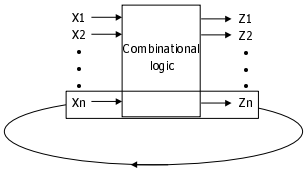
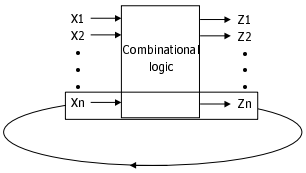
Sequential logic

- Up until now, we've built combination circuits: outputs are just a function of the inputs.
- Now, we get into circuits with feedback.



Sequential logic diagram



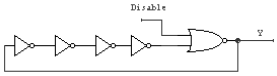
- But, how do we know that the outputs Z1..Zn will stabilize? Isn't possible that the outputs will endlessly change, never reaching a stable state?
- Yes, this can happen! We will have to make sure that perpetual osciallation doesn't happen (unless we WANT it to happen...)

Sequential logic: what's the point?

- But, feedback seems to make things more complicated (e.g.: need to make sure things don't oscillate). So what's the point?
- Well, feeding outputs back into inputs lets us do interesting things, like...
  - memory!
  - oscillating circuits!

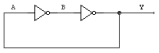
Experiment a little

- Let's start experimenting with feedback.



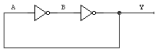
- This circuits oscillates because there are an odd number of inversions in the feedback. So... what happens if we only have an even number of inversions in the feedback? Let's take a look.

Cascaded inverters



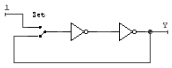
- **IF** A is "1", then B is "0", which forces A again to "1", which forces B again to "0", and so on. Thus, the ouput Y is "1", and stays "1" forever. This is a steady state (contrast this to the oscillator you did in your assignment).
- Similarly, **IF** A is "0", then Y will stay at "0" forever.
- Wow, this looks like a bit of memory (if you ignore the magical **IF**...)
- But, wait a second... How can this circuit store a value forever, it doesn't seem to be using any power: after all we are not applying any inputs...
- This is a common fallacy! Remember, we don't draw the power supplies, but they are ASSUMED to be there.

Cascaded inverters (cont'd)



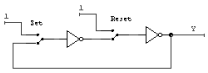
- In describing this circuit, we used the magical **IF**. But what happens if you build this in the lab? Which state will it go in? Will it go into one of the two states described before (Y = "0", or Y = "1")? Or is there a possibility that A and B will just remain undefined (maybe both will stay at 2.5 Volts...)
- Well, in theory this circuit has an undefined behavior. But if you build it, it WILL go into one of the two states. Why?
  - Say the circuit lands in an undefined state, maybe A=2.5 Volts, and B = 2.5 Volts.
  - The only way you'll stay in this state is if you are in perfect equilibrium.
  - But then, any perturbation in A or B (caused maybe by electromagnetic radiation from the moon...) will cause your circuit to spiral towards one of the two states.
- What's worse is that we **don't know** what state it will go in! That's not good... We need a way to guarantee that we land in the state we **WANT**.

Cascaded inverters with "Set"

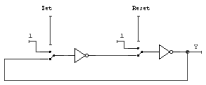


- Think of the switch as a push button, with the default state as shown. When you activate the button, it connects a "1" to the input of the first inverter. When you release the button, it bounces back to the default state.
- When you push the button, the loop is broken. The effect is that it sets Y to "1". When we release the button, the value of "1" at the output remains there.
- We'll call this a "Set" (we're setting the output…). This is good, but this means we can only store a "1". What about a "0"?

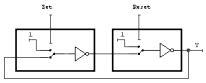
Cascaded inverters with "Set" and "Reset"



- Add a "Reset" switch, which makes the output Y go to 0. Again, when the switch is released, Y stays at 0.
- What happens if none of the switches are pressed? Well, the value of Y that was there before will stay there. This is called a "Hold".
- So, we have a one bit of memory that we can set, reset, or just leave as is.
- Now, pushing switches is not a scalable solution. We need to control our memory cell with digital signals:

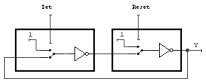


Cascaded inverters with "Set" and "Reset"

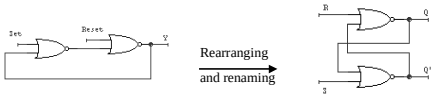


- Now, take a look at the black boxes above. Each has two inputs. When the first input is 1, the output should go to 0, whereas when the first input is 0, then the output should be the negation of the second input. What is that?

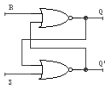
Cascaded inverters with "Set" and "Reset"



- The black boxes are NOR! So we can redraw the circuit with NORs instead of the black boxes:



Cross-coupled NOR gates



- This is called an RS latch. We can build a table that shows how this circuit behaves:

S	R	Q
0	0	hold
0	1	0
1	0	1
1	1	???

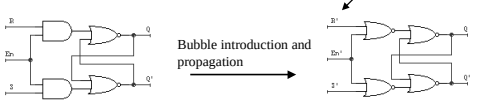
- What happens in the 1, 1 state? At first, both Y and Y' will be 0 (that's already a problem, since that means we can't call them Y and Y'…). But, then, if you "release" the S and R inputs (thus, setting S=0, and R=0), Y and Y' will oscillate. Forever!
- So… **We DISALLOW this state.**

RS Latch

- R-S latch always samples its inputs.
- That means that a glitch in the inputs is fatal (called the 1's catching problem). Wouldn't it be great if the latch didn't always sample its inputs? That way, we wouldn't be required to build hazard free circuits (which lessens not only the design time, but also the hardware).

- First attempt at solving this: use an enable signal.
  - When latch is enabled, Set and Reset work.
  - When latch is disabled, hold.

Note inverted inputs



Carry-lookahead logic

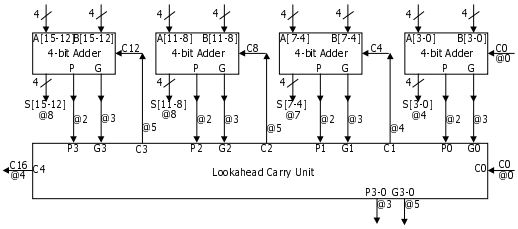
- Carry generate: Gi = Ai Bi
  - must generate carry when A = B = 1
- Carry propagate: Pi = Ai xor Bi
  - carry-in will equal carry-out here
- Sum and Cout can be re-expressed in terms of generate/propagate:
  - Si = Ai xor Bi xor Ci
    = Pi xor Ci
  - Ci+1 = Gi + Ci Pi

Carry-lookahead logic (cont'd)

- Re-express the carry logic as follows:
  - C1 = G0 + P0 C0
  - C2 = G1 + P1 C1 = G1 + P1 G0 + P1 P0 C0
  - C3 = G2 + P2 C2 = G2 + P2 G1 + P2 P1 G0 + P2 P1 P0 C0
  - C4 = G3 + P3 C3 = G3 + P3 G2 + P3 P2 G1 + P3 P2 P1 G0
    + P3 P2 P1 P0 C0

Carry-lookahead adder with cascaded carry-lookahead logic

- Carry-lookahead adder
  - 4 four-bit adders with internal carry lookahead
  - second level carry lookahead unit extends lookahead to 16 bits



Carry-select adder

- Redundant hardware to make carry calculation go faster
  - compute two high-order sums in parallel while waiting for carry-in
  - one assuming carry-in is 0 and another assuming carry-in is 1
  - select correct result once carry-in is finally computed