

# Computer Architecture

Daniel Page

Department of Computer Science,  
University Of Bristol,  
Merchant Venturers Building,  
Woodland Road,  
Bristol, BS8 1UB. UK.  
([csdsp@bristol.ac.uk](mailto:csdsp@bristol.ac.uk))

September 5, 2025

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
  - ▶ the associated notes page may be pre-populated with extra, written explanation of material covered in lecture(s), plus
  - ▶ anything with a “grey’ed out” header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

Notes:

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

**finite automata**  $\equiv$  **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.

Notes:

## Part 1: in theory (1)

### Definition

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

### Definition

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

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

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  $\epsilon$ . It can be useful, and is common to write elements 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**  $\Lambda$  is a set of strings.

Notes:

## Definition

A (formal) **grammar** is a tuple

$$G = (N, n, T, P)$$

including

1.  $N$ , a finite set of **non-terminal symbols** that includes a **start symbol**  $n \in N$ ,
2.  $T$ , a finite set of **terminal symbols** (which is disjoint from  $N$ ), and
3.  $P$ , a finite set of **production rules** each of the form

$$P_i : (N \cup T)^* \rightarrow (N \cup T)^*.$$

Notes:

## Part 1: in theory (3)

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

Notes:

## 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.

Notes:

## 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  $\Sigma$ , one symbol at a time; each symbol induces a change in state.

Notes:

## 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  $\Sigma$ , 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

Notes:

## 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  $\Sigma$ , 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  $\Lambda$  of all possible input strings  $C$  could accept, we say
$$C \text{ accepts (or recognises) } \Lambda \equiv \Lambda \text{ is the language of } C$$
and use  $\Lambda$  to classify  $C$  ...

Notes:

Part 1: in theory (4)

Definition					
<div>less powerful ← → more powerful</div>					
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 ( $X \rightarrow x$ or $X \rightarrow xY$ )	context free ( $X \rightarrow \gamma$ )	context sensitive ( $\alpha X \beta \rightarrow \alpha \gamma \beta$ )	unrestricted ( $\alpha \rightarrow \beta$ )
Chomsky-Schützenberger hierarchy		type-3	type-2	type-1	type-0

Notes:

- In the description of grammars, the idea is (read from left-to-right) there are progressively less and less restrictions:  $X$  and  $Y$  both represent non-terminal symbols,  $x$  represents a terminal symbol, and  $\alpha$ ,  $\beta$  and  $\gamma$  are strings of either terminal or non-terminal symbols, all with respect to some alphabet and grammar. So, the far right-hand case has rules that are totally unrestricted, for example, whereas the far left-hand case has rules that must be of a particular form.

Part 1: in theory (5)

Definition	
A (deterministic) <b>Finite State Machine (FSM)</b> is a tuple	
$C = (S, s, A, \Sigma, \Gamma, \delta, \omega)$	
including	
1. $S$ , a finite set of <b>states</b> that includes a <b>start state</b> $s \in S$ ,	
2. $A \subseteq S$ , a finite set of <b>accepting states</b> ,	
3. an <b>input alphabet</b> $\Sigma$ and an <b>output alphabet</b> $\Gamma$ ,	
4. a <b>transition function</b>	
	$\delta : S \times \Sigma \rightarrow S$
and	
5. an <b>output function</b>	
	$\omega : S \rightarrow \Gamma$
in the case of a <b>Moore FSM</b> , or	
	$\omega : S \times \Sigma \rightarrow \Gamma$
in the case of a <b>Mealy FSM</b> ,	
noting an <b>empty</b> input denoted $\epsilon$ allows a transition that can <i>always</i> occur.	

Notes:

- Even if the state space is *huge*, it is still finite: to cope,  $\delta$  can a **partial function** meaning it needn't be defined for all inputs.

Part 1: in theory (6)

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

Notes:

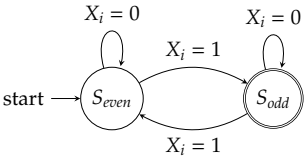
Part 1: in theory (6)

- **Solution:**

Algorithm (tabular)

$Q$	$\delta$	
	$X_i = 0$	$X_i = 1$
$S_{even}$	$S_{even}$	$S_{odd}$
$S_{odd}$	$S_{odd}$	$S_{even}$

Algorithm (diagram)



Notes:

where, e.g.,

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

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

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

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

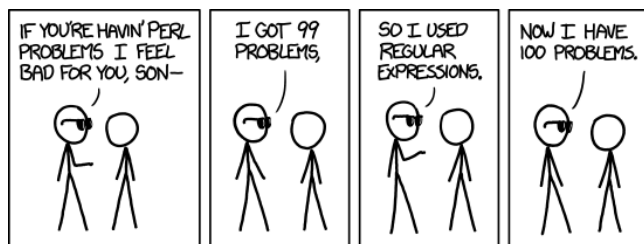
so the input is rejected (i.e., has an even number of 1 elements).

## Part 2.1: in practice, application (1)

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

### ► Context:

- –ve perspective:



- +ve perspective: we could say that

arithmetic expression	evaluate $\leadsto$	number
regular expression	evaluate $\leadsto$	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>

## Part 2.1: in practice, application (2)

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

### 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 concatenation 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

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

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

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

Notes:

Notes:



## Part 2.1: in practice, application (3)

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

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

$$0^*10^* \equiv \left\{ s \mid \begin{array}{l} s \text{ is a string containing} \\ \text{a single } 1 \end{array} \right\}$$

Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will differ versus use of `grep` itself. The technical reason for this is that we have ignored (or disallowed) backtracking; if the FSM accepts a '1' followed by something other than '0' then it fails, but obviously this ignores the possibility that there might be *another* '1' later which is followed by '0'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one '1' character, and b) the first '1' character is followed by at least one '0' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point. In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

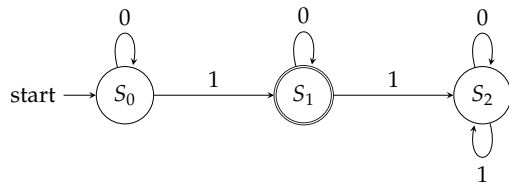
## Part 2.1: in practice, application (3)

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

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

$$0^*10^* \equiv \left\{ s \mid \begin{array}{l} s \text{ is a string containing} \\ \text{a single } 1 \end{array} \right\}$$

which can be realised using



Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will differ versus use of `grep` itself. The technical reason for this is that we have ignored (or disallowed) backtracking; if the FSM accepts a '1' followed by something other than '0' then it fails, but obviously this ignores the possibility that there might be *another* '1' later which is followed by '0'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one '1' character, and b) the first '1' character is followed by at least one '0' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point. In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

## Part 2.1: in practice, application (3)

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

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

$$\Sigma^*001\Sigma^* \equiv \left\{ s \mid \begin{array}{l} s \text{ is a string containing} \\ 001 \text{ as a sub-string} \end{array} \right\}$$

which can be realised using

Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will differ versus use of `grep` itself. The technical reason for this is that we have ignored (or disallowed) backtracking; if the FSM accepts a '1' followed by something other than '0' then it fails, but obviously this ignores the possibility that there might be *another* '1' later which is followed by '0'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one '1' character, and b) the first '1' character is followed by at least one '0' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point. In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

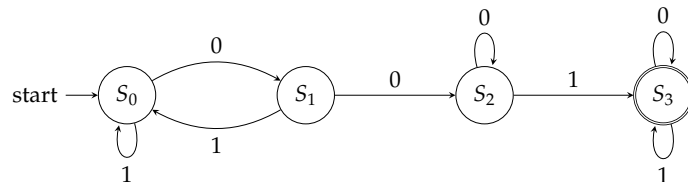
## Part 2.1: in practice, application (3)

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

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

$$\Sigma^*001\Sigma^* \equiv \left\{ s \mid \begin{array}{l} s \text{ is a string containing} \\ 001 \text{ as a sub-string} \end{array} \right\}$$

which can be realised using



Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will differ versus use of `grep` itself. The technical reason for this is that we have ignored (or disallowed) backtracking; if the FSM accepts a '1' followed by something other than '0' then it fails, but obviously this ignores the possibility that there might be *another* '1' later which is followed by '0'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one '1' character, and b) the first '1' character is followed by at least one '0' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point. In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

## Part 2.1: in practice, application (3)

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

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

$$(\Sigma\Sigma)^* \equiv \left\{ s \mid \begin{array}{l} s \text{ is a string} \\ \text{of even length} \end{array} \right\}$$

Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will differ versus use of `grep` itself. The technical reason for this is that we have ignored (or disallowed) backtracking; if the FSM accepts a 't' followed by something other than 'o' then it fails, but obviously this ignores the possibility that there might be *another* 't' later which is followed by 'o'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one 't' character, and b) the first 't' character is followed by at least one 'o' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point. In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

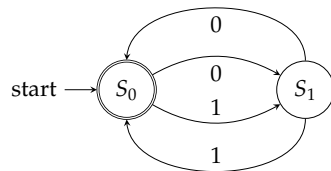
## Part 2.1: in practice, application (3)

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

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

$$(\Sigma\Sigma)^* \equiv \left\{ s \mid \begin{array}{l} s \text{ is a string} \\ \text{of even length} \end{array} \right\}$$

which can be realised using



Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will differ versus use of `grep` itself. The technical reason for this is that we have ignored (or disallowed) backtracking; if the FSM accepts a 't' followed by something other than 'o' then it fails, but obviously this ignores the possibility that there might be *another* 't' later which is followed by 'o'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one 't' character, and b) the first 't' character is followed by at least one 'o' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point. In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

## Part 2.1: in practice, application (3)

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

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

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

Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will differ versus use of `grep` itself. The technical reason for this is that we have ignored (or disallowed) backtracking; if the FSM accepts a '1' followed by something other than '0' then it fails, but obviously this ignores the possibility that there might be *another* '1' later which is followed by '0'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one '1' character, and b) the first '1' character is followed by at least one '0' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point. In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

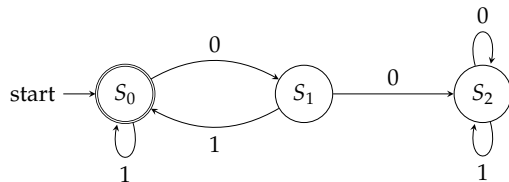
## Part 2.1: in practice, application (3)

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

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

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

which can be realised using



Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will differ versus use of `grep` itself. The technical reason for this is that we have ignored (or disallowed) backtracking; if the FSM accepts a '1' followed by something other than '0' then it fails, but obviously this ignores the possibility that there might be *another* '1' later which is followed by '0'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one '1' character, and b) the first '1' character is followed by at least one '0' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point. In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

## Part 2.1: in practice, application (3)

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

► **Example** [4, Example 1.53]: if  $\Sigma = \{ 'a', 'b', \dots, 'z' \}$ , then

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

Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will differ versus use of `grep` itself. The technical reason for this is that we have ignored (or disallowed) backtracking; if the FSM accepts a 't' followed by something other than 'o' then it fails, but obviously this ignores the possibility that there might be *another* 't' later which is followed by 'o'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one 't' character, and b) the first 't' character is followed by at least one 'o' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point. In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

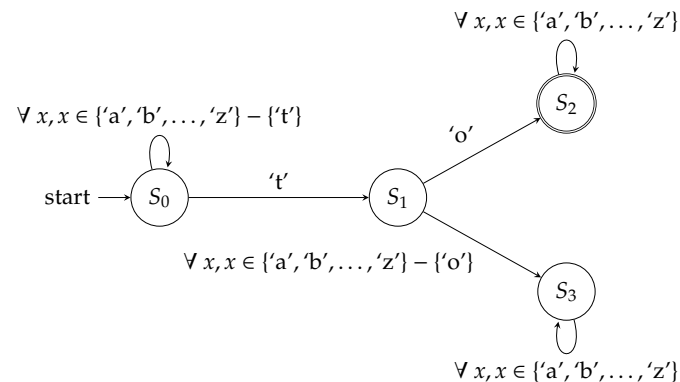
## Part 2.1: in practice, application (3)

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

► **Example** [4, Example 1.53]: if  $\Sigma = \{ 'a', 'b', \dots, 'z' \}$ , then

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

which can be realised using



Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will differ versus use of `grep` itself. The technical reason for this is that we have ignored (or disallowed) backtracking; if the FSM accepts a 't' followed by something other than 'o' then it fails, but obviously this ignores the possibility that there might be *another* 't' later which is followed by 'o'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one 't' character, and b) the first 't' character is followed by at least one 'o' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point. In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

## Part 2.1: in practice, application (3)

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

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

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

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, Xi)
6   end
7   if Q ∈ A then
8     | print line X to stdout
10  end
11 end
```

Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will differ versus use of `grep` itself. The technical reason for this is that we have ignored (or disallowed) backtracking; if the FSM accepts a 't' followed by something other than 'o' then it fails, but obviously this ignores the possibility that there might be *another* 't' later which is followed by 'o'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one 't' character, and b) the first 't' character is followed by at least one 'o' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point. In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

## Part 2.1: in practice, application (3)

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

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

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

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 }
```

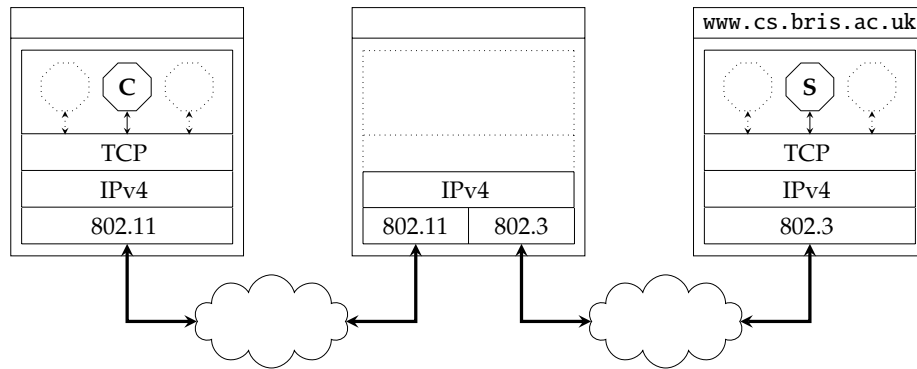
Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will differ versus use of `grep` itself. The technical reason for this is that we have ignored (or disallowed) backtracking; if the FSM accepts a 't' followed by something other than 'o' then it fails, but obviously this ignores the possibility that there might be *another* 't' later which is followed by 'o'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one 't' character, and b) the first 't' character is followed by at least one 'o' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point. In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

