# Counters

- A counter is a circuit whose outputs go through a prescribed sequence of values and repeats over and over. The outputs of the counter are held at the outputs of flip flops.
- The change from one value to the next value happens on the arrival of the active edge of some signal; e.g., a clock signal.
- Counters come in two varieties: *asynchronous* and *synchronous* counters.
- We will consider mainly synchronous counters.

# Asynchronous counters

- Asynchronous counters are characterized by the observation that different signals drive different clock inputs on different flip flops. This means that the outputs of each flip flop *do not change at the same time are not synchronized with some master clock*.

# Asynchronous binary ripple counter

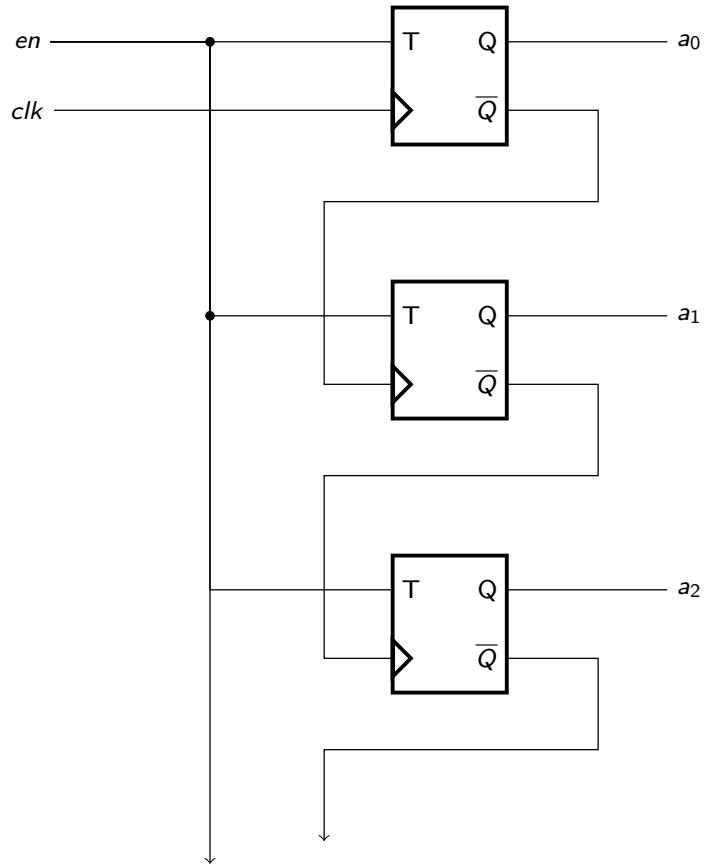
- Design a 3-bit counter that counts $0, 1, \cdots 7$ and repeats. The circuit should have a signal *en* such that $en = 1$ the circuit should count upon the arrival of the active clock edge. When $en = 0$, the circuit should not count (hold its current value).
- Can figure this out by looking at the sequence of numbers we want to produce (ignore the *en* input for the moment and assume the counter is always counting...)

| current count | | | next count | | | |
|---|---|---|---|---|---|---|
| $a_2$ | $a_1$ | $a_0$ | $a_2$ | $a_1$ | $a_0$ | |
| 0 | 0 | 0 | 0 | 0 | 1 | |
| 0 | 0 | 1 | 0 | 1 | 0 | |
| 0 | 1 | 0 | 0 | 1 | 1 | |
| 0 | 1 | 1 | 1 | 0 | 0 | |
| 1 | 0 | 0 | 1 | 0 | 1 | |
| 1 | 0 | 1 | 1 | 1 | 0 | |
| 1 | 1 | 0 | 1 | 1 | 1 | |
| 1 | 1 | 1 | 0 | 0 | 0 | $\leftarrow$ *repeat* |

- Observations: i) $a_0$ always toggles; ii) $a_1$ toggles when $a_0 = 1 \rightarrow 0$; iii) $a_2$ toggles when $a_1 = 1 \rightarrow 0$.
- In other words, the $i$-th bit toggles when the $(i-1)$-th bit makes a transition from $1 \rightarrow 0$.

# Asynchronous binary ripple counter

- The observation that outputs need to toggle motivates the use of *TFF*s. The observation that things toggle when a signal changes $1 \rightarrow 0$ motivates using these signals as inputs to flip flop clock inputs.
- The circuit...




- Note the creative use of the enable signal... If $en = 0$ all inputs to the toggle flip flops is 0 and the flip flops will not toggle.

# Asynchronous binary ripple counter

- If we don't care about how fast the circuit works, then there is really no problem with this sort of counter.
- However, it can be *slow* if the number of bits is very large. Here's why...
  - Consider the number of bits $n$ to be very large (e.g., 128 or something like that).
  - Further, consider all of the flip flops to be 1; the next count value is 0 and *all flip flop outputs need to toggle*.
  - See the problem?
  - Output $a_0$ must toggle (and it takes $T_{co}$ for $a_0$ to change...), then $a_1$ will toggle (and it takes another $T_{co}$ for $a_1$ to change), then $a_2$ will toggle (and it takes another $T_{co}$ for $a_2$ to change), and so on...
- Because every flip flop is clocked by a different clock and the clocks form a *chain*, this counter can be quite slow.
- Better to create a circuit in which all flip flop outputs change at the same time.

# Synchronous counters

- Synchronous counters are characterized by the observation that a single clock signal is used to clock every flip flop. This means that all the flip flop outputs *change simultaneously and are synchronized with the clock*.

# Synchronous binary up counter

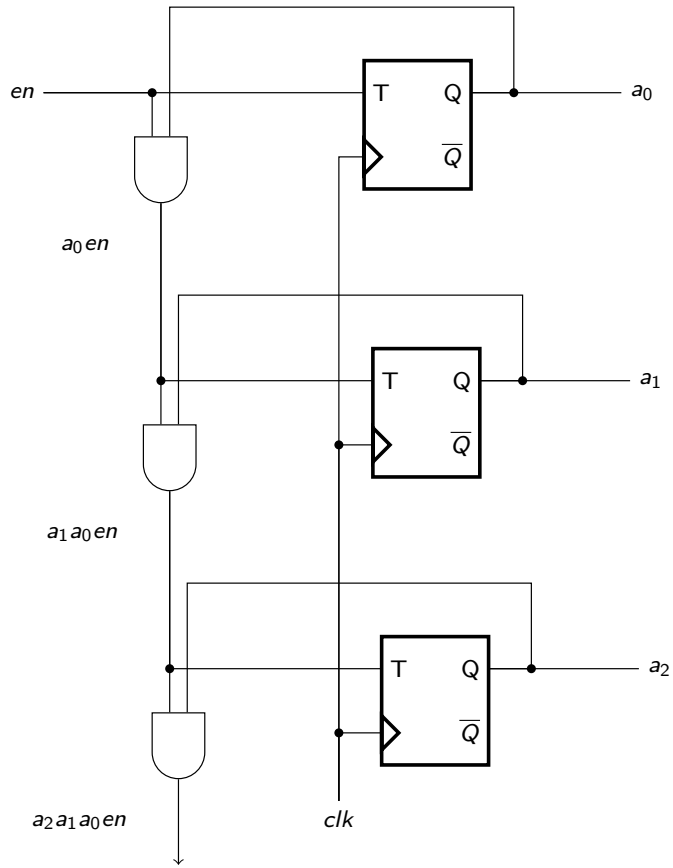
- Design a 3-bit synchronous binary up counter with an enable input *en*. When $en = 0$, the counter should not count. When $en = 1$, the circuit should count upon the arrival of the active clock edge. The output sequence always increases by 1 and repeats over and over...
- Design using intuition... Consider the count sequence for a binary up counter and assume that $en = 1$...

| current count | | | next count | | | |
|---|---|---|---|---|---|---|
| $a_2$ | $a_1$ | $a_0$ | $a_2$ | $a_1$ | $a_0$ | |
| 0 | 0 | 0 | 0 | 0 | 1 | |
| 0 | 0 | 1 | 0 | 1 | 0 | |
| 0 | 1 | 0 | 0 | 1 | 1 | |
| 0 | 1 | 1 | 1 | 0 | 0 | |
| 1 | 0 | 0 | 1 | 0 | 1 | |
| 1 | 0 | 1 | 1 | 1 | 0 | |
| 1 | 1 | 0 | 1 | 1 | 1 | |
| 1 | 1 | 1 | 0 | 0 | 0 | $\leftarrow$ *repeat* |

- Observations: i) $a_0$ always toggles; ii) $a_1$ toggles when the current value of $a_0 = 1$; iii) $a_2$ toggles when the current value of $a_1 a_0 = 11$.
- In other words, $a_i$ should toggle when bits $a_{i-1}$ down to $a_0$ are all 1. This would apply for a binary up counter with any number of bits.

# Synchronous binary up counter

- The observation that bits need to toggle motivates the use of *TFF*.
- Our circuit...




- Note the creative use of the enable signal... If $en = 0$ all inputs to the *TFF* will be forced to 0 and they will not toggle. Otherwise, the inputs to the *TFF* are the conditions required for the flip flops to toggle appropriately.
- We could use set/reset control signals to initialize the counter to any *initial* state.

# Synchronous binary down counter

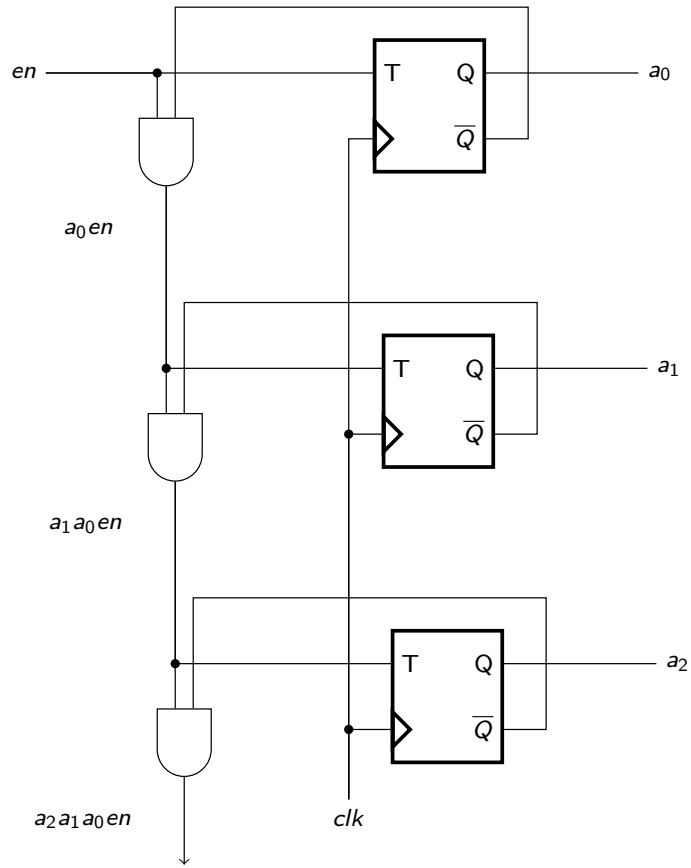
- Design a 3-bit synchronous binary down counter with an enable input *en*. When $en = 0$, the counter should not count. When $en = 1$, the circuit should count upon the arrival of the active clock edge. The count should always decrease by 1 and repeats over and over...
- Design using intuition... Consider the count sequence for a binary down counter and assume that $en = 1$...

| current count | | | next count | | | |
|---|---|---|---|---|---|---|
| $a_2$ | $a_1$ | $a_0$ | $a_2$ | $a_1$ | $a_0$ | |
| 1 | 1 | 1 | 1 | 1 | 0 | |
| 1 | 1 | 0 | 1 | 0 | 1 | |
| 1 | 0 | 1 | 1 | 0 | 0 | |
| 1 | 0 | 0 | 0 | 1 | 1 | |
| 0 | 1 | 1 | 0 | 1 | 0 | |
| 0 | 1 | 0 | 0 | 0 | 1 | |
| 0 | 0 | 1 | 0 | 0 | 0 | |
| 0 | 0 | 0 | 1 | 1 | 1 | $\leftarrow$ *repeat* |

- Observations: i) $a_0$ always toggles; ii) $a_1$ toggles when the current value of $a_0 = 0$; iii) $a_2$ toggles when the current value of $a_1 a_0 = 00$.
- In other words, $a_i$ should toggle when bits $a_{i-1}$ down to $a_0$ are all 0. This would apply for a binary down counter with any number of bits.

# Synchronous binary down counter

- The observation that bits need to toggle motivates the use of *TFF*.
- Our circuit...




- Note the creative use of the enable signal... If $en = 0$ all inputs to the *TFF* will be forced to 0 and they will not toggle. Otherwise, the inputs to the *TFF* are the conditions required for the flip flops to toggle appropriately.
- We could use set/reset control signals to initialize the counter to any *initial* state.

# Synchronous binary up/down counter

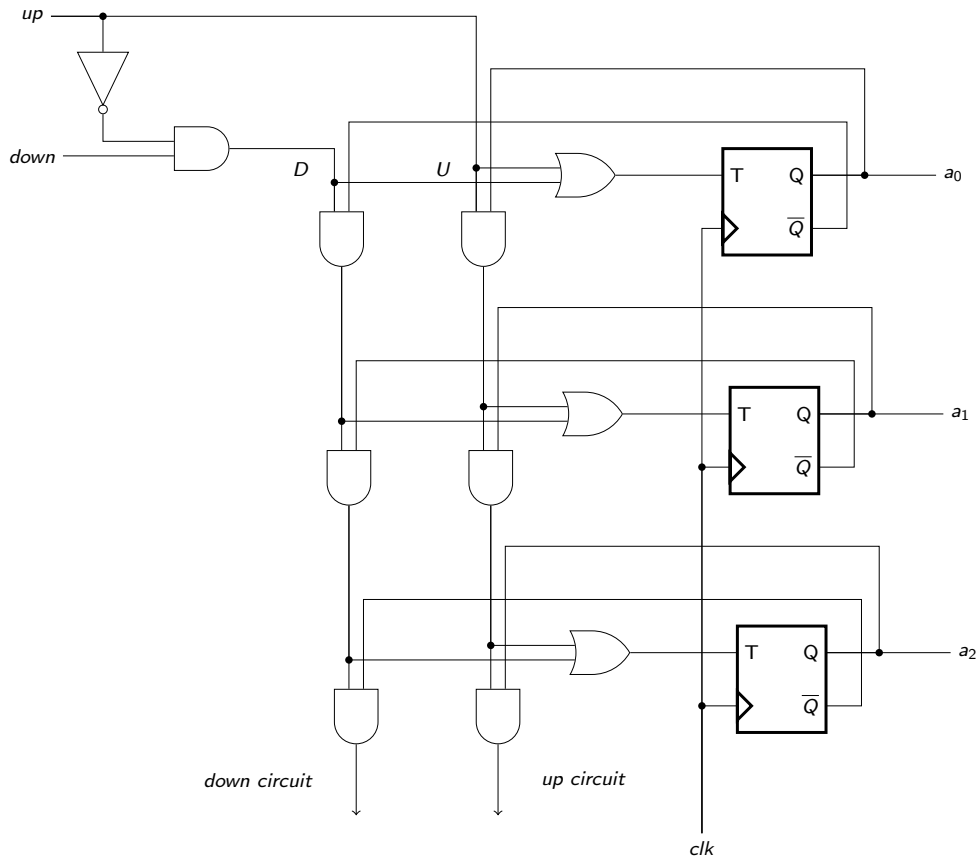
- Can combine different operations together... Design a single circuit capable of counting either up or down according to the following:

| **Inputs** | | **Action** |
|---|---|---|
| *up* | *down* | |
| 0 | 0 | *don't count* |
| 0 | 1 | *count down* |
| 1 | X | *count up* |

- The approach we can take... Use circuits we already know and then add extra circuitry to control what signal gets to the inputs of the flip flops *depending on the operation we want to perform*.

# Synchronous binary up/down counter

- Our circuit... includes both the binary up counter circuitry, the binary down counter circuitry and extra logic to control the operation...




- The extra gates and control logic are effectively performing a multiplexer operation to get the correct signal to the input of the flip flop.
- Consider the signals $D$ and $U$ in the circuit — these are the enable signals for the down counter and up counter, respectively... Think about it...

# Synchronous binary up/down counter with parallel load

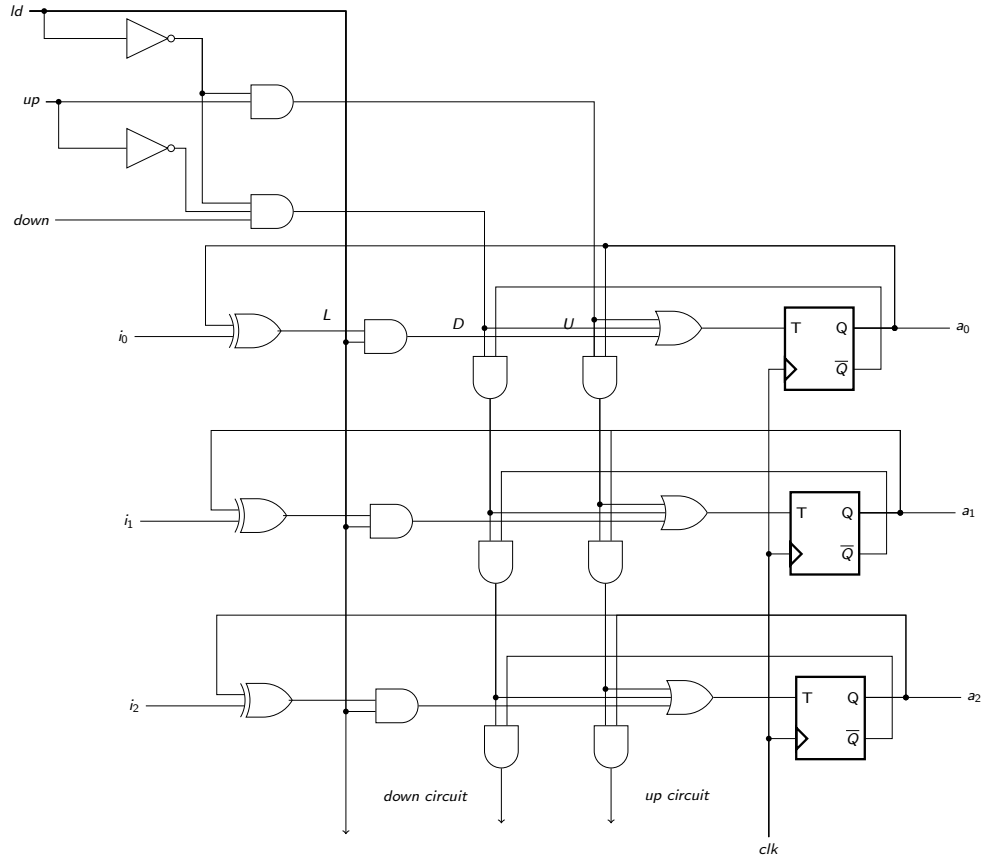
- Design a binary up/down counter that also has the ability to load an arbitrary value.
- This requires another control line and results in a new table of operation:

| **Inputs** | | | **Action** |
|---|---|---|---|
| *load* | *up* | *down* | |
| 0 | 0 | 0 | *hold* |
| 0 | 0 | 1 | *count down* |
| 0 | 1 | X | *count up* |
| 1 | X | X | *load* |

- The approach we can take... Use circuits we already know and then add extra circuitry to control what signal gets to the inputs of the flip flops *depending on the operation we want to perform*.

# Synchronous binary up/down counter with parallel load

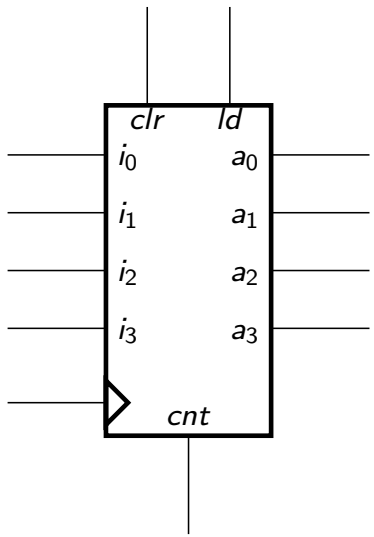
- The circuit...




- The extra gates and control logic are effectively performing a multiplexer operation to get the correct signal to the input of the flip flop.
- **Q:** What is the purpose of the **XOR** gates? **A:** Since we are using *TFF*, we need to *compare* $i_j$ and $a_j$ to decide how to load (do we need to invert $a_j$ or not?).

# Counter symbols

- We migh have a symbol for a counter... Likely accompanied by a table explaining its operation.



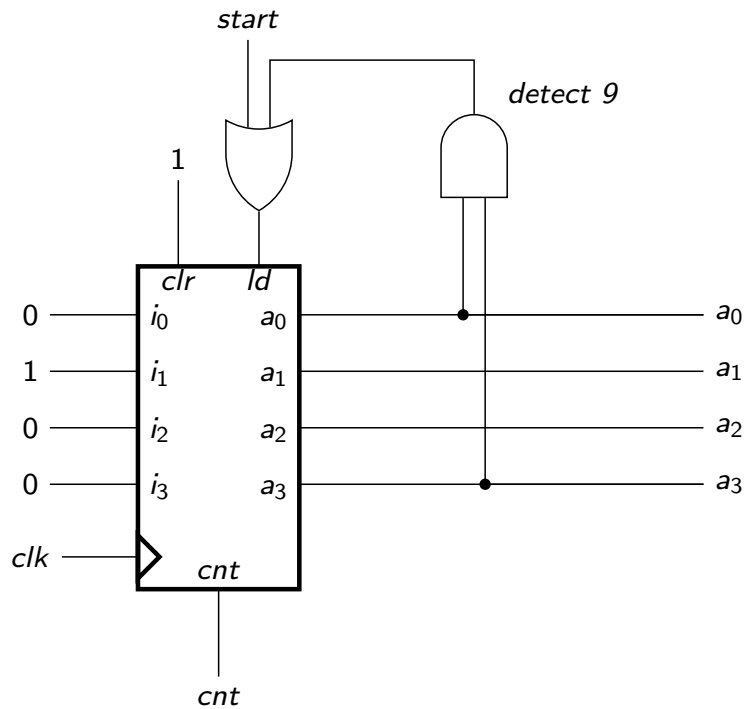

| *clr* | *ld* | *cnt* | *clk* | Function |
|---|---|---|---|---|
| 0 | X | X | X | Output to zero |
| 1 | 1 | X | ↑ | Parallel load |
| 1 | 0 | 1 | ↑ | Count up |
| 1 | 0 | 0 | ↑ | Hold |

# Modulo counter

- If we don't want to count through the entire binary sequence, then we might want to build a *modulo counter*. A modulo-$n$ counter would count $0, 1, \cdots n-1$ and repeat.
- We could also decide to start the count sequence at some value other than 0.
- Could design such a counter from scratch or use a "off-the-shelf" binary counter and make modifications.
  - Take a binary counter and add additional logic to detect when we have reached our maximum count.
  - Then, upon the next clock cycle, do a parallel load to restart the count sequence.

# Modulo counter

- Example... design a counter that counts $2, 3, \cdots, 9$ and then repeats.
- Solution...



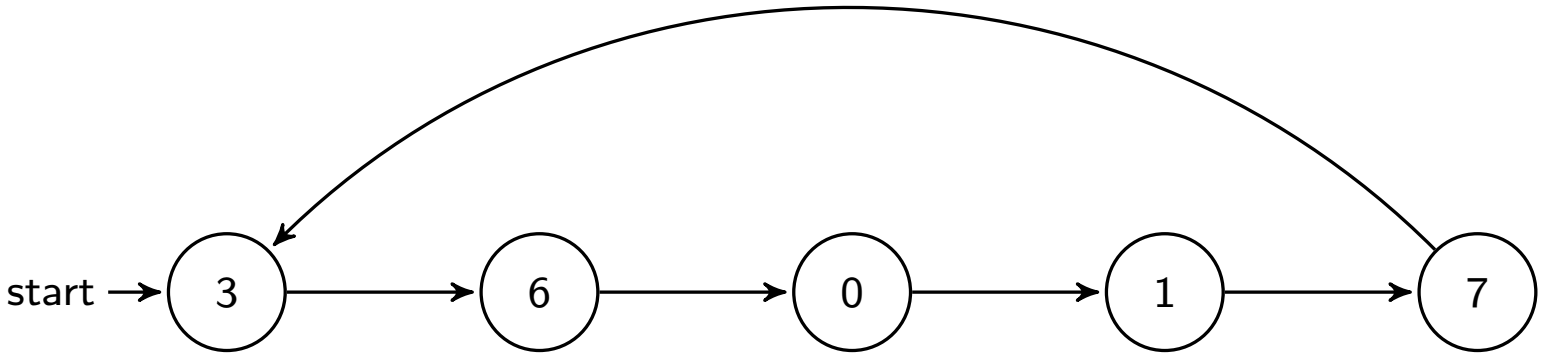

- When this circuit hits 9, the output of the **AND** gate will be 1 and the *ld* signal will be active. At the next active clock edge, the circuit will load the binary value of 2 and start counting correctly.
- Also added a *start* signal to be able to reset the counter and start counting from 2.

# Generic counter design

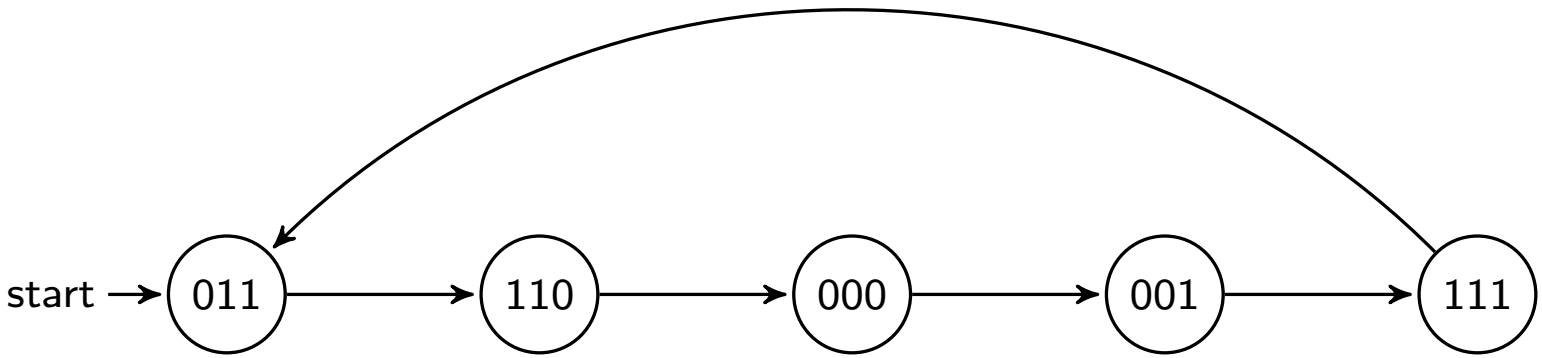
- Counter design can also be generic — we can design a counter which counts in any sequence we want. Further, we can use any type of flip flop.
- To do this, we can start by drawing a version of a *state diagram* to illustrate the count sequence.
- Example... design a counter which counts in the sequence $3 \rightarrow 6 \rightarrow 0 \rightarrow 1 \rightarrow 7$ and repeats.

# Generic counter design

- The state diagram...




- The state diagram in terms of binary values...




- Each bubble represents a state. Each edge tells us how we transition from state to state when the active clock edge arrives.
- The values (at least in the second state diagram) is the required circuit output — it tells us that we need 3 flip flops for this counter (one flip flop for each bit).

# Generic counter design

- We can convert this diagram to a table which is a version of a *state table*. The state table shows the *current state* and the *next state*. More or less, the *current state* represents the current output of the flip flops (so the current count value) and the *next state* is what the flip flop outputs need to change to (so the next count value).
- The *state table*...

| Current State | | | Next State | | |
|---|---|---|---|---|---|
| $a_2$ | $a_1$ | $a_0$ | $a_2$ | $a_1$ | $a_0$ |
| 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 1 |

- Again, this table tells us how the flip flop outputs need to change when the active clock edge arrives.

# Generic counter design

- Assume we want to use *DFF*... Recall how they work:

| $D$ | $Q(t+1)$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

or

| $D$ | $Q(t) \rightarrow Q(t+1)$ |
|---|---|
| 0 | $0 \rightarrow 0$ |
| 0 | $1 \rightarrow 0$ |
| 1 | $0 \rightarrow 1$ |
| 1 | $1 \rightarrow 1$ |

- The table assuming we want to use *DFF*...

| Current State | | | Next State | | | Inputs for *DFF* | | |
|---|---|---|---|---|---|---|---|---|
| $a_2$ | $a_1$ | $a_0$ | $a_2$ | $a_1$ | $a_0$ | $d_2$ | $d_1$ | $d_0$ |
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |

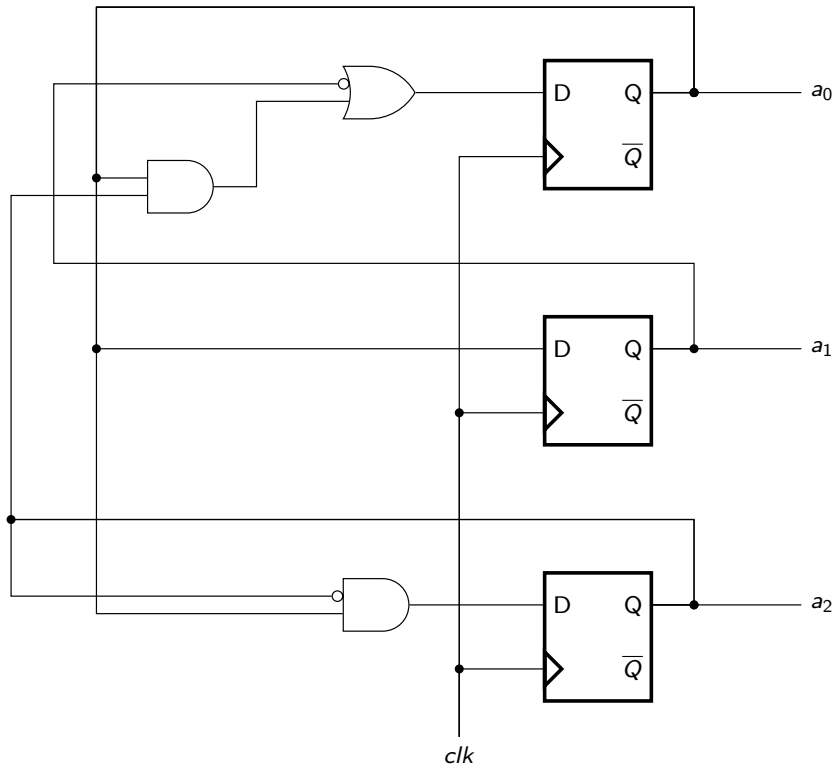
# Generic counter design

- We can use the *DFF* inputs to write down equations for each of the *DFF* inputs... We could use Karnaugh maps to get these equations...

$$\begin{aligned} d_2 &= a_2' a_0 \\ d_1 &= a_0 \\ d_0 &= a_1' + a_2 a_0 \end{aligned}$$

# Generic counter design

- The counter using *DFF*...




- What is not shown is that we should use the set and reset inputs on the *DFF*s in order the ensure that we can start the count sequence at 3.

# Generic counter design

- Assume we want to use *TFF*... Recall how they work:

| $T$ | $Q(t+1)$ |
|---|---|
| 0 | $Q(t)$ |
| 1 | $\overline{Q(t)}$ |

or

| $T$ | $Q(t) \rightarrow Q(t+1)$ |
|---|---|
| 0 | $0 \rightarrow 0$ |
| 0 | $1 \rightarrow 1$ |
| 1 | $0 \rightarrow 1$ |
| 1 | $1 \rightarrow 0$ |

- The table assuming we want to use TFF...

| Current State | | | Next State | | | Inputs for *TFF* | | |
|---|---|---|---|---|---|---|---|---|
| $a_2$ | $a_1$ | $a_0$ | $a_2$ | $a_1$ | $a_0$ | $t_2$ | $t_1$ | $t_0$ |
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |

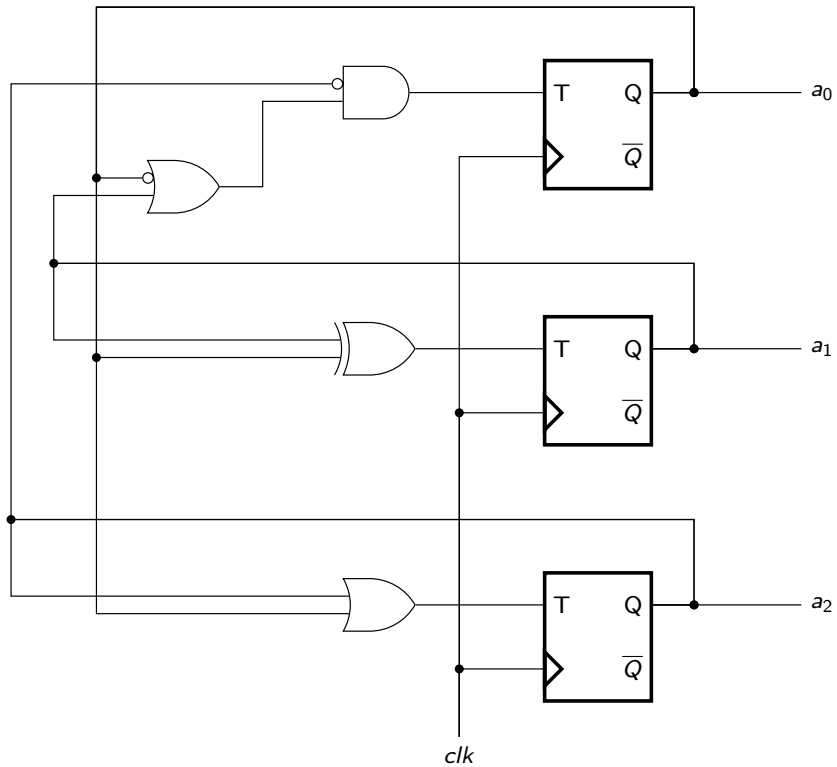
# Generic counter design

- We can use the *TFF* inputs to write down equations for each of the *TFF* inputs... We could use Karnaugh maps to get these equations...

$$\begin{aligned} t_2 &= a_2 + a_0 \\ t_1 &= a_1 \oplus a_0 \\ t_0 &= a_2' a_0' + a_2' a_1 = a_2'(a_0' + a_1) \end{aligned}$$

# Generic counter design

- The counter using *TFF*...




- What is not shown is that we should use the set and reset inputs on the *TFF*s in order the ensure that we can start the count sequence at 3.

# Generic counter design

- Assume we want to use *JKFF*... Recall how they work: Recall how a JK flip flop works.

| $J$ | $K$ | $Q(t+1)$ |
|---|---|---|
| 0 | 0 | $Q(t)$ |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | $\overline{Q(t)}$ |

or

| $J$ | $K$ | $Q(t) \rightarrow Q(t+1)$ |
|---|---|---|
| 0 | X | $0 \rightarrow 0$ |
| X | 0 | $1 \rightarrow 1$ |
| 1 | X | $0 \rightarrow 1$ |
| X | 1 | $1 \rightarrow 0$ |

- The table assuming we want to use *JKFF*...

| Current State | | | Next State | | | Inputs for *JKFF* | | |
|---|---|---|---|---|---|---|---|---|
| $a_2$ | $a_1$ | $a_0$ | $a_2$ | $a_1$ | $a_0$ | $j_2\,k_2$ | $j_1\,k_1$ | $j_0\,k_0$ |
| 0 | 1 | 1 | 1 | 1 | 0 | 1X | X0 | X1 |
| 1 | 1 | 0 | 0 | 0 | 0 | X1 | X1 | 0X |
| 0 | 0 | 0 | 0 | 0 | 1 | 0X | 0X | 1X |
| 0 | 0 | 1 | 1 | 1 | 1 | 1X | 1X | X0 |
| 1 | 1 | 1 | 0 | 1 | 1 | X1 | X0 | X0 |

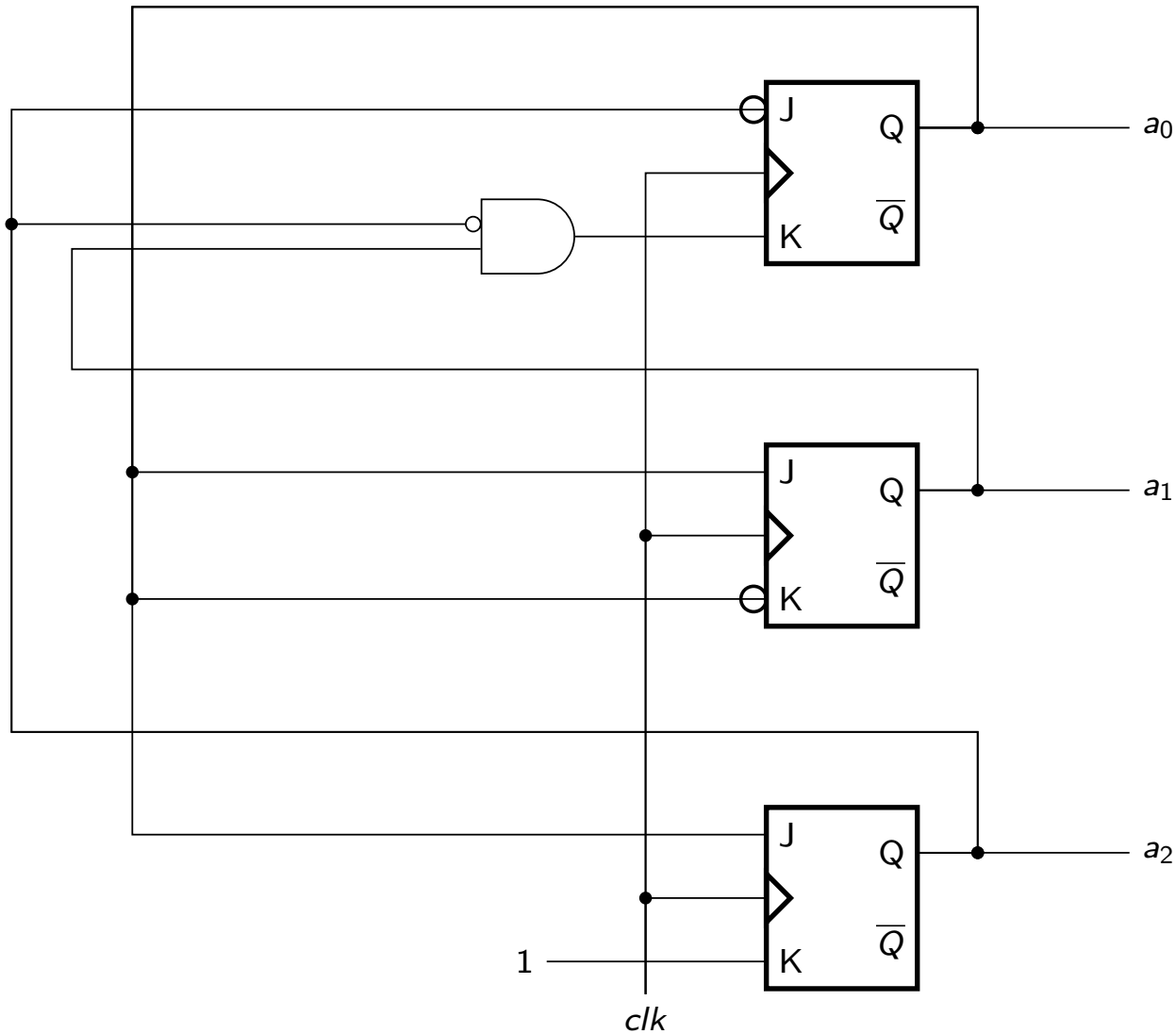
# Generic counter design

- We can use the *JKFF* inputs to write down equations for each of the *JKFF* inputs... We could use Karnaugh maps to get these equations...

$$
\begin{aligned}
j_2 &= a_0 \\
k_2 &= 1 \\
j_1 &= a_0 \\
k_1 &= a_0' \\
j_0 &= a_2' \\
k_0 &= a_2' a_1
\end{aligned}
$$

# Generic counter design

- The counter using *JKFF*...




- What is not shown is that we should use the set and reset inputs on the *JKFF*s in order the ensure that we can start the count sequence at 3.

# Generic counter design

- The type of flip flop we select has an impact on the *complexity* of the circuit we end up with (that is, how much logic gates we need to produce the inputs to the flip flops).
- In this example, using *DFF* wasn't any more or less complex than using *TFF*. This might not always be the case!
- However, using *JKFF* resulted in almost no extra logic gates. *JKFF* are a bit more difficult to work with, but often result in less combinational logic — it's because *JKFF* has a lot of don't care situations when considering the required inputs to cause the outputs to change as required.