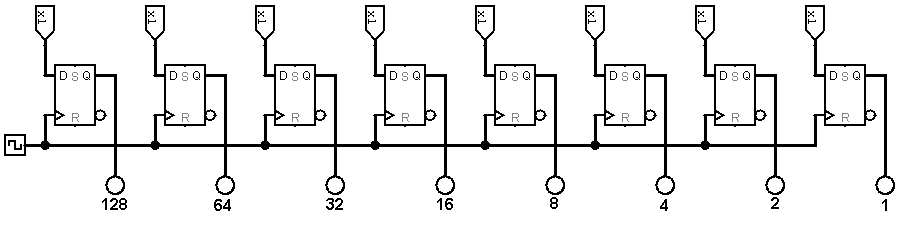
*A D flip-flop (above) and T flip-flop (below) simulated using two JK flip-flops sharing one clock.*

**Week 3 - Registers and counters**

**Register, word and endianness**

**Registers** are circuits made from multiple flip-flops (most commonly D flip-flops) that can store strings of bits. Registers are built into the CPU itself for storage of instructions and other data. Registers are **not** the building blocks of RAM.

The most basic register is built with multiple D flip-flops sharing the same clock. Each D flip-flop captures the value of its corresponding input signal on the clock pulse.

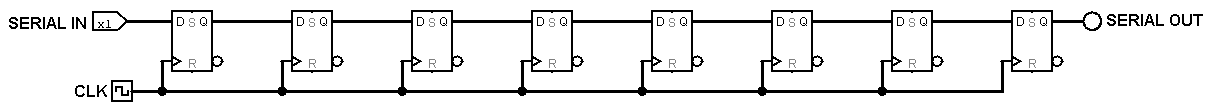
*A 8-bit big-endian register*

A **word** is the largest chunk of data a processor can handle at once (in computing or transferring data to and from memory). Most registers in a CPU are word-sized.

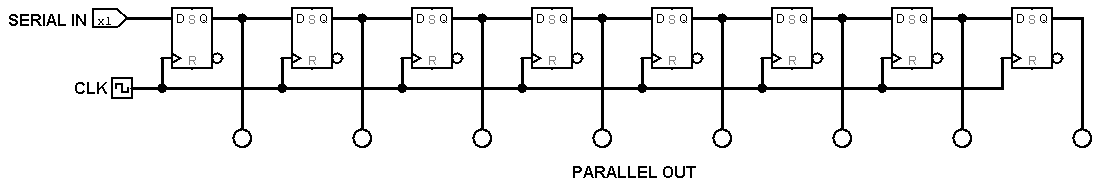
A word can consist of multiple bytes (conventionally the equivalent of 8 bits); therefore, the order in which these bytes are read matters. **Endianness** is a concept used to indicate the direction or order of bytes. If a system is **big-endian**, the most significant byte is stored on the lowest memory address. By contrast, if a system is **little-endian**, the least significant byte is stored on the lowest memory address. Typically, we read bytes from left to right with big-endian systems and from right to left with little-endian systems.