## Part 2.1: in practice, application (4)

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

### ► Context:

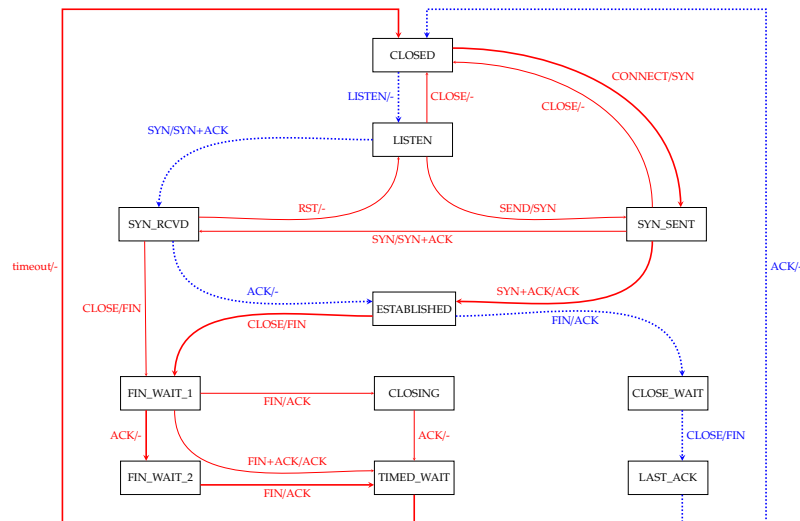


Notes:

## Part 2.1: in practice, application (5)

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

### ► Example:



Notes:

## Part 2.1: in practice, application (6)

Example #3: typical video game “loop”  $\leadsto$  FSMs as systems

### ► Context:



#### Algorithm

```
1 reset the game state
2 while  $\neg$  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
```

Notes:

## Part 2.1: in practice, application (6)

Example #3: typical video game “loop”  $\leadsto$  FSMs as systems

### ► Context:



#### Algorithm

```
1  $Q \leftarrow s$ 
2 while  $Q \notin A$  do
3    $X_i \leftarrow$  control pad
4    $Q \leftarrow \delta(Q, X_i)$ 
5    $\{\text{graphics, sound}\} \leftarrow \omega(Q)$ 
6 end
```

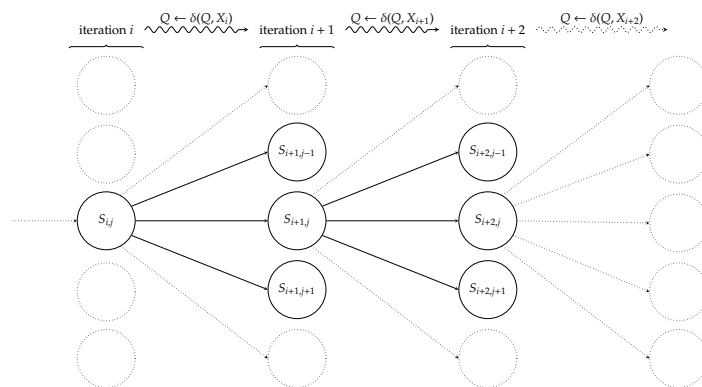
Notes:



► Example:

iterations of game loop  $\leadsto$  game tree  
 $\approx$  state space

i.e.,



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

Notes:

Part 2.2: in practice, implementation (1)  
 Design framework

► 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**  $\Sigma$  and an **output alphabet**  $\Gamma$ ,
4. a **transition function**

$$\delta : S \times \Sigma \rightarrow S$$

and

5. an **output function**

$$\omega : S \rightarrow \Gamma$$

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

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

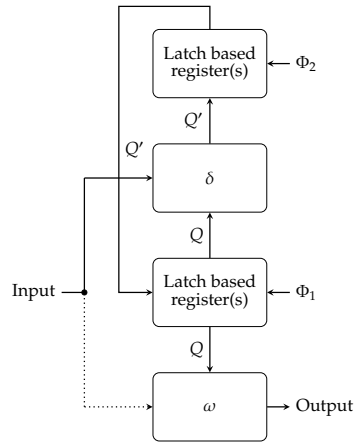
in the case of a **Mealy FSM**,

noting an **empty** input denoted  $\epsilon$  allows a transition that can *always* occur.

Notes:

► Concept:

Algorithm (latch version)



► Note that

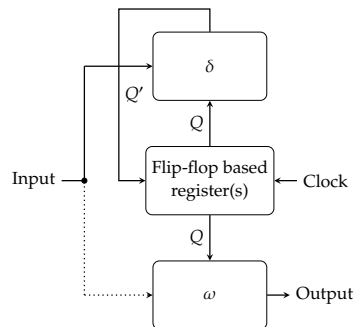
1. the state is retained in a register (i.e., a group of latches, resp. flip-flops),
2.  $\delta$  and  $\omega$  are simply combinatorial logic,
3. within the current clock cycle
  - $\omega$  computes the output from the current state and input, and
  - $\delta$  computes the next state from the current 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*!

Notes:

► Concept:

Algorithm (flip-flop version)



► Note that

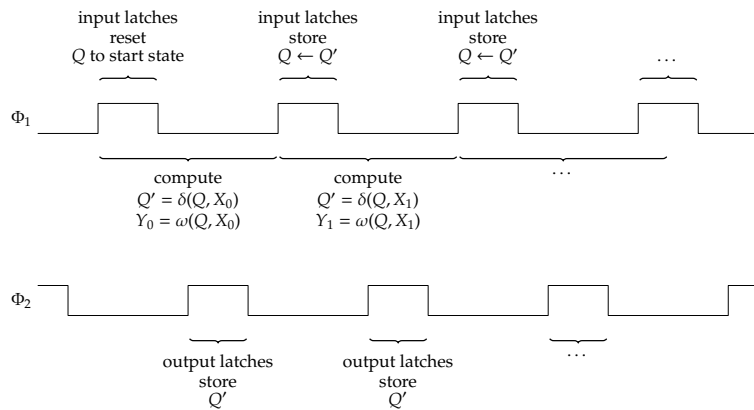
1. the state is retained in a register (i.e., a group of latches, resp. flip-flops),
2.  $\delta$  and  $\omega$  are simply combinatorial logic,
3. within the current clock cycle
  - $\omega$  computes the output from the current state and input, and
  - $\delta$  computes the next state from the current 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*!

Notes:

► Concept:

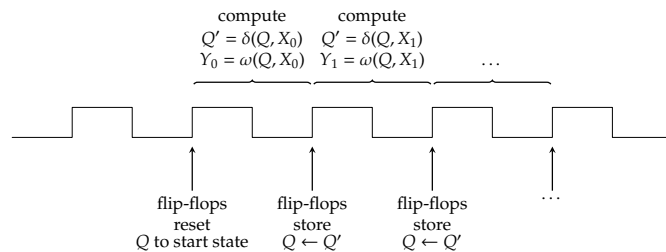
Example (latch version: input  $X = \langle X_0, X_1, \dots \rangle$ , output  $Y = \langle Y_0, Y_1, \dots \rangle$ )



Notes:

► Concept:

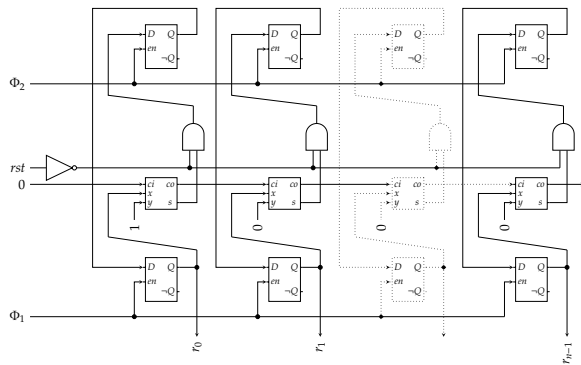
Example (flip-flop version: input  $X = \langle X_0, X_1, \dots \rangle$ , output  $Y = \langle Y_0, Y_1, \dots \rangle$ )



Notes:

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

### Circuit (latch version)

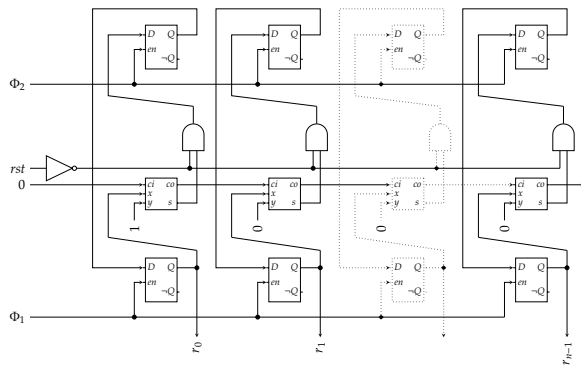


it *now* becomes clear that

- $2^n$  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 familiar, because from

### Circuit (latch version)



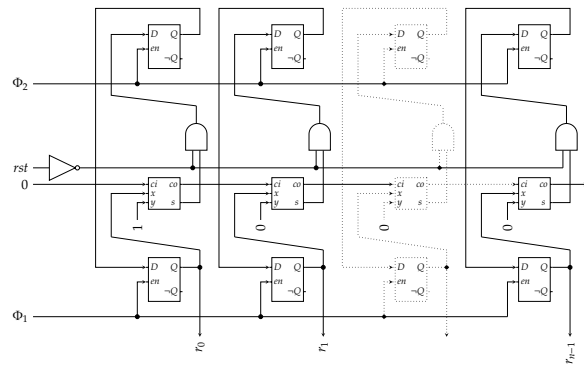
it *now* becomes clear that

- the  $\delta$  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 familiar, because from

### Circuit (latch version)



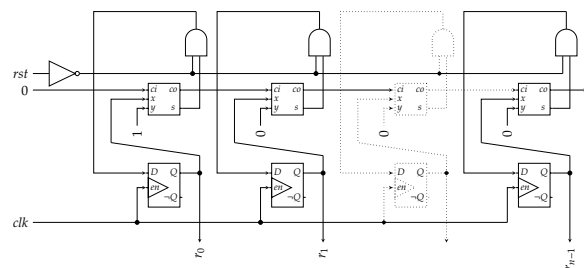
it *now* becomes clear that

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

Notes:

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

### Circuit (flip-flop version)



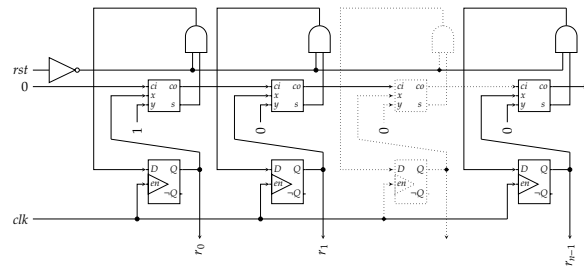
it *now* becomes clear that

- $2^n$  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  $A = \emptyset$ ),

Notes:

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

### Circuit (flip-flop version)



it *now* becomes clear that

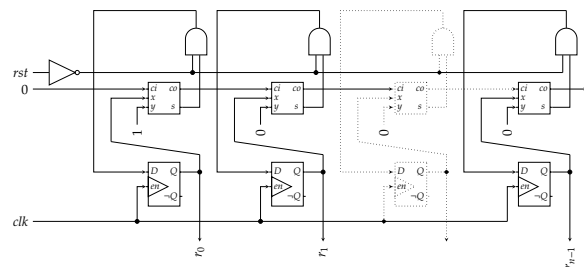
- the  $\delta$  function is

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

Notes:

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

### Circuit (flip-flop version)



it *now* becomes clear that

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

Notes:

- **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  $\delta$  and  $\omega$  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.

Notes:

- **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.,

$S_0$	$\mapsto$	$\langle 0, 0, 0 \rangle$
$S_1$	$\mapsto$	$\langle 1, 0, 0 \rangle$
$S_2$	$\mapsto$	$\langle 0, 1, 0 \rangle$
$S_3$	$\mapsto$	$\langle 1, 1, 0 \rangle$
$S_4$	$\mapsto$	$\langle 0, 0, 1 \rangle$
$S_5$	$\mapsto$	$\langle 1, 0, 1 \rangle$

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.,

$S_0$	$\mapsto$	$\langle 1, 0, 0, 0, 0, 0 \rangle$
$S_1$	$\mapsto$	$\langle 0, 1, 0, 0, 0, 0 \rangle$
$S_2$	$\mapsto$	$\langle 0, 0, 1, 0, 0, 0 \rangle$
$S_3$	$\mapsto$	$\langle 0, 0, 0, 1, 0, 0 \rangle$
$S_4$	$\mapsto$	$\langle 0, 0, 0, 0, 1, 0 \rangle$
$S_5$	$\mapsto$	$\langle 0, 0, 0, 0, 0, 1 \rangle$

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.

Notes:

## Part 2.2: in practice, implementation (10)

Example #1: a modulo 6 ascending counter

- **Problem:** design an FSM that acts as a cyclic counter modulo  $n = 6$  (versus  $2^n$ ).

Notes:

- If  $n = 6$  for example, then we want a component whose output  $r$  steps through values

$0, 1, 2, 3, 4, 5, 0, 1, \dots,$

with the modular reduction representing control behaviour (versus the uncontrolled counter that was cyclic by default).

## Part 2.2: in practice, implementation (10)

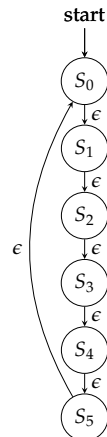
Example #1: a modulo 6 ascending counter

- **Solution:**

### Algorithm (tabular)

	$\delta$	$\omega$
$Q$	$Q'$	$r$
$S_0$	$S_1$	0
$S_1$	$S_2$	1
$S_2$	$S_3$	2
$S_3$	$S_4$	3
$S_4$	$S_5$	4
$S_5$	$S_0$	5

### Algorithm (diagram)



Notes:

- If  $n = 6$  for example, then we want a component whose output  $r$  steps through values

$0, 1, 2, 3, 4, 5, 0, 1, \dots,$

with the modular reduction representing control behaviour (versus the uncontrolled counter that was cyclic by default).



