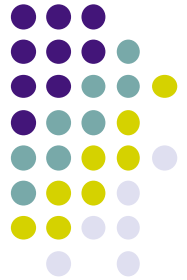
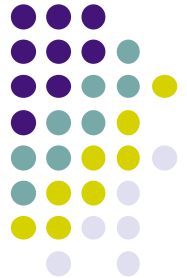
Digital logic simulation



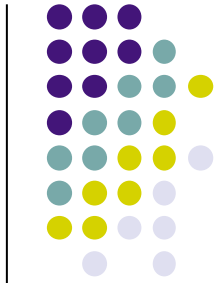
- The deterministic dataflow paradigm makes it easy to model digital logic circuits
- We show how to model combinational logic circuits (no memory) and sequential logic circuits (with memory)
- Signals in time are represented as streams; logic gates are represented as agents

Modeling digital circuits



- Real digital circuits consist of active circuit elements called gates which are interconnected using wires that carry digital signals
- A digital signal is a voltage in function of time
  - Digital signals are meant to carry two possible values, called 0 and 1, but they may have noise, glitches, ringing, and other undesirable effects
- A digital gate has input and output signals
  - The output signal is slightly delayed with respect to the input
- We will model gates as agents and signals as streams
  - This assumes perfectly clean signals and zero gate delay
  - We will later add a delay gate in order to model gate delay

Digital signals as streams

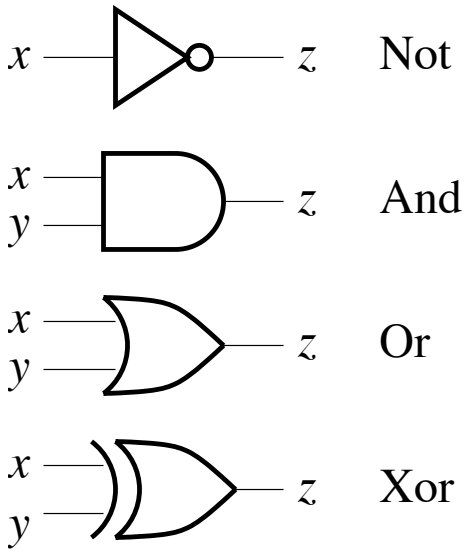


- A signal is modeled by a stream that contains elements with values 0 or 1

  $S=a_0|a_1|a_2|...|a_i|...$

- Time instants are numbered from when the circuit starts running
- At instant i, the signal's value $a_i \in \{0,1\}$

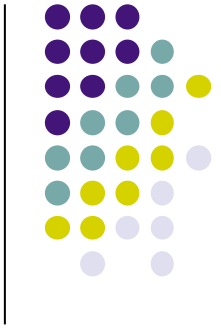
Digital logic gates



		z			
x	y	Not	And	Or	Xor
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

- Some typical logic gates with their standard pictorial symbols and the boolean functions that define them
- But gates are not just boolean functions!

Digital gates as agents

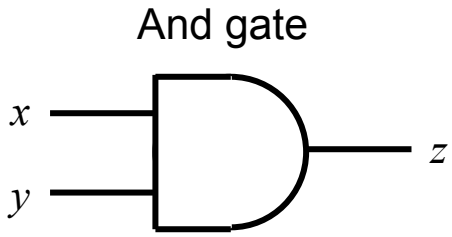


- A gate is much more than a boolean function; it is an active entity that takes input streams and calculates an output stream

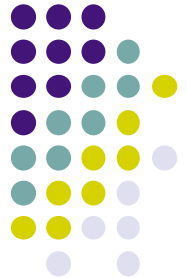
```
fun {And A B} if A==1 andthen B==1 then 1 else 0 end end
fun {Loop S1 S2}
   case S1#S2 of (A|T1)#(B|T2) then {And A B}|{Loop T1 T2} end
end
thread Sc={Loop Sa Sb} end
```

- Example execution:

```
Sx=0|1|0|Tx  % input signal x
Sy=1|1|0|Ty  % input signal y
Sz=0|1|0|Tz  % output signal z
```

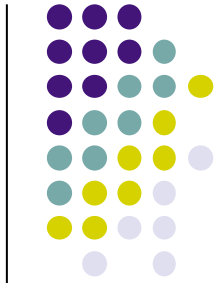


Creating many gates



- Let us define a proper abstraction for building all the different kinds of logic gates we need
  - We define the function GateMaker that takes a two-argument boolean function Fun, where {GateMaker Fun} returns a function FunG that creates gates
  - Each call to FunG creates a running gate based on Fun
- This gives three levels of abstraction that we can compare with object-oriented programming:
  - GateMaker is analogous to a generic class
  - FunG is analogous to a class
  - A running gate is analogous to an object

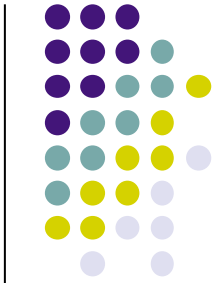
GateMaker implementation



- Calling {GateMaker F} creates a gate maker:

```
fun {GateMaker F}
   fun {$ Xs Ys}
      fun {GateLoop Xs Ys}
         case Xs#Ys of (X|Xr)#(Y|Yr) then
            {F X Y}|{GateLoop Xr Yr}
         end
      end
   in
      thread {GateLoop Xs Ys} end
   end
end
```

Making gates



- Each of these functions can make gates:

```
AndG={GateMaker fun {$ X Y} X*Y end}
OrG={GateMaker fun {$ X Y} X+Y-X*Y end}
NandG={GateMaker fun {$ X Y} 1-X*Y end}
NorG={GateMaker fun {$ X Y} 1-X-Y+X*Y end}
XorG={GateMaker fun {$ X Y} X+Y-2*X*Y end}
```