Формалне методе пројектовања и верификације хардвера

Вук Врањковић

Факултет техничких наука, Нови Сад

bykbpa@gmail.com

12. новембар 2015.

Преглед

Предмет

- Организација
- Полагање

Потребна/стечена знања

- Потребно добро познавање језика VHDL
- Писања особина у језику PSL
- Пројектовање и верификација RTL модела помоћу алата IFV
- Пројектовање модела базирано на “assertion”
- Теоријске основе рада алата који се користе у формалној верификацији

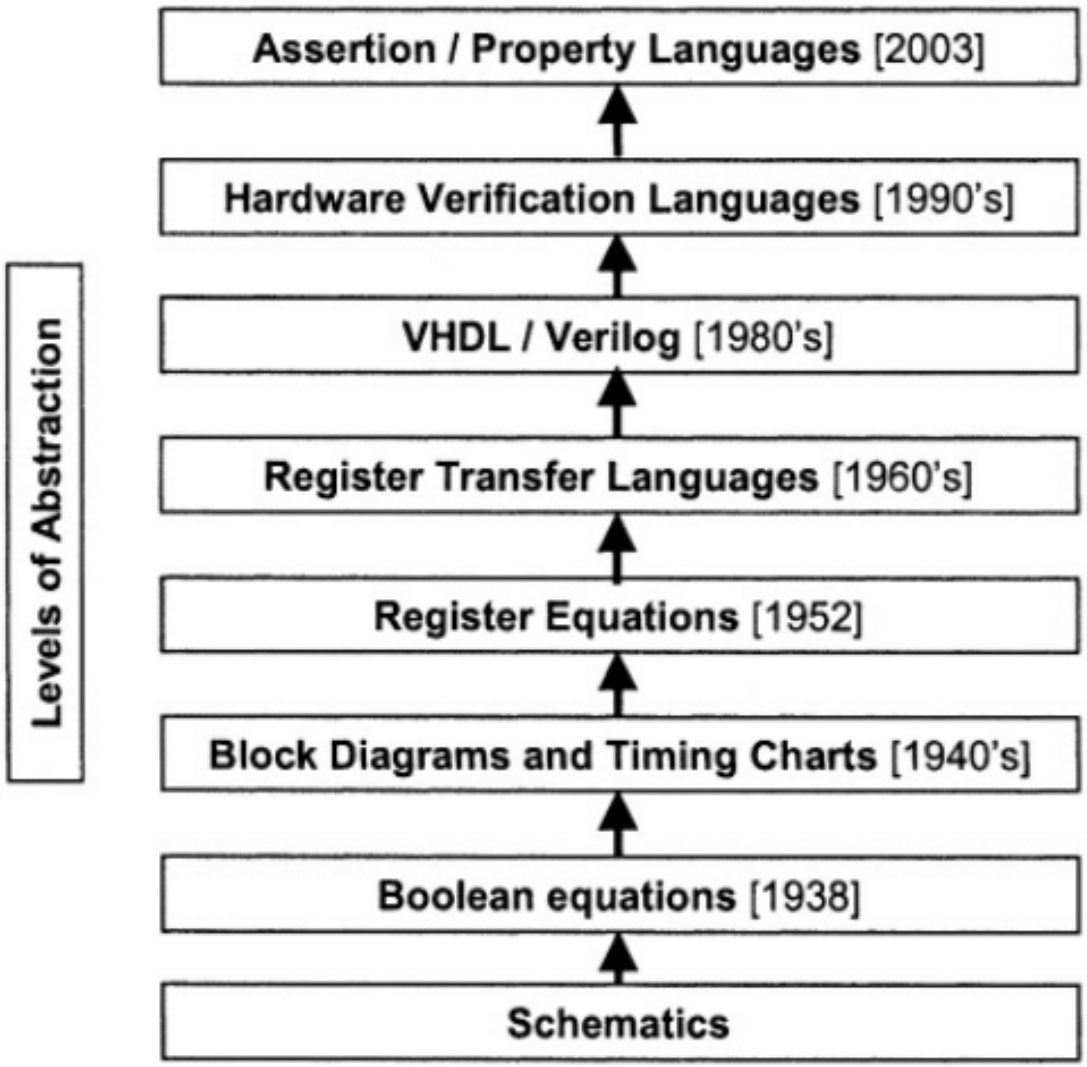
Почетни фокус

- Методологију базирану на assertion исказима ћемо посматрати из угла RTL дизајнер и верификационог инжењера
- Теоријску позадину рада алата за формалну проверу модела ћемо дискутовати у другом делу предмета

Провера особина

- Особина је очекивано понашање дизајна
- Алати за формалну верификацију статички анализирају и доказују да ли особине важе за дизајн
- Неке особине се могу аутоматски извући из дизајна (достижна стања)
- Комплексне особине се морају описати
- До 2000 године није било језика за формалан опис особина

Историјски поглед на нивое абстракције



Верификационе технике

- Традиционално се користи “black box” приступ
- Ручно => аутоматско проверавање резултата
- Ручно => аутоматско стварање стимулуса
- Тест бенчеви постају комплексна верификациона окружења
- Мана “black box” приступа
- “White box” приступ и “assertion”

Assertion у HDL

- VHDL: assert ((a = ‘1’) and (b = ‘1’)) report “error: something bad happened.” severity 0;
- Verilog нема
- SystemVerilog има читав део језика посвећен assertion - SVA

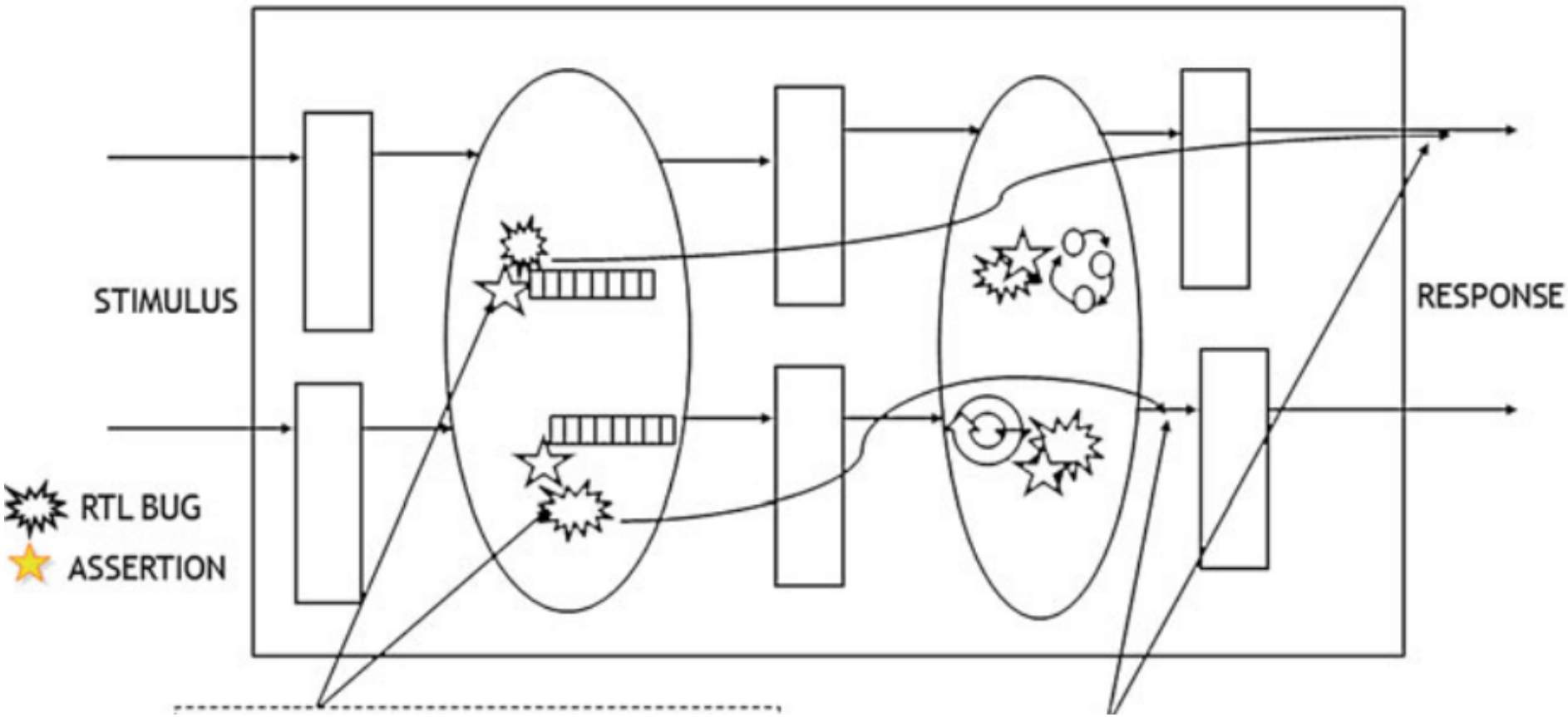
Статистика употребе assertion-a

- Компаније
  - Cisco Systems, Inc.
  - Digital Equipment Corporation
  - Hewlett-Packard Company
  - IBM Corporation
  - Intel Corporation
  - LSI Logic Corporation
  - Motorola, Inc.
  - Silicon Graphics, Inc.
- Статистика употребе
  - 34% of all bugs were found by assertions on DEC Alpha 21164 project
  - 17% of all bugs were found by assertions on Cyrix M3(p1) project
  - 25% of all bugs were found by assertions on DEC Alpha 21264 project - The DEC 21264 Microprocessor
  - 25% of all bugs were found by assertions on Cyrix M3(p2) project
  - 85% of all bugs were found using OVL assertions on HP

Предности Assertion-a #1

- Унапређење видљивости унутар дизајна
  - White box - Black box
  - Observability
  - Грешка се види брже и ближе делу дизајна где се десила
- Скраћује се време верификације

Предности Assertion-a #2



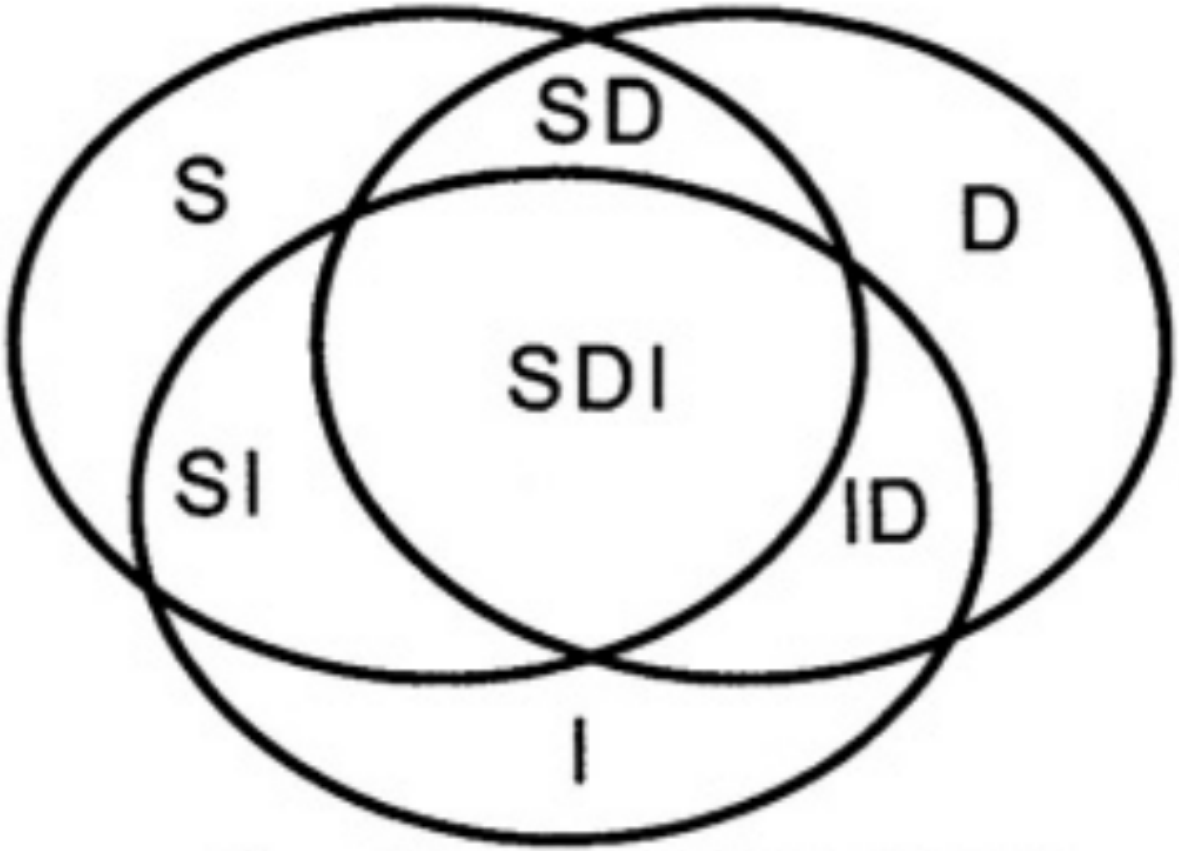
Предности Assertion-a #3

- Аутоматска провера интерфејса
- Провере раде стално
- Грешке се проналазе раније у циклусу развоја
- Пишу их и дизајнери
- Када се користи формални алат, провера је потпуна

Мане Assertion-а

- Продужују време писања дизајна
- Потребно је време да се нађу грешке и у самим assertion исказима
- Велики ментални напор
- Нова методологија, споро се усваја

Фазе процеса дизајнирања



S – Specification intent

D – Architect/Design intent

I – RTL Implementation intent

Специфицирање RTL особина

- Неформално, особина је генерални атрибут понашања којим се карактерише дизајн
- Формално: A collection of logical and temporal relationships between and among subordinate Boolean expressions, sequential expressions, and other properties that in aggregate represent a set of behavior [Accellera PSL-1.1 2004].
- Пример: након сваког REQ активира се ACK
- PSL - Property Specification Language
  - SVA - SystemVerilog Assertion
  - OVL - Open Verification Language

Структура особине (property)

- Булов ниво, Boolean layer
- Временски ниво, Tempolar layer
- Ниво моделовања, Modeling layer
- Верификациони ниво, Verification layer

Булов ниво (Boolean layer)

- Булов израз који се састоји од Булових варијабли које се користе у дизајну.
- Овај ниво се може исказати у два “укуса” (flavor), VHDL или Verilog
- Пример: Сигнали “a” и “b” су узајамно искључиви
  - VHDL : not (a and b);
  - Verilog : ! (a & b);
- Не постоји никаква информација о времену.

Временски ниво (Tempolar layer)

- Додаје се временска зависност између Булових израза
- Пример: Сигнали “a” и “b” су увек узајамно искључиви
  - always not (a and b);
  - always ! (a & b);

Ниво моделовања (Modeling layer)

- Додају се помоћни сигнали који могу да олакшају писање особина
- Сви језичке конструкције HDL језика за моделовање RTL дизајна су допуштене
- Пример: Сигнали “a” и “b” су увек узајамно искључиви
  - c <= a and b;
  - always not c;
  - assign c = a & b;
  - always ! c;

Верификациони ниво (Verification layer)

- Нижи нивои моделовања описују каква је особина генерално али не одређују како се она користи током верификације
- Особина се можре користити тако што ће се одредити да она треба да важи али може и у друге сврхе
  - Assertion
  - Constraint
- Пример: Потврди да су Сигнали “a” и “b” увек узајамно искључиви
  - assert always not (a and b);
  - assert always ! (a & b);
- Која кључна реч одговара ком нивоу моделовања?

Класификација особина

- Класификација на основу временског нивоа
- Класификација на основу верфикационог нивоа
- Класификација на основу места писања особина

Класификација на основу временског нивоа

- Safety
- Пример: Сигнали “a” и “b” увек узајамно искључиви
- Liveness
- Пример: Након сваког “req” кад тад уследиће “ack”

Класификација на основу верфикационог нивоа

- Assertions
- Пример: Потврди да су сигнали “a” и “b” увек узајамно искључиви
- Constrains
- Пример: Ограничи сигнале “a” и “b” да буду узајамно искључиви

Класификација на основу места писања особина

- Procedural
- Особина се налази у секвенцијалном RTL коду, најчешће у некој угњежденој if или case наредби
- Concurent
- Особина је посебан процес који се одвија у паралели са свим осталим процесима

Начини специфицирања неких особина коришћењем PSL-а

- Подразумевани синхронизациони сигнал
- Специфицирање ресет сигнала
- Safety
- Вишеструко коришћење дефинисаних особина
- Специфицирање догађаја у следећем циклусу

Подразумевани синхронизациони сигнал

- У сваком упиту се мора дефинисати синхронизациони сигнал у односу на који се броје догађаји
- Пример: assert always not (a and b) @risning_edge(clk);
- У PSL-у се може дефинисати за сваки модул подразумевани синхронизациони сигнал.
- default clock is rising_edge(clk);
- Потом се упити у одговарајућем модулу могу писати без специфицирања “clock” сигнала
- Пример: assert always not (a and b);

Специфицирање ресет сигнала

- Док је дизајн у ресету обићно особине не треба важе
- У PSL језику се ово може специфицирати abort конструкцијом
- Пример: assert always not (a and b) abort rst;
- Алати за формалну верификацију допуштају да се ресет сигналом рукује на посебан, другачији начин
- У IFV алату помоћу TCL команди може се у потпуности контролисати ресет

Специфицирање safety особине

- FIFO се никада неће препунити
- assert never ovr = '1';
- Никада се неће читати и писати у исту локацију у FIFO
- assert never (rd_adr = wr_adr) and en_rd and en_wr;
- Адресе су увек валидне
- assert always adr < valid_range

Вишеструко коришћење дефинисаних особина

- Особин се може дефинисати...
- Пример: property mutex(a, b) = always not (a and b);
- ... и потом поново користити
- assert mutex(as, bs);

Специфицирање догађаја у следећем циклусу

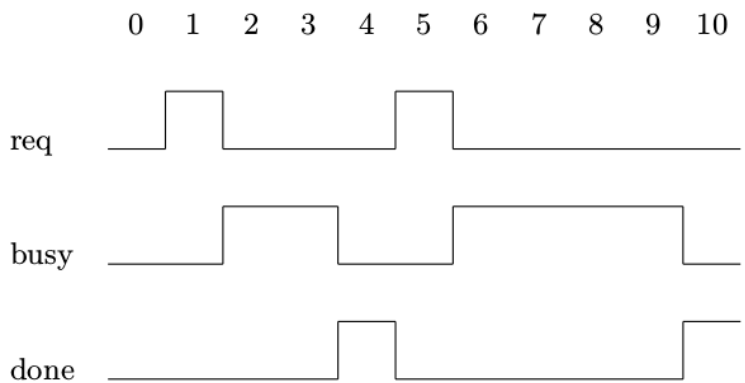
- Ако је активан сигнал req, наредни циклус ће бити активан сигнал ack
- Овај упис се лако може специфицирати коришћењем PSL оператора next и логичке импликације
- assert always (req -> next ack);
- Шта би значиле следеће особине:
- assert always (req -> next (next ack));
- assert always (req -> next[3] ack);
- Алат IFV не поджрава све операторе језика PSL.

PSL оператори

- until, before
- eventually!

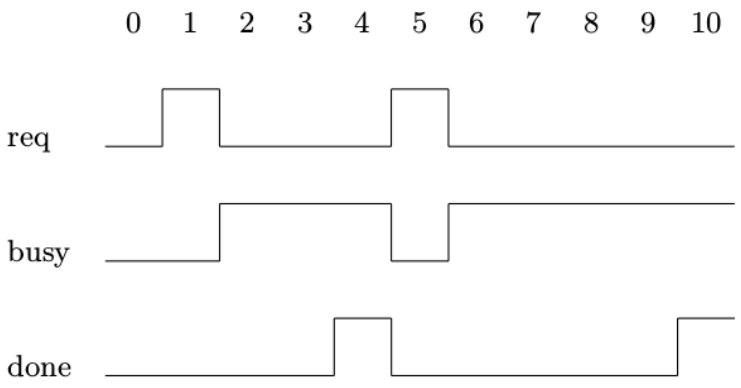
until #1

- Након сваког “req” сигнал “busy” треба да је активан све док се “done” не актвира
- Подразумева се да су “req” и “done” пулсеви.
- Пример 1: assert always (req -> next (busy until done));
- Пример 2: assert always (req -> next (busy until_ done));

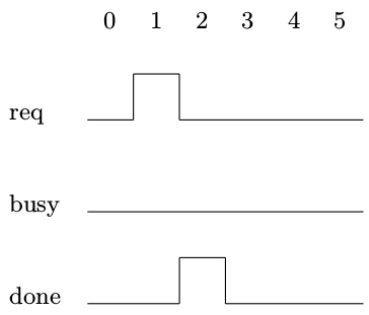


- T, F

until #2



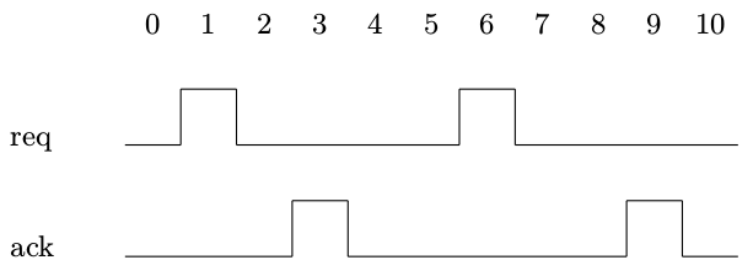
- T, T



- T, F

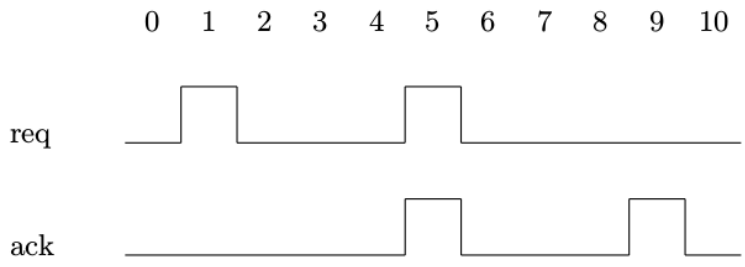
before #1

- Сваки “req” мора бити праћен са “ack” пре новог “req”
- Пример 1: assert always (req -> next (ack before req)); – good
- Пример 2: assert always (req -> (ack before req)); – bad
- Пример 3: assert always (req -> next (ack before_ req)); – good
- Пример 4: assert always (req -> (ack or next (ack before req))); – good

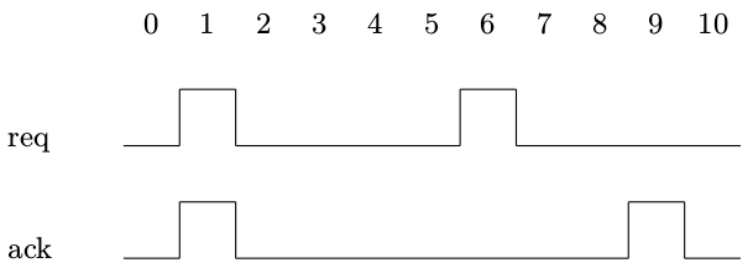


- T, F, T, T

before #2



- F, F, T, F



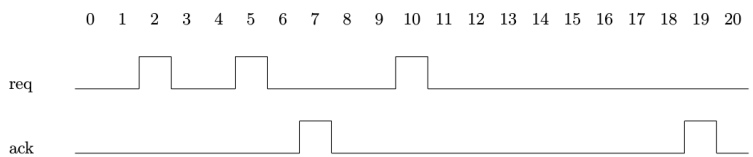
- F, F, F, T

until, before вежба

- Сигнали “a” и “b” треба да су активни све дој је “c” неактиван.
- Након сваког “a” следи “b” пре “c”. Сви сигнали су пулсеви.
- Два циклуса након што се “req” активирао, “busy” мора да је активан, све док се “done” не активира, укључујући и циклус када се “done” активирао.

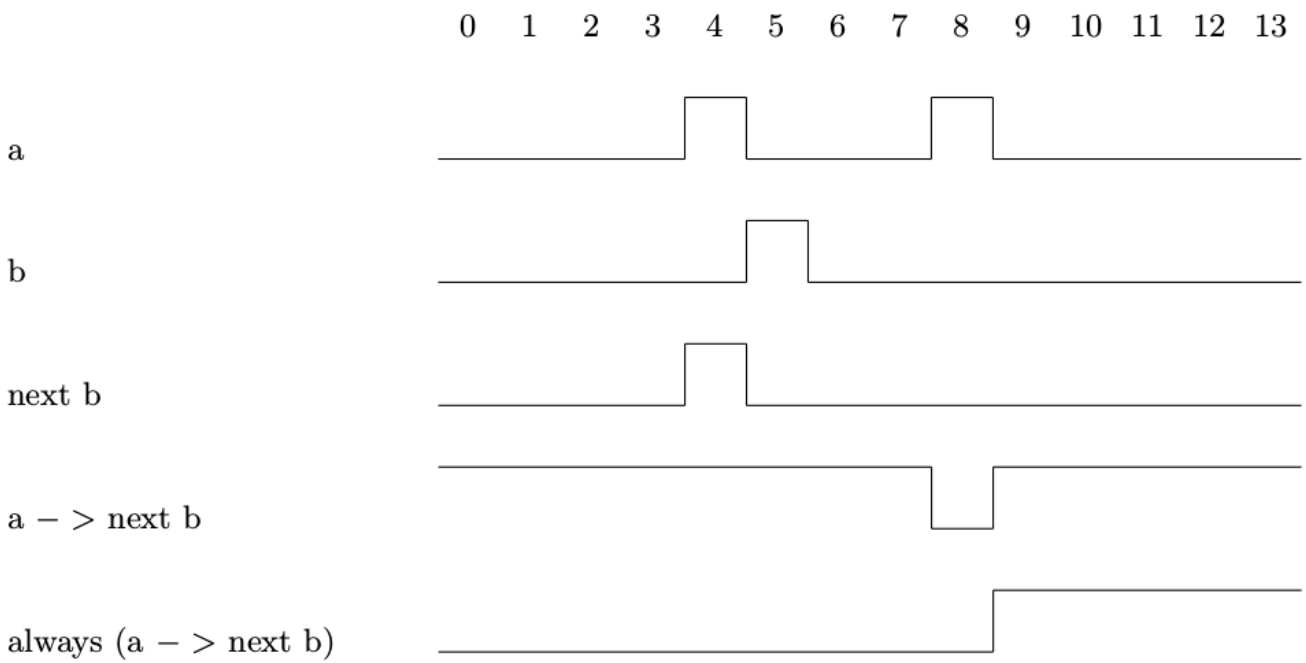
eventually!

- Сваки “req” ће бити праћен са “ack”.
- Коментар: Не пише да сваки “req” мора бити праће тачно једним “ack”.
- Пример: assert always (req -> eventually! ack);



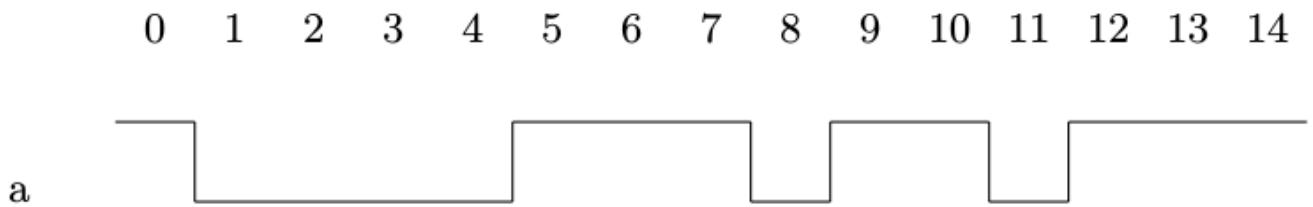
Ток прорачуна особине #1

- Прорачун, доказ, особине почиње увек од тренутка 0
- Пример: assert always (a -> next b);



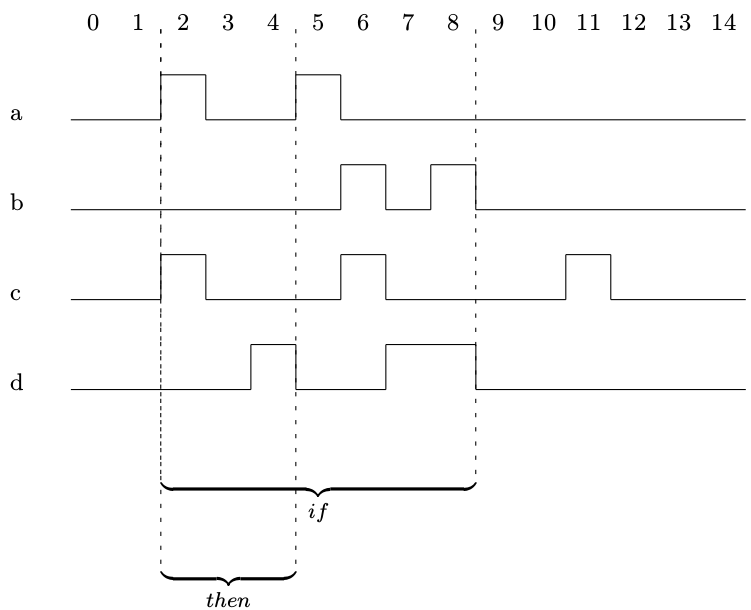
Ток прорачуна особине #2

- Зашто је always тако битан...
- Која од наредних особина важи за таласни дијаграм?
- assert a;
- assert always a;
- assert next[12] (always a);



Ток прорачуна особине #3

- Прорачун се увек одвија у тренутном циклусу
- Понекад, у случају необично написаних особина не мора бити тако очигледно
- Пример: assert always ((a and next[6](b)) -> (c and next[2](d)));

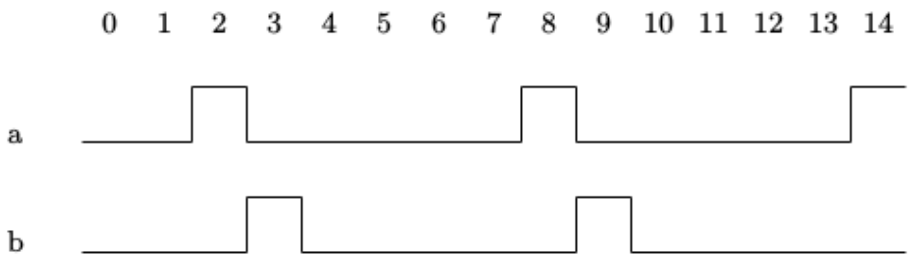


Слаби и јаки оператори

- Јаке верзије оператора имају !
- У формалној верификацији нема разлике између ове две врсте оператора
- Ови оператори се разлику у функционалној верификацији
- Слаба верзија оператора ће важити уколико се секвенца не заврши
- Јака верзија оператора неће важити уколико се секвенца не заврши

Слаби, јаки next

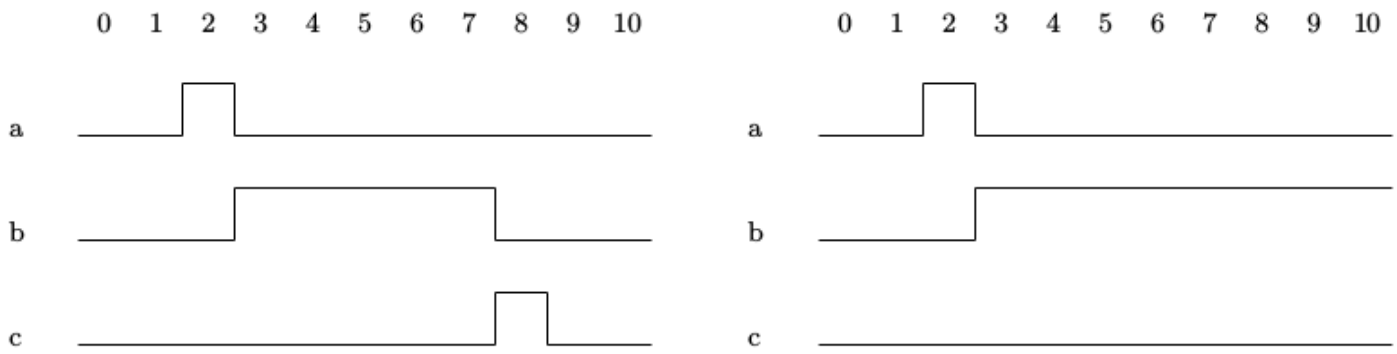
```
assert always (a -> next b);  // 1.
assert always (a -> next! b); // 2.
```



1. T
2. F

Слаби, јаки until

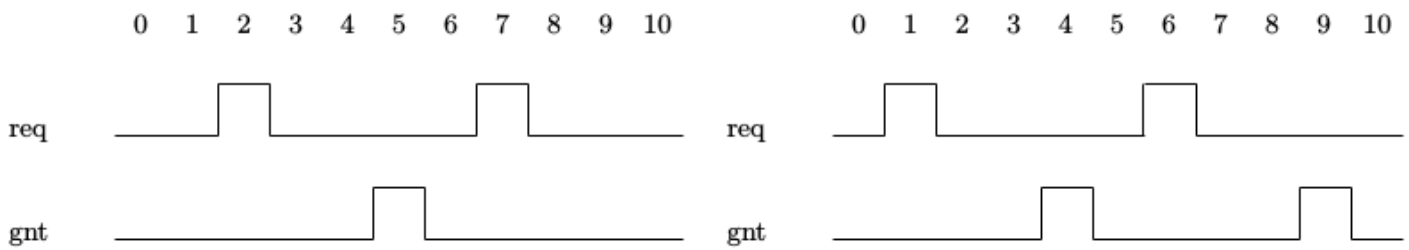
```
assert always(a -> next (b until c));
assert always(a -> next (b until! c));
```



1. T T
2. T F

Слаби, јаки before

```
assert always (req -> next (gnt before req));
assert always (req -> next (gnt before! req));
```



1. T T
2. F T

PSL начини упита

- PSL = Foundation Language (FL) + Optional Branching Extension (OBE)
- FL = Linear Time Language (LTL) + Sequential Extended Regular Expressions (SERE)
- Сви оператори до сада су припадали LTL делу језика: next, until...
- OBE је заправо Computation Tree Logic (CTL)
- IFV не подржава OBE

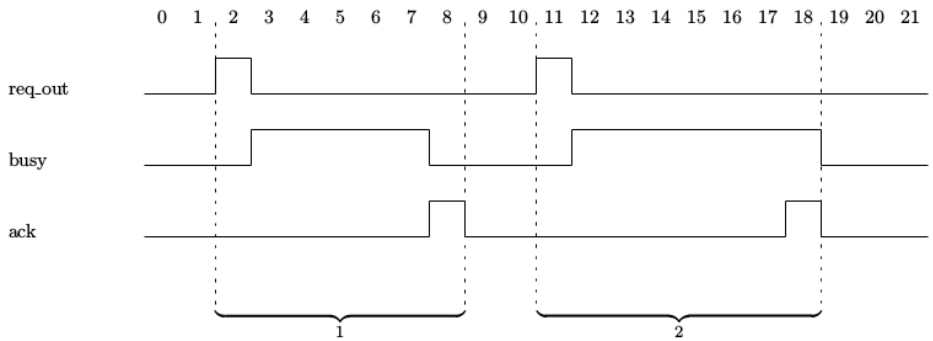
SERE генерално

- Упити подсећају на регуларне изразе у програмским језицима
- Упити су затворени у {}
- Свака SERE је особина, али свака особина није SERE
  - always ((req and gnt) -> next {(req and not ack) ; (busy and not ack)[*] ; ack}); // good
  - always {a; b until c; a}

  // bad
- Пример: {a;b;c}

Пример SERE

```
{(req out && !ack) ; (busy && !ack)[*] ; ack}
{(req out && !ack) ; (busy && !ack)[*] ; ack && !busy}
```



Упит #1

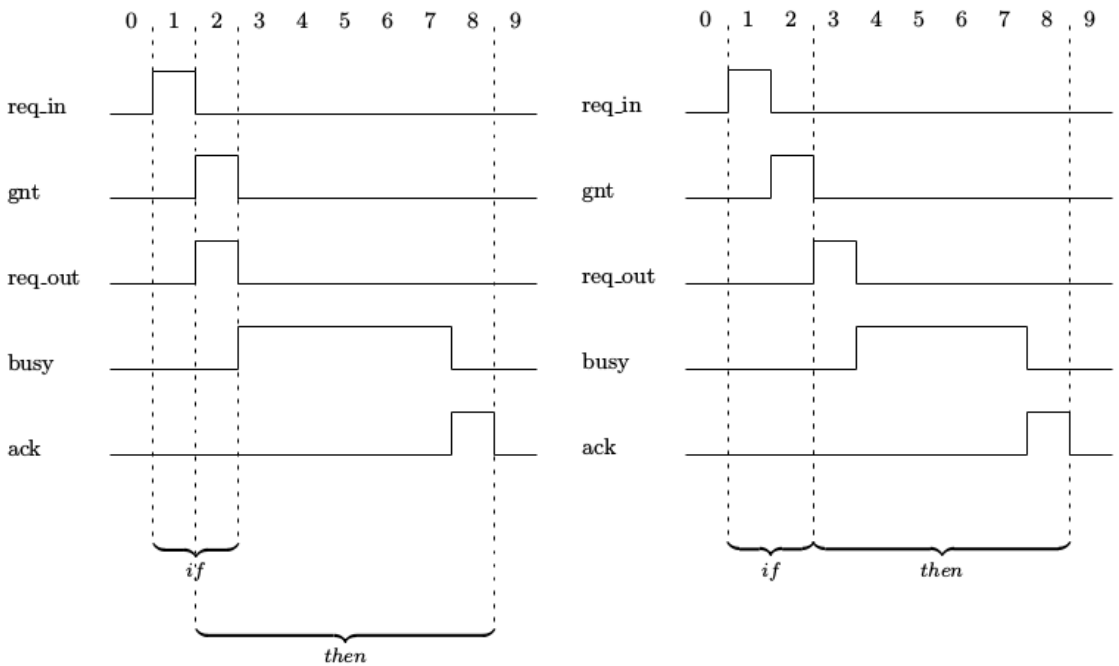
- Исказати као SERE наредне особине:
- Након што је a активно и b није, следи два циклуса када је b активно док a није.
- Сигнал c је пулс.
- Не сме се десити да је a три циклуса узастопно неактивно.

Суфиксна импликација

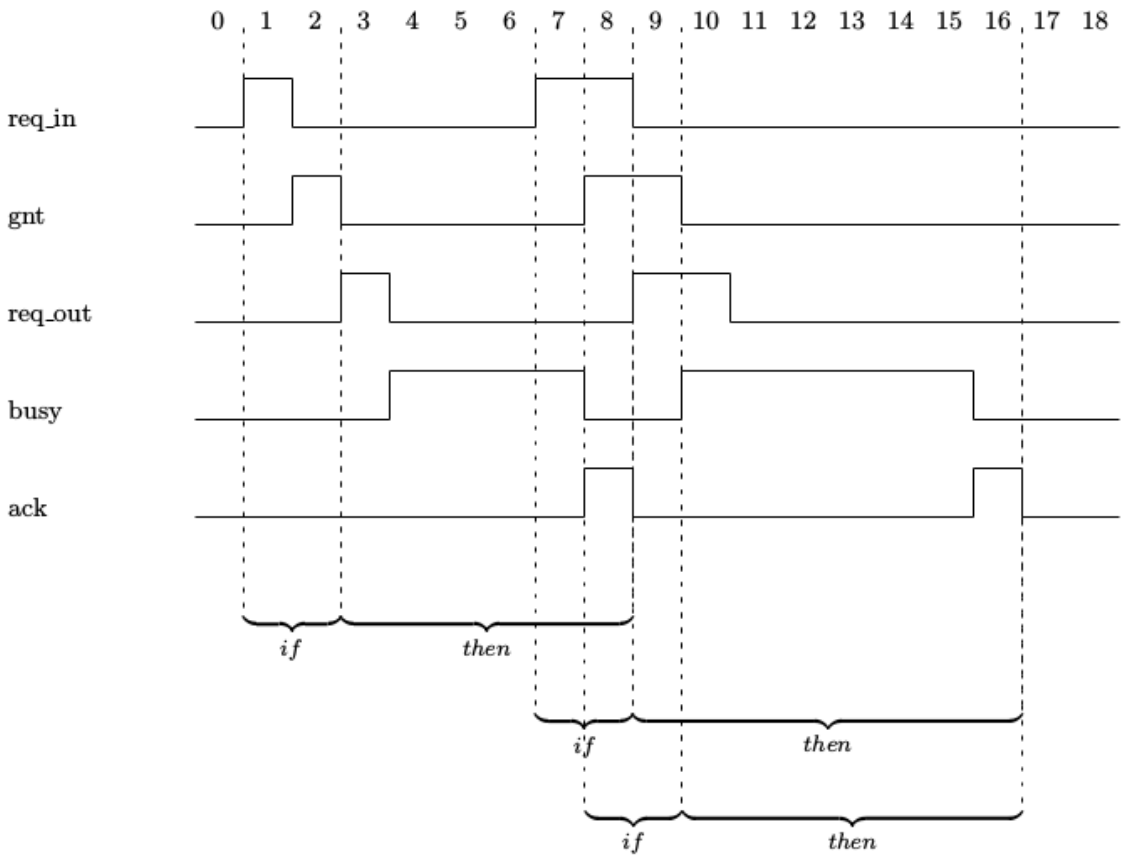
- Оператори суфиксне импликације су |-> и |=>
- Ови оператори спајају две SERE секвенце
- Оператор |-> је преклапајући
- Оператор |=> је непреклапајући

Примери суфиксне импликације #1

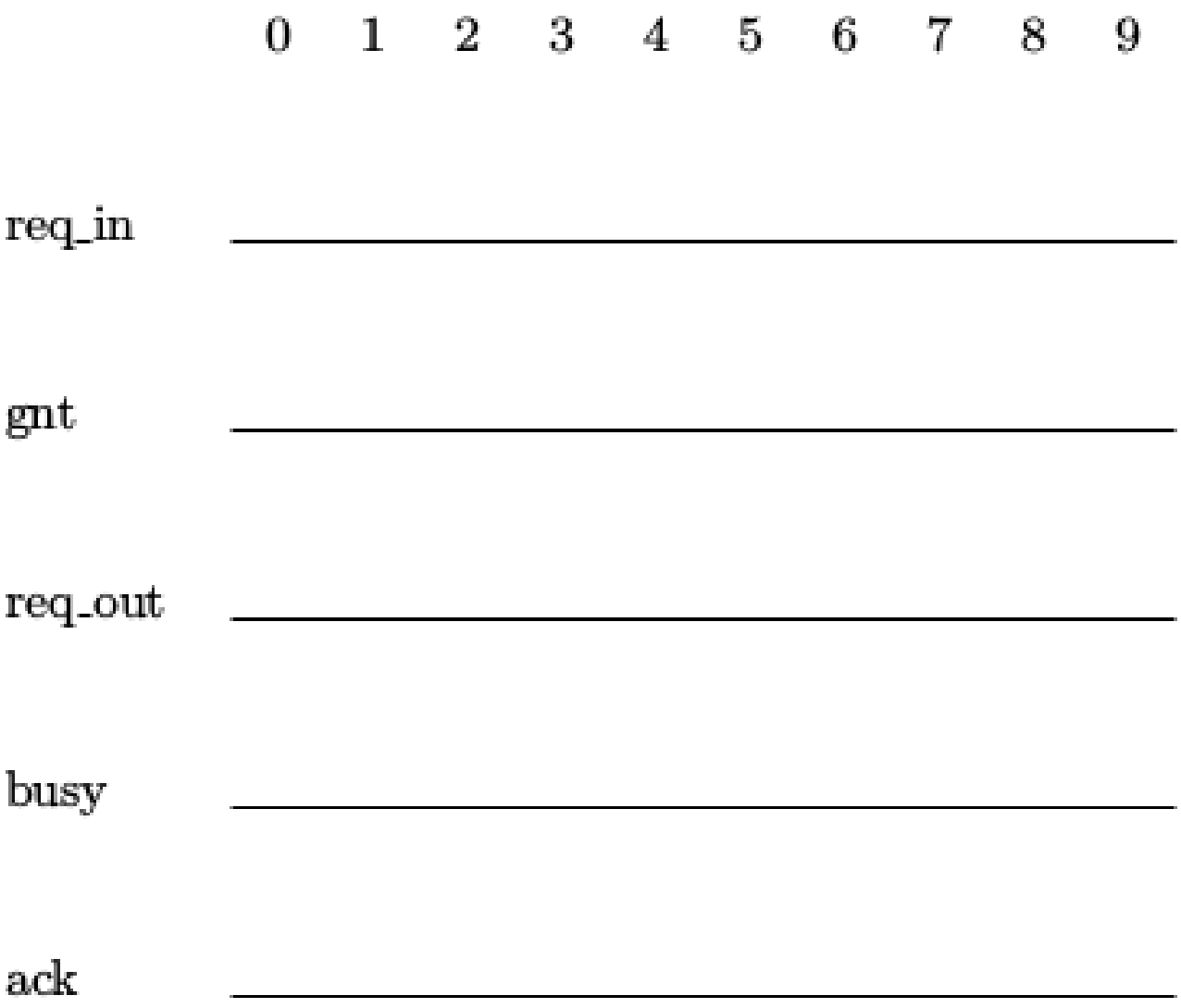
```
always {req in;gnt} |-> {(req out && !ack) ;
  (busy && !ack)[*] ; ack};

always {req in;gnt} |=> {(req out && !ack) ;
  (busy && !ack)[*] ; ack};
```



Примери суфиксне импликације #2

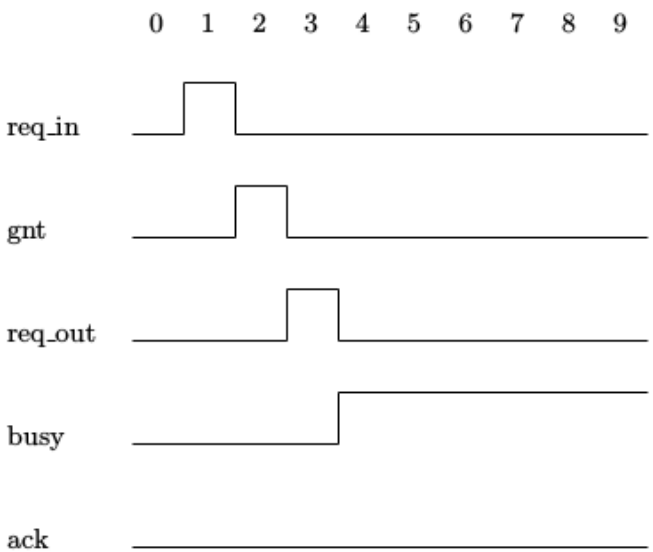


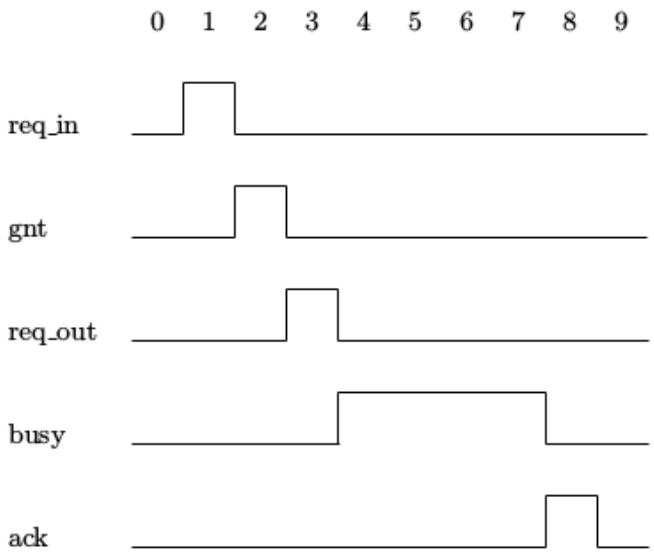
Примери суфиксне импликације #3



Слаби, јаки SERE

```
always {req in;gnt} |=>
    {(req out && !ack) ; (busy && !ack)[*] ; ack};
always {req in;gnt} |=>
    {(req out && !ack) ; (busy && !ack)[*] ; ack}!;
```





1. T T
2. F T

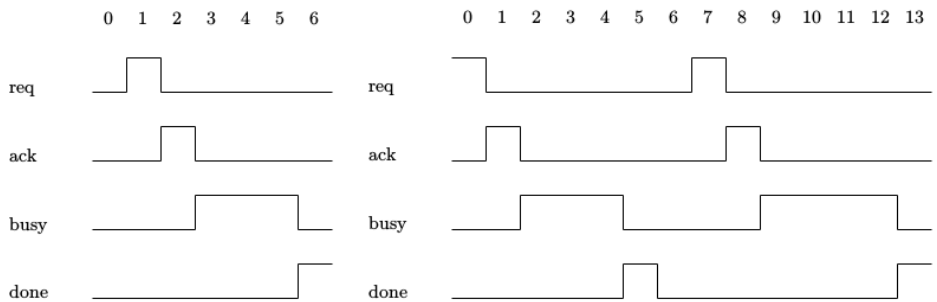
SERE понављања

- Постоје неколико фамилија оператора помоћу којих могу да се искажу разне врсте понављања у SERE
- Због ових оператора SERE има Regular Expressions у свом имену
- Те фамилије оператора су:

1. `[*]`
2. `[+]`
3. `[=]`
4. `[->]`

SERE понављања * фамилија оператора

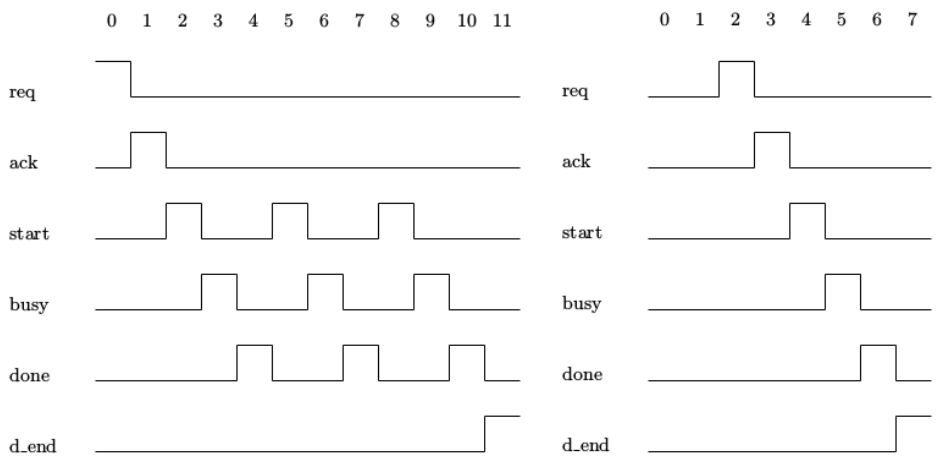
```
// * = None or more consecutive repetition
assert always {req} |=> {ack;busy;busy;busy;done};   // 1.
assert always {req} |=> {ack ; busy[*3] ; done};     // 2.
assert always {req} |=> {ack ; busy[*3:5] ; done};   // 3.
```



1. T F
2. T F
3. T T

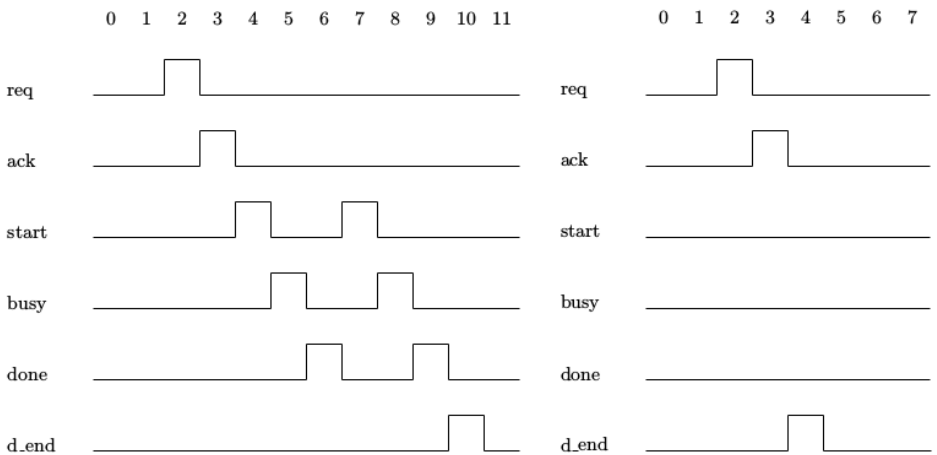
SERE понављања * фамилија оператора и + #1

```
// Whole SERE is repeating!
// + at least one or more consecutive repetitions
always {req;ack} |=> {{start;busy;done}[*3] ; d_end}
always {req;ack} |=> {{start;busy;done}[*] ; d_end}
always {req;ack} |=> {{start;busy;done}[+] ; d_end}
```



1. T F
2. T T
3. T T

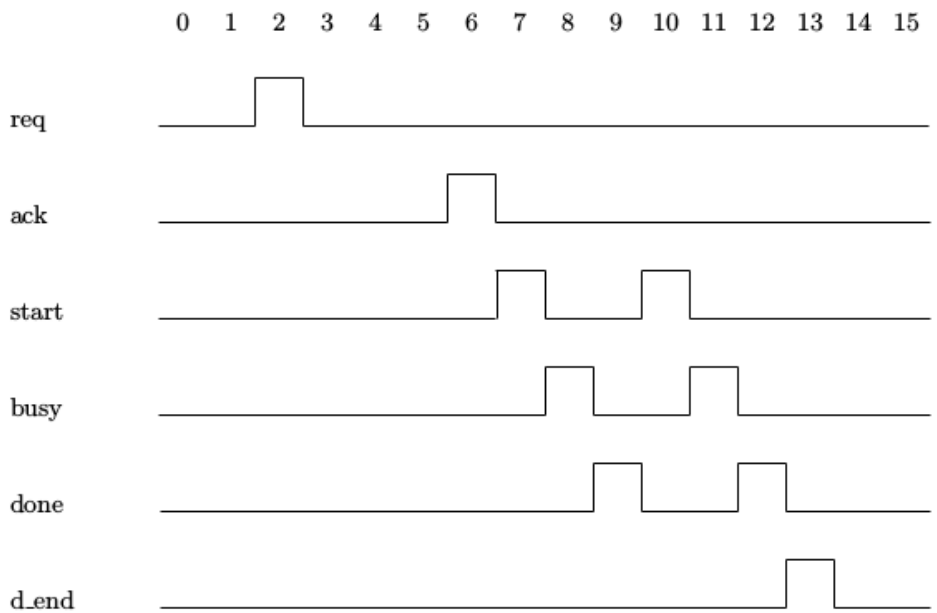
SERE понављања * фамилија оператора и + #2



1. F F
2. T T
3. T F

Усамљени SERE * оператора

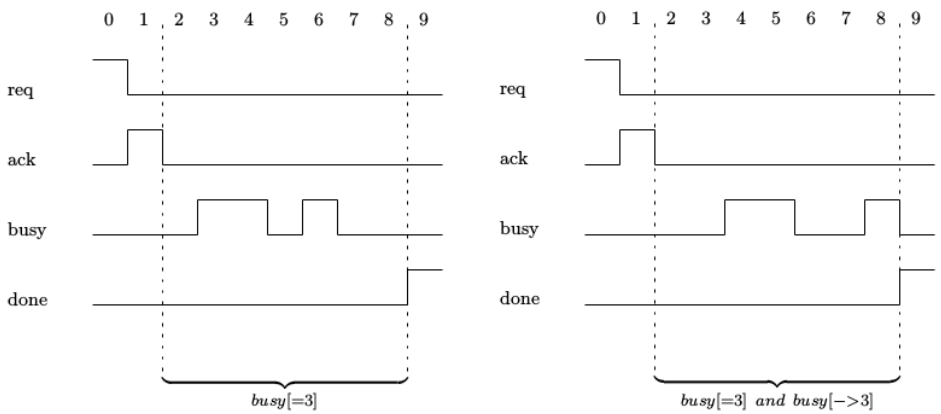
```
// Skipping 3 cycles
always {req;[*3];ack} |=> {{start;busy;done}[*] ; d_end}
```



1. T

SERE понављања = и -> фамилија оператора

```
// Nonconsecutive repetitions
assert always {req} |=> {ack ; busy[=3] ; done};
assert always {req} |=> {ack ; busy[->3] ; done};
```



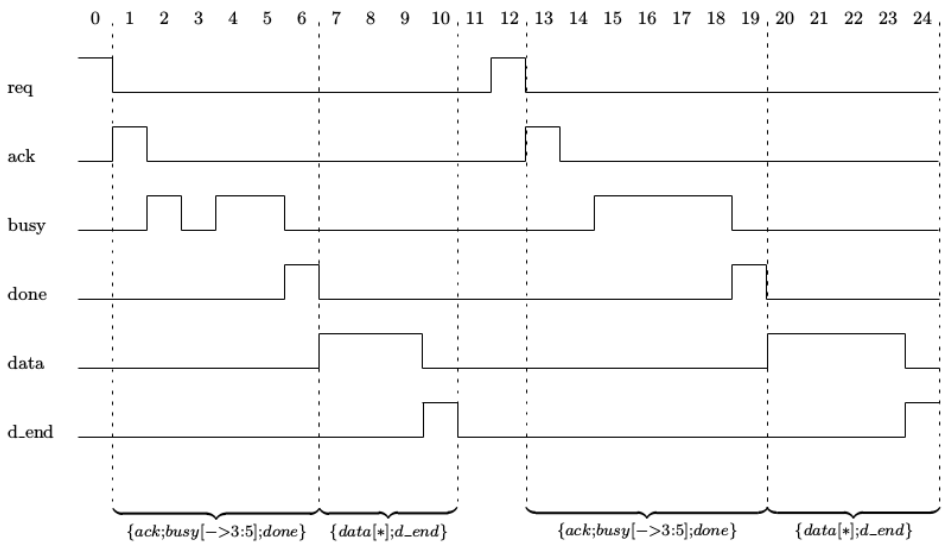
1. T T
2. F T

Спајање и фузија

- До сада смо често користили оператор који спаја два SERE израза. Ово се зове спајање.
- Уз овај оператор спајања коже се користити и оператор фузије да би се од једноставнијих SERE секвенци добиле комплексније
- Оператор спајања не преклапа секвенце док их оператор фузије преклапа.

Спајање

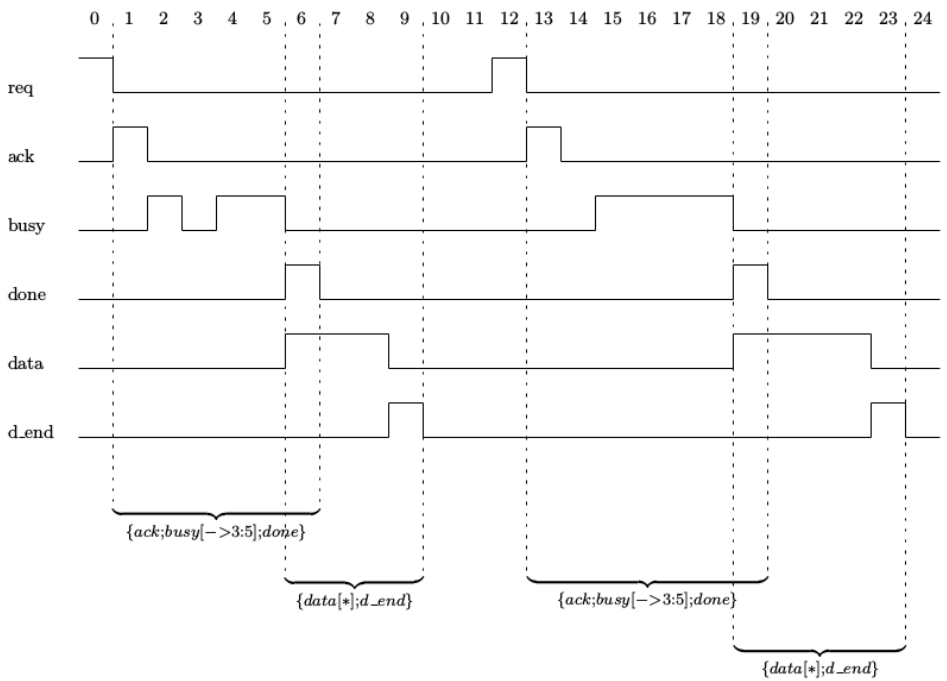
```
assert always {req} |=>
{{ack ; busy[->3:5] ; done} ; {data[*] ; d end}};
```



1. T

Фузија

```
assert always {req} |=>
{{ack ; busy[->3:5] ; done} : {data[*] ; d_end}};
```



1. T

Комплексни SERE изрази

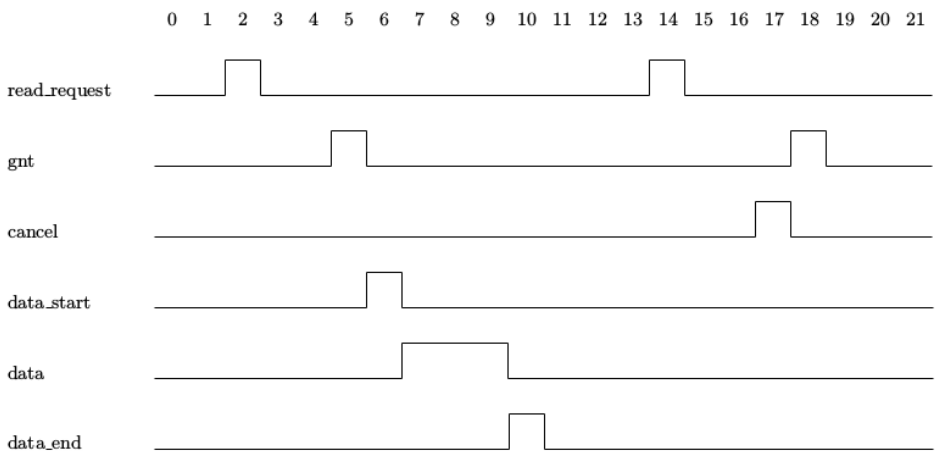
- Више SERE израза се може комбиновати и са логичким операторима
- Логичко SERE или
- Логичко SERE и са истом дужином
- Логичко SERE и са различитом дужином
- Особина коју треба исказати: На крају сваког циклуса читања је потребно подићи сигнал read_complete. Постоји кратко читање које се започиње подизањем short_rd сигнала. Након овог сигнала читање се обавља са 8 неузастопних подизања data сигнала. Дуго читање се започиње са подизањем long_rd сигнала, након чега следи процес читања који се састоји од 32 неузастопна подизања data сигнала.

Комплексни SERE или исказ

```
always {short_rd ; data[->8]} |-> {read_complete};

always {long_rd ; data[->32]} |-> {read_complete};

always {{short_rd ; data[->8]} |
  {long_rd ; data[->32]}} |-> {read_complete};
```

Комплексни SERE и исказ #1

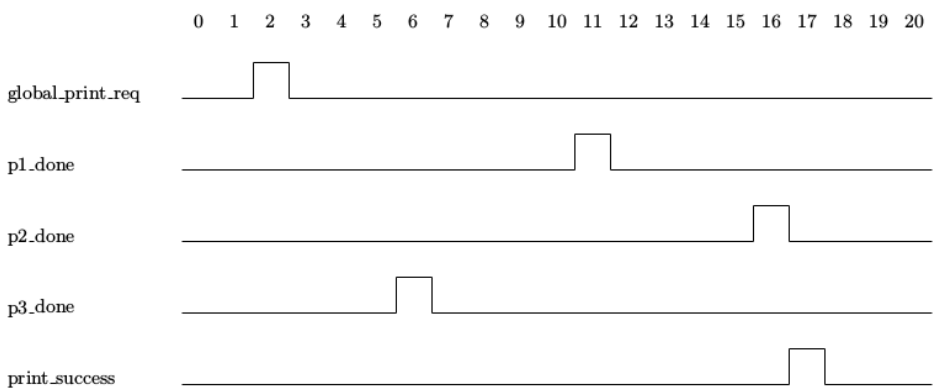
```
assert always
    {{read req ; [*0:4] ; gnt} && {cancel[=0]}} |=>
    {data start ; data[*] ; data end};

assert always
    {{read req ; [*0:4] ; gnt} & {cancel[=0]}} |=>
    {data start ; data[*] ; data end};
```



1. T
2. F

Комплексни SERE и исказ #2

```
assert always {global print req} |=>
    {{p1 done[->] & p2 done[->] & p3 done[->]} ;
      print success};

assert always {global print req} |=>
    {{p1 done[->] && p2 done[->] && p3 done[->]} ;
    print success};
```



1. T
2. F

Модуларност и скалабилност у PSL-у

- property
- sequence
- forall
- Параметризоване особине

Property - Именоване особине

- Формална дефиниција именоване особине:

```
property getgrant (boolean req; numeric n; boolean gnt, clk
    always (req -> next[n] (gnt));
```

- Инстацирање особине:

```
assert getgrant(req1, 3, gnt1);
assert getgrant(req2, 5, gnt2);
```

- Упити еквивалентни претходним:

```
assert always (req1 -> next[3] (gnt1));
assert always (req2 -> next[5] (gnt2));
```

Sequence - Именоване SERE

- Формална дефиниција именоване SERE:

```
sequence reqseq(boolean req,ack,cancel,gnt) =
    {req ; ack & !cancel ; gnt};
sequence dataseq(boolean start,data,done; numeric n) =
    {start ; data[*n] ; done};
```

- Инстацирање SERE:

```
assert reqseq(req_out, ack_in, cancel_out, gnt_in) |=>
    dataseq(start_data, data, end_data, 8);
```

- Упити еквивалентни претходним:

```
assert {req_out ; ack_in & !cancel_out ; gnt_in} |=>
    {start_data ; data[*8] ; end_data};
```

Property и Sequence као формални параметри

- Формална дефиниција особине која као параметар прима другу особину и именовану SERE:

```
sequence ahead_of(sequence s; boolean fin) =
    {{s && fin[=0]} ; fin};
property seqimplies(sequence s; property p) =
    always {s} |=> (p);
```

- Инстанцирање комплексне особине:

```
assert seqimplies(ahead_of({a;b;c}[*],d), e before f);
```

- Упити еквивалентан претходној особини:

```
assert always {{{a;b;c}[*] && d[=0]} ; d} |=>
    (e before f);
```

Формални параметри за именоване особине и SERE

- boolean
- bit
- bitvector
- numeric
- string
- sequence
- property
- Типови података VHDL или Verilog језика, hdltype

```
sequence type_example (hdltype MYTYPE a) =
    {a==‘NULL ; (a!=‘NULL)[*] ; a==‘NULL};
```

forall - Фамилија особина

- Оператор forall омогућава да се напише фамилија сличних особина
- Особине које се добију коришћењем forall оператора зову се и реплициране особине (replicated property)
- Пример forall фамилије особина:

```
assert forall i in {0:7}:
    always ((read_request && tag[2:0]==i) ->
    next[4](tag[2:0]==i));
```

- Једна од добијених реплицираних особина:

```
always ((read request && tag[2:0]==3) ->
    next[4](tag[2:0]==3))
```

forall - меморијска интерпретација

- Оператор forall може се користити за памћење неке вредности у једном тренутку, која се затим користи у наредном.
- Могуће је угњездити један forall оператор унутар другог
- Пример за обе наведен ставке:

```
assert forall i in {0:7}:
    forall j[127:0] in boolean:
        always ((reqin && tag[2:0]==i &&
            dat[127:0]==j[127:0]) ->
        next_event(reqout && tag[2:0]==i)
            (dat[127:0]==j[127:0]));
```

- Треба бити свестан да forall оператор прави особине које могу бити веома комплексне за доказивање алатима за формалну верификацију

Параметризоване особине

- Скалабилност се постиже параметризацијом оператора “и” и “или”
- Пример параметризованог оператора “и”:

```
for i in {0:3}: && (P(i))
```

- Када се размота овако параметризован оператор добија се:

```
P(0) && P(1) && P(2) && P(3)
```

- Пример параметризоване особине:

```
assert always
    ((for i in {0:31}: || (data[i]!=expected[i])) ->
    data corrupted);
```

Параметризоване SERE

- Параметризација се постиже на сличан начин као и код параметризованих особина
- Пример параметризованог оператора “и”:

```
for i in {0:3}: && S(i)
```

- Када се размота овако параметризована SERE добија се:

```
{S(0) && S(1) && S(2) && S(3)}
```

- Пример параметризоване SERE:

```
assert always {global_print req} |->
    {{for i in {0:7}: & {prt_done[i]}[->]} ; prt_success};
```

- Како изгледа последња особина када се размота?