## Part 2.2: in practice, implementation (10)

Example #1: a modulo 6 ascending counter

### ► Solution:

- there are 6 abstract labels

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

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

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

- 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).

Notes:

- If  $n = 6$  for example, then we want a component whose output  $r$  steps through values

$$0, 1, 2, 3, 4, 5, 0, 1, \dots,$$

with the modular reduction representing control behaviour (versus the uncontrolled counter that was cyclic by default).

## Part 2.2: in practice, implementation (10)

Example #1: a modulo 6 ascending counter

### ► Solution:

- rewriting the abstract labels yields the following concrete truth table

			$\delta$			$\omega$		
$Q_2$	$Q_1$	$Q_0$	$Q'_2$	$Q'_1$	$Q'_0$	$r_2$	$r_1$	$r_0$
0	0	0	0	0	1	0	0	0
0	0	1	0	1	0	0	0	1
0	1	0	0	1	1	0	1	0
0	1	1	1	0	0	0	1	1
1	0	0	1	0	1	1	0	0
1	0	1	0	0	0	1	0	1
1	1	0	?	?	?	?	?	?
1	1	1	?	?	?	?	?	?

- note that our state assignment means  $r = Q$ , such that

$$r = \omega(Q) = Q$$

is basically just the identity function (so we'll ignore it).

Notes:

- If  $n = 6$  for example, then we want a component whose output  $r$  steps through values

$$0, 1, 2, 3, 4, 5, 0, 1, \dots,$$

with the modular reduction representing control behaviour (versus the uncontrolled counter that was cyclic by default).

## Part 2.2: in practice, implementation (10)

Example #1: a modulo 6 ascending counter

### ► Solution:

- the truth table can be translated into

		$Q_0$		$Q_1$	
$Q_2$		00	01	11	10
0		0	0	1	0
1		1	0	?	?

		$Q_0$		$Q_1$	
$Q_1$		00	01	11	10
0		0	1	0	1
1		0	0	?	?

		$Q_0$		$Q_1$	
$Q_0$		00	01	11	10
0		1	0	0	1
1		1	0	?	?

- doing so yields the following Boolean expressions for  $\delta$ :

$$Q_2' = (Q_2 \wedge Q_1 \wedge \neg Q_0) \vee$$

$$Q_1' = (\neg Q_2 \wedge \neg Q_1 \wedge Q_0) \vee$$

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

Notes:

- If  $n = 6$  for example, then we want a component whose output  $r$  steps through values

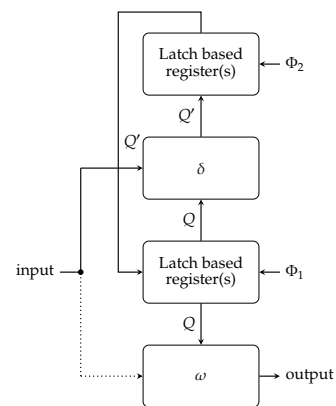
$0, 1, 2, 3, 4, 5, 0, 1, \dots$ ,

with the modular reduction representing control behaviour (versus the uncontrolled counter that was cyclic by default).

## Part 2.2: in practice, implementation (10)

Example #1: a modulo 6 ascending counter

### ► Solution:



Notes:

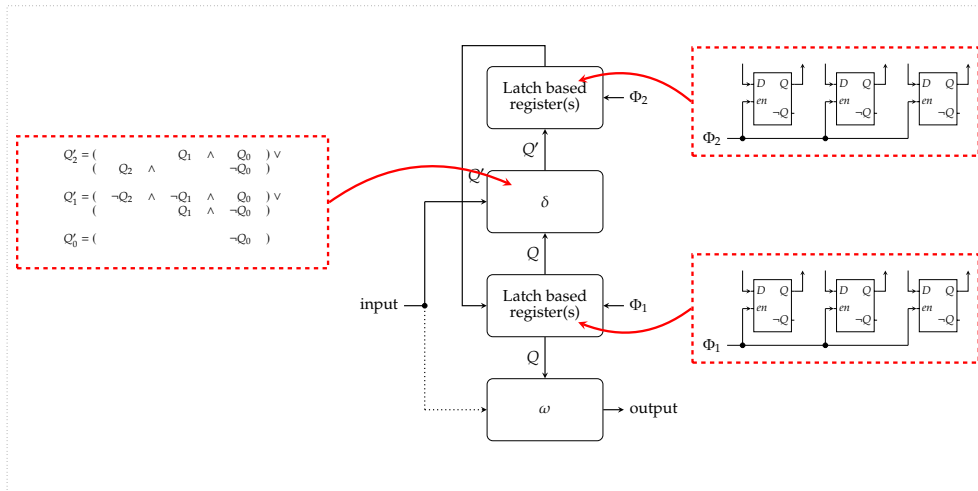
- If  $n = 6$  for example, then we want a component whose output  $r$  steps through values

$0, 1, 2, 3, 4, 5, 0, 1, \dots$ ,

with the modular reduction representing control behaviour (versus the uncontrolled counter that was cyclic by default).

Example #1: a modulo 6 ascending counter

► Solution:



Notes:

- If  $n = 6$  for example, then we want a component whose output  $r$  steps through values

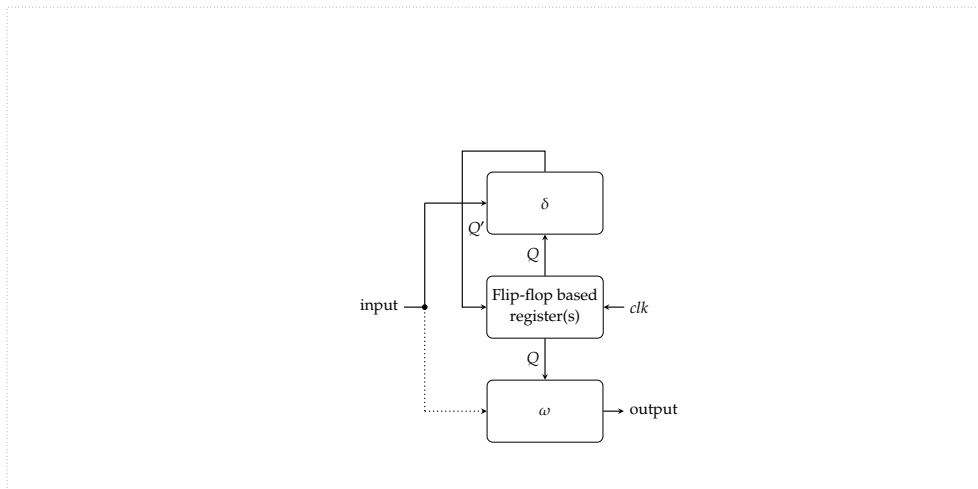
$0, 1, 2, 3, 4, 5, 0, 1, \dots,$

with the modular reduction representing control behaviour (versus the uncontrolled counter that was cyclic by default).

## Part 2.2: in practice, implementation (10)

Example #1: a modulo 6 ascending counter

► Solution:



Notes:

- If  $n = 6$  for example, then we want a component whose output  $r$  steps through values

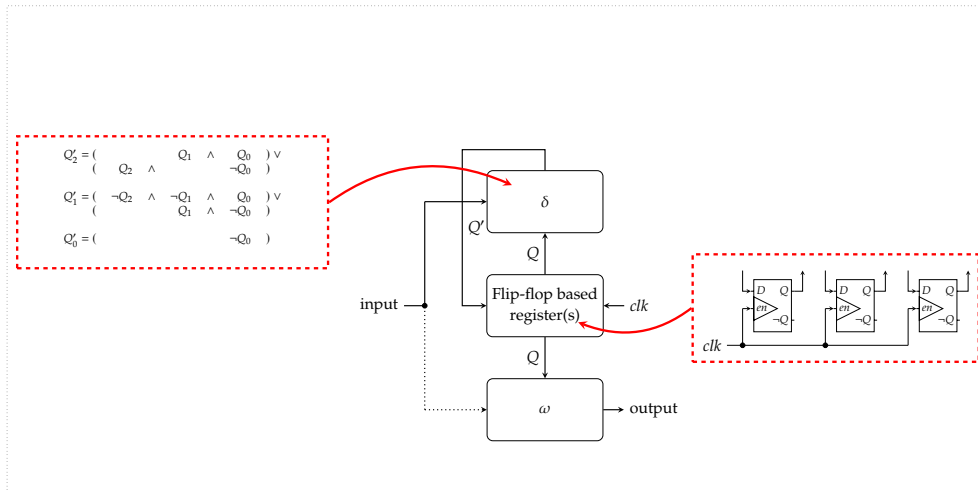
$0, 1, 2, 3, 4, 5, 0, 1, \dots,$

with the modular reduction representing control behaviour (versus the uncontrolled counter that was cyclic by default).

## Part 2.2: in practice, implementation (10)

Example #1: a modulo 6 ascending counter

### ► Solution:



Notes:

- If  $n = 6$  for example, then we want a component whose output  $r$  steps through values

0, 1, 2, 3, 4, 5, 0, 1, ... ,

with the modular reduction representing control behaviour (versus the uncontrolled counter that was cyclic by default).

## 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.

Notes:

- If  $n = 6$ , for example, then
  - if  $d = 0$ 
    - the output  $r$  steps through values 0, 1, 2, 3, 4, 5, 0, 1, ...
    - the output  $f = 1$  iff.  $r = 5$
  - if  $d = 1$ 
    - the output  $r$  steps through values 0, 5, 4, 3, 2, 1, 0, 5, ...
    - the output  $f = 1$  iff.  $r = 0$ .

► Solution:

Algorithm (tabular)					
Q	$\delta$		r	$\omega$	
	$d = 0$	$d = 1$		$d = 0$	$d = 1$
$S_0$	$S_1$	$S_5$	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)

Notes:

- If  $n = 6$ , for example, then
  - if  $d = 0$ 
    - the output  $r$  steps through values  $0, 1, 2, 3, 4, 5, 0, 1, \dots$
    - the output  $f = 1$  iff.  $r = 5$
  - if  $d = 1$ 
    - the output  $r$  steps through values  $0, 5, 4, 3, 2, 1, 0, 5, \dots$
    - the output  $f = 1$  iff.  $r = 0$ .

► Solution:

► there are 6 abstract labels

$S_0$	$\mapsto$	0
$S_1$	$\mapsto$	1
$S_2$	$\mapsto$	2
$S_3$	$\mapsto$	3
$S_4$	$\mapsto$	4
$S_5$	$\mapsto$	5

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

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

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

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

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

Notes:

- If  $n = 6$ , for example, then
  - if  $d = 0$ 
    - the output  $r$  steps through values  $0, 1, 2, 3, 4, 5, 0, 1, \dots$
    - the output  $f = 1$  iff.  $r = 5$
  - if  $d = 1$ 
    - the output  $r$  steps through values  $0, 5, 4, 3, 2, 1, 0, 5, \dots$
    - the output  $f = 1$  iff.  $r = 0$ .

## Part 2.2: in practice, implementation (11)

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

### ► Solution:

- rewriting the abstract labels yields the following concrete truth table

				$\delta$			$\omega$			
$d$	$Q_2$	$Q_1$	$Q_0$	$Q'_2$	$Q'_1$	$Q'_0$	$r_2$	$r_1$	$r_0$	$f$
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	?	?	?	?	?	?	?

Notes:

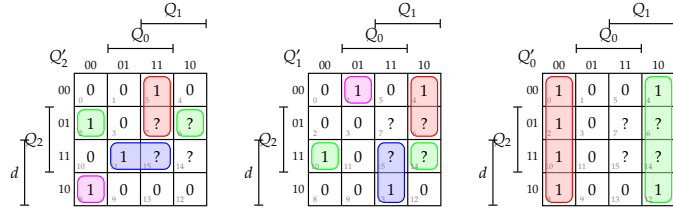
- If  $n = 6$ , for example, then
  - if  $d = 0$ 
    - the output  $r$  steps through values 0, 1, 2, 3, 4, 5, 0, 1, ...
    - the output  $f = 1$  iff.  $r = 5$
  - if  $d = 1$ 
    - the output  $r$  steps through values 0, 5, 4, 3, 2, 1, 0, 5, ...
    - the output  $f = 1$  iff.  $r = 0$ .

## Part 2.2: in practice, implementation (11)

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

### ► Solution:

- the truth table can be translated into



- doing so yields the following Boolean expressions for  $\delta$ :

$$Q'_2 = ( \neg d \quad \quad \quad \wedge \quad Q_1 \quad \wedge \quad Q_0 ) \vee \\ ( \neg d \quad \wedge \quad Q_2 \quad \quad \quad \wedge \quad \neg Q_0 ) \vee \\ ( \quad d \quad \wedge \quad Q_2 \quad \quad \quad \wedge \quad Q_0 ) \vee \\ ( \quad d \quad \wedge \quad \neg Q_2 \quad \wedge \quad \neg Q_1 \quad \wedge \quad \neg Q_0 )$$

$$Q'_1 = ( \neg d \quad \wedge \quad \neg Q_2 \quad \wedge \quad \neg Q_1 \quad \wedge \quad Q_0 ) \vee \\ ( \neg d \quad \quad \quad \wedge \quad Q_1 \quad \wedge \quad \neg Q_0 ) \vee \\ ( \quad d \quad \wedge \quad Q_2 \quad \quad \quad \wedge \quad \neg Q_0 ) \vee \\ ( \quad d \quad \quad \quad \wedge \quad Q_1 \quad \wedge \quad Q_0 )$$

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

Notes:

- If  $n = 6$ , for example, then
  - if  $d = 0$ 
    - the output  $r$  steps through values 0, 1, 2, 3, 4, 5, 0, 1, ...
    - the output  $f = 1$  iff.  $r = 5$
  - if  $d = 1$ 
    - the output  $r$  steps through values 0, 5, 4, 3, 2, 1, 0, 5, ...
    - the output  $f = 1$  iff.  $r = 0$ .

