CS201 W'00 Lecture Notes
Lecture 10

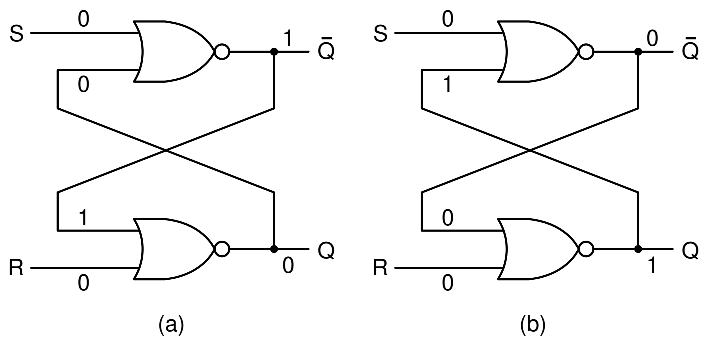
Basic memory circuits

Recall:

- **combinational** circuits without loops compute pure functions.

- Circuits with loops (**sequential** circuits) can "remember" values.

Simple example: SR latch



A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

(c)

Figure 3-22. (a) NOR latch in state 0. (b) NOR latch in state 1. (c) Truth table for NOR.

If R = S = 0, there are two stable states: Q = 0 or Q = 1.

Setting S $\rightarrow$ 1 forces Q $\rightarrow$ 1; if S $\rightarrow$ 0, Q remains 1.

Setting R $\rightarrow$ 1 forces Q $\rightarrow$ 0; if R $\rightarrow$ 0, Q remains 0.

If both S $\rightarrow$ 1 and R $\rightarrow$ 1 simultaneously, Q $\rightarrow$ 0, but its value is **nondeterministic** if both S $\rightarrow$ 0 and R $\rightarrow$ 0 simultaneously.

Clocked Latches

Typically want to restrict state change according to a **clocking** discipline.

Also, desirable to avoid danger of setting R and S to 1 at the same time.

Can solve these problems with Clocked D Latch

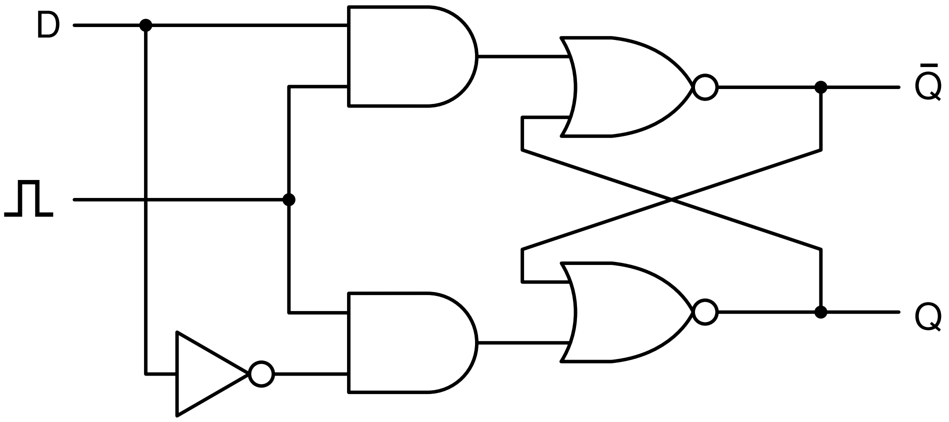


Figure 3-24. A clocked D latch.

This is a **level-triggered** gate.

Flip-flops

We often want to remember values at the instant of **transition** between clock levels.

We need to generate a brief signal that is high only at the moment of transition. This is a **pulse generator**:

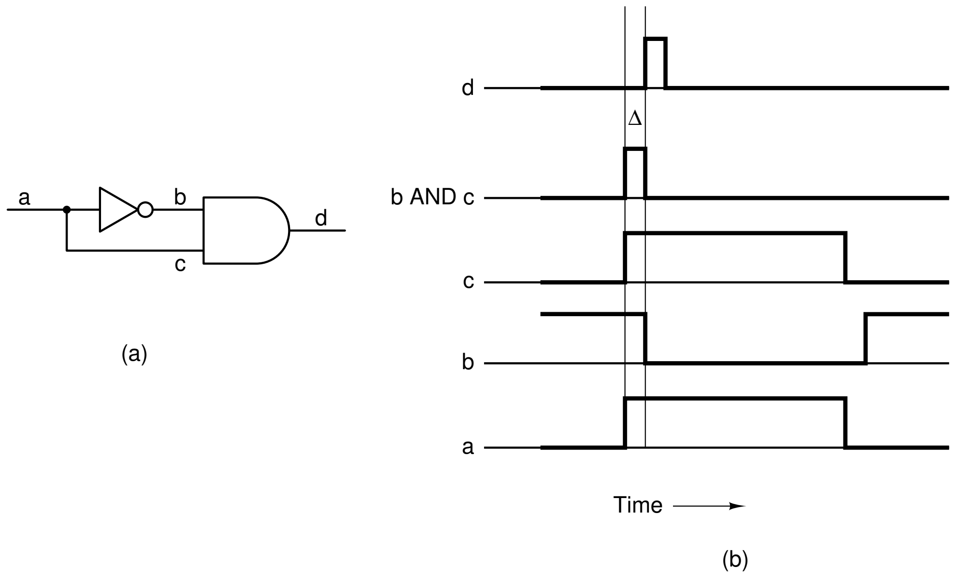


Figure 3-25. (a) A pulse generator. (b) Timing at four points in the circuit.

Flip-flops (2)

Now we use the pulse generator instead of the clock signal itself to get a **D flip-flop**:

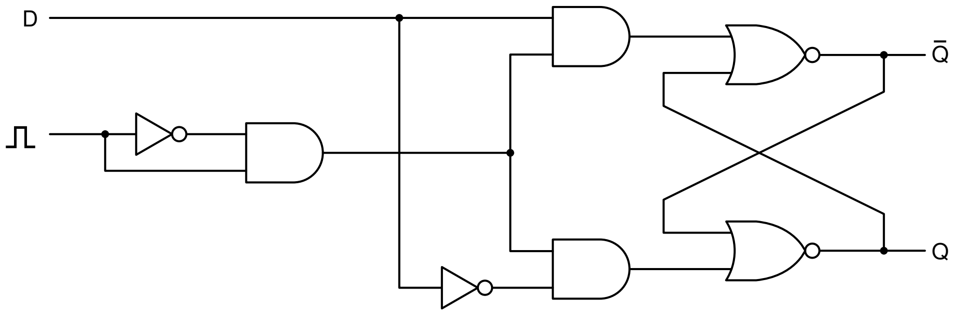


Figure 3-26. A D flip-flop.

The terms flip-flop and latch are often used interchangeably.

Can implement a **register** using one flip-flop per bit, with shared clocking and control logic.

Memory Arrays

Larger memories need to support **addressing** of individual words.

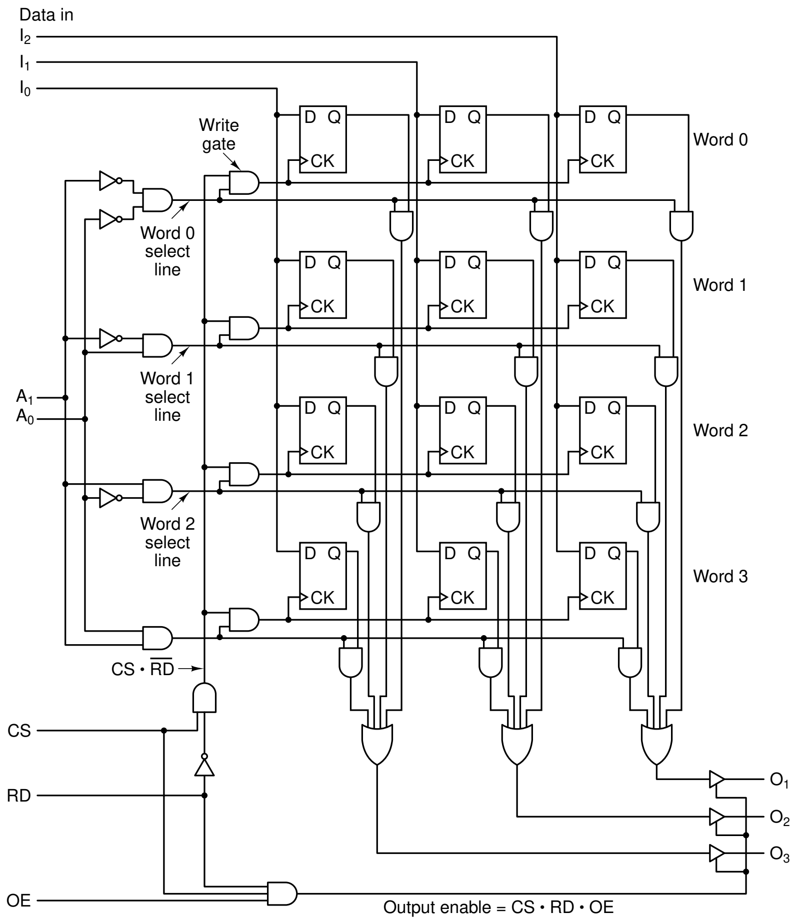


Figure 3-29. Logic diagram for a 4 × 3 memory. Each row is one of the four 3-bit words. A read or write operation always reads or writes a complete word.

CS = chip select; RD = read /write; OE = output enable
A = word address
I = data in; O = data out; wired together
Note **noninverting buffers**

RAM

The term **RAM** (Random-Access Memory) is used for ordinary read/write memory.

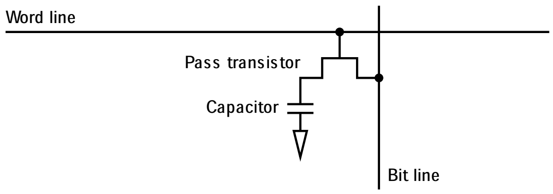
Static RAM is built from elements similar to D flip-flops. It can maintain its contents as long as power is kept on.

Actually, usually built using even simpler 6-transistor circuit based on a pair of cross-coupled inverters.

Static RAMs have fast access times (a few nanoseconds); often used for cache memories.

Dynamic RAM

We can actually store a bit using a single transistor and capacitor.



Bit state is indicated by capacitor charge level. Reading state destroys it.

Also, charge decays over time, so all bits must be **refreshed** (by reading and writing back) regularly.

Hence the term "**dynamic** RAM." Because they are much smaller, and hence cheaper, than SRAMs, they are commonly used for large main memories.

But they have slower access times (10s of nanoseconds).

More memory acronyms

ROM = Read-only memory: bits set during manufacture.

PROM = Programmable ROM: bits can be set (once) in the field.

EPROM = Erasable PROM: bits can be erased by UV light.

EEPROM = Electronically erasable PROM: bits can be erased in place.

VRAM = Video DRAM: allows fast serial access to bits even during regular read/write activity.

SDRAM = Synchronous DRAM: supports reads/writes of multiple bits given a single address.

Locality

The whole main memory can't be made up of circuits (e.g., SRAM) operating at the CPU's clock speed:

- too expensive
- too large.

So how can we keep the CPU supplied with instructions and data?

Fortunately, memory access patterns are ordinarily **not** random; they usually exhibit

- **spatial locality**: if a program accesses a location it is likely to access nearby locations.

– e.g.: sequential code; stacks; consecutive array accesses.

- **temporal locality**: if a program accesses a location it is likely to access it again soon.

– e.g.: loop code; local and global scalars; stacks; repeated array accesses.

Locality is an **empirical** fact in most programs, and can be enhanced by programmer and/or compiler strategies.

(But it is certainly possible to write programs that do **not** exhibit locality.)

So we can keep CPU happy by providing fast access to a (relatively small) subset of memory locations.

Caches

Idea: maintain a **copy** of frequently accessed memory contents in a small, fast memory called a **cache**, which sits between the CPU and the main memory.

When the CPU references memory, the cache controller first checks to see if the desired value is in the cache.

– If so (**cache hit**), it is quickly read/written there.

– If not (**cache miss**), the value is fetched from (slow) main memory to the CPU, and also stored in the cache, so that subsequent accesses to it will be fast.

To exploit spatial locality (and to simplify control hardware), when we load value into the cache, we actually load a small set of contiguous addresses at the same time. Each such set is traditionally called a **cache line**.

Since the cache has finite size, when we load a new line after a miss we have to throw an existing line out.

Direct-Mapped Cache

The simplest cache organization assigns a **fixed** cache address to each memory address, by simply ignoring the high-order bits.

If the desired line is in the cache, it must be at that address.

For example, with a cache containing 2048 lines and a 32-byte line size:

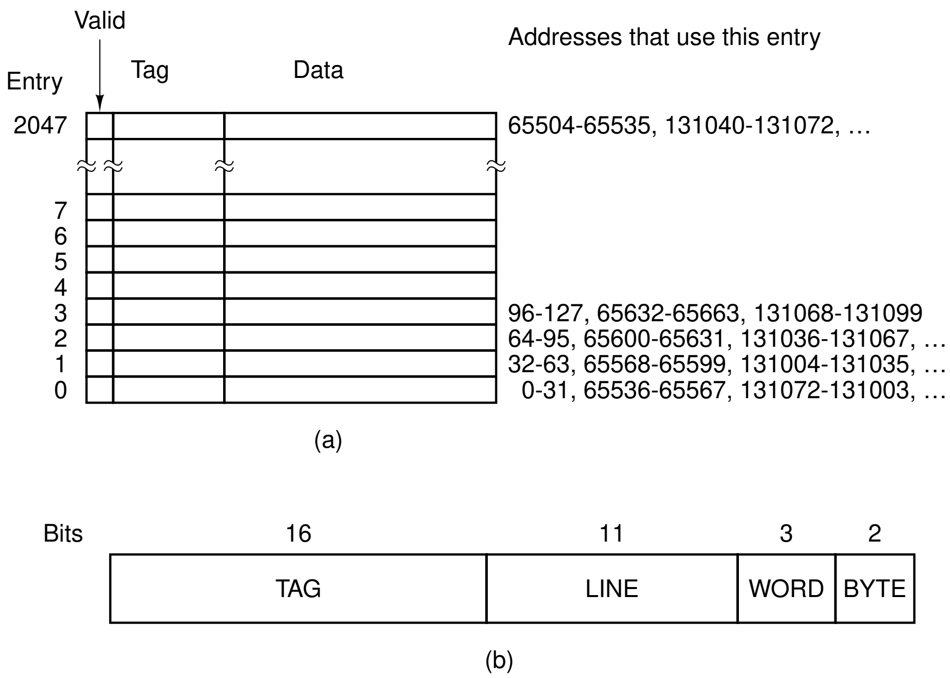


Figure 4-38. (a) A direct-mapped cache. (b) A 32-bit virtual address.

When a line is stored in the cache, the high-order bits of its memory address (the **tag**) are stored along with the data.

On each access:

- the low-order bits are used to select the cache line,
- the high-order bits are compared with the tag to see if the current cache entry is the right one.

Set-Associative Caches

Direct-mapped caches are cheap and simple, but can have awful worst-case behavior. [Lab 4.]

To address this problem, an n-**way set-associative** cache allows $n > 1$ different lines in each cache entry, each with a tag. To check for a cache hit, the controller compares the desired address with **all** n tags in parallel.

Lots more circuitry, but can reduce number of misses substantially over direct-mapped cache.

Of course, still get some misses. Which line should be booted out on a miss?

Typically use a **least-recently used (LRU)** algorithm to decide: kick out the line that was least-recently referenced. (Requires hardware to keep track.)

LRU exploits temporal locality.

Write Policy

Several options are possible for handling writes.

If write access hits the cache, certainly want to update the cache line.

- **Write through** caches also update main memory immediately.
- **Write back** caches defer the main memory update until the cache line is kicked out.

What if write access misses?

Write allocate caches bring the relevant line into the cache (as well as updating main memory)

Write no-allocate caches do not.

Multiple Cache Organization

Often use multiple levels of caches.

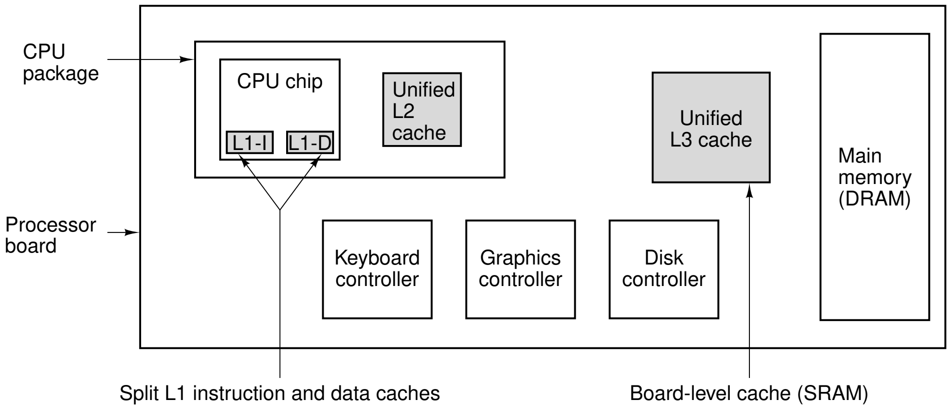


Figure 4-37. A system with three levels of cache.

Split caches contain exclusively data or instructions.

Unified caches contain both.