- Agenda: introduce the topic [4, Part 1] of

finite automata ≡ Finite State Machines (FSMs)

via

1. an "in theory", i.e., concept-oriented perspective, and
2. an "in practice", i.e., perspective, spanning
   - 2.1 general application, and
   - 2.2 specific implementation in sequential logic design.

Definition

An **alphabet** is a non-empty set of **symbols**.

Definition

A **string** X with respect to some alphabet Σ is a sequence, of finite length, whose elements are members of Σ, i.e.,

$$X = \langle X_0, X_1, \ldots, X_{n-1} \rangle$$

for some n such that $X_i \in \Sigma$ for $0 \leq i < n$; if n is zero, we term X the **empty string** and denote it ϵ. It can be useful, and is common to write elements in in human-readable form termed a **string literal**: this basically means writing them from right-to-left without any associated notation (e.g., brackets or commas).

Definition

A **language** Λ is a set of strings.

Part 1: in theory (3)

- Concept: **Finite State Machines (FSMs)** are a model of **computation**.

- Concept: **Finite State Machines (FSMs)** are a model of **computation**.
  - An FSM is an (idealised) computer C, which, at a given point in time, is in one of some finite set of states.

Part 1: in theory (3)

- Concept: **Finite State Machines (FSMs)** are a model of **computation**.
  - An FSM is an (idealised) computer C, which, at a given point in time, is in one of some finite set of states.
  - C accepts an input string with respect to some alphabet Σ, one symbol at a time; each symbol induces a change in state.

- Concept: **Finite State Machines (FSMs)** are a model of **computation**.
  - An FSM is an (idealised) computer C, which, at a given point in time, is in one of some finite set of states.
  - C accepts an input string with respect to some alphabet Σ, one symbol at a time; each symbol induces a change in state.
  - Once the input is exhausted, C halts: depending on the state it halts in, we say either
    1. C accepts (or recognises) the input string
    2. C rejects the input string

- Concept: **Finite State Machines (FSMs)** are a model of **computation**.
  - An FSM is an (idealised) computer C, which, at a given point in time, is in one of some finite set of states.
  - C accepts an input string with respect to some alphabet Σ, one symbol at a time; each symbol induces a change in state.
  - Once the input is exhausted, C halts: depending on the state it halts in, we say either
    1. C accepts (or recognises) the input string
    2. C rejects the input string
  - For a language Λ of all possible input strings C could accept, we say

    $$C \text{ accepts (or recognises) } \Lambda \quad \equiv \quad \Lambda \text{ is the language of } C$$

    and use Λ to classify C ...

Definition

less powerful ⟷ more powerful

Machine	Combinatorial logic	Finite automaton	Push-down automaton	Linear-bounded automaton	Turing machine
Memory		0 stacks	1 stacks	2 stacks	2 stacks
Language		regular	context free	context sensitive	recursively enumerable
Grammar		regular ($X \to x$ or $X \to xY$)	context free ($X \to \gamma$)	context sensitive ($\alpha X \beta \to \alpha \gamma \beta$)	unrestricted ($\alpha \to \beta$)
Chomsky-Schützenberger hierarchy		type-3	type-2	type-1	type-0

Definition

A (deterministic) **Finite State Machine (FSM)** is a tuple

$$C = (S, s, A, \Sigma, \Gamma, \delta, \omega)$$

including

1. S, a finite set of **states** that includes a **start state** $s \in S$,
2. $A \subseteq S$, a finite set of **accepting states**,
3. an **input alphabet** Σ and an **output alphabet** Γ,
4. a **transition function**

   $$\delta : S \times \Sigma \to S$$

   and
5. an **output function**

   $$\omega : S \to \Gamma$$

   in the case of a **Moore FSM**, or

   $$\omega : S \times \Sigma \to \Gamma$$

   in the case of a **Mealy FSM**,

noting an **empty** input denoted ϵ allows a transition that can *always* occur.

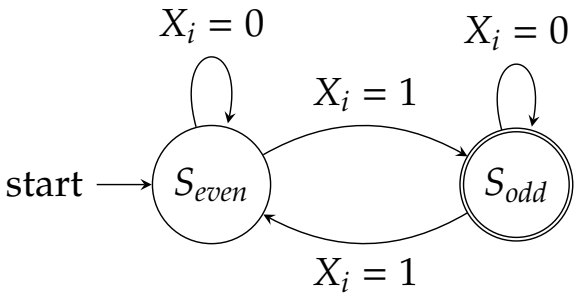
- Problem: design an FSM that decides whether a binary sequence X has an odd number of 1 elements in it.

- Solution:

Algorithm (tabular)

Q	δ	
	Q'	
	$X_i = 0$	$X_i = 1$
S_{even}	S_{even}	S_{odd}
S_{odd}	S_{odd}	S_{even}

Algorithm (diagram)



where, e.g.,

1. for the input string $X = \langle 1, 0, 1, 1 \rangle$ the transitions are

$$\rightsquigarrow S_{even} \overset{X_0=1}{\rightsquigarrow} S_{odd} \overset{X_1=0}{\rightsquigarrow} S_{odd} \overset{X_2=1}{\rightsquigarrow} S_{even} \overset{X_3=1}{\rightsquigarrow} S_{odd}$$

   so the input is accepted (i.e., has an odd number of 1 elements).
2. for the input string $X = \langle 0, 1, 1, 0 \rangle$ the transitions are

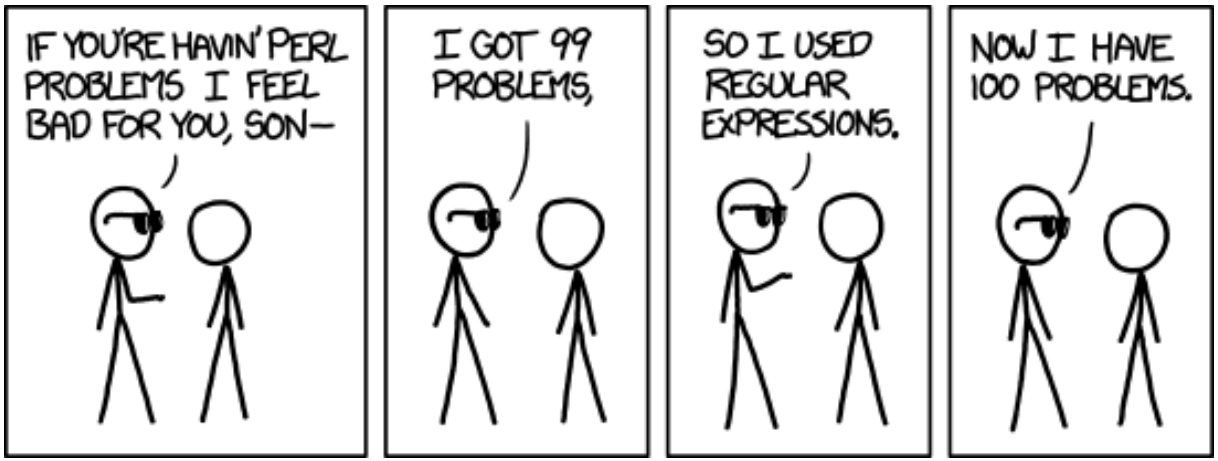
$$\rightsquigarrow S_{even} \overset{X_0=0}{\rightsquigarrow} S_{even} \overset{X_1=1}{\rightsquigarrow} S_{odd} \overset{X_2=1}{\rightsquigarrow} S_{even} \overset{X_3=0}{\rightsquigarrow} S_{even}$$

   so the input is rejected (i.e., has an even number of 1 elements).

Part 2.1: in practice, application (1)

Example #1: regular expressions + `grep` $\rightsquigarrow$ FSMs as recognisers

- Context:
  - −ve perspective:



  - +ve perspective: we could say that

$$\textbf{arithmetic expression} \overset{\text{evaluate}}{\rightsquigarrow} \text{number}$$

$$\textbf{regular expression} \overset{\text{evaluate}}{\rightsquigarrow} \text{language}$$

so a regular expression (or regex) can be used as

1. a pattern used to describe or generate a language, *or*
2. a pattern used to identify (i.e., match) members of a language.

---

https://xkcd.com/1171

Definition

We say X is a **regular expression** if it is

1. a symbol in the alphabet, i.e., $\{x\}$ for $x \in \Sigma$,
2. the union of regular expressions X and Y such that

$$X \cup Y = \{x \mid x \in X \vee x \in Y\},$$

3. the concatination of regular expressions X and Y such that

$$X \parallel Y = \{\langle x, y \rangle \mid x \in X \wedge y \in Y\},$$

or

4. the **Kleene star** of regular expression X such that

$$X^* = \{\langle x_0, x_1, \ldots, x_{n-1} \rangle \mid n \geq 0, x_i \in X\}.$$

allowing for various short-hands, e.g.,

$$\begin{array}{lcl} x & \equiv & \{x\} \\ xy & \equiv & \{x\} \parallel \{y\} \\ X^+ & \equiv & X \parallel X^* \end{array}$$

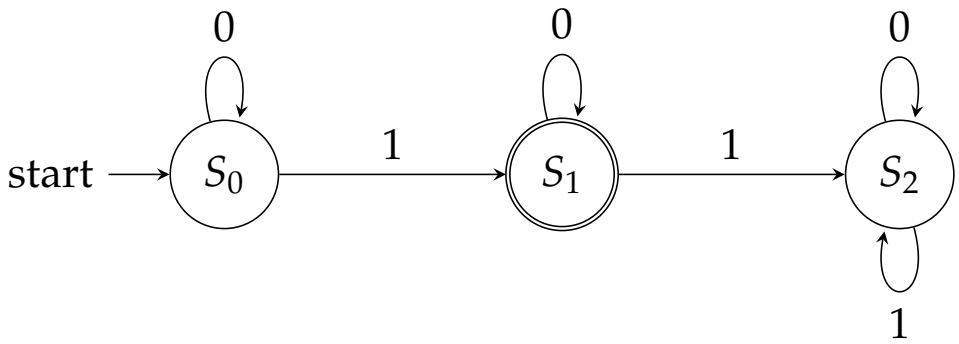
- Example [4, Example 1.53]: if $\Sigma = \{0, 1\}$, then

$$0^*10^* \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a string containing} \\ \text{a single 1} \end{array} \right\}$$

- Example [4, Example 1.53]: if $\Sigma = \{0, 1\}$, then

$$0^*10^* \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a string containing} \\ \text{a single 1} \end{array} \right\}$$

which can be realised using



- Example [4, Example 1.53]: if $\Sigma = \{0, 1\}$, then

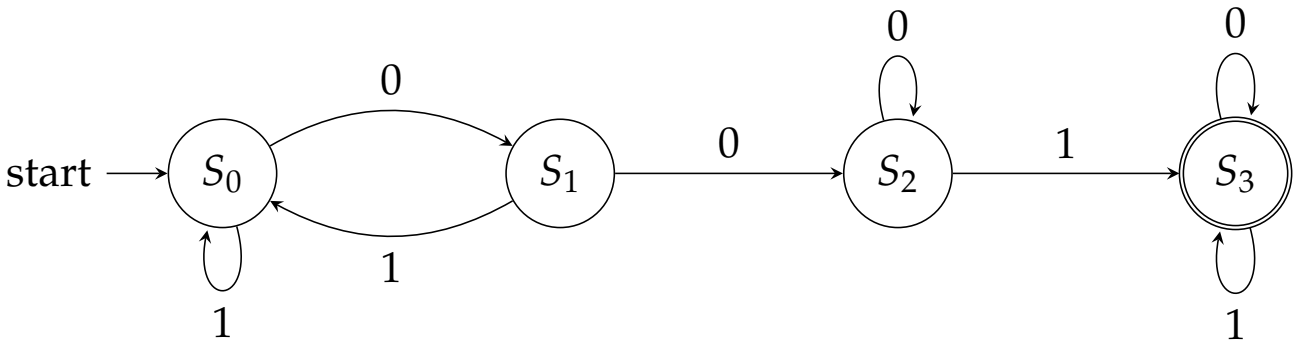
$$\Sigma^* 001 \Sigma^* \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a string containing} \\ 001 \text{ as a sub-string} \end{array} \right\}$$

which can be realised using

- Example [4, Example 1.53]: if $\Sigma = \{0, 1\}$, then

$$\Sigma^* 001 \Sigma^* \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a string containing} \\ 001 \text{ as a sub-string} \end{array} \right\}$$

which can be realised using



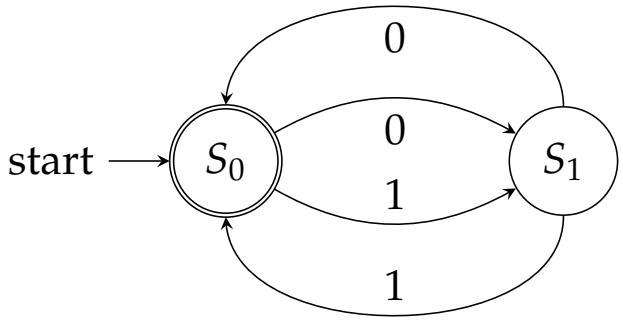
- Example [4, Example 1.53]: if $\Sigma = \{0, 1\}$, then

$$(\Sigma\Sigma)^* \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a string} \\ \text{of even length} \end{array} \right\}$$

- Example [4, Example 1.53]: if $\Sigma = \{0, 1\}$, then

$$(\Sigma\Sigma)^* \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a string} \\ \text{of even length} \end{array} \right\}$$

which can be realised using



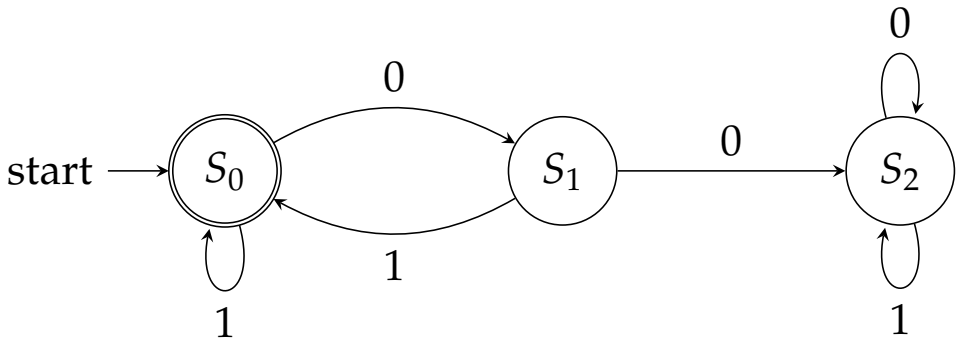
- Example [4, Example 1.53]: if $\Sigma = \{0, 1\}$, then

$$1^*(01^+)^* \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a string in which} \\ \text{every 0 is followed by at least one 1} \end{array} \right\}$$

- Example [4, Example 1.53]: if $\Sigma = \{0, 1\}$, then

$$1^*(01^+)^* \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a string in which} \\ \text{every 0 is followed by at least one 1} \end{array} \right\}$$

which can be realised using



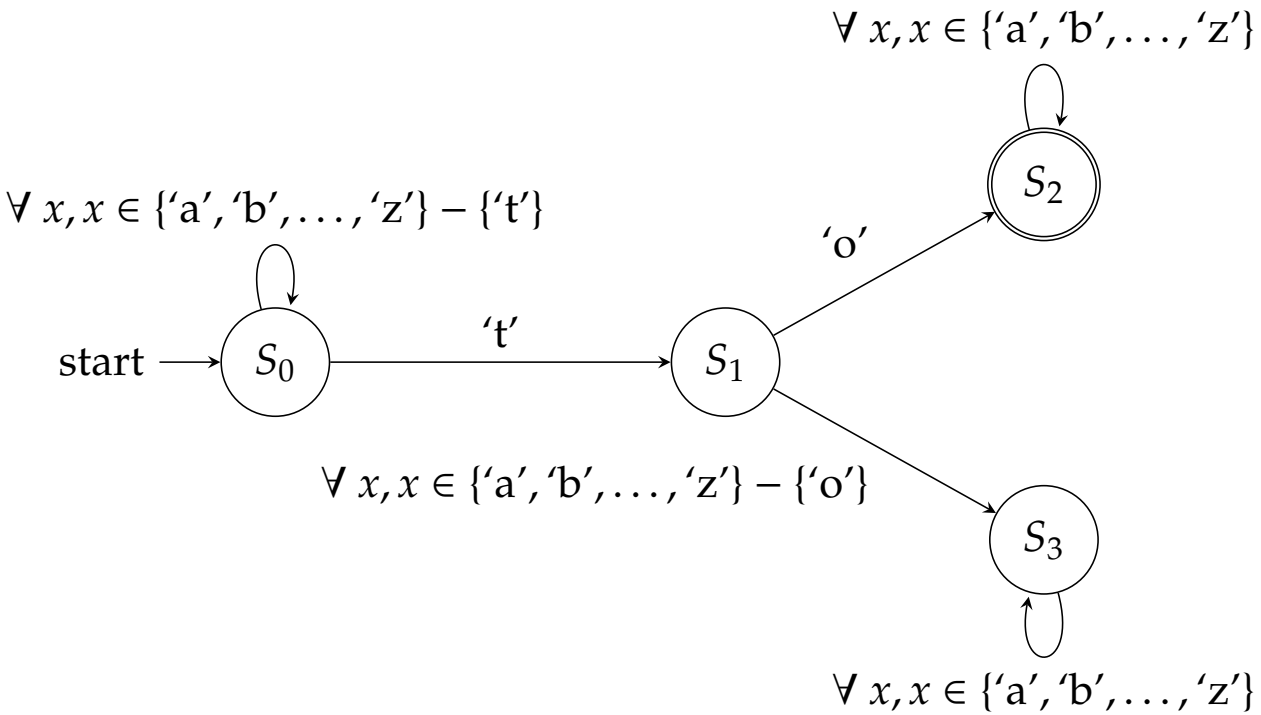
- Example [4, Example 1.53]: if $\Sigma = \{\text{'a'}, \text{'b'}, \ldots, \text{'z'}\}$, then

$$\texttt{grep -E '.*to+.*'} \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a line read from } \texttt{stdin} \text{ containing} \\ \text{a 't' followed by at least one 'o' character} \end{array} \right\}$$

- Example [4, Example 1.53]: if $\Sigma = \{\text{'a'}, \text{'b'}, \ldots, \text{'z'}\}$, then

$$\texttt{grep -E '.*to+.*'} \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a line read from } \texttt{stdin} \text{ containing} \\ \text{a 't' followed by at least one 'o' character} \end{array} \right\}$$

which can be realised using



- Example [4, Example 1.53]: if $\Sigma = \{\text{'a'}, \text{'b'}, \ldots, \text{'z'}\}$, then

$$\texttt{grep -E '.*to+.*'} \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a line read from } \texttt{stdin} \text{ containing} \\ \text{a 't' followed by at least one 'o' character} \end{array} \right\}$$

which can be realised using

```
1  forall lines X read from stdin do
3  |  Q ← s
4  |  for i = 0 upto n − 1 do
5  |  |  Q ← δ(Q, X_i)
6  |  end
7  |  if Q ∈ A then
8  |  |  print line X to stdout
10 |  end
11 end
```

- Example [4, Example 1.53]: if $\Sigma = \{\text{'a'}, \text{'b'}, \ldots, \text{'z'}\}$, then

$$\texttt{grep -E '.*to+.*'} \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a line read from } \texttt{stdin} \text{ containing} \\ \text{a 't' followed by at least one 'o' character} \end{array} \right\}$$

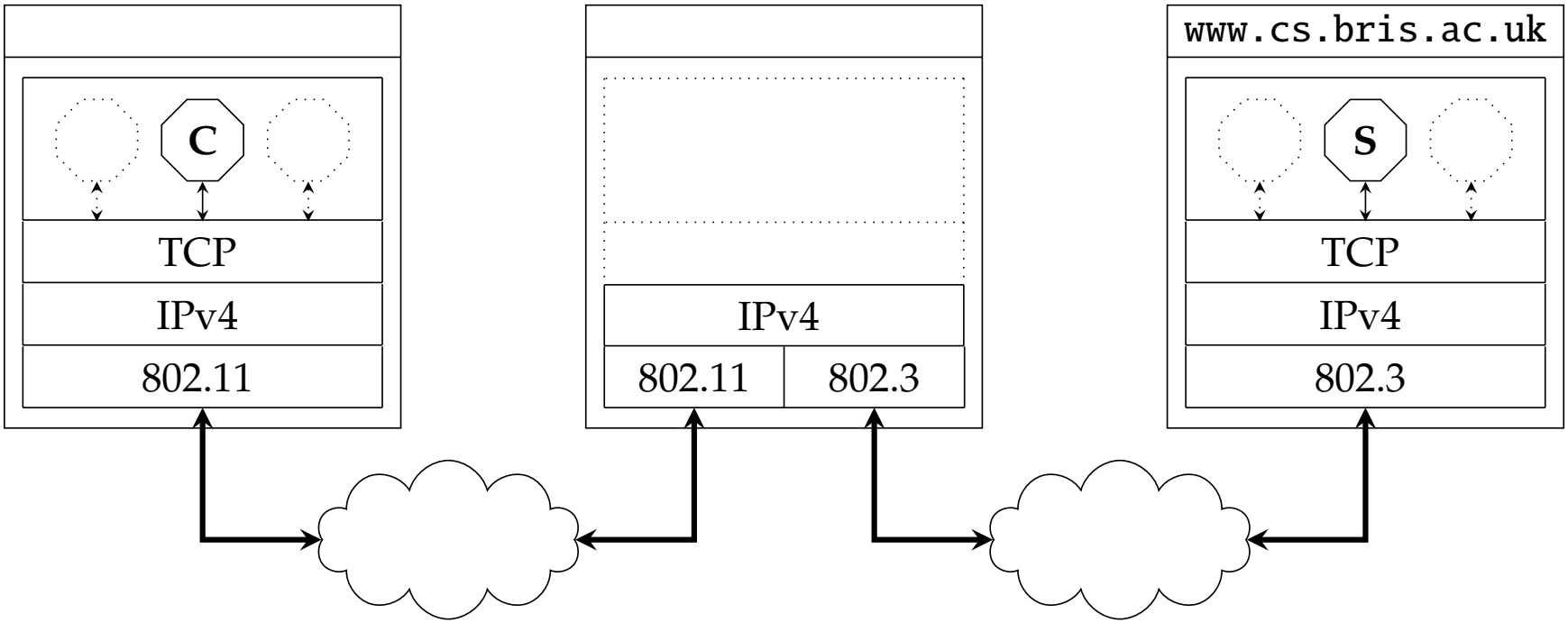
which can be realised using

```
1  void grep() {
2    char X[ 1024 ];
3
4    while( NULL != fgets( X, 1024, stdin ) ) {
5      int n = strlen( X ), Q = start;
6
7      if( X[ n - 1 ] == '\n' ) {
8        X[ n - 1 ] = '\0'; n--;
9      }
10
11     for( int i = 0; i < n; i++ ) {
12       Q = delta[ Q ][ X[ i ] ];
13     }
14
15     if( accept[ Q ] ) {
16       fprintf( stdout, "%s\n", X );
17     }
18   }
19 }
```

Part 2.1: in practice, application (4)

Example #2: networked communication via TCP ⇝ FSMs as controllers

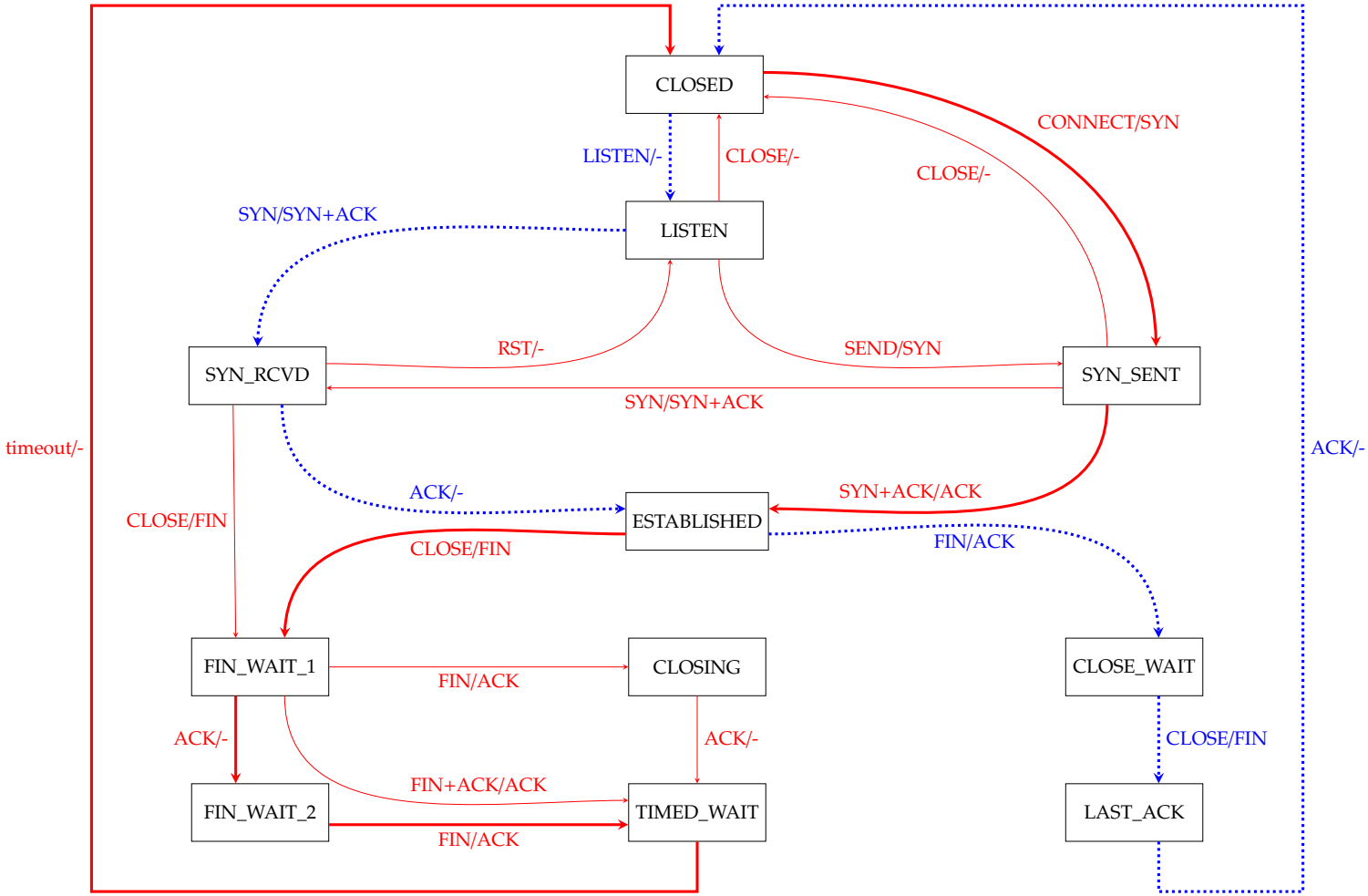
- Context:



Part 2.1: in practice, application (5)

Example #2: networked communication via TCP ⇝ FSMs as controllers

- Example:



Part 2.1: in practice, application (6)

Example #3: typical video game "loop" ⇝ FSMs as systems

- Context:

Algorithm

```
1 reset the game state
2 while ¬ game over do
3 |   read control pad (e.g., check if button pressed)
4 |   update game state (e.g., move player)
5 |   produce graphics and/or sound
6 end
```

Part 2.1: in practice, application (6)

Example #3: typical video game "loop" $\rightsquigarrow$ FSMs as systems

- Context:

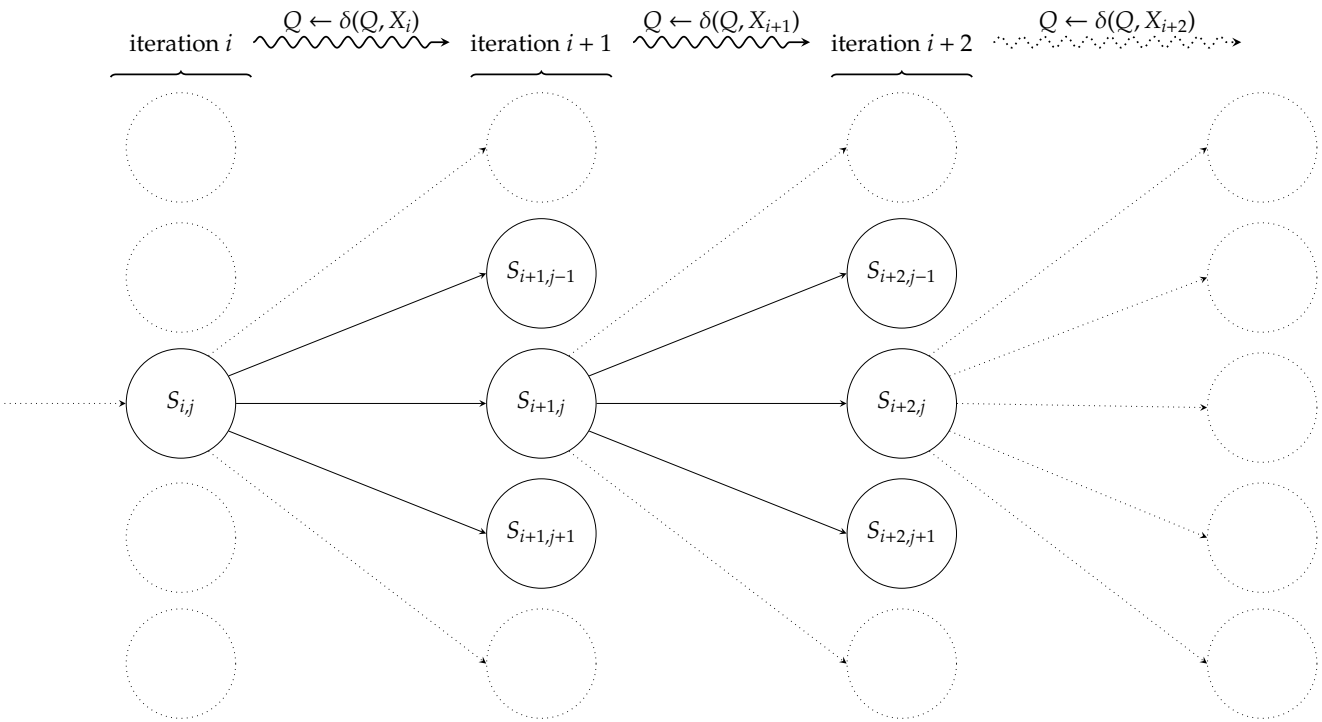
Algorithm

$Q \leftarrow s$
while $Q \notin A$ **do**
  $X_i \leftarrow$ control pad
  $Q \leftarrow \delta(Q, X_i)$
  $\{\text{graphics}, \text{sound}\} \leftarrow \omega(Q)$
end

- Example:

$$\begin{aligned} \text{iterations of game loop} &\rightsquigarrow \text{game tree} \\ &\simeq \text{state space} \end{aligned}$$

i.e.,



which is most obvious with respect to turn-based games (e.g., chess).

- Recall:

Definition

A (deterministic) **Finite State Machine (FSM)** is a tuple

$$C = (S, s, A, \Sigma, \Gamma, \delta, \omega)$$

including

1. S, a finite set of **states** that includes a **start state** $s \in S$,
2. $A \subseteq S$, a finite set of **accepting states**,
3. an **input alphabet** Σ and an **output alphabet** Γ,
4. a **transition function**

   $$\delta : S \times \Sigma \to S$$

   and
5. an **output function**

   $$\omega : S \to \Gamma$$

   in the case of a **Moore FSM**, or

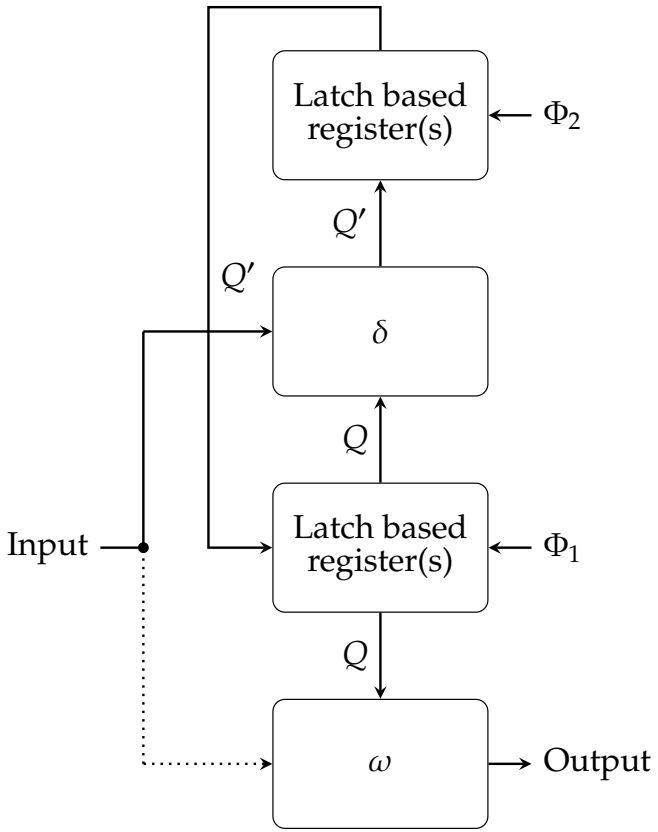
   $$\omega : S \times \Sigma \to \Gamma$$

   in the case of a **Mealy FSM**,

noting an **empty** input denoted ϵ allows a transition that can *always* occur.

- Concept:

Algorithm (latch version)

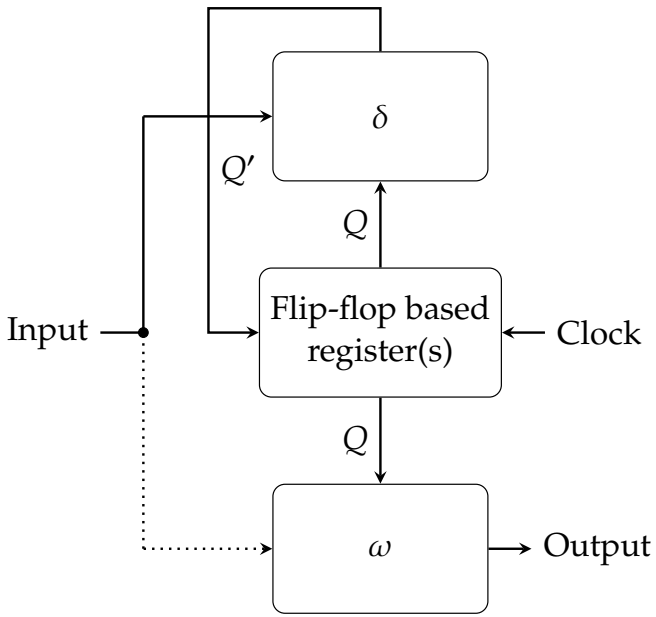


- Note that
  1. the state is retained in a register (i.e., a group of latches, resp. flip-flops),
  2. δ and ω are simply combinatorial logic,
  3. within the current clock cycle
     - ω computes the output from the current state and input, and
     - δ computes the next state from the current state and input,
  4. the next state is latched by an appropriate feature (i.e., level, resp. edge) in the clock

  i.e., it's a *computer* we can *build*!

- Concept:

Algorithm (flip-flop version)

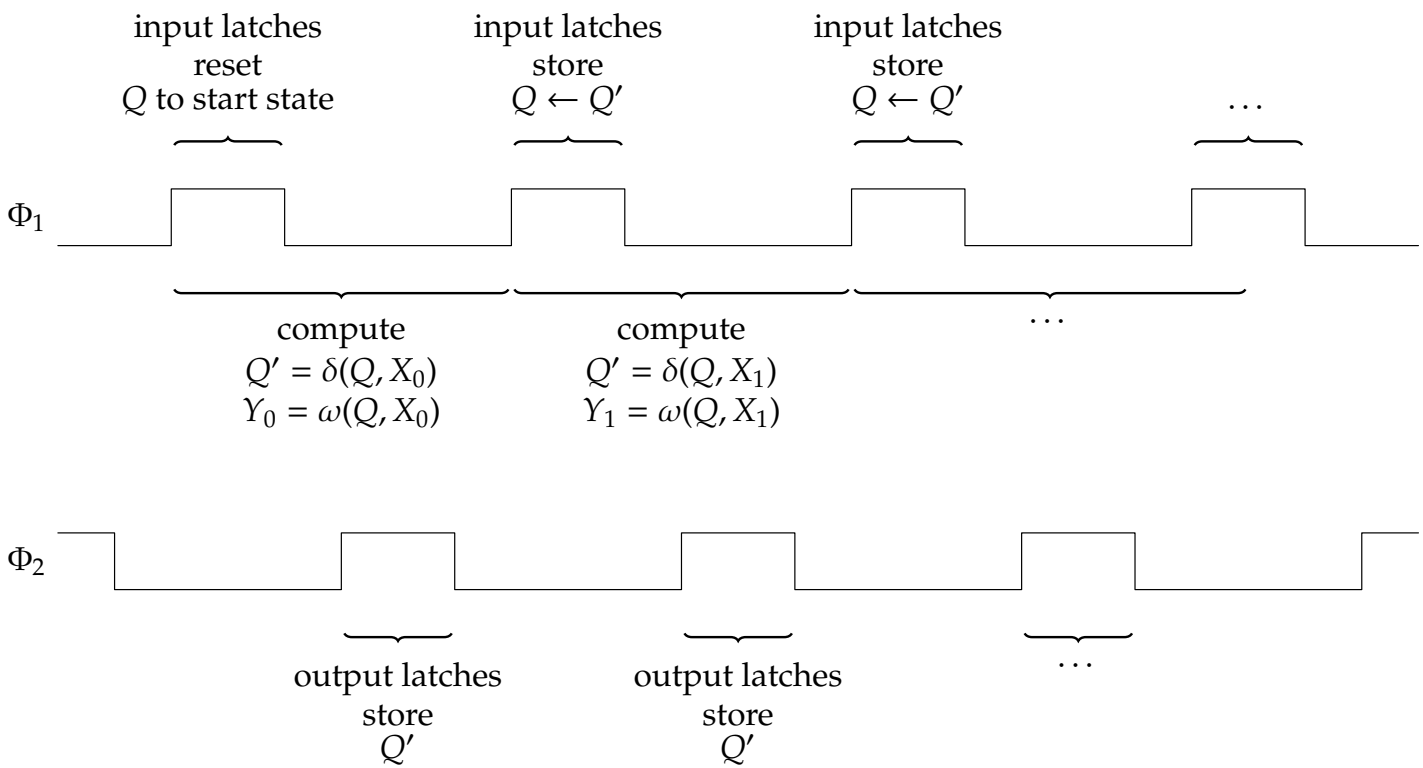


- Note that
  1. the state is retained in a register (i.e., a group of latches, resp. flip-flops),
  2. δ and ω are simply combinatorial logic,
  3. within the current clock cycle
     - ω computes the output from the current state and input, and
     - δ computes the next state from the current state and input,
  4. the next state is latched by an appropriate feature (i.e., level, resp. edge) in the clock

  i.e., it's a *computer* we can *build*!

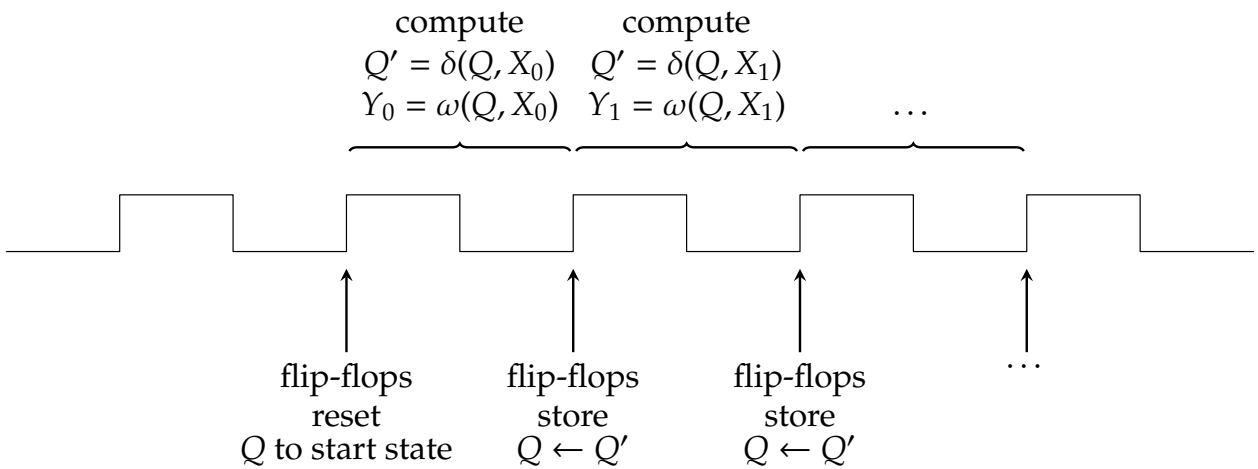
- Concept:

Example (latch version: input $X = \langle X_0, X_1, \ldots \rangle$, output $Y = \langle Y_0, Y_1, \ldots \rangle$)



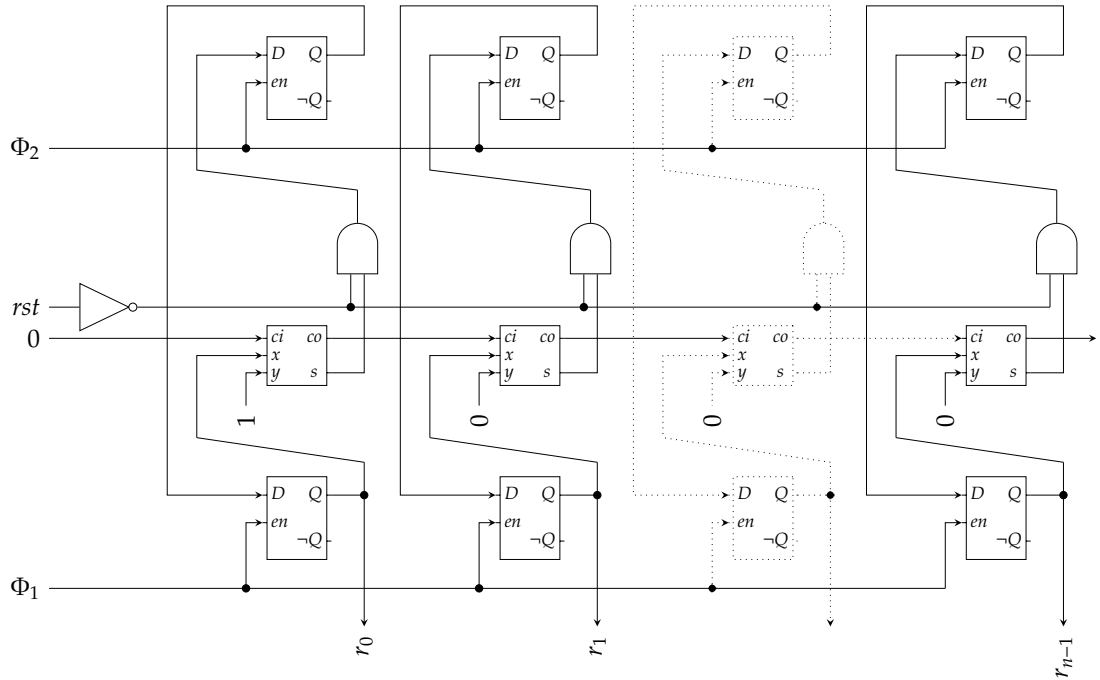
- Concept:

Example (flip-flop version: input $X = \langle X_0, X_1, \ldots \rangle$, output $Y = \langle Y_0, Y_1, \ldots \rangle$)



- Concept: this *should* sound familar, because from

Circuit (latch version)

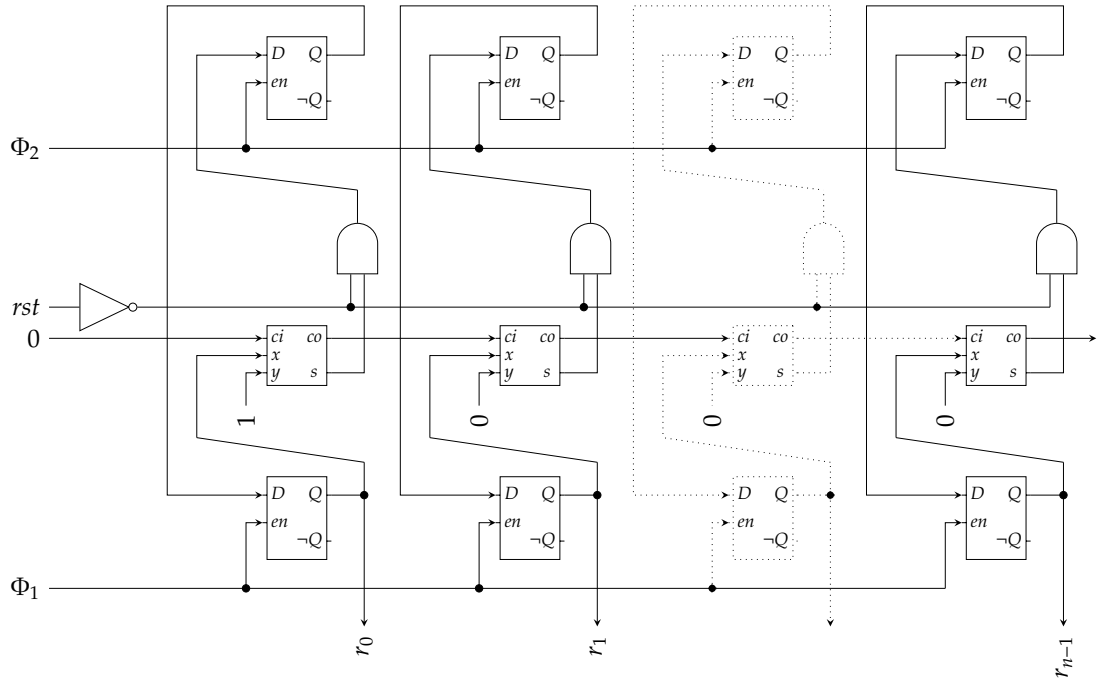


it *now* becomes clear that

- 2^n states, labelled S_0 through S_{2^n-1}; state S_i represented as (unsigned) n-bit integer i,
- the start state is $s = S_0$ and there are no accepting states (so $A = \emptyset$),

- Concept: this *should* sound familar, because from

Circuit (latch version)



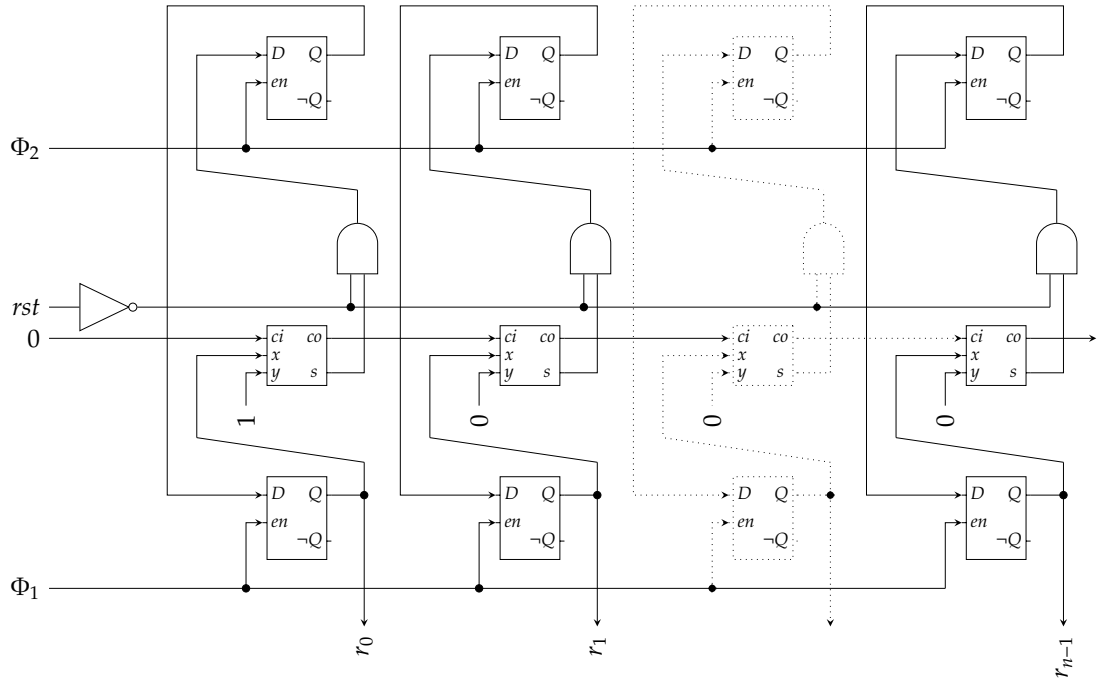
it *now* becomes clear that

- the δ function is

$$Q' \leftarrow \delta(Q, rst) = \begin{cases} Q + 1 \pmod{2^n} & \text{if } rst = 0 \\ 0 & \text{if } rst = 1 \end{cases}$$

- Concept: this *should* sound familar, because from

Circuit (latch version)

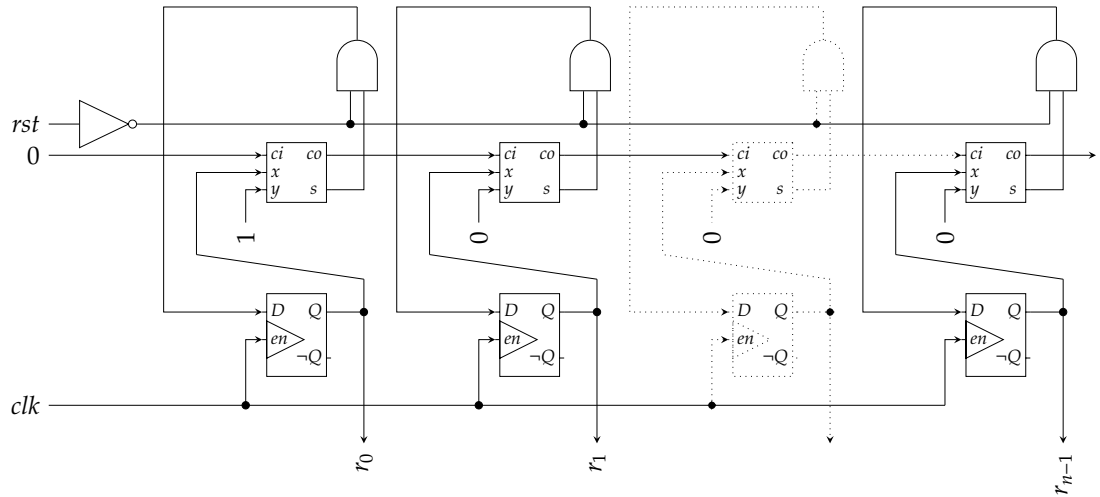


it *now* becomes clear that

- the ω function is $r \leftarrow \omega(Q) = Q$.

- Concept: this *should* sound familar, because from

Circuit (flip-flop version)

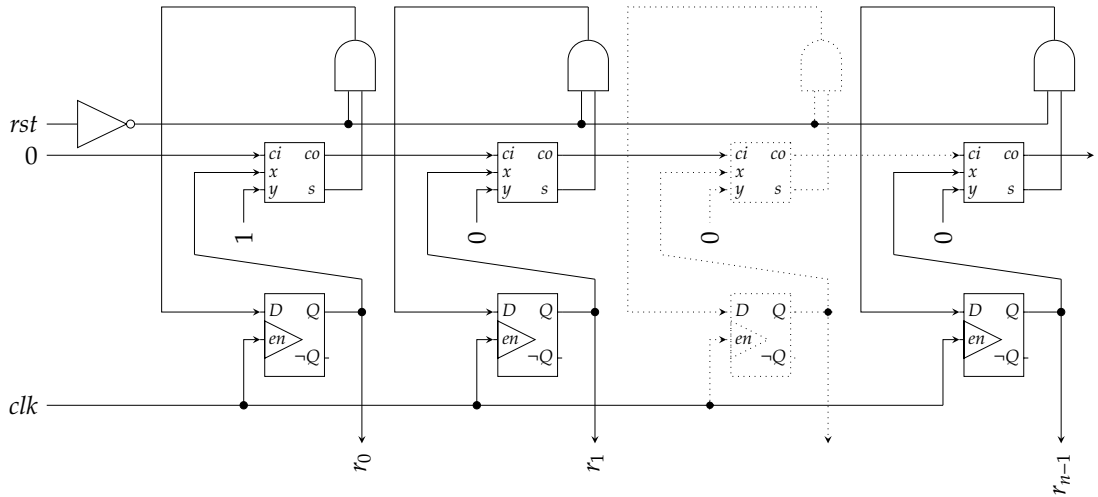


it *now* becomes clear that

- 2^n states, labelled S_0 through S_{2^n-1}; state S_i represented as (unsigned) n-bit integer i,
- the start state is $s = S_0$ and there are no accepting states (so $A = \emptyset$),

- Concept: this *should* sound familar, because from

Circuit (flip-flop version)



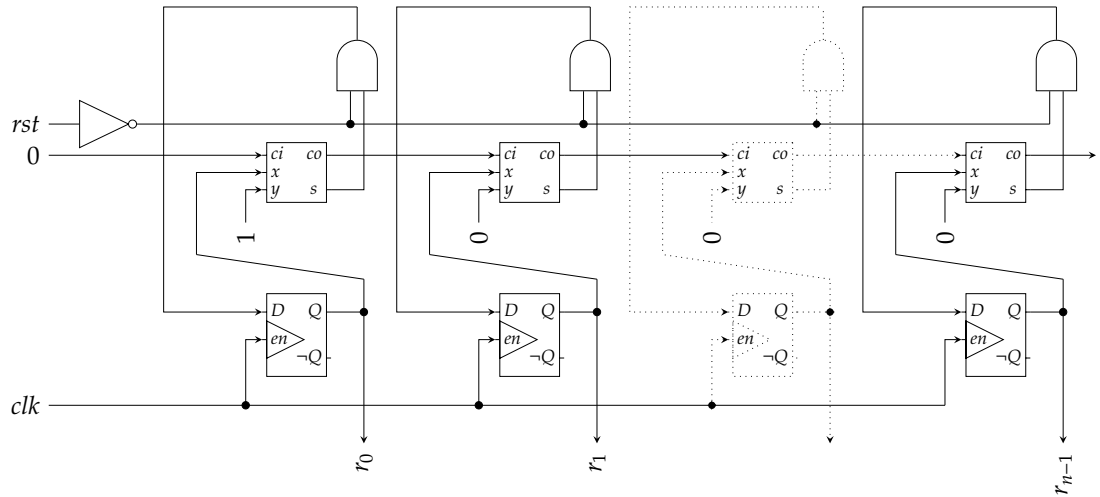
it *now* becomes clear that

- the δ function is

$$Q' \leftarrow \delta(Q, rst) = \begin{cases} Q + 1 \pmod{2^n} & \text{if } rst = 0 \\ 0 & \text{if } rst = 1 \end{cases}$$

- Concept: this *should* sound familar, because from

Circuit (flip-flop version)



it *now* becomes clear that

- the ω function is $r \leftarrow \omega(Q) = Q$.

- Concept to solve a concrete problem, we follow a (fairly) standard sequence of steps

Algorithm

1. Count the number of states required, and give each state an abstract label.
2. Describe the state transition and output functions using a tabular or diagrammatic approach.
3. Perform state assignment, i.e., decide how concrete values will represent the abstract labels, allocating appropriate register(s) to hold the state.
4. Express the functions δ and ω as (optimised) Boolean expressions, i.e., combinatorial logic.
5. Place the registers and combinatorial logic into the framework.

noting that it's common to

- include a **reset** input that (re)initialises the FSM into the start state,
- replace the accepting state(s) with an **idle** or **error** state since "halting" doesn't make sense in hardware, and
- use the FSM to control an associated data-path using the outputs, rather than (necessarily) solve some problem outright.

- Concept: we can optimise the state representation based on use of it, e.g.,
  1. a **binary encoding** represents the i-th of n states as a $(\lceil \log_2(n) \rceil)$-bit unsigned integer i, e.g.,

$$\begin{aligned} S_0 &\mapsto \langle 0,0,0 \rangle \\ S_1 &\mapsto \langle 1,0,0 \rangle \\ S_2 &\mapsto \langle 0,1,0 \rangle \\ S_3 &\mapsto \langle 1,1,0 \rangle \\ S_4 &\mapsto \langle 0,0,1 \rangle \\ S_5 &\mapsto \langle 1,0,1 \rangle \end{aligned}$$

  2. a **one-hot encoding** represents the i-th of n states as a sequence X such that $X_i = 1$ and $X_j = 0$ for $j \neq i$, e.g.,

$$\begin{aligned} S_0 &\mapsto \langle 1,0,0,0,0,0 \rangle \\ S_1 &\mapsto \langle 0,1,0,0,0,0 \rangle \\ S_2 &\mapsto \langle 0,0,1,0,0,0 \rangle \\ S_3 &\mapsto \langle 0,0,0,1,0,0 \rangle \\ S_4 &\mapsto \langle 0,0,0,0,1,0 \rangle \\ S_5 &\mapsto \langle 0,0,0,0,0,1 \rangle \end{aligned}$$

     noting that we have a larger state (i.e., n bits instead of $\lceil \log_2(n) \rceil$), *but*
     - transition between states is easier, *and*
     - switching behaviour (and hence power consumption) is reduced.

Part 2.2: in practice, implementation (11)

Example #2: a modulo 6 ascending *or* decending counter, with cycle alert

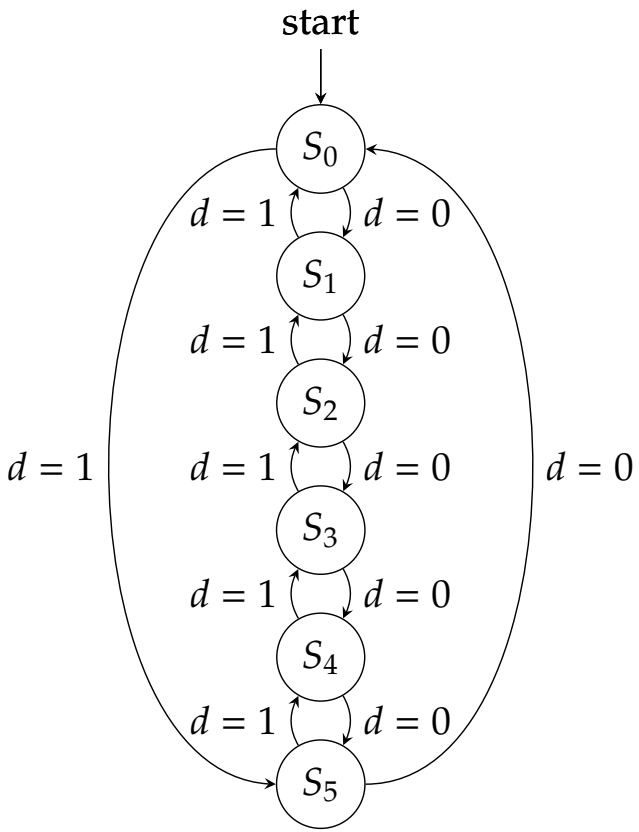
- Problem: design an FSM that
  1. acts as a cyclic counter modulo $n = 6$ (versus 2^n),
  2. has an input d which selects between increment and decrement, and
  3. has an output f which signals when a cycle occurs.

- Solution:

Algorithm (tabular)

	δ		ω		
Q	Q'		r	f	
	$d=0$	$d=1$		$d=0$	$d=1$
S_0	S_1	S_5	0	0	1
S_1	S_2	S_0	1	0	0
S_2	S_3	S_1	2	0	0
S_3	S_4	S_2	3	0	0
S_4	S_5	S_3	4	0	0
S_5	S_0	S_4	5	1	0

Algorithm (diagram)



- Solution:
  - there are 6 abstract labels

$$\begin{array}{lcl} S_0 & \mapsto & 0 \\ S_1 & \mapsto & 1 \\ S_2 & \mapsto & 2 \\ S_3 & \mapsto & 3 \\ S_4 & \mapsto & 4 \\ S_5 & \mapsto & 5 \end{array}$$

  we can represent using 6 concrete values, e.g.,

$$\begin{array}{lcll} S_0 & \mapsto & \langle 0,0,0 \rangle & \equiv & 000_{(2)} \\ S_1 & \mapsto & \langle 1,0,0 \rangle & \equiv & 001_{(2)} \\ S_2 & \mapsto & \langle 0,1,0 \rangle & \equiv & 010_{(2)} \\ S_3 & \mapsto & \langle 1,1,0 \rangle & \equiv & 011_{(2)} \\ S_4 & \mapsto & \langle 0,0,1 \rangle & \equiv & 100_{(2)} \\ S_5 & \mapsto & \langle 1,0,1 \rangle & \equiv & 101_{(2)} \end{array}$$

  - since $2^3 = 8 > 6$, we can capture each of

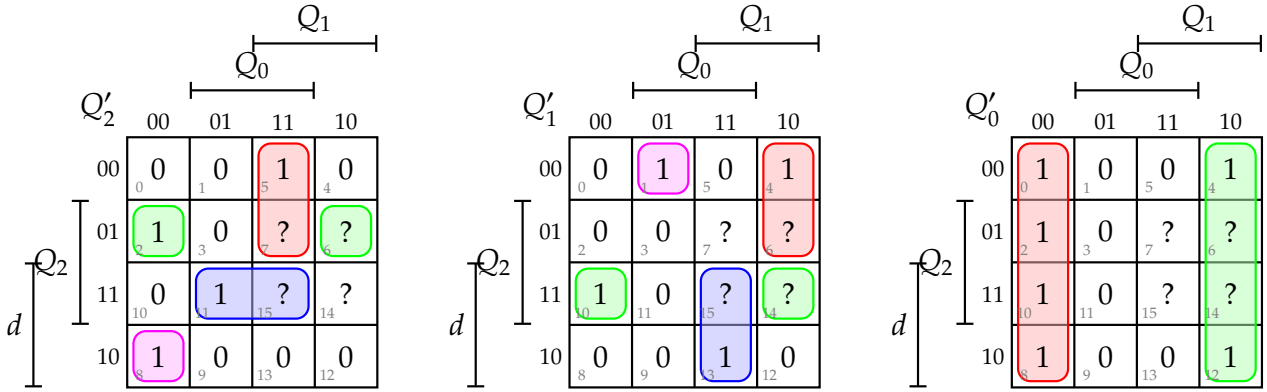
1. $Q = \langle Q_0, Q_1, Q_2 \rangle \equiv$ the current state
2. $Q' = \langle Q'_0, Q'_1, Q'_2 \rangle \equiv$ the next state

  in a 3-bit register (i.e., via 3 latches or flip-flops).

- Solution:
  - rewriting the abstract labels yields the following concrete truth table

				δ			ω			
d	Q_2	Q_1	Q_0	Q'_2	Q'_1	Q'_0	r_2	r_1	r_0	f
0	0	0	0	0	0	1	0	0	0	0
0	0	0	1	0	1	0	0	0	1	0
0	0	1	0	0	1	1	0	1	0	0
0	0	1	1	1	0	0	0	1	1	0
0	1	0	0	1	0	1	1	0	0	0
0	1	0	1	0	0	0	1	0	1	1
0	1	1	0	**?**	**?**	**?**	**?**	**?**	**?**	**?**
0	1	1	1	**?**	**?**	**?**	**?**	**?**	**?**	**?**
1	0	0	0	1	0	1	0	0	0	1
1	0	0	1	0	0	0	0	0	1	0
1	0	1	0	0	0	1	0	1	0	0
1	0	1	1	0	1	0	0	1	1	0
1	1	0	0	0	1	1	1	0	0	0
1	1	0	1	1	0	0	1	0	1	0
1	1	1	0	**?**	**?**	**?**	**?**	**?**	**?**	**?**
1	1	1	1	**?**	**?**	**?**	**?**	**?**	**?**	**?**

- Solution:
  - the truth table can be translated into



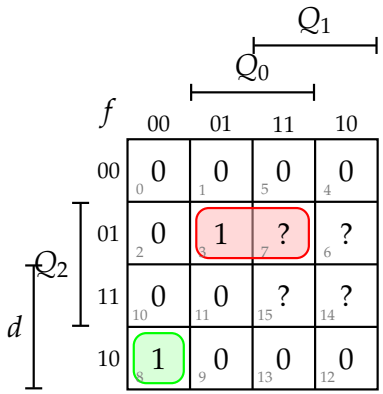
  - doing so yields the following Boolean expressions for δ:

$$
\begin{aligned}
Q'_2 = (\ & \neg d & & & \wedge & Q_1 & \wedge & Q_0 &) \vee \\
(\ & \neg d & \wedge & Q_2 & & & \wedge & \neg Q_0 &) \vee \\
(\ & d & \wedge & Q_2 & & & \wedge & Q_0 &) \vee \\
(\ & d & \wedge & \neg Q_2 & \wedge & \neg Q_1 & \wedge & \neg Q_0 &)
\end{aligned}
$$

$$
\begin{aligned}
Q'_1 = (\ & \neg d & \wedge & \neg Q_2 & \wedge & \neg Q_1 & \wedge & Q_0 &) \vee \\
(\ & \neg d & & & \wedge & Q_1 & \wedge & \neg Q_0 &) \vee \\
(\ & d & \wedge & Q_2 & & & \wedge & \neg Q_0 &) \vee \\
(\ & d & & & \wedge & Q_1 & \wedge & Q_0 &)
\end{aligned}
$$

$$
Q'_0 = (\ \neg Q_0\ )
$$

- Solution:
  - the truth table can be translated into



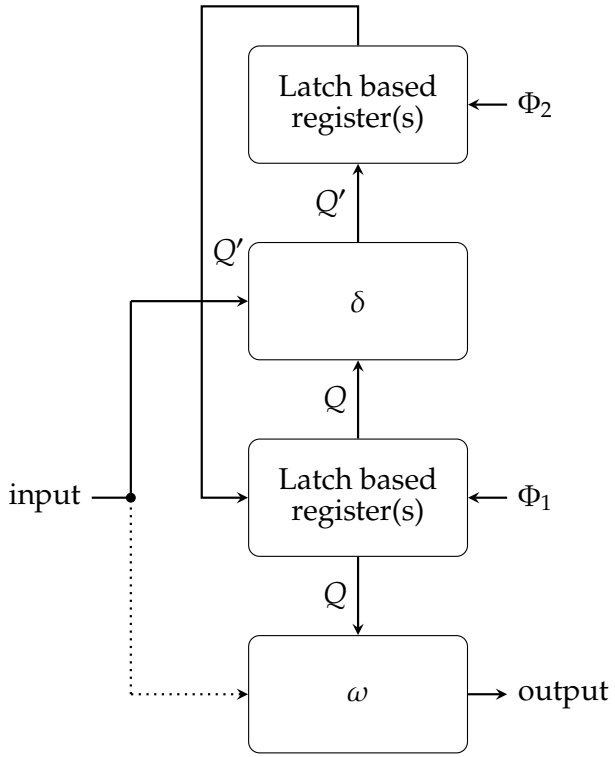
  - doing so yields the following Boolean expressions for ω:

$$f = (\ \neg d \ \wedge\ Q_2 \ \wedge\ Q_0\ ) \vee (\ d \ \wedge\ \neg Q_2 \ \wedge\ \neg Q_1 \ \wedge\ \neg Q_0\ )$$

Part 2.2: in practice, implementation (11)

Example #2: a modulo 6 ascending *or* decending counter, with cycle alert

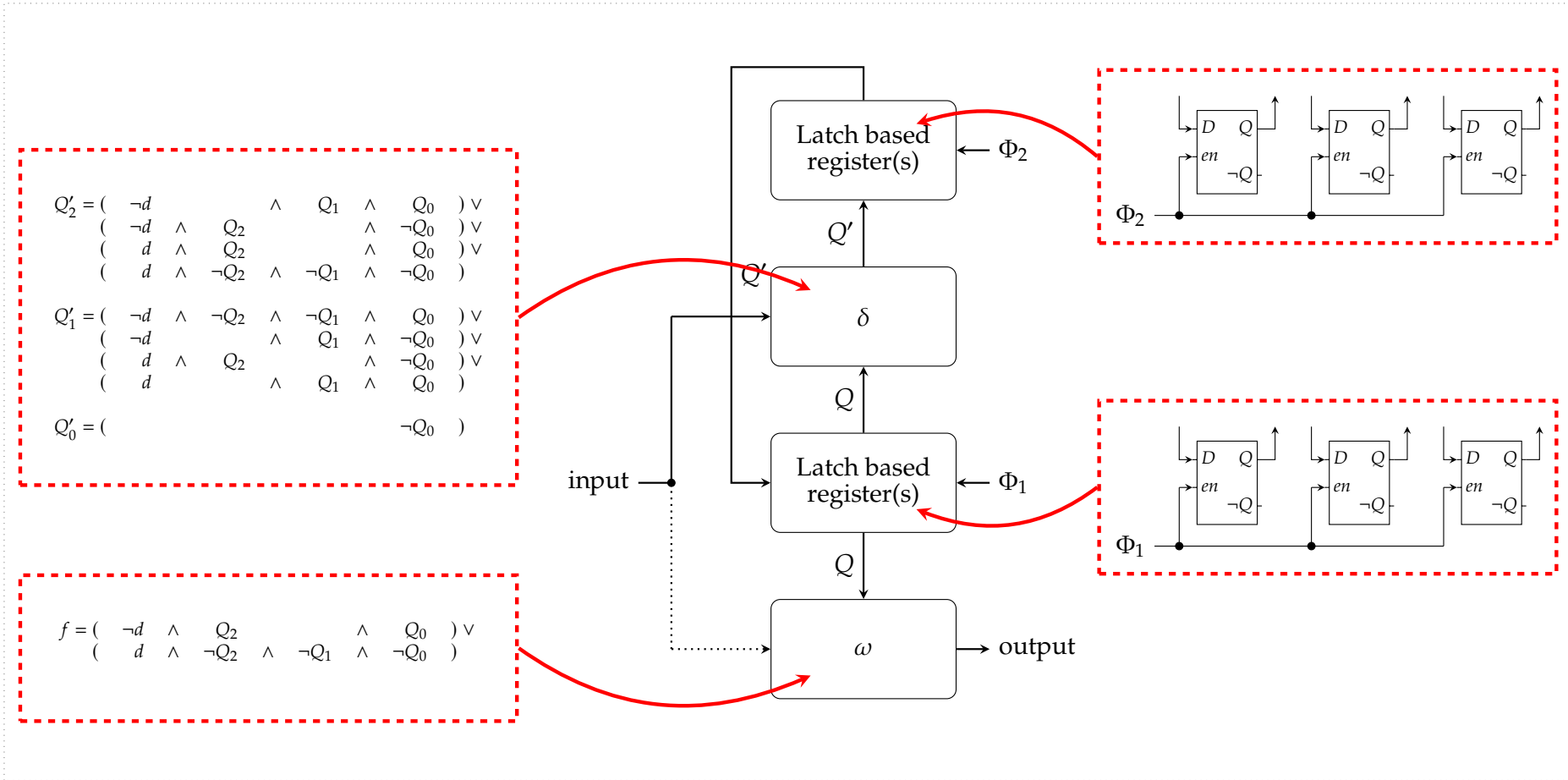
- Solution:



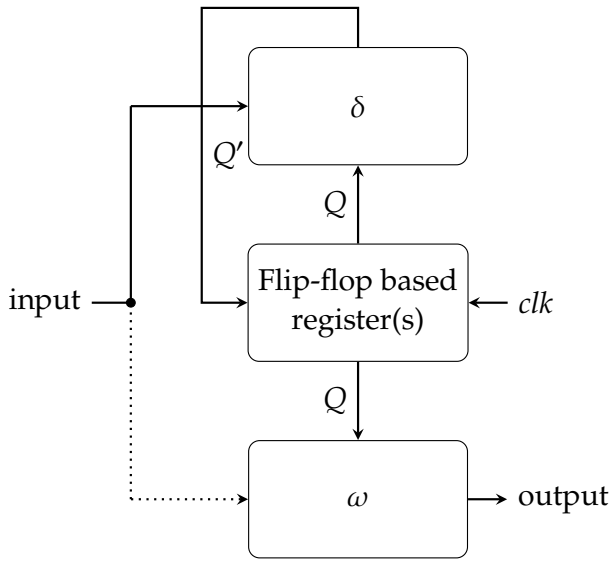
Part 2.2: in practice, implementation (11)

Example #2: a modulo 6 ascending *or* decending counter, with cycle alert

- Solution:



- Solution:

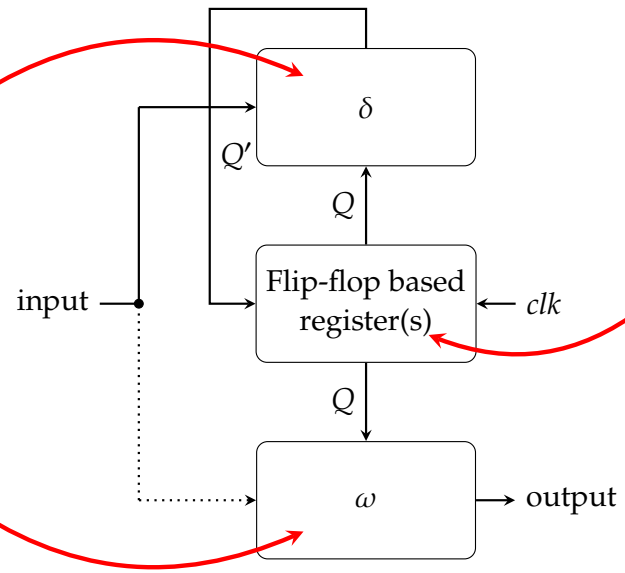


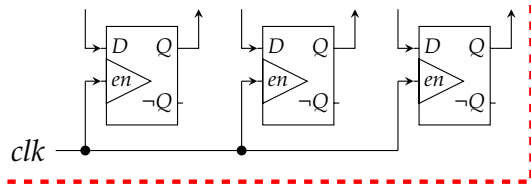
- Solution:

$$
\begin{aligned}
Q'_2 = (\ & \neg d & & & & \wedge & Q_1 & \wedge & Q_0 &) \vee \\
(\ & \neg d & \wedge & Q_2 & & \wedge & & \wedge & \neg Q_0 &) \vee \\
(\ & d & \wedge & Q_2 & & \wedge & & \wedge & Q_0 &) \vee \\
(\ & d & \wedge & \neg Q_2 & \wedge & \neg Q_1 & \wedge & \neg Q_0 &) &
\end{aligned}
$$

$$
\begin{aligned}
Q'_1 = (\ & \neg d & \wedge & \neg Q_2 & \wedge & \neg Q_1 & \wedge & Q_0 &) \vee \\
(\ & \neg d & & & \wedge & Q_1 & \wedge & \neg Q_0 &) \vee \\
(\ & d & \wedge & Q_2 & & & \wedge & \neg Q_0 &) \vee \\
(\ & d & & & \wedge & Q_1 & \wedge & Q_0 &)
\end{aligned}
$$

$$
Q'_0 = (\quad \neg Q_0 \quad)
$$





$$
\begin{aligned}
f = (\ & \neg d & \wedge & Q_2 & & & \wedge & Q_0 &) \vee \\
(\ & d & \wedge & \neg Q_2 & \wedge & \neg Q_1 & \wedge & \neg Q_0 &)
\end{aligned}
$$

- Problem: design an FSM that
  1. replicates the behaviour of a controlled loop counter, e.g., `i` within a C-style `for` loop such as

```
for( int i = m; i < n; i++ ) {
  ...
}
```

  2. has an interface that allows signalling for

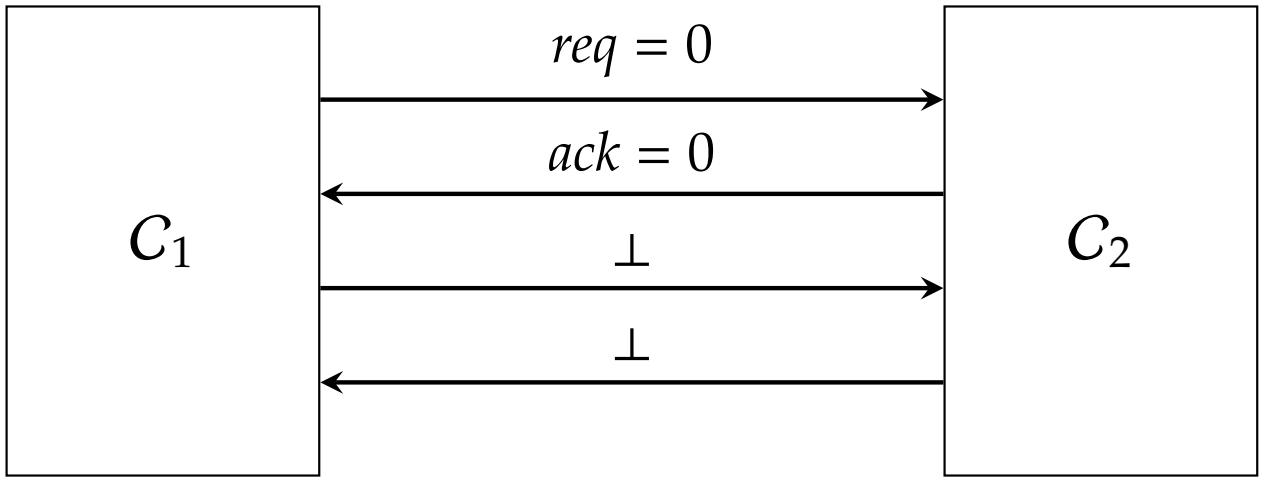
the start of iteration	≡	so `i = m`
the end of iteration	≡	when `i = n`

focused wlog. on 4-bit values of `i`, `m`, and `n`.

- Design:
  - given a user C_1 of some component C_2, how does
    - C_2 know when to start computation (e.g., when any input x is available), and
    - C_1 know when computation has finished (e.g., when any output $r = f(x)$ is available).
  - we could implement an the interface which
    1. uses a shared clock signal to synchronise events,
    2. uses a **control protocol**, e.g., via additional *req* (or **request**) and *ack* (or **acknowledge**) signals,
    3. ...

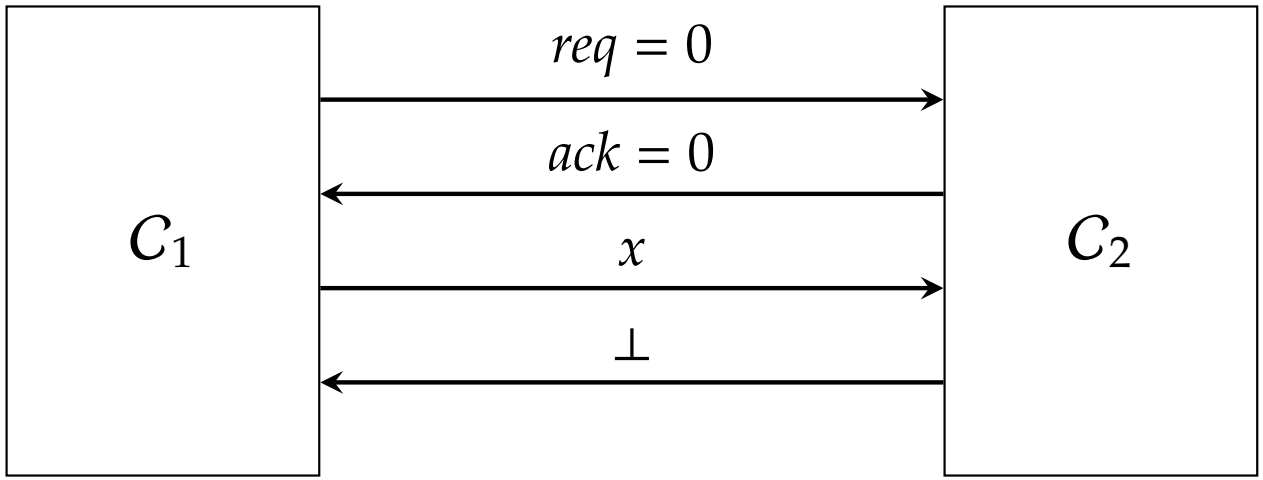
- Design:
  - given a user C_1 of some component C_2, how does
    - C_2 know when to start computation (e.g., when any input x is available), and
    - C_1 know when computation has finished (e.g., when any output $r = f(x)$ is available).
  - we could implement an the interface which
    1. uses a shared clock signal to synchronise events,
    2. uses a **control protocol**, e.g., via additional *req* (or **request**) and *ack* (or **acknowledge**) signals,
    3. ...
- Example:

Algorithm

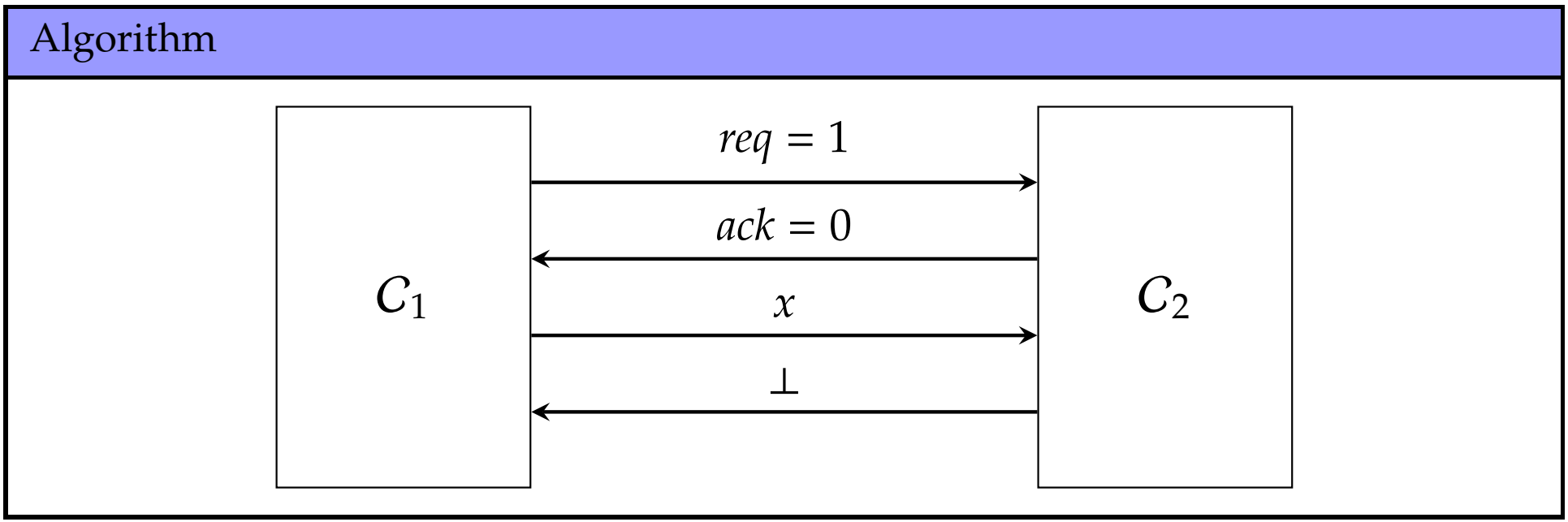


- Design:
  - given a user C_1 of some component C_2, how does
    - C_2 know when to start computation (e.g., when any input x is available), and
    - C_1 know when computation has finished (e.g., when any output $r = f(x)$ is available).
  - we could implement an the interface which
    1. uses a shared clock signal to synchronise events,
    2. uses a **control protocol**, e.g., via additional *req* (or **request**) and *ack* (or **acknowledge**) signals,
    3. ...
- Example:

Algorithm

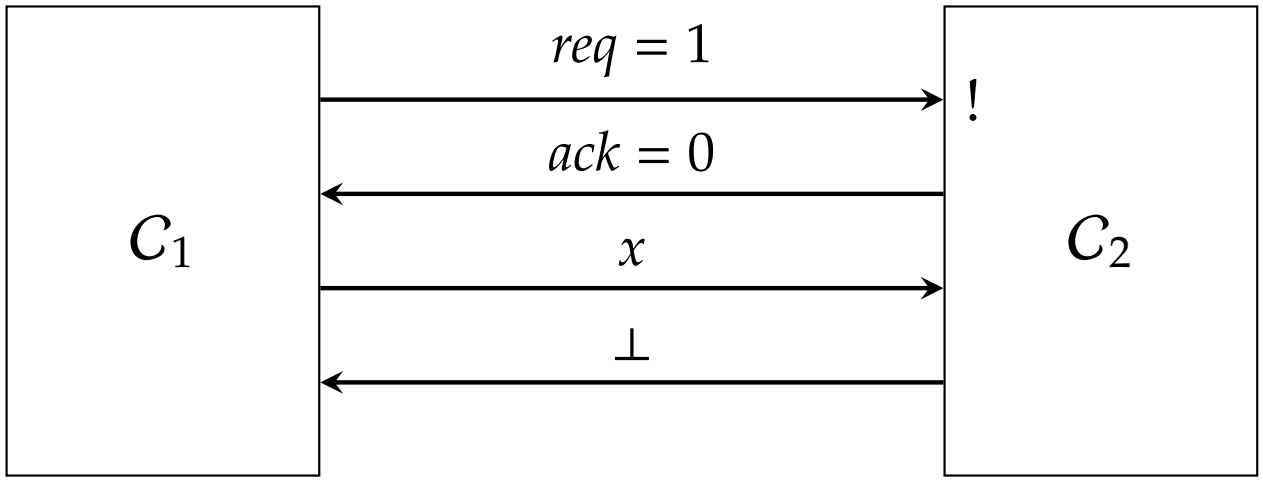


- Design:
  - given a user C_1 of some component C_2, how does
    - C_2 know when to start computation (e.g., when any input x is available), and
    - C_1 know when computation has finished (e.g., when any output $r = f(x)$ is available).
  - we could implement an the interface which
    1. uses a shared clock signal to synchronise events,
    2. uses a **control protocol**, e.g., via additional *req* (or **request**) and *ack* (or **acknowledge**) signals,
    3. ...
- Example:



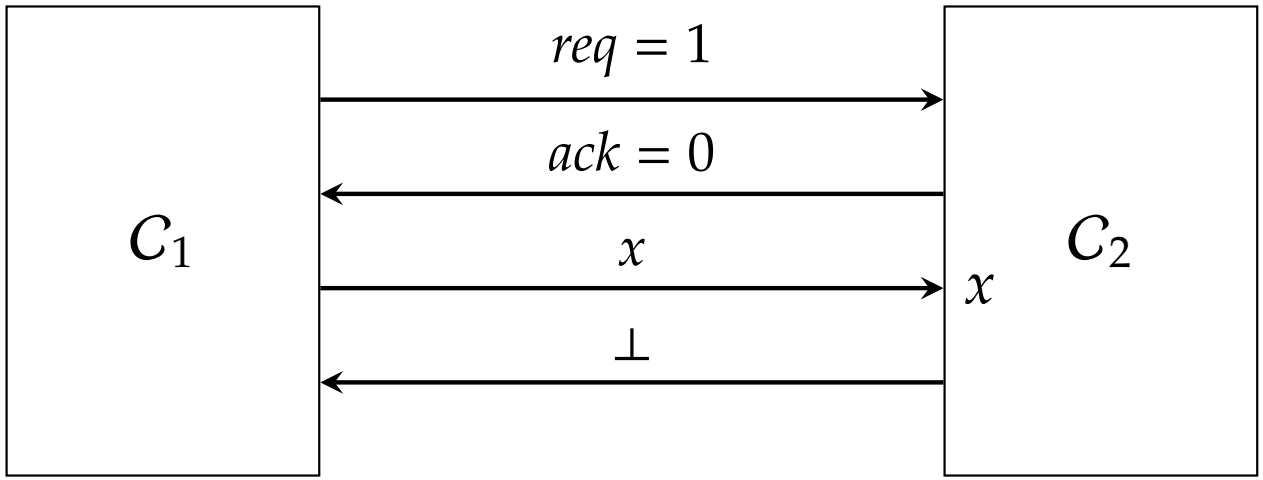
- Design:
  - given a user C_1 of some component C_2, how does
    - C_2 know when to start computation (e.g., when any input x is available), and
    - C_1 know when computation has finished (e.g., when any output $r = f(x)$ is available).
  - we could implement an the interface which
    1. uses a shared clock signal to synchronise events,
    2. uses a **control protocol**, e.g., via additional *req* (or **request**) and *ack* (or **acknowledge**) signals,
    3. ...
- Example:

Algorithm



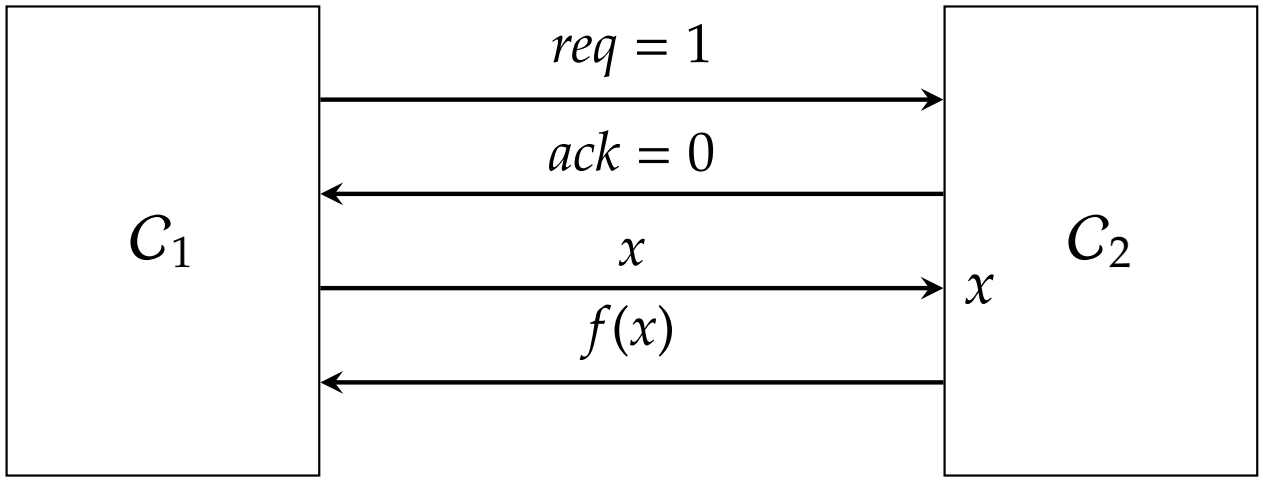
- Design:
  - given a user C_1 of some component C_2, how does
    - C_2 know when to start computation (e.g., when any input x is available), and
    - C_1 know when computation has finished (e.g., when any output $r = f(x)$ is available).
  - we could implement an the interface which
    1. uses a shared clock signal to synchronise events,
    2. uses a **control protocol**, e.g., via additional *req* (or **request**) and *ack* (or **acknowledge**) signals,
    3. ...
- Example:

Algorithm



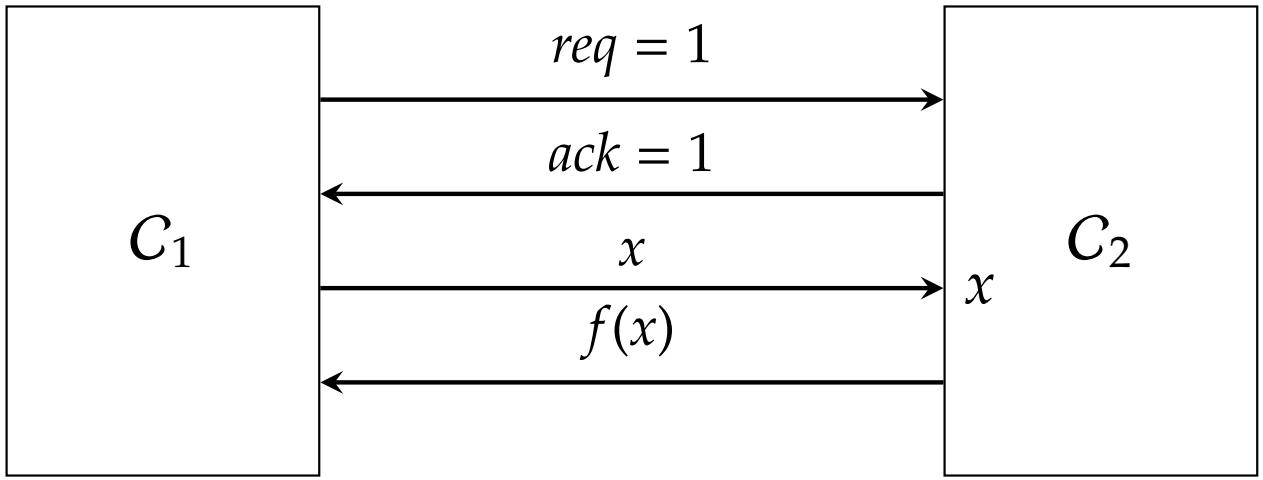
- Design:
  - given a user C_1 of some component C_2, how does
    - C_2 know when to start computation (e.g., when any input x is available), and
    - C_1 know when computation has finished (e.g., when any output $r = f(x)$ is available).
  - we could implement an the interface which
    1. uses a shared clock signal to synchronise events,
    2. uses a **control protocol**, e.g., via additional *req* (or **request**) and *ack* (or **acknowledge**) signals,
    3. ...
- Example:

Algorithm



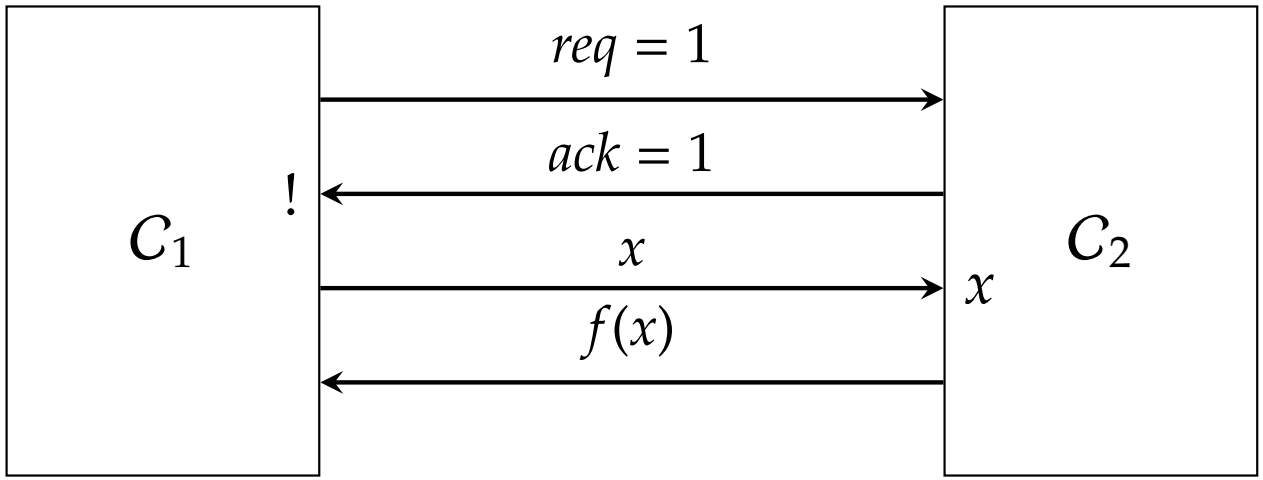
- Design:
  - given a user C_1 of some component C_2, how does
    - C_2 know when to start computation (e.g., when any input x is available), and
    - C_1 know when computation has finished (e.g., when any output $r = f(x)$ is available).
  - we could implement an the interface which
    1. uses a shared clock signal to synchronise events,
    2. uses a **control protocol**, e.g., via additional *req* (or **request**) and *ack* (or **acknowledge**) signals,
    3. ...
- Example:

Algorithm

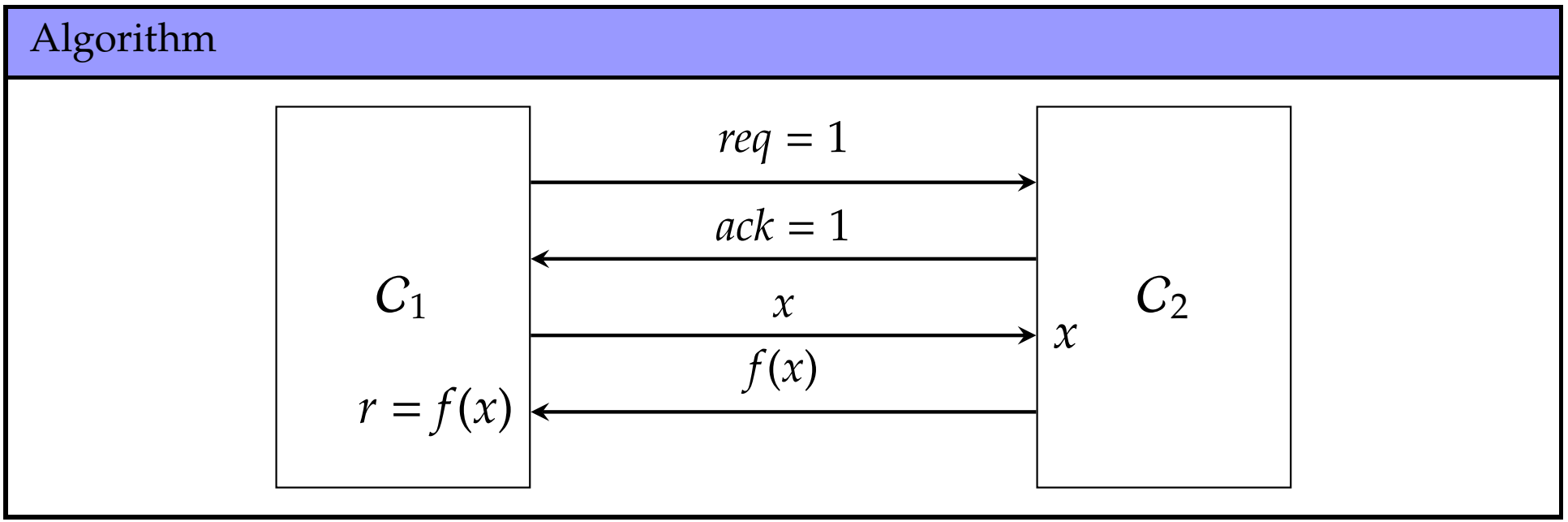


- Design:
  - given a user C_1 of some component C_2, how does
    - C_2 know when to start computation (e.g., when any input x is available), and
    - C_1 know when computation has finished (e.g., when any output $r = f(x)$ is available).
  - we could implement an the interface which
    1. uses a shared clock signal to synchronise events,
    2. uses a **control protocol**, e.g., via additional *req* (or **request**) and *ack* (or **acknowledge**) signals,
    3. ...
- Example:

Algorithm

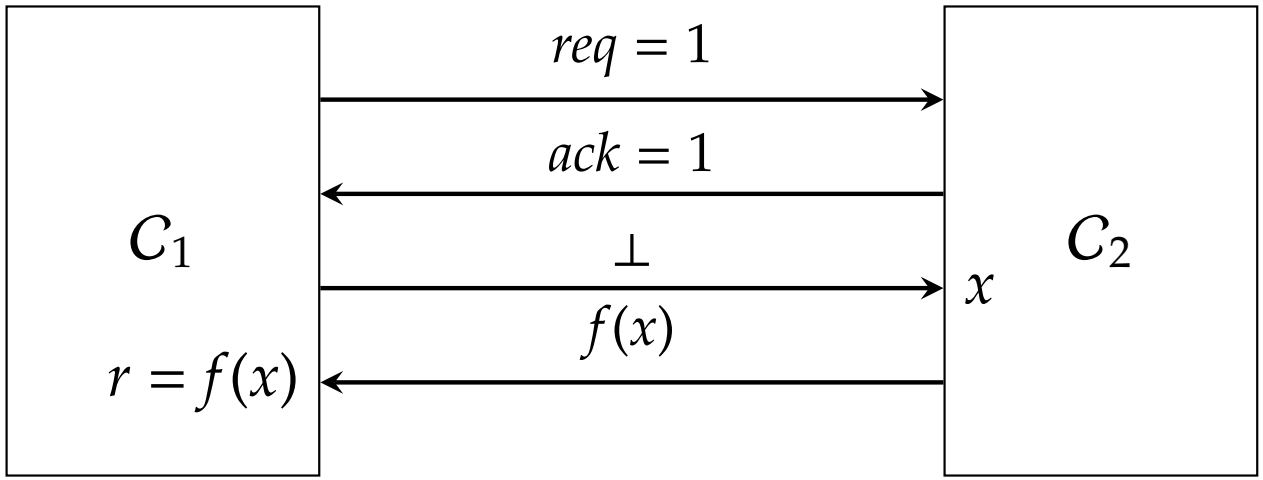


- Design:
  - given a user C_1 of some component C_2, how does
    - C_2 know when to start computation (e.g., when any input x is available), and
    - C_1 know when computation has finished (e.g., when any output $r = f(x)$ is available).
  - we could implement an the interface which
    1. uses a shared clock signal to synchronise events,
    2. uses a **control protocol**, e.g., via additional *req* (or **request**) and *ack* (or **acknowledge**) signals,
    3. ...
- Example:



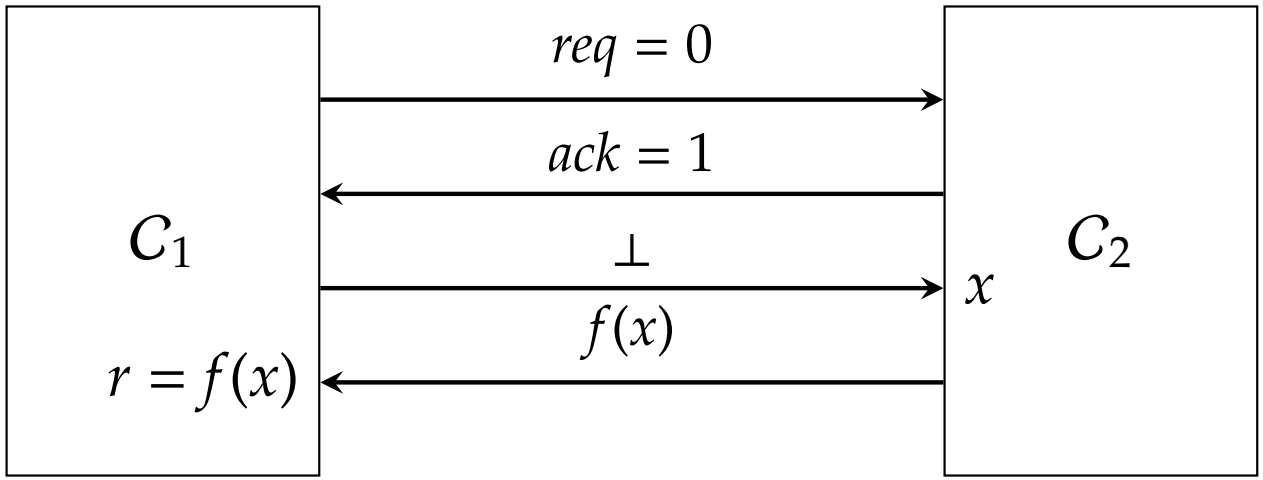
- Design:
  - given a user C_1 of some component C_2, how does
    - C_2 know when to start computation (e.g., when any input x is available), and
    - C_1 know when computation has finished (e.g., when any output $r = f(x)$ is available).
  - we could implement an the interface which
    1. uses a shared clock signal to synchronise events,
    2. uses a **control protocol**, e.g., via additional *req* (or **request**) and *ack* (or **acknowledge**) signals,
    3. ...
- Example:

Algorithm



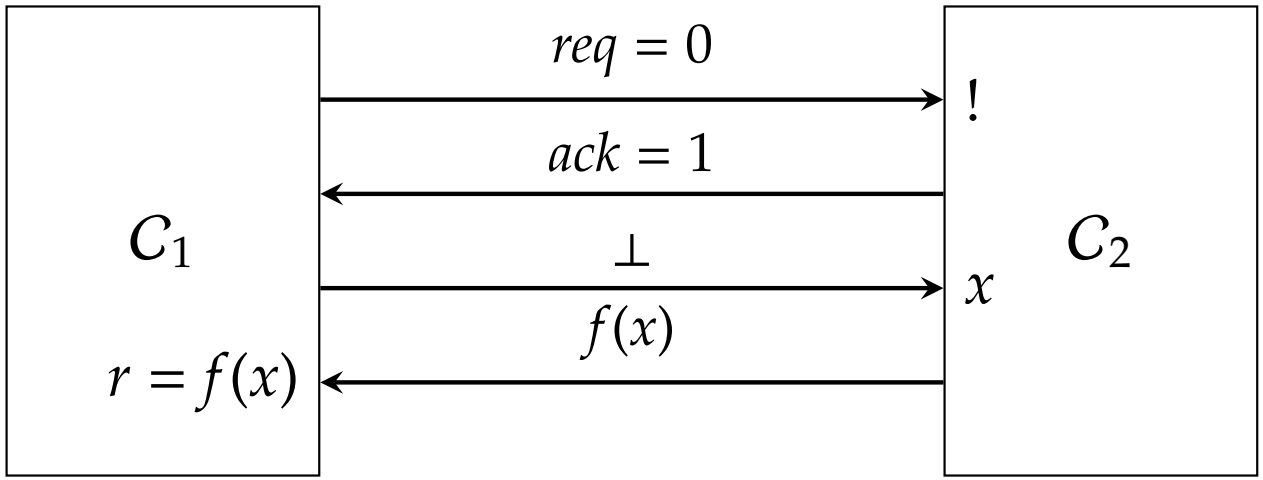
- Design:
  - given a user C_1 of some component C_2, how does
    - C_2 know when to start computation (e.g., when any input x is available), and
    - C_1 know when computation has finished (e.g., when any output $r = f(x)$ is available).
  - we could implement an the interface which
    1. uses a shared clock signal to synchronise events,
    2. uses a **control protocol**, e.g., via additional *req* (or **request**) and *ack* (or **acknowledge**) signals,
    3. ...
- Example:

Algorithm

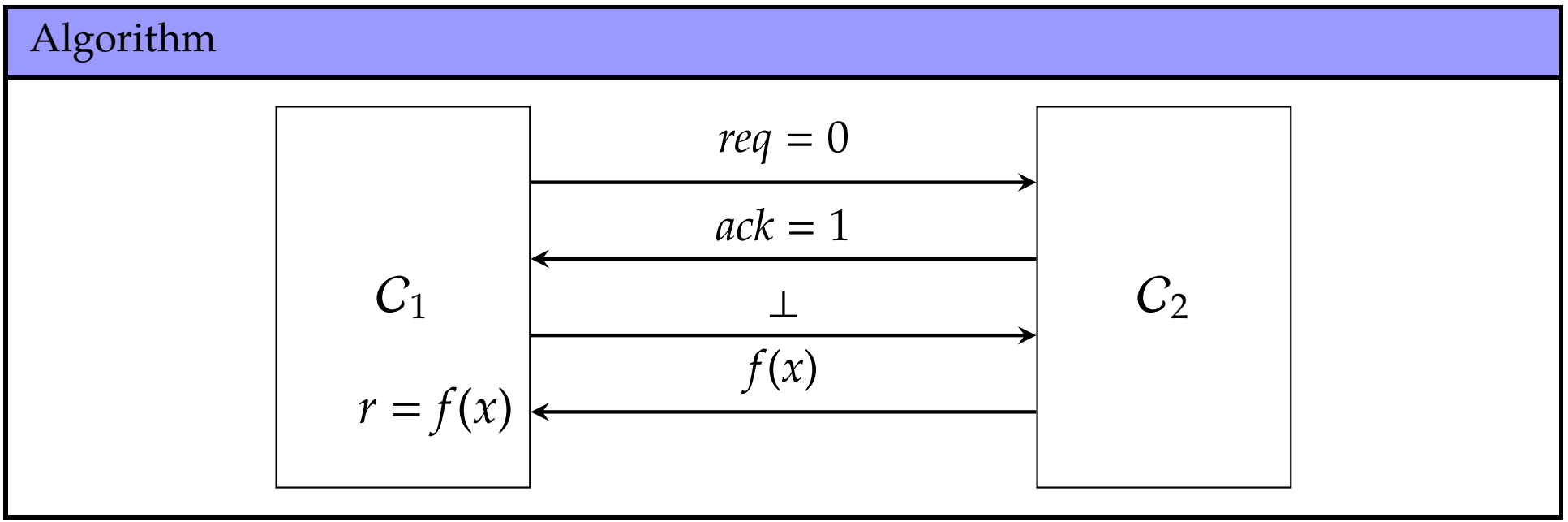


- Design:
  - given a user C_1 of some component C_2, how does
    - C_2 know when to start computation (e.g., when any input x is available), and
    - C_1 know when computation has finished (e.g., when any output $r = f(x)$ is available).
  - we could implement an the interface which
    1. uses a shared clock signal to synchronise events,
    2. uses a **control protocol**, e.g., via additional *req* (or **request**) and *ack* (or **acknowledge**) signals,
    3. ...
- Example:

Algorithm

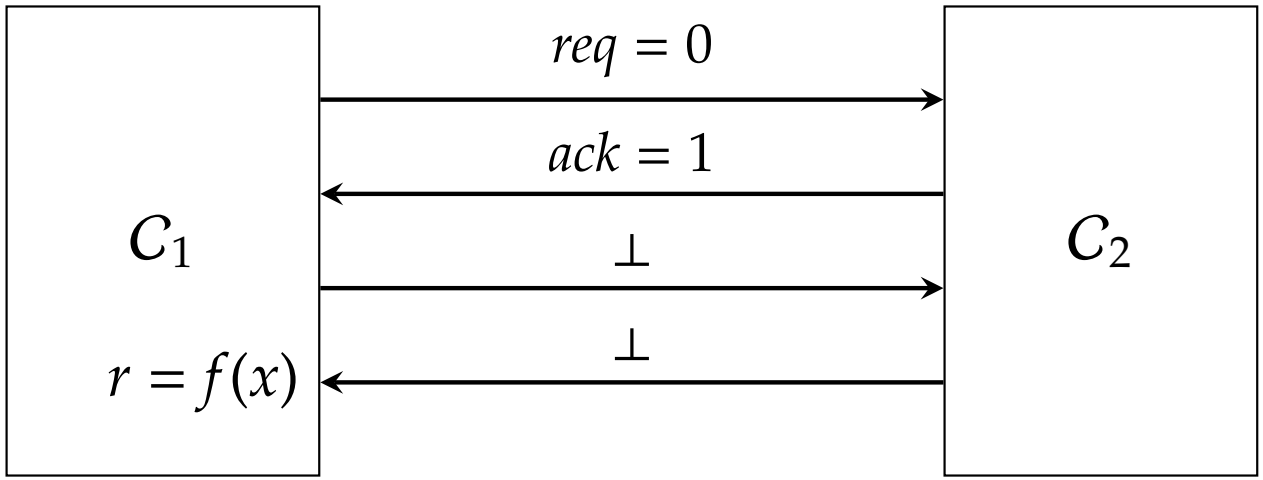


- Design:
  - given a user C_1 of some component C_2, how does
    - C_2 know when to start computation (e.g., when any input x is available), and
    - C_1 know when computation has finished (e.g., when any output $r = f(x)$ is available).
  - we could implement an the interface which
    1. uses a shared clock signal to synchronise events,
    2. uses a **control protocol**, e.g., via additional *req* (or **request**) and *ack* (or **acknowledge**) signals,
    3. ...
- Example:

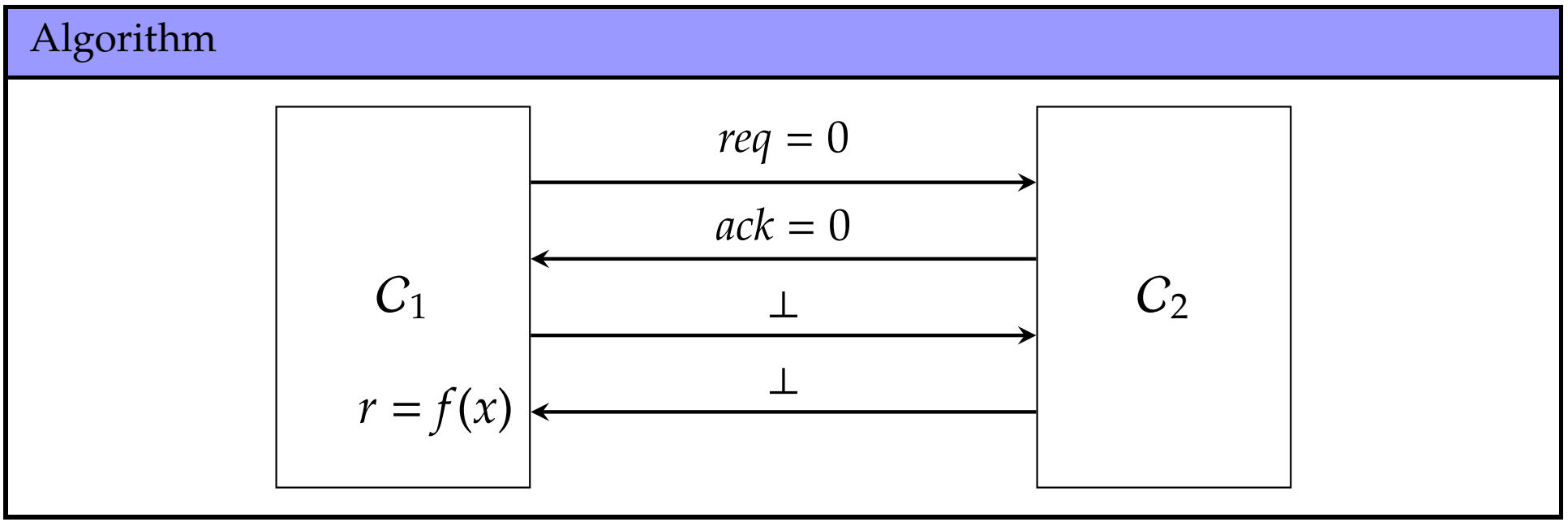


- Design:
  - given a user C_1 of some component C_2, how does
    - C_2 know when to start computation (e.g., when any input x is available), and
    - C_1 know when computation has finished (e.g., when any output $r = f(x)$ is available).
  - we could implement an the interface which
    1. uses a shared clock signal to synchronise events,
    2. uses a **control protocol**, e.g., via additional *req* (or **request**) and *ack* (or **acknowledge**) signals,
    3. ...
- Example:

Algorithm

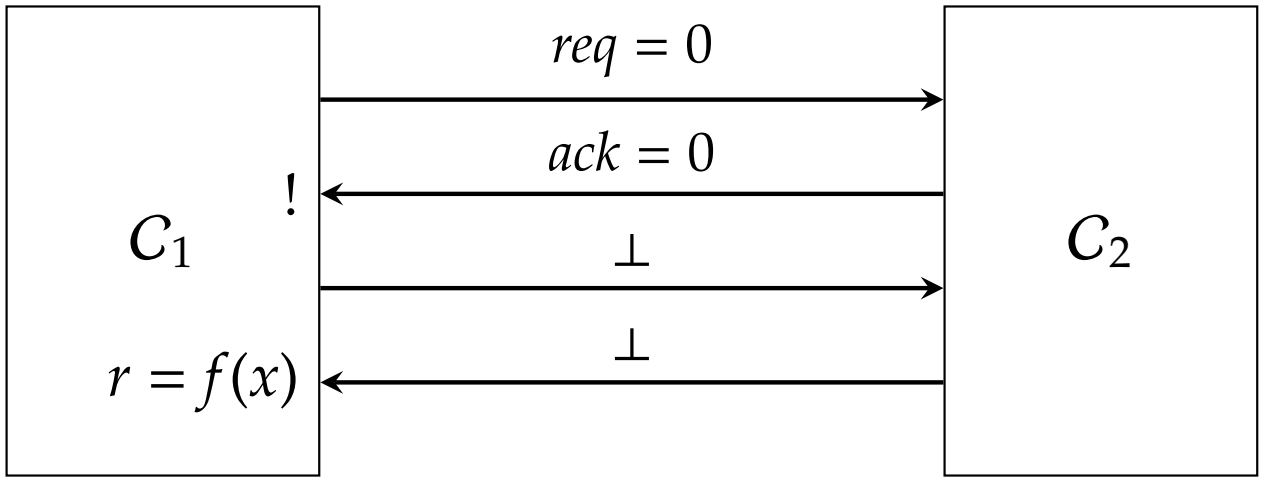


- Design:
  - given a user C_1 of some component C_2, how does
    - C_2 know when to start computation (e.g., when any input x is available), and
    - C_1 know when computation has finished (e.g., when any output $r = f(x)$ is available).
  - we could implement an the interface which
    1. uses a shared clock signal to synchronise events,
    2. uses a **control protocol**, e.g., via additional *req* (or **request**) and *ack* (or **acknowledge**) signals,
    3. ...
- Example:

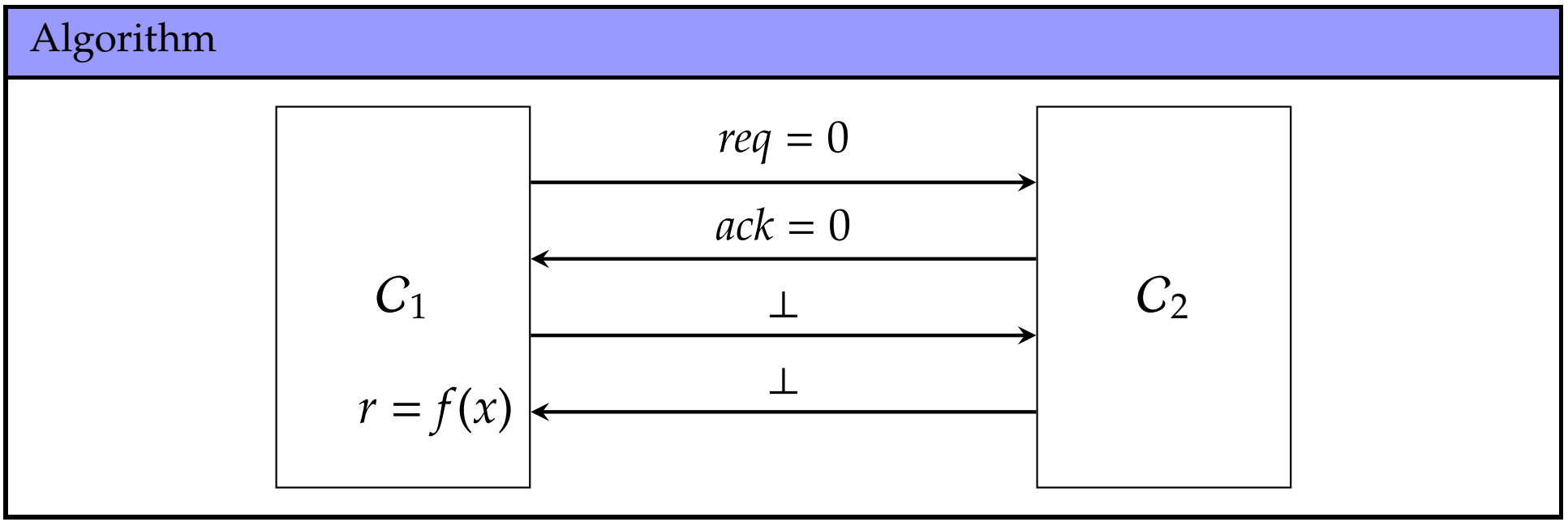


- Design:
  - given a user C_1 of some component C_2, how does
    - C_2 know when to start computation (e.g., when any input x is available), and
    - C_1 know when computation has finished (e.g., when any output $r = f(x)$ is available).
  - we could implement an the interface which
    1. uses a shared clock signal to synchronise events,
    2. uses a **control protocol**, e.g., via additional *req* (or **request**) and *ack* (or **acknowledge**) signals,
    3. ...
- Example:

Algorithm

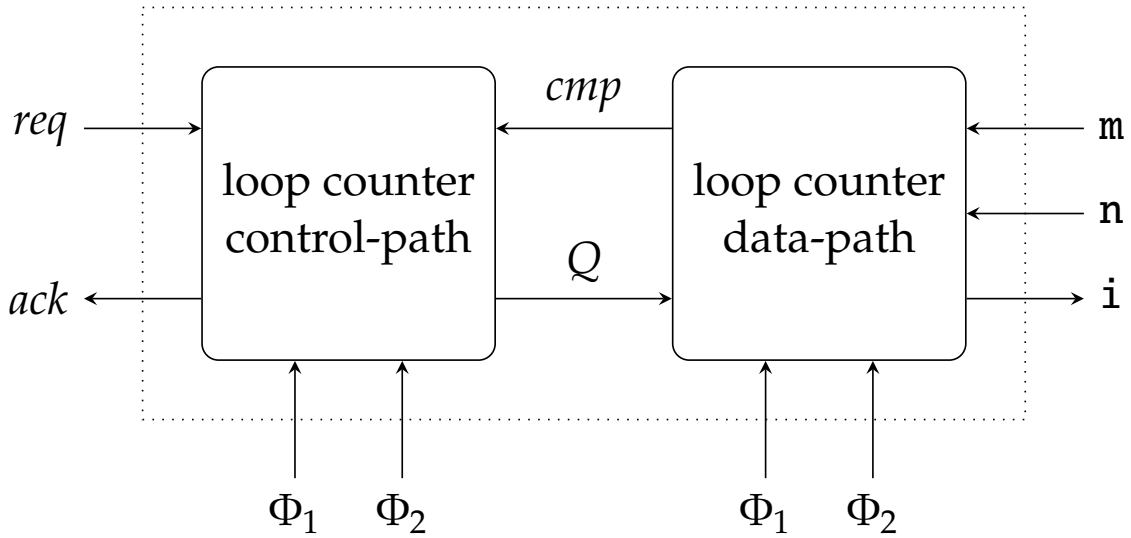


- Design:
  - given a user C_1 of some component C_2, how does
    - C_2 know when to start computation (e.g., when any input x is available), and
    - C_1 know when computation has finished (e.g., when any output $r = f(x)$ is available).
  - we could implement an the interface which
    1. uses a shared clock signal to synchronise events,
    2. uses a **control protocol**, e.g., via additional *req* (or **request**) and *ack* (or **acknowledge**) signals,
    3. ...
- Example:



- Design:

Circuit (latch version)



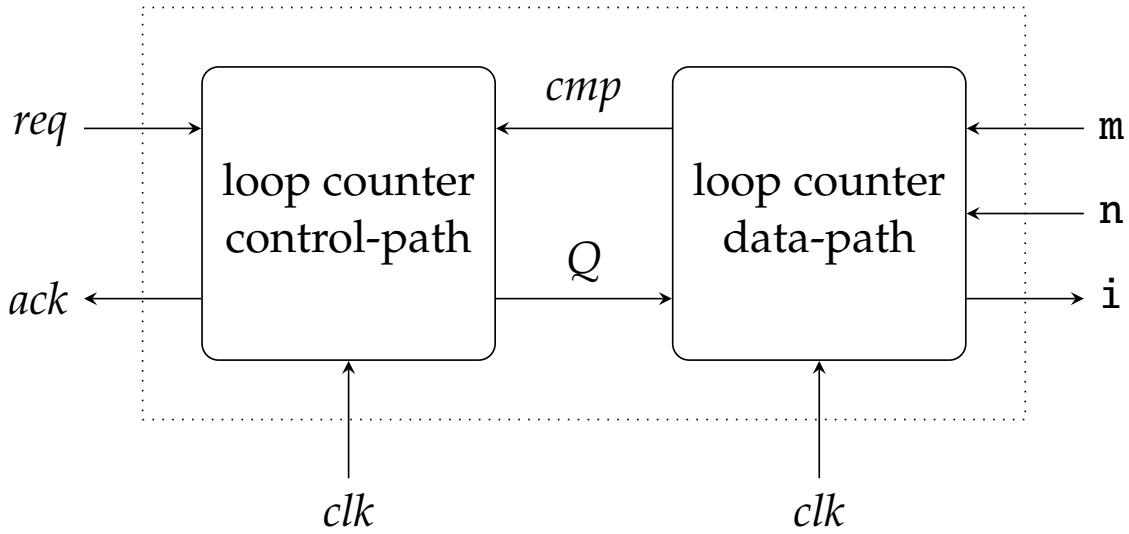
i.e., the design is *itself* the combination of

- a **data-path**, of computational and/or storage components, and
- a **control-path**, that tells components in the data-path what to do and when to do it,

with the latter more overtly realised using an FSM.

- Design:

Circuit (flip-flop version)



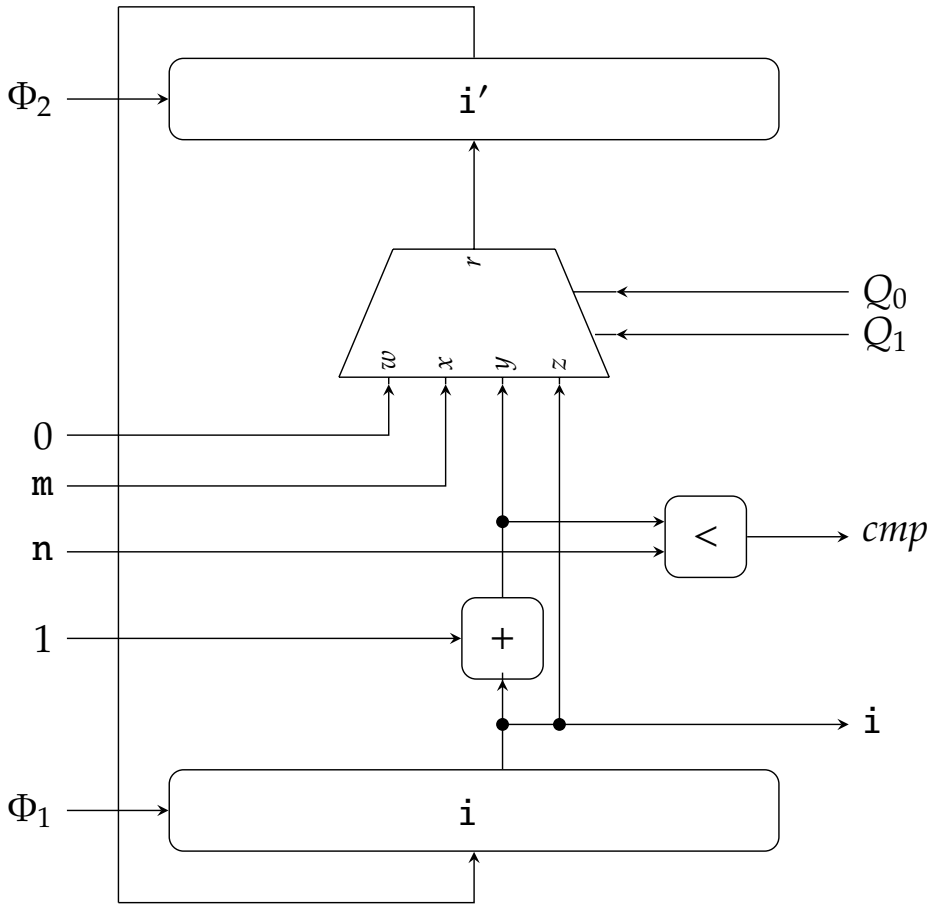
i.e., the design is *itself* the combination of

- a **data-path**, of computational and/or storage components, and
- a **control-path**, that tells components in the data-path what to do and when to do it,

with the latter more overtly realised using an FSM.

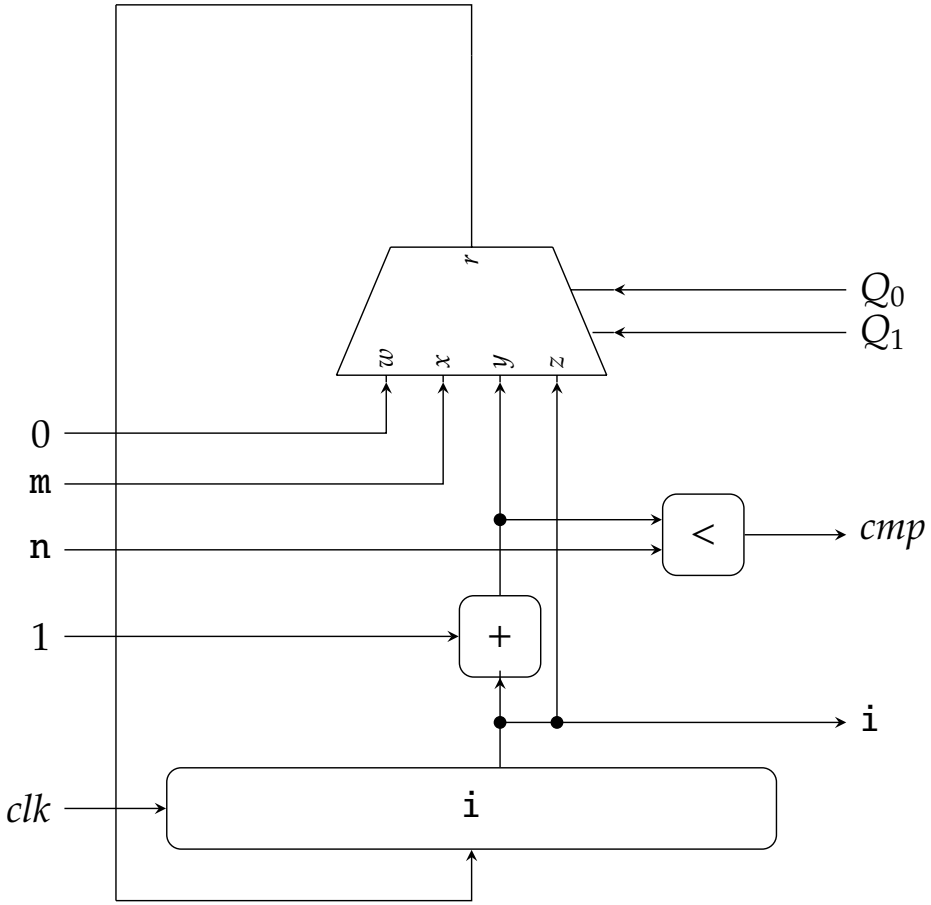
- Solution: the data-path.

Circuit (latch version)



- Solution: the data-path.

Circuit (flip-flop version)

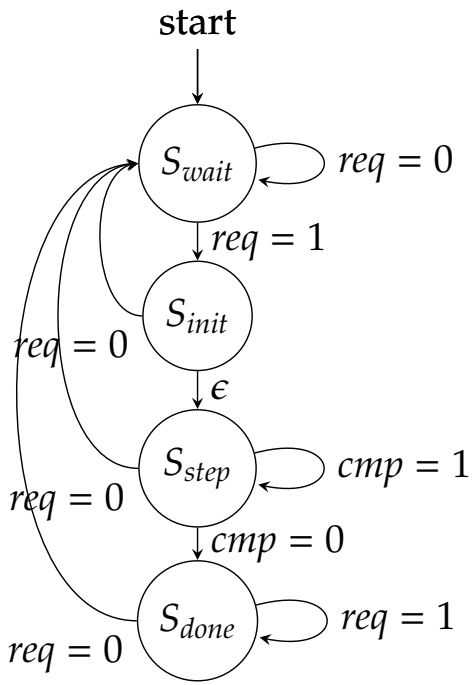


- Solution: the control-path.

Algorithm (tabular)

	Q	δ Q' $cmp = 0$	δ Q' $cmp = 1$	ω ack $cmp = 0$	ω ack $cmp = 1$
$req = 0$	S_{wait}	S_{wait}	S_{wait}	0	0
	S_{init}	S_{wait}	S_{wait}	0	0
	S_{step}	S_{wait}	S_{wait}	0	0
	S_{done}	S_{wait}	S_{wait}	1	1
$req = 1$	S_{wait}	S_{init}	S_{init}	0	0
	S_{init}	S_{step}	S_{step}	0	0
	S_{step}	S_{done}	S_{step}	0	0
	S_{done}	S_{done}	S_{done}	1	1

Algorithm (diagram)



i.e.,

- in S_{wait} it waits for $req = 1$,
- in S_{init} it uses any input to initialise itself (e.g., setting the initial loop counter value),
- in S_{step} it performs an iteration of the loop, and
- in S_{done} it waits for $req = 0$ while setting $ack = 1$.

- Solution: the control-path.
  - there are 4 abstract labels

$$\begin{array}{lcl} S_{wait} & \mapsto & 0 \\ S_{init} & \mapsto & 1 \\ S_{step} & \mapsto & 2 \\ S_{done} & \mapsto & 3 \end{array}$$

  we can represent using 4 concrete values, e.g.,

$$\begin{array}{lcll} S_{wait} & \mapsto & \langle 0, 0 \rangle & \equiv & 00_{(2)} \\ S_{init} & \mapsto & \langle 1, 0 \rangle & \equiv & 01_{(2)} \\ S_{step} & \mapsto & \langle 0, 1 \rangle & \equiv & 10_{(2)} \\ S_{done} & \mapsto & \langle 1, 1 \rangle & \equiv & 11_{(2)} \end{array}$$

  - since $2^2 = 4$, we can capture each of

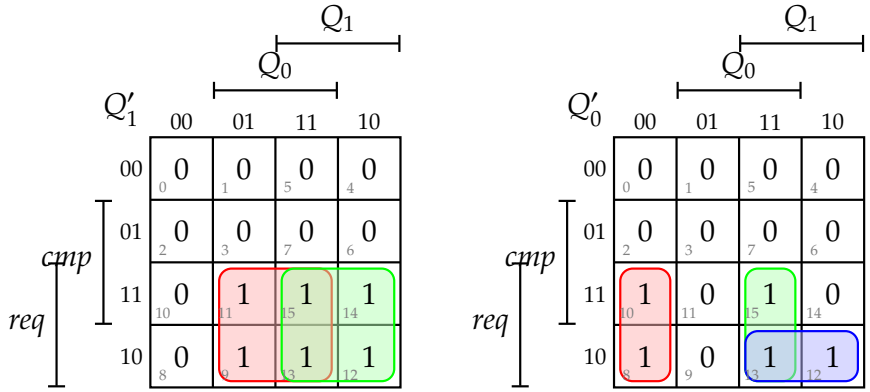
1. $Q = \langle Q_0, Q_1 \rangle \equiv$ the current state
2. $Q' = \langle Q'_0, Q'_1 \rangle \equiv$ the next state

  in a 2-bit register (i.e., via 2 latches or flip-flops).

- Solution: the control-path.
  - rewriting the abstract labels yields the following concrete truth table

				δ		ω
req	*cmp*	Q_1	Q_0	Q'_1	Q'_0	*ack*
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	0	1	1	0	0	1
0	1	0	0	0	0	0
0	1	0	1	0	0	0
0	1	1	0	0	0	0
0	1	1	1	0	0	1
1	0	0	0	0	1	0
1	0	0	1	1	0	0
1	0	1	0	1	1	0
1	0	1	1	1	1	1
1	1	0	0	0	1	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	1	1	1

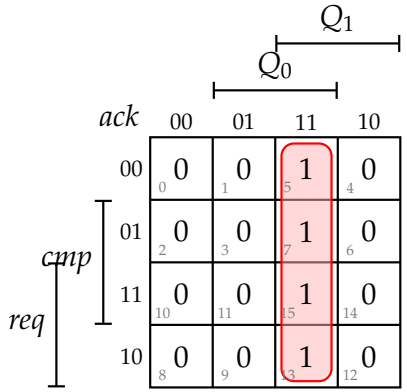
- Solution: the control-path.
  - the truth table can be translated into



  - doing so yields the following Boolean expressions for δ:

$$
\begin{aligned}
Q_1' = (\ & req & & & & & \wedge & Q_0 &) \vee \\
(\ & req & & & \wedge & Q_1 & & &) \\
\\
Q_0' = (\ & req & & & \wedge & \neg Q_1 & \wedge & \neg Q_0 &) \vee \\
(\ & req & & & \wedge & Q_1 & \wedge & Q_0 &) \vee \\
(\ & req & \wedge & \neg cmp & \wedge & Q_1 & & &)
\end{aligned}
$$

- Solution: the control-path.
  - the truth table can be translated into



  - doing so yields the following Boolean expressions for ω:

$$ack = Q_1 \wedge Q_0$$

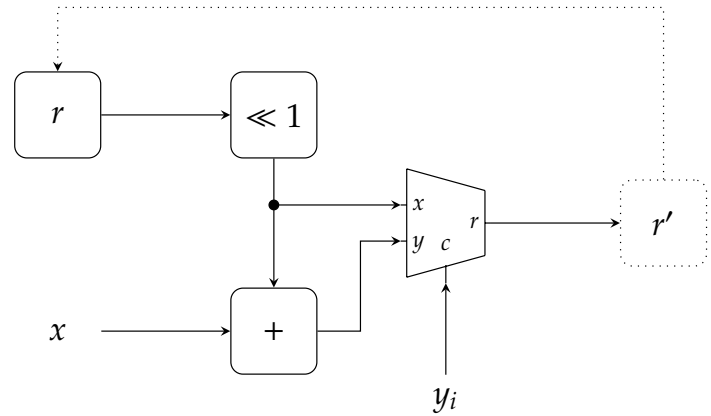
- Use-case:
  - we want(ed) to implement a bit-serial multiplier, i.e.,

Algorithm

Input: Two unsigned, n-bit, base-2 integers x and y
Output: An unsigned, $2n$-bit, base-2 integer $r = y \cdot x$

```
1 r ← 0
2 for i = n − 1 downto 0 step −1 do
3 |   r ← 2 · r
4 |   if y_i = 1 then
5 |   |   r ← r + x
6 |   end
7 end
8 return r
```

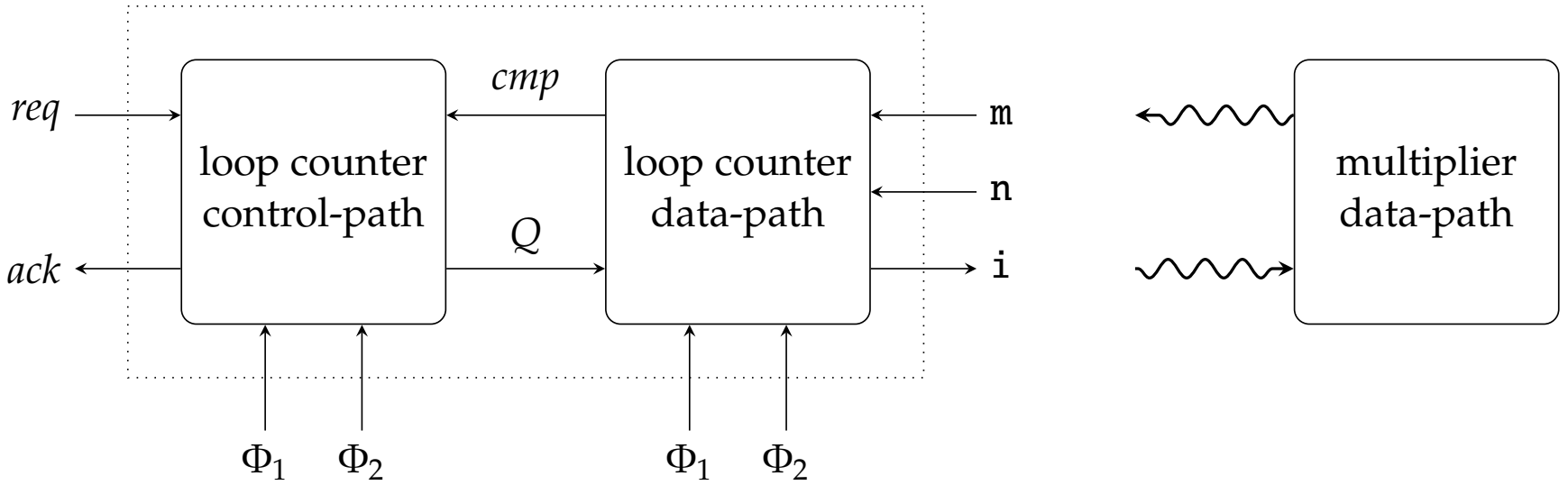
Circuit



- we *did* have the data-path,
- we *didn't* have the control-path.

- Use-case:
  - we now have the loop counter implemented, i.e.

Circuit (latch version)

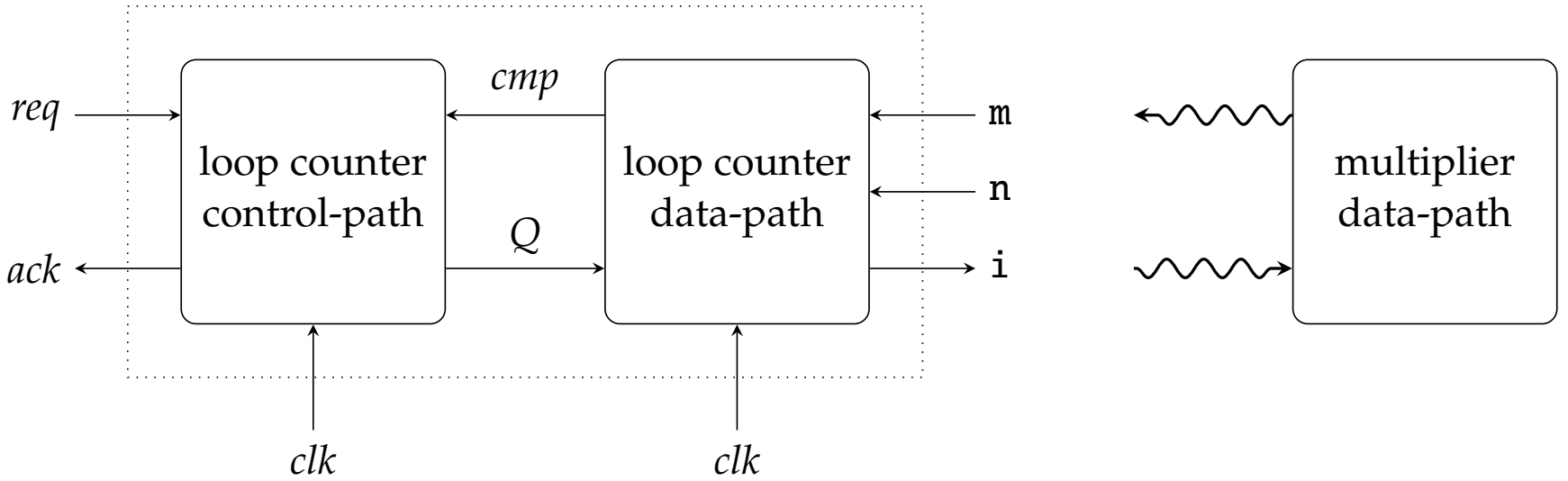


- the remaining challenge is integration, e.g., specifying
  - any additional data-path components required, and
  - how loop counter (the control-path) controls them

  so we end up with a bit-serial multiplier.

- Use-case:
  - we now have the loop counter implemented, i.e.

Circuit (flip-flop version)



- the remaining challenge is integration, e.g., specifying
  - any additional data-path components required, and
  - how loop counter (the control-path) controls them

  so we end up with a bit-serial multiplier.

Conclusions

- Take away points:
  1. FSMs are abstract computational models, but we can used them to solve concrete problems, e.g.,
     - recognisers,
     - controllers,
     - ...
     - *specifications*: like an algorithm, but more easily able to cater for asynchronous events.
  2. The "killer application" of FSMs for *us* is as a general-purpose way to realise controlled step-by-step forms of computation.
  3. Clearly more complex problem $\Rightarrow$ more complex solution, *but*
     - same framework and process (both conceptual, *and* practical),
     - same components (e.g., interface, implementation; data-path, control-path),

     so difference is (arguably) creativity re. design.

Additional Reading

- *Wikipedia: Finite State Machine (FSM)*. URL: https://en.wikipedia.org/wiki/Finite-state_machine.
- D. Page. "Chapter 2: Basics of digital logic". In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer, 2009.
- M. Sipster. "Chapter 1: Regular languages". In: *Introduction to the Theory of Computation*. 2nd ed. Thomson Course Technology, 2006.

References

[1] *Wikipedia: Finite State Machine (FSM)*. URL: https://en.wikipedia.org/wiki/Finite-state_machine (see p. 86).

[2] D. Page. "Chapter 2: Basics of digital logic". In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer, 2009 (see p. 86).

[3] M. Sipster. "Chapter 1: Regular languages". In: *Introduction to the Theory of Computation*. 2nd ed. Thomson Course Technology, 2006 (see p. 86).

[4] M. Sipster. *Introduction to the Theory of Computation*. 2nd ed. Thomson Course Technology, 2006 (see pp. 1, 14–25).