## Part 2.2: in practice, implementation (11)

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

### ► Solution:

- the truth table can be translated into

		$Q_1$			
		$Q_0$			
$f$		00	01	11	10
	00	0	0	0	0
	01	0	1	?	?
	11	0	0	?	?
	10	1	0	0	0
$d$		00	01	11	10
	00	0	0	?	?
	01	0	1	?	?
	11	0	0	?	?
	10	1	0	0	0

- doing so yields the following Boolean expressions for  $\omega$ :

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

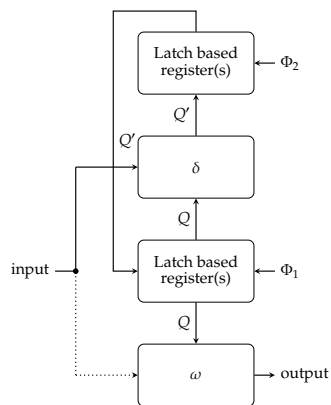
Notes:

- If  $n = 6$ , for example, then
  - if  $d = 0$ 
    - the output  $r$  steps through values 0, 1, 2, 3, 4, 5, 0, 1, ...
    - the output  $f = 1$  iff.  $r = 5$
  - if  $d = 1$ 
    - the output  $r$  steps through values 0, 5, 4, 3, 2, 1, 0, 5, ...
    - the output  $f = 1$  iff.  $r = 0$ .

## Part 2.2: in practice, implementation (11)

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

### ► Solution:



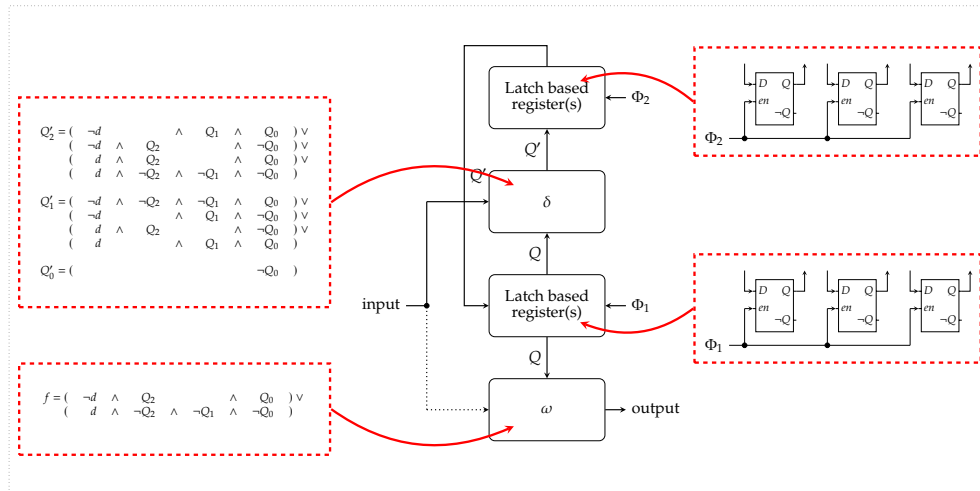
Notes:

- If  $n = 6$ , for example, then
  - if  $d = 0$ 
    - the output  $r$  steps through values 0, 1, 2, 3, 4, 5, 0, 1, ...
    - the output  $f = 1$  iff.  $r = 5$
  - if  $d = 1$ 
    - the output  $r$  steps through values 0, 5, 4, 3, 2, 1, 0, 5, ...
    - the output  $f = 1$  iff.  $r = 0$ .

## Part 2.2: in practice, implementation (11)

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

### ► Solution:



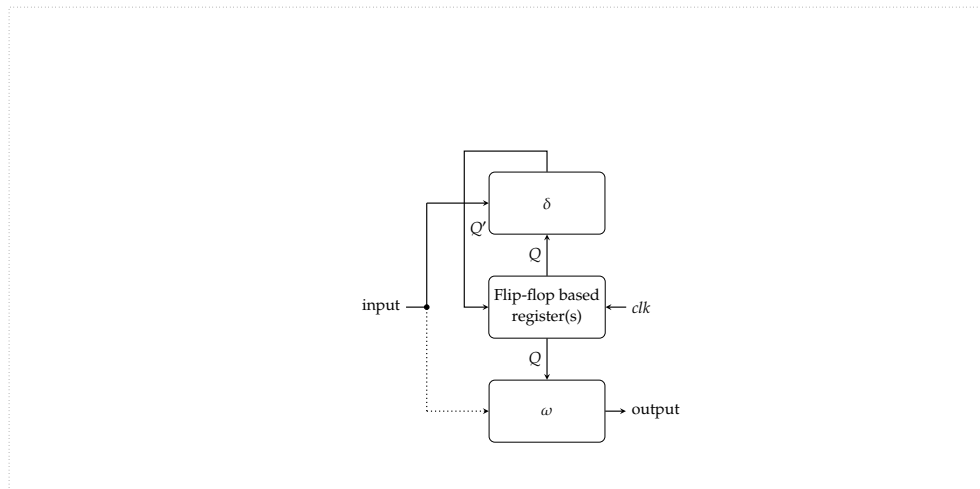
Notes:

- If  $n = 6$ , for example, then
  - if  $d = 0$ 
    - the output  $r$  steps through values 0, 1, 2, 3, 4, 5, 0, 1, ...
    - the output  $f = 1$  iff.  $r = 5$
  - if  $d = 1$ 
    - the output  $r$  steps through values 0, 5, 4, 3, 2, 1, 0, 5, ...
    - the output  $f = 1$  iff.  $r = 0$ .

## Part 2.2: in practice, implementation (11)

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

### ► Solution:



Notes:

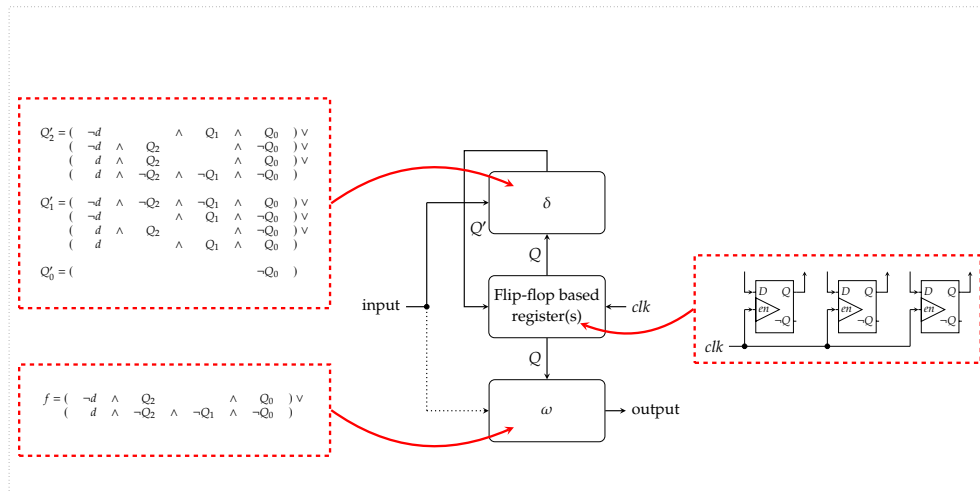
- If  $n = 6$ , for example, then
  - if  $d = 0$ 
    - the output  $r$  steps through values 0, 1, 2, 3, 4, 5, 0, 1, ...
    - the output  $f = 1$  iff.  $r = 5$
  - if  $d = 1$ 
    - the output  $r$  steps through values 0, 5, 4, 3, 2, 1, 0, 5, ...
    - the output  $f = 1$  iff.  $r = 0$ .



## Part 2.2: in practice, implementation (11)

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

### ► Solution:



Notes:

- If  $n = 6$ , for example, then
  - if  $d = 0$ 
    - the output  $r$  steps through values 0, 1, 2, 3, 4, 5, 0, 1, ...
    - the output  $f = 1$  iff.  $r = 5$
  - if  $d = 1$ 
    - the output  $r$  steps through values 0, 5, 4, 3, 2, 1, 0, 5, ...
    - the output  $f = 1$  iff.  $r = 0$ .

## Part 2.2: in practice, implementation (12)

Example #3: a loop counter

### ► 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     $\equiv$     so     $i = m$   
the end of iteration     $\equiv$     when  $i = n$

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

Notes:

## Part 2.2: in practice, implementation (13)

Example #3: a loop counter

### ► 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. ...

Notes:

- A shared clock might *seem* the ideal option, but there are (at least) two scenarios where it doesn't work out so well:
  1. When  $C_2$  and  $C_1$  are physically separate: distribution of a synchronised clock will be significantly hard(er). and
  2. When the number of steps  $C_2$  takes is variable: even if  $C_2$  and  $C_1$  are synchronised, the latter still needs some way to tell when the former has completed a given computation.
- The diagrammatic example can be explained as follows:
  - Initially,  $req = 0$  and  $ack = 0$ .
  - At some point,  $C_1$  wants to start a computation. It proceeds by 1) driving values onto any inputs (e.g.,  $x$ ) then 2) changing  $req$  from 0 to 1.
  - $C_2$  notices the change to (e.g., positive edge on)  $req$  and concludes that the inputs are available.
  - $C_2$  computes the outputs from the inputs (e.g.,  $r = f(x)$ ).
  - At some point,  $C_2$  finishes the computation. It proceeds by 1) driving values onto any outputs (e.g.,  $r$ ) then 2) changing  $ack$  from 0 to 1.
  - $C_1$  notices the change to (e.g., positive edge on)  $ack$  and concludes that the outputs are available. It proceeds by 1) storing any outputs ready for subsequent use, then 2) changing  $req$  from 1 to 0.
  - $C_2$  notices the change to (e.g., negative edge on)  $req$  and concludes that the interaction is finished. It proceeds by changing  $ack$  from 1 to 0.
  - $C_1$  notices the change to (e.g., negative edge on)  $ack$  and concludes that the interaction is finished.
  - Since both  $req$  and  $ack$  are 0 again, the module and user are ready to engage in successive interactions if/when need be.

## Part 2.2: in practice, implementation (13)

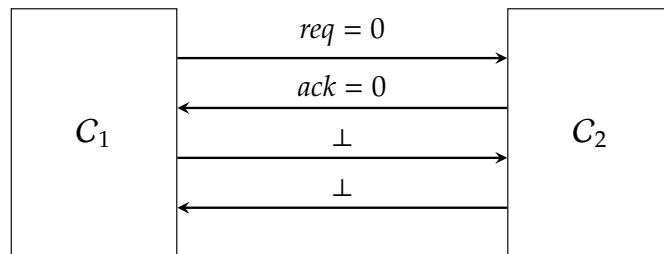
Example #3: a loop counter

### ► 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



Notes:

- A shared clock might *seem* the ideal option, but there are (at least) two scenarios where it doesn't work out so well:
  1. When  $C_2$  and  $C_1$  are physically separate: distribution of a synchronised clock will be significantly hard(er). and
  2. When the number of steps  $C_2$  takes is variable: even if  $C_2$  and  $C_1$  are synchronised, the latter still needs some way to tell when the former has completed a given computation.
- The diagrammatic example can be explained as follows:
  - Initially,  $req = 0$  and  $ack = 0$ .
  - At some point,  $C_1$  wants to start a computation. It proceeds by 1) driving values onto any inputs (e.g.,  $x$ ) then 2) changing  $req$  from 0 to 1.
  - $C_2$  notices the change to (e.g., positive edge on)  $req$  and concludes that the inputs are available.
  - $C_2$  computes the outputs from the inputs (e.g.,  $r = f(x)$ ).
  - At some point,  $C_2$  finishes the computation. It proceeds by 1) driving values onto any outputs (e.g.,  $r$ ) then 2) changing  $ack$  from 0 to 1.
  - $C_1$  notices the change to (e.g., positive edge on)  $ack$  and concludes that the outputs are available. It proceeds by 1) storing any outputs ready for subsequent use, then 2) changing  $req$  from 1 to 0.
  - $C_2$  notices the change to (e.g., negative edge on)  $req$  and concludes that the interaction is finished. It proceeds by changing  $ack$  from 1 to 0.
  - $C_1$  notices the change to (e.g., negative edge on)  $ack$  and concludes that the interaction is finished.
  - Since both  $req$  and  $ack$  are 0 again, the module and user are ready to engage in successive interactions if/when need be.

## Part 2.2: in practice, implementation (13)

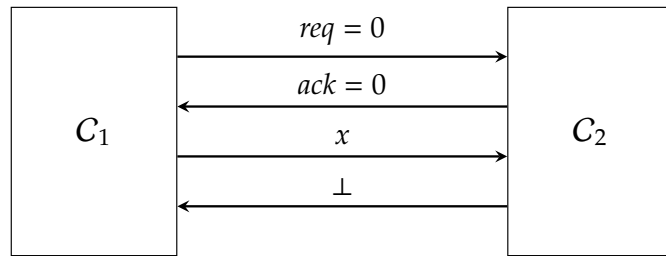
Example #3: a loop counter

### ► 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



#### Notes:

- A shared clock might *seem* the ideal option, but there are (at least) two scenarios where it doesn't work out so well:
  1. When  $C_2$  and  $C_1$  are physically separate: distribution of a synchronised clock will be significantly hard(er). and
  2. When the number of steps  $C_2$  takes is variable: even if  $C_2$  and  $C_1$  are synchronised, the latter still needs some way to tell when the former has completed a given computation.
- The diagrammatic example can be explained as follows:
  - Initially,  $req = 0$  and  $ack = 0$ .
  - At some point,  $C_1$  wants to start a computation. It proceeds by 1) driving values onto any inputs (e.g.,  $x$ ) then 2) changing  $req$  from 0 to 1.
  - $C_2$  notices the change to (e.g., positive edge on)  $req$  and concludes that the inputs are available.
  - $C_2$  computes the outputs from the inputs (e.g.,  $r = f(x)$ ).
  - At some point,  $C_2$  finishes the computation. It proceeds by 1) driving values onto any outputs (e.g.,  $r$ ) then 2) changing  $ack$  from 0 to 1.
  - $C_1$  notices the change to (e.g., positive edge on)  $ack$  and concludes that the outputs are available. It proceeds by 1) storing any outputs ready for subsequent use, then 2) changing  $req$  from 1 to 0.
  - $C_2$  notices the change to (e.g., negative edge on)  $req$  and concludes that the interaction is finished. It proceeds by changing  $ack$  from 1 to 0.
  - $C_1$  notices the change to (e.g., negative edge on)  $ack$  and concludes that the interaction is finished.
  - Since both  $req$  and  $ack$  are 0 again, the module and user are ready to engage in successive interactions if/when need be.