**Shift register**

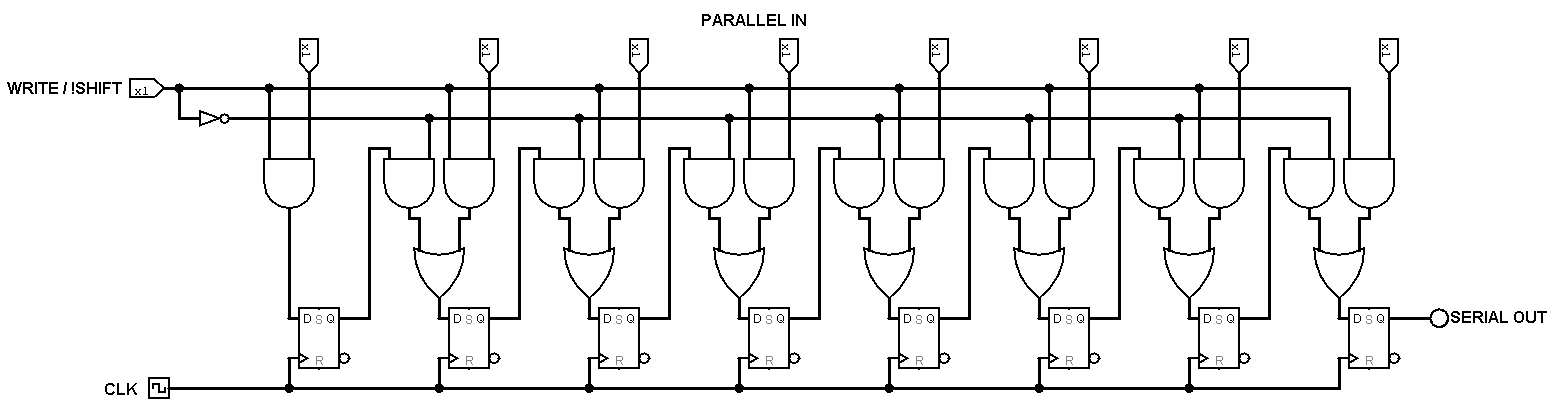
**Shift registers** are circuits whose purpose is to shift or move bits. They are composed of multiple D flip-flops driven by a common clock. On every clock pulse, bits are moved left or right in the circuit.

The input to and output from a shift register can be either **serial or parallel**. Serial means that data comes in and out of the circuit bit by bit while parallel means that data comes in and out of the circuit all at once.

*A 8-bit serial-in to serial-out shift register*



*A 8-bit serial-in to parallel-out shift register*

**

*A 8-bit parallel-in to serial-out shift register. When WRITE is high, data is fed in parallel to the flip-flops on every clock pulse. When WRITE is low, data is shifted right bit by bit on every clock pulse.*

Shift registers have many applications in computers. First, they help to move bits from higher/lower order to lower/high order. Second, they provide a method for multiplying and dividing by 2. Third, they give programmers a means to convert between serial and parallel data.

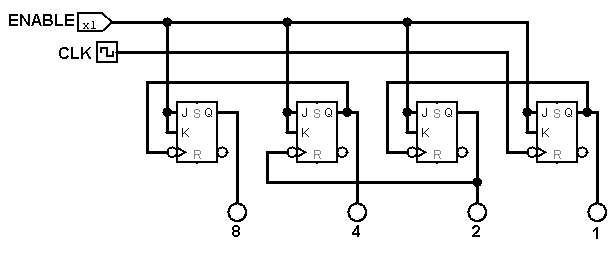
**Ripple counter**

**Counters** are circuits that can keep track of the number of times an event has occurred, often in relation to a clock.

The **ripple counter** is an asynchronous counter that works by utilizing the toggle setting of JK flip-flops. It is formed by multiple JK flip-flops chained together where the output of one flip-flop is fed as the clock input to the next.

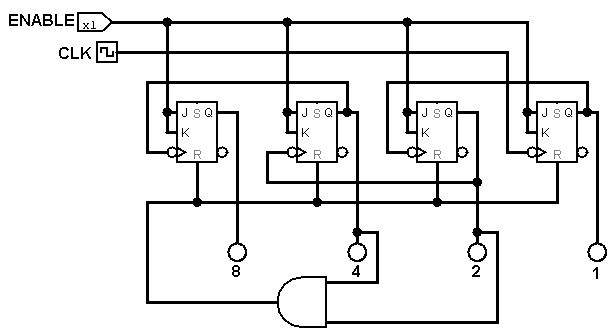
The first JK flip-flop takes a clock signal and its output oscillates at half the frequency of the clock. The second flip-flop then takes the output of the first as its own clock and its output oscillates at half the frequency. This pattern repeats for subsequent flip-flops, creating a ripple effect. The action of halving the clock frequency for each pass through a JK flip-flop imitates binary counting.

The ripple counter counts up when all flip-flops are set to activate at the falling edge of the clock, counts down when all flip-flops are set to activate at the rising edge of the clock.



*A 4-bit big-endian ripple counter*

By using a few logic gates to check when the counter reaches a certain value, we can set a maximum or minimum for the counter. The RESET input of each JK flip-flop can be used to reset the counter.



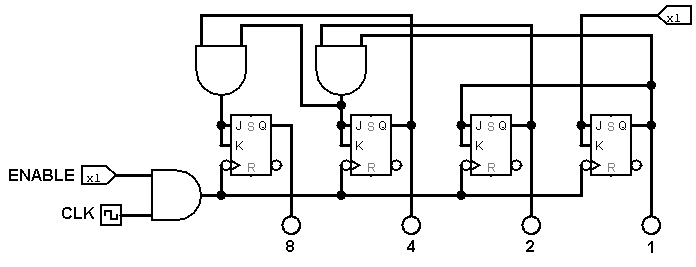
*A 4-bit big-endian ripple counter counting from 0 to 5 only*

Due to the delay between the detection of the limit and the resetting of the counter, this circuit can briefly slip into the illegal state 6. This can be disastrous when the counter is used for critical operations. In order to fix this issue, a clocked or synchronous counter is needed.

**Synchronous counter or common clock counter**

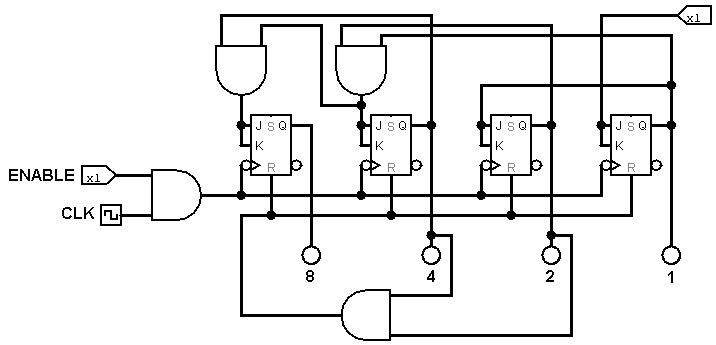
The **synchronous counter** or **common clock counter** is similar to the ripple counter in that it also uses multiple JK flip-flops in the toggle setting. However, in this circuit, the output of one JK flip-flop is not fed as the clock input but rather as the J and K inputs to the next. The flip-flops are driven by a common clock, making the circuit synchronous.

In contrast to the ripple counter, the synchronous counter will always count up irrespective of whether the flip-flops are positive edge-triggered or negative edge-triggered.



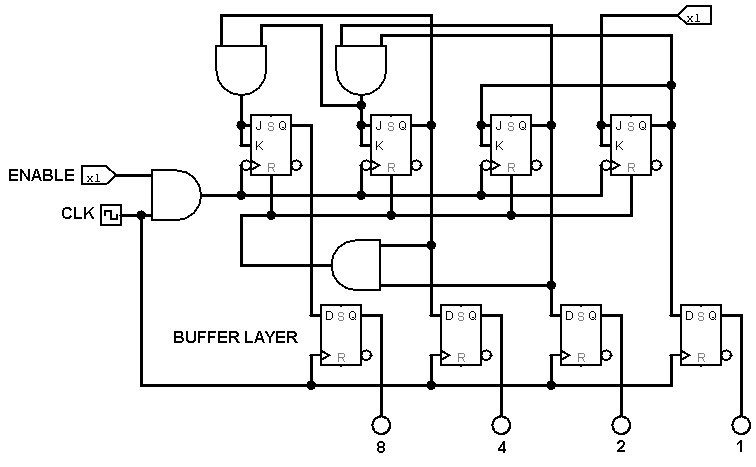
*A 4-bit big-endian synchronous counter. The AND gates ensure that a bit only flips only if all previous bits are 1. This matches the pattern in binary counting.*

Again, logic gates can be used to set a limit for the counter.



*A 4-bit big-endian synchronous counter counting from 0 to 5 only*

However, this alone does not solve the fundamental problem of the illegal state leaking out of the circuit. In order to prevent this, D flip-flops driven by the same clock as the counter must be used as a **buffer**, delaying the output by 1 clock pulse to allow the values to settle.



*A buffered 4-bit big-endian synchronous counter limited at 5*

**Week 4 - Memory, stacks and computer architectures**

**Memory**

Computer memory falls into many categories, two of which include ROM and RAM.

ROM (read-only memory) is a type of non-volatile memory (that is, memory that does not require power to retain information) which cannot be electronically modified after manufacture. ROM is used to store read-only data or data that is likely to stay unchanged throughout the lifespan of a system.

PROM (programmable read-only memory) is a type of ROM which allows data to be changed only once after manufacture. PROMs are blank when manufactured, with all bits reading 1. Bits can be changed to 0 afterwards if needed, in an irreversible process.

EPROM (erasable programmable read-only memory) is a type of reusable PROM that allows data to be erased after programming, often with the aid of UV light. EPROM programming is slow.

EEPROM (electrically erasable programmable read-only memory) can be considered an extension of EPROM which offers programming and erasing capabilities in-circuit through special programming signals. Writing in EEPROM is slower than reading from it, and EEPROM has a limit to the number of writes.

Flash memory is a type of EEPROM that offers high speed and compact data storage at the cost of large erase chunks and limited number of writes. Flash memory degrades on every write since this operation requires the injection of electrical energy which damages the memory cell. Flash memory is used in memory cards, USB flash drives among other devices.

RAM (random-access memory) is a type of computer memory that allows reading and writing data in any location. It is used to store working data and machine code due to it being volatile memory (that is, memory that loses information when the power cuts off.) There are two main types of RAM: Static RAM and Dynamic RAM.

**Memory addressing**

**Stacks**

**Computer architectures**

**Interrupt and polling**

**Week 5 - Encoders, decoders, (de)multiplexers, advanced number representation**

Two ways to represent signed integers in computer systems:

Sign magnitude: Use the most significant bit to represent the sign.

Operation: Take the most significant bit and set it to 0 if the number is positive, 1 if the number is negative.

Pros: Simple to perform.

Cons: Reduced value range, 0 ambiguity. Binary addition algorithm does not work.

2’s complement: Shift the number range so that 0 sits in the middle. New number range becomes

Operation: Represents the number in binary as normal if it is positive. If negative, flip each bit in the binary representation and add 1.

Pros: Maintain representational capacity of bits, preserves the property that the most significant bit still indicates the sign, no 0 ambiguity.

Cons: Harder to perform

Sign extension:

* Used when we want to represent a signed binary number in a larger word size.
* Repeat the sign bit all the way up to the largest bit of the desired size.

Representing real numbers in binary:

Fixed point representation: Idea is similar to

**Week 6 - Data communication**

**Week 7 - Introduction to ARM Assembly programming**

**Week 8 - Conditionals and timers in ARM Assembly**

**Limitations of MOV**

MOV combines the operation (24 bits) and the operand (8 bits) into one 32-bit word so that the instruction can be carried out in one clock cycle. Therefore, MOV only works with 8 bit values. To overcome this limitation, MOV can be combined with ORR to construct the number part by part.

**Week 9 - Functions and arrays in ARM Assembly**

**Arrays**

An array in ARM Assembly is a contiguous block of memory which represents an indexable list of values of the same type.

To work with arrays, we need to know its starting address, its constant offset between each element and its size.

Create an array in ARM Assembly with a label, followed by a data type (dw - 4 bytes - for numbers; or db - 1 byte - for characters)

To read the value at index k in the array (assuming 0-index), use the command ldr targetReg,[arrAddress,offset]

To store a value at index k in the array, use the command str.

Example:

myArray: ; label

dw 1, 3, 4, 5, 6, 7 ; data type and values

**Week 10 - Reading GPIO input and screen writing in ASM Assembly**

**Week 11 - Microprocessors and microcontrollers**