

Computer Architecture

Daniel Page

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

September 17, 2026

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:

- **Problem:** design a cyclic n -bit **counter**, i.e., a component whose output r steps through values

$$0, 1, \dots, 2^n - 1, 0, 1, \dots$$

but is otherwise uncontrolled (or “free running”).

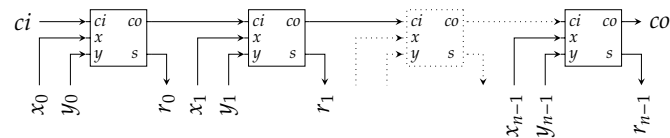
Notes:

- **Problem:** design a cyclic n -bit **counter**, i.e., a component whose output r steps through values

$$0, 1, \dots, 2^n - 1, 0, 1, \dots$$

but is otherwise uncontrolled (or “free running”).

- (Potential) **solution:** we already have an n -bit adder that can compute $x + y$, i.e.,



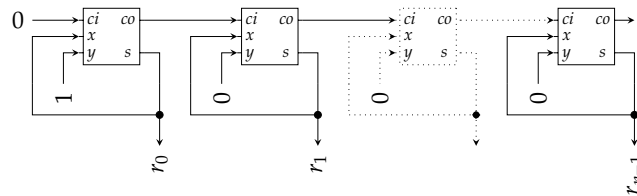
Notes:

- **Problem:** design a cyclic n -bit **counter**, i.e., a component whose output r steps through values

$$0, 1, \dots, 2^n - 1, 0, 1, \dots$$

but is otherwise uncontrolled (or “free running”).

- (Potential) **solution:** we already have an n -bit adder that can compute $x + y$, i.e.,



so we'll just have it compute $r \leftarrow r + 1$ over and over again.

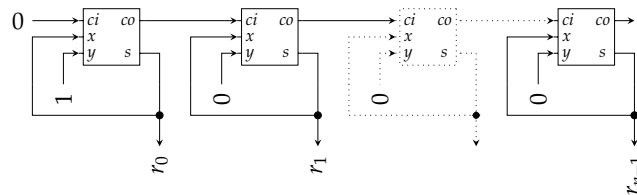
Notes:

- **Problem:** design a cyclic n -bit **counter**, i.e., a component whose output r steps through values

$$0, 1, \dots, 2^n - 1, 0, 1, \dots$$

but is otherwise uncontrolled (or “free running”).

- (Potential) **solution:** we already have an n -bit adder that can compute $x + y$, i.e.,



so we'll just have it compute $r \leftarrow r + 1$ over and over again;

- (New) **problem(s):** this **won't work**, because, for example,
1. we can't initialise the value,
 2. we can't control when a given output is produced,
 3. we don't let the output settle before it's used again as an input,
 4. ...

Notes:

- ▶ (Actual) **problem**: combinatorial logic has some limitations, namely we can't
 - ▶ control *when* a design computes some output (it does so continuously), nor
 - ▶ *remember* the output when produced.
- ▶ (Actual) **solution**, and so **agenda**: **sequential logic** design, where, crucially,
 - ▶ the output is a function of the input plus any state (e.g., stemming from previous inputs),
 - ▶ computation is viewed as being discrete, i.e., step-by-step,

via coverage of

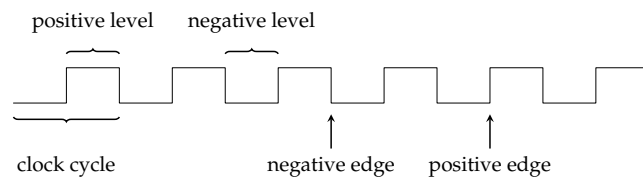
1. synchronisation of components $\leadsto$ clocks
2. components that maintain state $\leadsto$ latches, flip-flops, and registers
3. mechanism for computational steps $\leadsto$ structure plus strategy

Notes:

Part 1: clocks (1)

- ▶ **Concept**: a **clock** is a signal that oscillates (or alternates) between 1 and 0.