## Part 2.2: in practice, implementation (13)

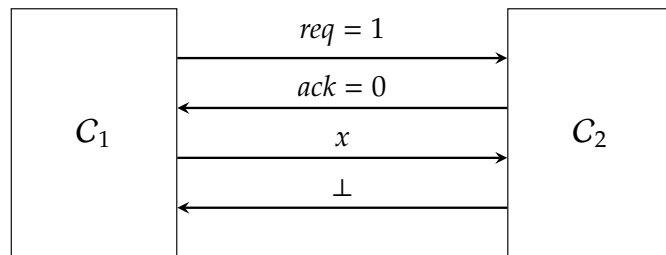
Example #3: a loop counter

### ► 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



#### Notes:

- A shared clock might *seem* the ideal option, but there are (at least) two scenarios where it doesn't work out so well:
  1. When  $C_2$  and  $C_1$  are physically separate: distribution of a synchronised clock will be significantly hard(er). and
  2. When the number of steps  $C_2$  takes is variable: even if  $C_2$  and  $C_1$  are synchronised, the latter still needs some way to tell when the former has completed a given computation.
- The diagrammatic example can be explained as follows:
  - Initially,  $req = 0$  and  $ack = 0$ .
  - At some point,  $C_1$  wants to start a computation. It proceeds by 1) driving values onto any inputs (e.g.,  $x$ ) then 2) changing  $req$  from 0 to 1.
  - $C_2$  notices the change to (e.g., positive edge on)  $req$  and concludes that the inputs are available.
  - $C_2$  computes the outputs from the inputs (e.g.,  $r = f(x)$ ).
  - At some point,  $C_2$  finishes the computation. It proceeds by 1) driving values onto any outputs (e.g.,  $r$ ) then 2) changing  $ack$  from 0 to 1.
  - $C_1$  notices the change to (e.g., positive edge on)  $ack$  and concludes that the outputs are available. It proceeds by 1) storing any outputs ready for subsequent use, then 2) changing  $req$  from 1 to 0.
  - $C_2$  notices the change to (e.g., negative edge on)  $req$  and concludes that the interaction is finished. It proceeds by changing  $ack$  from 1 to 0.
  - $C_1$  notices the change to (e.g., negative edge on)  $ack$  and concludes that the interaction is finished.
  - Since both  $req$  and  $ack$  are 0 again, the module and user are ready to engage in successive interactions if/when need be.

## Part 2.2: in practice, implementation (13)

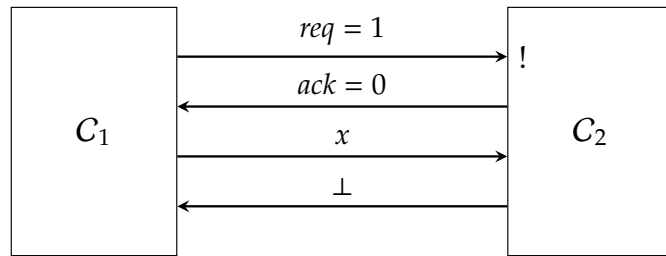
Example #3: a loop counter

### ► 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



## Part 2.2: in practice, implementation (13)

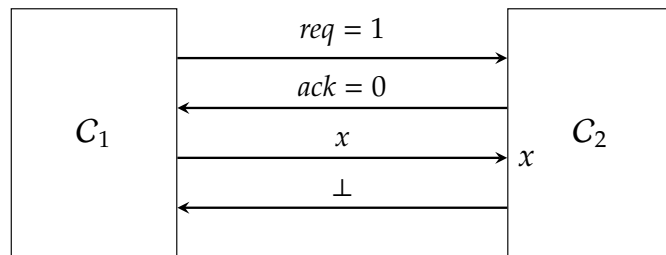
Example #3: a loop counter

### ► 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



#### Notes:

- A shared clock might *seem* the ideal option, but there are (at least) two scenarios where it doesn't work out so well:
  1. When  $C_2$  and  $C_1$  are physically separate: distribution of a synchronised clock will be significantly hard(er). and
  2. When the number of steps  $C_2$  takes is variable: even if  $C_2$  and  $C_1$  are synchronised, the latter still needs some way to tell when the former has completed a given computation.
- The diagrammatic example can be explained as follows:
  - Initially,  $req = 0$  and  $ack = 0$ .
  - At some point,  $C_1$  wants to start a computation. It proceeds by 1) driving values onto any inputs (e.g.,  $x$ ) then 2) changing  $req$  from 0 to 1.
  - $C_2$  notices the change to (e.g., positive edge on)  $req$  and concludes that the inputs are available.
  - $C_2$  computes the outputs from the inputs (e.g.,  $r = f(x)$ ).
  - At some point,  $C_2$  finishes the computation. It proceeds by 1) driving values onto any outputs (e.g.,  $r$ ) then 2) changing  $ack$  from 0 to 1.
  - $C_1$  notices the change to (e.g., positive edge on)  $ack$  and concludes that the outputs are available. It proceeds by 1) storing any outputs ready for subsequent use, then 2) changing  $req$  from 1 to 0.
  - $C_2$  notices the change to (e.g., negative edge on)  $req$  and concludes that the interaction is finished. It proceeds by changing  $ack$  from 1 to 0.
  - $C_1$  notices the change to (e.g., negative edge on)  $ack$  and concludes that the interaction is finished.
  - Since both  $req$  and  $ack$  are 0 again, the module and user are ready to engage in successive interactions if/when need be.

#### Notes:

- A shared clock might *seem* the ideal option, but there are (at least) two scenarios where it doesn't work out so well:
  1. When  $C_2$  and  $C_1$  are physically separate: distribution of a synchronised clock will be significantly hard(er). and
  2. When the number of steps  $C_2$  takes is variable: even if  $C_2$  and  $C_1$  are synchronised, the latter still needs some way to tell when the former has completed a given computation.
- The diagrammatic example can be explained as follows:
  - Initially,  $req = 0$  and  $ack = 0$ .
  - At some point,  $C_1$  wants to start a computation. It proceeds by 1) driving values onto any inputs (e.g.,  $x$ ) then 2) changing  $req$  from 0 to 1.
  - $C_2$  notices the change to (e.g., positive edge on)  $req$  and concludes that the inputs are available.
  - $C_2$  computes the outputs from the inputs (e.g.,  $r = f(x)$ ).
  - At some point,  $C_2$  finishes the computation. It proceeds by 1) driving values onto any outputs (e.g.,  $r$ ) then 2) changing  $ack$  from 0 to 1.
  - $C_1$  notices the change to (e.g., positive edge on)  $ack$  and concludes that the outputs are available. It proceeds by 1) storing any outputs ready for subsequent use, then 2) changing  $req$  from 1 to 0.
  - $C_2$  notices the change to (e.g., negative edge on)  $req$  and concludes that the interaction is finished. It proceeds by changing  $ack$  from 1 to 0.
  - $C_1$  notices the change to (e.g., negative edge on)  $ack$  and concludes that the interaction is finished.
  - Since both  $req$  and  $ack$  are 0 again, the module and user are ready to engage in successive interactions if/when need be.

## Part 2.2: in practice, implementation (13)

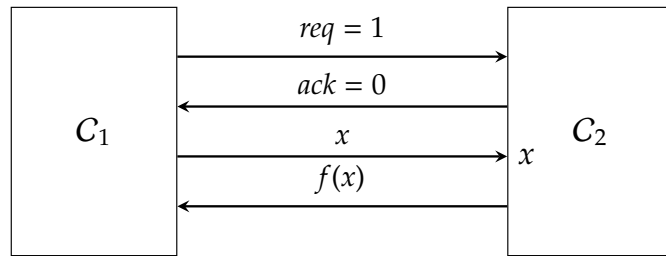
Example #3: a loop counter

### ► 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



## Part 2.2: in practice, implementation (13)

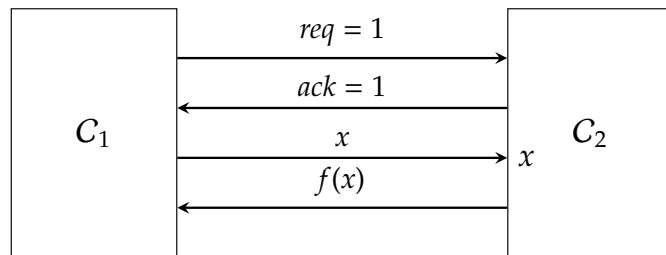
Example #3: a loop counter

### ► 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



#### Notes:

- A shared clock might *seem* the ideal option, but there are (at least) two scenarios where it doesn't work out so well:
  1. When  $C_2$  and  $C_1$  are physically separate: distribution of a synchronised clock will be significantly hard(er). and
  2. When the number of steps  $C_2$  takes is variable: even if  $C_2$  and  $C_1$  are synchronised, the latter still needs some way to tell when the former has completed a given computation.
- The diagrammatic example can be explained as follows:
  - Initially,  $req = 0$  and  $ack = 0$ .
  - At some point,  $C_1$  wants to start a computation. It proceeds by 1) driving values onto any inputs (e.g.,  $x$ ) then 2) changing  $req$  from 0 to 1.
  - $C_2$  notices the change to (e.g., positive edge on)  $req$  and concludes that the inputs are available.
  - $C_2$  computes the outputs from the inputs (e.g.,  $r = f(x)$ ).
  - At some point,  $C_2$  finishes the computation. It proceeds by 1) driving values onto any outputs (e.g.,  $r$ ) then 2) changing  $ack$  from 0 to 1.
  - $C_1$  notices the change to (e.g., positive edge on)  $ack$  and concludes that the outputs are available. It proceeds by 1) storing any outputs ready for subsequent use, then 2) changing  $req$  from 1 to 0.
  - $C_2$  notices the change to (e.g., negative edge on)  $req$  and concludes that the interaction is finished. It proceeds by changing  $ack$  from 1 to 0.
  - $C_1$  notices the change to (e.g., negative edge on)  $ack$  and concludes that the interaction is finished.
  - Since both  $req$  and  $ack$  are 0 again, the module and user are ready to engage in successive interactions if/when need be.

#### Notes:

- A shared clock might *seem* the ideal option, but there are (at least) two scenarios where it doesn't work out so well:
  1. When  $C_2$  and  $C_1$  are physically separate: distribution of a synchronised clock will be significantly hard(er). and
  2. When the number of steps  $C_2$  takes is variable: even if  $C_2$  and  $C_1$  are synchronised, the latter still needs some way to tell when the former has completed a given computation.
- The diagrammatic example can be explained as follows:
  - Initially,  $req = 0$  and  $ack = 0$ .
  - At some point,  $C_1$  wants to start a computation. It proceeds by 1) driving values onto any inputs (e.g.,  $x$ ) then 2) changing  $req$  from 0 to 1.
  - $C_2$  notices the change to (e.g., positive edge on)  $req$  and concludes that the inputs are available.
  - $C_2$  computes the outputs from the inputs (e.g.,  $r = f(x)$ ).
  - At some point,  $C_2$  finishes the computation. It proceeds by 1) driving values onto any outputs (e.g.,  $r$ ) then 2) changing  $ack$  from 0 to 1.
  - $C_1$  notices the change to (e.g., positive edge on)  $ack$  and concludes that the outputs are available. It proceeds by 1) storing any outputs ready for subsequent use, then 2) changing  $req$  from 1 to 0.
  - $C_2$  notices the change to (e.g., negative edge on)  $req$  and concludes that the interaction is finished. It proceeds by changing  $ack$  from 1 to 0.
  - $C_1$  notices the change to (e.g., negative edge on)  $ack$  and concludes that the interaction is finished.
  - Since both  $req$  and  $ack$  are 0 again, the module and user are ready to engage in successive interactions if/when need be.

## Part 2.2: in practice, implementation (13)

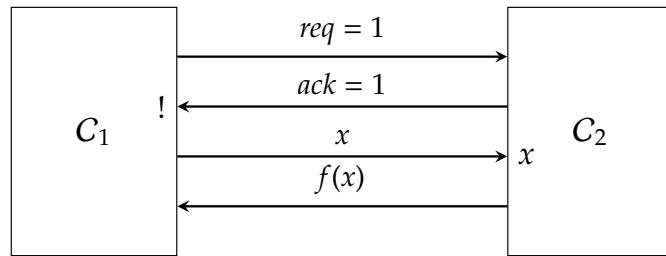
Example #3: a loop counter

### ► 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



## Part 2.2: in practice, implementation (13)

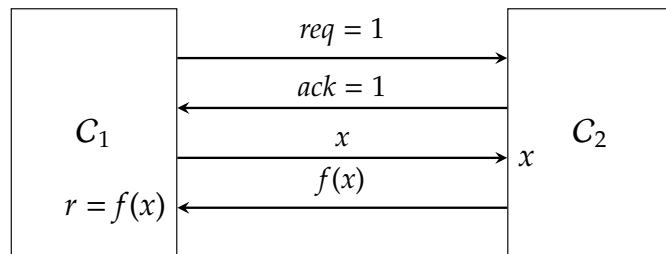
Example #3: a loop counter

### ► 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



#### Notes:

- A shared clock might *seem* the ideal option, but there are (at least) two scenarios where it doesn't work out so well:
  1. When  $C_2$  and  $C_1$  are physically separate: distribution of a synchronised clock will be significantly hard(er). and
  2. When the number of steps  $C_2$  takes is variable: even if  $C_2$  and  $C_1$  are synchronised, the latter still needs some way to tell when the former has completed a given computation.
- The diagrammatic example can be explained as follows:
  - Initially,  $req = 0$  and  $ack = 0$ .
  - At some point,  $C_1$  wants to start a computation. It proceeds by 1) driving values onto any inputs (e.g.,  $x$ ) then 2) changing  $req$  from 0 to 1.
  - $C_2$  notices the change to (e.g., positive edge on)  $req$  and concludes that the inputs are available.
  - $C_2$  computes the outputs from the inputs (e.g.,  $r = f(x)$ ).
  - At some point,  $C_2$  finishes the computation. It proceeds by 1) driving values onto any outputs (e.g.,  $r$ ) then 2) changing  $ack$  from 0 to 1.
  - $C_1$  notices the change to (e.g., positive edge on)  $ack$  and concludes that the outputs are available. It proceeds by 1) storing any outputs ready for subsequent use, then 2) changing  $req$  from 1 to 0.
  - $C_2$  notices the change to (e.g., negative edge on)  $req$  and concludes that the interaction is finished. It proceeds by changing  $ack$  from 1 to 0.
  - $C_1$  notices the change to (e.g., negative edge on)  $ack$  and concludes that the interaction is finished.
  - Since both  $req$  and  $ack$  are 0 again, the module and user are ready to engage in successive interactions if/when need be.

#### Notes:

- A shared clock might *seem* the ideal option, but there are (at least) two scenarios where it doesn't work out so well:
  1. When  $C_2$  and  $C_1$  are physically separate: distribution of a synchronised clock will be significantly hard(er). and
  2. When the number of steps  $C_2$  takes is variable: even if  $C_2$  and  $C_1$  are synchronised, the latter still needs some way to tell when the former has completed a given computation.
- The diagrammatic example can be explained as follows:
  - Initially,  $req = 0$  and  $ack = 0$ .
  - At some point,  $C_1$  wants to start a computation. It proceeds by 1) driving values onto any inputs (e.g.,  $x$ ) then 2) changing  $req$  from 0 to 1.
  - $C_2$  notices the change to (e.g., positive edge on)  $req$  and concludes that the inputs are available.
  - $C_2$  computes the outputs from the inputs (e.g.,  $r = f(x)$ ).
  - At some point,  $C_2$  finishes the computation. It proceeds by 1) driving values onto any outputs (e.g.,  $r$ ) then 2) changing  $ack$  from 0 to 1.
  - $C_1$  notices the change to (e.g., positive edge on)  $ack$  and concludes that the outputs are available. It proceeds by 1) storing any outputs ready for subsequent use, then 2) changing  $req$  from 1 to 0.
  - $C_2$  notices the change to (e.g., negative edge on)  $req$  and concludes that the interaction is finished. It proceeds by changing  $ack$  from 1 to 0.
  - $C_1$  notices the change to (e.g., negative edge on)  $ack$  and concludes that the interaction is finished.
  - Since both  $req$  and  $ack$  are 0 again, the module and user are ready to engage in successive interactions if/when need be.

## Part 2.2: in practice, implementation (13)

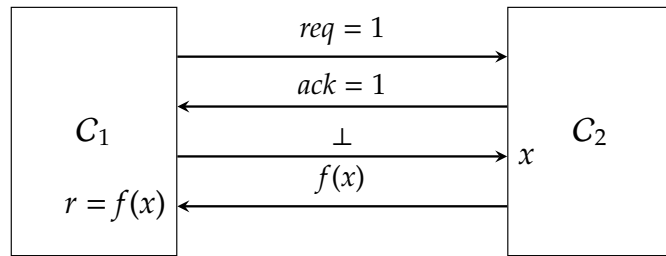
### Example #3: a loop counter

#### ► 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



##### Notes:

- A shared clock might *seem* the ideal option, but there are (at least) two scenarios where it doesn't work out so well:
  1. When  $C_2$  and  $C_1$  are physically separate: distribution of a synchronised clock will be significantly hard(er). and
  2. When the number of steps  $C_2$  takes is variable: even if  $C_2$  and  $C_1$  are synchronised, the latter still needs some way to tell when the former has completed a given computation.
- The diagrammatic example can be explained as follows:
  - Initially,  $req = 0$  and  $ack = 0$ .
  - At some point,  $C_1$  wants to start a computation. It proceeds by 1) driving values onto any inputs (e.g.,  $x$ ) then 2) changing  $req$  from 0 to 1.
  - $C_2$  notices the change to (e.g., positive edge on)  $req$  and concludes that the inputs are available.
  - $C_2$  computes the outputs from the inputs (e.g.,  $r = f(x)$ ).
  - At some point,  $C_2$  finishes the computation. It proceeds by 1) driving values onto any outputs (e.g.,  $r$ ) then 2) changing  $ack$  from 0 to 1.
  - $C_1$  notices the change to (e.g., positive edge on)  $ack$  and concludes that the outputs are available. It proceeds by 1) storing any outputs ready for subsequent use, then 2) changing  $req$  from 1 to 0.
  - $C_2$  notices the change to (e.g., negative edge on)  $req$  and concludes that the interaction is finished. It proceeds by changing  $ack$  from 1 to 0.
  - $C_1$  notices the change to (e.g., negative edge on)  $ack$  and concludes that the interaction is finished.
  - Since both  $req$  and  $ack$  are 0 again, the module and user are ready to engage in successive interactions if/when need be.

## Part 2.2: in practice, implementation (13)

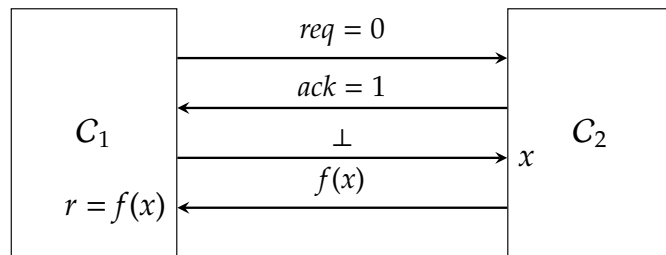
### Example #3: a loop counter

#### ► 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



##### Notes:

- A shared clock might *seem* the ideal option, but there are (at least) two scenarios where it doesn't work out so well:
  1. When  $C_2$  and  $C_1$  are physically separate: distribution of a synchronised clock will be significantly hard(er). and
  2. When the number of steps  $C_2$  takes is variable: even if  $C_2$  and  $C_1$  are synchronised, the latter still needs some way to tell when the former has completed a given computation.
- The diagrammatic example can be explained as follows:
  - Initially,  $req = 0$  and  $ack = 0$ .
  - At some point,  $C_1$  wants to start a computation. It proceeds by 1) driving values onto any inputs (e.g.,  $x$ ) then 2) changing  $req$  from 0 to 1.
  - $C_2$  notices the change to (e.g., positive edge on)  $req$  and concludes that the inputs are available.
  - $C_2$  computes the outputs from the inputs (e.g.,  $r = f(x)$ ).
  - At some point,  $C_2$  finishes the computation. It proceeds by 1) driving values onto any outputs (e.g.,  $r$ ) then 2) changing  $ack$  from 0 to 1.
  - $C_1$  notices the change to (e.g., positive edge on)  $ack$  and concludes that the outputs are available. It proceeds by 1) storing any outputs ready for subsequent use, then 2) changing  $req$  from 1 to 0.
  - $C_2$  notices the change to (e.g., negative edge on)  $req$  and concludes that the interaction is finished. It proceeds by changing  $ack$  from 1 to 0.
  - $C_1$  notices the change to (e.g., negative edge on)  $ack$  and concludes that the interaction is finished.
  - Since both  $req$  and  $ack$  are 0 again, the module and user are ready to engage in successive interactions if/when need be.

## Part 2.2: in practice, implementation (13)

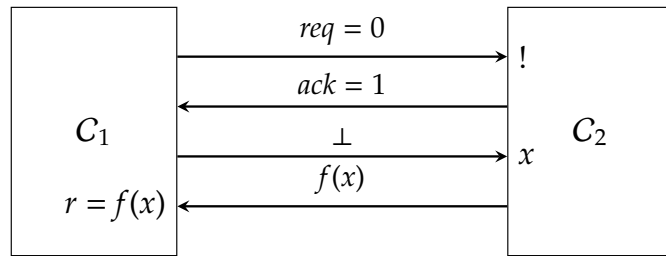
Example #3: a loop counter

### ► 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



#### Notes:

- A shared clock might *seem* the ideal option, but there are (at least) two scenarios where it doesn't work out so well:
  1. When  $C_2$  and  $C_1$  are physically separate: distribution of a synchronised clock will be significantly hard(er). and
  2. When the number of steps  $C_2$  takes is variable: even if  $C_2$  and  $C_1$  are synchronised, the latter still needs some way to tell when the former has completed a given computation.
- The diagrammatic example can be explained as follows:
  - Initially,  $req = 0$  and  $ack = 0$ .
  - At some point,  $C_1$  wants to start a computation. It proceeds by 1) driving values onto any inputs (e.g.,  $x$ ) then 2) changing  $req$  from 0 to 1.
  - $C_2$  notices the change to (e.g., positive edge on)  $req$  and concludes that the inputs are available.
  - $C_2$  computes the outputs from the inputs (e.g.,  $r = f(x)$ ).
  - At some point,  $C_2$  finishes the computation. It proceeds by 1) driving values onto any outputs (e.g.,  $r$ ) then 2) changing  $ack$  from 0 to 1.
  - $C_1$  notices the change to (e.g., positive edge on)  $ack$  and concludes that the outputs are available. It proceeds by 1) storing any outputs ready for subsequent use, then 2) changing  $req$  from 1 to 0.
  - $C_2$  notices the change to (e.g., negative edge on)  $req$  and concludes that the interaction is finished. It proceeds by changing  $ack$  from 1 to 0.
  - $C_1$  notices the change to (e.g., negative edge on)  $ack$  and concludes that the interaction is finished.
  - Since both  $req$  and  $ack$  are 0 again, the module and user are ready to engage in successive interactions if/when need be.

## Part 2.2: in practice, implementation (13)

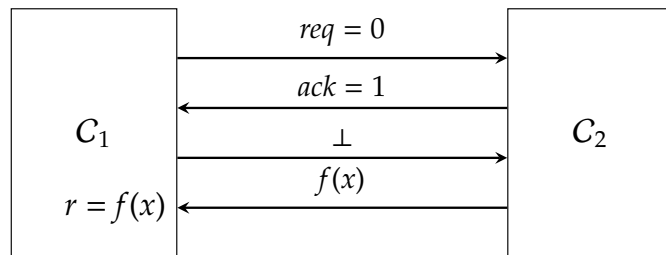
Example #3: a loop counter

### ► 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



#### Notes:

- A shared clock might *seem* the ideal option, but there are (at least) two scenarios where it doesn't work out so well:
  1. When  $C_2$  and  $C_1$  are physically separate: distribution of a synchronised clock will be significantly hard(er). and
  2. When the number of steps  $C_2$  takes is variable: even if  $C_2$  and  $C_1$  are synchronised, the latter still needs some way to tell when the former has completed a given computation.
- The diagrammatic example can be explained as follows:
  - Initially,  $req = 0$  and  $ack = 0$ .
  - At some point,  $C_1$  wants to start a computation. It proceeds by 1) driving values onto any inputs (e.g.,  $x$ ) then 2) changing  $req$  from 0 to 1.
  - $C_2$  notices the change to (e.g., positive edge on)  $req$  and concludes that the inputs are available.
  - $C_2$  computes the outputs from the inputs (e.g.,  $r = f(x)$ ).
  - At some point,  $C_2$  finishes the computation. It proceeds by 1) driving values onto any outputs (e.g.,  $r$ ) then 2) changing  $ack$  from 0 to 1.
  - $C_1$  notices the change to (e.g., positive edge on)  $ack$  and concludes that the outputs are available. It proceeds by 1) storing any outputs ready for subsequent use, then 2) changing  $req$  from 1 to 0.
  - $C_2$  notices the change to (e.g., negative edge on)  $req$  and concludes that the interaction is finished. It proceeds by changing  $ack$  from 1 to 0.
  - $C_1$  notices the change to (e.g., negative edge on)  $ack$  and concludes that the interaction is finished.
  - Since both  $req$  and  $ack$  are 0 again, the module and user are ready to engage in successive interactions if/when need be.



## Part 2.2: in practice, implementation (13)

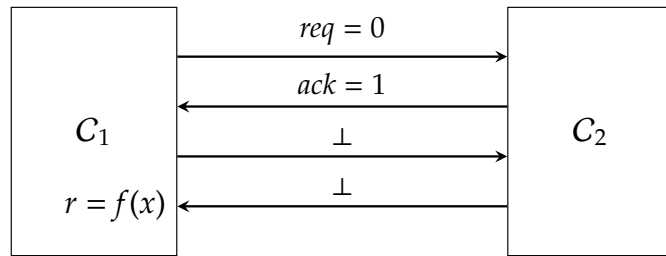
Example #3: a loop counter

### ► 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



#### Notes:

- A shared clock might *seem* the ideal option, but there are (at least) two scenarios where it doesn't work out so well:
  1. When  $C_2$  and  $C_1$  are physically separate: distribution of a synchronised clock will be significantly hard(er). and
  2. When the number of steps  $C_2$  takes is variable: even if  $C_2$  and  $C_1$  are synchronised, the latter still needs some way to tell when the former has completed a given computation.
- The diagrammatic example can be explained as follows:
  - Initially,  $req = 0$  and  $ack = 0$ .
  - At some point,  $C_1$  wants to start a computation. It proceeds by 1) driving values onto any inputs (e.g.,  $x$ ) then 2) changing  $req$  from 0 to 1.
  - $C_2$  notices the change to (e.g., positive edge on)  $req$  and concludes that the inputs are available.
  - $C_2$  computes the outputs from the inputs (e.g.,  $r = f(x)$ ).
  - At some point,  $C_2$  finishes the computation. It proceeds by 1) driving values onto any outputs (e.g.,  $r$ ) then 2) changing  $ack$  from 0 to 1.
  - $C_1$  notices the change to (e.g., positive edge on)  $ack$  and concludes that the outputs are available. It proceeds by 1) storing any outputs ready for subsequent use, then 2) changing  $req$  from 1 to 0.
  - $C_2$  notices the change to (e.g., negative edge on)  $req$  and concludes that the interaction is finished. It proceeds by changing  $ack$  from 1 to 0.
  - $C_1$  notices the change to (e.g., negative edge on)  $ack$  and concludes that the interaction is finished.
  - Since both  $req$  and  $ack$  are 0 again, the module and user are ready to engage in successive interactions if/when need be.

## Part 2.2: in practice, implementation (13)

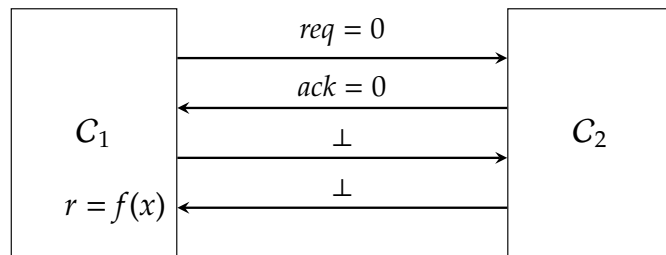
Example #3: a loop counter

### ► 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



#### Notes:

- A shared clock might *seem* the ideal option, but there are (at least) two scenarios where it doesn't work out so well:
  1. When  $C_2$  and  $C_1$  are physically separate: distribution of a synchronised clock will be significantly hard(er). and
  2. When the number of steps  $C_2$  takes is variable: even if  $C_2$  and  $C_1$  are synchronised, the latter still needs some way to tell when the former has completed a given computation.
- The diagrammatic example can be explained as follows:
  - Initially,  $req = 0$  and  $ack = 0$ .
  - At some point,  $C_1$  wants to start a computation. It proceeds by 1) driving values onto any inputs (e.g.,  $x$ ) then 2) changing  $req$  from 0 to 1.
  - $C_2$  notices the change to (e.g., positive edge on)  $req$  and concludes that the inputs are available.
  - $C_2$  computes the outputs from the inputs (e.g.,  $r = f(x)$ ).
  - At some point,  $C_2$  finishes the computation. It proceeds by 1) driving values onto any outputs (e.g.,  $r$ ) then 2) changing  $ack$  from 0 to 1.
  - $C_1$  notices the change to (e.g., positive edge on)  $ack$  and concludes that the outputs are available. It proceeds by 1) storing any outputs ready for subsequent use, then 2) changing  $req$  from 1 to 0.
  - $C_2$  notices the change to (e.g., negative edge on)  $req$  and concludes that the interaction is finished. It proceeds by changing  $ack$  from 1 to 0.
  - $C_1$  notices the change to (e.g., negative edge on)  $ack$  and concludes that the interaction is finished.
  - Since both  $req$  and  $ack$  are 0 again, the module and user are ready to engage in successive interactions if/when need be.

## Part 2.2: in practice, implementation (13)

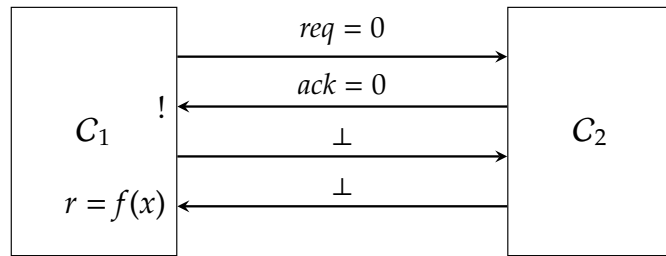
Example #3: a loop counter

### ► 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



## Part 2.2: in practice, implementation (13)

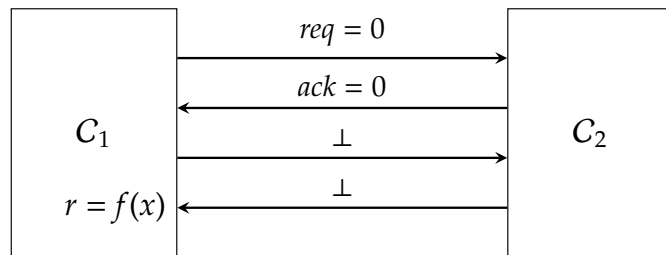
Example #3: a loop counter

### ► 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



#### Notes:

- A shared clock might *seem* the ideal option, but there are (at least) two scenarios where it doesn't work out so well:
  1. When  $C_2$  and  $C_1$  are physically separate: distribution of a synchronised clock will be significantly hard(er). and
  2. When the number of steps  $C_2$  takes is variable: even if  $C_2$  and  $C_1$  are synchronised, the latter still needs some way to tell when the former has completed a given computation.
- The diagrammatic example can be explained as follows:
  - Initially,  $req = 0$  and  $ack = 0$ .
  - At some point,  $C_1$  wants to start a computation. It proceeds by 1) driving values onto any inputs (e.g.,  $x$ ) then 2) changing  $req$  from 0 to 1.
  - $C_2$  notices the change to (e.g., positive edge on)  $req$  and concludes that the inputs are available.
  - $C_2$  computes the outputs from the inputs (e.g.,  $r = f(x)$ ).
  - At some point,  $C_2$  finishes the computation. It proceeds by 1) driving values onto any outputs (e.g.,  $r$ ) then 2) changing  $ack$  from 0 to 1.
  - $C_1$  notices the change to (e.g., positive edge on)  $ack$  and concludes that the outputs are available. It proceeds by 1) storing any outputs ready for subsequent use, then 2) changing  $req$  from 1 to 0.
  - $C_2$  notices the change to (e.g., negative edge on)  $req$  and concludes that the interaction is finished. It proceeds by changing  $ack$  from 1 to 0.
  - $C_1$  notices the change to (e.g., negative edge on)  $ack$  and concludes that the interaction is finished.
  - Since both  $req$  and  $ack$  are 0 again, the module and user are ready to engage in successive interactions if/when need be.

#### Notes:

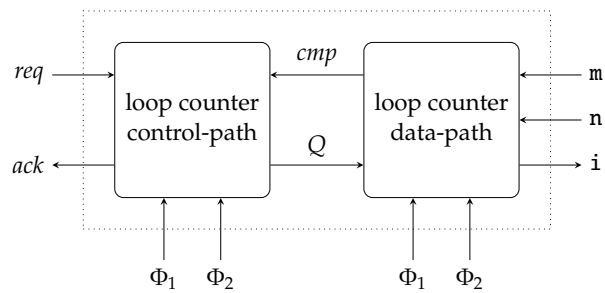
- A shared clock might *seem* the ideal option, but there are (at least) two scenarios where it doesn't work out so well:
  1. When  $C_2$  and  $C_1$  are physically separate: distribution of a synchronised clock will be significantly hard(er). and
  2. When the number of steps  $C_2$  takes is variable: even if  $C_2$  and  $C_1$  are synchronised, the latter still needs some way to tell when the former has completed a given computation.
- The diagrammatic example can be explained as follows:
  - Initially,  $req = 0$  and  $ack = 0$ .
  - At some point,  $C_1$  wants to start a computation. It proceeds by 1) driving values onto any inputs (e.g.,  $x$ ) then 2) changing  $req$  from 0 to 1.
  - $C_2$  notices the change to (e.g., positive edge on)  $req$  and concludes that the inputs are available.
  - $C_2$  computes the outputs from the inputs (e.g.,  $r = f(x)$ ).
  - At some point,  $C_2$  finishes the computation. It proceeds by 1) driving values onto any outputs (e.g.,  $r$ ) then 2) changing  $ack$  from 0 to 1.
  - $C_1$  notices the change to (e.g., positive edge on)  $ack$  and concludes that the outputs are available. It proceeds by 1) storing any outputs ready for subsequent use, then 2) changing  $req$  from 1 to 0.
  - $C_2$  notices the change to (e.g., negative edge on)  $req$  and concludes that the interaction is finished. It proceeds by changing  $ack$  from 1 to 0.
  - $C_1$  notices the change to (e.g., negative edge on)  $ack$  and concludes that the interaction is finished.
  - Since both  $req$  and  $ack$  are 0 again, the module and user are ready to engage in successive interactions if/when need be.

## Part 2.2: in practice, implementation (14)

Example #3: a loop counter

### ► 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.

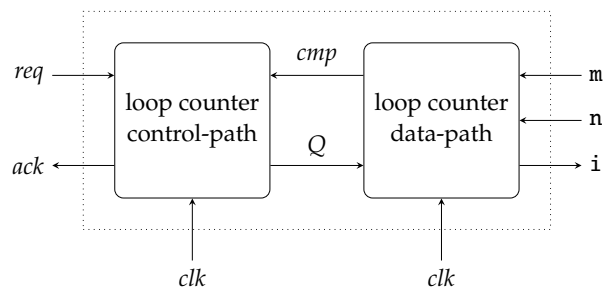
Notes:

## Part 2.2: in practice, implementation (14)

Example #3: a loop counter

### ► 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.

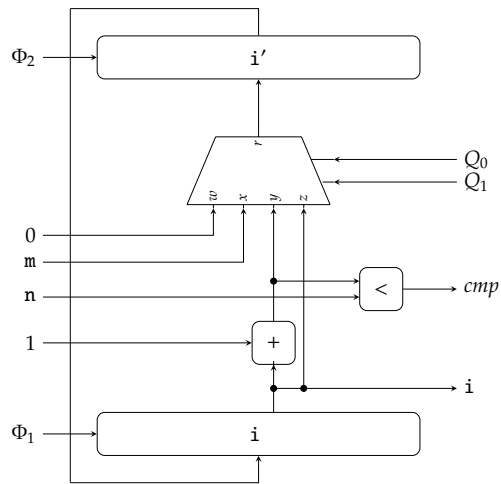
Notes:

## Part 2.2: in practice, implementation (15)

### Example #3: a loop counter

#### ► Solution: the data-path.

#### Circuit (latch version)



#### Notes:

- The general structure here is the same as the previous, uncontrolled counter example: for example there is 1) there is still a register to store the current counter state, 2) there is still have an adder, but 3) there is *also* now a multiplexer to decide between several options for the next state.
- Some of the inputs (e.g.,  $Q$ ) and outputs (e.g.,  $cmp$ ) shown need to be considered in combination with the control-path. For example, it should now be clear that
  - if  $Q = (0, 0) \mapsto S_{init}$ , then the multiplexer updates the output register with 0,
  - if  $Q = (1, 0) \mapsto S_{init}$  then the multiplexer updates the output register with  $m$ , i.e., the initial counter value,
  - if  $Q = (0, 1) \mapsto S_{step}$  then the multiplexer updates the output register with  $i + 1$ , i.e., the incremented counter value produced by the adder,
  - if  $Q = (1, 1) \mapsto S_{done}$  then the multiplexer updates the output register with  $i$ , i.e., the current counter value.

Likewise, it should be clear that

$$cmp = \begin{cases} 0 & \text{if } i < n \\ 1 & \text{if } i \geq n \end{cases}$$

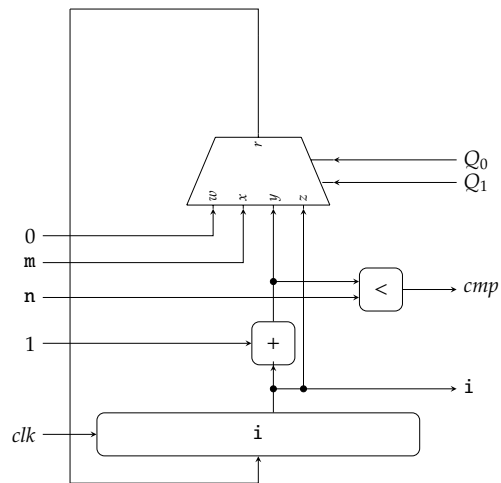
which controls the transition from  $S_{step}$  into  $S_{done}$ .

## Part 2.2: in practice, implementation (15)

### Example #3: a loop counter

#### ► Solution: the data-path.

#### Circuit (flip-flop version)



#### Notes:

- The general structure here is the same as the previous, uncontrolled counter example: for example there is 1) there is still a register to store the current counter state, 2) there is still have an adder, but 3) there is *also* now a multiplexer to decide between several options for the next state.
- Some of the inputs (e.g.,  $Q$ ) and outputs (e.g.,  $cmp$ ) shown need to be considered in combination with the control-path. For example, it should now be clear that
  - if  $Q = (0, 0) \mapsto S_{init}$ , then the multiplexer updates the output register with 0,
  - if  $Q = (1, 0) \mapsto S_{init}$  then the multiplexer updates the output register with  $m$ , i.e., the initial counter value,
  - if  $Q = (0, 1) \mapsto S_{step}$  then the multiplexer updates the output register with  $i + 1$ , i.e., the incremented counter value produced by the adder,
  - if  $Q = (1, 1) \mapsto S_{done}$  then the multiplexer updates the output register with  $i$ , i.e., the current counter value.

Likewise, it should be clear that

$$cmp = \begin{cases} 0 & \text{if } i < n \\ 1 & \text{if } i \geq n \end{cases}$$

which controls the transition from  $S_{step}$  into  $S_{done}$ .

## Part 2.2: in practice, implementation (16)

Example #3: a loop counter

► **Solution:** the control-path.

Algorithm (tabular)				
		$\delta$		$\omega$
	$Q$	$Q'$		$ack$
		$cmp = 0$	$cmp = 1$	$cmp = 0$ $cmp = 1$
$req = 0$	$S_{wait}$	$S_{wait}$	$S_{wait}$	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)

```

graph TD
    start((start)) --> S_wait((S_wait))
    S_wait -- req = 0 --> S_wait
    S_wait -- req = 1 --> S_init((S_init))
    S_init -- req = 0 --> S_step((S_step))
    S_step -- cmp = 1 --> S_step
    S_step -- cmp = 0 --> S_done((S_done))
    S_done -- req = 1 --> S_wait
    S_done -- req = 0 --> S_done
  
```

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  $ack = 1$ .

## Part 2.2: in practice, implementation (16)

Example #3: a loop counter

► **Solution:** the control-path.

- there are 4 abstract labels

$S_{wait}$	$\mapsto$	0
$S_{init}$	$\mapsto$	1
$S_{step}$	$\mapsto$	2
$S_{done}$	$\mapsto$	3

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

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

- 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).

