

# Hardware Description Languages

## Basic Concepts

Dinesh Sharma

Microelectronics Group, EE Department  
IIT Bombay, Mumbai

Jan 2014

# TEXT BOOKS

## VHDL

VHDL

A VHDL PRIMER

THE DESIGNER'S GUIDE TO VHDL

Z. NAVABI

J. BHASKER

P.J. ASHENDEN

## VERILOG

A VERILOG HDL PRIMER

VERILOG HDL

J. BHASKER

S. PALNITKAR

# The Design Process

We ask our selves the question:  
**What is Electronic Design?**

# The Design Process

We ask our selves the question:  
**What is Electronic Design?**

Given specifications, we want to develop a circuit by connecting *known* electronic devices, such that the circuit meets given specifications.

# The Design Process

We ask our selves the question:  
**What is Electronic Design?**

Given specifications, we want to develop a circuit by connecting *known* electronic devices, such that the circuit meets given specifications.

“Specifications” refer to the description of the desired behaviour of the circuit.

# The Design Process

We ask our selves the question:  
**What is Electronic Design?**

Given specifications, we want to develop a circuit by connecting *known* electronic devices, such that the circuit meets given specifications.

“Specifications” refer to the description of the desired behaviour of the circuit.

“Known” devices are those whose behaviour can be modeled by known equations or algorithms, with known values of parameters.

# Electronic Design

Electronic Design is the process of converting  
a **behavioural description** (What happens when ..)


to

a **structural description** (What is connected to what and how ..)

After conversion to a structural description, we may need to do  
“Physical Design” which involves choosing device sizes,  
placement of blocks, routing of interconnect lines etc.

This part is already done for us in **FPGA based design**.

# Conquest over Complexity

- The main challenge for modern electronic design is that the circuits being designed these days are extremely complex.
-  While IC technology has moved at a rapid pace, capabilities of human brain have remained the same :-)
- The human mind cannot handle too many objects at the same time. So a complex design has to be broken down into a small number of 'manageable' objects.
- If each object is still too complex to handle, the above process has to be repeated recursively. This leads to hierarchical design.
- Systematic procedures have to be developed to handle complexity.



# A page out of the software designer's book

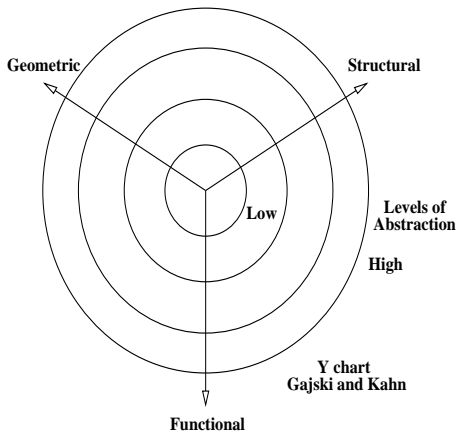
We must learn from the experience of software designers for handling complexity.

We must adopt:

- Hierarchical Design.
- Modular architecture.
- Text based, rather than pictorial descriptions.
- Re-use of existing resources

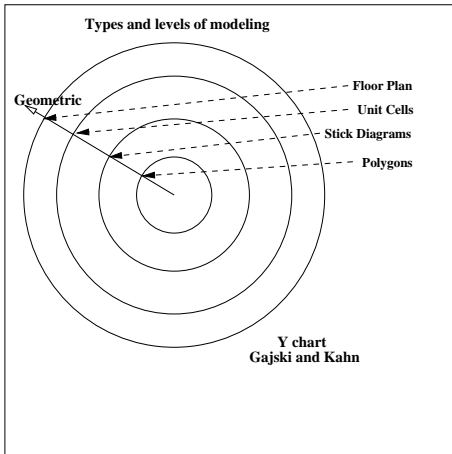
# Abstraction Levels

Types and levels of modeling



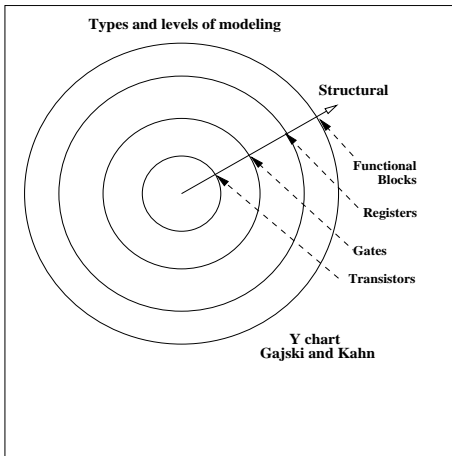
- Abstraction levels refer to functional, structural or geometric *views* of the design.
- Top down design begins with higher levels of abstraction.
- As we go to lower levels of abstraction, the level of detail goes up.
- It is advantageous to do as much work as possible at higher levels of abstraction, when the detail is low.

# Abstraction Levels: Geometric



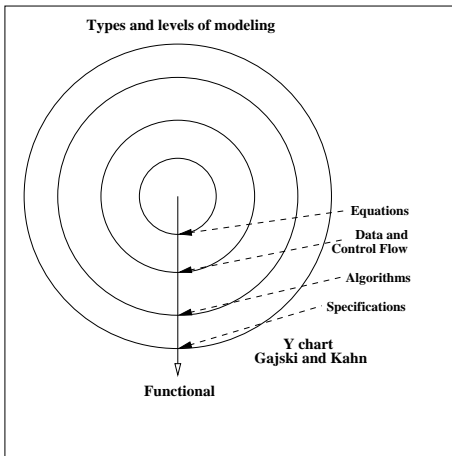
- At high levels of geometric abstraction, we view the layout as a floor plan with blocks.
- At lower levels, we look at basic cells.
- At lower levels still, we view transistors as stick diagrams.
- At the lowest level, we have to worry about all rectangles and polygons making up the layout.

# Abstraction Levels: Structural



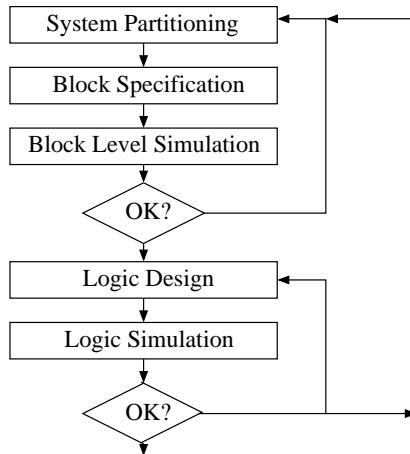
- At high levels of abstraction, we view the structure in terms of functional blocks or IP cores.
- At lower levels, we see it in terms of registers, simple blocks
- At still lower levels, we view it in terms of logic gates etc.
- At the lowest level, we have to see full details at transistor level.

# Abstraction Levels: Functional

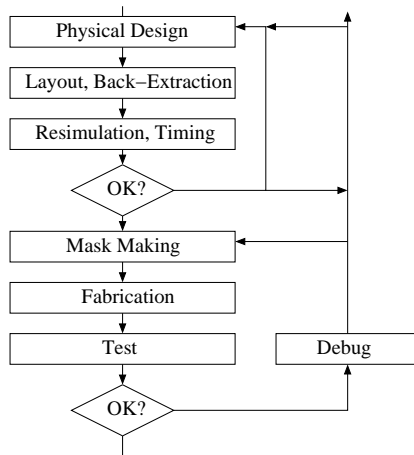


- At the top level, we have the functional specifications.
- At lower levels, we view the design in terms of protocols and algorithms.
- At still lower levels, we view it in terms of data and control flow etc.
- At the highest level of detail, we have to worry about all the governing equations at all nodes.

# Design Flow: System and logic level



# Design Flow: Physical Level



# Hierarchical Design

- The design process has to be hierarchical.
- A complex circuit is converted to a structural description of blocks which have not yet been designed - but whose behaviour can be described.
- Each of these blocks is then designed as if it was an independent design problem of lower complexity.
- This process is continued till all blocks are broken down into “known” devices.
- It is essential that any departure from proper operation is detected early - at a higher level of abstraction.
- A hardware description language must be able to simulate a system whose components have been designed to different levels of detail.



# But Hardware is different!

Hardware components are concurrent  
(all parts work at the same time).

Whereas (traditional) software is sequential -  
(executes an instruction at a time).

Description of hardware behaviour has timing as an integral part.

Traditional software is not real time sensitive.

Therefore, design of complex hardware involves many more basic concepts beyond those of programming languages.

# Hardware Description Languages

Hardware description languages need the ability to

- Describe
- Simulate at
  - Behavioural
  - Structural
  - and Mixed

level.

- and to synthesize (structure from behaviour).

# Basic HDL Concepts

- Timing
- Concurrency
- Hardware Simulation process which involves:
  - Analysis
  - Elaboration
  - and Simulation
- Simulation proceeds in two distinct phases
  - Signal update
  - Selective re-simulation

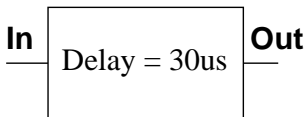
# HDL Uses

Hardware Description Languages are used for:

- Description of
  - Interfaces
  - Behaviour
  - Structure
- Test Benches
- Synthesis

# Delays

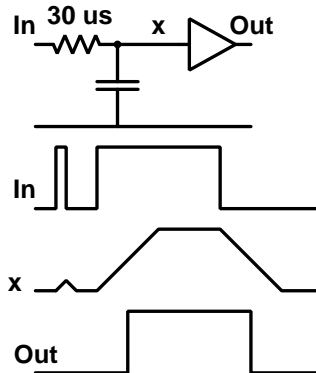
How do we describe delays?



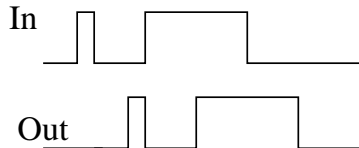
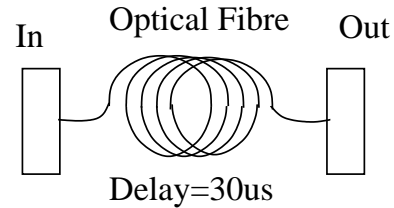
`Out <= In AFTER 30 us;`

Is this description unambiguous?

# Delay: Inertial



# Delay: Transport



# Modeling Delay

So the *same* amount of delay ( $30\ \mu\text{s}$  in our example), can result in *qualitatively* different phenomena!

We have to define two different *kinds* of delay

**Inertial Delay** is the RC kind of delay, which swallows pulses much narrower than the delay amount.

**Transport Delay** is the optical fibre kind of delay, which lets all pulses pass through irrespective of their width.

In most hardware description languages, Delays are **inertial by default**.

The delay amount is taken to be **zero** if not specified.



# Signal Assignments: Transactions

To represent real hardware, each signal assignment has to be associated with a delay.

When a value is assigned to a signal, the target signal does not acquire the assigned value immediately. The value is acquired after some delay.

Remembering that a signal is scheduled to acquire a value in the future is called a **“Transaction”**

Thus, when an assignment is made, we imply that the target signal will acquire *this value* after *so much* delay of *this type*.

# Concept of delta delay

When a transaction is placed on a signal, the default type of delay is inertial and the default amount of delay is zero.

Zero delay is implemented as a small ( $\delta$ ) delay which goes to zero in the limit.

This has scheduling implications.

Events occurring at  $t$ ,  $t + \delta$ ,  $t + 2\delta$  are all reported as having occurred at  $t$ , but are time ordered as if  $\delta$  were non zero.

# Handling Concurrency

Concurrency is handled by following an even driven architecture.

- In a concurrent system many things can happen at the same time.
- We can efficiently handle only one thing at a time,
- Therefore we need to 'control' the passage of time.
- Time is treated as a global variable. Things which happen simultaneously are handled one after the other, keeping the time value the same. Time is incremented explicitly after all events at the current time have been handled.
- Obviously, the value of the time variable represents the time during the operation of the concurrent system - and has nothing to do with the actual time taken by a computer to simulate the system.

# Hardware Simulation

Hardware simulation involves three stages:

**Analysis** Syntax of hardware description is checked and interpreted.

**Elaboration** This is a preparatory step which sets up a hierarchically described circuit for simulation.

- **Flattening the hierarchy:** For structural descriptions, components are expanded, till the circuit is reduced to an interconnection of simple components which are described behaviourally.
- **Data structures describing “sensitivity lists”** of all elemental components are built up.

**Simulation** Event driven simulation is carried out.

# Analysis

- Check for Syntax and Semantics

  - Syntax:** Grammar of the language

  - Semantics:** Meaning of the model

- Analyse each design unit separately

- Place analysed units in a working library,  
(generally in an implementation dependent internal form to  
enhance efficiency).

# Elaboration

This step ‘builds up’ a detailed circuit from a hierarchical description.

- ‘Flatten’ the design hierarchy
  - Create ports (interfaces with other blocks).
  - Create signals and processes.
  - For each instantiated component, copy the component ‘template’ to the instance.
  - Repeat recursively till we are left only with behaviourally described ‘atomic’ modules.
- The end result of elaboration is a flat collection of signal nets connected to behaviourally described modules through defined ports.

# Event Driven Simulation

- We maintain a time-ordered queue of signals which are waiting to acquire their assigned values.
- The time variable is advanced to the earliest entry in this queue.
- All signals waiting for acquiring their values *at this time* are updated.
- If this updating results in a *change* in the value of a signal, an **Event** is said to have occurred on this signal.

# Sensitivity List

During the elaboration phase, we determine which pieces of hardware are affected by (are *sensitive to*) which event.

This is called a ‘sensitivity list’

The data structure is optimized for reverse look up:

That is, given an event, one can quickly get a list of *all* hardware which is sensitive to it.

Notice that hardware could be sensitive to a particular kind of change— for example to a *rising* edge of the clock.



# The Simulation Cycle

The time variable is advanced to the earliest time entry in the time ordered queue of transactions.

**The update phase** Update *all* signals which were to acquire their values at the current time (and then delete their entry from the queue).

**Event handling phase** If the value of a signal changes due to the above update, it is said to have had an **event**. All events which resulted at the current time are handled by a scheduler.

# Scheduling

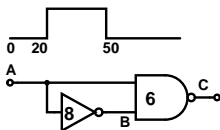
For *each* event that took place at the current time,

- We re-simulate *all* modules which are sensitive to this event.
- As a result of re-simulation, fresh transactions will be placed on various signals. These are inserted at appropriate positions in the time ordered queue.

This is done for all events which occurred at the current time.

When all events have been handled, we advance the time to the earliest entry in the time ordered transactions list and start the update phase again.

# A Simulation Example



Nodes: A,B and C

Input A, Output C

Inverter Delay: 8 units

NAND delay: 6 units

Sensitivity List

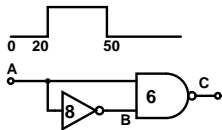
Event on A Inverter, NAND

Event on B NAND

Time ordered Transaction List:

Time	Trans.
0	A = 0
20	A = 1
50	A = 0

# A Simulation Example



At Time = 0, update A = 0.

Time	A	B	C
Initial	X	X	X
0	0	X	X

**A has an event.**

Inverter and NAND are sensitive to A.

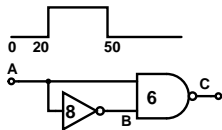
Initial Time	Trans.
0	A = 0
20	A = 1
50	A = 0

Re-evaluate:

Inverter: B  $\rightarrow$  1 at 8;  
NAND: C  $\rightarrow$  1 at 6

After Re-sim Time	Trans.
6	C = 1
8	B = 1
20	A = 1
50	A = 0

# A Simulation Example

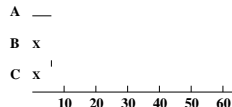


At Time = 6, update C = 1.

Time	A	B	C
0	0	X	X
6	0	X	1

**C has an event.**

No module is sensitive to C.



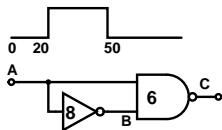
Initial Time	Trans.
6	C = 1
8	B = 1
20	A = 1
50	A = 0

Re-evaluate:

None Required

After Re-sim Time	Trans.
8	B = 1
20	A = 1
50	A = 0

# A Simulation Example

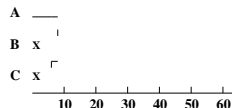


At Time = 8, update B = 1.

Time	A	B	C
6	0	X	1
8	0	1	1

**B has an event.**

Only NAND is sensitive to B.



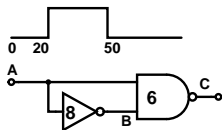
Initial Time	Trans.
8	B = 1
20	A = 1
50	A = 0

Re-evaluate:

NAND: C  $\rightarrow$  1 at 14

After Re-sim Time	Trans.
14	C = 1
20	A = 1
50	A = 0

# A Simulation Example

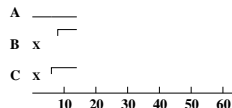


At Time = 14, update  $C = 1$ .

Time	A	B	C
8	0	1	1
14	0	1	1

There is no event.

No Sensitivity is triggered.



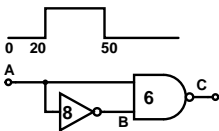
Initial Time	Trans.
14	$C = 1$
20	$A = 1$
50	$A = 0$

Re-evaluate:

None Required

After Re-sim Time	Trans.
20	$A = 1$
50	$A = 0$

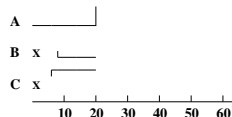
# A Simulation Example



At Time = 20, update A = 1.

Time	A	B	C
14	0	1	1
20	1	1	1

A has an event.

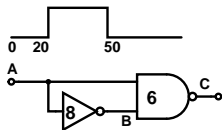


Inverter and NAND are sensitive to A.

Initial		Re-evaluate:	After Re-sim	
Time	Trans.		Time	Trans.
20	A = 1	Inverter: B → 0 at 28; NAND: C → 0 at 26	26	C = 0
50	A = 0		28	B = 0
			50	A = 0



# A Simulation Example

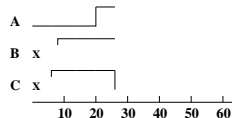


At Time = 26, update C = 0.

Time	A	B	C
20	1	1	1
26	1	1	0

**C has an event.**

No module is sensitive to C



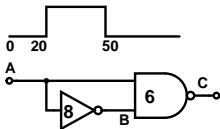
Initial Time	Trans.
26	C = 0
28	B = 0
50	A = 0

Re-evaluate:

No update is required.

After Re-sim Time	Trans.
28	B = 0
50	A = 0

# A Simulation Example

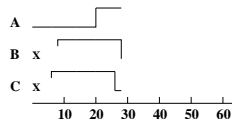


At Time = 28, update B = 0.

Time	A	B	C
26	1	1	0
28	1	0	0

**B has an event.**

Only NAND is sensitive to B.



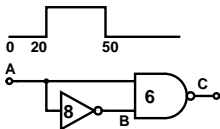
Initial Time	Trans.
28	B = 0
50	A = 0

Re-evaluate:

NAND:  $C \rightarrow 1$  at 34

After Re-sim Time	Trans.
34	C = 1
50	A = 0

# A Simulation Example

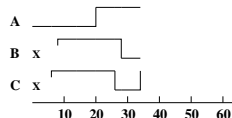


At Time = 34, update C = 1.

Time	A	B	C
28	1	0	0
34	1	0	1

**C has an event.**

No module is sensitive to C.



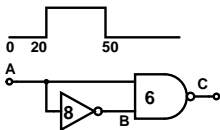
Initial Time	Trans.
34	C = 1
50	A = 0

Re-evaluate:  
No evaluation needed.

After Re-sim Time	Trans.
50	A = 0

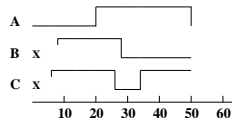
# A Simulation Example

At Time = 50, update A = 0.



Time	A	B	C
34	1	0	1
50	0	0	1

A has an event.



Inverter and NAND are sensitive to  
A.

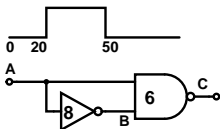
Initial Time	Trans.
50	A = 0

Re-evaluate:

Inverter: B  $\rightarrow$  1 at 58;  
NAND: C  $\rightarrow$  1 at 56

After Re-sim Time	Trans.
56	C = 1
58	B = 1

# A Simulation Example

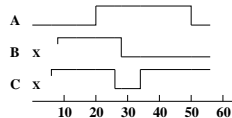


At Time = 56, update C = 1.

Time	A	B	C
50	0	0	1
56	0	0	1

There is no event

No Sensitivity is triggered.

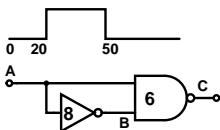


Initial Time	Trans.
56	C = 1
58	B = 1

Re-evaluate:  
  
No re-evaluation  
required.

After Re-sim Time	Trans.
58	B = 1

# A Simulation Example

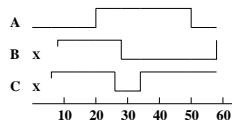


At Time = 58, update B = 1.

Time	A	B	C
56	0	0	1
58	0	1	1

**B has an event**

Only NAND is sensitive to B

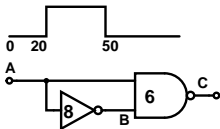


Initial Time	Trans.
58	B = 1

Re-evaluate:  
NAND: C  $\rightarrow$  1 at 64

After Re-sim Time	Trans.
64	C = 1

# A Simulation Example

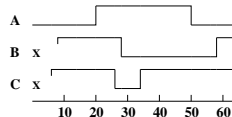


At Time = 64, update C = 1.

Time	A	B	C
58	0	1	1
64	0	1	1

There is no event

No sensitivity is triggered.



Initial Time	Trans.
64	C = 1

Re-evaluate:

No re-evaluation required.

After Re-sim

Time ordered list is empty.

# Scheduling for Delay types

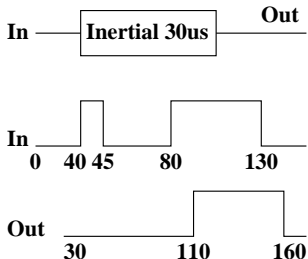
What do we do if there is more than one transaction waiting for the same signal?

**Inertial Delay** A transaction scheduled for later time results in deletion of waiting transactions for a different value on the same signal.

**Transport Delay** All transactions are retained and signal assignments made at their respective times.



# Inertial Delay Example



Time Transaction

0 In := 0

40 In := 1

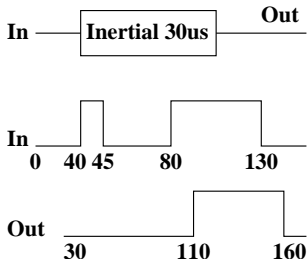
45 In := 0

80 In := 1

130 In := 0

# Inertial Delay Example

## Time Transaction



30 out := 0

40 In := 1

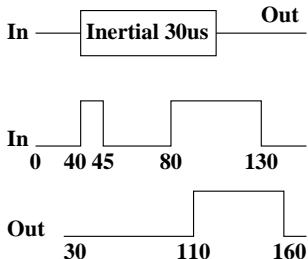
45 In := 0

80 In := 1

130 In := 0

# Inertial Delay Example

## Time Transaction



40 In := 1

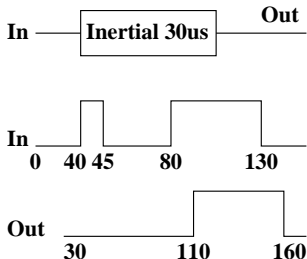
45 In := 0

80 In := 1

130 In := 0

# Inertial Delay Example

## Time Transaction



45 In := 0

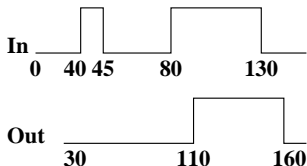
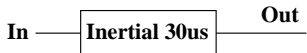
70 Out := 1

80 In := 1

130 In := 0

# Inertial Delay Example

## Time Transaction



70 Out := 1

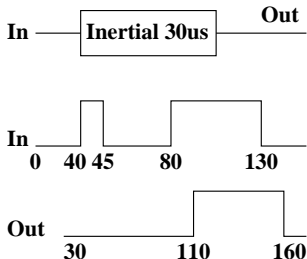
75 Out := 0

80 In := 1

130 In := 0

# Inertial Delay Example

## Time Transaction



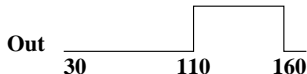
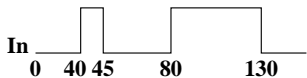
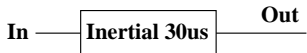
75 Out := 0

80 In := 1

130 In := 0

# Inertial Delay Example

## Time Transaction

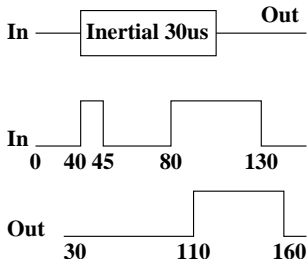


80 In := 1

130 In := 0

# Inertial Delay Example

## Time Transaction



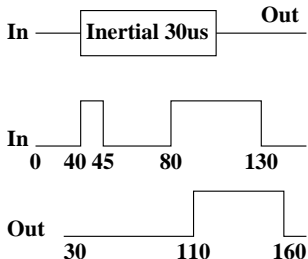
110 Out := 1

130 In := 0



# Inertial Delay Example

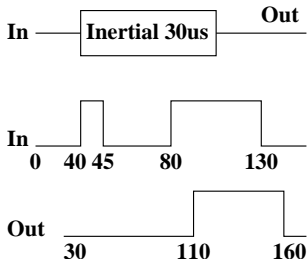
## Time Transaction



130 In := 0

# Inertial Delay Example

## Time Transaction



160 Out := 0

# Concurrent Descriptions

- The order of placing 'concurrent' descriptions in a hardware description language is immaterial.
- As seen in the example described earlier, each concurrent block is handled when its 'sensitivity' is struck, wherever it is placed in the overall description.
- So what defines the limits of a 'concurrent block'?
- If it is a single line, there is no problem.
- If the description of a concurrent block needs multiple lines, How are these lines to be executed?

# Multi-line concurrent descriptions

- A multi-line concurrent block has to be executed completely when its sensitivity is struck.
- Therefore, the multi-line description of a complex concurrent block must be executed *sequentially*, line by line.
- A hardware description language must therefore provide a syntax to distinguish sequential parts from concurrent parts.  
(After all, a single line of description could be a stand-alone concurrent description or part of a multi-line sequential code).
- Multi-line descriptions of hardware blocks are concurrent outside and sequential inside!

# Sequential Descriptions

Describing hardware by sequential code raises a problem!  
What happens when the sequential description reaches its end?

- Hardware blocks are perpetual objects. These cannot 'terminate' like software routines.

# Sequential Descriptions

Describing hardware by sequential code raises a problem!  
What happens when the sequential description reaches its end?

- Hardware blocks are perpetual objects. These cannot ‘terminate’ like software routines.
- We can make sequential descriptions perpetual by adding the convention that a sequential description loops back to its beginning when it reaches its end.

# Sequential Descriptions

Describing hardware by sequential code raises a problem!  
What happens when the sequential description reaches its end?

- Hardware blocks are perpetual objects. These cannot 'terminate' like software routines.
- We can make sequential descriptions perpetual by adding the convention that a sequential description loops back to its beginning when it reaches its end.

This, however, leads to yet another problem!



# Suspending endless loops

An endless loop will never terminate.  
Then how can we handle the next event?

Indeed, when can we advance the time variable?



# Suspending endless loops

An endless loop will never terminate.  
Then how can we handle the next event?

Indeed, when can we advance the time variable?

The convention should therefore be that when a sequential description ends, execution will loop back to the beginning, *and execution of the loop will be suspended here!*

The suspended loop will restart only when the sensitivity of this block is struck again.

Now we can handle multiple blocks waiting to be handled at any given time.

We handle each block whose sensitivity has been triggered, till it is suspended.

Then we handle the next block and so on, till all blocks have been done.

Now we update the time to the next earliest entry in the time order queue and go through the next signal update - event handling cycle.

# Hardware Description Languages

This ends

The first part of the lecture series on

## HARDWARE DESCRIPTION LANGUAGES

### Fundamental Concepts