Assertions

Testing & Verification

Dept. of Computer Science & Engg, IIT Kharagpur

Pallab Dasgupta

Professor, Dept. of Computer Science & Engg.,
Professor-in-charge, AVLSI Design Lab,
Indian Institute of Technology Kharagpur

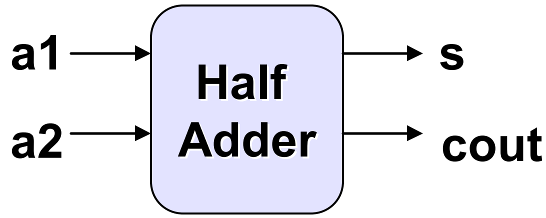
Agenda

- The Basic Temporal Operators
- Logics for Temporal Specification
- SystemVerilog Assertions
- Architectural Styles for Assertion IPs

Reference: *A Roadmap for Formal Property Verification,*
Pallab Dasgupta
Springer

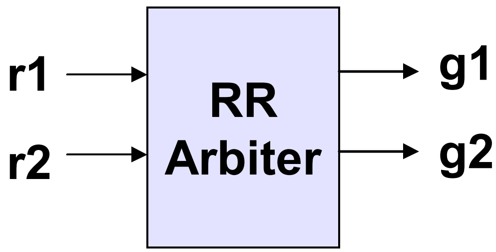
Why do we need "temporal" logic?

- **Propositional Logic – *Boolean formulas***



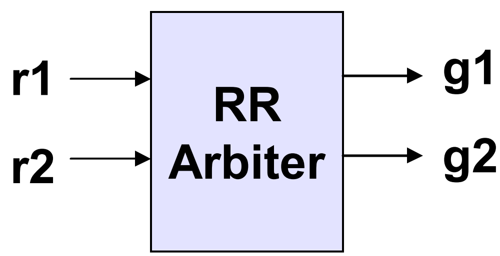
$cout \Leftrightarrow a1 \wedge a2$

$s \Leftrightarrow a1 \oplus a2$



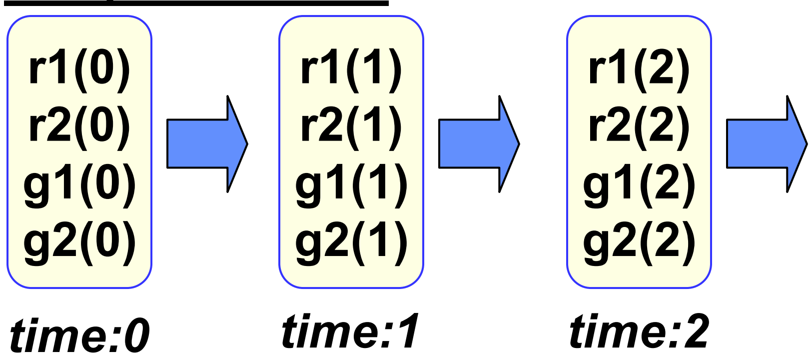
- **Temporal Logic**
  - Properties span across cycle boundaries
  - Consider a property of a two way round-robin arbiter
    - *If the request bit r1 is true in a cycle then the grant bit g1 has to be true within the next two cycles*

What does “temporal” mean?



If r1 is true in a cycle then g1 has to be true within the next two cycles

<u>Temporal worlds</u>



$\forall t\ [\ r1(t) \Rightarrow g1(t+1) \vee g1(t+2)\ ]$

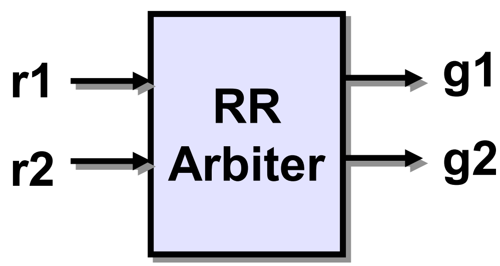
In *propositional* temporal logic, the time variable t is implicit.

- For example, we may write:

always r1 → (next g1) or (next next g1)

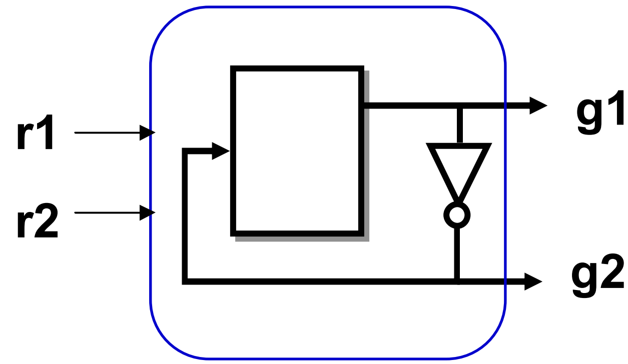
Implementations may not be logically equivalent

Specification:

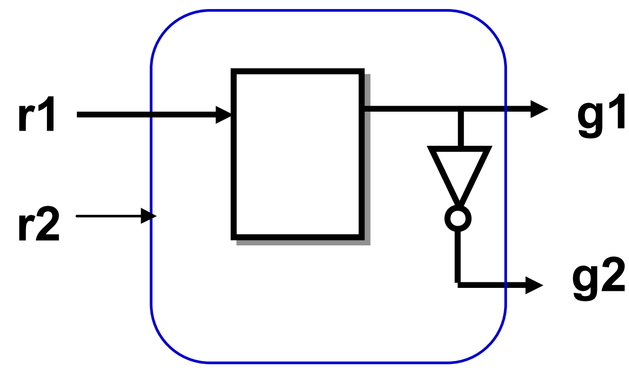


Design an arbiter with the following properties:

1. Whenever r_1 is raised, the arbiter must assert g_1 within the next two cycles
2. Whenever r_2 is raised, the arbiter must eventually assert g_2
3. The grant lines g_1 and g_2 are never asserted together



Implementation-1
(neither reads r_1 nor r_2 !!)

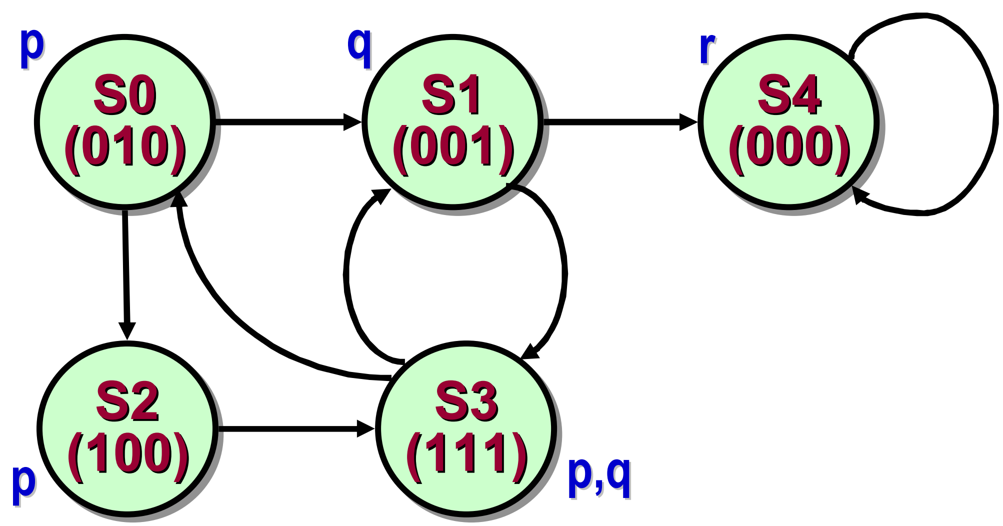


Implementation-2
(reads r_1 but not r_2 !!)

Kripke Structure

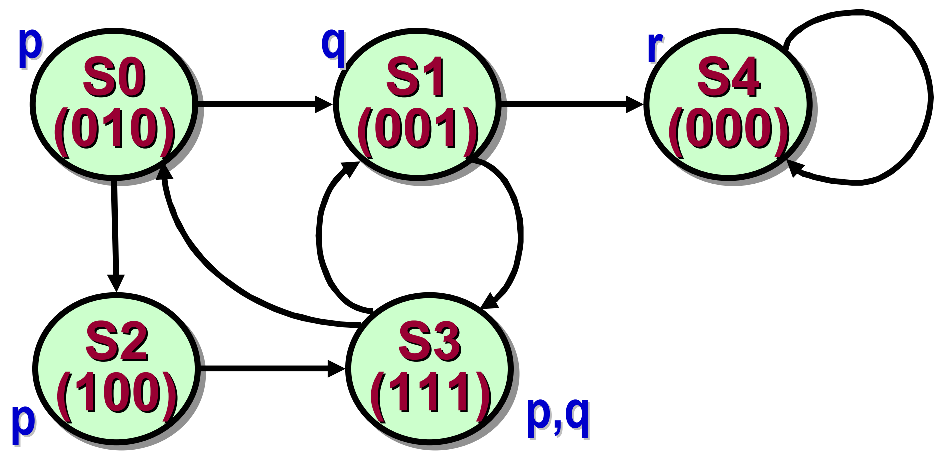
$K = (AP, S, S_0, T, L)$

- AP is a set of atomic propositions
- S is a set of states
- S_0 is a set of initial states
- $T \subseteq S \times S$, is a *total* transition relation
- $L: S \rightarrow 2^{AP}$ is a labeling function



Path

A path π = n0, n1, … in a Kripke structure, K = (AP, S, S_0, T, L), is a sequence of states such that $\forall k, (n_k, n_{k+1}) \in T$



Sample paths:

s0, s1, s4, s4, s4, …

s0, s2, s3, s0, s2, s3, …

s0, s2, s3, s1, s3, s0, …

π^k – suffix of n_k in π

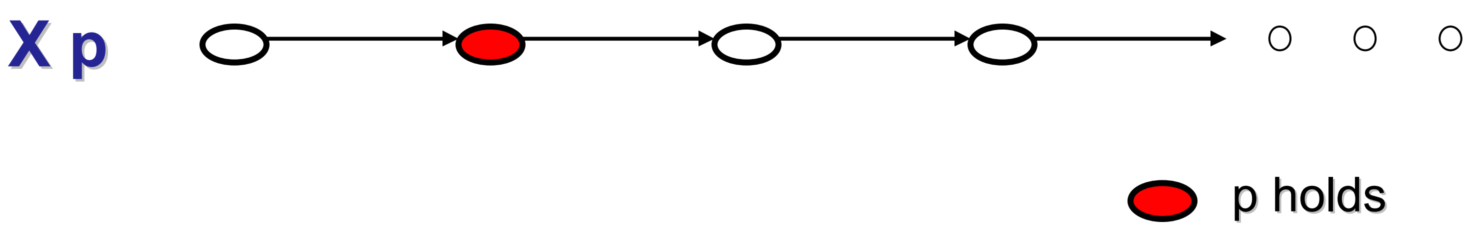
π = n0, n1, …, n_k, n_{k+1}, …

prefix of n_k in π

Temporal Operators

- Two fundamental path operators:
  - Next operator
    - Xp – *property p holds in the next state*
  - Until operator
    - p U q – *property p holds in all states up to the state where property q holds*
- Several derived (and commonly used operators)
  - Eventual operator
    - Fp – *property p holds eventually (at some future state)*
  - Always operator
    - Gp – *property p holds always (at all states)*
- All these operators are interpreted over paths of the underlying Kripke structure
- Temporal logics also support all the Boolean operators

The *Next* Operator

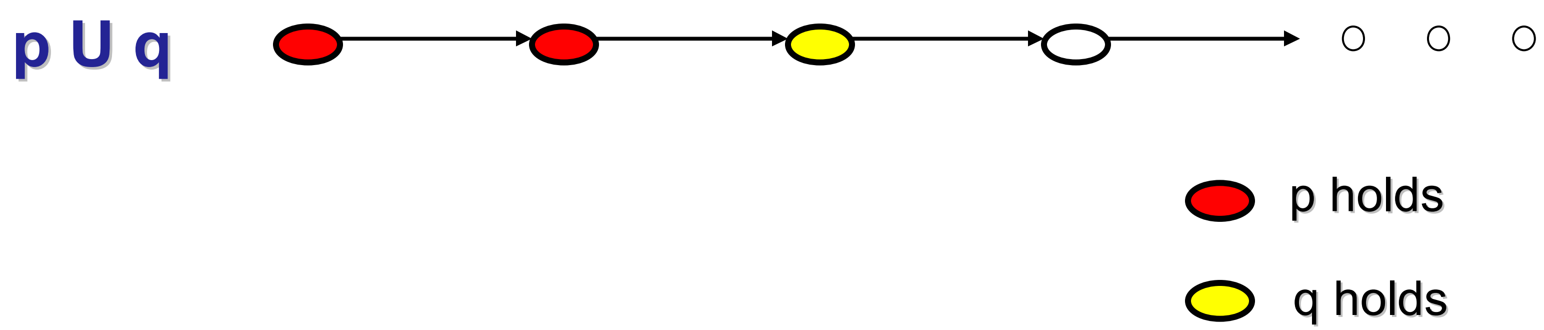


- p holds in the next state of the path

Formally:

$$\pi \models Xf \text{ iff } \pi^1 \models f$$

The *Until* Operator



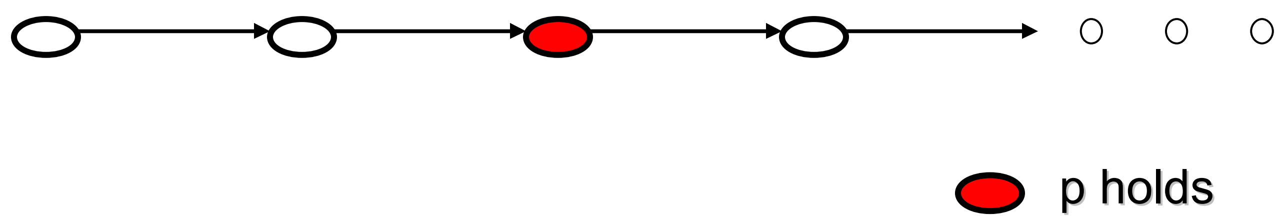
- **q holds eventually** and **p holds until q holds**

Formally:

$\pi \models f \ U \ g$ iff $\exists k$ such that $\pi^k \models g$ and $\forall j, 0 \leq j < k$ we have $\pi^j \models f$

The *eventual* Operator

F p



- p holds eventually (in future)

  alternatively

- ¬p does not hold always

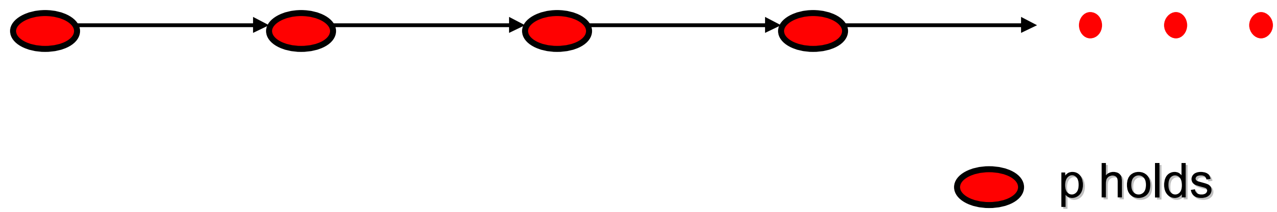
Formally:

$\pi \models Fg$ iff $\exists k$ such that $\pi^k \models g$

… which has the same meaning as true U g

The *always* Operator

G p



- p holds always (globally)

  alternatively

- ¬p does not hold eventually

Formally:

π |= Gf iff $\forall$j we have π^j |= f

… which has the same meaning as ¬ (F ¬f) or ¬ (true U ¬ f)

Duality between Always and Eventual Operators

eventually f

$= f \vee (\text{next } f) \vee (\text{next next } f) \vee (\text{next next next } f) \ldots$

$= \neg(\neg f \wedge (\text{next } \neg f) \wedge (\text{next next } \neg f) \wedge (\text{next next next } \neg f) \ldots)$

$= \neg(\text{ always } \neg f \text{ })$

… this is a variant of DeMorgan's Laws!!

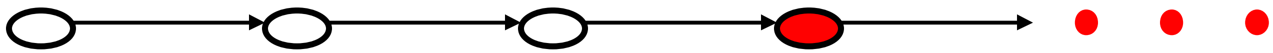
Thus:

$\neg Fp = G(\neg p)$

$\neg Gp = F(\neg p)$

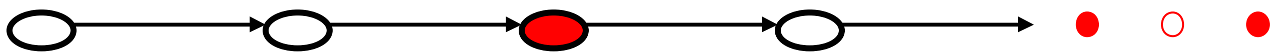
Nesting of Temporal Operators

F G p



Along the path there exists a state from which *p* will hold forever

G F p



Along the path for all states there will eventually be some state where *p* holds

alternatively

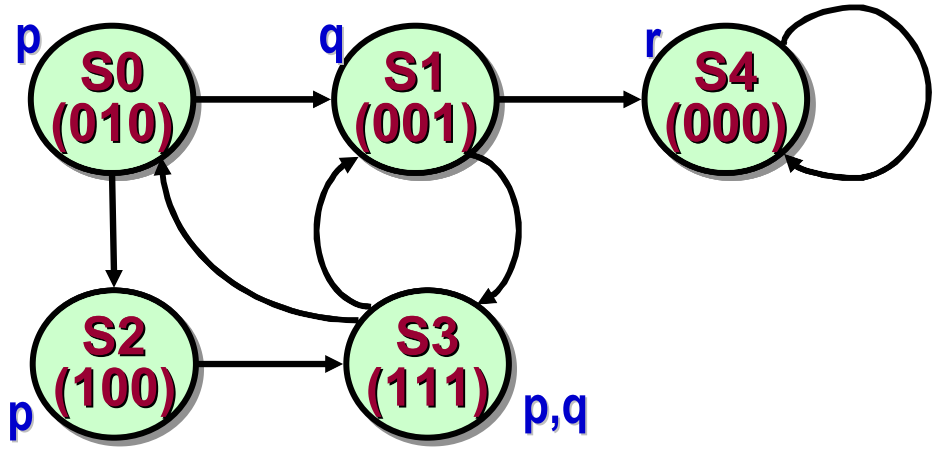
Along the path p will hold *infinitely often*

Linear Temporal Logic (LTL)

- **Syntax:**
  - Given a set, AP, of atomic propositions:
    - All Boolean formulas over AP are LTL properties, and
    - If *f* and *g* are LTL properties, then so are $\neg$f, Xf, and fUg

- **Semantics:**
  - A Kripke structure K models a LTL property g (denoted as K |= g) iff for every path π, which starts at some initial state of K, π |= g
  - This means that the property does not hold on K if there is any path in K which refutes the property

Examples



- The property pUq holds
- The property Fq holds
- The property GFq does not hold
  - Counterexample trace: s0, s1, s4, s4*
- The property pU(qUr) does not hold
  - Counterexample trace: s0, s2, s3, s0, (s2, s3, s0)*

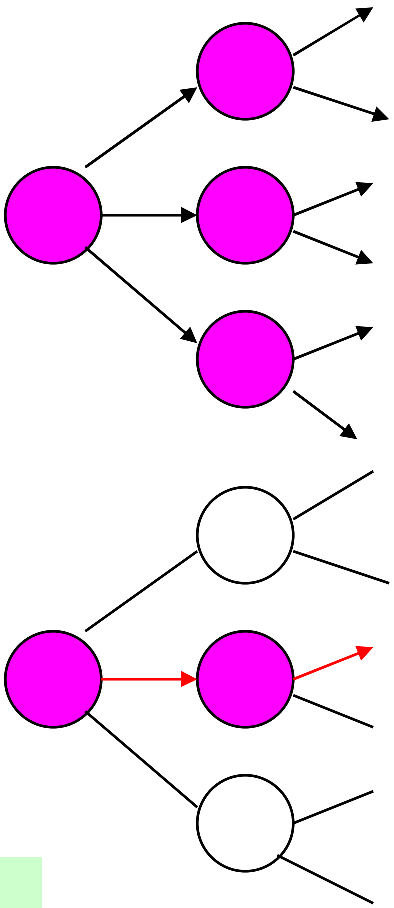
Path Quantifiers

- **A**

  " for all paths … "

- **E**

  " there exists a path … "

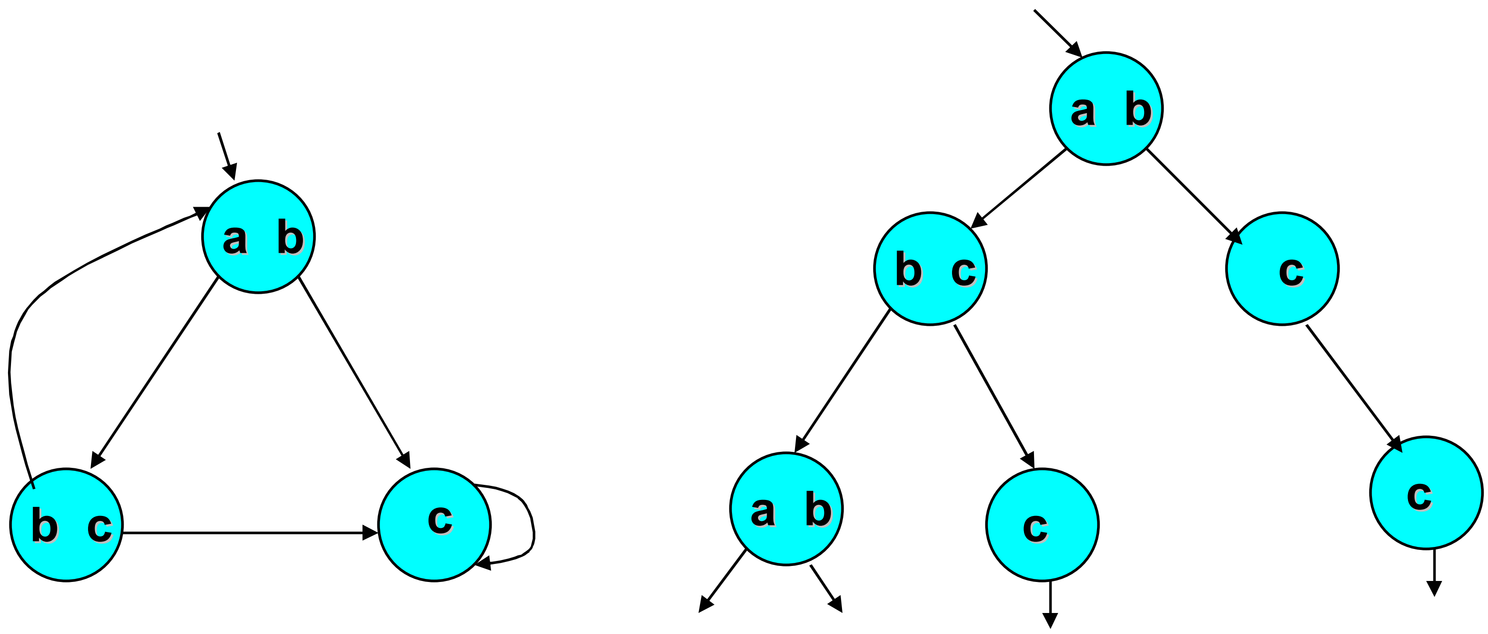


Used to specify that all of the paths or some of the paths starting at a particular state have some property

Branching Time Logic

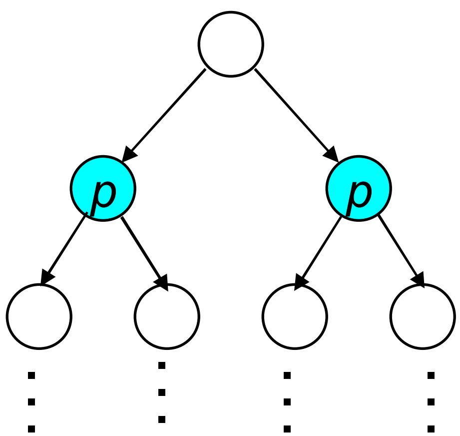
- **Branching time paradigm:**
  - Interpreted over computation trees, not linear traces

- **Computation tree:**



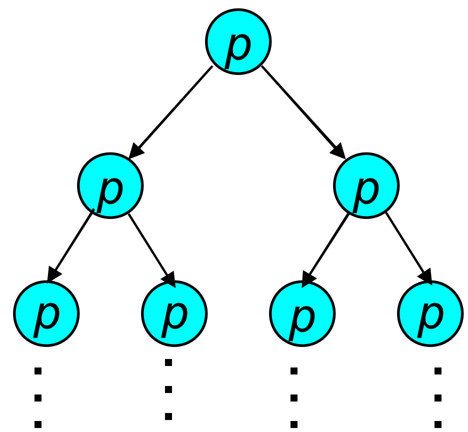
Universal Path Quatification

AX *p*



In all the next states *p* holds

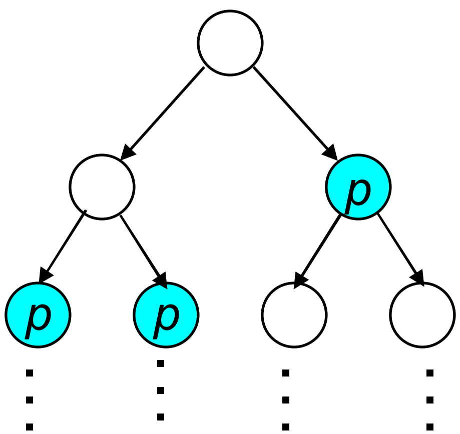
AG *p*



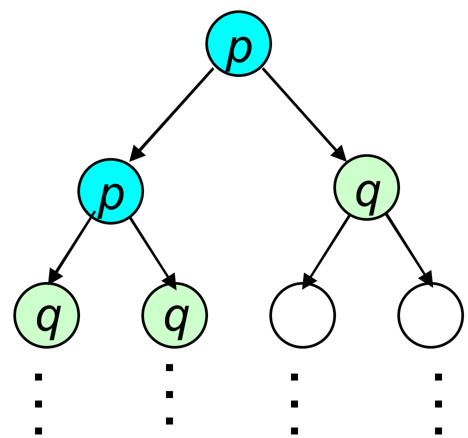
Along all the paths *p* holds forever

Universal Path Quantification

AF *p*



A(*p* U *q*)

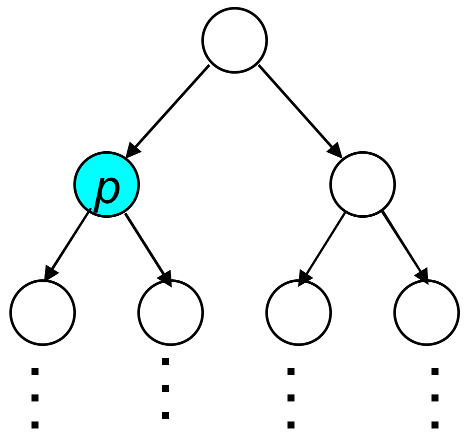


Along all the paths *p* holds eventually

Along all paths *p* holds until *q* holds

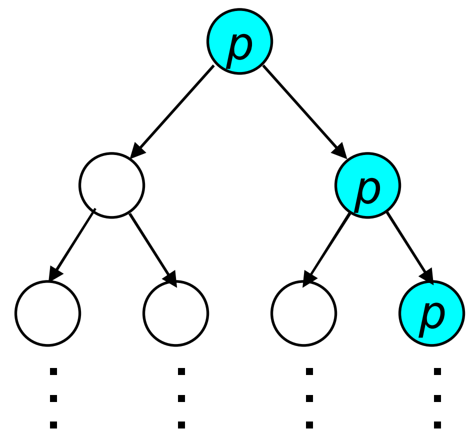
Existential Path Quantification

EX *p*



There exists a next state where *p* holds

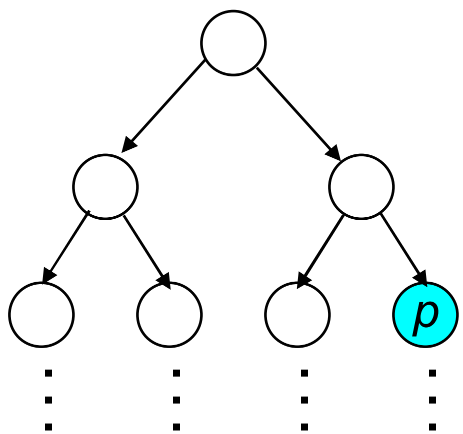
EG *p*



There exists a path along which *p* holds forever

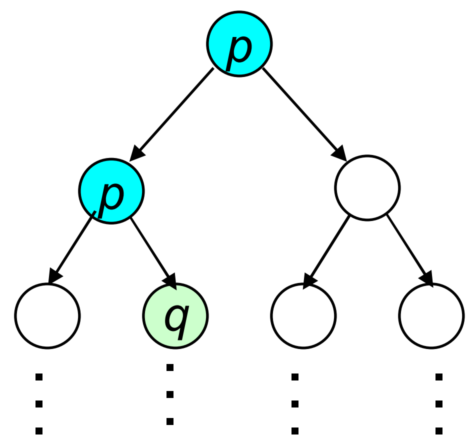
Existential Path Quantification

EF *p*



There exists a path along which *p* holds eventually

E(*p* U *q*)



There exists a path along which *p* holds until *q* holds

Computation Tree Logic (CTL)

- **Syntax:**
  - Given a set, AP, of atomic propositions:
    - All Boolean formulas over AP are CTL properties, and
    - If *f* and *g* are LTL properties, then so are ¬f, AXf, EXf, A[fUg] and E[fUg]
  - We also have derived properties like EFg, AFg, EGf, and AGf

- **Semantics:**
  - The property Af is true at a state s of the Kripke structure, iff the path property f holds on all paths starting at s
  - The property Ef is true at a state s of the Kripke structure, iff the path property f holds on some path starting at s

Nested Properties in CTL

- **AX AG p**

  “ from all the next states p holds forever along all paths ”

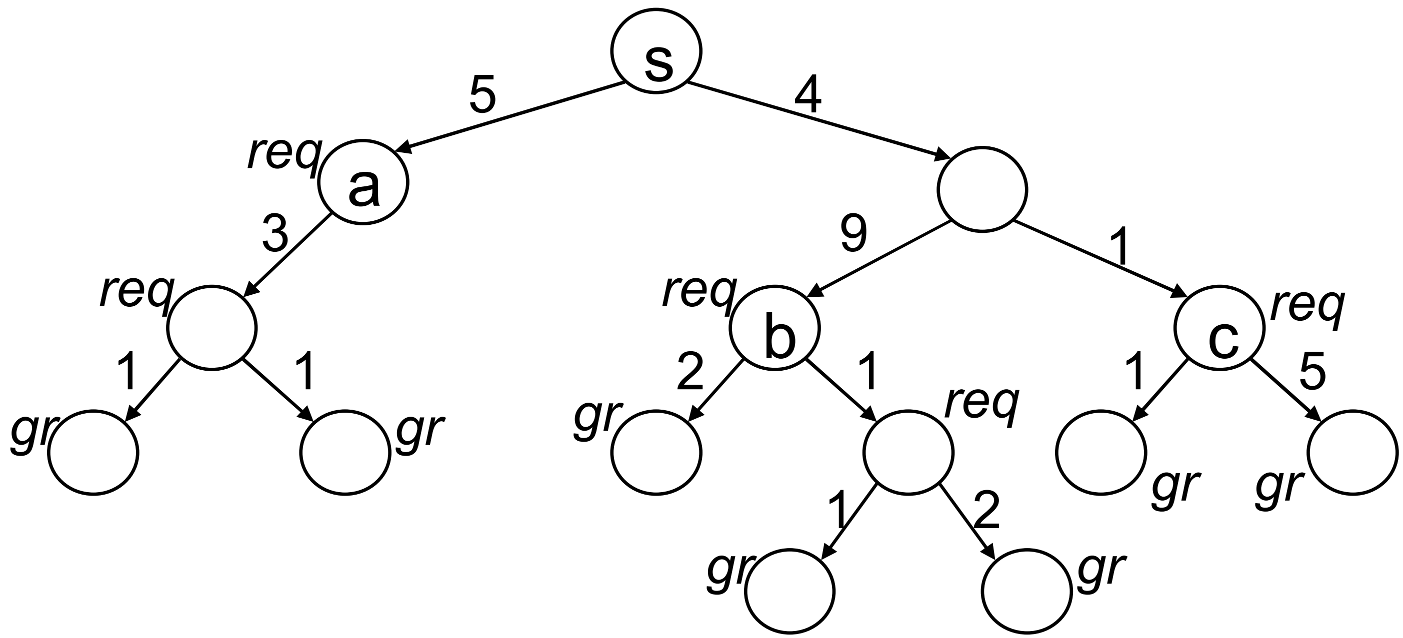
- **EX EF q**

  “ there exists a next state from which there exists a path to a state where q holds ”

- **AG EF r**

  “ from any state there exists a path to a state where r holds ”

Example: *Analyzing Request and Grants*



From s the system always makes a request in future: *AFreq*

All requests are eventually granted: *AG(req → AFgr)*

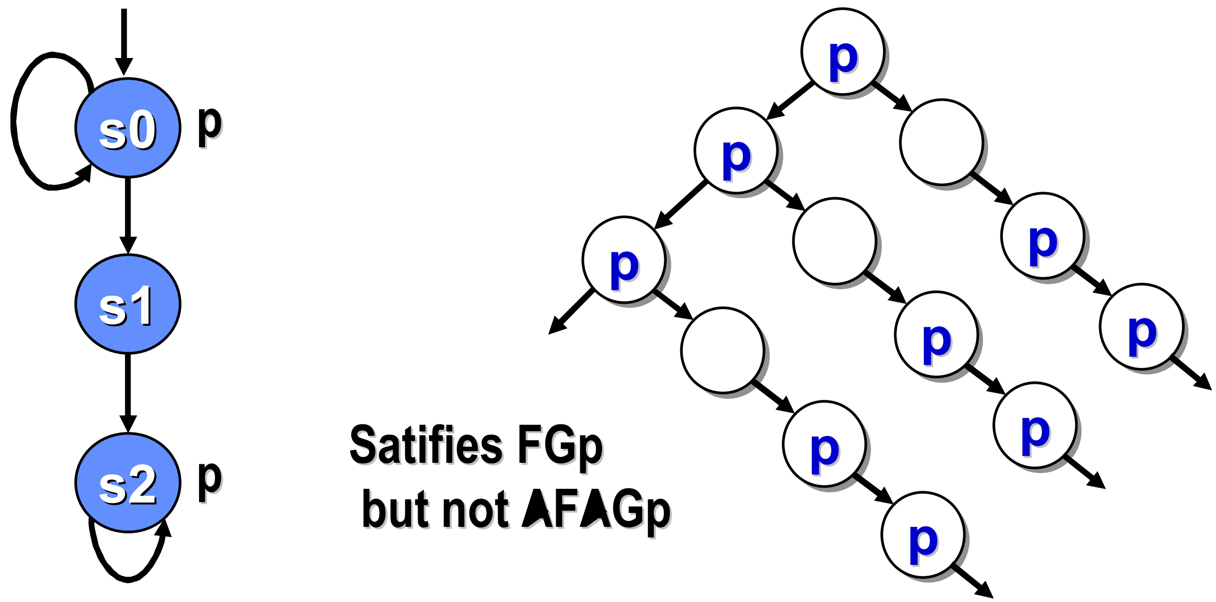
Sometimes requests are immediately granted: *EF(req → EXgr)*

Requests are not always immediately granted: *¬ AG(req → AXgr)*

Requests are held till grant is received: *AG(req → AF(req U gr))*

LTL versus CTL

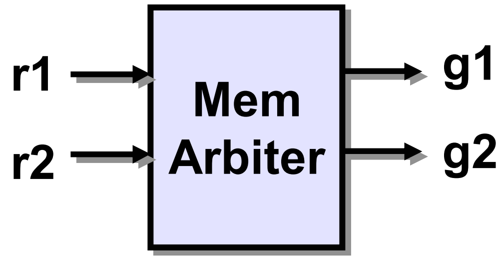
- CTL has more operators than LTL – *which allows us to specify branching time properties (not supported in LTL).*
- Can all LTL properties be expressed in CTL?
  - No.
  - For example, FGp cannot be expressed in CTL
  - Note that FGp is not equivalent to AFAGp



Satisfies FGp
but not AFAGp

Simple Case Study: *A Memory Arbiter*

mem-arbiter(input r1, r2, clk, output g1, g2)



Properties:

1. Request line r1 has higher priority than request line r2. Whenever r1 goes high, the grant line g1 must be asserted for the next two cycles

   $$G[\, r1 \Rightarrow Xg1 \wedge XXg1 \,]$$

2. When none of the request lines are high, the arbiter parks the grant on g2 in the next cycle

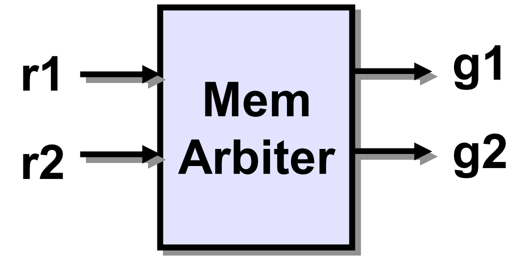
   $$G[\, \neg r1 \wedge \neg r2 \Rightarrow Xg2 \,]$$

3. The grant lines g1 and g2 are mutually exclusive

   $$G[\, \neg g1 \vee \neg g2 \,]$$

Memory Arbiter: Is the Specification Correct?

mem-arbiter(input r1, r2, clk, output g1, g2)

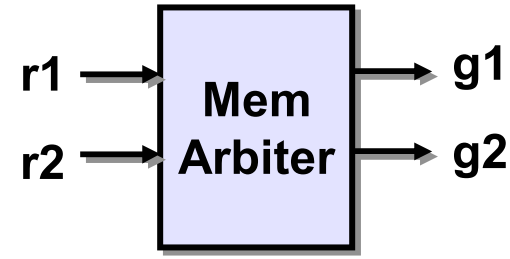


1. $G[\, r1 \Rightarrow Xg1 \wedge XXg1 \,]$
2. $G[\, \neg r1 \wedge \neg r2 \Rightarrow Xg2 \,]$
3. $G[\, \neg g1 \vee \neg g2 \,]$

- Consider the case when r1 is high at time t and low at time t+1, and r2 is low at both time steps.
  - The first property forces g1 to be high at time t+2
  - The second property forces g2 to be high at time t+2
  - The third property says g1 and g2 cannot be high together
  - We have a conflict !!
  - Lets go back to the specification

Memory Arbiter: Revised Specs

mem-arbiter(input r1, r2, clk, output g1, g2)



Properties:

1. Request line r1 has higher priority than request line r2. Whenever r1 goes high, the grant line g1 must be asserted for the next two cycles

   $G[\, r1 \Rightarrow Xg1 \wedge XXg1 \,]$

2. When none of the request lines are high, the arbiter parks the grant on g2 in the next cycle

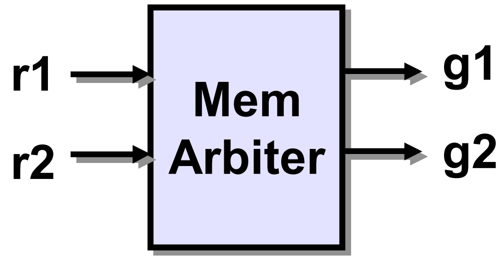
   ~~$G[\, \neg r1 \wedge \neg r2 \Rightarrow Xg2 \,]$~~ revised to $G[\, \neg g1 \Rightarrow g2 \,]$

3. The grant lines g1 and g2 are mutually exclusive

   $G[\, \neg g1 \vee \neg g2 \,]$

Memory Arbiter: Is the Specification Complete?

mem-arbiter(input r1, r2, clk, output g1, g2)

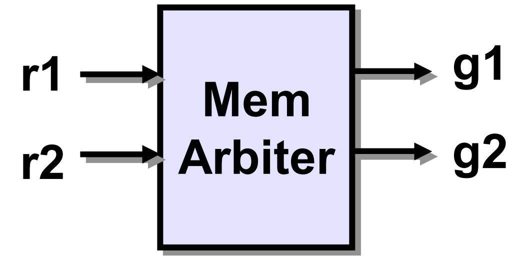


1. $G[\ r1 \Rightarrow Xg1 \wedge XXg1\ ]$
2. $G[\ \neg g1 \Rightarrow g2\ ]$
3. $G[\ \neg g1 \vee \neg g2\ ]$

- **Observation:** *We can satisfy the specification by designing an arbiter which always asserts g1 and never asserts g2!!*
  - We need to add either of the following types of properties:
    - Ones which specify when g2 should be high, or
    - Ones which specify when g1 should be low
  - Lets go back to the specification

Memory Arbiter: Revised Specs

mem-arbiter(input r1, r2, clk, output g1, g2)



Properties:

1. Request line r1 has higher priority than request line r2. Whenever r1 goes high, the grant line g1 must be asserted for the next two cycles

   $G[\ r1 \Rightarrow Xg1 \wedge XXg1\ ]$

2. When none of the request lines are high, the arbiter parks the grant on g2 in the next cycle

   $G[\ \neg g1 \Rightarrow g2\ ]$

3. When r1 is low for consecutive cycles, then g1 should be low in the next cycle

   $G[\ \neg r1 \wedge X\neg r1 \Rightarrow XX\neg g1\ ]$

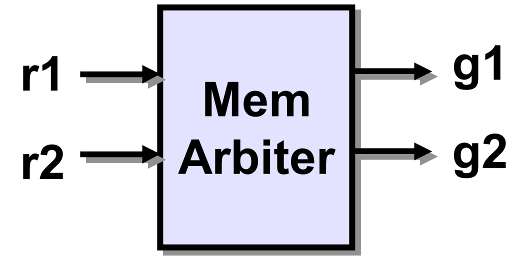
   New!!

4. The grant lines g1 and g2 are mutually exclusive

   $G[\ \neg g1 \vee \neg g2\ ]$

Memory Arbiter: Is the Specs Complete Now?

mem-arbiter(input r1, r2, clk, output g1, g2)



1. $G[\ r1 \Rightarrow Xg1 \wedge XXg1\ ]$
2. $G[\ \neg g1 \Rightarrow g2\ ]$
3. $G[\ \neg r1 \wedge X\neg r1 \Rightarrow XX \neg g1\ ]$
4. $G[\ \neg g1 \vee \neg g2\ ]$

- **Observation:** *We cannot satisfy the specs without reading the value of r1, but we can satisfy the specs without reading r2!!*
  - Consider the following implementation strategy:
    - Assert g1 for two cycles whenever we get r1
    - Assert g2 otherwise
  - Lets us live with this specification

Real-Time Properties

- **Real-time systems**
  - Predictable response times are essential for correctness
  - Example: controllers for aircraft, industrial machinery, robots, etc

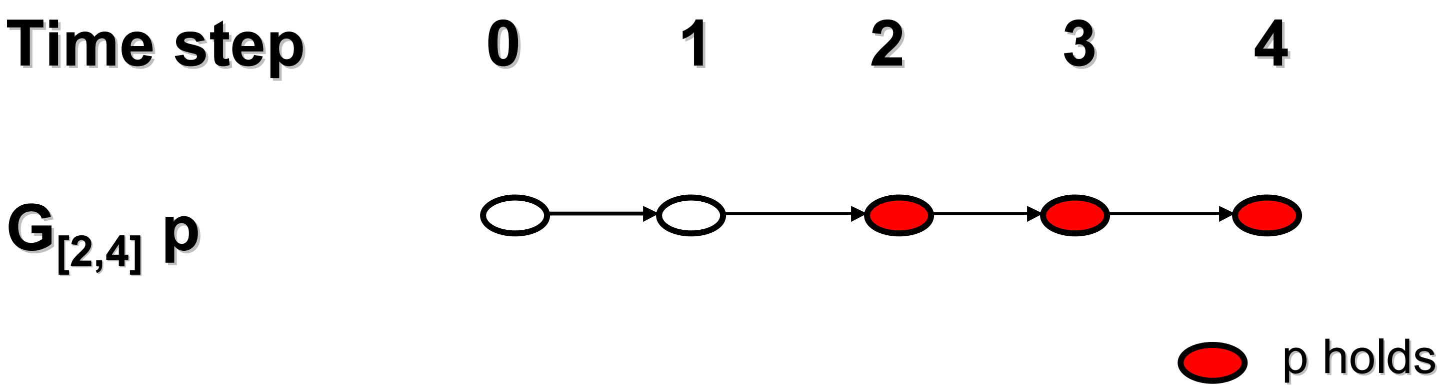
- **It is difficult to express complex timing properties**
  - Simple: “event p will happen in the future”
    - Fp
  - Complex: “event p will happen within at most n time units”
    - $p \vee X\,p \vee XX\,p \vee \ldots \vee [XX \ldots (n \text{ times})]\,p$

Bounded Temporal Operators

- Specify real-time constraints
  - over bounded traces

- Various bounded temporal operators
  - $G_{[m, n]}\ p$ — p always holds between the m^{th} and n^{th} time step
  - $F_{[m, n]}\ p$ — p eventually holds between m^{th} and n^{th} time step
  - $X_m\ p$ — p holds at the m^{th} time step
  - $p\ U_{[m,n]}\ q$ — q eventually holds between m^{th} and n^{th} time step and p holds until that point of time

Examples

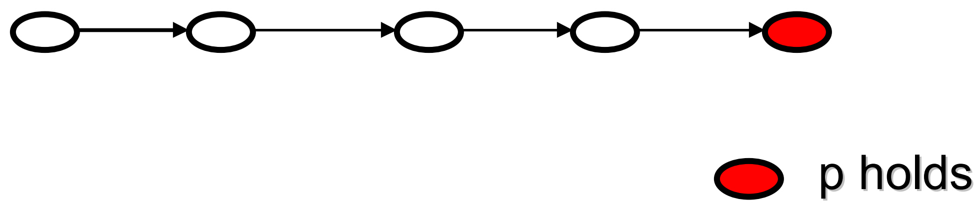


p holds always between 2nd and 4th time step

Examples

Time step 0 1 2 3 4

$F_{[2,4]}\ p$

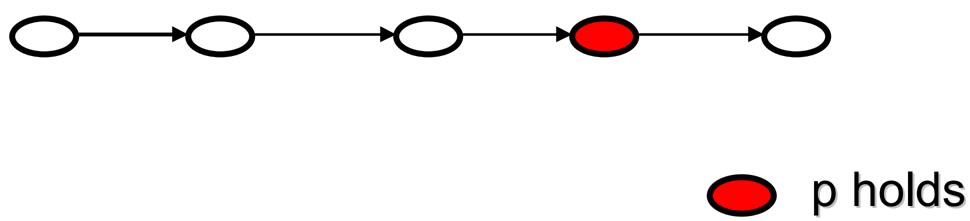


p holds eventually between 2nd and 4th time step

Examples

Time step **0** **1** **2** **3** **4**

X_3 p

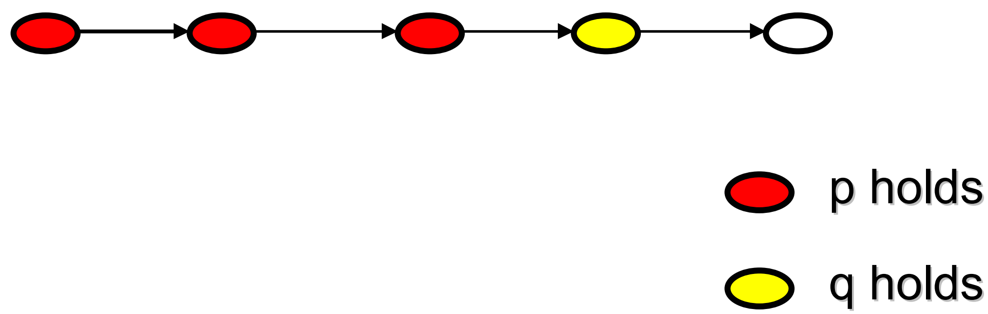


p holds in the 3rd time step

Examples

Time step 0 1 2 3 4

p $U_{[2,4]}$ q



q holds eventually between 2nd and 4th time step and p holds until q holds

Timing Properties

- Whenever a hpreq is recorded, the hpgrant should take place within 4 units of time.

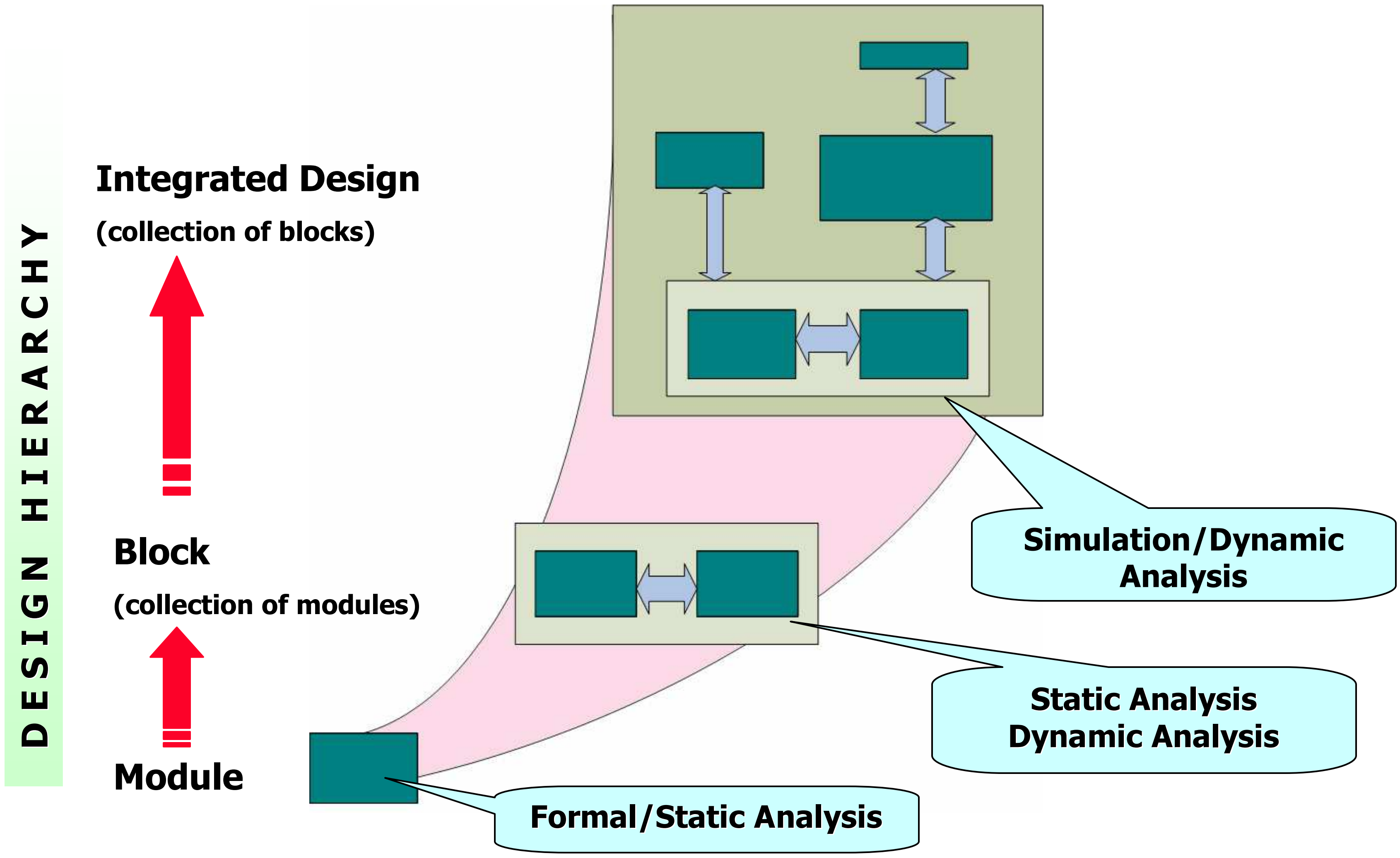
  $AG(posedge(hpreq) \rightarrow AF_{[0,4]}\ posedge(hpgrant)))$

- The arbiter will provide exactly 64 units of time to high-priority users in each grant.

  $AG(posedge(hpusing) \rightarrow$

  $A(\neg negedge(hpusing)\ U_{[64,64]}\ negedge(hpusing)))$

Assertion Based Verfication (ABV) Methodology



Assertions: *Industry Standards*

- **Predecessors**
  - Sugar from IBM Haifa
  - Forspec from Intel
  - Open Vera Assertions (OVA) from Synopsys
- **Three main standards today**
  - Property Specification Language (PSL)
    - Supports both branching time and linear time properties
  - SystemVerilog Assertions (SVA)
    - An integral part of SystemVerilog
  - Open Verification Library (OVL)
    - A collection of simple monitor libraries that can be stitched together to monitor more complex behaviors
  - Developed by Accellera. PSL has become IEEE 1850 PSL and SVA is a part of IEEE 1800 SystemVerilog

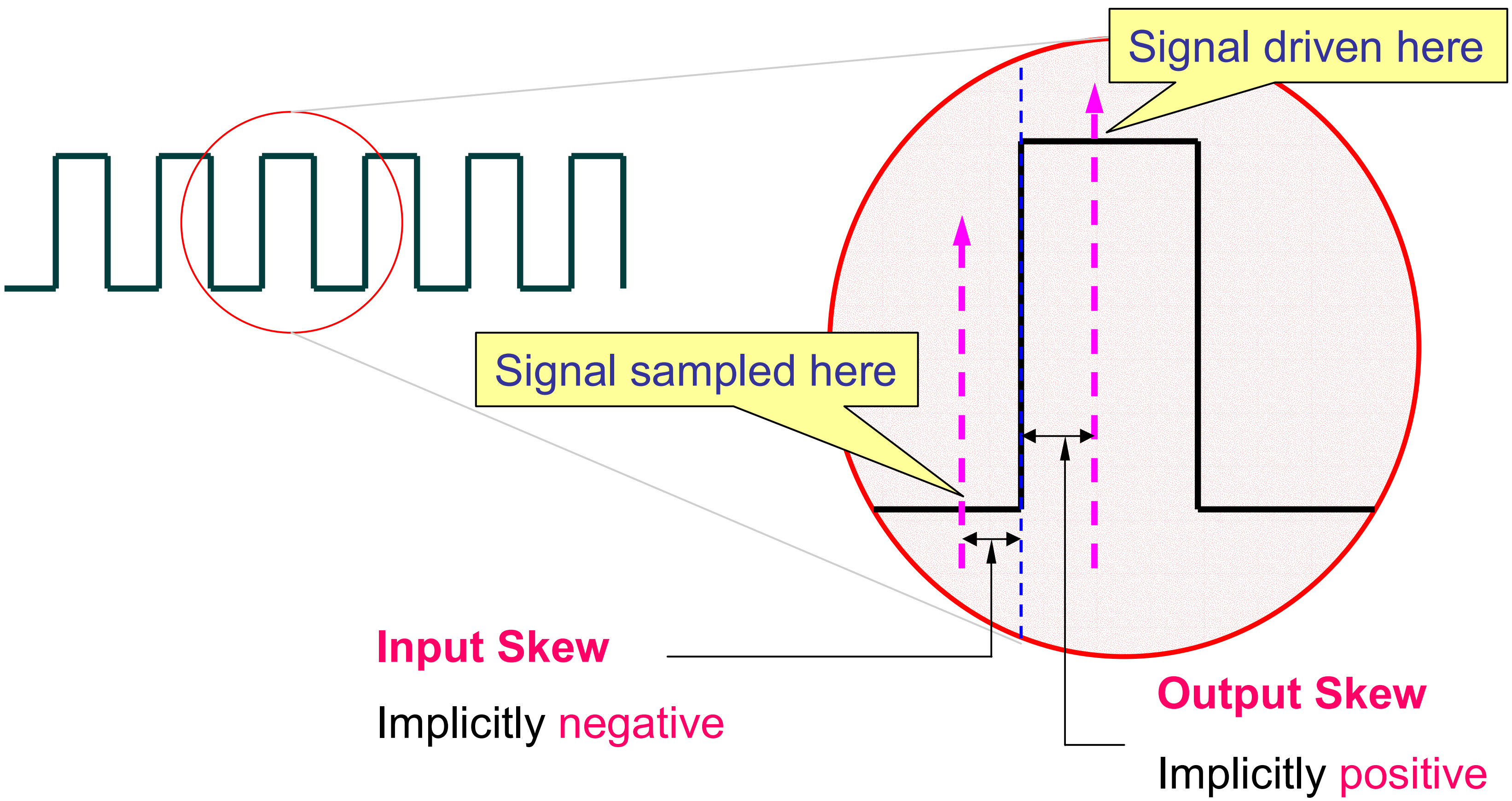
SystemVerilog Scheduling Semantics

1. Preponed
2. Pre-active
3. Active
4. Inactive
5. Pre-NBA
6. NBA
7. Post-NBA
8. Observed
9. Post-observed
10. Reactive
11. Postponed

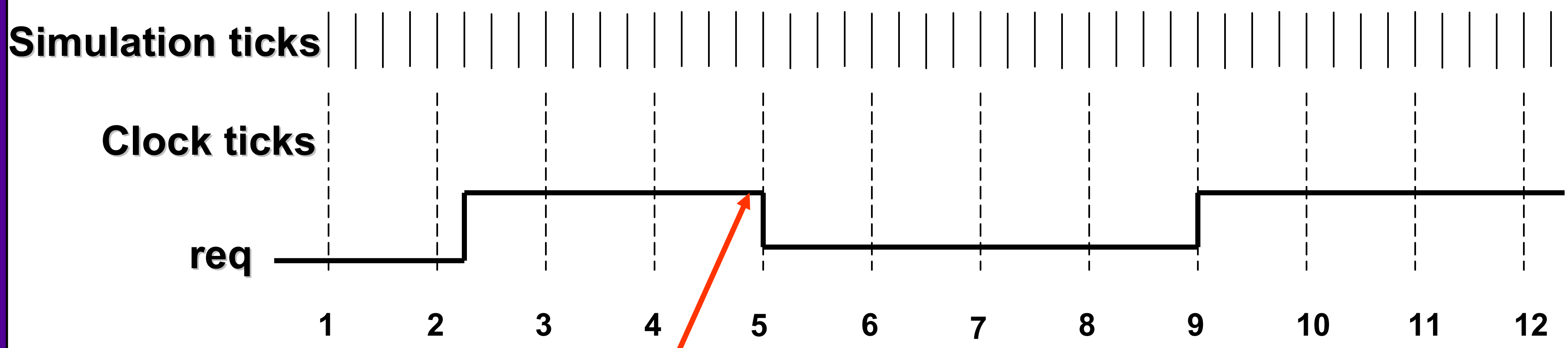
SystemVerilog Scheduling Semantics

- **Preponed**
  - It allows for user code to access data at the current time slot before any net or variable has changed state
- **Observed**
  - Evaluates property expressions if they are triggered
- **Reactive**
  - Evaluates pass/fail code of the properties

Signal Sampling



Example



1. Value of req at clock tick 5 is 1 not 0
2. Value of req at clock tick 9 is 0 not 1

SVA: *A Quick Overview*

- **The Memory Arbiter Example:**

  mem-arbiter(input r1, r2, clk, output g1, g2)

Properties:

P1: $G[\ r1 \Rightarrow Xg1 \wedge XXg1\ ]$

P2: $G[\ \neg g1 \Rightarrow g2\ ]$

P3: $G[\ \neg r1 \wedge X\neg r1 \Rightarrow XX \neg\ g1\ ]$

P4: $G[\ \neg g1 \vee \neg g2\ ]$

- **We will first code these properties in SVA.**
- **We will then see how to bind these properties with the interface of the DUT**

SVA: *A Quick Overview*

```
property P1;
  @( posedge clk )
  r1 |-> ##1 g1 ##1 g1;
endproperty
```

```
property P2;
  @( posedge clk )
  !g1 |-> g2;
endproperty
```

LTL Properties:

P1: $G[\, r1 \Rightarrow Xg1 \wedge XXg1 \,]$

P2: $G[\, \neg g1 \Rightarrow g2 \,]$

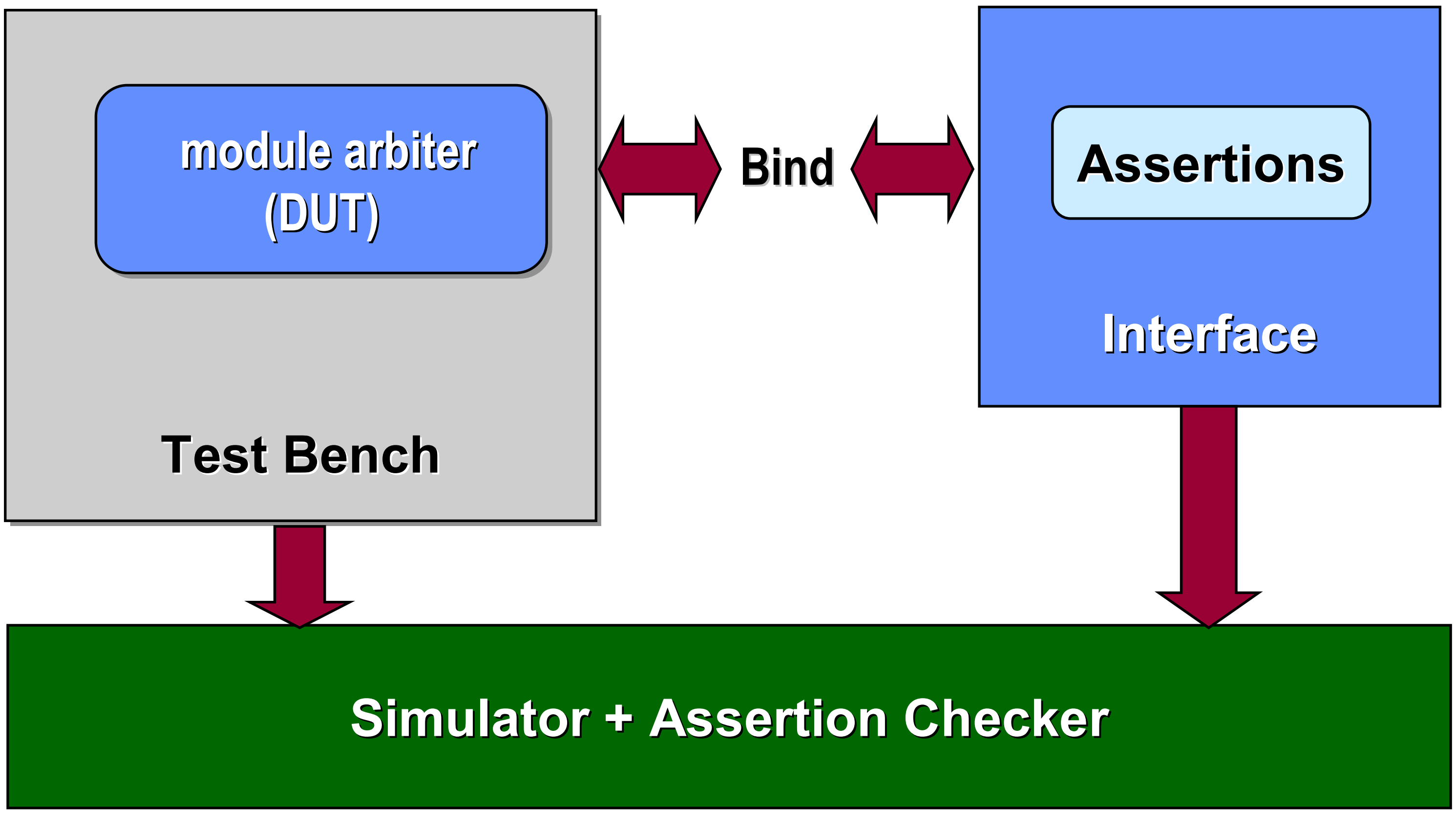
P3: $G[\, \neg r1 \wedge X\neg r1 \Rightarrow XX \neg g1 \,]$

P4: $G[\, \neg g1 \vee \neg g2 \,]$

```
property P3;
  @( posedge clk )
  !r1 ##1 !r1 |-> ##1 !g1;
endproperty
```

```
property P4;
  @( posedge clk )
  !g1 || !g2;
endproperty
```

Interfaces and Binding



Interface: *Memory Arbiter*

```
interface ArbChecker( input g1, g2, r1, r2, clk ) ;
property P1;
    @(posedge clk) r1 |→ ##1 g1 ##1 g1;
endproperty
property P2;
    @(posedge clk) !g1 |→ g2;
endproperty
----
----
GrantWhenRequest:
    assert property(P1)
    else $display("Property P1 has failed");
OneGrantHigh:
    assert property(P2)
    else $display("Property P2 has failed");
----
endinterface
```

Test Bench: *Memory Arbiter*

```
module Top;
wire r1, r2, g1, g2;
reg clk;

arbiter A(r1, r2, g1, g2, clk);  // Instantiation of the module arbiter

initial begin
    clk = 1;
    forever begin
            #1 clk = ~clk;
    end
end

// Rest of the test bench code …
----
----
endmodule
```

Binding

- We need to *bind* the interface, ArbChecker, with the test bench
  - This can be done using the following statement:

    bind Top ArbChecker ArbC(g1, g2, r1, r2, clk)

SVA: Sequence Expressions

- Sequence expressions are the basic building blocks of SVA
- Examples:

  ##0 r1 *// r1 is true in this cycle*

  ##1 r1 *// r1 is true in the next cycle*

  ##5 r1 *// r1 is true exactly after 5 cycles*

  ##[5:9] r1 *// r1 is true sometime between the 5th and 9th cycle*

- Comparison with Timed LTL
  - ##1 r1 is the same as Xr1
  - ##5 r1 is the same as $F_{[5,5]}$ r1
  - ##[5:9] r1 is the same as $F_{[5,9]}$ r1
- What is the meaning of the following sequence expression?

  a ##[1:5] (b||c) ##3 d

SVA: Sequence Expressions

- Sequence expressions can be given a name
- For example, we may rewrite **a ##[1:5] (b||c) ##3 d** as:

```
sequence s1;
  (b||c) ##3 d;
endsequence

sequence s2;
  a ##[1:5] s1;
endsequence
```

Note the use of s1 here

Sequence Operations: *Repetition*

- **Consecutive Repetition**
  - p[*5] matches when 5 consecutive states satisfy p
  - p[*3:5] ##1 q k (3≤k≤5) consecutive matches followed by q
  - p[*3:$] ##1 q At least 3 consecutive matches followed by q

  - *The request r must remain high until the grant g is asserted*

    r |→ r[*1:$] ##1 g

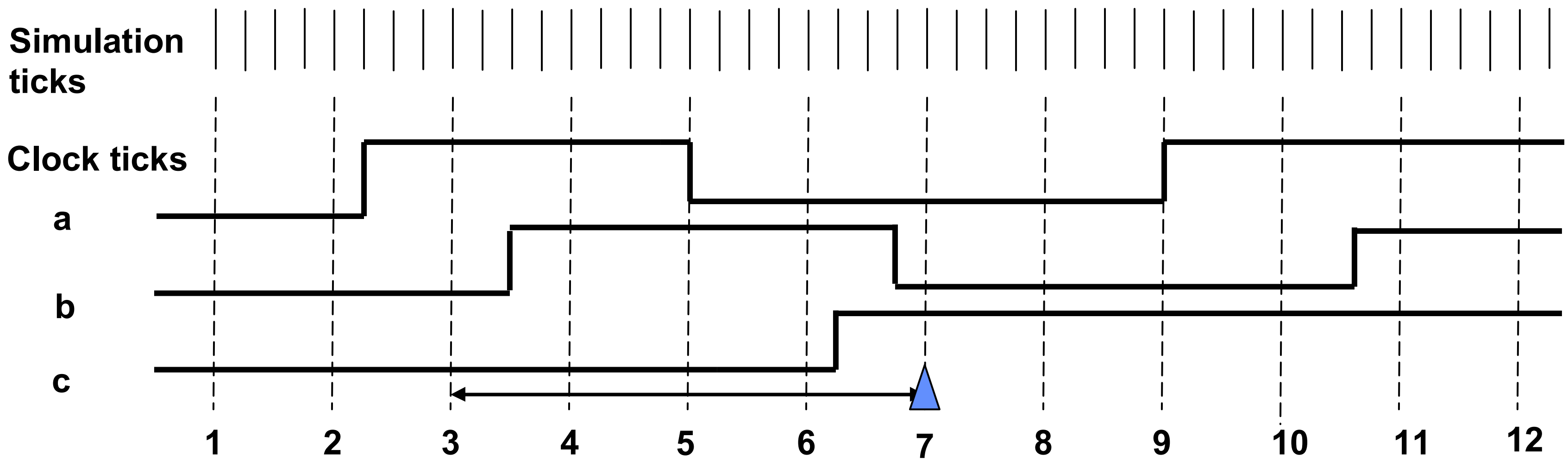
  - *The LTL property, p U q, is equivalent to:*

    p[*0:$] ##1 q

    Note the 0 here

Consecutive Repetitions (contd..)

```
sequence s1;
    @(posedge clk) a ##1 b [*3] ##1 c;
endsequence
```



Sequence Operations: *Repetition*

- **Goto Repetition**
  - p[*→5] ##1 q — the match of q at some time t is preceded by 5 matches (not necessarily consecutive) of p, including one at time t – 1.

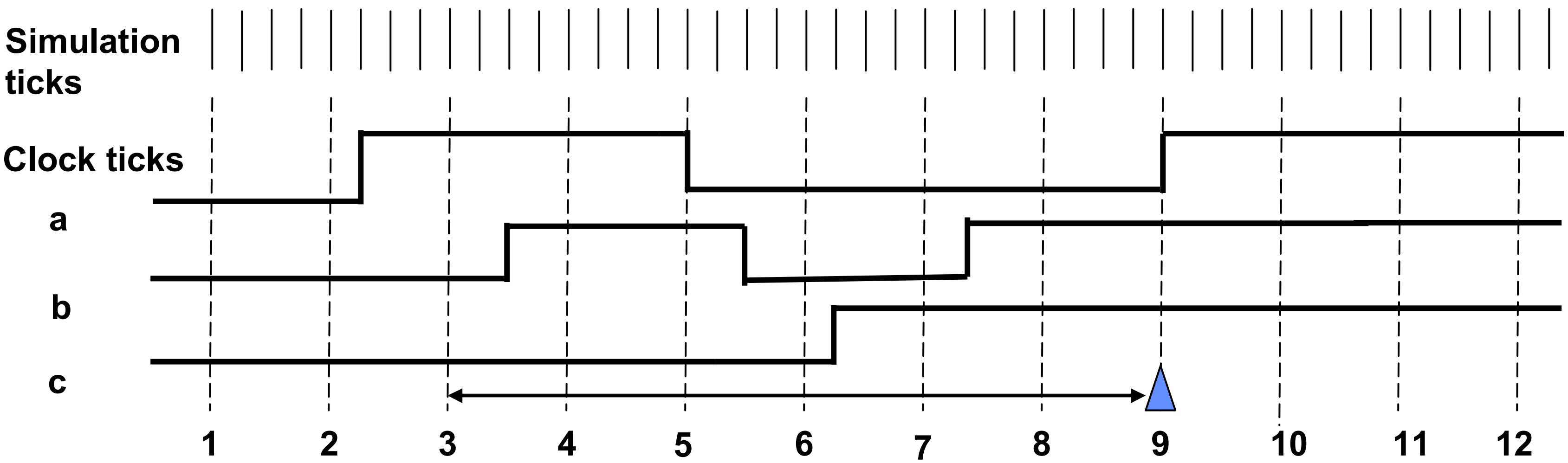
  - *The transfer must be aborted if the transfer is "split" more than once*

    split[*→2] ##1 abort

  - p[*→3:5] ##1 q — the match of q at some time t is preceded by 3 to 5 matches (not necessarily consecutive) of p, including one at time t – 1.

Goto Repetitions

```
sequence s1;
        @(posedge clk) a ##1 b [*->3] ##1 c;
endsequence
```

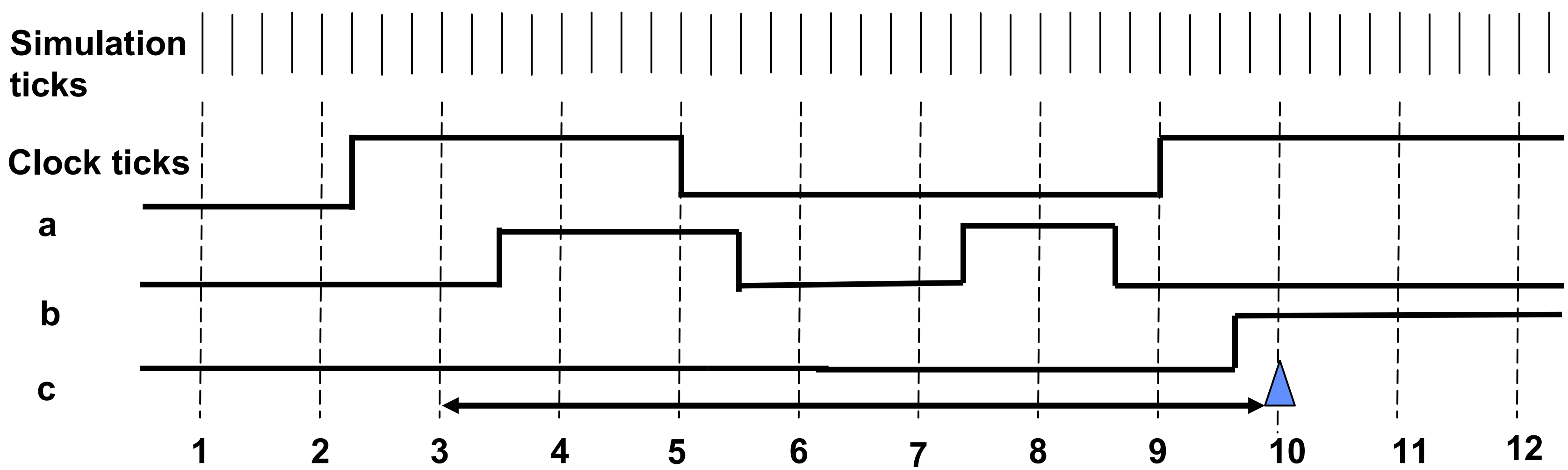


Sequence Operations: *Repetition*

- **Non-consecutive Repetition**
  - **split[*=2] ##1 abort**
    - ***The transfer is aborted if it is split more than once, but it is not necessary that the abort takes place immediately after the second split.***
  - **p[*=3:5] ##1 q** **matches at time t, if q matches at time t and p matches 3 to 5 times before time t.**

Non- consecutive Repetitions

```
sequence s1:
    @(posedge clk) a ##1 b [*=3] ##1 c;
endsequence
```

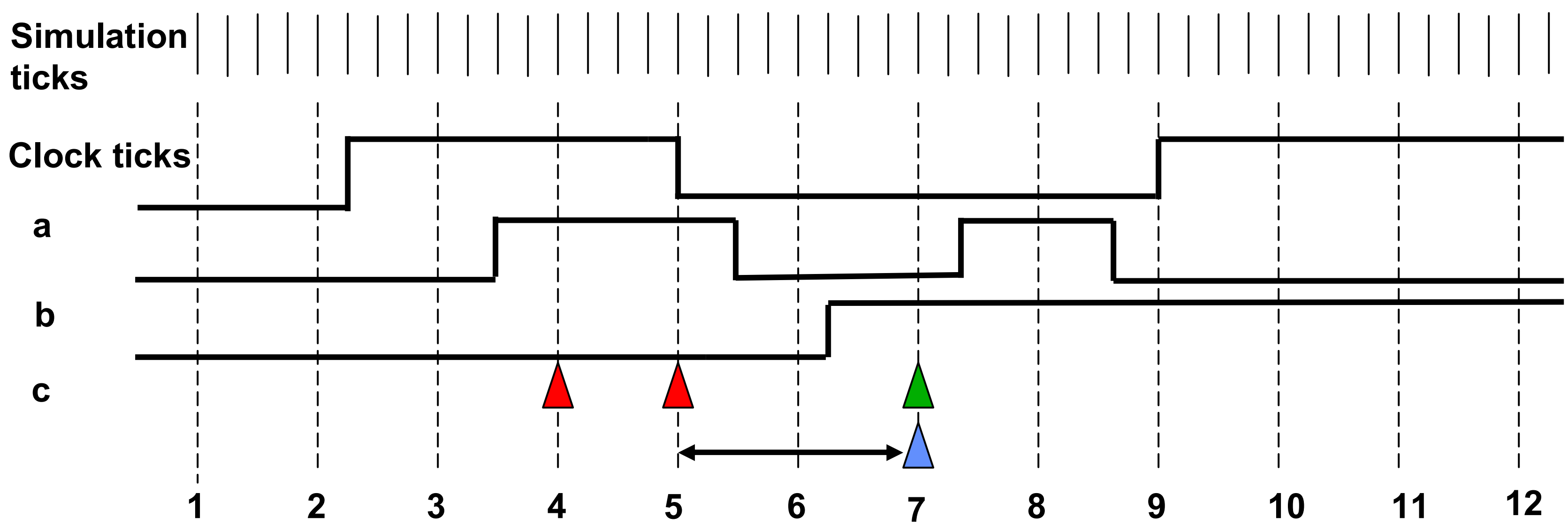


AND - operation

- The binary operator "and" is used when both the operand expressions are expected to succeed
- End time of the operands can be different

Example:

(a ##1 b) and (a ##1 b ##2 c)



Intersection

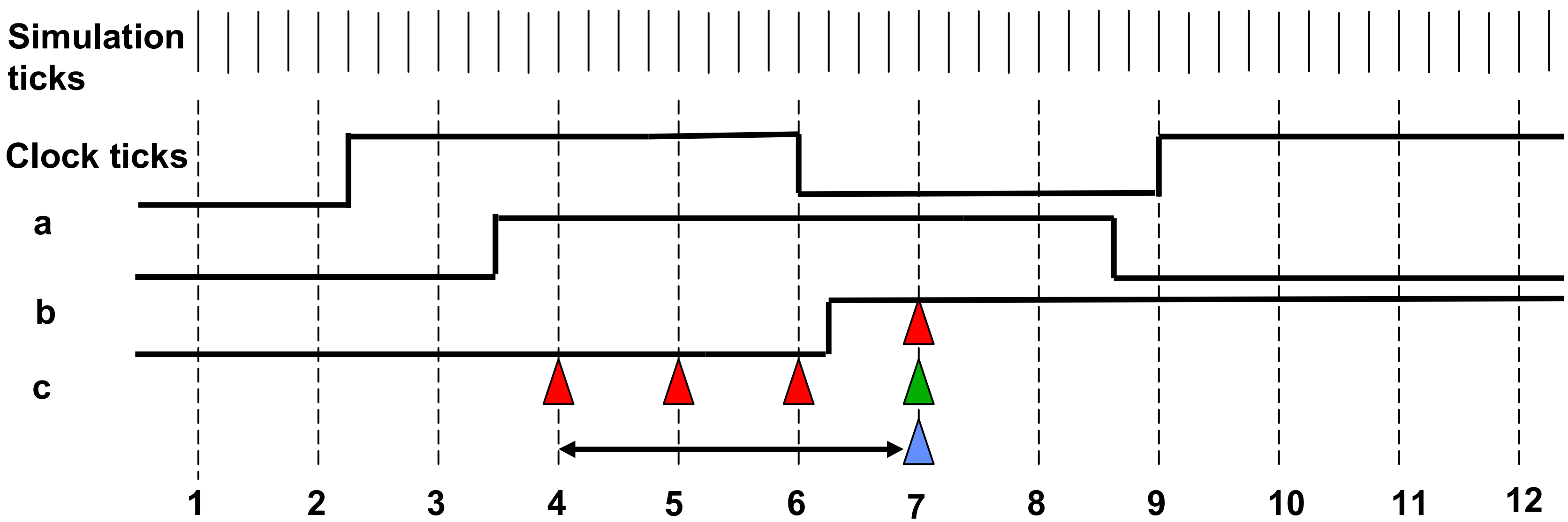
- The binary operator intersect is used when both operand expressions are expected to succeed
- End times of the operand expressions must be the same
- Length of the two operand sequences must be same

Example:

(a ##1 b) intersect (a ##1 b ##2 c)

Intersection – contd..

(a ##[1:3] b) intersect (a ##1 b ##2 c)

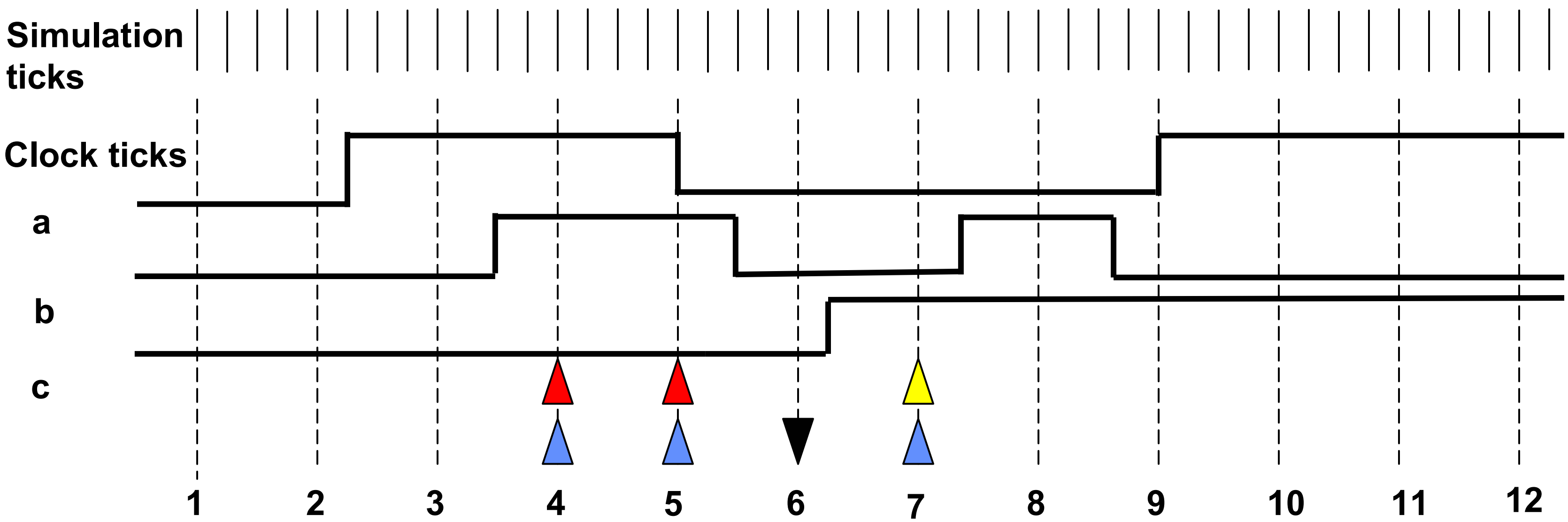


OR - operation

- The binary operator and is used when at least one of the operand expressions is expected to match
- End timed of the operand can be different

  Example:

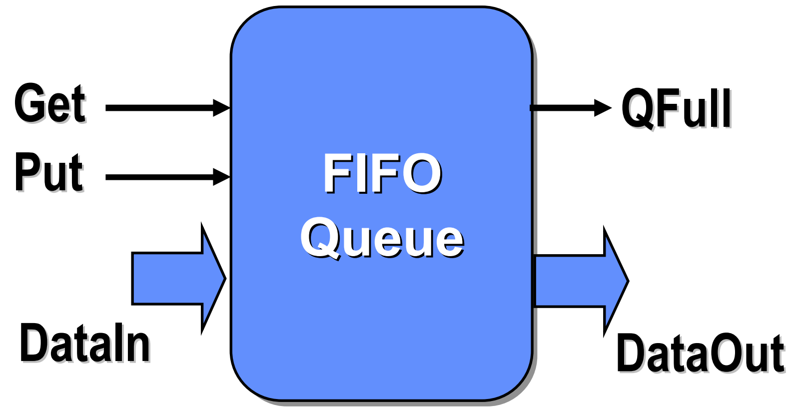
  (a ##1 b) or (a ##1 b ##2 c)



Local Variables

- *If X and Y are any two data items such that X was received before Y, then X will come out of the queue before Y*

```
property FIFO_check;
   int x;
   int y;
   @( posedge clk )
   (( Put && !QFull, x = DataIn ) ##[1,$]
     ( Put && !QFull, y = DataIn )) |->
        ##[1,$] (( Get && x == DataOut ) ##[1,$]
                              (Get && y == DataOut )) ;
endproperty
```



The property definition

- A property defines a behavior of the design.

- A property can be used
  - As an assumption
  - As a checker
  - As a coverage specification

```
property p;
  @(posedge clk) seq1;
endproperty;
```

Properties and Implication

- **Use of *if-then-else*:**

```
property P;
    @(posedge clk)
    if (r1) then ##1 (g1 && !r1) else ##1 g2;
endproperty
```

- **The condition of *if* cannot be a sequence expression:**

```
property ThisIsNotOkay ;
    @(posedge clk)
    if (r2 ##1 (!g2 && r2) ##1 !g2) then ##1 !r2;
endproperty
```

  - **Can be written as:**

```
property ThisIsOkay ;
        @(posedge clk)
        r2 ##1 (!g2 && r2) ##1 !g2 |→ ##1 !r2;
endproperty
```

Two types of implication

- **Overlapped Implication Operator:**
  - In the property, s1 |→ s2, the match of s2 starts from the same cycle as the one in which we complete a match for s1.

- **Non-overlapped Implication Operator:**
  - In the property, s1 |=> s2, the match of s2 starts from the cycle *after* the one in which we complete a match for s1.

Use of *DisableIff*

- y must be asserted within 16 cycles of x, unless reset is asserted in between

```
property DisableOnReset;
        @(posedge clk)
        disable iff (reset) x |→ ##[1:16] y;
endproperty
```

Immediate and Concurrent Assertions

- **Immediate Assertions**
  - Immediate assertions follow simulation event semantics for their execution
  - Immediate assertions are executed like a statement in a procedural block

```
assert (expression) Action_block
Action_block ::= statement_or_null | [statement] else statement
```

- **Concurrent Assertions**
  - Describe behavior that spans over time
  - Evaluation model is based on a clock
  - The values of variables used are the sampled values in the specified clock edge

```
prop_p1: assert property (p1) pass_stat else fail_stat
```

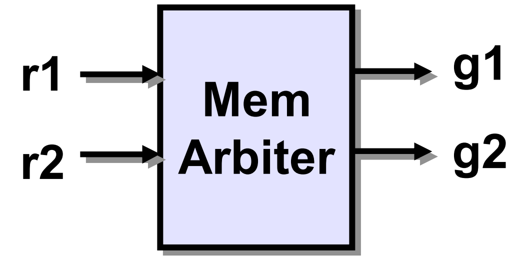
Property Usage

A property can be used

- **As an assertion (guarantee)**
  - **We call them *assert properties***
- **As an assumption**
  - **We call them *assume properties***
- **As a coverage specification**
  - **We call them *cover properties***

What are *assume* properties?

- **Example:** *Every low priority request, r2, is eventually granted by the arbiter*




```
property NoStarvation;
  @(posedge clk) r2 |→ ##[1:$] g2 ;
end property
```

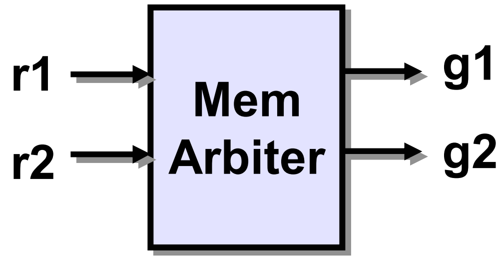
- **This requirement conflicts with our earlier property P1:**

```
property P1;
  @(posedge clk) r1 |→ ##1 g1 ##1 g1;
endproperty
```

- **Suppose we are now given that whenever g1 is asserted, r1 remains low for the next 4 cycles**

```
property FairnessOfr1;
  @(posedge clk) g1 |→(!r1) [*4] ;
endproperty
```

Assume properties




```
property FairnessOfr1;
    @(posedge clk) g1 |→(!r1) [*4] ;
endproperty
AssumeR1IsFair: assume property (FairnessOfr1);

property NoStarvation;
    @(posedge clk) r2 |→ ##[1:$] g2 ;
endproperty
AssertNoStarvation: assert property (NoStarvation);
```

- Under assumption AssumeR1IsFair, there is no conflict between the properties GrantWhenRequest and AssertNoStarvation

```
property P1;
  @(posedge clk) r1 |→ ##1 g1 ##1 g1;
endproperty
GrantWhenRequest: assert property (P1);
```

Assume versus Assert

- Both *assume* and *assert* properties may use input and output variables of the DUT
- The *assume* properties are not related to any specific *assert* property – they are generic assumptions about behaviors
- In dynamic assertion verification, both the *assume* and *assert* properties are checked over the simulation run
  - If one or more *assume* properties fail, then the monitoring of the *assert* properties become redundant
- In formal property verification, *assume* properties may be used to prune the state space before checking the *assert* propeties

Cover properties

```
property P4;
  @(posedge clk) !r1 ##1 !r1 |→ ##1 !g1;
endproperty
```

- The property is interpreted non-vacuously only when r1 is low in two consecutive cycles

- Cover property:

```
property P4;
  @(posedge clk) !r1 ##1 !r1 |→ ##1 !g1;
endproperty
cover property (P4)
```

Coverage Results

- Coverage Results are divided into
  - Coverage for properties
  - Coverage for sequences
- The results of coverage statement for a property contain:
  - Number of times attempted
  - Number of times succeeded
  - Number of times failed
  - Number of times succeeded for vacuity
  - Each attempt with an attemptID and time
  - Each success/failure with an attemptID and time
- Vacuity rules are applied only to the implication operator

Multiple clock support

- Multiple clock is allowed in
  - Concatenation of two sequences, where each sequence can have a different clock

```
sequence s1;
@(posedge clk0) sig0 ## @(posedge clk1) sig1;
endsequence
```

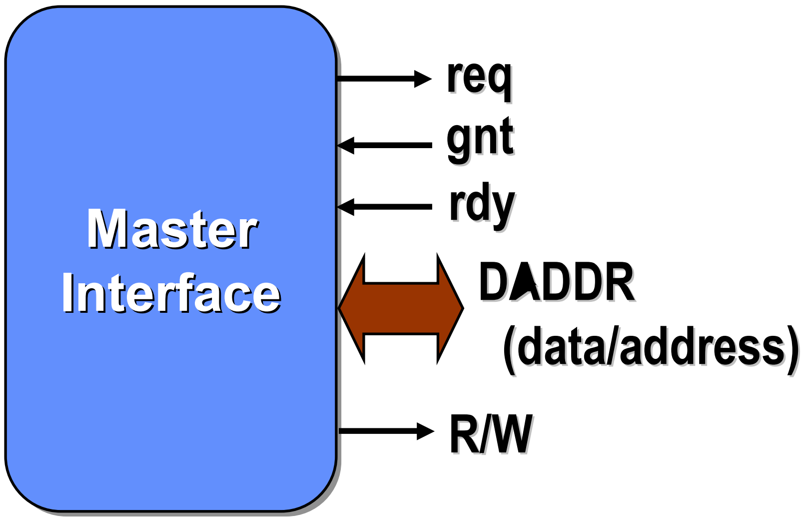
  - The antecedent of an implication on one clock, while the consequent is on another clock

```
property s2;
@(posedge clk0) sig0 |=> @(posedge clk1) sig1;
endproperty
```

Architectural Styles for Assertion IPs

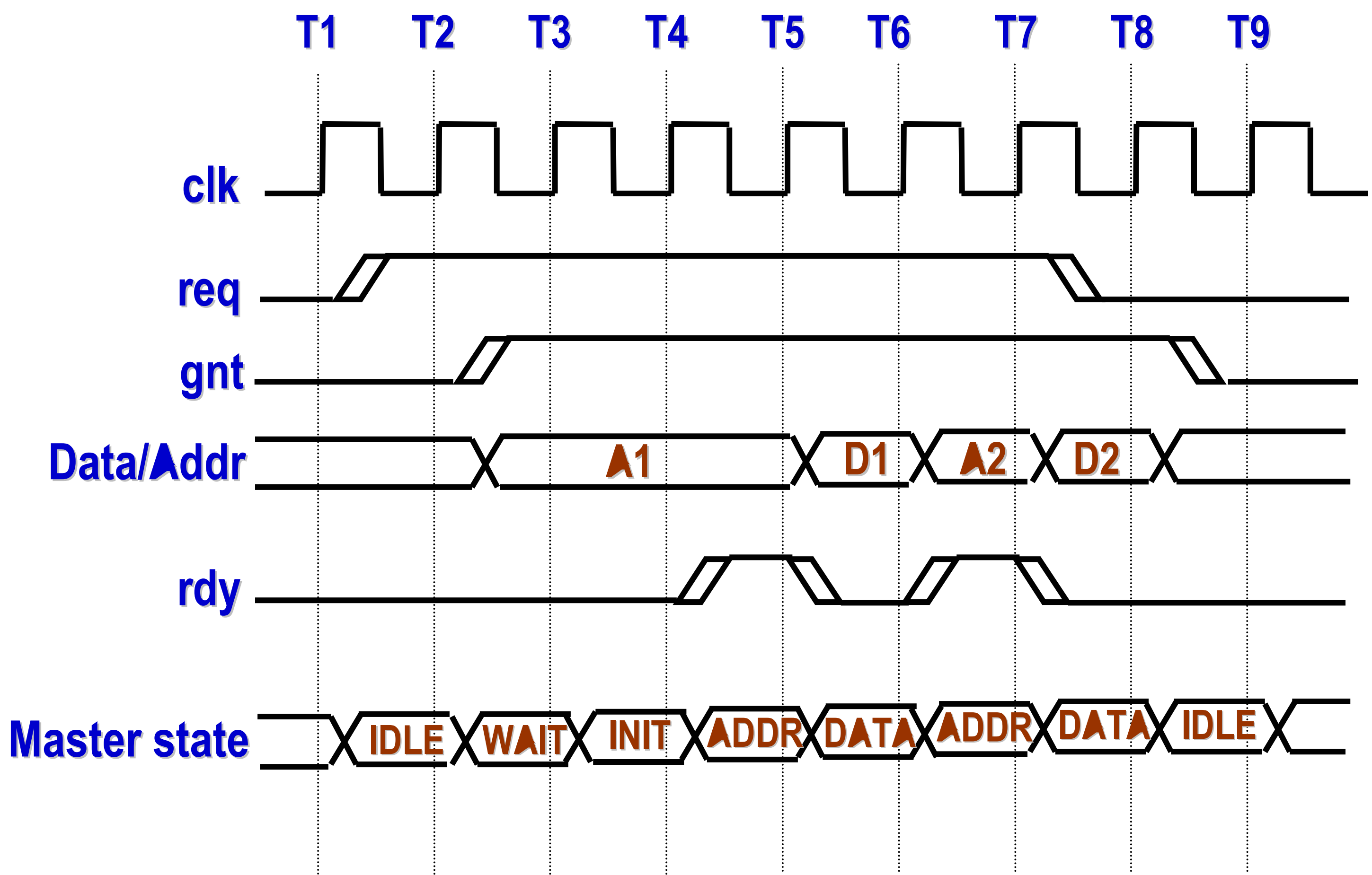
- Event-based Specifications
  - Only properties defined over interface signals
- State-based Specifications
  - Auxiliary state machines (ASM)
  - Properties specified using state-bits of ASM and interface signals

The MyBus Protocol



- Address and data multiplexed
- Master asserts req, waits for gnt
- Address Cycle: Then it floats the address and waits for rdy from slave
- Data Cycle: On receiving rdy, it expects data in next cycle (if READ), or floats data in next cycle (if WRITE)
- R/W indicates intent: read/write
- After each data cycle, the master may start another address cycle by floating the next address

A Sample Transfer



Properties

- The protocol is non-preemptive. Once granted, the master owns the Bus until it lowers its *req* line
- If the master is in the ADDRESS cycle, it should not change the address floated in the Bus until it receives the *rdy* signal from the slave
- Each DATA cycle is of unit cycle duration

Event-based Coding

- The protocol is non-preemptive. Once granted, the master owns the Bus until it lowers its *req* line

```
property NoPreemption;
  @(posedge clk) $rose(gnt) |→ ##1 gnt [*1:$] ##0 !req ;
endproperty
```

  - $rose(gnt) is true in a cycle if the signal *gnt* rose in that cycle

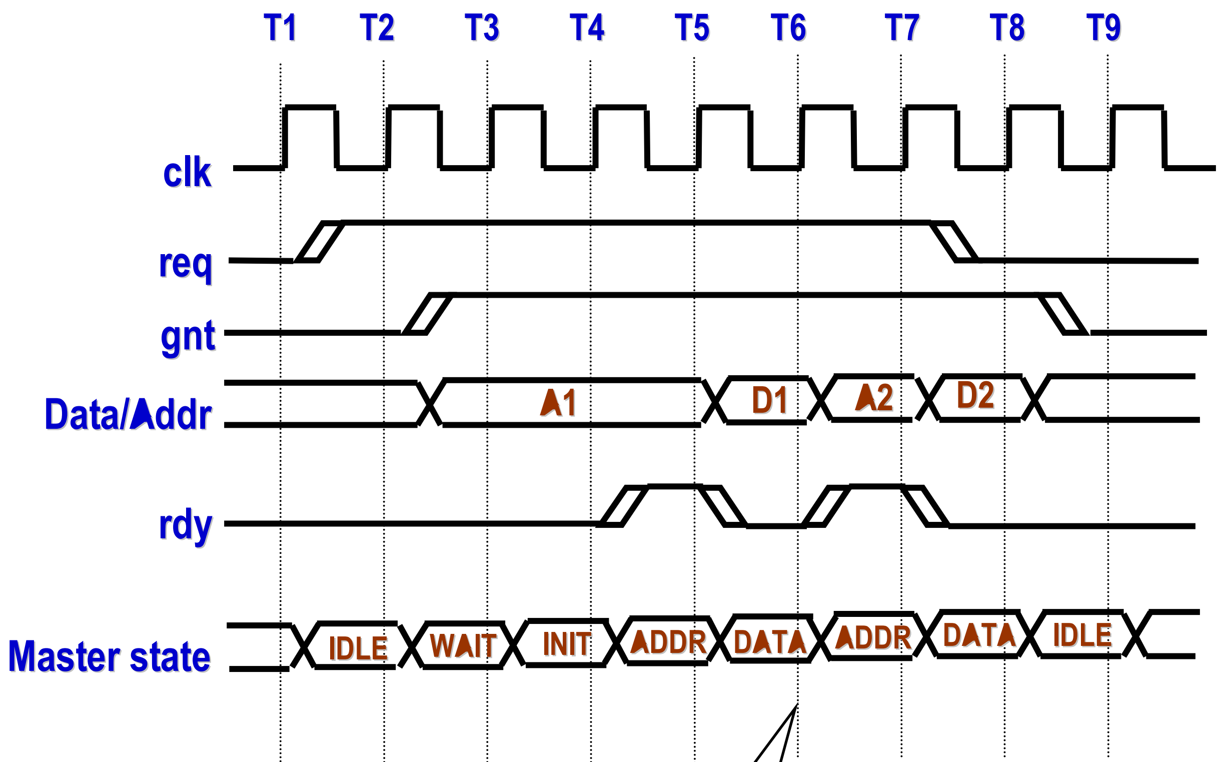
Event-based Coding

- If the master is in the ADDRESS cycle, it should not change the address floated in the Bus until it receives the *rdy* signal from the slave

```
property IncorrectAddressStable;
  int x;
  @(posedge clk) (req && gnt && !rdy, x = DADDR)
             |→ ##1 (x == DADDR) ;
endproperty
```

- This coding is not correct, since (req && gnt && !rdy) may be true at other places also.

The problem




```
property IncorrectAddressStable;
int x;
@(posedge clk) (req && gnt && !rdy, x = DADDR) |→ ##1 (x == DADDR) ;
endproperty
```

The context is important

- What's the problem with this property?

```
property IncorrectAddressStable;
  int x;
  @(posedge clk) (req && gnt && !rdy, x = DADDR)
              |→ ##1 (x == DADDR) ;
endproperty
```

  - We want to check this property only in the ADDRESS cycles, not in the DATA cycles
  - How should be distinguish between an ADDRESS cycle and a data cycle?

```
property AddressStable;
  int x;
  @(posedge clk) (req && gnt && !rdy && !$fell(rdy), x = DADDR)
              |→ ##1 (x == DADDR) ;
endproperty
```

Event-based Coding

- Each DATA cycle is of unit cycle duration

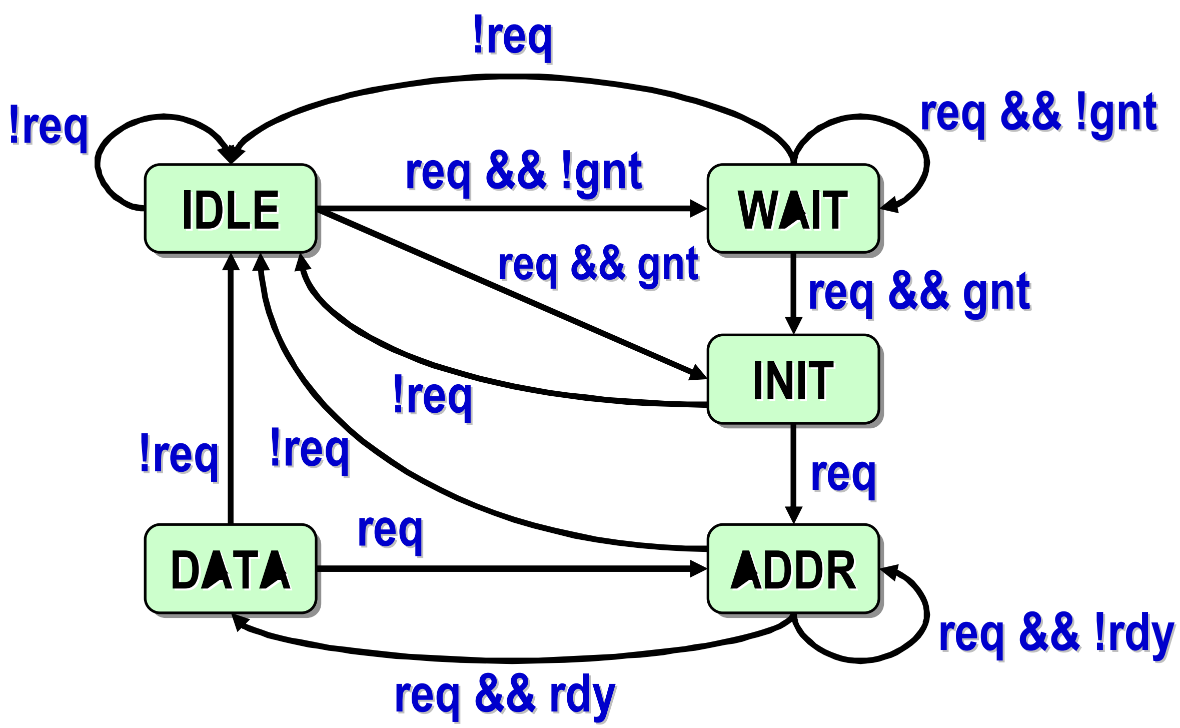
```
property SingleCycleDataTransfer;
  @(posedge clk)
      (gnt && $fell(rdy)) |-> ##1 (!gnt || !$fell(rdy)) ;
endproperty
```

- The expression (gnt && $fell(rdy)) characterizes a DATA cycle. *Not obvious*!!

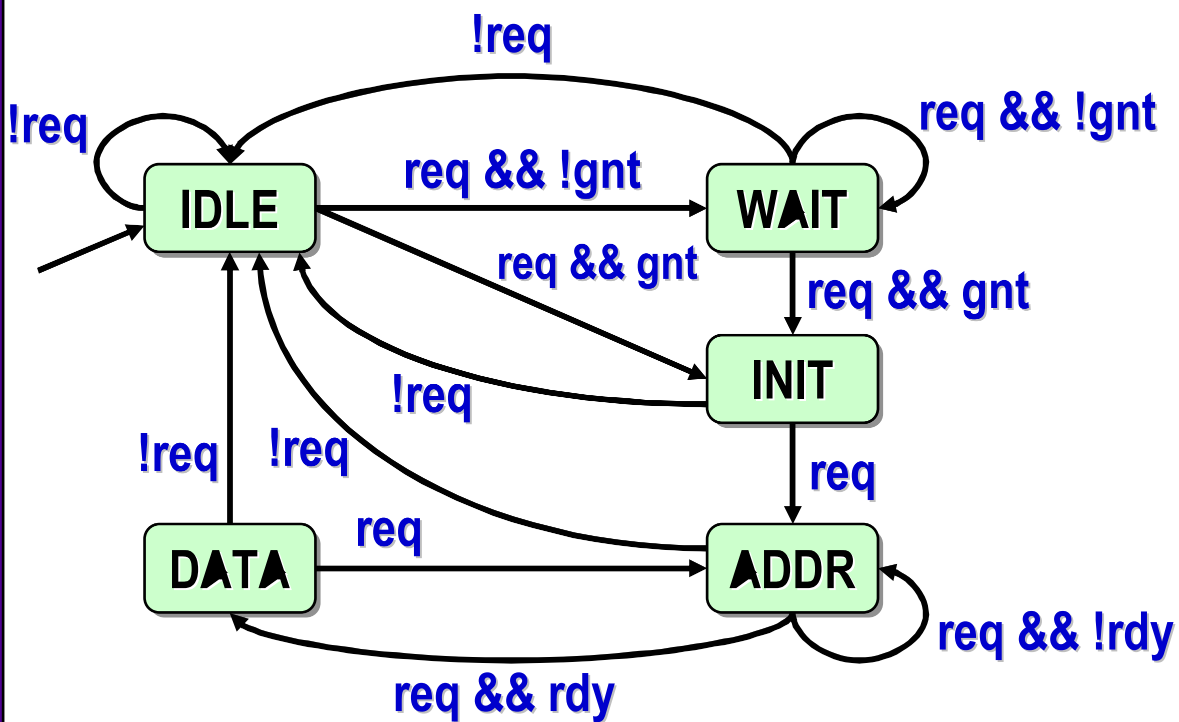
State-based Coding

- Characterizing the context is a major problem in event-based coding
- In state-based coding we use an auxiliary state machine to capture the contexts and the transitions between them
  - We use the state labels for coding the actual properties
  - Improves readability
  - Reduces coding errors

Auxiliary State Machine Example



State-based Coding




```
property SingleCycleDataTransfer;
  @(posedge clk)
  (state == 'DATA) |→ ##1 !(state == 'DATA) ;
endproperty
```

```
property AddressStable;
  int x;
  @(posedge clk)
  (state == 'ADDR, x = DADDR)
                |→ ##1 (x == DADDR) ;
endproperty
```

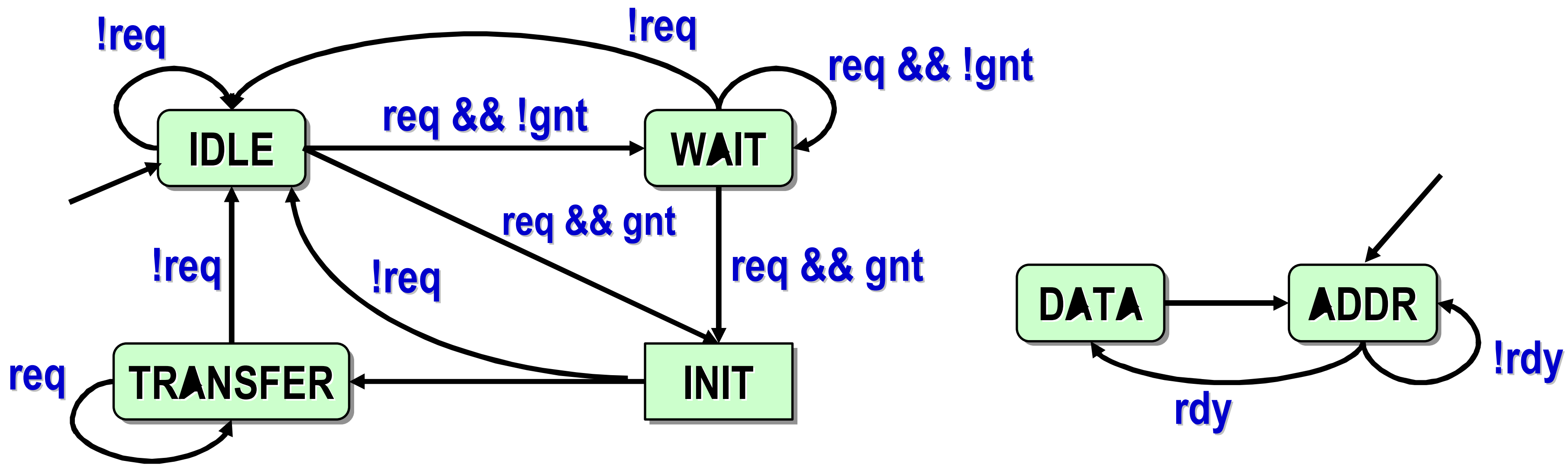
Encoding the Auxiliary State Machine

```
interface MasterInterface( input req, gnt, rdy, clk, int DADDR) ;
logic [2:0] state;
'define IDLE 3'b000
'define WAIT 3'b001
'define INIT 3'b010
'define ADDR 3'b011
'define DATA 3'b100
always @( posedge clk )
  case (state)
    'IDLE: state <= req? (gnt? 'INIT : 'WAIT) : 'IDLE;
    'WAIT: state <= req? (gnt? 'INIT : 'WAIT) : 'IDLE;
    'INIT: state <= req? 'ADDR : 'IDLE;
    'ADDR: state <= req? (rdy? 'DATA : ' ADDR) : 'IDLE;
    'DATA: state <= req? 'ADDR : 'IDLE;
  endcase
initial begin state = 'IDLE; end
```

State encoding (the `define lines)

State transition relation (the case statement)

Factored State Machines




```
property AddressStable;
   int x;
   @(posedge clk) (state1 == ‘TRANSFER && state2 == ‘ADDR, x = DADDR)
                  |→ ##1 (x == DADDR) ;
endproperty
```

```
property SingleCycleDataTransfer;
   @(posedge clk)
   (state1 == ‘TRANSFER && state2 == ‘DATA) |→ ##1 !(state2 == ‘DATA) ;
endproperty
```