Notes:

Notes:

## Part 2.2: in practice, implementation (16)

Example #3: a loop counter

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

				$\delta$		$\omega$
$req$	$cmp$	$Q_1$	$Q_0$	$Q'_1$	$Q'_0$	$ack$
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	0	0
1	1	1	1	1	1	1

Notes:

## Part 2.2: in practice, implementation (16)

Example #3: a loop counter

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

		$Q_1$			
		$Q_0$			
		00	01	11	10
$cmp$	00	0	0	0	0
	01	0	0	0	0
	11	0	1	1	1
	10	0	1	1	1
$req$	00	0	0	0	0
	01	0	0	0	0
	11	1	0	1	0
	10	1	0	1	1

- doing so yields the following Boolean expressions for  $\delta$ :

$$Q'_1 = ( req \wedge Q_0 ) \vee ( req \wedge Q_1 )$$

$$Q'_0 = ( req \wedge \neg Q_1 \wedge \neg Q_0 ) \vee ( req \wedge Q_1 \wedge Q_0 ) \vee ( req \wedge \neg cmp \wedge Q_1 )$$

Notes:

## Part 2.2: in practice, implementation (16)

Example #3: a loop counter

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

		$Q_0$		$Q_1$	
		00	01	11	10
$ack$	00	0	0	1	0
	01	0	0	1	0
$cmp$	11	0	0	1	0
	10	0	0	1	0
$req$	00	0	0	1	0
	01	0	0	1	0

- doing so yields the following Boolean expressions for  $\omega$ :

$$ack = Q_1 \wedge Q_0$$

Notes:

## Part 2.2: in practice, implementation (17)

Example #3: a loop counter

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

Algorithm	Circuit
<p><b>Input:</b> Two unsigned, <math>n</math>-bit, base-2 integers <math>x</math> and <math>y</math></p> <p><b>Output:</b> An unsigned, <math>2n</math>-bit, base-2 integer <math>r = y \cdot x</math></p> <pre> 1  <math>r \leftarrow 0</math> 2  <b>for</b> <math>i = n - 1</math> <b>downto</b> 0 <b>step</b> -1 <b>do</b> 3    <math>r \leftarrow 2 \cdot r</math> 4    <b>if</b> <math>y_i = 1</math> <b>then</b> 5      <math>r \leftarrow r + x</math> 6    <b>end</b> 7  <b>end</b> 8  <b>return</b> <math>r</math>                     </pre>	

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

Notes:

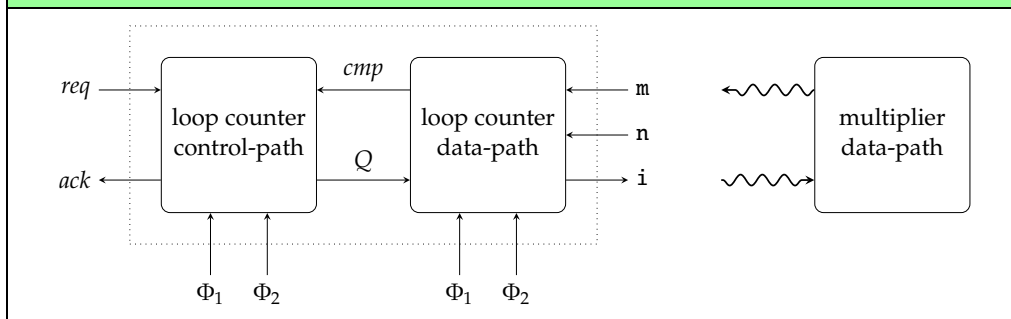
## Part 2.2: in practice, implementation (18)

Example #3: a loop counter

### ► 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.

Notes:

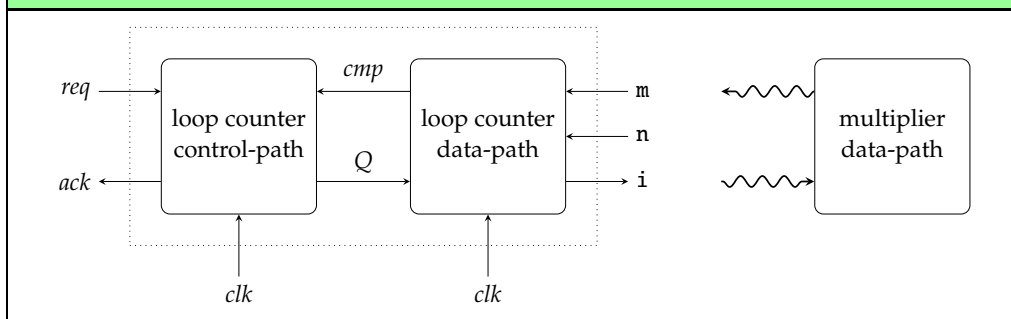
## Part 2.2: in practice, implementation (18)

Example #3: a loop counter

### ► 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.

Notes:



## Part 2.2: in practice, implementation (19)

Example #4: a traffic light controller

- **Problem:** design an FSM that controls two sets of UK-style traffic lights:
  - the traffic lights at the intersection is between a main road and an access road,
  - they should stop cars crashing into each other, displaying
    - green on main road and red on access road, then
    - amber on main road and red on access road, then
    - red on main road and amber on access road, then
    - red on main road and green on access road, then
    - red on main road and amber on access road, then
    - amber on main road and red on access road,
  - and then cycle, and
  - they should allow use of an emergency stop button, which
    - forces red on both main and access roads while pushed, then
    - reset the system into an initial start state when released.

Notes:

## Part 2.2: in practice, implementation (19)

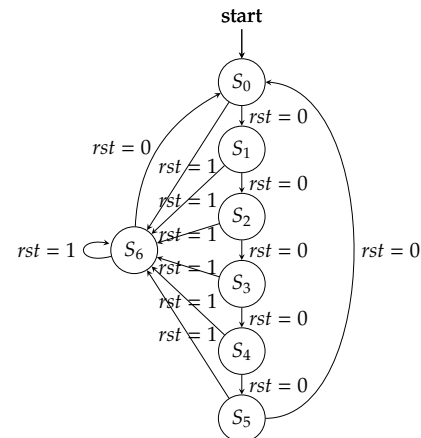
Example #4: a traffic light controller

- **Solution:**

### Algorithm (tabular)

Q	$\delta$		$\omega$					
	$Q'$		$M_g$	$M_a$	$M_r$	$A_g$	$A_a$	$A_r$
	$rst = 0$	$rst = 1$						
$S_0$	$S_1$	$S_6$	1	0	0	0	0	1
$S_1$	$S_2$	$S_6$	0	1	0	0	0	1
$S_2$	$S_3$	$S_6$	0	0	1	0	1	0
$S_3$	$S_4$	$S_6$	0	0	1	1	0	0
$S_4$	$S_5$	$S_6$	0	0	1	0	1	0
$S_5$	$S_0$	$S_6$	0	1	0	0	0	1
$S_6$	$S_0$	$S_6$	0	0	1	0	0	1

### Algorithm (diagram)



Notes:

## Part 2.2: in practice, implementation (19)

Example #4: a traffic light controller

### ► Solution:

- there are 7 abstract labels

$S_0$	$\mapsto$	0
$S_1$	$\mapsto$	1
$S_2$	$\mapsto$	2
$S_3$	$\mapsto$	3
$S_4$	$\mapsto$	4
$S_5$	$\mapsto$	5
$S_6$	$\mapsto$	6

we can represent using 7 concrete values, e.g.,

$S_0$	$\mapsto$	$\langle 0, 0, 0 \rangle$	$\equiv$	$000_{(2)}$
$S_1$	$\mapsto$	$\langle 1, 0, 0 \rangle$	$\equiv$	$001_{(2)}$
$S_2$	$\mapsto$	$\langle 0, 1, 0 \rangle$	$\equiv$	$010_{(2)}$
$S_3$	$\mapsto$	$\langle 1, 1, 0 \rangle$	$\equiv$	$011_{(2)}$
$S_4$	$\mapsto$	$\langle 0, 0, 1 \rangle$	$\equiv$	$100_{(2)}$
$S_5$	$\mapsto$	$\langle 1, 0, 1 \rangle$	$\equiv$	$101_{(2)}$
$S_6$	$\mapsto$	$\langle 0, 1, 1 \rangle$	$\equiv$	$110_{(2)}$

- since  $2^3 = 8 > 7$ , 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).

Notes:

## Part 2.2: in practice, implementation (19)

Example #4: a traffic light controller

### ► Solution:

- rewriting the abstract labels yields the following concrete truth table

				$\delta$			$\omega$					
$rst$	$Q_2$	$Q_1$	$Q_0$	$Q'_2$	$Q'_1$	$Q'_0$	$M_g$	$M_a$	$M_r$	$A_g$	$A_a$	$A_r$
0	0	0	0	0	0	1	1	0	0	0	0	1
0	0	0	1	0	1	0	0	1	0	0	0	1
0	0	1	0	0	1	1	0	0	1	0	1	0
0	0	1	1	1	0	0	0	0	1	1	0	0
0	1	0	0	1	0	1	0	0	1	0	1	0
0	1	0	1	0	0	0	0	1	0	0	0	1
0	1	1	0	0	0	0	0	0	1	0	0	1
0	1	1	1	?	?	?	?	?	?	?	?	?
1	0	0	0	1	1	0	1	0	0	0	0	1
1	0	0	1	1	1	0	0	1	0	0	0	1
1	0	1	0	1	1	0	0	0	1	0	1	0
1	0	1	1	1	1	0	0	0	1	1	0	0
1	1	0	0	1	1	0	0	0	1	0	1	0
1	1	0	1	1	1	0	0	1	0	0	0	1
1	1	1	0	1	1	0	0	0	1	0	0	1
1	1	1	1	?	?	?	?	?	?	?	?	?

Notes:

## Part 2.2: in practice, implementation (19)

Example #4: a traffic light controller

### ► Solution:

- the truth table can be translated into

Three Karnaugh maps for the traffic light controller. Each map has  $Q_0$  and  $Q_1$  as columns and  $Q_2$  and  $rst$  as rows. The maps show the following groupings:

- $Q_2'$ : 1s at (00,01), (01,01), (01,11), (11,11), (11,10), (10,11), (10,10). 0s at (00,00), (00,10), (01,00), (01,10), (11,00), (11,01), (10,00), (10,01).
- $Q_1'$ : 1s at (00,01), (01,01), (01,11), (11,11), (11,10), (10,11), (10,10). 0s at (00,00), (00,10), (01,00), (01,10), (11,00), (11,01), (10,00), (10,01).
- $Q_0'$ : 1s at (00,01), (01,01), (01,11), (11,11), (11,10), (10,11), (10,10). 0s at (00,00), (00,10), (01,00), (01,10), (11,00), (11,01), (10,00), (10,01).

- doing so yields the following Boolean expressions for  $\delta$ :

$$Q_2' = (rst \vee (Q_2 \wedge \neg Q_1 \wedge \neg Q_0) \vee (Q_1 \wedge Q_0))$$

$$Q_1' = (rst \vee (\neg Q_2 \wedge \neg Q_1 \wedge Q_0) \vee (\neg Q_2 \wedge Q_1 \wedge \neg Q_0))$$

$$Q_0' = (\neg rst \wedge \neg Q_2 \wedge \neg Q_1 \wedge \neg Q_0) \vee (\neg rst \wedge \neg Q_2 \wedge Q_1 \wedge \neg Q_0)$$

Notes:

## Part 2.2: in practice, implementation (19)

Example #4: a traffic light controller

### ► Solution:

- the truth table can be translated into

Six Karnaugh maps for the traffic light controller. Each map has  $Q_0$  and  $Q_1$  as columns and  $Q_2$  and  $rst$  as rows. The maps show the following groupings:

- $M_g$ : 1s at (00,01), (01,01), (01,11), (11,11), (11,10), (10,11), (10,10).
- $M_a$ : 1s at (00,01), (01,01), (01,11), (11,11), (11,10), (10,11), (10,10).
- $M_r$ : 1s at (00,01), (01,01), (01,11), (11,11), (11,10), (10,11), (10,10).
- $A_g$ : 1s at (00,01), (01,01), (01,11), (11,11), (11,10), (10,11), (10,10).
- $A_a$ : 1s at (00,01), (01,01), (01,11), (11,11), (11,10), (10,11), (10,10).
- $A_r$ : 1s at (00,01), (01,01), (01,11), (11,11), (11,10), (10,11), (10,10).

- doing so yields the following Boolean expressions for  $\delta$ :

$$M_g = (\neg Q_2 \wedge \neg Q_1 \wedge \neg Q_0) \vee (Q_2 \wedge \neg Q_1 \wedge \neg Q_0)$$

$$M_a = (\neg Q_2 \wedge \neg Q_1 \wedge Q_0) \vee (Q_2 \wedge \neg Q_1 \wedge Q_0)$$

$$M_r = (Q_2 \wedge \neg Q_1 \wedge \neg Q_0) \vee (Q_2 \wedge \neg Q_1 \wedge Q_0)$$

$$A_g = (Q_1 \wedge Q_0) \vee (\neg Q_2 \wedge \neg Q_1 \wedge \neg Q_0)$$

$$A_a = (Q_1 \wedge Q_0) \vee (\neg Q_2 \wedge \neg Q_1 \wedge \neg Q_0)$$

$$A_r = (Q_1 \wedge Q_0) \vee (\neg Q_2 \wedge \neg Q_1 \wedge \neg Q_0)$$

Notes:

## Conclusions

### ► Take away points:

1. FSMs are abstract computational models, but we can use 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.

Notes:

## Additional Reading

- *Wikipedia: Finite State Machine (FSM)*. URL: [https://en.wikipedia.org/wiki/Finite-state\\_machine](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.

Notes:

## References

- [1] [Wikipedia: Finite State Machine \(FSM\)](https://en.wikipedia.org/wiki/Finite-state_machine). URL: [https://en.wikipedia.org/wiki/Finite-state\\_machine](https://en.wikipedia.org/wiki/Finite-state_machine) (see p. 207).
- [2] D. Page. “Chapter 2: Basics of digital logic”. In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer, 2009 (see p. 207).
- [3] M. Sipster. “Chapter 1: Regular languages”. In: *Introduction to the Theory of Computation*. 2nd ed. Thomson Course Technology, 2006 (see p. 207).
- [4] M. Sipster. *Introduction to the Theory of Computation*. 2nd ed. Thomson Course Technology, 2006 (see pp. 5, 33–56).

Notes: