CPS311 Lecture: Sequential Circuits

Last revised May 30, 2019

Objectives:

1. To introduce asynchronous latches and synchronous flip-flops, plus asynchronous preset/clear)
2. To introduce SR, D, and JK configurations
3. To show how to implement a finite state machine using flip-flops and combinatorial networks
4. To give a first introduction to the idea of parallelism

Materials:

1. Circuit Sandbox and demonstration circuits Asynchronous SR Latch, Level-Triggered D Latch, Race, Master-Slave D Flip-Flop, Master-Slave SR Flip-Flop, Master-Slave D Flip-Flop with Preset+Clear, JK Flip-Flop, Traffic Light Controller, Traffic Light Controller with Walk Flip-Flop, Divide by Three Counter
2. Latch and Flip-flop summary handout

I. Introduction

A. Thus far, we have been discussing COMBINATORIAL logic circuits. These have the property that the output at any time is a boolean function of the inputs (with some propagation delay when inputs change).

B. Complete computer systems also require circuits that have MEMORY - that is, circuits whose output can be a function of the input AT SOME TIME in the past. Such circuits are used for:

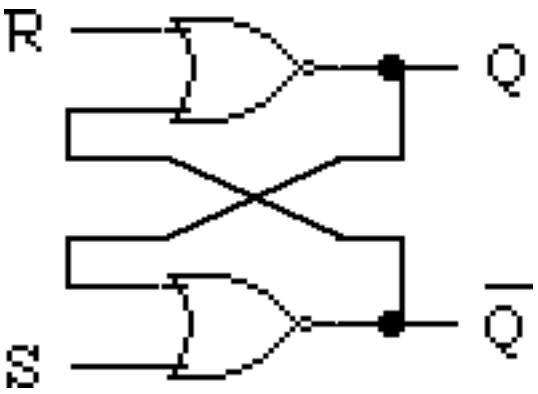
1. Internal memory: registers, main memory

2. Control circuits that cycle the system through the series of steps comprising the algorithm for a given task. (The system must keep track of its state - what step it is currently on - in order to know what to do next.

II. Latches and Flip Flops

A. It turns out to be quite easy to use our combinatorial building blocks to construct a circuit that has memory - i.e. whose output is a function of past as well as present inputs.

The following is a simple example of such a circuit:



(Note that it has two outputs, labeled Q and Q' - implying that they are intended to be opposites. Therefore, it suffices to specify the value of Q to specify the state of the device.)

1. When S is 1 (high) and R is 0 (low), what will be the state of the device?

   ASK

   Q will be 1 (high)

   (The name S stands for "set" - S = 1 sets the state to 1)

2. What about when S is 0 (low) and R is 1 (high)?

   ASK

   Q will be 0 (low)

   (The name R stands for "reset" - R = 1 resets the state to 0)

3. Now consider the case where both S and R are 1 (high). What is the state now?

   ASK

   This would result in both Q and Q' being 0 (low) - which violates our design intention that they be opposites. That is, this input pattern is not allowed.

4. Finally, what happens when both S and R are 0 (low)?

   ASK

   There are two stable states:

   a) If Q is 0 (low), then Q' is 1 (high) (since both inputs of the lower gate are 0) This state is stable, since the upper NOR gate now has one 1 (high) input, making its output 0 (low).

   b) If Q is 1 (high), then Q' is 0 (low) (since the lower gate now has one input 1 (high)). This state is also stable, since the upper NOR gate has both inputs 0 (low), making its output 1 (high).

   c) When the circuit is first turned on, the circuit will go into one of these two states non-deterministically. (Actually, slight physical differences in the transistors in the two gates will usually serve to decide the state for any particular device.)

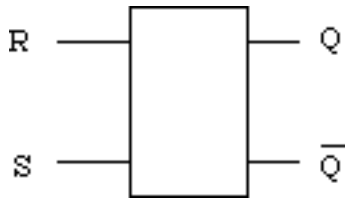
5. Effect of momentarily setting either S or R to 1 (high).

   a) If the circuit is in the stable state with Q 0 then inputting a 1 on R has no effect. However, inputting a 1 on S changes the state to the state with Q 1, where the circuit will remain even after S returns to 0.

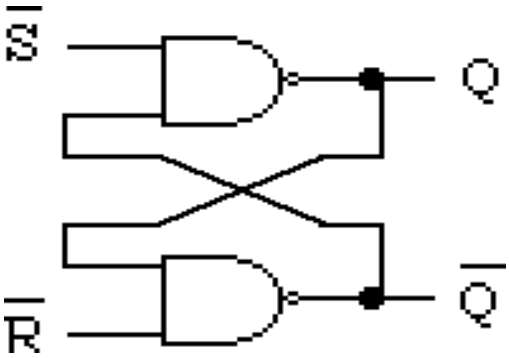
b) If the circuit is in the stable state with Q 1 then inputting a 1 on S has no effect. However, inputting a 1 on R changes the state to the state with Q 0, where the circuit will remain even after R returns to 0.

6. DEMONSTRATE (File Asynchronous SR latch)

7. This is variously called an RS latch or an SR latch, and has its own special symbol:

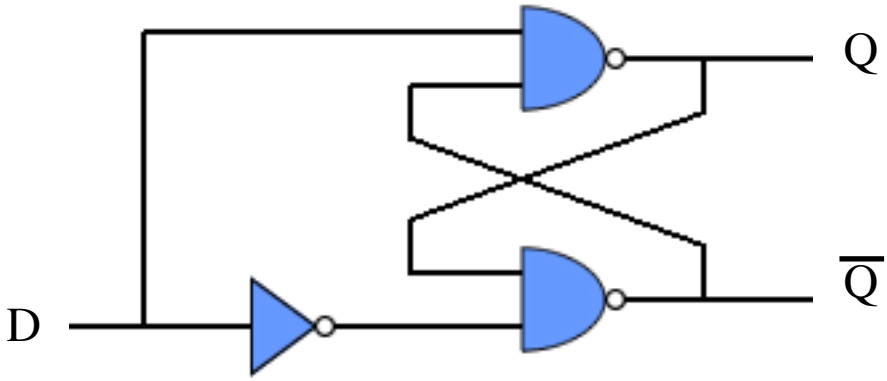


8. One can build a similar circuit from NAND gates - which is what we will actually use in our demonstrations and later in the lecture:



Notice that, in this case, the S and R inputs are ACTIVE LOW - that is a 0 on the appropriate input sets or resets the circuit, and the stable state arises when both are 1. (Both 0 produces inconsistent behavior.) This is denoted by the inversion symbol (a bar) over the two inputs.

B. The circuit we have used is called an SR or RS circuit. It has the important problem that one of the input patterns produces undefined results and thus is disallowed. We can easily convert SR circuit into one that always behaves consistently by adding a single inverter:

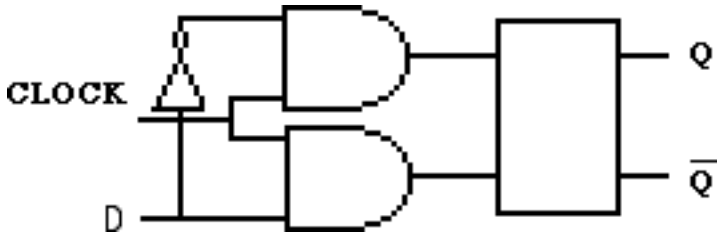


We call this circuit a D-Latch

C. Another important characteristic of the latches we have been considering is they are ASYNCHRONOUS - that is, the output changes almost instantaneously when the input changes. (There is a slight delay due to internal switching times of the gate.)

1. This can pose a problem if one builds a register, in which several latches are used to represent a multi-bit value (e.g. 32 latches could be used to represent a 32 bit number). In this case, one would like all the latches to change state at the same time - which we call SYNCHRONOUS behavior. (The state changes are synchronized)

2. One can easily build a clocked variant of our D circuit (so as to also eliminate the possibility of inconsistent input.) In this circuit, the state changes only when a special clock pulse is received. This makes it possible to synchronize state changes.



a) This circuit has two inputs, called D (data) and Clock.

b) When the clock input is low (0), the outputs of both AND gates are necessarily 0 (low) - which ensures that the circuit will remain in whatever state it is in, regardless of the D input.

c) When the clock input is 1 (high), the circuit can change state.

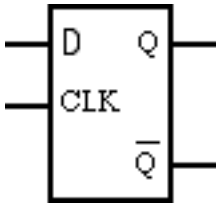
(1)If D is 0 (low), the upper AND gate receives two 1 (high) inputs, so the R input to the RS device is 1 and the device goes into the 0 state if it is not already there, or else simply remains there.

(2)If D is 1 (high), the lower AND gate receives two 1 (high) inputs, so the S input to the RS device is 1 and the device goes into the 1 state if it is not already there, or else simply remains there.

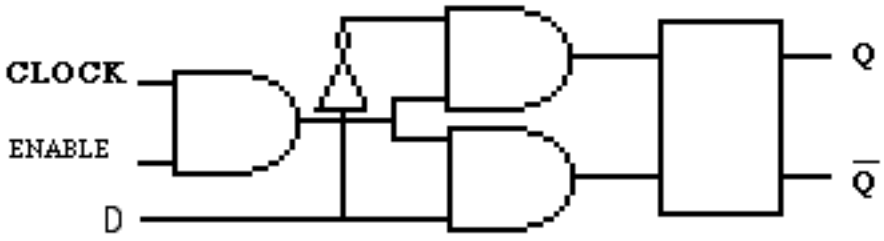
(3)That is, this device remembers what the value of D was the last time the Clock was 1 (high).

d) DEMONSTRATE (file Level-Triggered D-Latch)

e) This kind of circuit is called a level-triggered latch (it latches onto the input when the clock is high) and has the following symbol (a bit different from the one in the book, but one used more conventionally):



3. A variant of the D latch has an enable input in addition to the clock that is anded with the clock - i.e.

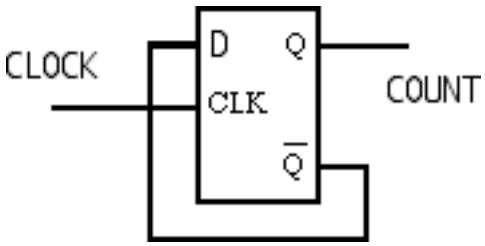


a) For a state change to occur, <u>both</u> CLOCK and ENABLE must be 1 (high).

b) This variant is often used for register sets - all registers in the set receive the same CLOCK signal, but only the specific register that is to change receives a 1 (high) on its ENABLE.

D. However, even this device is not adequate for all situations.

1. Consider what would be involved in building a device whose output feeds back to its input. Such a situation might arise, for example, if one were trying to build a COUNTER - a device that counts the number of clock pulses it receives. For example, the following is a one-bit counter - it is intended to go through the sequence of states 0, 1, 0 , 1, 0, 1 ...



This circuit will not work as intended. Why?

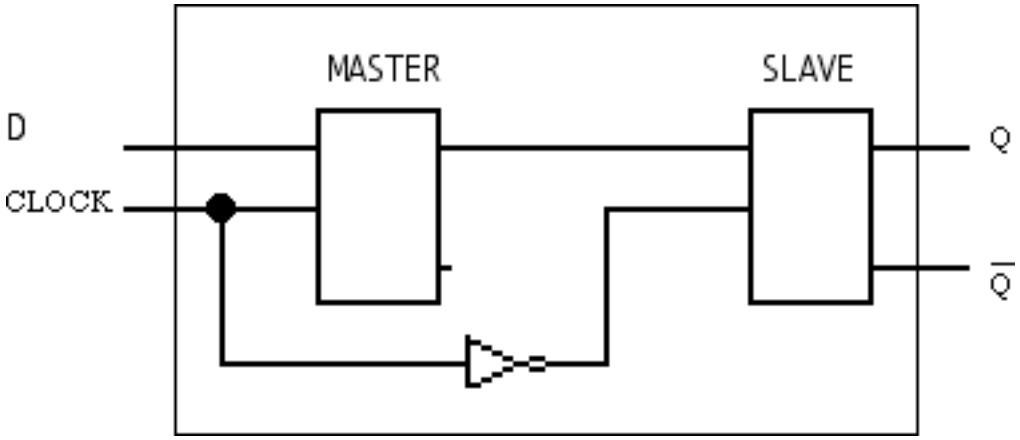
ASK

There is a race condition in the circuit - if the clock remains high after the output has changed state, the latch will change state again, and again, and again. To get the desired behavior, the clock would have to be high just long enough for one state change to occur - a critical time that actually varies slightly from gate to gate due to slight variations in manufacturing.

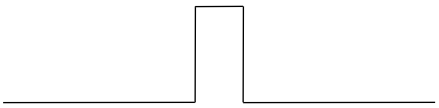
DEMONSTRATE - file Race. Set simulation speed to 1 second before running. (note: the Circuit Sandbox simulation includes a reset circuit that is needed to start the latch off in a known state. The switch is off to reset, on to allow the counter to run.)

2. To prevent race conditions such as this, it is possible to construct a master-slave flip-flop that changes state on a PULSE or an EDGE, rather than on a clock LEVEL. For example, the following flip-flop changes state just when the clock transitions from high back to low. (Note: the outlined portion is internal and is not visible to the user - it's a "black box". The user sees only the inputs D and CLOCK and the two outputs.)

   Note: The change in terminology. We call a device like this a flip-flop rather than a latch. (Sometimes the term "flip-flop" is also used for what we have called a latch, but we're being careful to follow the terminology used in the book here to avoid confusion.)



   a) Consider what happens when the clock input receives a pulse like the following:



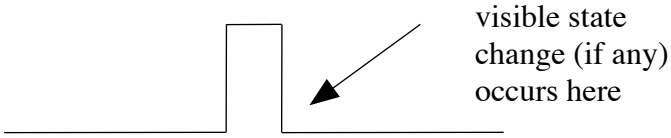
   (1)When the clock is low (0), no state changes can occur, because the master latch cannot change state.

   (2)When the clock goes to high (1), the master latch changes state in accordance with the D input. However, when the external clock is high, the clock input to the internal slave latch is low, due to the inverter, so it remains in whatever state it was in. Thus, any state changes occurring in the master are not visible externally.

(Note: we assume that the external inputs do NOT change during the brief period that the clock is high. This is a requirement that designers using this kind of flip-flop must comply with.)

(3)When the clock goes back to low (0), the clock input to the slave becomes high (1), and it simply copies the state of the master. This may cause a visible state change on the external outputs. However, at this point further changes to the state of the master are impossible, since its clock is low, so the flip flop remains in whatever state it changes to.

(4)That is, the external change of state is visible shortly after the falling edge of the clock.



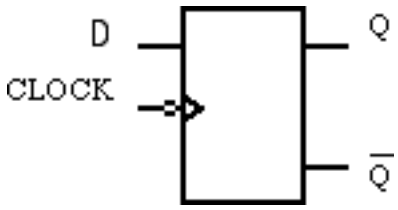
At all other times, no externally-visible change can occur, even though internal state changes might happen

(5)IMPORTANT: The Harris text develops this along slightly different lines, with the uninverted clock going to the slave and the inverted clock going to the master. This yields a device whose state transitions occur on the rising edge of the clock, rather than the falling edge.

In practice, both types of master-slave flip-flops are used, with the choice sometimes depending on which way it is easier to build the circuit. Because the flip-flops we will use in lab change external state on the falling edge of the clock, we will use this style in lecture; but realize that the Harris book - and hence the homework problems from the book - do it the other way!

b) DEMONSTRATE - File Master Slave D Flip-Flop

c) Another name for such a flip flop is a PULSE TRIGGERED FLIP-FLOP. It has the following special symbol:

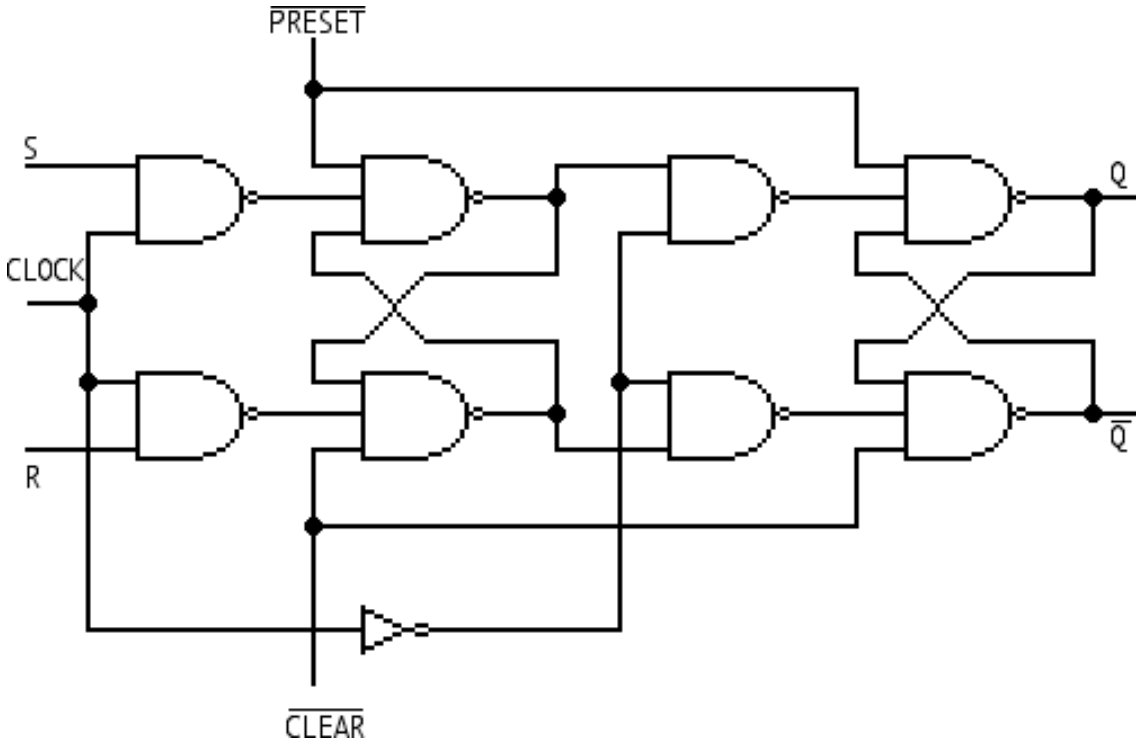


(Note special symbol at clock; triangle indicates operation is triggered by a clock edge; presence of a "bubble" indicates that the device is triggered by the falling edge of the clock) (It should also be noted that the exact circuit given here is not necessarily used in actual commercial IC's.)

d) We have developed a master-slave D flip flop - but it is also possible to build a master-slave SR flip flop.

DEMONSTRATE: File master-slave SR

E. Finally, it is possible to produce a master-slave flip flop that has both clocked and asynchronous inputs. (This time we will use SR inputs rather than a D input based on NAND gates in anticipation of where we will go next.)



1. This is basically a master-slave SR flip flop of the sort we have just been looking at, but with all the gates shown explicitly. The S and R inputs are synchronous - they only affect the state of the flip-flop on a clock.

2. What is new is the (active-low) inputs labelled Preset' and Clear'. These are asynchronous - their effect happens immediately, without regard to the clock.

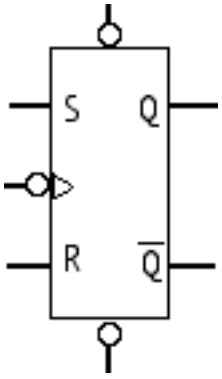
   a) A low on Preset' forces the flip-flop into the 1 state, regardless of what state it is currently in.

   b) A low on Clear' forces it into the zero state, regardless of what state it is currently in.

   c) (Simultaneous lows on both of these inputs are disallowed)

3. DEMONSTRATION - file Master Slave D Flip-Flop with Preset+Clear. (Note: In the demonstration, preset and clear are wired to buttons via an inverter, so that pushing the button results in a low on the input.)

4. This circuit has the following symbol:



   (Note the symbols for preset and clear. “Bubbles” are used on these because they are active low.)

F. Summary: we have considered four basic types of sequential circuit.

1. An ASYNCHRONOUS latch-flop changes state immediately when its input(s) change.

2. A SYNCHRONOUS latch changes state when a clock signal calls for a change. This means that multiple latches can have their state changes synchronized - hence the name. But this still allows for the possibility that multiple state changes might occur on a single clock.

   It can be built by adding two gates to an asynchronous latch - thus 4 gates total.

3. A MASTER-SLAVE FLIP-FLOP can guarantee that no more than one state change occurs on any clock. The example we have used requires 8 gates total, plus an inverter.

   a) Because a master-slave flip-flop is at least twice as complex as a latch, it is most useful in situations where it is necessary to guarantee a single state change - e.g when there exists a data path between output and input (as is often the case, it turns out.)

      Note: When the term "flip-flop" is used without further qualification, a master-slave device is often what is meant. Latches are usually explicitly called latches.

4. Finally, we considered the possibility of incorporating both synchronous and asynchronous behavior in the same device by adding clear and/or preset inputs to a synchronous device (this could be either a latch or a master-slave flip-flop, though the example we used was a master-slave.)

   It is common to find that a hardware device incorporates a "power-up reset" circuit that puts flip flops into a known initial state (often 0) when the power is first turned on. Thus, one may find a flip-flop that has just a clear input, without a preset. (It would be rare to find the reverse.)

III. **The J-K flip flop**

A. The examples we have done have used various kinds of SR and D latches and flip flops.

1. A synchronous SR flip-flop can be described by the following table, which shows the state the flip-flop will be in after a clock pulse given its state before the pulse and inputs at the time of the pulse. Such a table is called a TRANSITION TABLE.

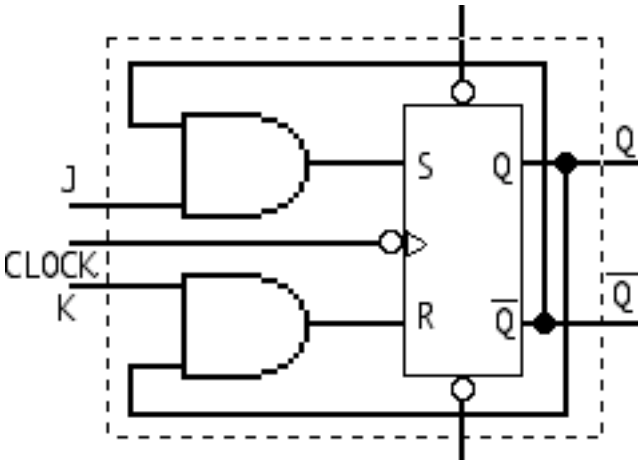
Q_{before}	S	R	Q_{after}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	disallowed
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	disallowed

Note: the table describing a flip-flop is called a transition table, whereas that describing a gate is called a truth table. The distinction - which is very important - is that the behavior of a gate depends only on its inputs, while the behavior of a flip-flop depends both on its inputs and on its current state.

2. In similar fashion, a synchronous D flip flop can be described by the following transition table - simpler than that for an SR, because it only has one synchronous input:

Q_{before}	D	Q_{after}
0	0	0
0	1	1
1	0	0
1	1	1

B. Now, suppose we build the following circuit, using a master-slave SR flip flop and two AND gates:



1. How will it behave?

ASK

a) The behavior of preset and clear is straightforward

b) To understand the behavior of the synchronous inputs, it is helpful to develop a transition table that shows internal signals as well as the external ones.

ASK CLASS TO FILL IN S, R, Q_{after}

Qbefore	J	K	S	R	Qafter
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	1	0	1
1	0	0	0	0	1
1	0	1	0	1	0
1	1	0	0	0	1
1	1	1	0	1	0

c) Of course, in practice we just include externally visible signals in the transition table, leading to the following;

Qbefore	J	K	Qafter
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

There is an interesting pattern to this table. Do you see it?

ASK

(1)When J and K are both 0, the state of the flip-flop does not change - it it was 0, it remains 0; it is was 1, it remains 1

(2)When J is 1 and K is 0, the flip flop sets (goes to the 1 state), regardless of its current state.

(3)When J is 0 and K is 1, the flip flop resets (goes to the 0 state), regardless of its current state.

(4)When J and K are both 1, the flip flop toggles (changes from 0 to 1 or 1 to 0 as the case may be)

d) DEMONSTRATION - File JK flip flop

2. The transition table is used when we want to do <u>analysis</u> - i.e. given a circuit, how does it behave. It is also possible to do <u>synthesis</u> - i.e. given a desired behavior, design a circuit that exhibits it. For this, we use a different tool called an EXCITATION TABLE, which answers the question "if I'm in __ state, what J/K values do I need to get to ___ state?

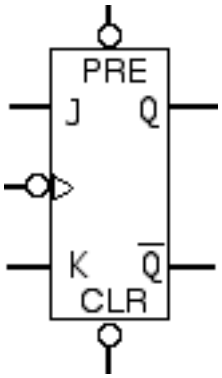
This table can be constructed from the transition table. For each row, one of the two values J and K will be specified, and the other will be a don't care.

ASK class to provide J and K values

Qcurrent	Qdesired	J	K
0	0	0	-
0	1	1	-
1	0	-	1
1	1	-	0

That is, if the flip-flop is currently in the 0 state, only the "J" value affects the next state (since K is anded with a 0); if it is in the 1 state, only the "K" value matters

3. The JK flip flops we will use in lab have the following symbol



a) The triangle on the clock input denotes that state changes are triggered by the edge of a clock.

b) The bubble on the clock input denotes that state changes are triggered by the <u>falling</u> edge of the clock. (Recall: the Harris book uses flip flops triggered on the <u>rising</u> edge - hence no bubble)

c) The bubbles on the preset and clear inputs denote that they are active low.

4. A few observations

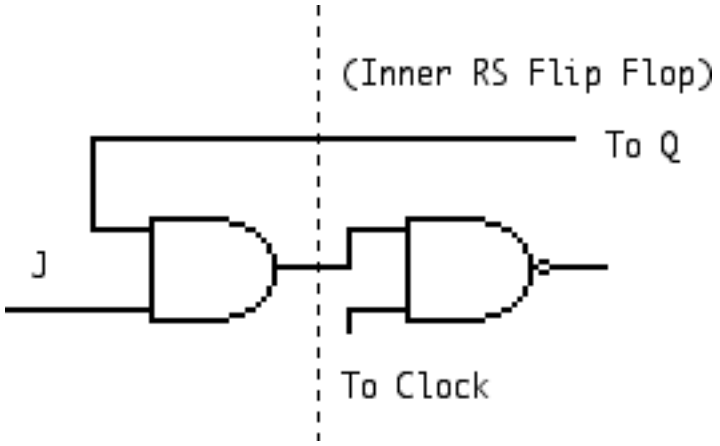
a) The JK flip-flop is the most general type of flip-flop. While D's have the advantage of one fewer input, and thus are often used in situations where data is to be stored (as in a register), JK's are widely used in control circuits to implement state machines, as we shall see.

b) JK flip-flops are only useful as master-slave flip-flops; there is no such thing as a JK Latch. Why?

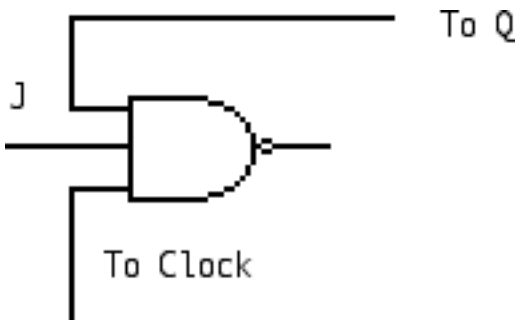
ASK

The circuit itself contains feedback from the outputs to the input.

c) In practice, JK flip flops are implemented by folding the and gates at the input in with the initial master latch - e.g. (if using SRs implemented with NAND gates) instead of



we have



IV. **Uses for Flip-Flops**

A. Flip flops are used in computer systems in two general ways:

1. They are used for implementing registers.

   a) For example, in lab you worked with a Z80 microprocessor that has 18 programmer-visible 8-bit registers (A, B, C, D, E, F, H, L, A', B', C', D', E', F', H', L', I and R) plus 4 16-bit registers (PC, SP, IX, IY). On the CPU chip, each 8-bit register is implemented by 8 flip-flops, and each 16 bit register by 16 flip-flops. While all registers receive the same clock, only the one that is to receive the result of an operation has its enable 1.

   b) Most computer systems use special registers to hold the information being transferred to/from external devices via the bus(ses) - e.g. if a computer system uses a 64 bit bus, then the CPU probably has a 64 bit register that is used to hold information being transmitted via the bus (or maybe two registers - one for each direction)

   c) etc. - we'll deal with this topic much more extensively later in the course.

2. They are used in building state machines. Any finite state machine (as discussed in CPS320) can be implemented by using one or more flip flops plus combinatorial networks, following an approach we will discuss now, with a clock used for synchronization.

B. Sequential circuits are most easily designed or analyzed by working with a STATE DIAGRAM. Each state represents one possible value of the different flip-flops - i.e. a sequential circuit with n flip flops can have up to 2^n states. Edges between states, labelled with possible inputs, show the various transitions between states.

C. As an example of designing a sequential control circuit this way, consider a controller for a traffic light to be positioned at an intersection between a north-south and an east-west street.

1. Specifications.

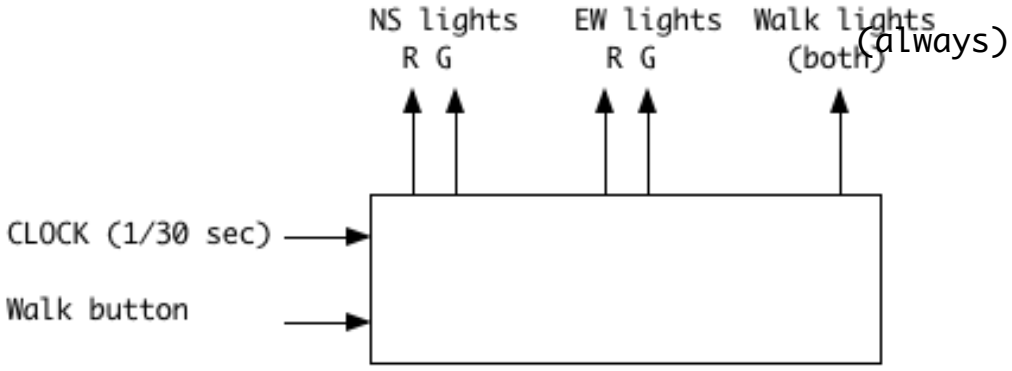
a) For simplicity, we assume only red and green lights plus a walk light, so that the following patterns may be displayed:

N-S street	E-W street
R	G
G	R
Walk	Walk

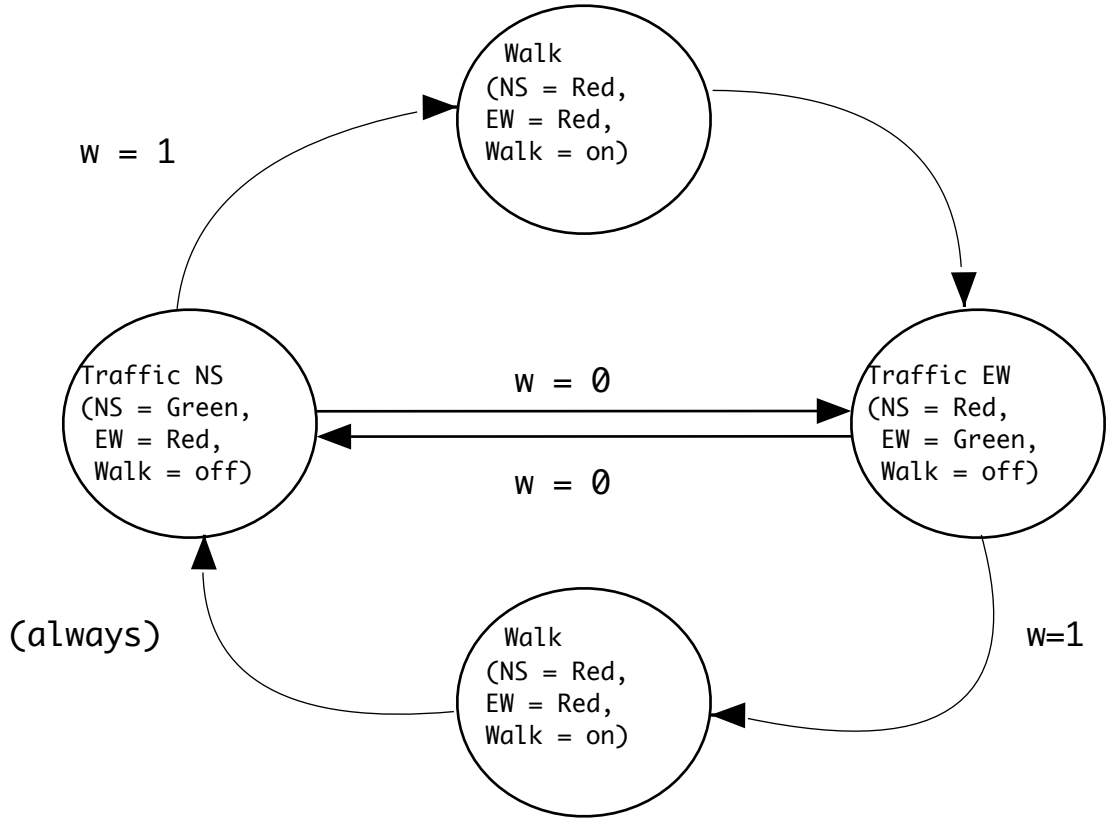
b) Ordinarily, the light changes from one Green-Red state to the other every 30 seconds.

c) However, if a pedestrian is pushing a walk button at the time of a change, then he gets a 30 second walk cycle. (This is unrealistic; normally the system would remember any push of the button within the cycle. We could add this simply later.

d) We can view the controller as a black box:



e) Its behavior can be described by the following STATE DIAGRAM (where w stands for the walk button input - w = 0 means it is not pressed, and w = 1 means it is pressed):



(1)Note that, when in a walk state, we always go to a traffic movement state, regardless of the state of the button. (Pedestrians can't get walk twice in a row)

(2)Note that there are two "walk" states. Why do we need two?

ASK

Because in a walk state, the system must remember which direction traffic was moving in before so that it can allow traffic movement in the correct direction after. (If walk always went back to, say, NS movement, drivers traveling east-west might get very annoyed!)

2. To build this device, we would need a minimum of two flip flops, since 2^2 gives 4 states. We will refer to the value of the two flip-flops as Q_1 and Q_0.

a) One immediate question is how to associate flip flop values with states. We will (arbitrarily) choose the following pattern. (In practice, one may consider several different assignments and choose the one giving the simplest overall circuit.)

State	Flip flop values (Q_1Q_0)
NS traffic	00
Walk after NS	01
EW traffic	10
Walk after EW	11

b) We can now convert our state diagram into a STATE TABLE. In this table, the "present" and "next" Q values come directly from the state diagram, the outputs come from the above discussion, and he J and K values come from the excitation table for a JK flip flop with $Q_{present}$ as the current state and Q_{next} as the desired state.

(Put up, with class filling in J's and K's.)

Present State		Input	Next State		FF Inputs			
Q_1	Q_0	w	Q_1	Q_0	J_1	K_1	J_0	K_0
0	0	0	1	0	1	-	0	-
0	0	1	0	1	0	-	1	-
0	1	0	1	0	1	-	-	1
0	1	1	1	0	1	-	-	1
1	0	0	0	0	-	1	0	-
1	0	1	1	1	-	0	1	-
1	1	0	0	0	-	1	-	1
1	1	1	0	0	-	1	-	1

3. We can now derive equations for the outputs and J and K values as a function of the present state (Q_1 Q_0) and input (w).

a) As noted earlier, in this machine, the outputs are a function only of the current state. In particular:

NS Green = $Q_1' \bullet Q_0'$

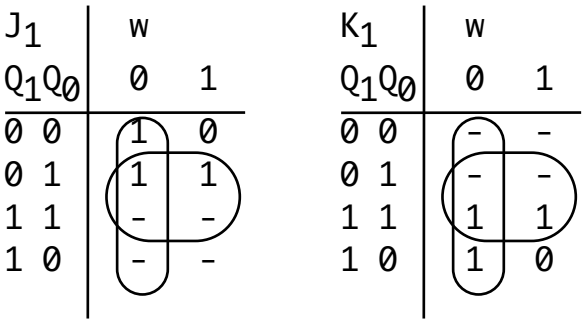
NS Red = NS Green ' = $Q_1 + Q_0$

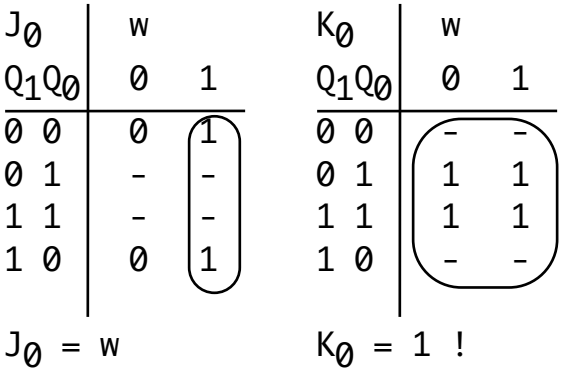
EW Green = $Q_1 \bullet Q_0'$

EW Red = EW Green ' = $Q_1' + Q_0$

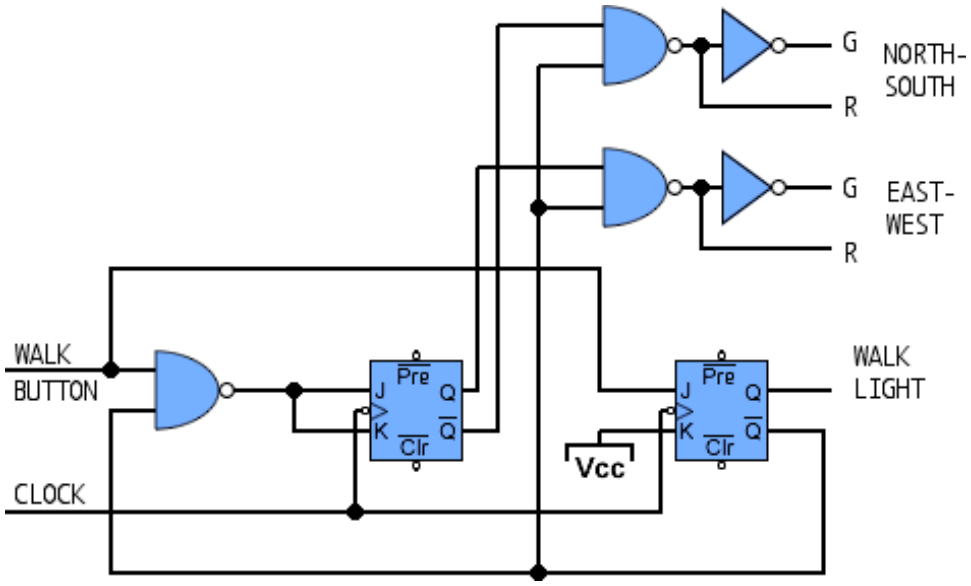
Walk = $Q_1' \bullet Q_0 + Q_1 \bullet Q_0 = Q_0$ (we could use a Karnaugh map if necessary to do the simplification - in this case, the simplification is fairly obvious)

b) We develop the following Karnaugh maps for the J's and K's:





c) Finally, we can create a circuit for our traffic light controller:



d) Note that only NAND gates (including inverters, which are one-input NAND gates) are used. As a result:

(1)Q_0 + w' is implemented as (Q_0 • w)'

(2)The NAND gates directly realize the Red light outputs, which are inverted to give the corresponding Green light outputs.

e) DEMONSTRATE: File Traffic Light Controller

Note: the Circuit Sandbox simulation has a reset line needed to put the circuit in a consistent initial state.

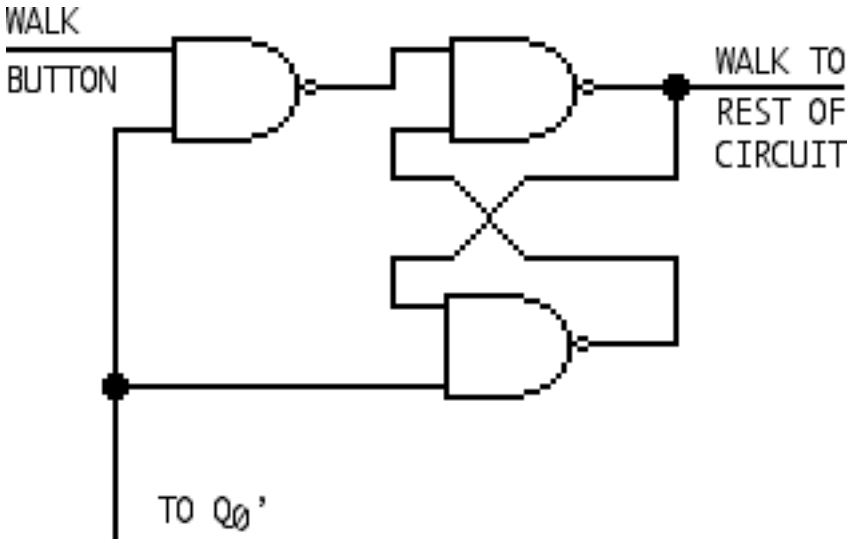
4. Adding memory for walk button

a) It would be nice if the circuit would "remember" when the walk button is pushed, so that a pedestrian would not have to continually lean on the button until the light changes. Our system so far is synchronous, changing states only on clock pulses. In this case, though, we need an asynchronous circuit that can change state whenever the button is pushed. A single asynchronous RS flip flop (with active low inputs) will suffice.

(1)We will ultimately want to connect the walk button to the set input.

(2)We observe that $Q_0 = 1$ implies that a walk request has been granted, so we connect Q_0' to the reset input (recall it is active low).

(3)To prevent an inconsistent input from arising, we allow the reset input to INHIBIT the set input. This leads to the following circuit (explicitly showing the asynchronous flip-flop as two NAND gates)



DEMONSTRATE: File Traffic Light Controller with walk FF

(Note need to first put controller in a consistent state, then press walk once to get walk FF in a consistent state)

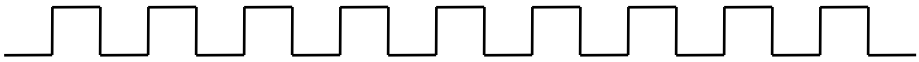
D. In the above example, the outputs were a function only of the state, not of the input. (The input served only to control state transitions.) Thus, the outputs could be recorded in the state nodes of the diagram.

1. Such a system is called a MOORE CIRCUIT.

2. It is also possible to have a sequential circuit where the output is a function BOTH of the state and of the input. In this case, we record the output on the edges of the state diagram, separated from the inputs by a slash. This is called a MEALY CIRCUIT.

3. Example: A divide-by-three counter which outputs one 1 for every 3 1's seen as input (not necessarily in succession.) After outputting a 1, it starts counting all over again.

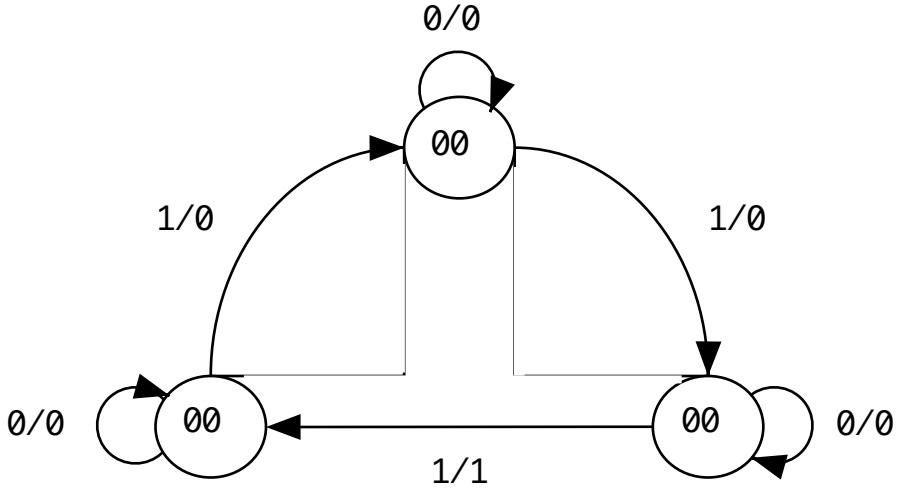
   Given the following input



   Such a device would produce the following output

4. To build this, will need three states, corresponding to 0, 1, or 2 1's seen so far.

5. We have the following state diagram (here we number the states to match the count of 1's seen)



6. We get the following state table. Note that, this time, we need to include the outputs in the state table, since they are a function of both current state and input. (We will call the input X and the output C for "carry").

   Because the next step will require the J and K values, we have included columns for them in the table, to be filled in from the excitation table for the JK flip-flop.

   (Have class fill in Next State and Output from the State Diagram, and J's and K's from the JK flip flop Excitation Table).

Current State Q_1 Q_0	Input X	Next State Q_1 Q_0	Output C	Excitation J_1	K_1	J_0	K_0
0 0	0	0 0	0	0	-	0	-
0 0	1	0 1	0	0	-	1	-
0 1	0	0 1	0	0	-	-	0
0 1	1	1 0	0	1	-	-	1
1 0	0	1 0	0	-	0	0	-
1 0	1	0 0	1	-	1	0	-
1 1	0	- -	-	-	-	-	-
1 1	1	- -	-	-	-	-	-

   Note that the 11 state is not used by the machine. Since we will use two flip flops, there is such a state, but the machine should never be in it. We will simplify our circuit if we make the next state and output for this state don't cares.

7. Derivation of circuit, using Karnaugh maps

C	X	
Q_1Q_0	0	1
0 0	0	0
0 1	0	0
1 1	-	-
1 0	0	1

$C = Q_1 \bullet X$

J_1	X	
Q_1Q_0	0	1
0 0	0	0
0 1	0	1
1 1	-	-
1 0	-	-

$J_1 = Q_0 \bullet X$

K_1	X	
Q_1Q_0	0	1
0 0	-	-
0 1	-	-
1 1	-	-
1 0	0	1

$K_1 = X$

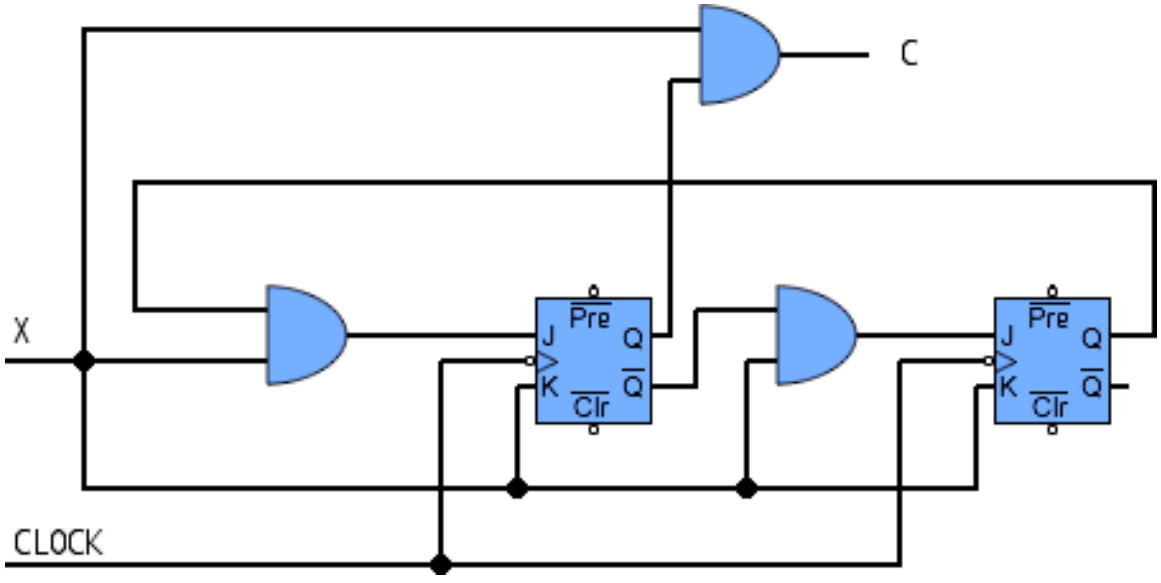
J_0	X	
Q_1Q_0	0	1
0 0	0	1
0 1	-	-
1 1	-	-
1 0	0	0

$J_0 = Q_1' \bullet X$

K_0	X	
Q_1Q_0	0	1
0 0	-	-
0 1	0	1
1 1	-	-
1 0	-	-

$K_0 = X$

8. Circuit: (Have class draw)



9. DEMONSTRATION - File Divide by Three (note reset added to Simulation)

V. A First Look at Parallelism

A. The book ends its discussion of sequential circuits by introducing the topic of parallelism. This is a topic that will come up again and again in our study of hardware systems.

B. The need for developing parallel systems arises because sequential systems of the sort we have been looking at are speed-limited by two factors.

1. Propagation delay through gate networks

2. Time required for a sequential device such as a flip-flop to change state. (Some of this is discussed in §3.5, which was not assigned).

3. Both of these are ultimately consequences of the physics of the physical devices being used.

C. As the book points out, greater speeds can be achieved by using either of two kinds of parallelism:

1. Spatial parallelism - replicating functional units so that multiple copies of a task can be performed at the same time.
2. Temporal parallelism - also called pipelining.
3. We will see examples of both of these later in the course.

D. One other topic this section discussed is two important measures of the time to perform a task: latency and throughput.

1. In brief, latency is the time it takes from the time a task is first initiated until its results are available.
2. Throughput is the rate at which tasks can be completed.

   Relate to "cookie" example in text.
3. Neither type of parallelism can reduce latency, but either can improve throughput.