- 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:

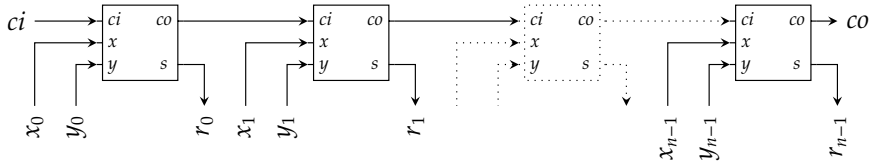
COMS12200 lecture: week #5

- 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 $x + y$ ...



Notes:

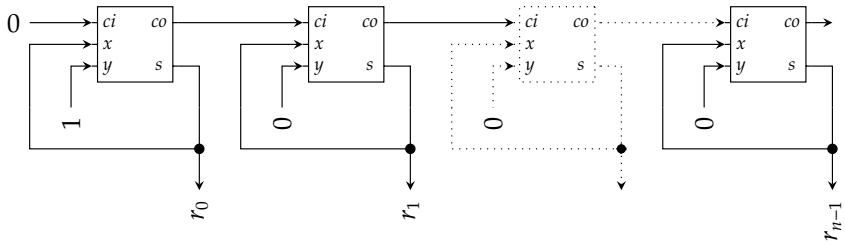
Dan Page ( ) © 2014-5
Computer Architecture (new format) Slide 1 of 24 University of BRISTOL

- 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:

COMS12200 lecture: week #5

- *Actual* problem: combinatorial logic has some limitations, namely we can't
  - control *when* a circuit computes some output (it does so continuously), nor
  - *remember* the output when produced.
- *Actual* solution: **sequential** logic, which demands

1. some way to synchronise components in the circuit,
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:

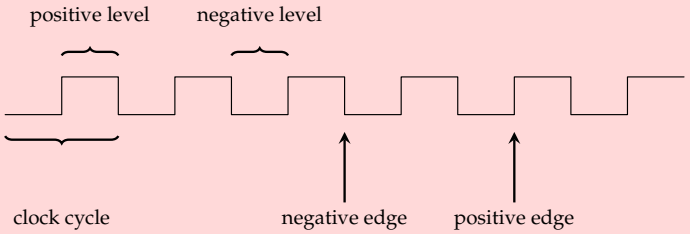
- Components that remember state can be classified as being either
  1. an **astable**, where the component is not stable in either state and flips uncontrolled between states,
  2. a **monostable**, where the component is stable in one states and flips uncontrolled but periodically between states, or
  3. a **bistable**, where the component is stable in two states and flips between states under control of an input,

Notes:

Clocks (1)

- A **clock** is a signal that oscillates (or alternates) between 1 and 0; we feed it into circuits to synchronise components in them:

Definition (1-phase clock)



- Idea:
  - The clock **triggers** events, e.g., steps in some sequence of computations.
  - 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

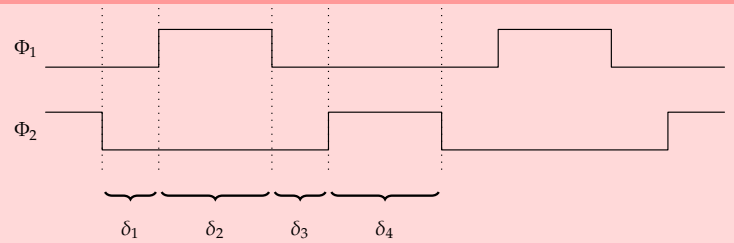
  i.e., the faster the clock "ticks" the faster we step though the computation, but if it's too fast we can't finish one step before the next one starts!

Notes:

Clocks (2)

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

Definition (2-phase clock)



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*:

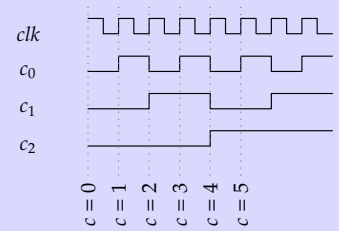
1. initialise a counter c to zero,
2. on each positive edge of *clk*, increment the counter c, then
3. the $(i-1)$-th bit of the counter c acts like the clock divided by 2^i.

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:

- The clock generator is typically a peizoelectric crystal or similar.
- The clock network is a bit like the need to distribute power (via power rails); it's power hungry!

Latches and Flip-Flops (1)

- We need a component which remembers what state it is in; that is, it
  - retains some **current state** Q (which can also be read as an output), and
  - can be updated to some **next state** Q' (which is provided as an input)

  essentially meaning it stores a 1-bit value of our choice.

Definition (latch versus flip-flop)

Given a suitable bistable component controlled using an **enable signal** *en* that determines when updates happen, we say it 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 typically termed a **latch**, with the latter termed a **flip-flop**.

Definition (active-high versus active-low)

Whether a positive or negative level (resp. edge) of some signal controls the component depends on whether it is **active high** or **active low**; a signal *en* is often written $\neg en$ to denote the latter case.

Notes:

- Latches are sometimes described as **transparent**: this term refers to the fact that while enabled, their input and output will match because the state (which matches the output) is being updated with the input. Flip-flops are not the same, because their state is only updated at the exact instant of an edge.

Latches and Flip-Flops (2)

Definition (SR-type latch/flip-flop)

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 (D-type latch/flip-flop)

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:

Definition (JK-type latch/flip-flop)

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 (T-type latch/flip-flop)

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

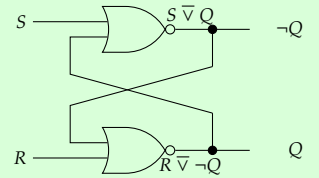
T-Latch/T-FlipFlop				
	Current		*Next*	
T	Q	$\neg Q$	Q'	$\neg Q'$
0	0	1	0	1
0	1	0	1	0
1	0	1	1	0
1	1	0	0	1

Notes:

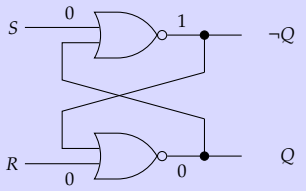
Latches and Flip-Flops (4)

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

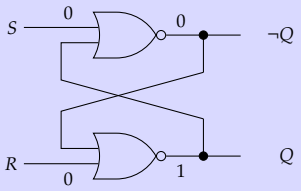
Circuit



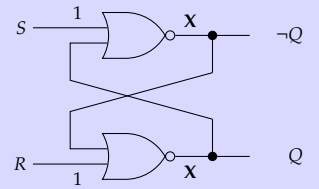
Example ($S = 0, R = 0$, valid)



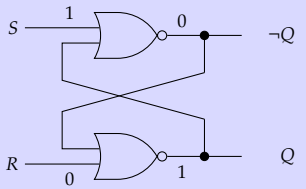
Example ($S = 0, R = 0$, valid)



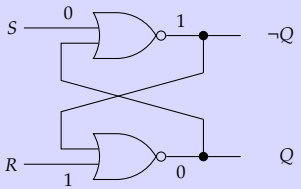
Example ($S = 1, R = 1$, invalid)



Example ($S = 1, R = 0$, valid)



Example ($S = 0, R = 1$, valid)

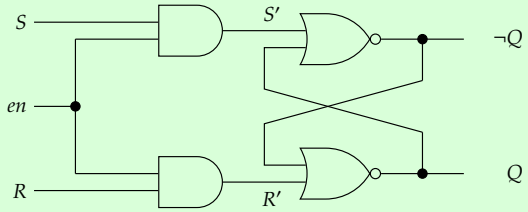


Notes:

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



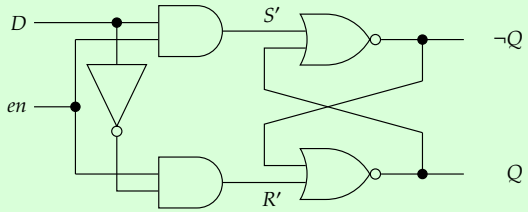
- Note that:
  - The new design is clearly level triggered in the sense S and R are only relevant during a positive level of en, e.g.,
    - if $en = 0$, $S' = S \wedge en = S \wedge 0 = 0$, whereas
    - if $en = 1$, $S' = S \wedge en = S \wedge 1 = S$.
  - If we "gate" a signal more generally, we mean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

Circuit



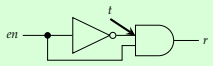
- Note that:
  - It might *look* like we've *lost* control in that we only have one input D, so cannot retain the current state (i.e., set $S = 0$ and $R = 0$) ...
  - ... but remember:
    1. when $en = 0$ the latch retains Q, and
    2. when $en = 1$ the latch updates Q with D.

Notes:

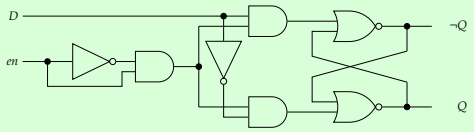
- Problem #4: the level triggered latch doesn't give very fine grained control over when it is enabled since levels can be quite long.
- Solution #1: "cheat" a bit and construct a **pulse generator** to approximate the idea of edge triggered control.

Circuit

The pulse generator component



is attached to the previous SR latch to give:

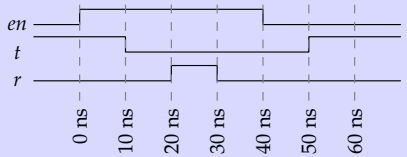


Example

Imagine we set the delay of

1. a NOT gate to 10 ns, and
2. an AND gate to 20 ns

and then flip $en = 0$ to $en = 1$ and back again:

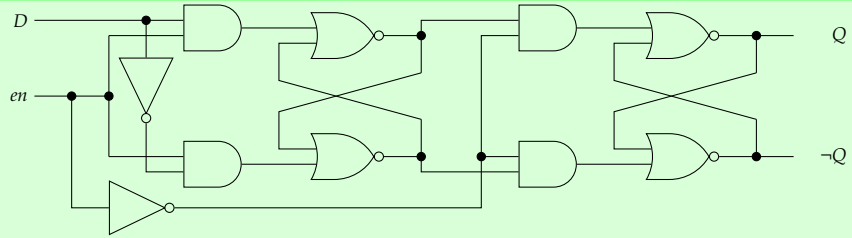


The result is a "pulse" matching the delay of a NOT gate that approximates an edge because it is so short.

Notes:

- 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

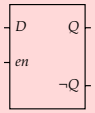
and hence we get an edge triggered component.

Notes:

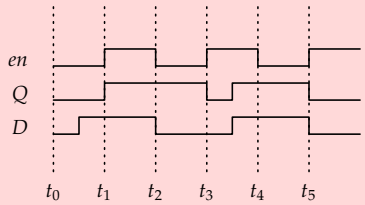
Latches and Flip-Flops (9)

Definition (D-type latch)

- Written symbolically as



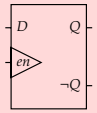
- Behaviour is, for example, as follows



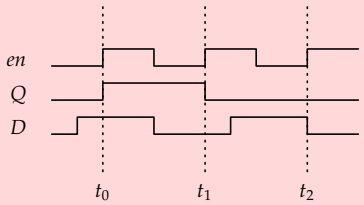
st. updates are level-triggered by *en*.

Definition (D-type flip-flop)

- Written symbolically as



- Behaviour is, for example, as follows



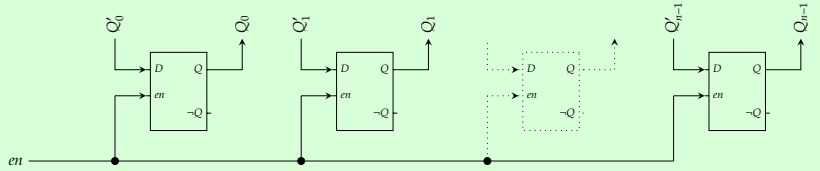
st. updates are edge-triggered by *en*.

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

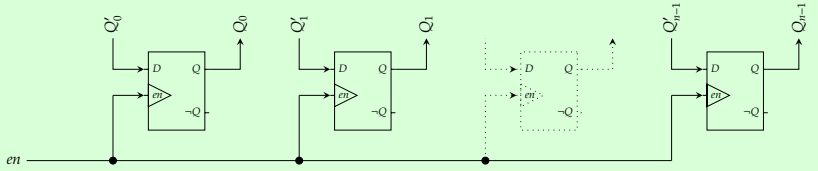


- Note that:
  - Q_i is the i-th bit, for $0 \leq i < n$, of the current value Q stored by the register.
  - To **latch** (or store) a some next value Q', we drive Q'_i onto D_i and wait for *en* to trigger an update.
  - Each component shares a common enable signal, so any such update is therefore synchronised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

Notes:

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

Circuit



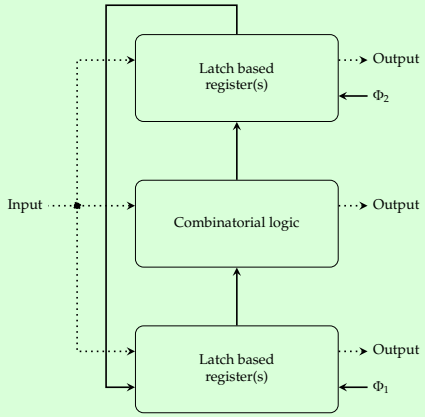
- Note that:
  - Q_i is the i-th bit, for $0 \leq i < n$, of the current value Q stored by the register.
  - To **latch** (or store) a some next value Q', we drive Q'_i onto D_i and wait for en to trigger an update.
  - Each component shares a common enable signal, so any such update is therefore synchronised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

Notes:

Conclusions

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

Circuit



that you can (roughly) view this as a 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,

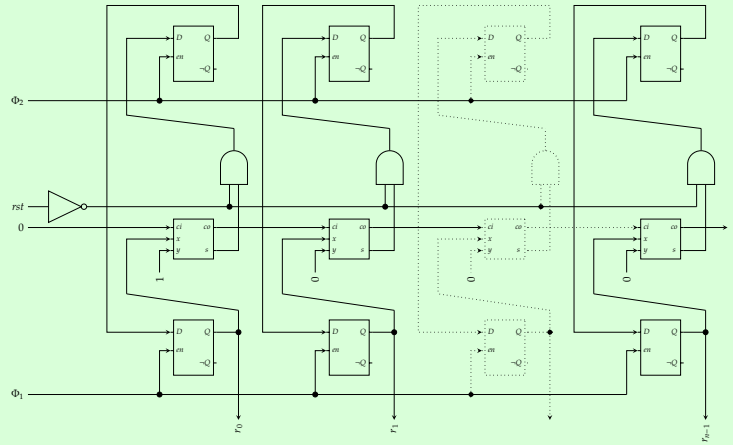
although here, the control-path is *very* simple ...

Notes:

Conclusions

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

Circuit



that you can (roughly) view this as a 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,

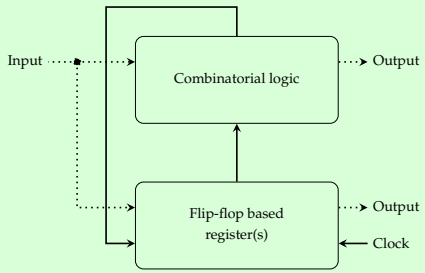
although here, the control-path is *very* simple ...

Notes:

Conclusions

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

Circuit



that you can (roughly) view this as a combination of

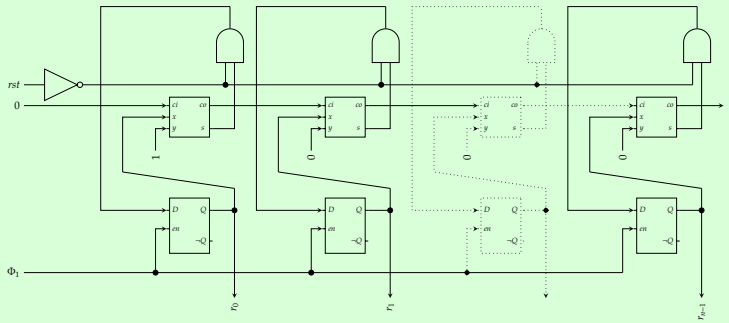
1. a **data-path**, of computational and/or storage components, and
2. a **control-path**, that tells components in the data-path what to do and when to do it.

although here, the control-path is *very* simple ...

