# Chapter 3 - Digital Logic Level

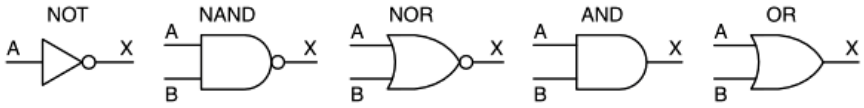
- **Gates**
- **Basic Digital Logic**
- **Memory**
  - **Storage Hierarchy**
- **CPU**
  - **PII**
  - **PicoJava**
- **Bus**
  - **PCI**
- **Homework:**
  - **Chapter 3 # 1, 4, 6, 12, 25, 35, 37, 40 (Due 4/22)**

Chapter 3 - digital logic. We'll look at gates, basic digital logic, and boolean algebra. Then we'll see how these are used to build memory, cpu, and busses - the three core elements of a computer.

Homework: Here is the next

# Gates and Boolean Algebra

- **X = !A;**


| A | X |
|---|---|
| 0 | 1 |
| 1 | 0 |

| A | B | X |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| A | B | X |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

| A | B | X |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| A | B | X |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

This is the lowest level we will talk about. You can make gates out of gears, relays, vacuum tubes, tinker toys, transistors, I don't care.

Lines carry information. In fact, they are boolean variables! They are like variables in a program. Boxes are operations, like plus, minus, etc.

So, that first picture is just X=!A;

Five basic boolean functions are shown below. All other boolean functions can be created by compositions of these, just like we can write complex algebraic functions $x= 3*y^2+5$

(powers are just repeated multiplications, at least in integerland, which can be built from boolean-land. Computers only approximate reals (at least numerically).

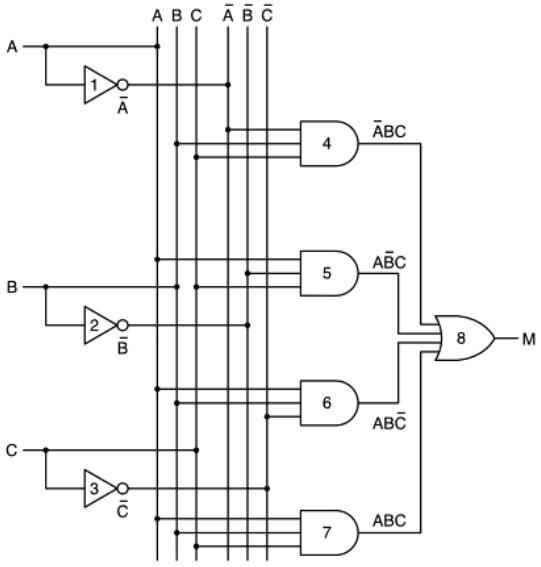
"Truth tables" are simply EXTENSIONAL representations of the functions. EXTENSIONAL means listing out all possible combinations of input values, and the corresponding output. This is reasonable because, unlike real or integer domain functions, boolean functions take a finite set of input values: 2^n, where n is the number of input parameters

# Functions

$M=\bar{A}BC+A\bar{B}C+AB\bar{C}+ABC$

```
M =(!A) && B && C
   || A &&(!B)&& C
   || A && B &&(!C)
   || A && B && C
```

| A | B | C | M |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |




AB means A && B

A + B means A or B

Why?

Because there is a close relation between or and plus, mathematically. Try it:

0+0 = 0, 0||0 = 0

1+0 = 1, 1||0 = 1

1+1 = 2, 1||1 = 1 (at least both results are non-zero.

Similary:

0*0 = 0, 0&&0=0

1*0 = 0. 1&&0 = 0

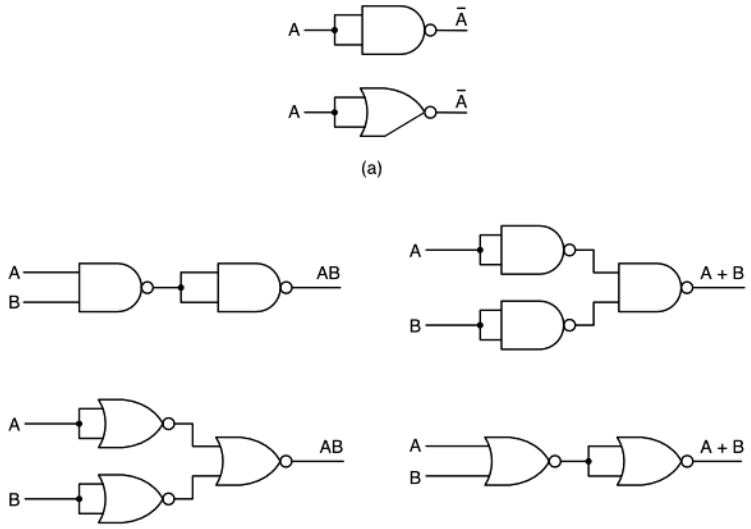
1*1 = 1, 1&&1 = 1

This under the intrepretation that 0 = false, 1 = true.

What else is interesting in the digital logic diagram above:

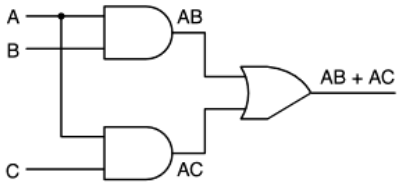
Convention - if lines cross but there is no circle, they don't actually touch, so

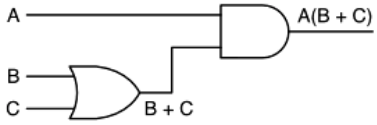
# Completeness of NAND and NOR




Any boolean function can be computed in terms of nand or nor alone!

# Equivalence and Minimization




| A | B | C | AB | AC | AB + AC |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 |

| A | B | C | A | B + C | A(B + C) |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 |

These two circuits compute the same function!

AB+AC = A(B+C)

Obviously we would like minimal realizations of functions we need, right?

We need some laws for how we can transform functions, so we can search for simpler forms

# Identities for formula transform

| Name | AND form | OR form |
|---|---|---|
| Identity law | $1A = A$ | $0 + A = A$ |
| Null law | $0A = 0$ | $1 + A = 1$ |
| Idempotent law | $AA = A$ | $A + A = A$ |
| Inverse law | $A\bar{A} = 0$ | $A + \bar{A} = 1$ |
| Commutative law | $AB = BA$ | $A + B = B + A$ |
| Associative law | $(AB)C = A(BC)$ | $(A + B) + C = A + (B + C)$ |
| Distributive law | $A + BC = (A + B)(A + C)$ | $A(B + C) = AB + AC$ |
| Absorption law | $A(A + B) = A$ | $A + AB = A$ |
| De Morgan's law | $\overline{AB} = \bar{A} + \bar{B}$ | $\overline{A + B} = \bar{A}\bar{B}$ |

Here are some laws, stated in two different forms, depending on whether we are working with and gates or with or gates.

Note that, just like in regular algebra, 0 is the additive identity and 1 is the multiplicative identity!

Math is cool!

You should know most of these, except maybe the null law, the Absorption law, and DeMorgan's law. (Actually, you should know all these from 231 - how many don't?)

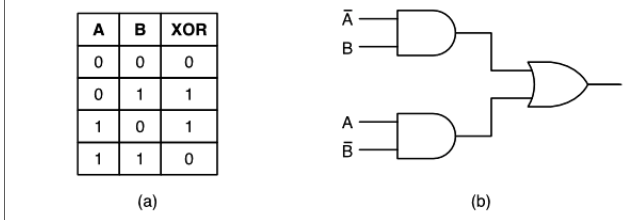
DeMorgan's law tells you how you can change from an AND gate to an OR gate, or vice-versa.

Also interesting - notice in reasoning about circuits, we switched from graphical representation to textual. Interesting.

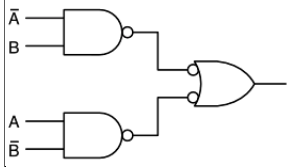
# Transforming formulas

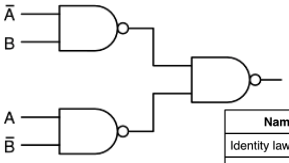
| A | B | XOR |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

(a)




(b)




| Name | AND form | OR form |
|---|---|---|
| Identity law | $1A = A$ | $0 + A = A$ |
| Null law | $0A = 0$ | $1 + A = 1$ |
| Idempotent law | $AA = A$ | $A + A = A$ |
| Inverse law | $A\bar{A} = 0$ | $A + \bar{A} = 1$ |
| Commutative law | $AB = BA$ | $A + B = B + A$ |
| Associative law | $(AB)C = A(BC)$ | $(A + B) + C = A + (B + C)$ |
| Distributive law | $A + BC = (A + B)(A + C)$ | $A(B + C) = AB + AC$ |
| Absorption law | $A(A + B) = A$ | $A + AB = A$ |
| De Morgan's law | $\overline{AB} = \bar{A} + \bar{B}$ | $\overline{A + B} = \bar{A}\bar{B}$ |

Let's try proving some equivalences. Let's look at the XOR implementations, and transform one to another.

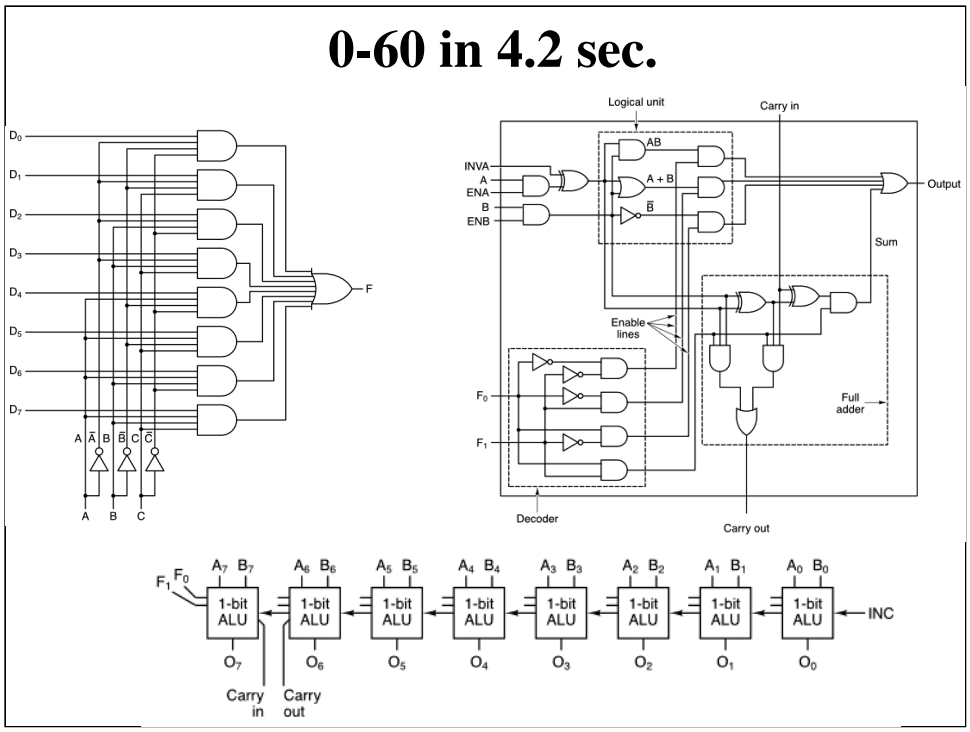
B is (!AB)+(A!B)

C Is (!( !(!AB)))+(!(!(A!B))

Well, that's easy, since (!(!X)) == x

How about d? (!(!(!AB))(!(A!B)))?

That is just deMorgan's law! (show it)




So, we can get from individual gates to a full ALU in just a few steps.

Upper left" simple muliplexer. Last gate is an or. How does this work?

Upper right: full alu: does

(1)ALU instruction decode (lower right, converts two bit number into four separate control lines, only one of which is on at a time - step throught this!

(2) Logic - A and B, A or B, ~B

(3) ALU - adds 2 1Bit numbers, with carry in and out

Do we need to step through boolean addition? Probably - add two three-bit numbers, showing carry out of each stage to next state

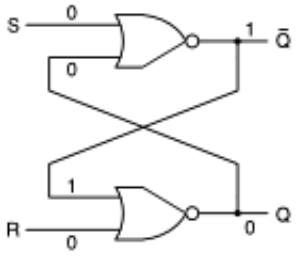
Bottom shows how 1 bit slices can be combined to create an 8 bit ALU. What is "INC" in at right? Usually 0, can be a "1" if our goal is just to increment by one

Huh? Well, that is what "ENA and ENB are for, set ENB to false, and B will look like all zero. So we can use the same hardware to increment A by 1, without having a separate register to hold the "one".

Hardware people do that all the time, sigh.

# Memory

- **SR Latch**




| A | B | NOR |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

So far we have looked at combinatorial logic - functions!. Functions are transformers, but they don't remember anything!

So how can we build a register?

The SR latch is a simple one-bit memory. How does it work? Let's try to do a truth table for it:

If S is 1 and R is 0… (S=1 -> !Q = 0 regardless of Q. Therefore inputs to lower gate are both zero, therefore Q is … 1!

Similarly, if S is 0 and R is 1… !Q = 1 and Q = 0

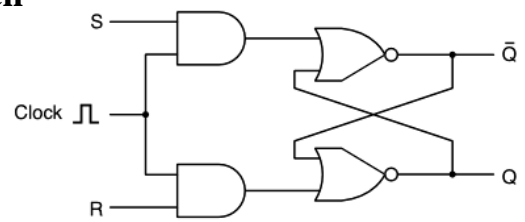
If S is 0 and R is 0… Then there are two stable states, depending on whether S was 1 or R was 1 last.

So with S and R both 0, outputs stay where they are!

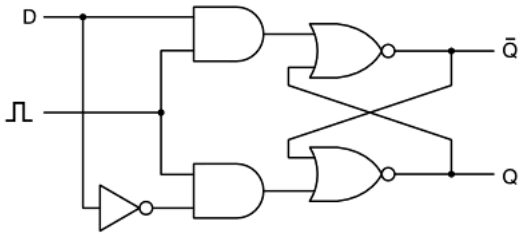
How about if S and R are both 1? That forces both Q and !Q to zero. Later, when both S and R go to zero, both nor gates want to output 1. But that will cause the other's output to go to zero. This is an unstable state, which eventually (ns) will be resolved by the latch going into one (unpredictable which) of its two stable states.

# Clocked latches

- **Clocked SR latch**




- **Clocked D latch**



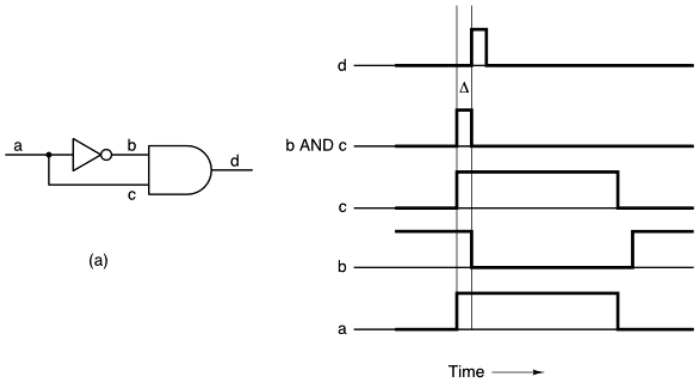

Cool. But usually we need to control WHEN a register stores something. We can use a “clock” to control this.

The Clocked SR latch above will only see 0 on its S and R input lines when the clock is 0, so it wil stay in its current state. When the clock is 1, it will see the actual values of S and R (remember - 1 is the identity operator for AND!)

Still a problem with that nasty undefined state: what if both S and R are 1 at the same time?

We can fix that by deriving R from S - it is just !S, right? Also the advantage that now we only need one line to carry the data bit around, instead of two. Cool.

# Clocking

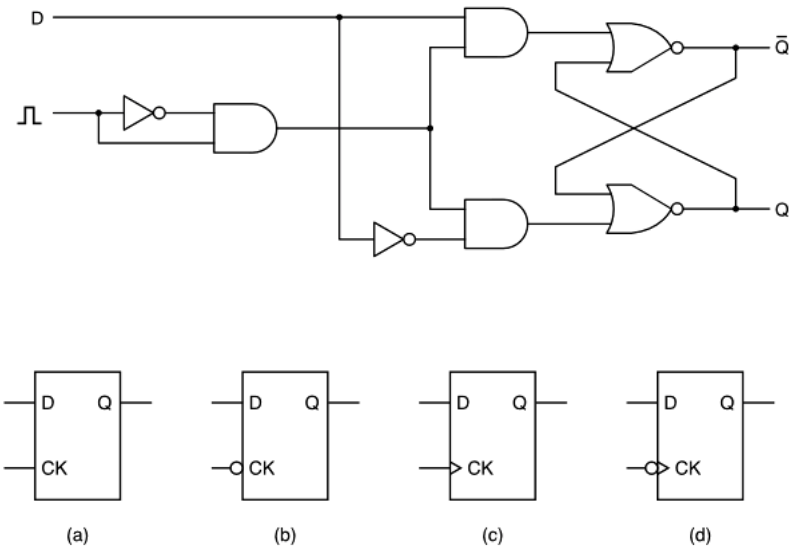



One more step to the most common storage: Because of various kinds of noise, including timing skew, it is easier to design such that we guarantee the signals will be present at a certain place at a particular time in the cycle, rather than for an extended duration.

So, a FLIP-FLOP stores data on an EDGE rather than during a duration. EDGE's can be generated locally, as shown above. We can then use that as the input to our D latch, generating a D flip-flop

(this is not completely standard notation. I am used to hearing "edge-triggered" vs "level-triggered" to be completely unambiguous).

# D Flip-Flop



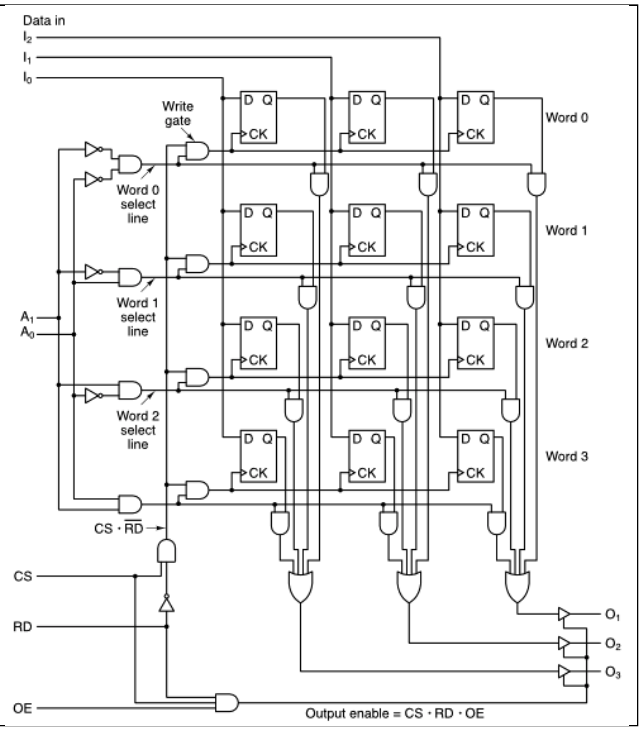

Above is the digital logic diagram for a D flip-flop. It uses our edge generator to extract an up edge from the clock

Below are some symbols typically used to represent various D latches and flip flops. What is the difference? Note the > and o. What do these mean?

- ">" Means edge triggered.

- "o" has its usual meaning - inversion. So a "o" on the clock input means this one stores when the clock value is 0, or, for edge triggered, when the clock is going from 1 to 0.

# 4 x 3 (3?)




Ok, what might a memory look like, then? For example, a register set inside a cpu?

Shown above is a 4 x 3 bit register set. Why three bit? Because that is how many fit on the page in the book!

Notice inconsistency in numbering input and output lines!

We see each bit is stored in a d flip-flop.

We see the three data in lines coming in at the top.

We see three output lines going out at the bottom.

What else? We see three control lines:

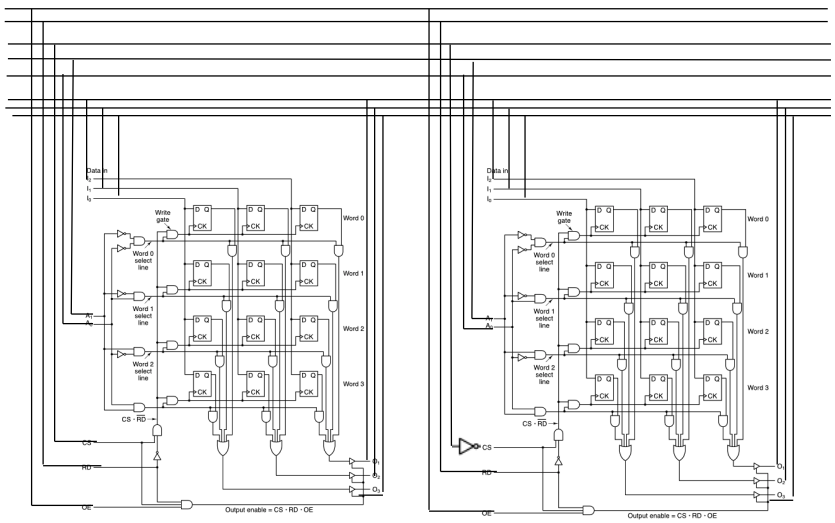
CS - chip select

RD - read

OE - output enable

So, if chip select is true, this block is active (that enables us to combine multiple chips to make a 8x3 or 12x3 or 16x3 or … using external logic to

# Simple Bus




Simple? Surely you jest! No really.

We'll look at buses in more detail shortly, but just to close off the discussion of memory, lets see how some of those features at the memory level translate into features at the bus level.

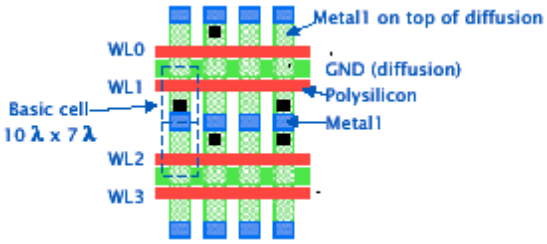
First of all, notice the overall bus organization: control, address, and data.

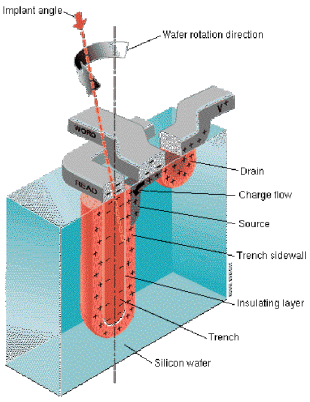
Second, notice the bus data wires are directionless! The memories can both read and write to the same wires.

But, notice this design can't transfer from one memory to the other directly, because there is no way to address both at once.

Notice the use of an inverter on the CS of the second ram to use the thurd address line as a select for which chip is selected.

# Dynamic RAM



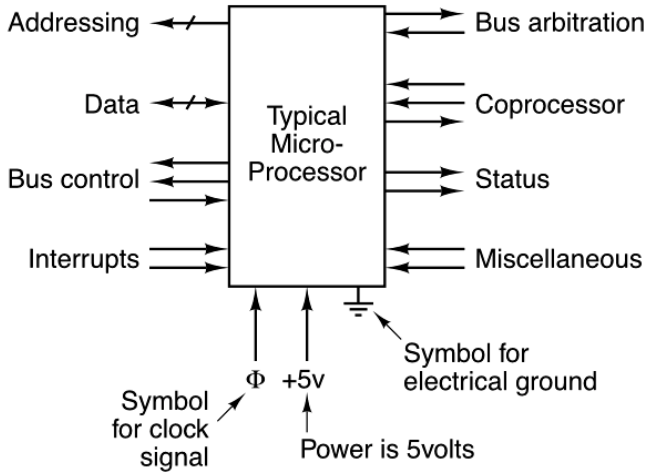




namic RAM 1-Transistor Cell: Lay

Dynamic ram uses a different storage mechanism that relies on storing charge in a capacitor.

ROM uses yet a different mechanism for storage.

# CPU chip I




Surprise: a cpu chip typically has a set of wires (address, data, control) that look just like what we just saw: this is the main CPU bus. There are a few others as well.

Clock

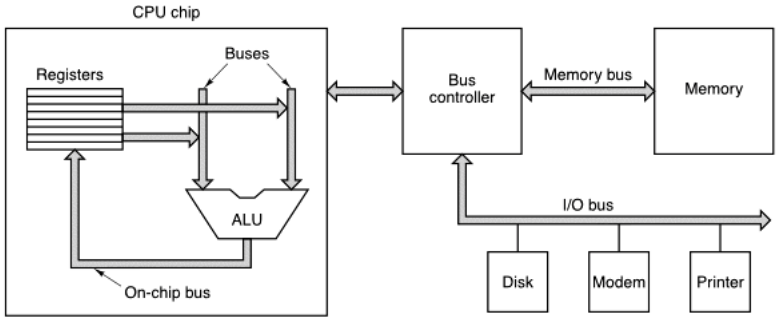
Power

Status - running, halted, etc.

Coprocessor - ignore.

Bus arbitration - we'll talk about that next.

# Buses Everywhere!



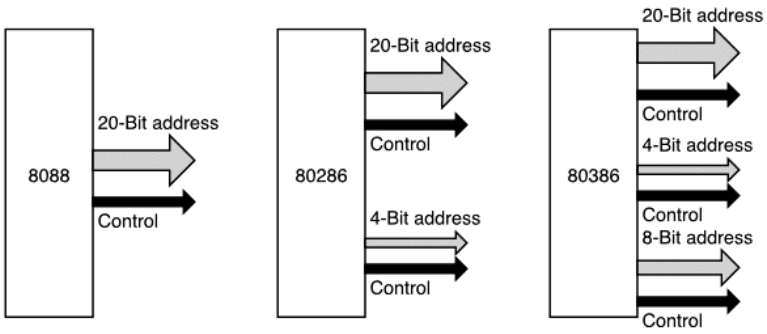

We looked at a bus to connect to internal cpu registers, at left.

We also saw the cpu bus, in terms of pins on the cpu chip itself - that is the bus connecting the cpu to the Bus controller.

Above is typical modern layout. The bus “chipset” is the dominant component you hear techies talking about when they say “oh, that motherboard has the Intel 8xyz!q###1 chipset!”

So what’s the big deal? Everyone knows about cpu clock, memory size, etc. Don’t hear nuch about busses. Let’s look at Buses

# Bus overview




- **P IV (.013 micron)**
  - **36 addr lines**
  - **64 data lines**
  - **MANY control lines**

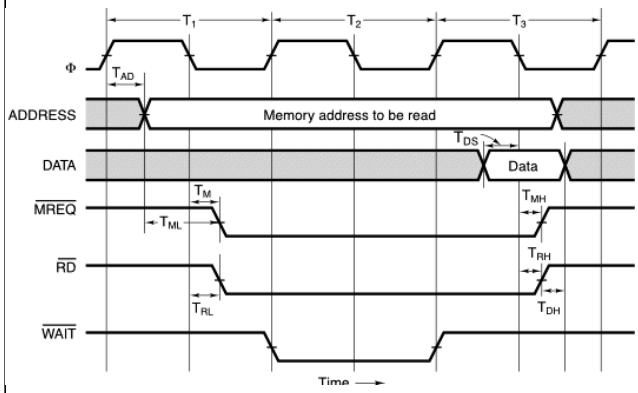
At the most abstract level, critical bus parameter is bandwidth: width * cycle-time

We can see at left how bus width (in this case address width) tends to expand over time. Twenty-bits is only enough to address 1 MB of memory (assuming byte-level addressing). 24 bit can address 16MB.

32 bits can address? 4 GB. We'll run out of that pretty soon.

Latest P IV has 36 addr lines - 64BG directly addressable ram.

# Synchronous bus overview




| Symbol | Parameter | Min | Max | Unit |
|---|---|---|---|---|
| $T_{AD}$ | Address output delay | | 11 | nsec |
| $T_{ML}$ | Address stable prior to $\overline{MREQ}$ | 6 | | nsec |
| $T_M$ | $\overline{MREQ}$ delay from falling edge of Φ in $T_1$ | | 8 | nsec |
| $T_{RL}$ | RD delay from falling edge of Φ in $T_1$ | | 8 | nsec |
| $T_{DS}$ | Data setup time prior to falling edge of Φ | 5 | | nsec |
| $T_{MH}$ | $\overline{MREQ}$ delay from falling edge of Φ in $T_3$ | | 8 | nsec |
| $T_{RH}$ | $\overline{RD}$ delay from falling edge of Φ in $T_3$ | | 8 | nsec |
| $T_{DH}$ | Data hold time from negation of $\overline{RD}$ | 0 | | nsec |

Ok, so here is a “synchronous” bus.

In a synchronous bus, everything happens to the tune of a master clock (think the old roman slave galleys, with a master drum beater setting the rowing pace. - everyone pulls on the oars when the drum beats.)

Let’s look at a typical simplified synchronous bus: The bus “speed” is the frequency of the clock. Although that is not necessarily the data rate, as we will see.

This bus has four control lines: clock, !MREQ, !RD, !WAIT

unknown number of address lines - doesn’t really matter for our purposes here.

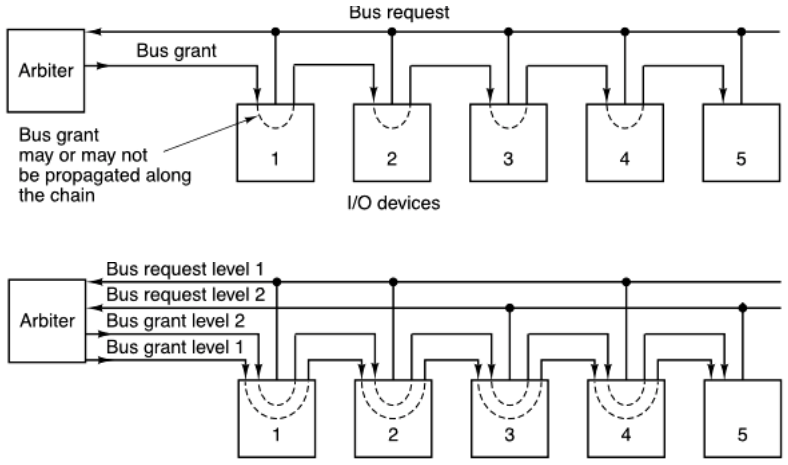
similarly, unknown number of data lines

Example is a 40 mhz bus - that is, T1 is 25 ns (as is T2, T3, ….)

Here’s the deal: suppose I want to read a memory location. Then I assert !MREQ and !RD simultaneously - that is, since they are both negative, I set them both to zero (why -in old day made it easy to share).

But note that BEFORE I do that, I have to make sure the address is on the

# Bus arbitration




Sometimes we have more than 1 bus master.

Bus arbitration decides who gets it.

Fixed order

Piorities

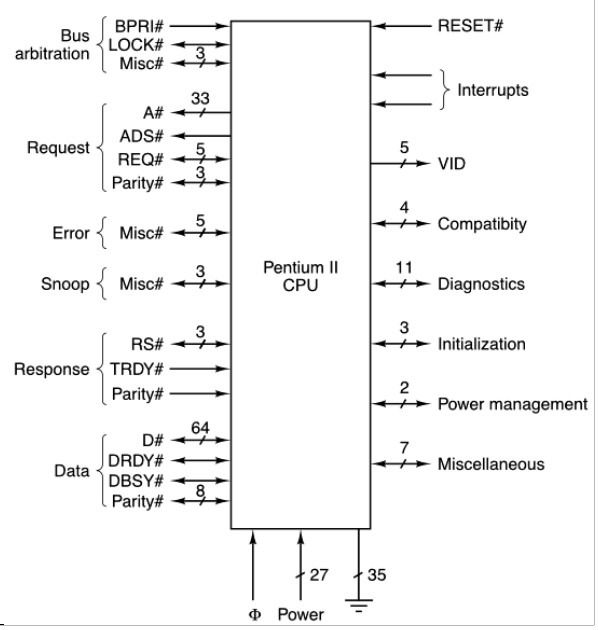
Top is example of fixed order. If any device requests bus, arbiter grants it. If I/O device 1 wants it, it takes it and leaves its outgoing line low. Otherwise it passes the bus grant line value out to next line.

Does this make sense? How might we implement it?

BusGrantOut = BusGrantIn * (!BusRequest_me) (what would that look like in digital logic? Good question!)

# P II cpu bus

- **33 ADDR**
- **64 Data**
- **Bus pipeline**
  - **Arbitration**
  - **Request**
  - **Error**
  - **Snoop**
  - **Response**
  - **Data**




Here is the P II pinout - 242 connectors (latest P IC has 478)

33 addr, but three extra set to 0! Addressing is to a 64 bit word! So addressability is actually 64 GB!

Note separate sets of pins for each phase of a bus transfer.

We've talked about bus arbitration - allows for multiple masters on the cpu bus (e.g., multiprocessors).

Request is obvious - once you have the bus, say what you want.

Error?

Snoop? What if the data you want is actually in another cpu's cache, and has been changed there?

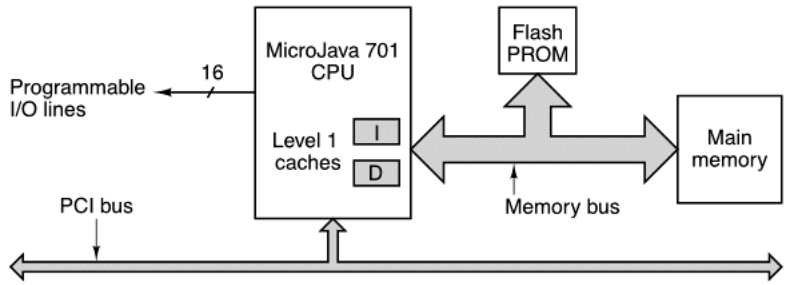
Response - "ok, here it comes"

Data - at last…

So a PII with a 400 mhz cpu bus (there is no such thing) can do what latency and data rate?

# PicoJava

- **Built-in ram/PROM bus**
- **Built-in PCI bus support**
- **16 Built-in I/O lines**

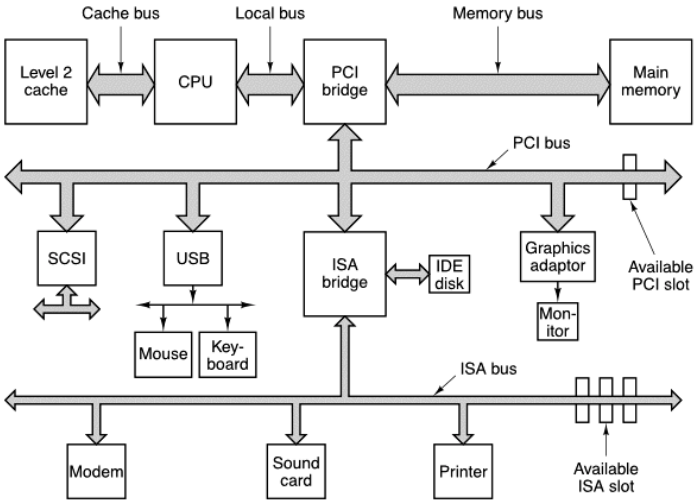



Embedded systems are HUGE. Far more embedded cpus than ones you see.

Many do very simple things for which speed isn't an issue.

Pico java is a chip designed to run Java without need for a JVM (software interpreter) in embedded apps.

# A real bus - PCI

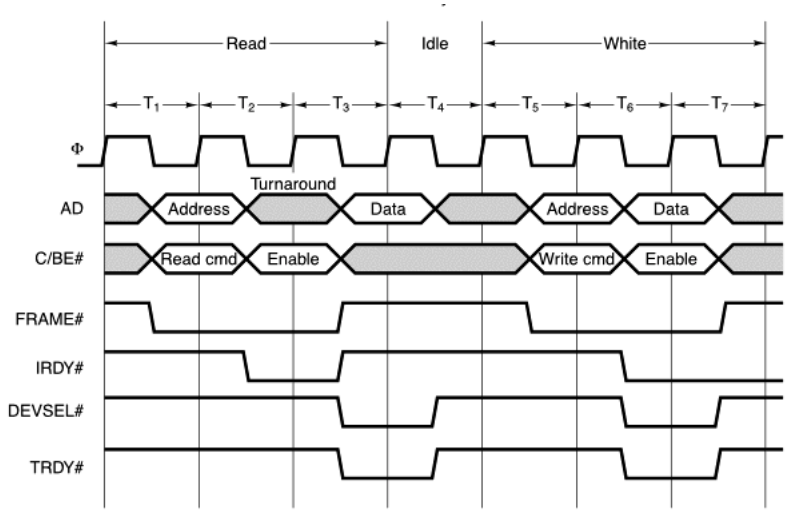



Overall arch of a motherboard.

Of course, these days it gets a bit worse, with AGP II for fast video.

Also, these days ISA may be gone completely, and modem, soundcard, ethernet, printe,r all on PCI

# PCI - more




Several varieties of PCI (voltage, width, clock) we will ignore this, assume 3.3V 66 mhz PCI bus

1. Bus is Synchronous
2. Bus is MULTIPLEXED! Data and address share same 64 physical lines, sent one at a time
    1. Complex bus protocol, sigh.

    Centralized arbitration

    Basic read and write cycles shown above:

    For a read, address put on A/D lines, the lines are turned around, then responder puts data on lines.

    For a write, turnaround is unnecessary, so write can be done in 3 bus cycles, whereas read takes 4.

    So, how many bytes/sec can a 66mhz PCI bus transfer in read mode? Write mode?