Intro. to Computer Architecture

Daniel Page

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

January 9, 2018

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 explaination 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:

- Question: imagine we need a cyclic n-bit **counter**, i.e., a component whose output r steps through values

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

but is otherwise uncontrolled (or "free running").

Notes:

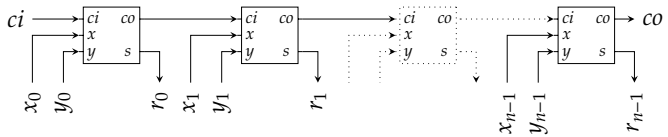
COMS12200 lecture: week #5

- Question: imagine we need a cyclic n-bit **counter**, i.e., a component whose output r steps through values

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

but is otherwise uncontrolled (or "free running").

- Solution (?): we already have an n-bit adder that can compute $x + y$ ...



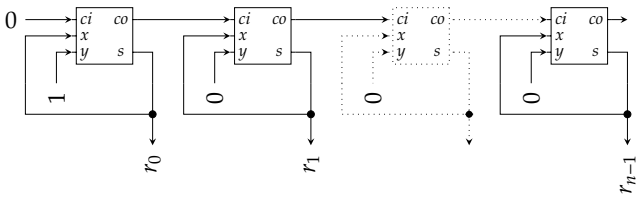
© Daniel Page ⟨csdsp@bristol.ac.uk⟩
Intro. to Computer Architecture git # 3b6f641 @ 2018-01-09 University of BRISTOL

Notes:

- Question: imagine we need a cyclic n-bit **counter**, i.e., a component whose output r steps through values

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

  but is otherwise uncontrolled (or "free running").
- Solution (?): we already have an n-bit adder that can compute $x + y$ ...



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

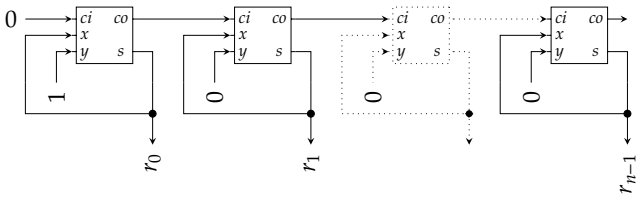
Notes:

COMS12200 lecture: week #5

- Question: imagine we need a cyclic n-bit **counter**, i.e., a component whose output r steps through values

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

  but is otherwise uncontrolled (or "free running").
- Solution (?): we already have an n-bit adder that can compute $x + y$ ...



... so we'll just compute $r \leftarrow r + 1$ over and over again; this *sounds* like a good idea, but **won't work** due to (at least) two flaws:

1. we can't initialise the value, and
2. we don't let the output of each full-adder settle before it's used again as an input.

Notes:

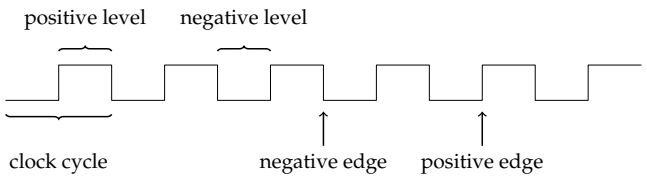
- *Actual* problem: combinatorial logic has some limitations, namely we can't
  - control *when* a design computes some output (it does so continuously), nor
  - *remember* the output when produced.
- *Actual* solution: **sequential** logic, which demands
  1. some way to control (e.g., synchronise) components,
  2. one or more components that remember what state they are in, and
  3. a mechanism to perform computation as a sequence of steps, rather than continuously.

Notes:

Clocks (1)

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

Definition



Notes:

Clocks (1)

- Idea:
  - The clock **triggers** events (e.g., steps in some sequence of computations) and/or synchronise components within a design.
  - The **clock frequency** is how many clock cycles happen per-second; it has to be
    - fast enough to to satisfy the design goals, yet
    - slow enough to cope with the critical path of a given step

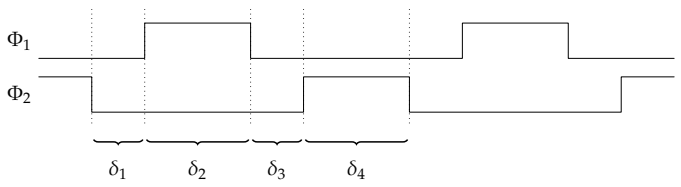
    i.e., the faster the clock "ticks" the faster we step though the computation, but if it's too fast we can't finish one step before the next one starts!

Notes:

Clocks (2)

- A n**-phase clock** is distributed as n separate signals along n separate wires

Definition



with a 2-phase instance particularly useful:

1. features in a 1-phase clock, e.g., the clock period, levels and edges, translate naturally to both Φ_1 and Φ_2,
2. there is a guarantee that positive levels of Φ_1 and Φ_2 don't overlap, and
3. the behaviour is parameterisable by altering δ_i.

Notes:

Clocks (3)

- A clock signal is typically
  1. an input which needs to be supplied externally, or
  2. produced internally by a **clock generator**

  then distributed by a **clock network** (e.g., a H-tree).
- It can be attractive to multiply or divide the clock (i.e., make it faster or slower):

Algorithm (Clock-Divide)

To divide (i.e., slow down) a reference clock *clk*:

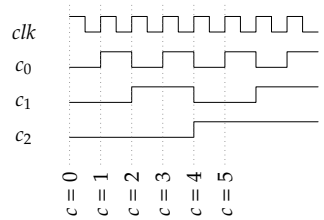
1. initialise a counter c to zero,
2. on each positive edge of *clk*, increment the counter c, then
3. the $(i-1)$-th bit of the counter c acts like the clock divided by 2^i.

So if we let $i = 1$, we get a to get a clock which is

$$\frac{1}{2^i} = \frac{1}{2^1} = \frac{1}{2}$$

the speed (i.e., twice as slow) by looking at the 0-th bit of c.

Example



Notes:

Latches and Flip-Flops (1)

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*. We say it is

- **level-triggered**, and hence a **latch**, if updated by a given level on *en*, or
- **edge-triggered**, and hence a **flip-flop**, if updated by a given edge on *en*.

Notes:

Latches and Flip-Flops (2)

Definition

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

- when enabled and
  - $S = 0, R = 0$ the component retains Q,
  - $S = 1, R = 0$ the component updates to $Q = 1$,
  - $S = 0, R = 1$ the component updates to $Q = 0$,
  - $S = 1, R = 1$ the component is meta-stable,

  but
- when not enabled, the component is in storage mode and retains Q.

The behaviour of the component is described by the truth table

SR-Latch/SR-FlipFlop					
		Current		Next	
S	R	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 and
  - $D = 1$ the component updates to $Q = 1$,
  - $D = 0$ the component updates to $Q = 0$,

  but
- when not enabled, the component is in storage mode and retains Q.

The behaviour of the component is described by the truth table

D-Latch/D-FlipFlop				
	Current		Next	
D	Q	$\neg Q$	Q'	$\neg Q'$
0	**?**	**?**	0	1
1	**?**	**?**	1	0

Notes:

Latches and Flip-Flops (3)

Definition

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

- when enabled and
  - $J = 0, K = 0$ the component retains Q,
  - $J = 1, K = 0$ the component updates to $Q = 1$,
  - $J = 0, K = 1$ the component updates to $Q = 0$,
  - $J = 1, K = 1$ the component toggles Q,

  but
- when not enabled, the component is in storage mode and retains Q.

The behaviour of the component is described by the truth table

JK-Latch/JK-FlipFlop					
		Current		Next	
J	K	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 and
  - $T = 0$ the component retains Q,
  - $T = 1$ the component toggles Q,

  but
- when not enabled, the component is in storage mode and retains Q.

The behaviour of the component is described by the truth table

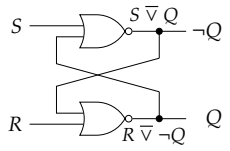
T-Latch/T-FlipFlop				
	Current		Next	
T	Q	$\neg Q$	Q'	$\neg Q'$
0	0	1	0	1
0	1	0	1	0
1	0	1	1	0
1	1	0	0	1

Notes:

Latches and Flip-Flops (4)

- Problem #1: how can we design a simple SR latch?
- Solution: use two *cross-coupled* NOR gates.

Circuit

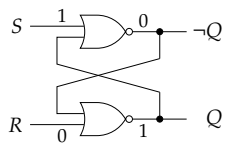


Notes:

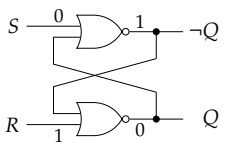
Latches and Flip-Flops (4)

- Problem #1: how can we design a simple SR latch?
- Solution: use two *cross-coupled* NOR gates.

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



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

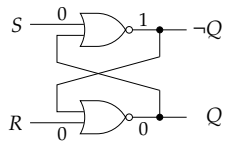


Notes:

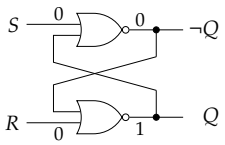
Latches and Flip-Flops (4)

- Problem #1: how can we design a simple SR latch?
- Solution: use two *cross-coupled* NOR gates.

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



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

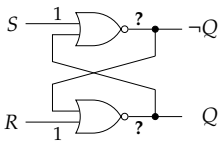


Notes:

Latches and Flip-Flops (4)

- Problem #1: how can we design a simple SR latch?
- Solution: use two *cross-coupled* NOR gates.

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

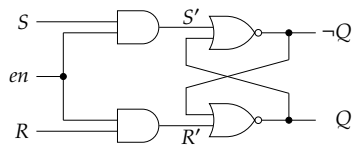


Notes:

Latches and Flip-Flops (5)

- Problem #2: we'd like to control when updates occur.
- Solution: **gate** S and R, controlling whether they act as input to the internal latch.

Circuit



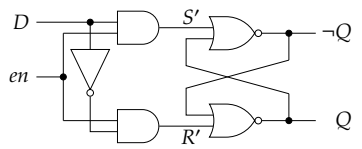
- Note that:
  - The new design is clearly level triggered in the sense S and R are only relevant during a positive level of en, e.g.,
    - if $en = 0$, $S' = S \wedge en = S \wedge 0 = 0$, whereas
    - if $en = 1$, $S' = S \wedge en = S \wedge 1 = S$.
  - If we "gate" a signal more generally, we mean "conditionally turn it off".

Notes:

Latches and Flip-Flops (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 $R = 0$.

Circuit



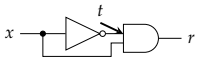
Notes:

Latches and Flip-Flops (7)

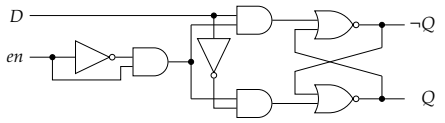
- Problem #4: the level triggered latch doesn't give very fine grained control over when it is enabled since levels can be quite long.
- Solution #1: "cheat" by constructing a **pulse generator** to approximate the idea of edge triggered control.

Circuit

The pulse generator component



is attached to the previous SR latch to give:

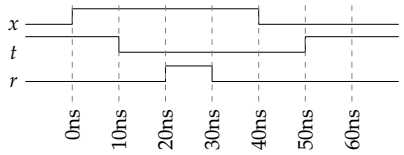


Example

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:



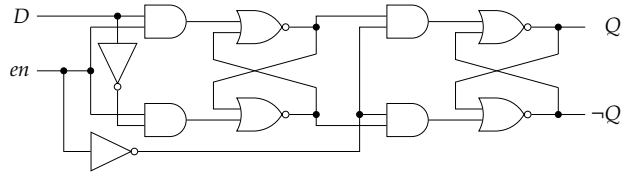
The result is a "pulse" matching the delay of a NOT gate that approximates an edge because it is so short.

Notes:

Latches and Flip-Flops (8)

- Problem #4: the level triggered latch doesn't give very fine grained control over when it is enabled since levels can be quite long.
- Solution #2: adopt a **master-slave** arrangement of two latches

Circuit



where the idea is to split a clock cycle into to half-cycles st.

1. while $en = 1$, i.e., during the first half-cycle, the **master** latch is enabled,
2. while $en = 0$, i.e., during the second half-cycle, the **slave** latch is enabled

meaning

- while $en = 1$, i.e., during a positive level on en, the master latches the input, then
- exactly as $en = 0$, i.e., a negative edge on en, the slave latches the output from the master

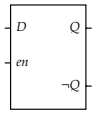
and hence we get an edge triggered component.

Notes:

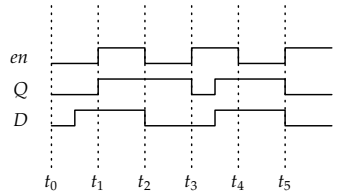
Latches and Flip-Flops (9)

Definition

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



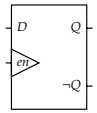
- The associated behaviour is, for example,



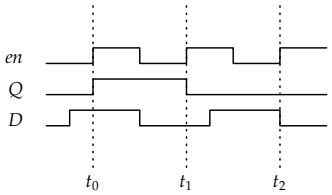
st. updates are level-triggered by *en*.

Definition

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



- The associated behaviour is, for example,



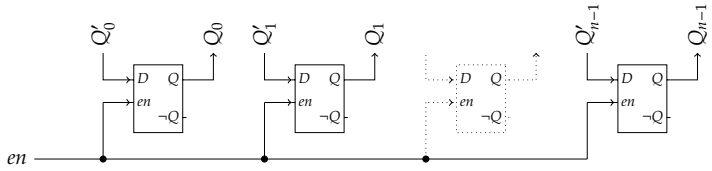
st. updates are edge-triggered by *en*.

Notes:

Latches and Flip-Flops (10)

- We typically group such components into **registers**, st. an n-bit register can then store an n-bit value:

Circuit (latch version)



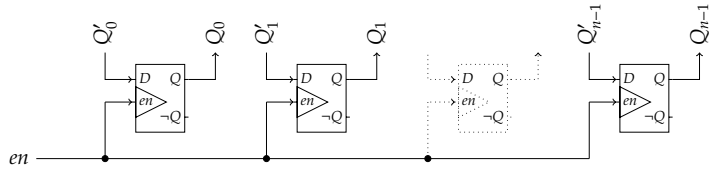
- Note that:
  - Q_i is the i-th bit, for $0 \leq i < n$, of the current value Q stored by the register.
  - To **latch** (or store) a some next value Q', we drive Q'_i onto D_i and wait for *en* to trigger an update.
  - Each component shares a common enable signal, so any such update is therefore synchronised across the whole register.

Notes:

Latches and Flip-Flops (10)

- We typically group such components into **registers**, st. an n-bit register can then store an n-bit value:

Circuit (flip-flop version)



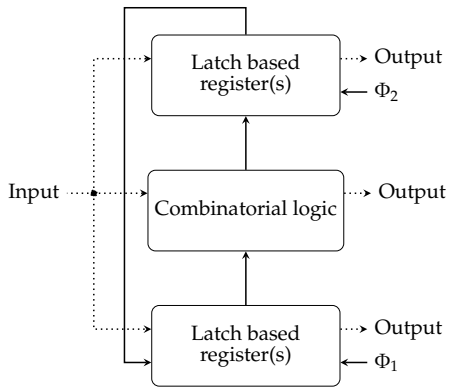
- Note that:
  - Q_i is the i-th bit, for $0 \leq i < n$, of the current value Q stored by the register.
  - To **latch** (or store) a some next value Q', we drive Q'_i onto D_i and wait for en to trigger an update.
  - Each component shares a common enable signal, so any such update is therefore synchronised across the whole register.

Notes:

Conclusions

- Now we can design a robust solution to the original problem:

Circuit (latch version)



that you can view this as a (fairly simplistic) combination of

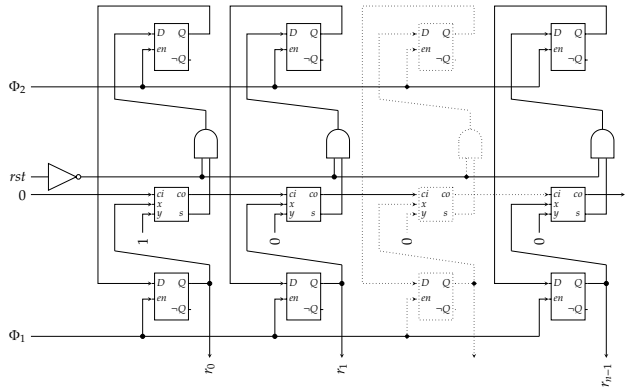
1. a **data-path**, of computational and/or storage components, and
2. a **control-path**, that tells components in the data-path what to do and when to do it.

Notes:

Conclusions

- Now we can design a robust solution to the original problem:

Circuit (latch version)



that you can view this as a (fairly simplistic) combination of

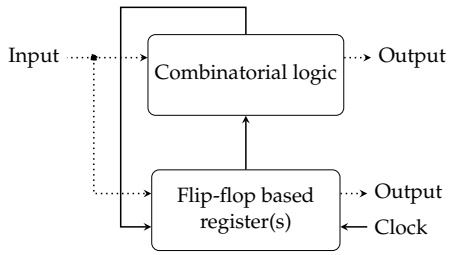
1. a **data-path**, of computational and/or storage components, and
2. a **control-path**, that tells components in the data-path what to do and when to do it.

Notes:

Conclusions

- Now we can design a robust solution to the original problem:

Circuit (flip-flop version)



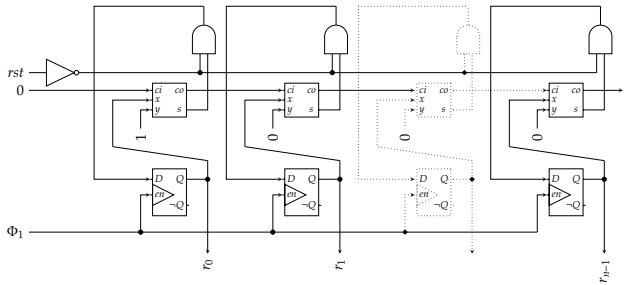
that you can view this as a (fairly simplistic) combination of

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

Notes:

- Now we can design a robust solution to the original problem:

Circuit (flip-flop version)

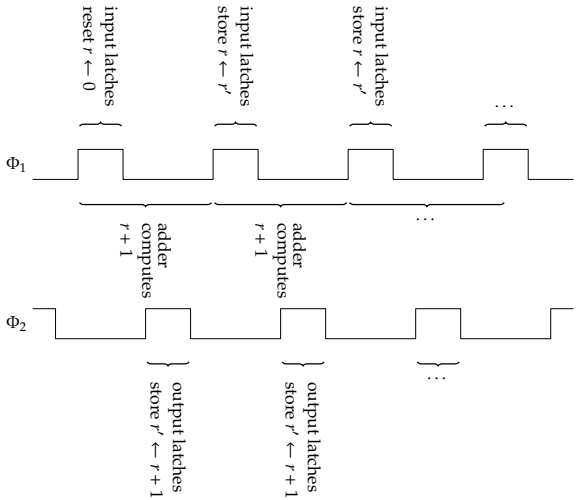


that you can view this as a (fairly simplistic) combination of

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

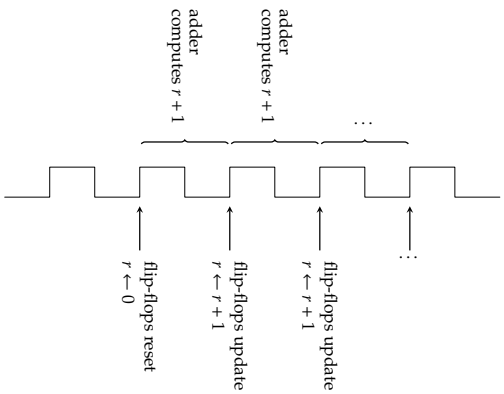
Notes:

Example (latch version)



Notes:

Example (flip-flop version)



Notes:

Conclusions

- Take away points:
  1. Sequential logic design is typically hard(er) to understand at first: spend the effort to do so, focusing on the concepts vs. the detail.
  2. The main such concept is that of *time*: not due to delay, but introduction of step-by-step, controlled (vs. continuous, uncontrolled) computation.
  3. The control-path in our counter is simple; the next step is to develop a design framework, allowing solution of more complex problems through more complex forms of control.

Notes:

Additional Reading

- *Wikipedia: Sequential logic*. URL: http://en.wikipedia.org/wiki/Sequential_logic.
- D. Page. "Chapter 2: Basics of digital logic". In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer-Verlag, 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: http://en.wikipedia.org/wiki/Sequential_logic (see p. 65).

[2] D. Page. "Chapter 2: Basics of digital logic". In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer-Verlag, 2009 (see p. 65).

[3] W. Stallings. "Chapter 11: Digital logic". In: *Computer Organisation and Architecture*. 9th ed. Prentice-Hall, 2013 (see p. 65).

[4] A.S. Tanenbaum and T. Austin. "Section 3.2.2: Clocks". In: *Structured Computer Organisation*. 6th ed. Prentice-Hall, 2012 (see p. 65).

[5] A.S. Tanenbaum and T. Austin. "Section 3.3.4: Flip-flops". In: *Structured Computer Organisation*. 6th ed. Prentice-Hall, 2012 (see p. 65).

[6] A.S. Tanenbaum and T. Austin. "Section 3.3.4: Latches". In: *Structured Computer Organisation*. 6th ed. Prentice-Hall, 2012 (see p. 65).

[7] A.S. Tanenbaum and T. Austin. "Section 3.3.4: Registers". In: *Structured Computer Organisation*. 6th ed. Prentice-Hall, 2012 (see p. 65).

Notes: