# *VIS: Verification Interacting with Synthesis*

- Three Main Operations
  - Formal Verification
    - equivalence checking (comb. and seq.)
    - property checking (CTL and lang. empty.)
  - Simulation
  - Logic Synthesis (mainly via SIS right now)
- Main Properties
  - Interactive
  - Hierarchical
  - User controlled parameters



# *Why Formal DesignVerification?*

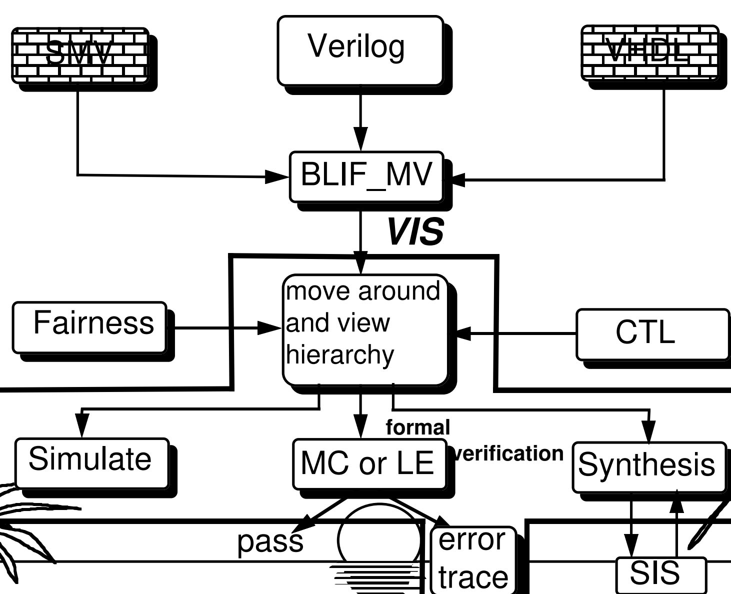
- Designs are getting more complex
  - simulation becoming more inadequate
  - time to market tighter and more critical
- Growing interest in many companies
  - some major successes
  - designers becoming convinced of need
  - small verification groups starting up
    - IBM, Intel, Motorola, ATT, Cadence, Siemens, NEC, Fujitsu, SGI, Sun, HP, …
- Tools coming up to task of real designs

# *Why VIS?*

- Provide state-of-the-art for verification
- Need for a clean platform
  - can be shared with others - academia and industry
  - foundation for commercial and in-house systems
- Research continues
  - need to easily upgrade as research results are obtained
  - platform to try out real examples
- Investigate synergy between
  - simulation, verification, and synthesis



# *Overview of VIS*



# *This Course*

- lots of hands-on experience with VIS
- use of abstraction and nondeterminism
  - writing abstract environments
- property checking and CTL
- using verification for debugging
- understanding underlying algorithms
  - controlling parameters
    - verifying large examples - increasing efficiency
- learning about software engineering aspects



# *Availability of VIS*

- VIS home page
  - http://www-cad.eecs.berkeley.edu/Respep/Research/vis/index.html
  - contains a lot of documentation and command usage
- Getting VIS
  - ftp://ic.eecs.berkeley.edu/pub/Vis/glu-1.0.tar.gz
  - ftp://ic.eecs.berkeley.edu/pub/Vis/vis-1.0.tar.gz
- Support - e-mail to: vis@ic.eecs.berkeley.edu
- Machines:
  - DEC - MIPS, Alpha PC - Linux 1.3.20
  - Sun - OS, Solaris HP - Snake

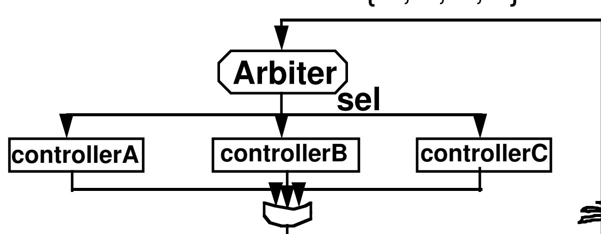
# *Outline of Course*

- overview
- getting to know VIS
- verification by simulation
- model checking
- fairness constraints
- all these use a common small example
- larger verification project introduction
- computational techniques
  - controlling efficiency and size
- hands-on design and verification
- project evaluation
- software engineering
- course evaluation



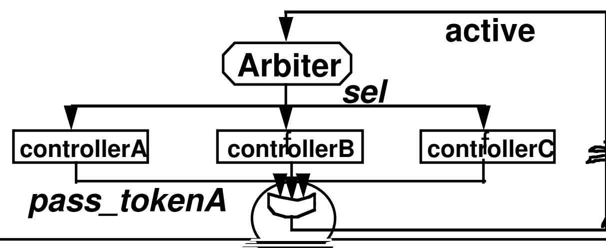
# *A Simple Example*

- The top level model has 3 controllers that want to get access (token) to a bus.
- They communicate with an arbiter such that only one controller has access (token) at a time.
- The arbiter output sel tells which controller gets the token next: sel $\epsilon$ {A,B,C,X}



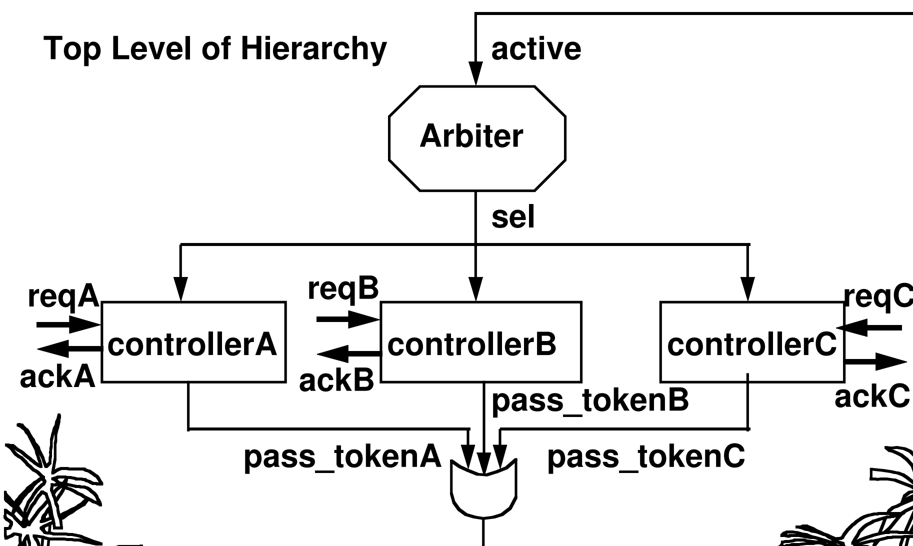
# *Simple Example*

- A controller has a single output pass_token:
- A controller passes “token” pass_token=1 if
  1. it has token and is finished or
  2. it is granted token but does not want it.
- The outputs of the controllers are OR’ed together giving the active signal.
- If  active=0, then no token is available and the arbiter puts out X.






**Top Level of Hierarchy**

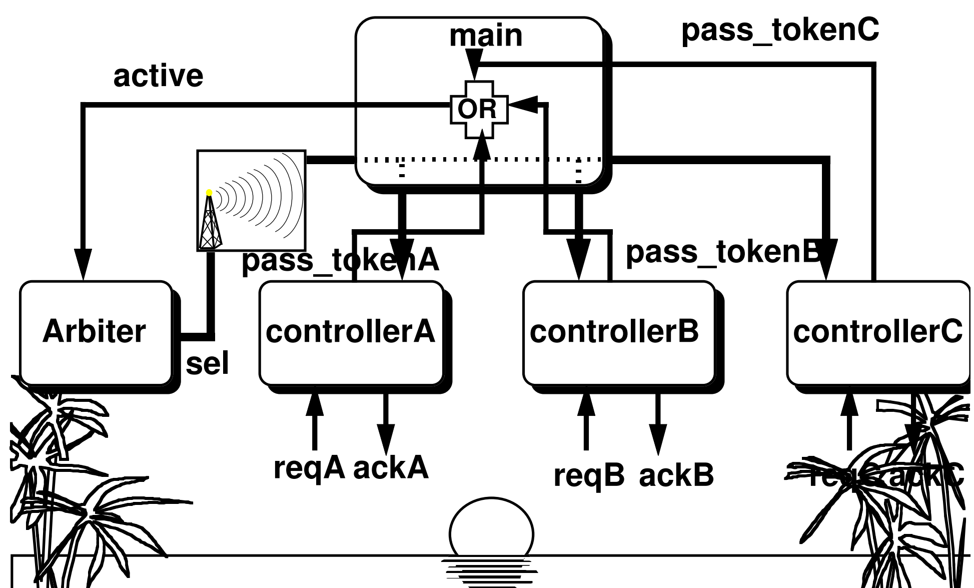


# VERILOG description

```
Module main(clk);
input clk, reqA, reqB, reqC;
output ackA, ackB, ackC;

selection wire sel;
wire active;

assign active = pass_tokenA || pass_tokenB || pass_tokenC;

controller controllerA(clk, reqA, ackA, sel, pass_tokenA, A);
controller controllerB(clk, reqB, ackB, sel, pass_tokenB, B);
controller controllerC(clk, reqC, ackC, sel, pass_tokenC, C);

arbiter arbiter(clk, sel, active);

endmodule
```

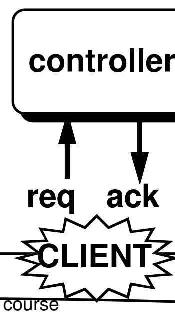


# Hierarchy



# ***Client Protocol***

- when client wants bus he raises req and holds it high - he continues to hold it high until he wants to release it
- controller should hold ack high precisely when it has a token

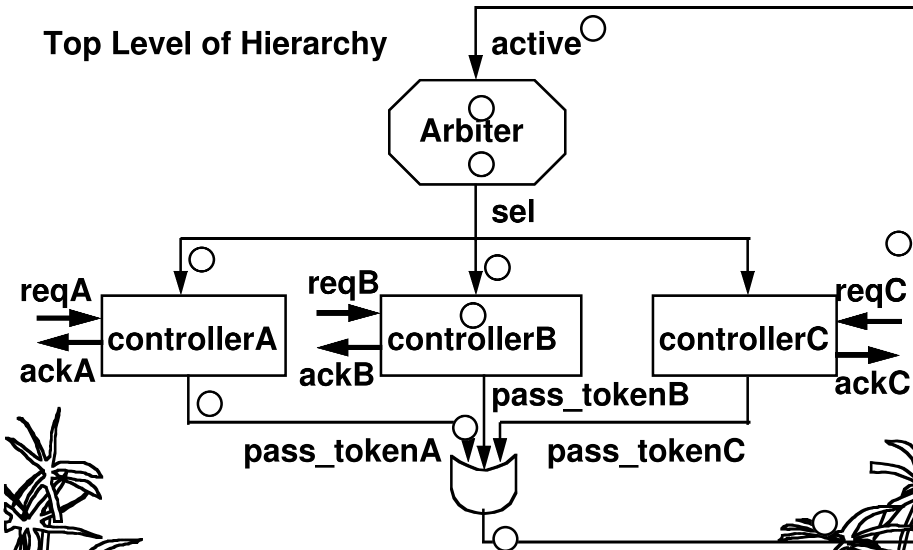





# ***Arbiter Behavior***

- it just cycles it's state from A->B->C->A each clock cycle
- it outputs sel=X if no controller is passing the control i.e. no token is being passed
- otherwise it outputs sel=present state, e.g. sel=A means that controllerA is offered the token
- this way each controller has the possibility of getting the token (we hope)

## Top Level of Hierarchy






# *Getting to Know VIS: Lab Exercise*

- start up NETSCAPE and go to VIS homepage and then to User Documentation and then to User Commands
- go to directory “session2”
- compile verilog file ex0.v using vl2mv
- start up VIS
- read in BLIF-MV file ex0.mv using read_blif_mv

# *Getting to Know VIS: Lab Exercise*

- cd to controllerA
- simulate this node with 10 random vectors
- write out this node in BLIF-MV
- synthesize this node in SIS and import result back into VIS
- compare with the BLIF-MV file written out; use both combinational and sequential verify



# *Some Commands You Will Need*

- ***read_blif_mv (rlmv)***
- ***write_blif_mv (wlmv)***
- ***cd***
- ***ls***
- ***init_verify (init)***
- ***seq_verify***
- ***comb_verify***
- ***print_hierarchy_stats (phs)***
- ***print_network_stats (pns)***
- ***write_blif (wl)***
- ***read_blif (rl) [-e f.enc]***
- ***simulate (sim) [-n 10]***

## ***Some Things You Need to Know***

- ***phs*** will tell information about node
- ***phs -R*** will tell about the entire hierarchy under a node
- some commands will not work without doing ***init*** - e.g. ***simulate, seq_verify***
- the ***read_blif*** command should be given the option ***-e modelname.enc***. Otherwise it will destroy the entire hierarchy and replace it with a new one.



## *read_blif*

- It has two modes
  - read_blif file.blif
  - read_blif -e encoding.enc file.blif
- The first is just like read_blif_mv but it accepts the blif format instead of blif_mv
- The second is “incremental” in that it just replaces the current node in the hierarchy
- The encoding file contains encoding information so that the node’s inputs and outputs remain in MV (multivalued) format. However, internally it is binary since it is a blif file.

# result of read_blif -e E.enc F.blif