Notes:

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

Circuit



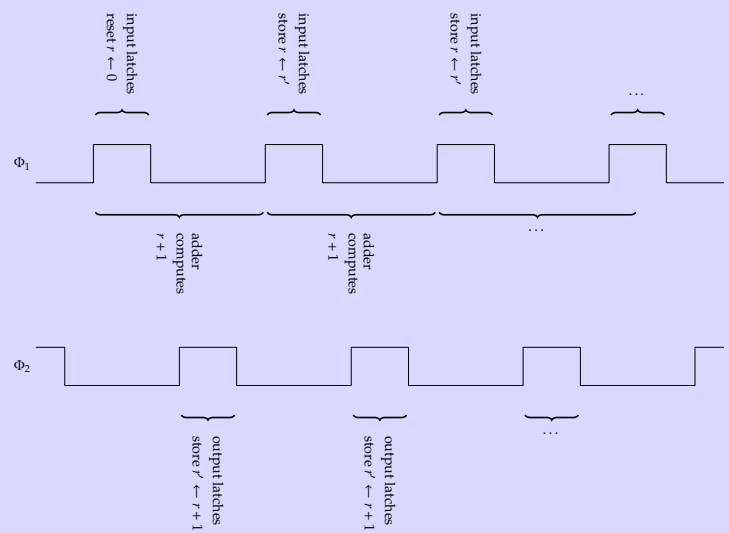
that you can (roughly) view this as a combination of

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.

although here, the control-path is *very* simple ...

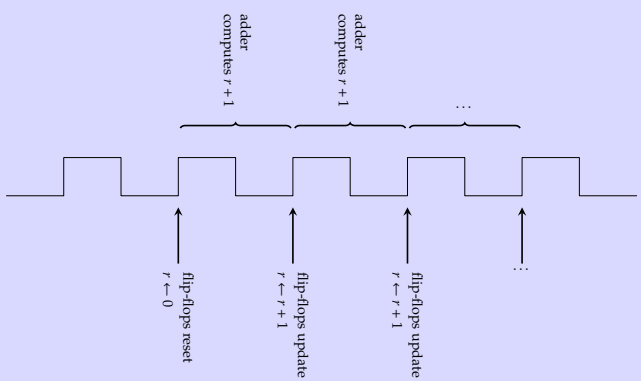
Notes:

Example



Notes:

Example



Notes:

Conclusions

- Take away points:
  1. The major concept to grasp is the introduction of **time**: not just delay, but step-by-step, controlled rather than continuous, uncontrolled computation.
  2. The design of storage, versus computational, components *can* be hard to grasp; probably more important than the detail is the *strategy* ...
  3. ... but the control-path in our counter is *very* simple; the next step is to examine how more complex control can be exerted over elements in the data-path.

Notes:

References and Further Reading

[1] Wikipedia: Clock signal.
http://en.wikipedia.org/wiki/Clock_signal.

[2] Wikipedia: Flip-flop.
http://en.wikipedia.org/wiki/Flip-flop_(electronics).

[3] Wikipedia: Latch.
http://en.wikipedia.org/wiki/Latch_(electronics).

[4] D. Page.
Chapter 2: Basics of digital logic.
In *A Practical Introduction to Computer Architecture*. Springer-Verlag, 1st edition, 2009.

[5] W. Stallings.
Chapter 11: Digital logic.
In *Computer Organisation and Architecture*. Prentice-Hall, 9th edition, 2013.

[6] A.S. Tanenbaum.
Section 3.1: Gates and Boolean algebra.
In *Structured Computer Organisation* [8].

[7] A.S. Tanenbaum.
Section 3.2: Basic digital logic circuits.
In *Structured Computer Organisation* [8].

Notes:

References and Further Reading

[8] A.S. Tanenbaum.
Structured Computer Organisation.
Prentice-Hall, 6th edition, 2012.

Notes: