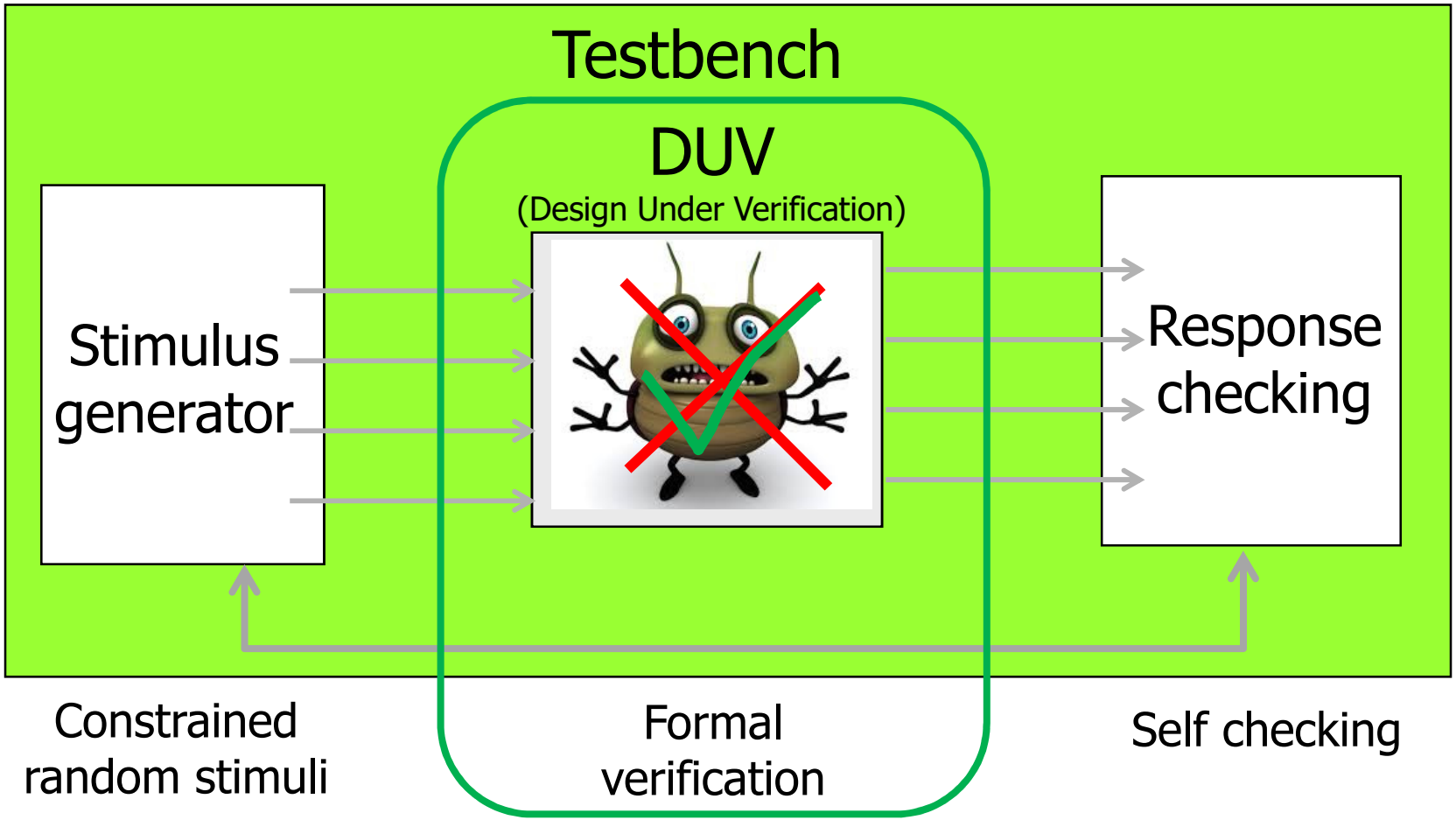
26 years ago…

*The **Pentium FDIV bug** was a **bug in the Intel P5 Pentium floating point unit** (FPU). Certain floating point division operations performed with these processors would produce **incorrect results**. According to Intel, there were **a few missing entries in the lookup table** used by the divide operation algorithm.*

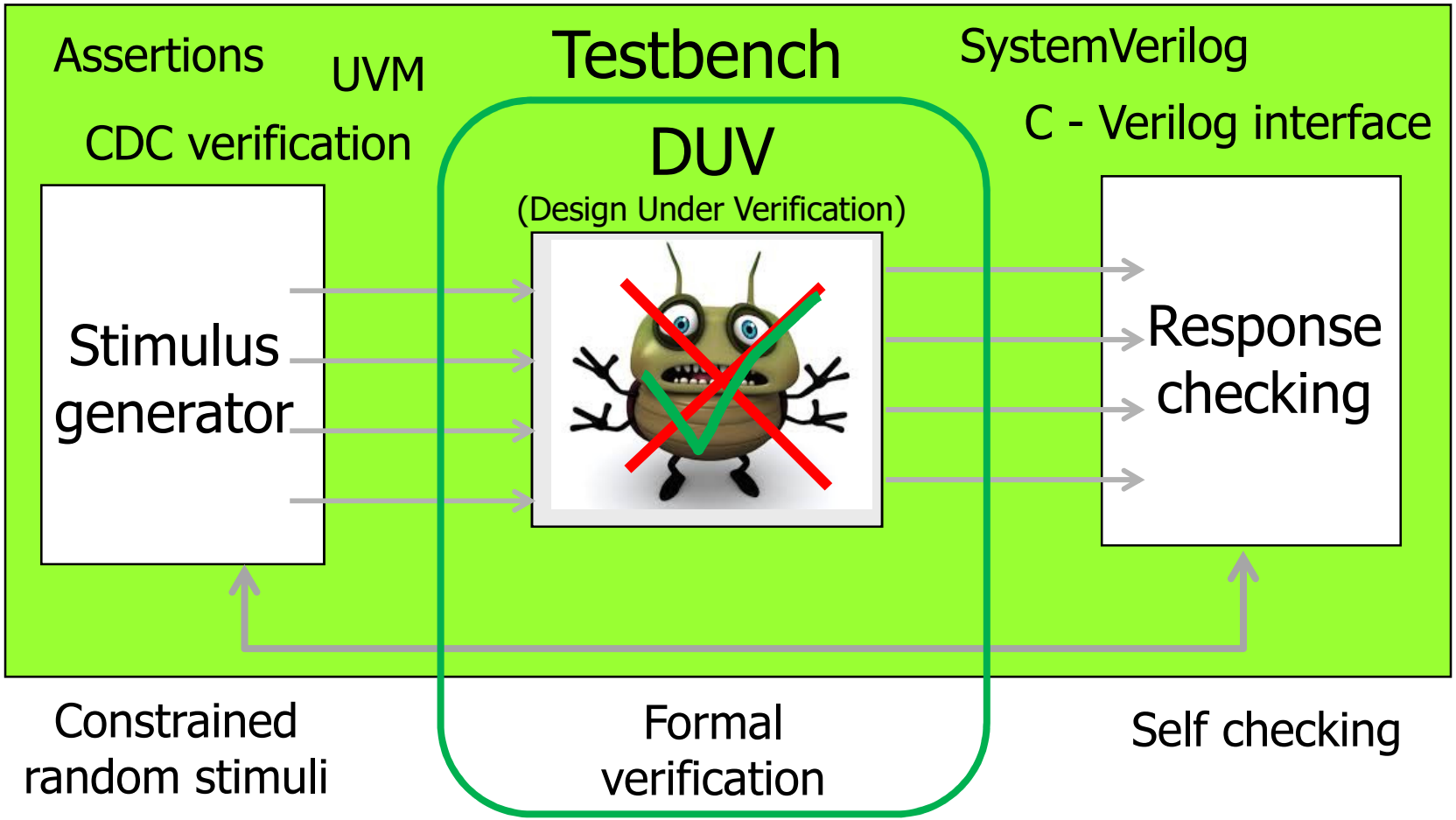
*Although encountering the flaw was **extremely rare** in practice (Byte magazine estimated that **1 in 9 billion floating point divides** with random parameters would produce inaccurate results), both the flaw and Intel's initial handling of the matter were heavily criticized. Intel ultimately recalled the defective processors.*

Source:
http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Pentium_FDIV_bug.html

Design verification
of RTL digital models



Design verification
of RTL digital models

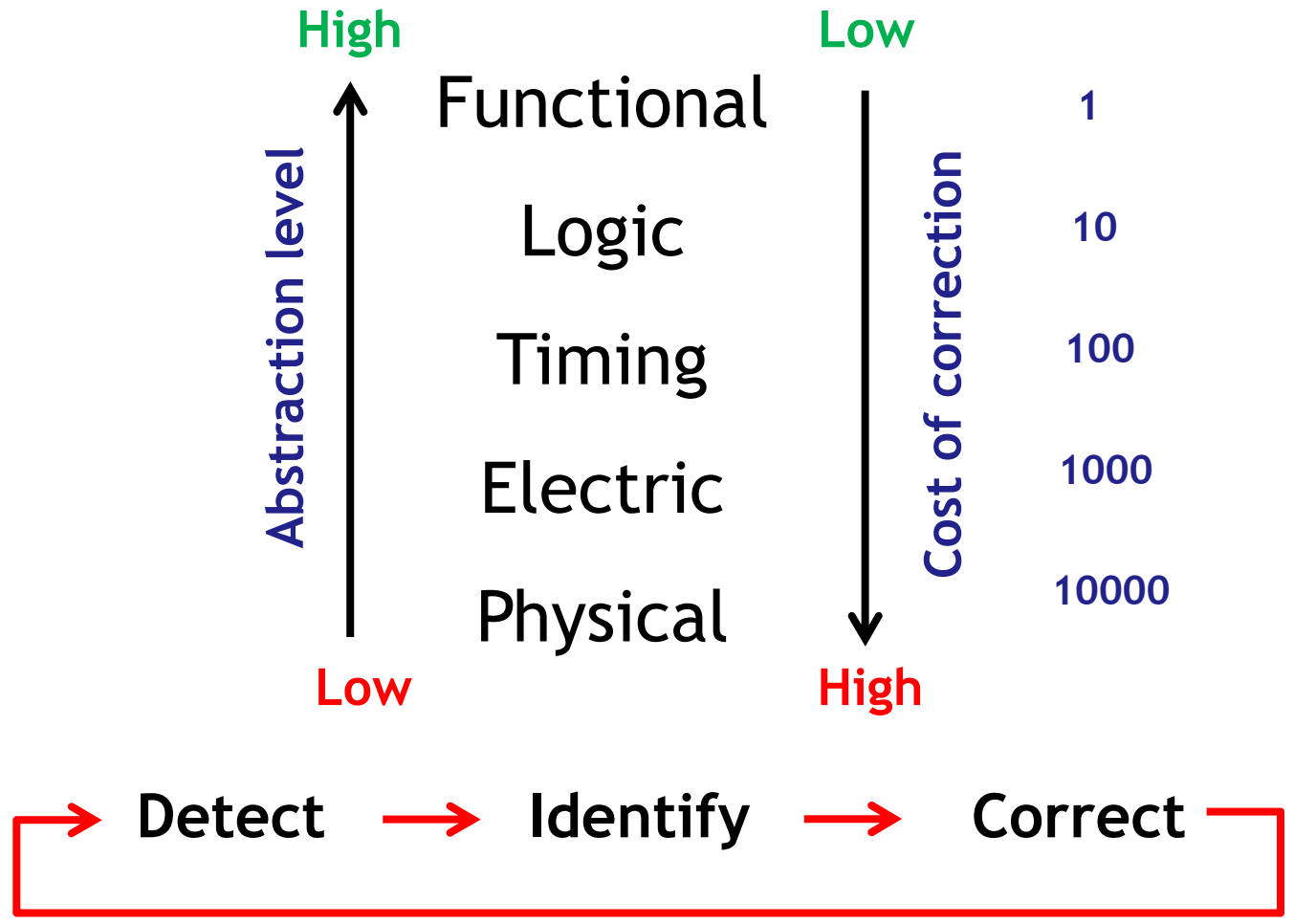


Design verification

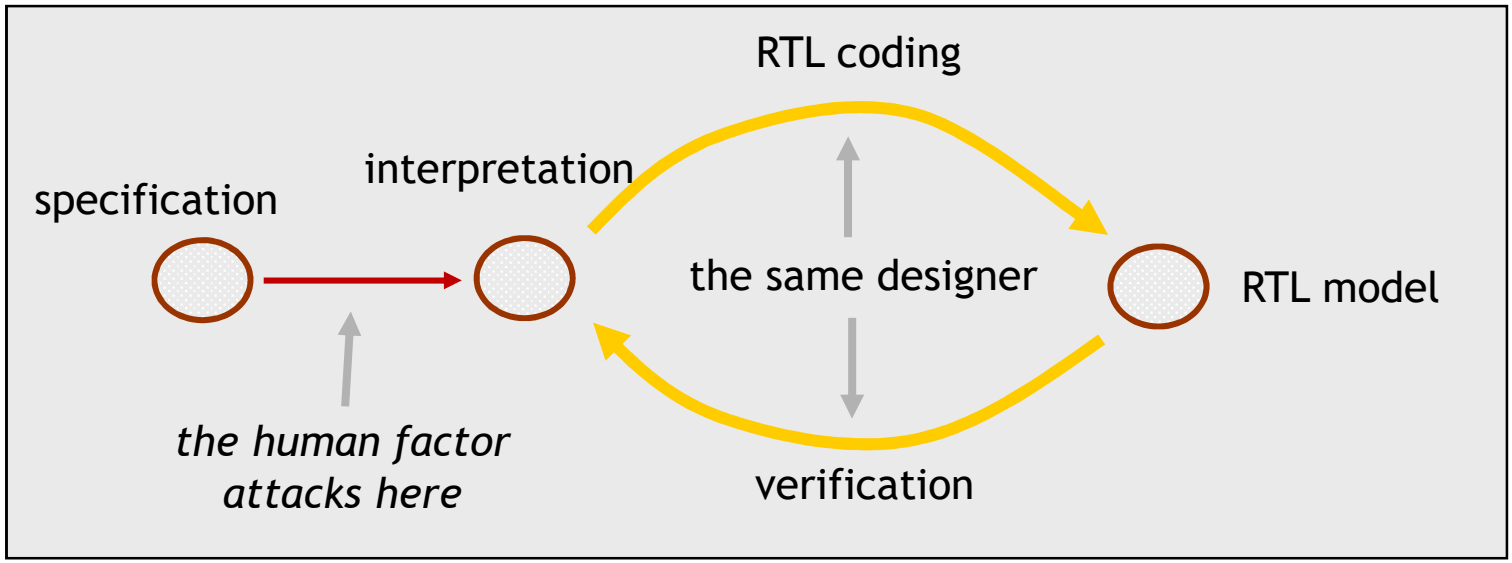
from specification to RTL signoff

- Design verification
  - *Prove* your RTL design meets the specification (functional requirements)
  - Along all abstraction levels: Algorithmic, RTL, Logic, Timing, Electrical, Physical
  - Design verification is different from (physical) circuit testing
- Verification is in the critical path
  - Typically more than 70% of design resources are allocated for verification
  - Testbenches may represent the major part of design code
  - Use appropriate verification tools and methodologies (simulation vs. formal check)
  - Create self-checking testbenches whenever possible
- Build a verification plan
  - Design a set of tests to check against the specifications
  - Establish coverage metrics and build tests to “close” any coverage “holes”
    - Coverage “*closure*”
  - Use tools for planning the verification

Verify as soon as possible



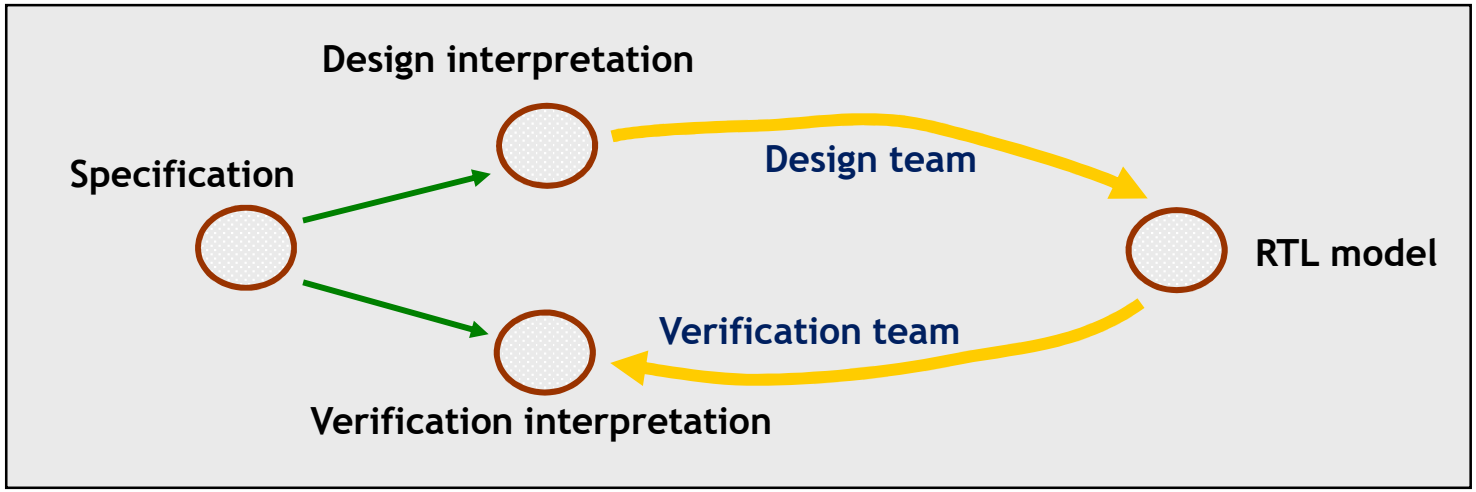
The human factor



Verification is done against ***one interpretation*** of a specification

Important to reduce the human factor

- Independent design and verification teams
  - Both use the same specification but interpretations may be different
- Automate the verification process as much as possible
  - Build/re-use self-checking test benches
  - Use test benches at higher abstraction levels (ex. System Verilog, C)
  - Whenever possible use formal verification methods



Verification techniques

- Logic simulation
  - Dynamic analysis of the logic behaviour, may include timing information
  - Requires building a testbench, generate test stimuli and analyze the results
  - Pass/fail analysis done manually (eg. waveforms) or with self-checking procedures
  - Can only show that the design under verification satisfies a (limited) set of tests
    - In practice it is impossible to run exhaustive verifications
    - A simple circuit with only 256 flops has more than 10^{77} states = 10^{57} years@1ps/state
  - The set of verification tests must cover all the specifications and all the RTL code
  - More suitable for dataflow and arithmetic datapaths
  - **Cycle-based simulators** simulate only the behaviour at each clock cycle transition
  - **Event-driven simulation** simulates discrete events (toggles) propagating through the DUV
  - A good testbench must…
    - Avoid false negative and false positive results (report as pass/fail **if and only if** pass/fail)
    - Be deterministic and repeatable, even if using random test vectors, across diferente sims and OSs
  - … and should also be re-usable, maintainable and easy to debug

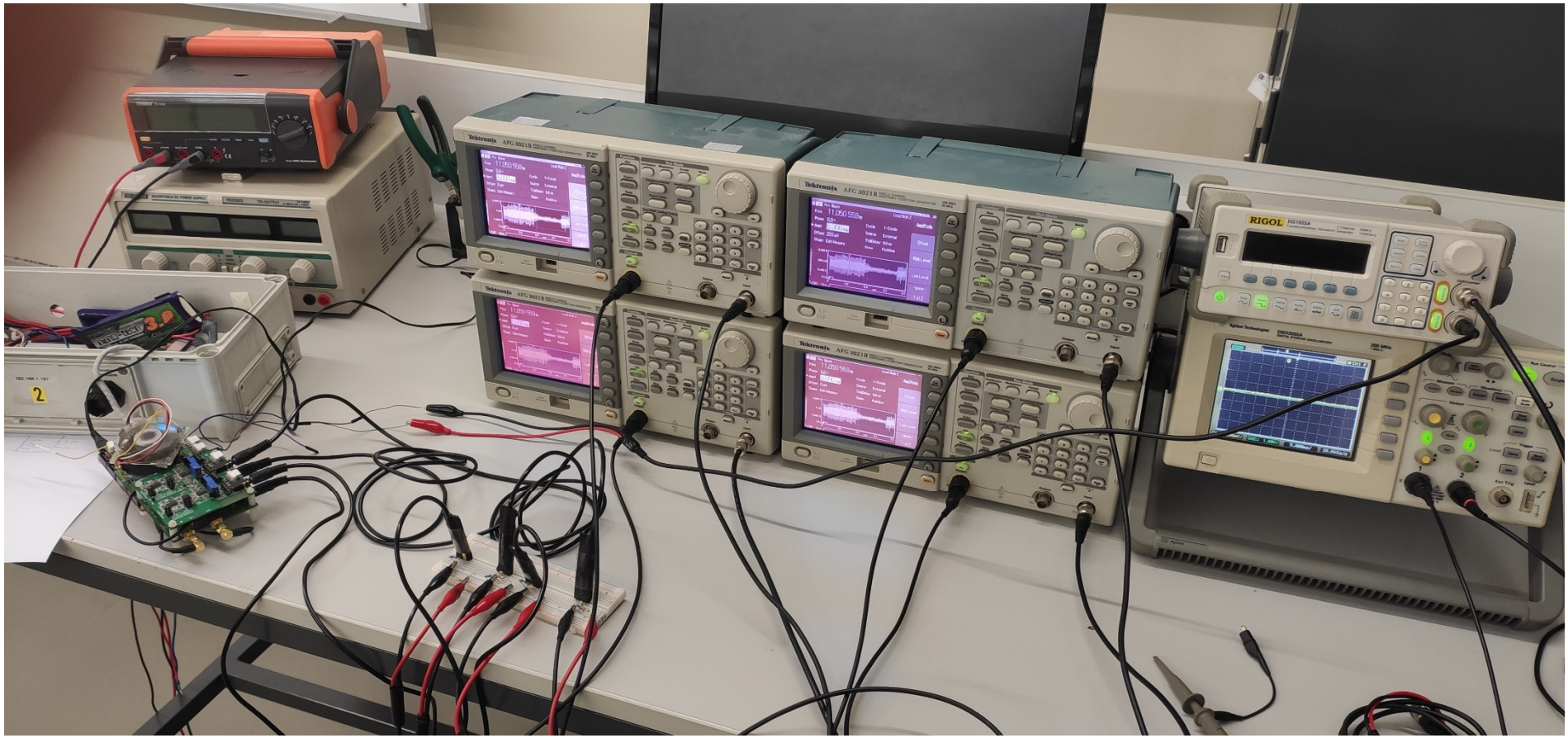
Verification techniques

- Formal verification
  - Do not explicitly use test benches and test vectors (stimuli)
  - Analyze the logic equations and the state diagrams defined by a RTL model or a logic circuit
  - More suitable for control dominated systems, data transport interfaces
- Formal model equivalence checking
  - Verifies whether two models are equivalent
  - The netlist created by synthesis is formally equivalent to the RTL model?
  - Differences due to bad coding, logic optimization (using of don't cares), scan-chain insertion
- Formal model property checking
  - Verify if design properties are satisfied / unsatisfied
  - Automatically extracted properties
    - Automatically detect unreachable code (or *dead code*), unreachable states, deadlock conditions, floating buses, multiple drivers to 3-state buses
  - User-defined properties using assertions / assumptions:
    - "*at every clock cycle the signal **Abus[7:0]** cannot be larger than 139*"
    - "*if **start_en** is 1 at this clock cycle, **ready** will be 1 between 6 and 17 clock cycles later*"

Verification techniques

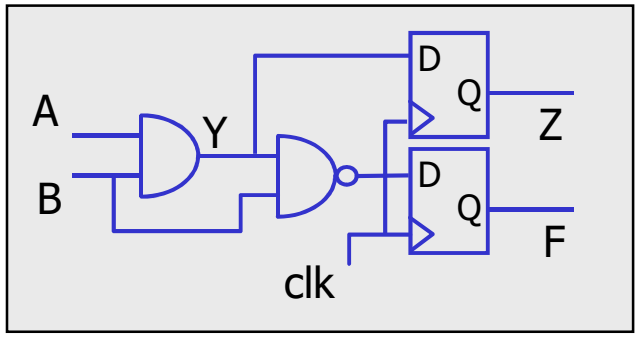
- In-circuit verification (or hardware prototyping/emulation)
  - Implement (part of) the design in a FPGA-based platform
  - Fast enough to run software in embedded processors at a fraction of speed
  - Necessary to validate a design with stimuli coming from real-time systems
  - Co-verification with logic/timing/electric simulators for specific parts of the DUV
  - Low observability of internal nodes, low controllability
    - If needed, DUV must be instrumented to gain access to internal registers
  - What can be verified?
    - In general the FPGA technology is not the target implementation technology
    - Timing of logic cells is different, synthesis optimization is not the same as for the target technology
    - Verification of the cycle-based behavior of the DUV, at clock speed
  - What cannot be verified ?
    - Timing related specifications
    - Assynchronous behavior,
    - Clock Domain Crossing (CDC)

Hardware prototyping/emulation



Logic simulation

- Compiled code simulation
  - Compile the RTL code to an executable program
  - Executes a simulation cycle-based for clocked synchronous systems
  - Combinational blocks between regs are compiled to native CPU code
  - Generates simulation data for posterior analysis (waveforms, text reports,…)
  - Example:



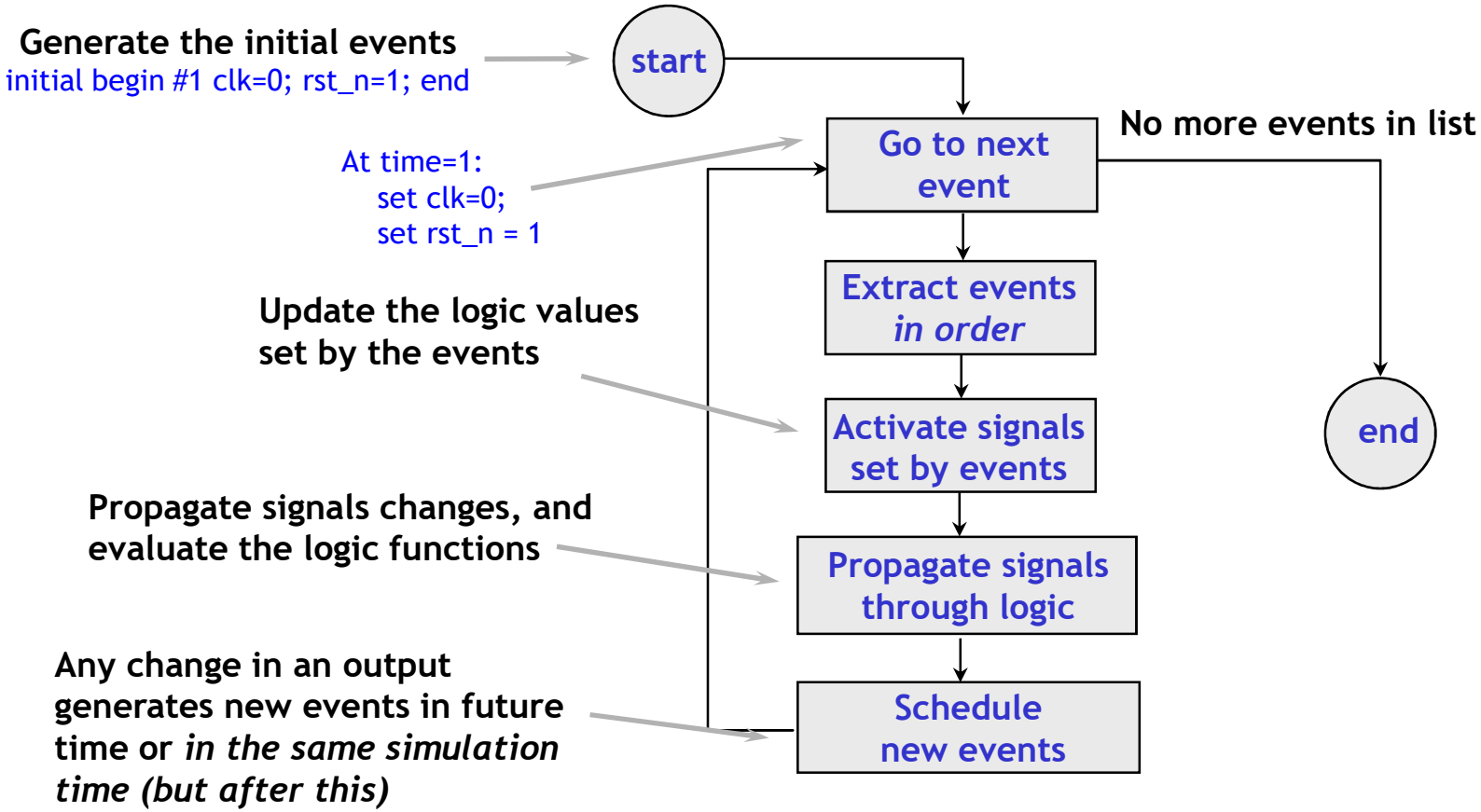
For each clock transition

```
MOV AL,A
AND AL,B
MOV Y,AL
MOV AL,B
AND AL,Y
MOV AH,AL
NOT AL
MOV Z,AH
MOV F,AL
```

Logic simulation

- Event-driven simulation
  - Discrete time simulation
    - Generate events (signal transitions generated by the testbench)
    - Activate events (what is the next signal transition and when does it happen?)
    - Propagate events through the logic circuit models (evaluate logic functions)
    - Schedule new events for future time (using the timing models)
  - Allows the simulation of the logic timing
    - Simulates asynchronous events
    - Multi-clock domain systems
    - Clock domain crossing analysis

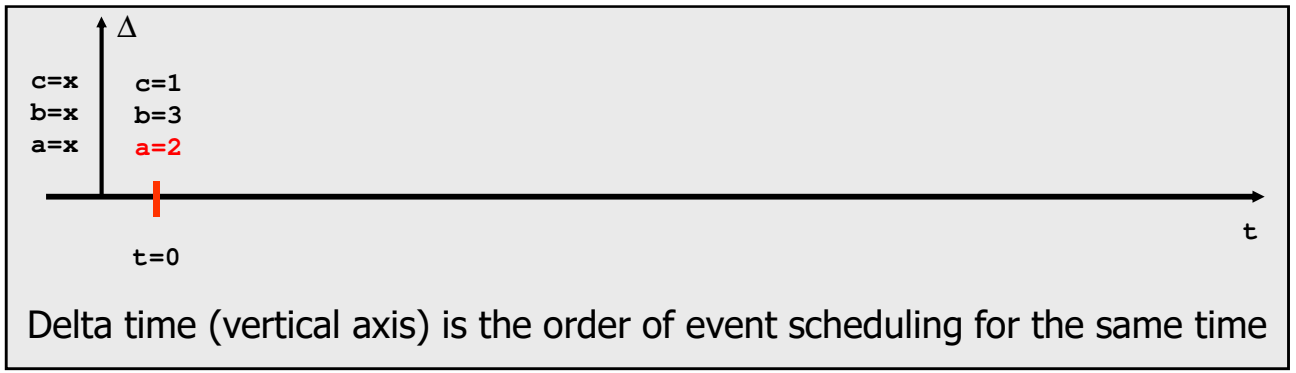
Event-driven simulation



Event-driven simulation

```
initial
begin
  a=2; b=3; c=1;
  #25 c=2;
end

always @*
begin
  #5  z=a+8;
  #10 y=b+c;
  #4  k=z+y;
end
```

Event list

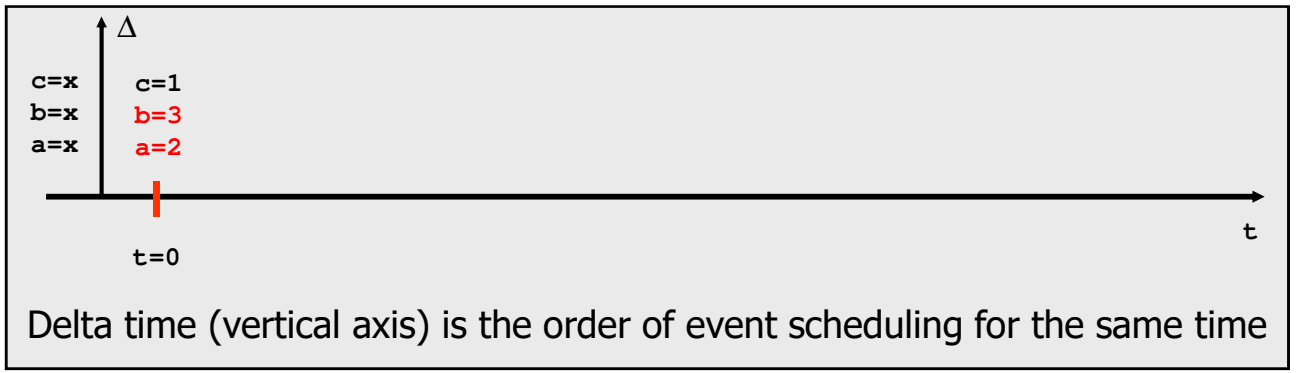


Delta time (vertical axis) is the order of event scheduling for the same time

Event-driven simulation

```
initial
begin
  a=2; b=3; c=1;
  #25 c=2;
end

always @*
begin
  #5  z=a+8;
  #10 y=b+c;
  #4  k=z+y;
end
```

Event list

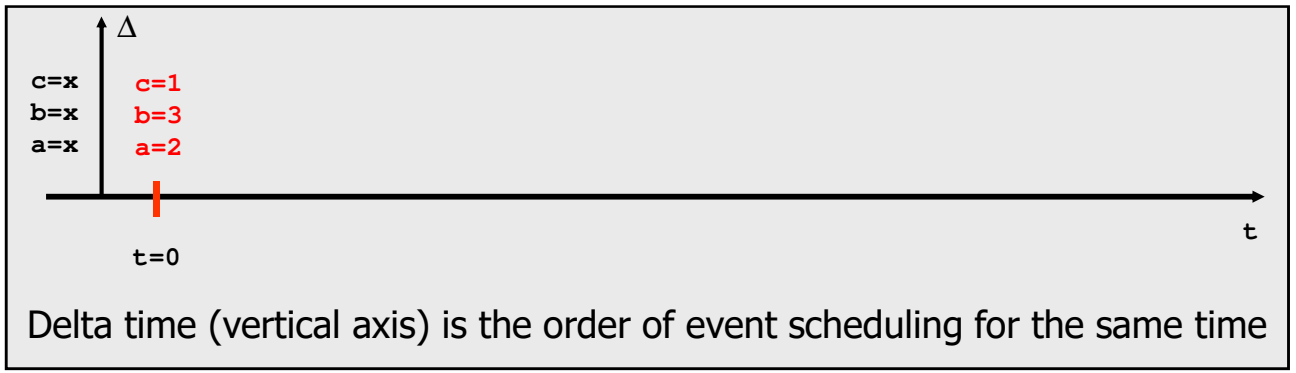


Delta time (vertical axis) is the order of event scheduling for the same time

Event-driven simulation

```
initial
begin
  a=2; b=3; c=1;
  #25 c=2;
end

always @*
begin
  #5  z=a+8;
  #10 y=b+c;
  #4  k=z+y;
end
```

Event list

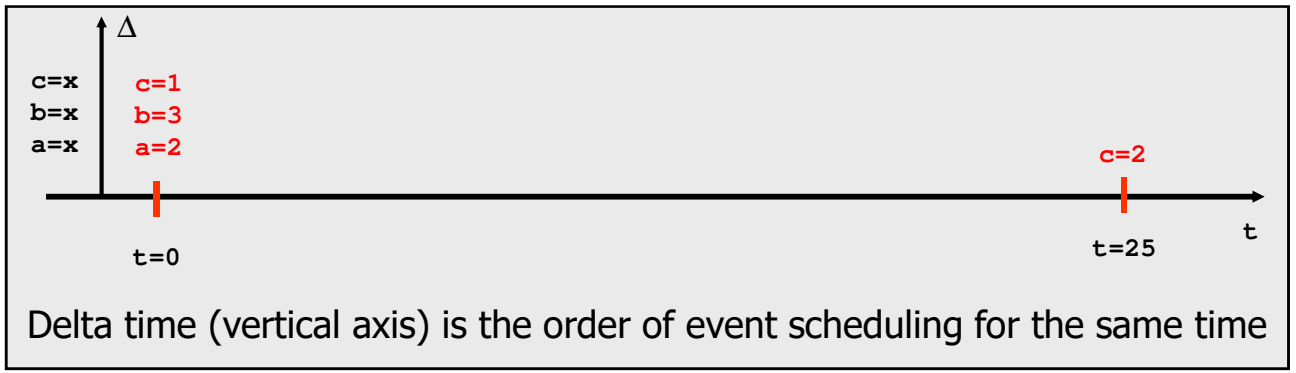


Delta time (vertical axis) is the order of event scheduling for the same time

Event-driven simulation

```
initial
begin
  a=2; b=3; c=1;
  #25 c=2;
end

always @*
begin
  #5  z=a+8;
  #10 y=b+c;
  #4  k=z+y;
end
```

Event list

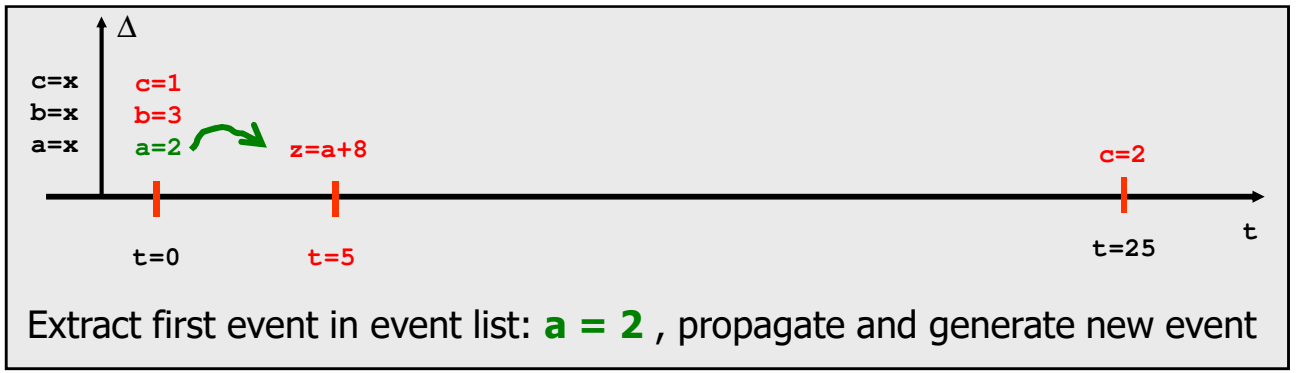


Delta time (vertical axis) is the order of event scheduling for the same time

Event-driven simulation

```
initial
begin
  a=2; b=3; c=1;
  #25 c=2;
end

always @*
begin
  #5  z=a+8;
  #10 y=b+c;
  #4  k=z+y;
end
```

Event list

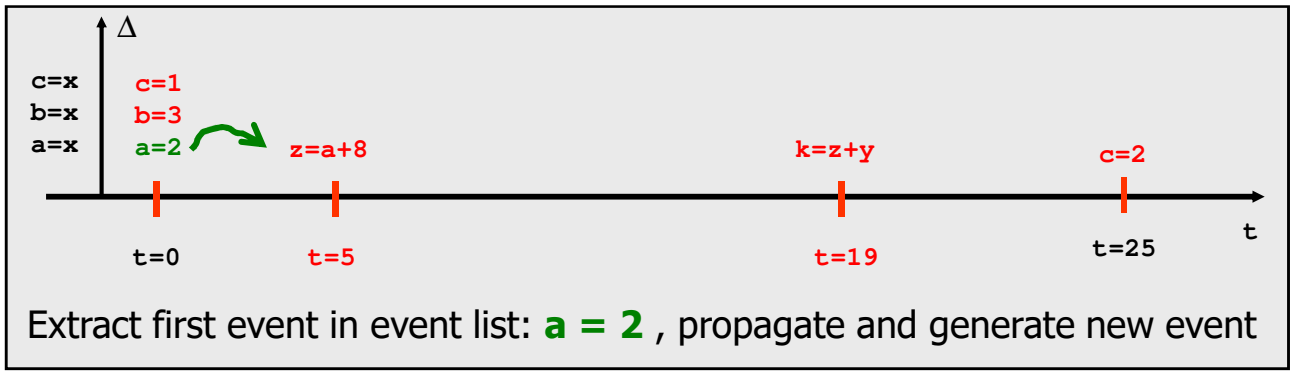


Extract first event in event list: **a = 2** , propagate and generate new event

Event-driven simulation

```
initial
begin
  a=2; b=3; c=1;
  #25 c=2;
end

always @*
begin
  #5  z=a+8;
  #10 y=b+c;
  #4  k=z+y;
end
```

Event list

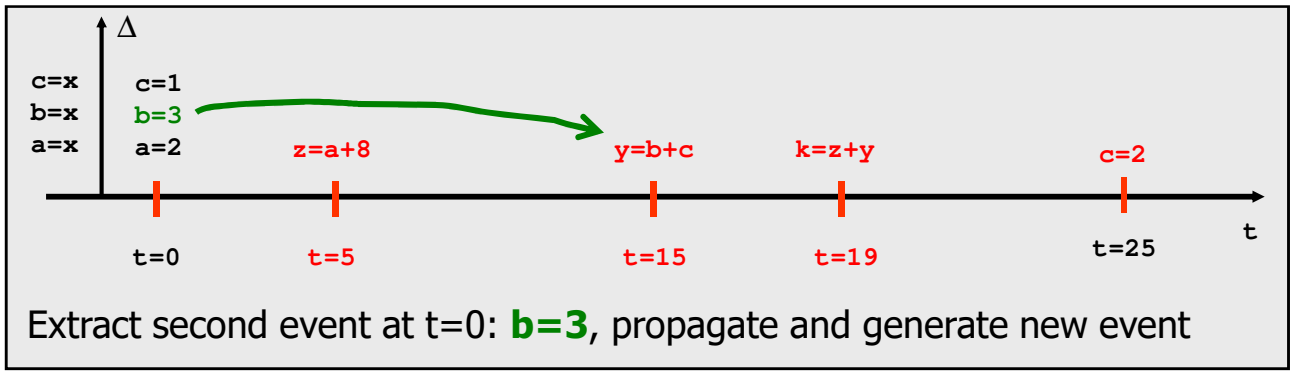


Extract first event in event list: **a = 2** , propagate and generate new event

Event-driven simulation

```
initial
begin
  a=2; b=3; c=1;
  #25 c=2;
end

always @*
begin
  #5  z=a+8;
  #10 y=b+c;
  #4  k=z+y;
end
```

Event list

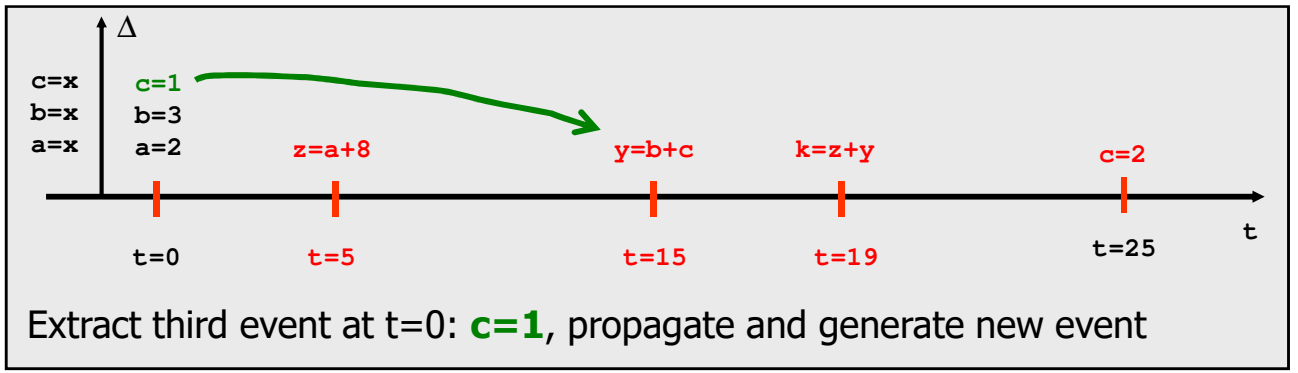


Extract second event at t=0: **b=3**, propagate and generate new event

Event-driven simulation

```
initial
begin
  a=2; b=3; c=1;
  #25 c=2;
end

always @*
begin
  #5  z=a+8;
  #10 y=b+c;
  #4  k=z+y;
end
```

Event list

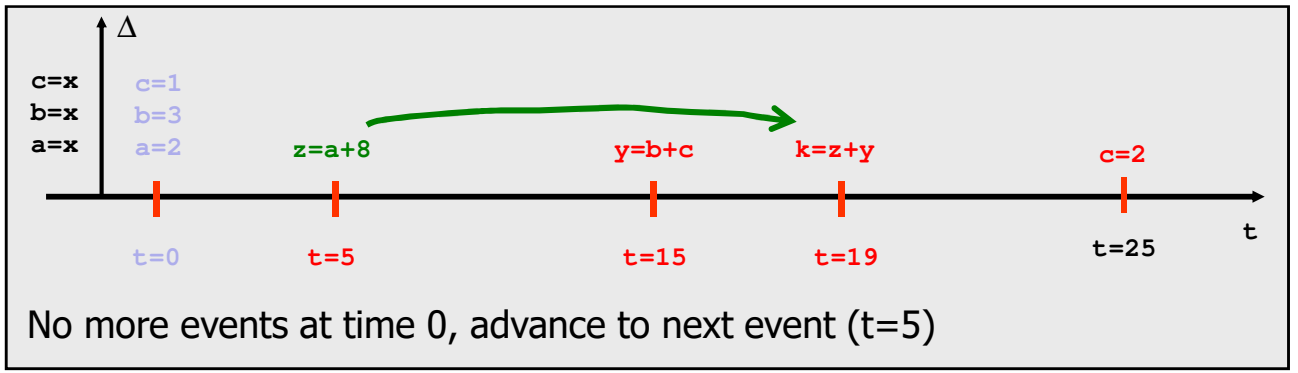


Extract third event at t=0: **c=1**, propagate and generate new event

Event-driven simulation

```
initial
begin
  a=2; b=3; c=1;
  #25 c=2;
end

always @*
begin
  #5  z=a+8;
  #10 y=b+c;
  #4  k=z+y;
end
```

Event list

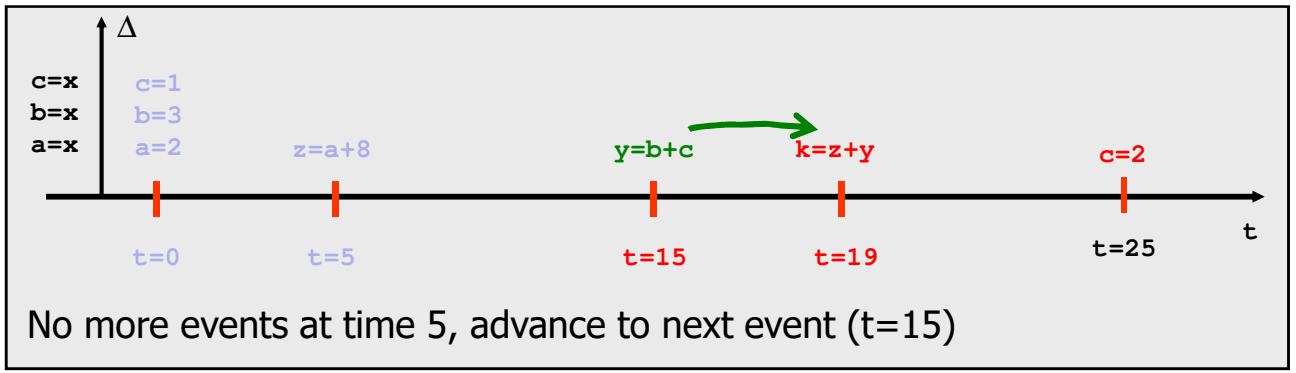


No more events at time 0, advance to next event (t=5)

Event-driven simulation

```
initial
begin
  a=2; b=3; c=1;
  #25 c=2;
end

always @*
begin
  #5  z=a+8;
  #10 y=b+c;
  #4  k=z+y;
end
```

Event list

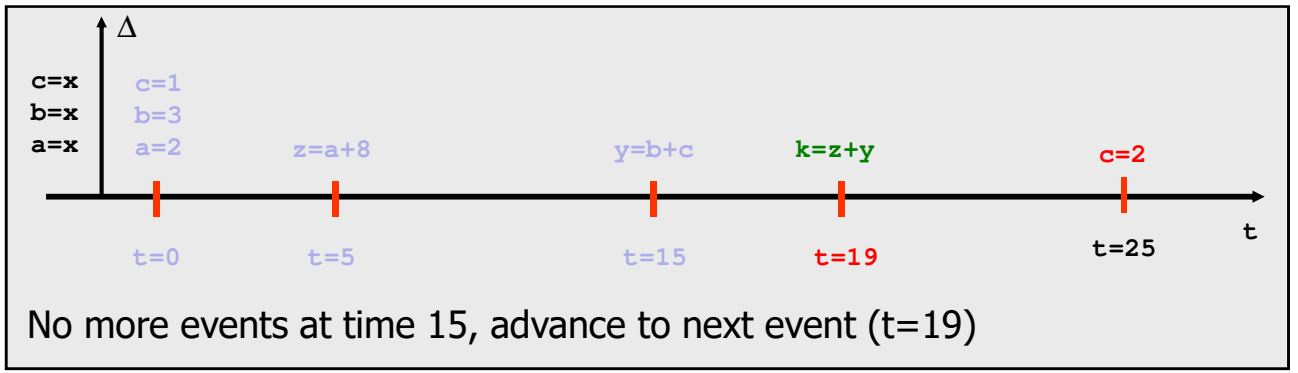


No more events at time 5, advance to next event (t=15)

Event-driven simulation

```
initial
begin
  a=2; b=3; c=1;
  #25 c=2;
end

always @*
begin
  #5  z=a+8;
  #10 y=b+c;
  #4  k=z+y;
end
```

Event list

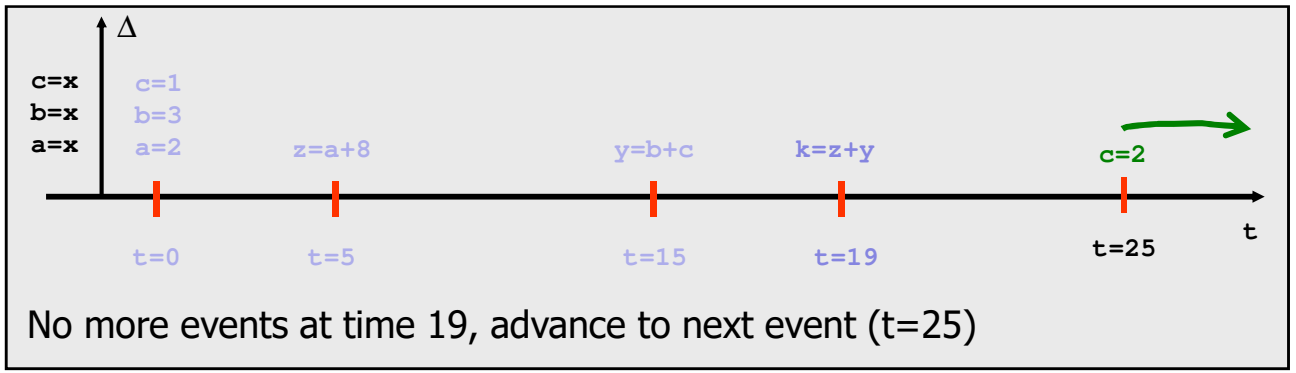


No more events at time 15, advance to next event (t=19)

Event-driven simulation

```
initial
begin
  a=2; b=3; c=1;
  #25 c=2;
end

always @*
begin
  #5  z=a+8;
  #10 y=b+c;
  #4  k=z+y;
end
```

Event list



No more events at time 19, advance to next event (t=25)

The story about the = and <= assignments

within inital and always statements

- **Remember:**
  - **Within an `initial` or `always` statement all the lefthand identifiers in assignment operations must be of type `reg`**
- **Blocking assignment (=)**

```
r_a = a + b;
r_b = r_a + c;
```

  - **The target reg `r_a` is loaded with the result before evaluating the next expression**
  - **The next expression (`r_a+c`) uses the new value already loaded to `r_a`**
- **Non-blocking assignment (<=)**

```
r_a <= a + b;
r_b <= r_a + c;
```

  - **Both target registers `r_a` and `r_b` are loaded at the same time**
  - **The value of `r_a` used to evaluate (`r_a+c`) is the previous value, before being updated with the result of the previous expression**

The story about the = and <= assignments

within inital and always statements

- **Within an `always @posedge(clk…)`**
  - **The left hand side identifiers in the assignment operations represent registers that are loaded synchronously with the clock transition**
  - **The right hand side expression uses registers (or wires connected to registers) that will also change their values in the same clock edge**
  - **When the clock triggers, the destination registers are loaded with the values present at that instant at their D inputs**
  - **These values are the results of the right hand side expressions that use the contents of the other regs before the clock**
  - **The non-blocking assignments model correctly this behavior: when the clock triggers, all the expressions activated by that event (the clock transition) are evaluated with the current values of the registers and their results are only loaded to the destination registers after all the expressions are evaluated.**

```
r_a <= a + b;
r_b <= r_a + c;
```

Verilog system tasks (for testbenches)

- **`$monitor(...)`**
  - Only one $monitor() process per testbench
- **`$time`**
  - Current simulation time
- **`$display(...), $strobe(…)`**
  - Similar to a printf(), $strobe uses the values *after* update
- **`$stop`**
  - Suspends the simulation (breakpoint)
- **`$finish`**
  - Finishes the simulation (closes the simulator application)
- **`$random`**
  - Returns a random integer (32 bit, signed)
- **`$fopen, $fclose, $fwrite, $fread, $readmem,…`**
  - File handling

Verilog

time scale, timing resolution

```
reg clk
parameter CYCLE=10;

initial clk = 1'b0;

always
begin
 #(CYCLE/2); clk = ~clk;
end
```

```
`timescale 1ns/1ns
reg clk
parameter CYCLE=15;
always
begin
 #(CYCLE/2.0); clk = 1'b0;
 #(CYCLE/2.0); clk = 1'b1;
end
```

```
`timescale 1ns/100ps
reg clk
parameter CYCLE=15;
always
begin
 #(CYCLE/2.0); clk = 1'b0;
 #(CYCLE/2.0); clk = 1'b1;
end
```

Input from text files

- Read a text file to a vector (*memory* initialization)

mem is an array of registers: **reg [15:0] mem[0:127]**
datafile "filename" is ASCII, with values formated in binary or hexadecimal

```
$readmemb( "filename", mem, start_index, end_index); // binary
```

```
//binary data file
@00 // at address 0x00 (zero, hexadecimal)
0000_0000_0000_0001 // data, char "_" is a separator
@10 // next data is stored at address 10 (hex)
0000000000001011
```

```
$readmemh( "filename", mem, start_index, end_index); // hex
```

Handling text files

- $fopen, $fclose, $fmonitor, $fwrite, $fscanf, $fdisplay, …

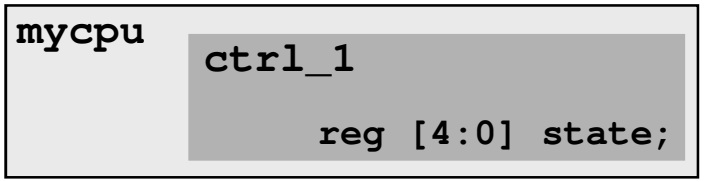
```
integer HI, HQ;  // File handlers

initial // open output files
begin
  HI = $fopen("dout_I.dat");
  HQ = $fopen("dout_Q.dat");
  // simulate ... close files
  $fclose(HI); $fclose(HQ);
end

initial
  $fmonitor(fh,...);  // $monitor to a text file

always @(posedge clock)
begin
  $fdisplay( HI, "%d", HI);  // appends newline
  $fwrite( HQ, "%d\n", HQ);  //
end
```

Hierarchical references

- Access to signal names in the design hierarchy

```
task dump_regfile; // dump CPU registers
integer i;
begin
  for(i=0;i<8;i=i+1)
    $display("R%d=%d (%h)", i, mycpu.regfile_1.regs[i],
                                mycpu.regfile_1.regs[i] )
end

initial
begin
  // apply reset and check initial state
  $display("Ctrl unit initial state: %b", mycpu.ctrl_1.state);
end
```

Signal **state** in instance **ctrl_1** instantiated in instance **mycpu**



Generation of random numbers

- System task generates an integer pseudo-random value

```
$random(seed); // initial seed
x = $random;   // returns a 32-bit random integer
x = $random & 32'h0000_000F; // integer between 0 and 15
```

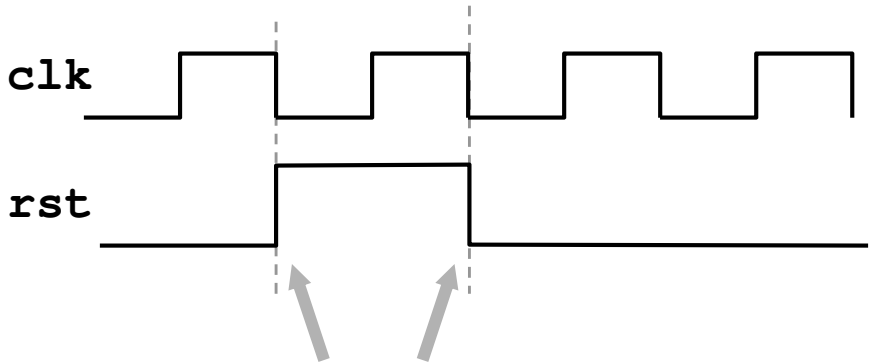
- Non-constant delays: **#(expression)**

```
integer x;
reg [15:0] y;
reg [7:0] cmd;
x = $random & 32'h0000_000F; y = $random / 32'h0fff_fffa;
cmd = t_gen_rnd_cmd($random);
for(i=0; i<MAXLOOP; i=i+1)
begin
  #(x) apply_initcommand;
  #(x*y/3) send_command( cmd, data);
end
```

Avoiding racing conditions

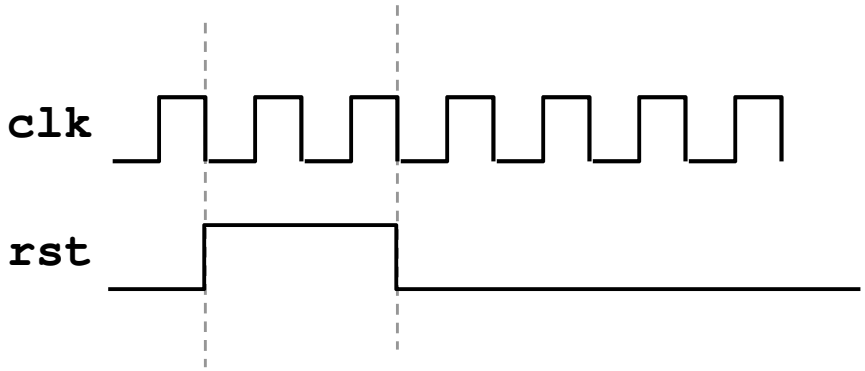
- For functional simulation logic models have zero-delay
  - Example: an asynchronous reset should not compete with clock

```
initial  // apply reset
begin
  rst = 1'b0; #200
  rst = 1'b1; #200
  rst = 1'b0;
end

always // generate clock
begin
 #50 clk = 1'b0;
 #50 clk = 1'b1;
end

always @(posedge clk or posedge rst)
…
```



Reset synchronization with clock

```
initial  // apply reset
begin    // for 2 clock cycles
  rst <= 1'b0;
  clk <= 1'b0;
  @(posedge clk); // wait rise of clk
  @(negedge clk); // sync to negedge
  rst <= 1'b1;
  repeat (2)
    @(negedge clk);
  rst <= 1'b0;
end

initial
  forever #(CLK_PERIOD/2) clk <= ~clk;
```



"Synchronous Resets? Asynchronous Resets? I am so confused!How will I ever know which to use?"
Clifford Cummings, Don Mills, Synospsys User Groups, 2002

SystemVerilog

- Extension of the Verilog language
  - Object oriented
    - Classes, methods, constructors,…
  - New datatypes
    - logic, bit, byte, int, shortint, longint,
  - Dynamic arrays, associative arrays, queues
    - Multi-dimensional, packed
  - Interfaces
    - Pack several interface signals
  - User defined data types (similar to C)
    - typedef, enum, struct, union
  - Assertions
    - Set conditions for automatic verification during simulation
  - UVM – Universal Verification Methodology

SystemVerilog

ALU instruction

```
typedef struct packed
{
 bit       typeinst;
 bit [3:0] opcode;
 bit [2:0] reg1;
 bit [2:0] reg2;
 bit [2:0] regdest;
} ty_alu_inst;
```

Jump instruction

```
typedef struct packed
{
 bit       typeinst;
 bit [3:0] opcode;
 bit [8:0] destaddr
} ty_jmp_inst;
```

```
typedef union packed
{
  ty_alu_inst inst1;
  ty_jmp_inst inst2;
} ty_cpu_inst;
```

```
ty_cpu_inst  oneinst;
  oneinst.inst1.reg1 = 3'b101;
```

SystemVerilog

- Assertions

```
//- ASSERTION #1
//- FIFO cannot have full and empty asserted at the same time

property p_not_full_and_empty; !(full && empty); endproperty

a_not_f_and_e: assert property (@(posedge clk) disable iff (!rst_n) p_not_full_and_empty);
```

- Coverage model

```
covergroup cg @ (posedge clk);
  coverpoint opcode {
    bins valid [] = { 5'h00,5'h02,5'h03,5'h05,5'h06,5'h0A};
    bins invalid = default; }

  coverpoint reg_a {
    bins low = {'h0};
    bins med = {['h1:'hFFE]};
    bins high = {'hFFF};   }
endgroup

cg cg_inst = new();
```