Definition



where

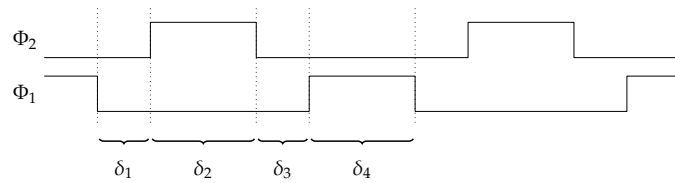
- ▶ the clock signal is typically either
 1. an input which needs to be supplied externally, or
 2. produced internally by a **clock generator**.
- ▶ we use features of the clock to
 1. trigger events (e.g., steps in some sequence of computations), and/or
 2. synchronise components.
- ▶ the **clock frequency** is how many clock cycles happen per-second; it must be
 - ▶ fast enough to satisfy the design goals, yet
 - ▶ slow enough to cope with the critical path of a given step.

Notes:

Part 1: clocks (2)

► **Concept:** an n -phase clock is distributed as n separate signals along n separate wires.

Definition



where for $n = 2$, for example,

- features in a 1-phase clock (e.g., cycle, levels and edges), generalise to Φ_1 and Φ_2 ,
- there is a guarantee that positive levels of Φ_1 and Φ_2 don't overlap, and
- the behaviour is parameterisable by altering δ_i .

Notes:

Part 2: latches, flip-flops, and registers (1)

Concepts

Definition

A bistable component can exist in two stable states, i.e., 0 or 1: at a given point, it can

- retain some **current state** Q (which can also be read as an output), *and*
- be updated to some **next state** Q' (which is provided as an input) under control of an **enable signal** en .

Definition

The behaviour of a bistable is described by an **excitation table**, and sometimes expressed using a **characteristic equation**: versus, e.g., a truth table, the idea is to capture the notion of time (cf. current and next).

Definition

A given bistable component controlled by an enable signal en can be

1. **level-triggered**, i.e., updated by a given level on en , or
2. **edge-triggered**, i.e., updated by a given edge on en .

The former type is termed a **latch**, whereas the latter type is termed a **flip-flop**.

Notes:

Part 2: latches, flip-flops, and registers (2)

Concepts

Definition

An “SR” latch/flip-flop component has two inputs S (or **set**) and R (or **reset**):

- ▶ when enabled, if
 - ▶ $S = 0$ and $R = 0$, the component retains Q ,
 - ▶ $S = 1$ and $R = 0$, the component updates to $Q = 1$,
 - ▶ $S = 0$ and $R = 1$, the component updates to $Q = 0$,
 - ▶ $S = 1$ and $R = 1$, the component is meta-stable
- but
- ▶ when not enabled, the component is in storage mode so retains Q .

The behaviour of such a component is specified by

$$Q' = S \vee (\neg R \wedge Q),$$

and/or

S	R	Current		Next	
		Q	$\neg Q$	Q'	$\neg Q'$
0	0	0	1	0	1
0	0	1	0	1	0
0	1	?	?	0	1
1	0	?	?	1	0
1	1	?	?	?	?

Definition

A “D” latch/flip-flop component has one input D :

- ▶ when enabled, if
 - ▶ $D = 1$, the component updates to $Q = 1$,
 - ▶ $D = 0$, the component updates to $Q = 0$,
- but
- ▶ when not enabled, the component is in storage mode so retains Q .

The behaviour of such a component is specified by

$$Q' = D,$$

and/or

D	Current		Next	
	Q	$\neg Q$	Q'	$\neg Q'$
0	?	?	0	1
1	?	?	1	0

Notes:

Part 2: latches, flip-flops, and registers (3)

Concepts

Definition

A “JK” latch/flip-flop component has two inputs J (or **set**) and K (or **reset**):

- ▶ when enabled, if
 - ▶ $J = 0$ and $K = 0$, the component retains Q ,
 - ▶ $J = 1$ and $K = 0$, the component updates to $Q = 1$,
 - ▶ $J = 0$ and $K = 1$, the component updates to $Q = 0$,
 - ▶ $J = 1$ and $K = 1$, the component toggles Q ,
- but
- ▶ when not enabled, the component is in storage mode so retains Q .

The behaviour of such a component is specified by

$$Q' = (J \wedge \neg Q) \vee (\neg K \wedge Q),$$

and/or

J	K	Current		Next	
		Q	$\neg Q$	Q'	$\neg Q'$
0	0	0	1	0	1
0	0	1	0	1	0
0	1	?	?	0	1
1	0	?	?	1	0
1	1	0	1	1	0
1	1	1	0	0	1

Definition

A “T” latch/flip-flop component has one input T :

- ▶ when enabled, if
 - ▶ $T = 0$, the component retains Q ,
 - ▶ $T = 1$, the component toggles Q ,
- but
- ▶ when not enabled, the component is in storage mode so retains Q .

The behaviour of such a component is specified by

$$Q' = (T \wedge \neg Q) \vee (\neg T \wedge Q) = T \oplus Q,$$

and/or

T	Current		Next	
	Q	$\neg Q$	Q'	$\neg Q'$
0	0	1	0	1
0	1	0	1	0
1	0	1	1	0
1	1	0	0	1

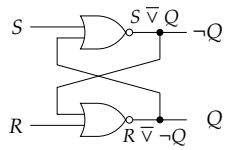
Notes:

Part 2: latches, flip-flops, and registers (4)

Design(s)

- **Problem #1:** we need an initial design for, e.g., an SR latch.
- **Solution:** use two *cross-coupled* NOR gates.

Circuit



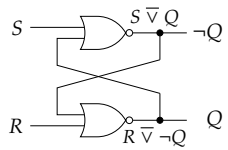
Notes:

Part 2: latches, flip-flops, and registers (4)

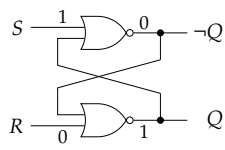
Design(s)

- **Problem #1:** we need an initial design for, e.g., an SR latch.
- **Solution:** use two *cross-coupled* NOR gates.

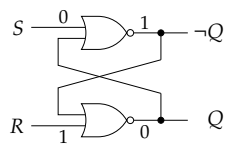
Circuit



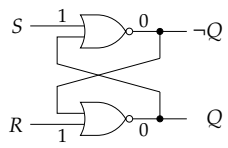
Example ($S = 1, R = 0, \checkmark$)



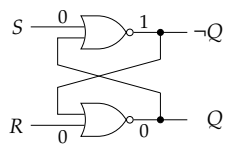
Example ($S = 0, R = 1, \checkmark$)



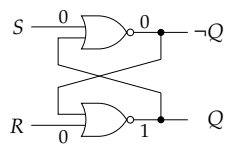
Example ($S = 1, R = 1, \times$)



Example ($S = 0, R = 0, \checkmark$)



Example ($S = 0, R = 0, \checkmark$)

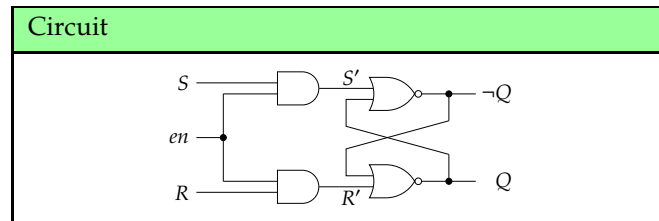


Notes:

Part 2: latches, flip-flops, and registers (5)

► **Problem #2:** we'd like to control when updates occur.

► **Solution:** *gate* S and R , i.e.,



noting that

- the same internal latch is evident, now with inputs S' and R' ,
- the external latch is such that

$$\begin{aligned} en = 0 &\Rightarrow S' = S \wedge en = S \wedge 0 = 0 \\ &\Rightarrow R' = R \wedge en = R \wedge 0 = 0 \end{aligned}$$

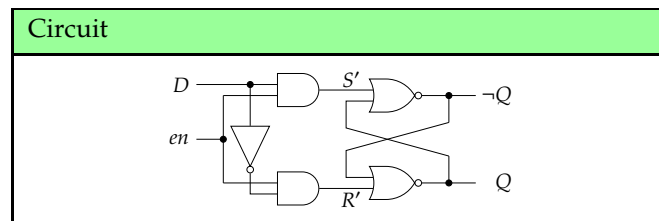
$$\begin{aligned} en = 1 &\Rightarrow S' = S \wedge en = S \wedge 1 = S \\ &\Rightarrow R' = R \wedge en = R \wedge 1 = R \end{aligned}$$

Notes:

Part 2: latches, flip-flops, and registers (6)

► **Problem #3:** we'd like to avoid the issue of meta-stability.

► **Solution:** force $R = \neg S$ so we get either $S = 0$ and $R = 1$, or $S = 1$ and $R = 0$.

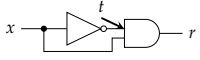
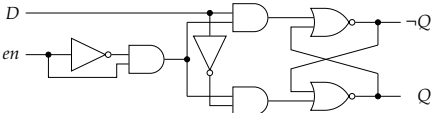
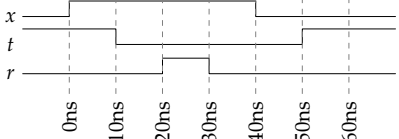


Notes:

Part 2: latches, flip-flops, and registers (7)

Design(s)

- **Problem #4:** we'd like an edge-triggered, rather than level-triggered design.
- **Solution #1:** "cheat" by approximation of an edge via a **pulse generator**.

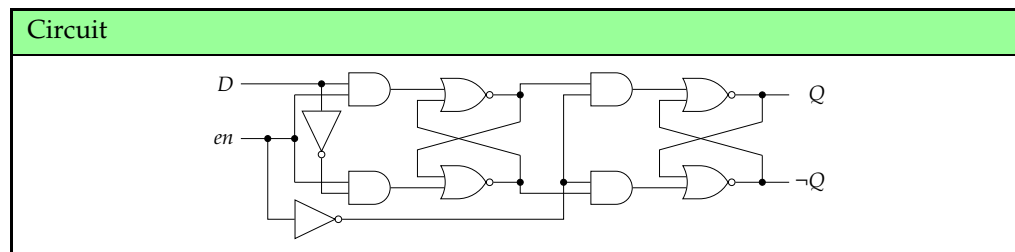
Circuit	Example
The pulse generator component  is attached to the previous SR latch to give: 	Imagine we set the delay of 1. a NOT gate to 10ns, and 2. an AND gate to 20ns and then flip $en = 0$ to $en = 1$ and back again:  The result is a "pulse" matching the delay of a NOT gate that approximates an edge because it is so short.

Notes:

Part 2: latches, flip-flops, and registers (8)

Design(s)

- **Problem #4:** we'd like an edge-triggered, rather than level-triggered design.
- **Solution #2:** adopt a **primary-secondary** organisation of *two* latches, i.e.,



where the idea is to split a clock cycle into two half-cycles:

$en = 1 \Rightarrow$ **primary** latch is enabled
 $en = 0 \Rightarrow$ **secondary** latch is enabled

meaning

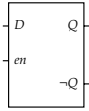
1. *while* $en = 1$, i.e., positive level on en , primary latch stores input, and
2. *when* $en = 0$, i.e., negative edge on en , secondary latch stores output of primary latch, and hence we get edge-triggered behaviour.

Notes:

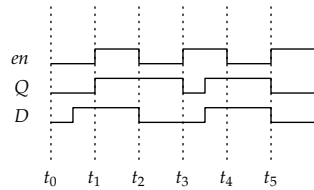
- Historically, the terms master and slave have often been used in place of primary and secondary. Per [8, Section 1.1], however, and despite some debate, the former are typically viewed as inappropriate now. We deliberately use the latter, therefore, noting that doing so may imply a need to translate the former when aligning with other literature.

Definition

- ▶ A **D-type latch** is described symbolically as



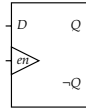
- ▶ The associated behaviour is, for example,



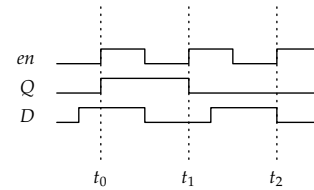
such that updates are level-triggered by *en*.

Definition

- ▶ A **D-type flip-flop** is described symbolically as



- ▶ The associated behaviour is, for example,



such that updates are edge-triggered by *en*.

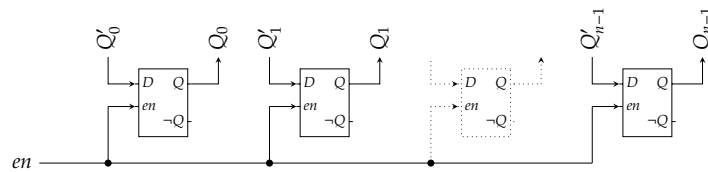
Notes:

Part 2: latches, flip-flops, and registers (10)

Aggregation, via registers

- ▶ **Concept:** we typically combine latches (resp. flip-flops) into **registers**, i.e.,

Circuit (latch version)

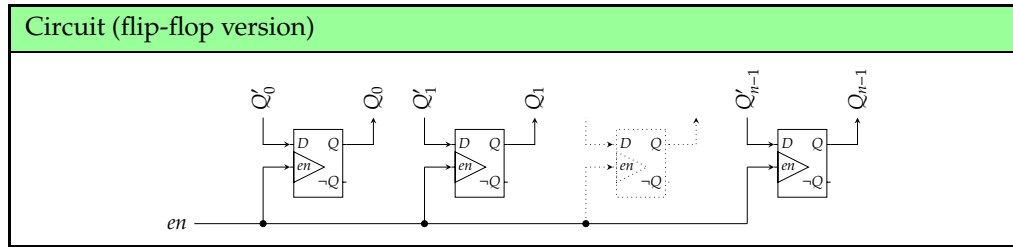


where

- ▶ there are n instances, so we can store an n -bit value: the i -th instance stores the i -th bit,
- ▶ access is conceptually straightforward:
 - read from register : use current value Q
 - write to register : drive next value onto Q' , then trigger update via en
- ▶ all instance share same en , so access to them is synchronised.

Notes:

- **Concept:** we typically combine latches (resp. flip-flops) into **registers**, i.e.,



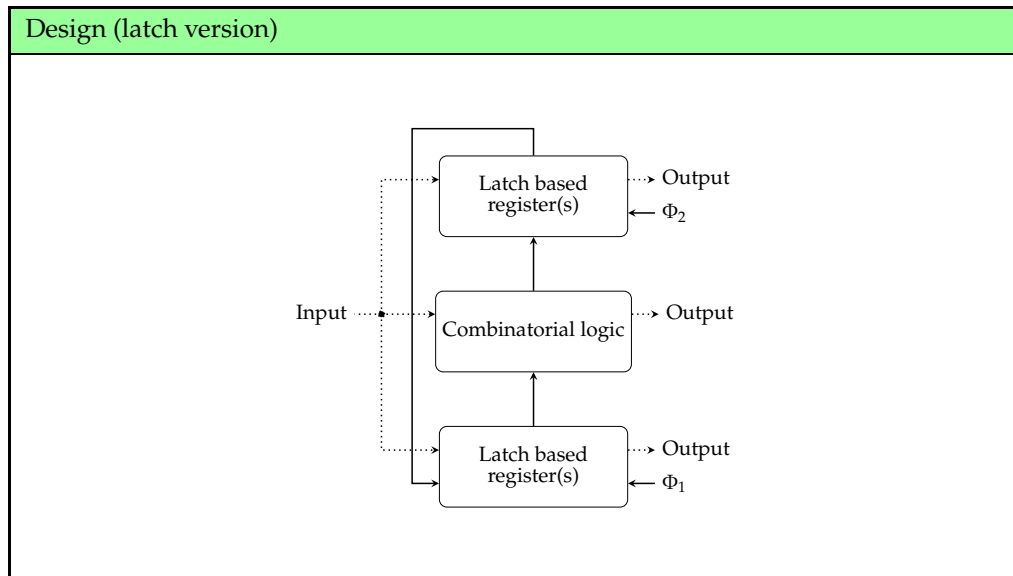
where

- there are n instances, so we can store an n -bit value: the i -th instance stores the i -th bit,
- access is conceptually straightforward:
 - read from register : use current value Q
 - write to register : drive next value onto Q' , then trigger update via en
- all instance share same en , so access to them is synchronised.

Notes:

Part 3: structure plus strategy (1)

Outcome: latch version, 2-phase clock

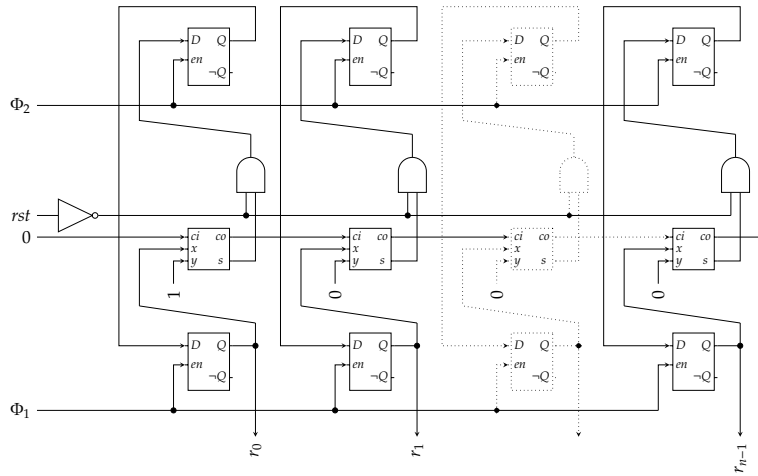


Notes:

Part 3: structure plus strategy (2)

Outcome: latch version, 2-phase clock

Circuit (latch version)

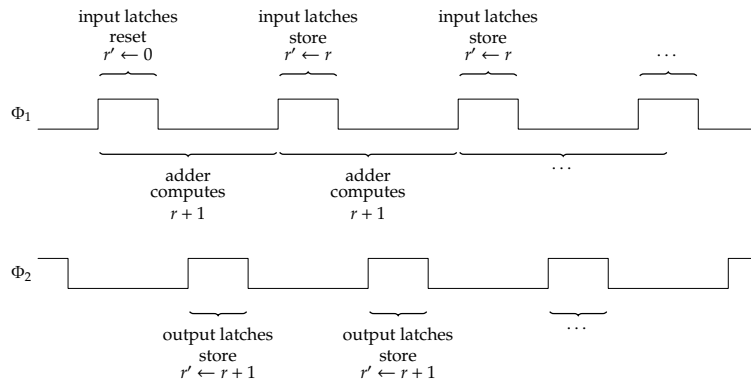


Notes:

Part 3: structure plus strategy (3)

Outcome: latch version, 2-phase clock

Example (latch version)

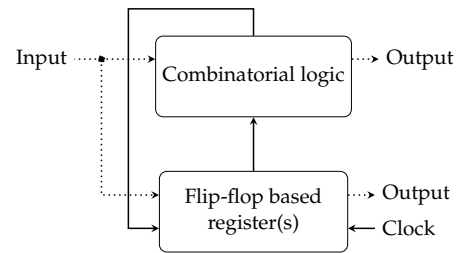


Notes:

Part 3: structure plus strategy (4)

Outcome: flip-flop version, 1-phase clock

Design (flip-flop version)

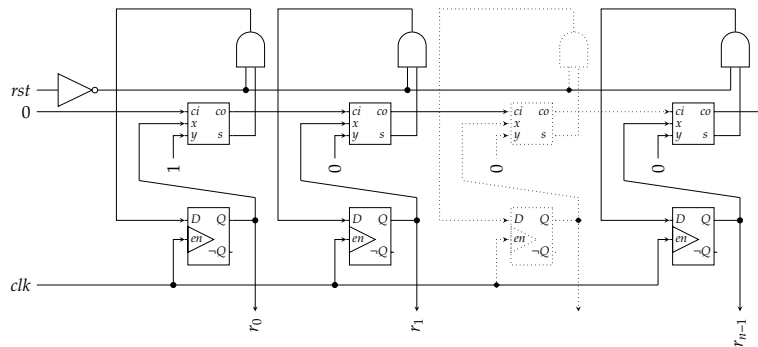


Notes:

Part 3: structure plus strategy (5)

Outcome: flip-flop version, 1-phase clock

Circuit (flip-flop version)

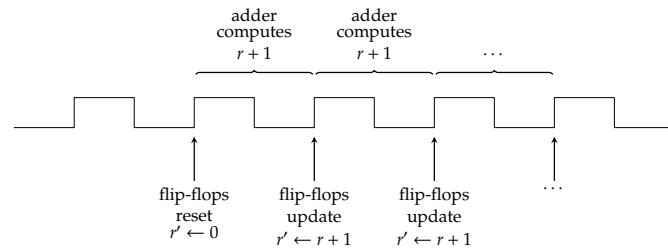


Notes:

Part 3: structure plus strategy (6)

Outcome: flip-flop version, 1-phase clock

Example (flip-flop version)



Notes:

Conclusions

► Take away points:

1. Sequential logic design is typically hard(er) to understand (at first) than combinatorial logic design: invest some effort to address this now!
2. The main concept and challenge is *time*:
 - this goes beyond time in the sense of delay,
 - the goal is step-by-step, controlled (versus continuous, uncontrolled) computation,
 - we need to understand and manage, e.g., with parallelism and synchronisation.

Notes:

Conclusions

► Take away points:

3. There is at least one higher-level design principle evident: we often see
 - 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, although the counter control-path is (very) simple.
4. The next step is to formalise this, allowing solution of more complex problems, e.g., through more complex forms of control.

Notes:

Additional Reading

- *Wikipedia: Sequential logic*. URL: https://en.wikipedia.org/wiki/Sequential_logic.
- D. Page. “Chapter 2: Basics of digital logic”. In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer, 2009.
- W. Stallings. “Chapter 11: Digital logic”. In: *Computer Organisation and Architecture*. 9th ed. Prentice Hall, 2013.
- A.S. Tanenbaum and T. Austin. “Section 3.2.2: Clocks”. In: *Structured Computer Organisation*. 6th ed. Prentice Hall, 2012.
- A.S. Tanenbaum and T. Austin. “Section 3.3.4: Latches”. In: *Structured Computer Organisation*. 6th ed. Prentice Hall, 2012.
- A.S. Tanenbaum and T. Austin. “Section 3.3.4: Flip-flops”. In: *Structured Computer Organisation*. 6th ed. Prentice Hall, 2012.
- A.S. Tanenbaum and T. Austin. “Section 3.3.4: Registers”. In: *Structured Computer Organisation*. 6th ed. Prentice Hall, 2012.

Notes:

References

- [1] [Wikipedia: Sequential logic](#). URL: https://en.wikipedia.org/wiki/Sequential_logic (see p. 59).
- [2] D. Page. “Chapter 2: Basics of digital logic”. In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer, 2009 (see p. 59).
- [3] W. Stallings. “Chapter 11: Digital logic”. In: *Computer Organisation and Architecture*. 9th ed. Prentice Hall, 2013 (see p. 59).
- [4] A.S. Tanenbaum and T. Austin. “Section 3.2.2: Clocks”. In: *Structured Computer Organisation*. 6th ed. Prentice Hall, 2012 (see p. 59).
- [5] A.S. Tanenbaum and T. Austin. “Section 3.3.4: Flip-flops”. In: *Structured Computer Organisation*. 6th ed. Prentice Hall, 2012 (see p. 59).
- [6] A.S. Tanenbaum and T. Austin. “Section 3.3.4: Latches”. In: *Structured Computer Organisation*. 6th ed. Prentice Hall, 2012 (see p. 59).
- [7] A.S. Tanenbaum and T. Austin. “Section 3.3.4: Registers”. In: *Structured Computer Organisation*. 6th ed. Prentice Hall, 2012 (see p. 59).
- [8] M. Knodel and N. ten Oever. *Terminology, Power and Oppressive Language*. Internet Engineering Task Force (IETF) Internet Draft. 2018. URL: <https://tools.ietf.org/id/draft-knodel-terminology-00.html> (see p. 36).

Notes: