heig-vd

HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD

www.heig-vd.ch

# La vérification formelle
## Quelques cas choisis

Yann Thoma, 1er novembre 2024, yann.thoma@heig-vd.ch

Hes·so
Haute École Spécialisée
de Suisse occidentale

# But de la vérification des systèmes numériques

Répondre à 2 questions

# But de la vérification des systèmes numériques

Répondre à 2 questions

- Est-ce que ça marche ?

# But de la vérification des systèmes numériques

Répondre à 2 questions

- Est-ce que ça marche ?
- Est-ce qu'on est sûr ?

# But de la vérification des systèmes numériques

Répondre à 2 questions

- Est-ce que ça marche ?
- Est-ce qu'on est sûr ?
- Vraiment ?

# But de la vérification des systèmes numériques

Répondre à 2 questions

- Est-ce que ça marche ?
- Est-ce qu'on est sûr ?
- Vraiment ?
- Non, mais vraiment vraiment ?

# But de la vérification des systèmes numériques

Répondre à 2 questions

- Est-ce que ça marche ?
- Est-ce qu'on est sûr ?
- Vraiment ?
- Non, mais vraiment vraiment ?
- Sans plaisanter ?

# Vérification des systèmes numériques

- La vérification doit permettre d'affirmer que la réalisation d'un système correspond à sa spécification

# Vérification des systèmes numériques

- La vérification doit permettre d'affirmer que la réalisation d'un système correspond à sa spécification

$\Rightarrow$ PREUVE du bon fonctionnement

# Vérification des systèmes numériques

- Moyens

# Vérification des systèmes numériques

- Moyens
  - Vérification fonctionnelle
    - Simulation
    - Emulation

# Vérification des systèmes numériques

- Moyens
  - Vérification fonctionnelle
    - Simulation
    - Emulation
  - Vérification formelle

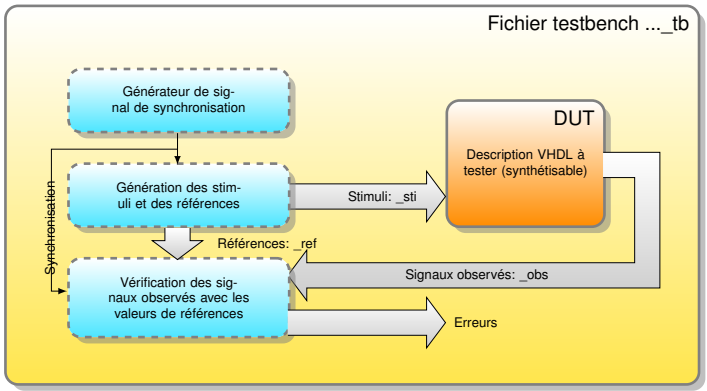
# Vérification des systèmes numériques

- Moyens
  - Vérification fonctionnelle
    - Simulation
    - Emulation
  - Vérification formelle
  - Tests réels

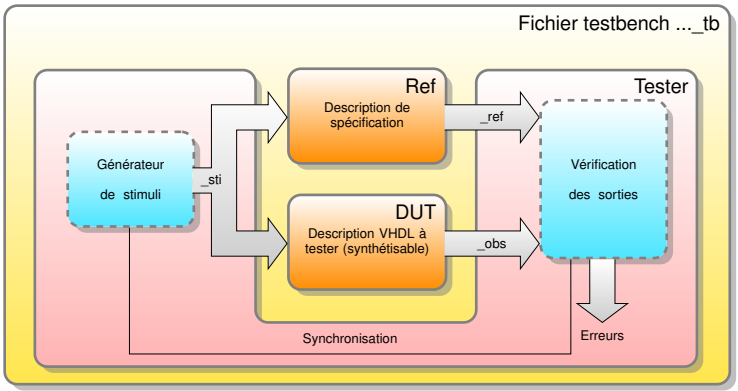
# Vérification fonctionnelle

- Simulation/émulation du système
- Mise en place de bancs de tests
- Simulation
  - Dirigée
    - Difficulté de penser à tous les cas intéressants/limites
  - Aléatoire contrainte + couverture
    - Pas forcément évident à réaliser pour garantir une bonne couverture des cas
  - Mixte
    - Meilleur compromis, mais chronophage

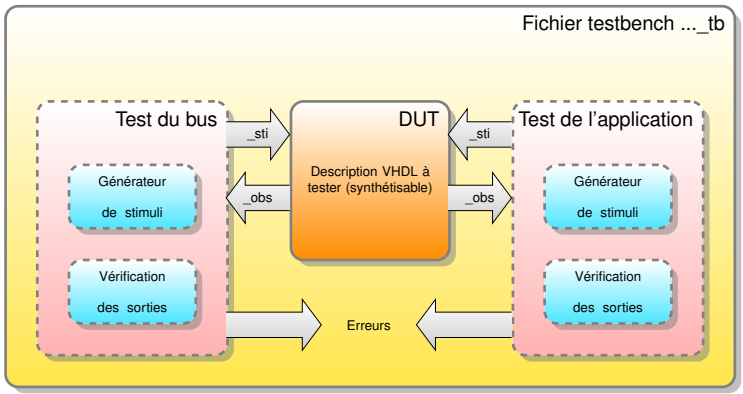
# Banc de tests standard



# Banc de tests avec référence



# Banc de tests pour multi-interface



heig-vd

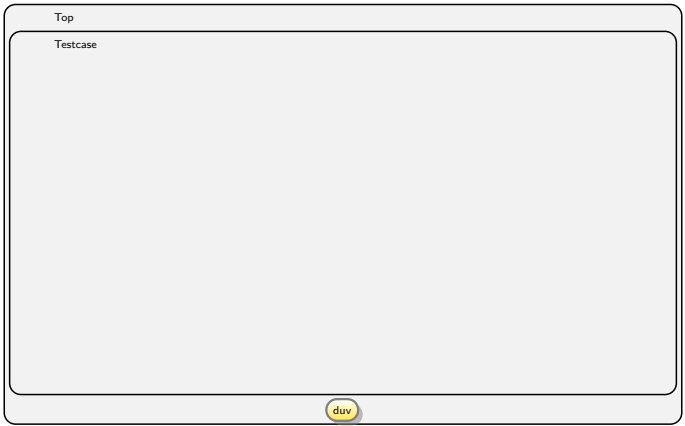
# Banc de test TLM / UVM

heig-vd

# Banc de test TLM / UVM

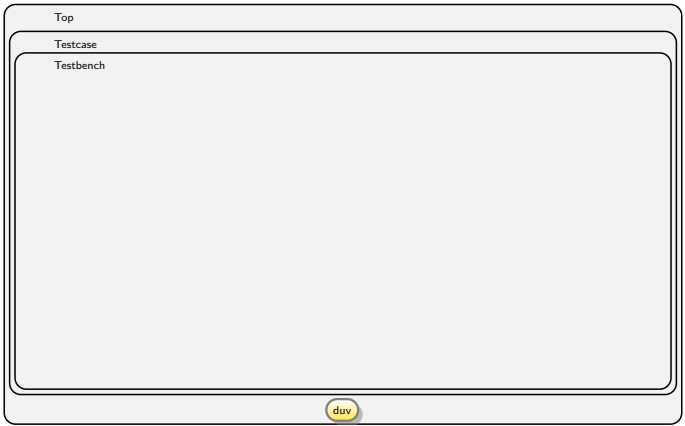
heig-vd

# Banc de test TLM / UVM



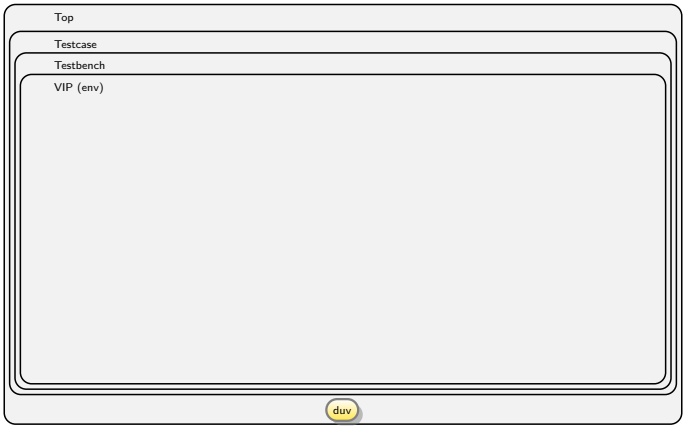
heig-vd

# Banc de test TLM / UVM



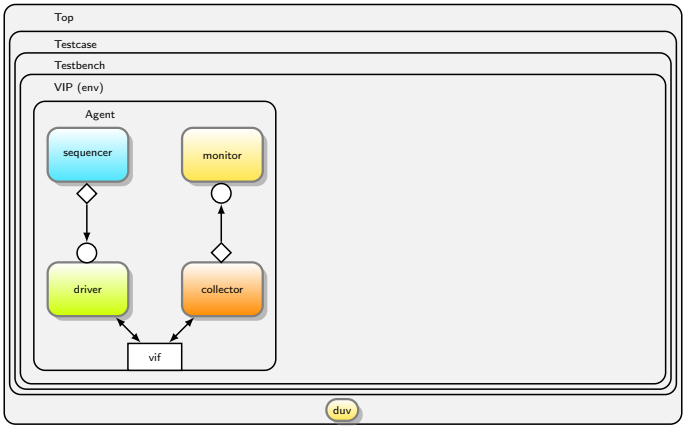
heig-vd

# Banc de test TLM / UVM



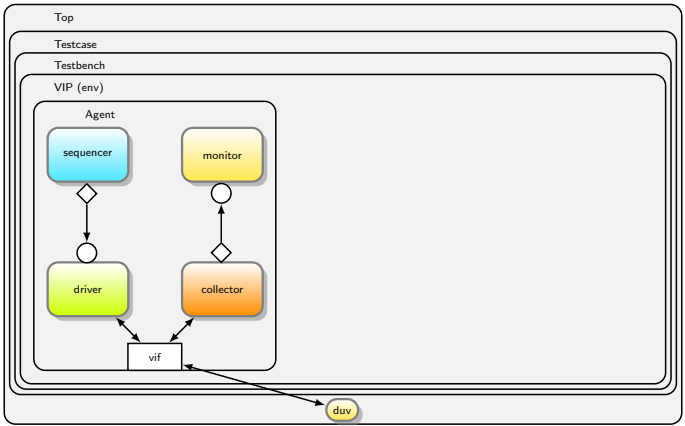
heig-vd

# Banc de test TLM / UVM

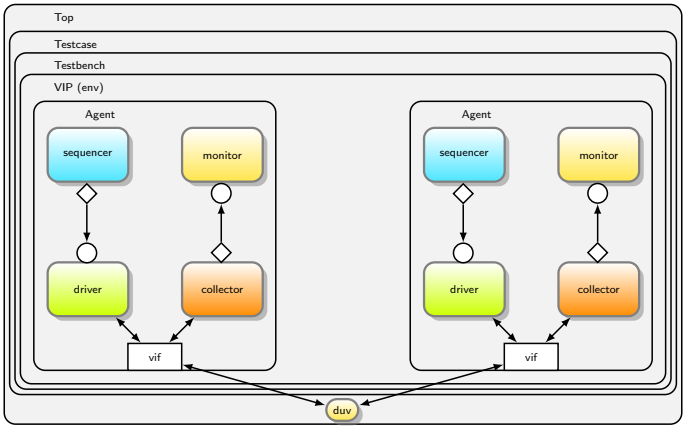


heig-vd

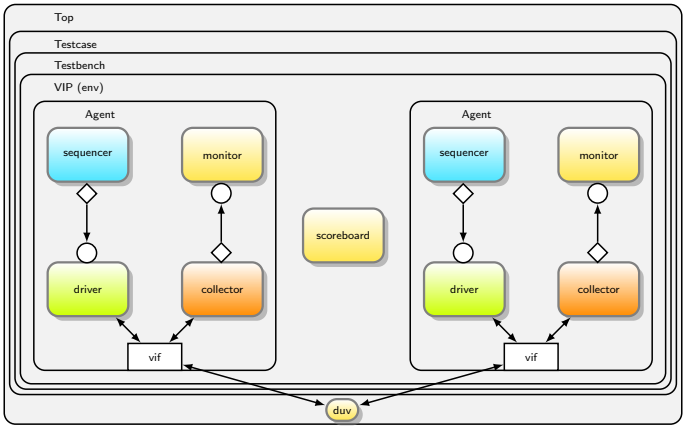
# Banc de test TLM / UVM



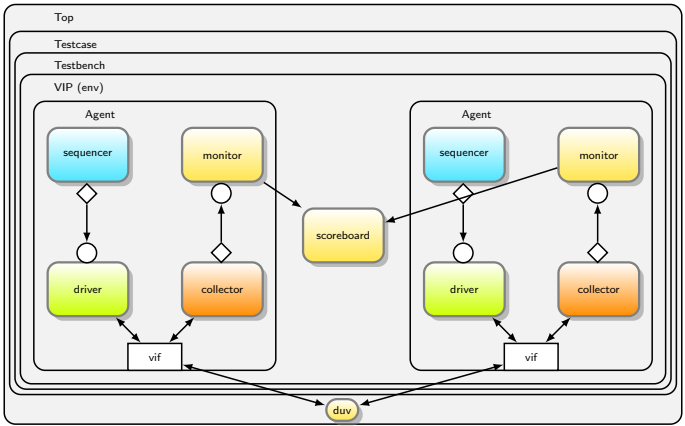
# Banc de test TLM / UVM



# Banc de test TLM / UVM

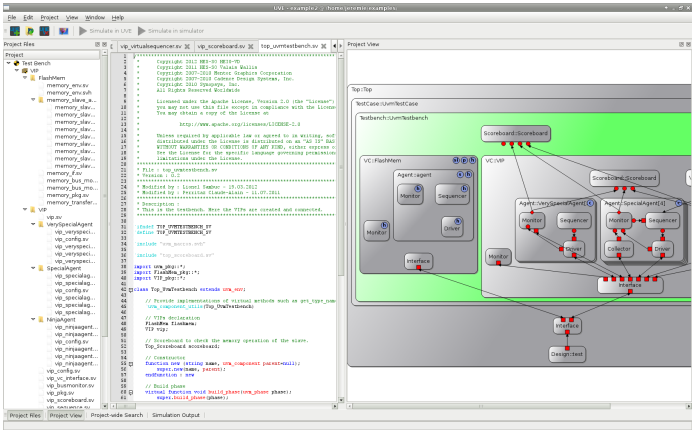


# Banc de test TLM / UVM



# Banc de test UVM : projet UVE

cf. http ://www.systemverilog.ch/



# Vérification formelle

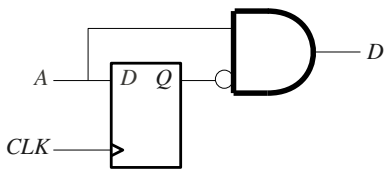
- Confrontation entre le système et des propriétés attendues
- Preuve de l'adéquation entre le système et les spécifications

# Vérification formelle

- Confrontation entre le système et des propriétés attendues
- Preuve de l'adéquation entre le système et les spécifications
- Avantages
  - Preuve formelle, donc irréfutable si les propriétés sont bien écrites
  - Permet de valider une partie du système qui n'a pas besoin d'être vérifiée ensuite par simulation fonctionnelle
- Désavantages
  - Pas forcément adapté à tous les systèmes
- Nécessite potentiellement d'être complémenté par de la vérification fonctionnelle

# Vérification formelle : Exemple simple

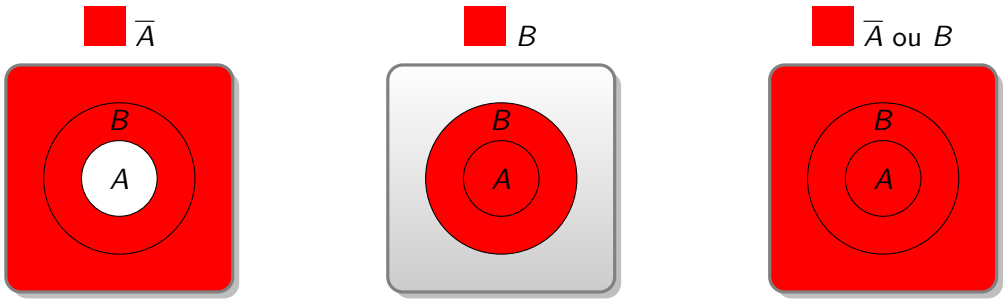
## Détecteur de flanc




```
entity Detector is
port(
    Clk_i : in  std_logic;
    A     : in  std_logic;
    D     : out std_logic
    );
end Detector;
architecture behave of Detector is
    signal reg_s: std_logic;
begin
    process(clk_i)
        begin
        if rising_edge(clk_i) then
            reg_s <= A;
        end if;
    end process;
    D <= A and not(reg_s);
end behave;
```

# Implication : $A \Rightarrow B \equiv \overline{A}$ ou $B$

Imaginez : $A \equiv$ est un corbeau, et $B \equiv$ est noir



# Vérification formelle : Exemple simple

Détecteur de flanc

## Propriété à vérifier

Si $A = 1$ alors au cycle d'horloge suivant : $D = 0$

# Vérification formelle : Exemple simple

Détecteur de flanc

## Propriété à vérifier

Si $A = 1$ alors au cycle d'horloge suivant : $D = 0$
$\Leftrightarrow (A_{t-1} = 1) \Rightarrow D_t = 0$

# Vérification formelle : Exemple simple

Détecteur de flanc

## Propriété à vérifier

Si $A = 1$ alors au cycle d'horloge suivant : $D = 0$
$\Leftrightarrow (A_{t-1} = 1) \Rightarrow D_t = 0$
$\Leftrightarrow A_{t-1} \Rightarrow \overline{D_t}$

# Vérification formelle : Exemple simple

Détecteur de flanc

## Propriété à vérifier

Si $A = 1$ alors au cycle d'horloge suivant : $D = 0$

$\Leftrightarrow (A_{t-1} = 1) \Rightarrow D_t = 0$

$\Leftrightarrow A_{t-1} \Rightarrow \overline{D_t}$

$\Leftrightarrow \overline{A_{t-1}} + \overline{D_t} = 1$ (car $A \Rightarrow B$ est identique à $\overline{A} + B$ )

# Vérification formelle : Exemple simple

Détecteur de flanc

## Propriété à vérifier

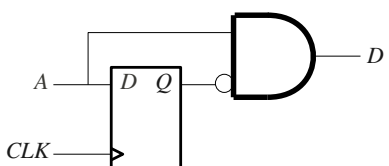
Si $A = 1$ alors au cycle d'horloge suivant : $D = 0$

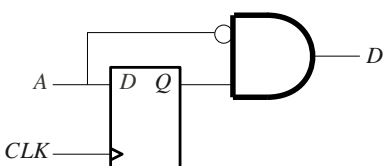
$\Leftrightarrow (A_{t-1} = 1) \Rightarrow D_t = 0$

$\Leftrightarrow A_{t-1} \Rightarrow \overline{D_t}$

$\Leftrightarrow \overline{A_{t-1}} + \overline{D_t} = 1$

(car $A \Rightarrow B$ est identique à $\overline{A} + B$ )



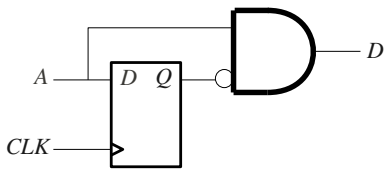




- Quel design est le bon ?

# Vérification formelle : Exemple simple

## Détecteur de flanc

- Propriété à vérifier : $\overline{A_{t-1}} + \overline{D_t} = 1$

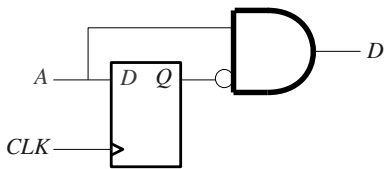



  - $D_t = A_t \cdot \overline{A_{t-1}}$

# Vérification formelle : Exemple simple

## Détecteur de flanc

- Propriété à vérifier : $\overline{A_{t-1}} + \overline{D_t} = 1$



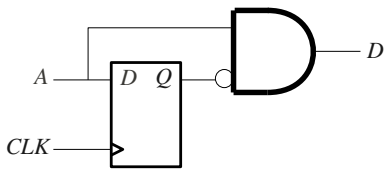

  - $D_t = A_t \cdot \overline{A_{t-1}}$
  - Donc

$\overline{A_{t-1}} + \overline{D_t} \quad =$

# Vérification formelle : Exemple simple

## Détecteur de flanc

- Propriété à vérifier : $\overline{A_{t-1}} + \overline{D_t} = 1$



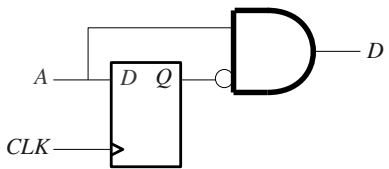

  - $D_t = A_t \cdot \overline{A_{t-1}}$
  - Donc

$$\overline{A_{t-1}} + \overline{D_t} \quad = \quad \overline{A_{t-1}} + \overline{A_t \cdot \overline{A_{t-1}}}$$

# Vérification formelle : Exemple simple

## Détecteur de flanc

- Propriété à vérifier : $\overline{A_{t-1}} + \overline{D_t} = 1$



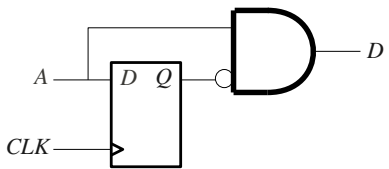

  - $D_t = A_t \cdot \overline{A_{t-1}}$
  - Donc

$$\begin{aligned} \overline{A_{t-1}} + \overline{D_t} &= \overline{A_{t-1}} + \overline{A_t \cdot \overline{A_{t-1}}} \\ &= \overline{A_{t-1}} + \overline{A_t} + A_{t-1} \end{aligned}$$

# Vérification formelle : Exemple simple

## Détecteur de flanc

- Propriété à vérifier : $\overline{A_{t-1}} + \overline{D_t} = 1$



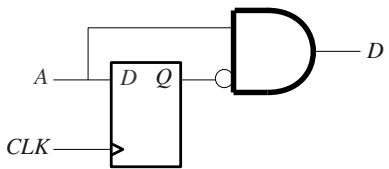

  - $D_t = A_t \cdot \overline{A_{t-1}}$
  - Donc

$$\begin{aligned} \overline{A_{t-1}} + \overline{D_t} &= \overline{A_{t-1}} + \overline{A_t \cdot \overline{A_{t-1}}} \\ &= \overline{A_{t-1}} + \overline{A_t} + A_{t-1} \\ &= 1 \end{aligned}$$

# Vérification formelle : Exemple simple

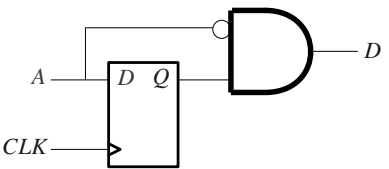
## Détecteur de flanc

- Propriété à vérifier : $\overline{A_{t-1}} + \overline{D_t} = 1$




- $D_t = A_t \cdot \overline{A_{t-1}}$
- Donc

$$\begin{aligned} \overline{A_{t-1}} + \overline{D_t} &= \overline{A_{t-1}} + \overline{A_t \cdot \overline{A_{t-1}}} \\ &= \overline{A_{t-1}} + \overline{A_t} + A_{t-1} \\ &= 1 \end{aligned}$$

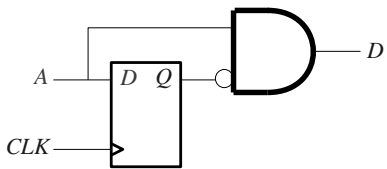



- $D_t = \overline{A_t} \cdot A_{t-1}$

# Vérification formelle : Exemple simple

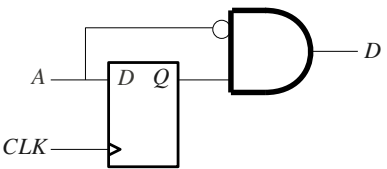
## Détecteur de flanc

- Propriété à vérifier : $\overline{A_{t-1}} + \overline{D_t} = 1$




- $D_t = A_t \cdot \overline{A_{t-1}}$
- Donc

$$\begin{aligned} \overline{A_{t-1}} + \overline{D_t} &= \overline{A_{t-1}} + \overline{A_t \cdot \overline{A_{t-1}}} \\ &= \overline{A_{t-1}} + \overline{A_t} + A_{t-1} \\ &= 1 \end{aligned}$$



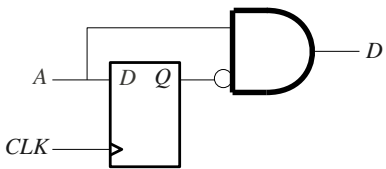

- $D_t = \overline{A_t} \cdot A_{t-1}$
- Donc

$$\overline{A_{t-1}} + \overline{D_t} =$$

# Vérification formelle : Exemple simple

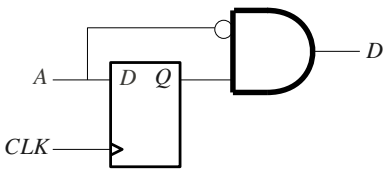
## Détecteur de flanc

- Propriété à vérifier : $\overline{A_{t-1}} + \overline{D_t} = 1$




- $D_t = A_t \cdot \overline{A_{t-1}}$
- Donc

$$\begin{aligned}\overline{A_{t-1}} + \overline{D_t} &= \overline{A_{t-1}} + \overline{A_t \cdot \overline{A_{t-1}}} \\ &= \overline{A_{t-1}} + \overline{A_t} + A_{t-1} \\ &= 1\end{aligned}$$



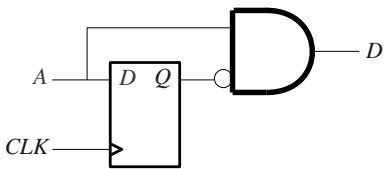

- $D_t = \overline{A_t} \cdot A_{t-1}$
- Donc

$$\overline{A_{t-1}} + \overline{D_t} = \overline{A_{t-1}} + \overline{\overline{A_t} \cdot A_{t-1}}$$

# Vérification formelle : Exemple simple

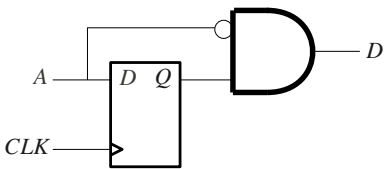
## Détecteur de flanc

- Propriété à vérifier : $\overline{A_{t-1}} + \overline{D_t} = 1$




- $D_t = A_t \cdot \overline{A_{t-1}}$
- Donc

$$\begin{aligned} \overline{A_{t-1}} + \overline{D_t} &= \overline{A_{t-1}} + \overline{A_t \cdot \overline{A_{t-1}}} \\ &= \overline{A_{t-1}} + \overline{A_t} + A_{t-1} \\ &= 1 \end{aligned}$$



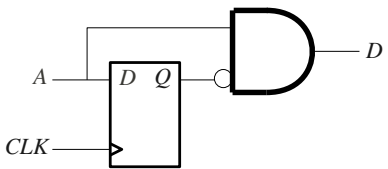

- $D_t = \overline{A_t} \cdot A_{t-1}$
- Donc

$$\begin{aligned} \overline{A_{t-1}} + \overline{D_t} &= \overline{A_{t-1}} + \overline{\overline{A_t} \cdot A_{t-1}} \\ &= \overline{A_{t-1}} + A_t + \overline{A_{t-1}} \end{aligned}$$

# Vérification formelle : Exemple simple

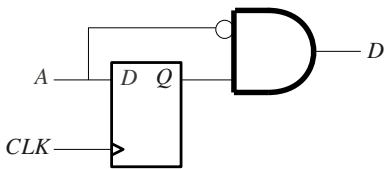
## Détecteur de flanc

- Propriété à vérifier : $\overline{A_{t-1}} + \overline{D_t} = 1$




- $D_t = A_t \cdot \overline{A_{t-1}}$
- Donc

$$\begin{aligned} \overline{A_{t-1}} + \overline{D_t} &= \overline{A_{t-1}} + \overline{A_t \cdot \overline{A_{t-1}}} \\ &= \overline{A_{t-1}} + \overline{A_t} + A_{t-1} \\ &= 1 \end{aligned}$$




- $D_t = \overline{A_t} \cdot A_{t-1}$
- Donc

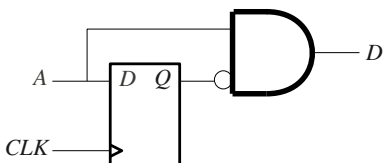
$$\begin{aligned} \overline{A_{t-1}} + \overline{D_t} &= \overline{A_{t-1}} + \overline{\overline{A_t} \cdot A_{t-1}} \\ &= \overline{A_{t-1}} + A_t + \overline{A_{t-1}} \\ &= \overline{A_{t-1}} + A_t \neq 1 \end{aligned}$$

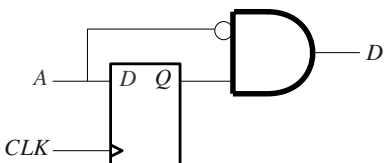
# Vérification formelle : Exemple simple (exercice)

## Détecteur de flanc

### Propriété à vérifier

$\overline{A_{t-1}} \cdot A_t \Rightarrow D_t$



# Vérification formelle : outils logiciels

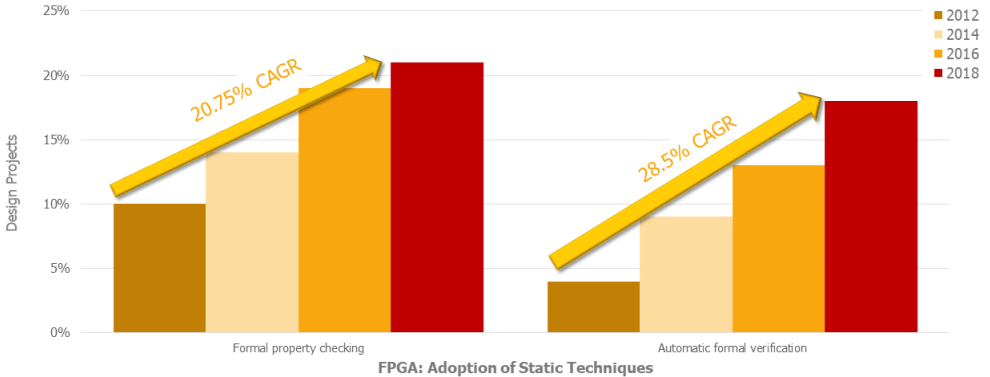
- Heureusement il existe des logiciels capable de réaliser cette preuve
- Notamment QuestaFormal de Mentor Graphics
  - Valide les propriétés
  - Trouve un contre-exemple et donne le chronogramme
  - Très proche des techniques de *model checking*

# Vérification formelle

## Outils

- La vérification formelle nécessite l'écriture de propriétés qu'un système doit avoir
  - Utiles pour :
    - Assertions
    - Hypothèses
    - Couverture
- Langages
  - PSL (Property Specification Language) : *language-agnostic*
  - SVA (SystemVerilog assertions) : Inclues dans le langage
  - OVL (Open Verification Library) : Construit au-dessus de VHDL/SVA

# FPGA: Formal Technology Adoption



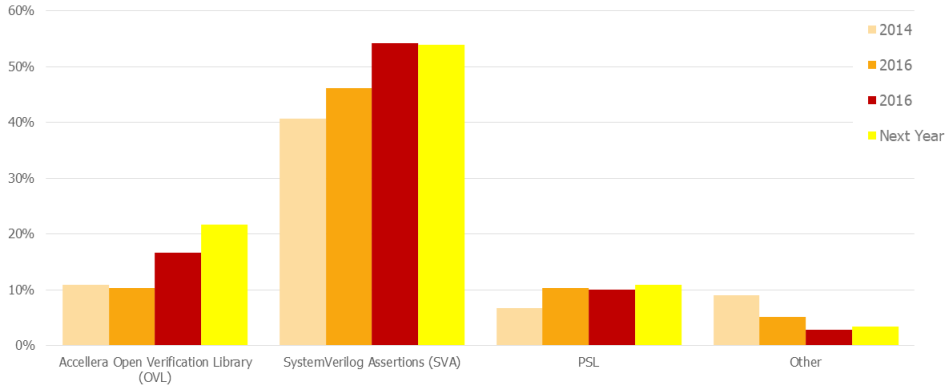

FPGA: Adoption of Static Techniques

*Source: Wilson Research Group and Mentor, A Siemens Business, 2018 Functional Verification Study*



Mentor
A Siemens Business

# FPGA: Assertion Language Adoption Next Twelve Months




FPGA Assertion Language Adoption

* Multiple answers possible

*Source: Wilson Research Group and Mentor, A Siemens Business, 2018 Functional Verification Study*



Mentor
A Siemens Business

# Du Model checking à la vérification des systèmes numériques

- Le *model checking* exploite la logique temporelle (temporal logic)
  - Linear Temporal Logic
  - Computational Tree Logic
  - ... Ou des variations autour du thème

# Linear Temporal Logic

| | | |
|---|---|---|
| $\phi ::=$ | $\top$ | true |
| | $\bot$ | false |
| | $\neg(\phi)$ | negation |
| | $p$ | proposition |
| | $(\phi \wedge \phi)$ | conjunction |
| | $(\phi \vee \phi)$ | disjunction |
| | $X\phi$ | next time $\phi$ |
| | $F\phi$ | eventually $\phi$ (strong operator) |
| | $G\phi$ | always $\phi$ |
| | $\phi U \phi$ | $\phi U \psi : \phi$ until $\psi$ (strong operator) |

# Computational Tree Logic

| | | |
|---|---|---|
| $\phi ::=$ | $\top$ | true |
| | $\bot$ | false |
| | $p$ | a proposition |
| | $\neg(\phi)$ | negation |
| | $(\phi \wedge \phi)$ | conjunction |
| | $(\phi \vee \phi)$ | disjunction |
| | $(\phi \Rightarrow \phi)$ | implication |
| | $(\phi \Leftrightarrow \phi)$ | equivalence |
| | $AX\phi$ | In all subsequent states $\phi$ holds |
| | $EX\phi$ | There exist a subsequent state such that $\phi$ holds |
| | $AF\phi$ | In all futures there is at least one state where $\phi$ holds |
| | $EF\phi$ | There exist at least a future in which $\phi$ holds |
| | $AG\phi$ | $\phi$ holds in every state in the future |
| | $EG\phi$ | There is one path on which $\phi$ holds in every state |
| | $A[\phi U \phi]$ | $\phi$ holds until $\psi$ on every path |
| | $E[\phi U \phi]$ | There exist one path on which $\phi$ holds until $\psi$ |

# PSL/SVA

## LTL ou ?

- PSL
  - LTL-style : proche de LTL (until, next, eventually, ...)
  - CTL-style : proche de CTL (opérateur existentiel), appelé Optional Branching Extension (OBE), mais pas disponible dans QuestaFormal
  - SERE-style : exploite des expressions régulières

# PSL/SVA

## LTL ou ?

- PSL
  - LTL-style : proche de LTL (until, next, eventually, ...)
  - CTL-style : proche de CTL (opérateur existentiel), appelé Optional Branching Extension (OBE), mais pas disponible dans QuestaFormal
  - SERE-style : exploite des expressions régulières
- SVA
  - SERE-style : exploite expressions régulières

# PSL/SVA

## LTL ou ?

- PSL
  - LTL-style : proche de LTL (until, next, eventually, ...)
  - CTL-style : proche de CTL (opérateur existentiel), appelé Optional Branching Extension (OBE), mais pas disponible dans QuestaFormal
  - SERE-style : exploite des expressions régulières
- SVA
  - SERE-style : exploite expressions régulières
- Différences
  - Certaines propriétés ne peuvent pas être exprimées via des expressions régulières
  - Donc PSL est plus expressif que SVA
  - Mais SVA est OK pour la plupart des cas

# Les assertions en 5 minutes

- Les assertions exploitent des propriétés
- Les propriétés exploitent des séquences et des implications
- Les séquences exploitent... le langage

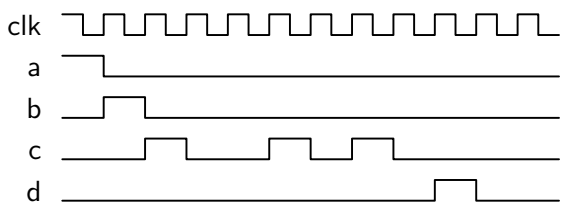
# Séquence

## Exemples

```
// a suivi de b
sequence seq_a;
    a ##1 b;
endsequence

// c suivi de d inactif
sequence seq_b;
    c ##1 !d;
endsequence
```

# Séquence

```
// b suivi de c trois fois (avec c inactif entre) suivi de d
sequence seq_2;
   (b ##1 c[=3] ##1 d);
endsequence
```



# Propriété

- Exploite des séquences et des implications

Exemple

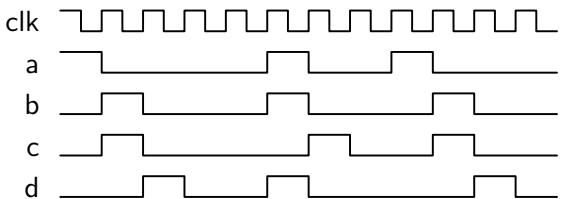
```
property prop;
    seq_a |=> seq_b;
endproperty
```

# Types d'implication

- Implication différée
  - |=>
  - L'évaluation de la séquence de droite commence après la fin de celle de gauche
- Implication directe
  - |->
  - L'évaluation de la séquence de droite commence au moment de la dernière phase de celle de gauche

# Opérateurs d'implication

## Exemple

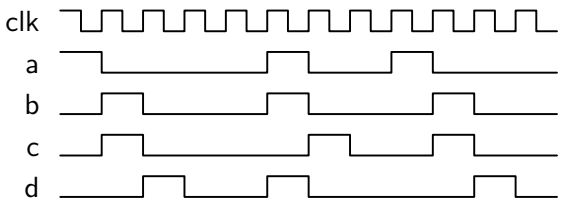



## Assertions

```
property a ##1 b |-> c ##1 d;

property a ##1 b |=> c ##1 d;
```

# Opérateurs d'implication

## Exemple



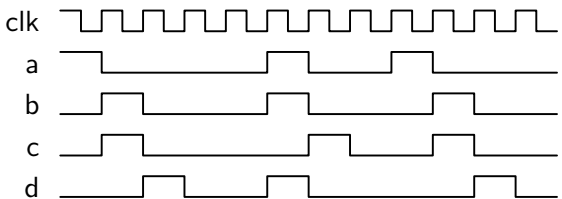

## Assertions

✓
```
property a ##1 b |-> c ##1 d;
```

```
property a ##1 b |=> c ##1 d;
```

# Opérateurs d'implication

## Exemple




## Assertions

✓ `property a ##1 b |-> c ##1 d;`

✗ `property a ##1 b |=> c ##1 d;`

# Opérateurs

- and
- or
- throughout
- intersect
- within

# Propriétés

- Les propriétés peuvent être exploitées par 3 instructions :
  - `assert` permet de vérifier qu'une propriété est vérifiée
  - `assume` permet de spécifier des hypothèses sur l'environnement de simulation. Est notamment utile pour la vérification formelle (permet de limiter le nombre de cas).
  - `cover` permet de monitorer l'observation de propriétés

## Exemples

```
a0 : assert property prop;

a1 : assume property never (a == b);

a2 : cover property (a ##1 b ##1 c) compteur = compteur + 1;
```

# Cas d'utilisation

- Timer
- Shift register
- Stepper motor
- Direction sensor
- Binary Coded Decimal Adder

# Cas pratiques

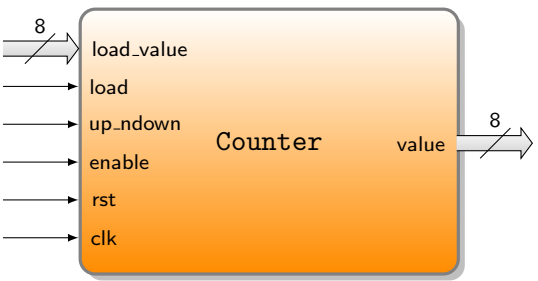
## Exemples

- Compteur
- FIFO simple
- FIFO multiple
- Contrôle/datapath

# Vérification d'un compteur

## DUV

- Compteur 8 bits
- Chargement parallèle
- Incrémentation
- Décrémentation
- Maintien



# Vérification d'un compteur

## Banc de test

1. Tests dirigés
   - Scénarios définis avec test des cas limites
2. Génération aléatoire des entrées
   - Couverture pour vérifier que les cas limites ont été vérifiés
     - Notamment passage des bornes

# Vérification d'un compteur

## Propriétés à vérifier

- `p_load` : Si `load` est actif, au cycle suivant la sortie doit valoir l'entrée courante
- `p_hold` : Si `load` et `enable` sont inactifs, le compteur doit garder sa valeur
- `p_incr` : Si `load` est inactif, `enable` est actif et `up_ndown` est actif, le compteur doit s'incrémenter
- `p_decr` : Si `load` est inactif, `enable` est actif et `up_ndown` est inactif, le compteur doit se décrémenter

# Vérification d'un compteur

## Assertions

# Vérification d'un compteur

## Assertions

```
// load operation
assert_load: assert property (@( posedge clk) disable iff (rst==1)
    (load==1) |=> value == $past(load_value));
```

# Vérification d'un compteur

## Assertions

```
// load operation
assert_load: assert property (@( posedge clk) disable iff (rst==1)
    (load==1) |=> value == $past(load_value));

// maintain operation
assert_maintain: assert property (@( posedge clk) disable iff (rst==1)
    (load==0) & (enable==0) |=> value == $past(value));
```

# Vérification d'un compteur

## Assertions

```
// load operation
assert_load: assert property (@( posedge clk) disable iff (rst==1)
    (load==1) |=> value == $past(load_value));

// maintain operation
assert_maintain: assert property (@( posedge clk) disable iff (rst==1)
    (load==0) & (enable==0) |=> value == $past(value));

// increment operation
assert_increment: assert property (@( posedge clk) disable iff (rst==1)
    (load==0) & (enable==1) & (up_ndown==1) |=> value == ($past(value)+1)%256);
```

# Vérification d'un compteur

## Assertions

```
// load operation
assert_load: assert property (@( posedge clk) disable iff (rst==1)
    (load==1) |=> value == $past(load_value));

// maintain operation
assert_maintain: assert property (@( posedge clk) disable iff (rst==1)
    (load==0) & (enable==0) |=> value == $past(value));

// increment operation
assert_increment: assert property (@( posedge clk) disable iff (rst==1)
    (load==0) & (enable==1) & (up_ndown==1) |=> value == ($past(value)+1)%256);

// decrement operation
assert_decrement: assert property (@( posedge clk) disable iff (rst==1)
    (load==0) & (enable==1) & (up_ndown==0) |=> value == ($past(value)-1)%256);
```

# Vérification d'un compteur

## Assertions

```
`define ASSERT_PROP(p) assert property (@( posedge clk) disable iff (rst==1) p );
```

# Vérification d'un compteur

## Assertions

```
`define ASSERT_PROP(p) assert property (@( posedge clk) disable iff (rst==1) p );

// load operation
assert_load: `ASSERT_PROP( (load==1) |=> value == $past(load_value) )
```

# Vérification d'un compteur

## Assertions

```
`define ASSERT_PROP(p) assert property (@( posedge clk) disable iff (rst==1) p );

// load operation
assert_load: `ASSERT_PROP( (load==1) |=> value == $past(load_value) )

// maintain operation
assert_maintain: `ASSERT_PROP( (load==0) & (enable==0) |=> value == $past(value) )
```

# Vérification d'un compteur

## Assertions

```
`define ASSERT_PROP(p) assert property (@( posedge clk) disable iff (rst==1) p );

// load operation
assert_load: `ASSERT_PROP( (load==1) |=> value == $past(load_value) )

// maintain operation
assert_maintain: `ASSERT_PROP( (load==0) & (enable==0) |=> value == $past(value) )

// increment operation
assert_increment: `ASSERT_PROP( (load==0) & (enable==1) & (up_ndown==1) |=>
                                value == ($past(value)+1)%256 )
```

# Vérification d'un compteur

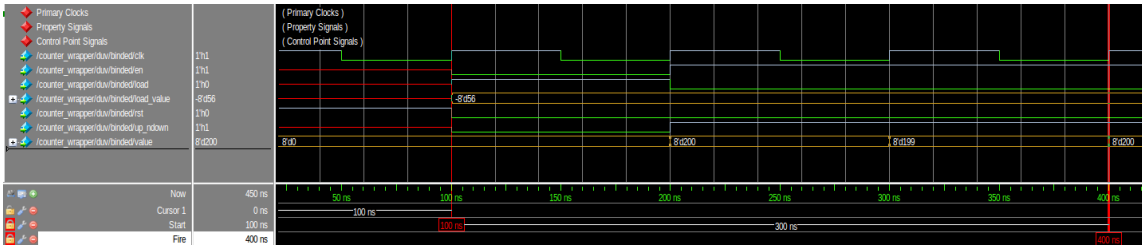
## Assertions

```
`define ASSERT_PROP(p) assert property (@( posedge clk) disable iff (rst==1) p );

// load operation
assert_load: `ASSERT_PROP( (load==1) |=> value == $past(load_value) )

// maintain operation
assert_maintain: `ASSERT_PROP( (load==0) & (enable==0) |=> value == $past(value) )

// increment operation
assert_increment: `ASSERT_PROP( (load==0) & (enable==1) & (up_ndown==1) |=>
                                value == ($past(value)+1)%256 )

// decrement operation
assert_decrement: `ASSERT_PROP( (load==0) & (enable==1) & (up_ndown==0) |=>
                                value == ($past(value)-1)%256 )
```

# Exemple d'une erreur

## Injectée dans le design

- Erreur injectée : Si value == 200, alors décrémente au lieu d'incrémenter

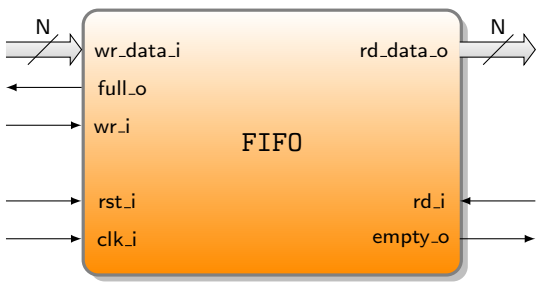



- Scénario : Charge 200, puis incrémente

# Vérification d'un FIFO

## DUV

- FIFO simple
- lecture/écriture synchrone
- Deux signaux pour l'état du FIFO



# Vérification d'un FIFO

## Banc de test

- Simulation de cas
- Tests dirigés
  - Que se passe-t'il quand le FIFO est vide
  - Que se passe-t'il quand le FIFO est plein
- Tests aléatoires
  - Laisser des accès aléatoires
  - Utiliser la couverture pour être sûr d'avoir testé les conditions pertinentes

# Vérification d'un FIFO

## Preuve formelle

- Propriétés à vérifier
  - P1. Si le nombre d'écritures est égal au nombre de lectures, alors `empty_o` doit être actif
  - P2. Si la différence entre le nombre d'écritures et le nombre de lectures est égale à la taille du FIFO, alors `full_o` doit être actif
  - P3. La $n^{ième}$ donnée écrite doit être la $n^{ième}$ donnée à sortir
  - P4. `full_o` et `empty_o` ne doivent jamais être actifs en même temps

# Vérification d'un FIFO

## Hypothèses

```
// The following disable reads when FIFO is empty
assume property ( @(posedge clk_i) (!(rd_i & empty_o)));

// The following disable write when FIFO is full
assume property ( @(posedge clk_i) (!(wr_i & full_o)));
```

# Vérification d'un FIFO

- Assertion *naïve*

```
assert_not_empty_after_write :
    assert property ( @(posedge clk_i)  (
        (wr_i) |=> !empty_o));
```

# Vérification d'un FIFO

## Compteurs utiles

```
int wcnt = 0;
int rcnt = 0;

always @(posedge clk_i or posedge rst_i)
    if (rst_i)
        wcnt = 0;
    else if (wr_i)
        wcnt = (wcnt + 1);

always @(posedge clk_i or posedge rst_i)
    if (rst_i)
        rcnt = 0;
    else if (rd_i)
        rcnt = (rcnt + 1);
```

# Vérification d'un FIFO

```
property p_full;
    @(posedge clk_i)
        ( full_o == (wcnt == rcnt + FIFOSIZE) );
endproperty

property p_empty;
    @(posedge clk_i)
        ( empty_o == (wcnt == rcnt) );
endproperty

property p_data_integrity;
    int cnt;
    logic[DATASIZE-1:0] data;
    @(posedge clk_i)
        (wr_i, cnt=wcnt, data=data_i) |=>
            ((##[0:$] (rd_i & (rcnt==cnt))) |->
             (data_o==data));
endproperty
```

# Vérification d'un FIFO

## Explosion combinatoire

```
assume property ( @(posedge clk_i) (wcnt < 4*FIFOSIZE));
```

# Erreurs découvertes

- Des signaux initialisés à la déclaration

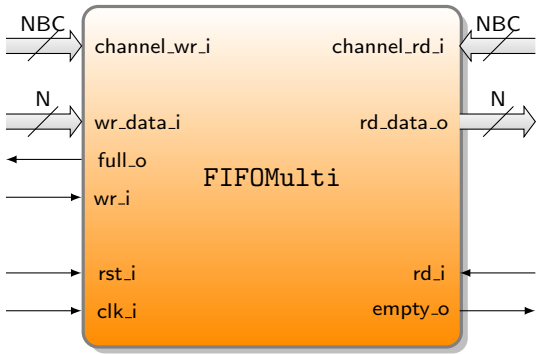
```
signal mon_signal : std_logic := '0';
```

- Problèmes de délai pour la présentation de données à la sortie

# Vérification d'un FIFO multiple

## DUV

- FIFO multiple
- lecture/écriture synchrone
- Deux signaux pour l'état du FIFO
- Un numéro de canal définissant quel FIFO utiliser
  - L'écriture se fait sur le canal `channel_wr_i`
  - La lecture se fait sur le canal `channel_rd_i`



# Vérification d'un FIFO multiple

## Preuve formelle

- Relativement semblable au FIFO simple
  - Utilisation de *generate* pour gérer les canaux

# Vérification d'un FIFO multiple

```
generate

    genvar channel_var;

    for (channel_var = 0; channel_var < NBFIFOS; channel_var = channel_var + 1)
    begin : channel

        property p_full;
        @(posedge clk_i)
          (channel_wr_i == channel_var) |->
            ( full_o == (wcnt[channel_var] == rcnt[channel_var] + FIFO_DEPTH_G) );
        endproperty

        assert_full : assert property (p_full);

    end

endgenerate
```

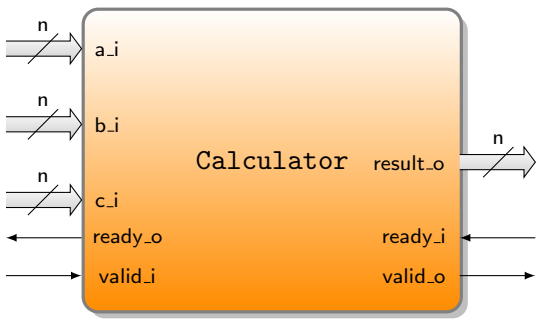
# Erreurs découvertes

- Erreur si une lecture se faisait en même temps qu'une écriture sur le même canal

# Calculateur

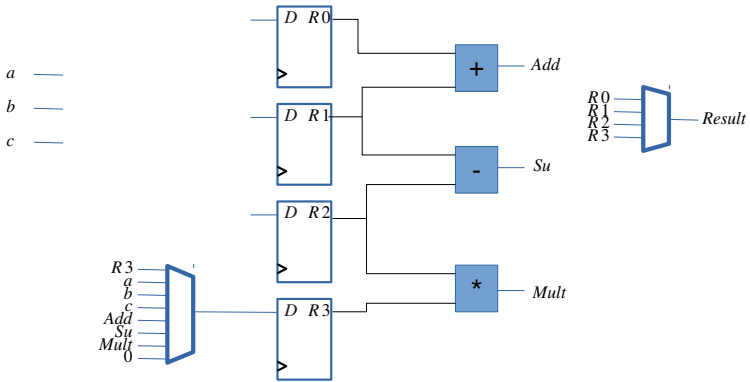
## DUV

- Un composant capable d'effectuer le calcul suivant :
- $F(a, b, c) = a + a + b - c$
  - Peut facilement être un autre calcul
- Plusieurs implémentations :
  - Datapath-control
  - Pipelinée
- Un seul banc de test pour valider les diverses implémentations



# Vérification d'un calculateur

## Chemin de données



# Vérification d'un calculateur

## Preuve formelle

- Propriétés à vérifier
  - P1. Lorsqu'un calcul est lancé un résultat doit être présenté à la sortie après un certain temps
  - P2. Lorsqu'aucun calcul n'est en cours, la sortie `ready_o` doit être activée
  - P3. Le résultat d'un calcul doit correspondre au calcul effectué (vérification des données)
    - Le $n^{ième}$ résultat doit correspondre aux $n^{ièmes}$ entrées données

# Propriété 1

```
property p_valid_eventually;
    @(posedge clk_i)
    (valid_i && ready_o)
        |=>
        ##[0:100] (valid_o);
endproperty
```

# 2 compteurs

```
int wcnt = 0;
int rcnt = 0;

always @(posedge clk_i or posedge rst_i)
    if (rst_i)
        wcnt = 0;
    else if (valid_i && ready_o)
        wcnt = (wcnt + 1);

always @(posedge clk_i or posedge rst_i)
    if (rst_i)
        rcnt = 0;
    else if (valid_o && ready_i)
        rcnt = (rcnt + 1);
```

# Propriété 2

```
property p_ready_once;
    @(posedge clk_i)
    (rcnt == wcnt) |-> (ready_o);
endproperty
```

# Propriété 3

```
property p_data_integrity;
    int cnt;
    logic[DATASIZE-1:0] a;
    logic[DATASIZE-1:0] b;
    logic[DATASIZE-1:0] c;
    @(posedge clk_i)
        (valid_i && ready_o,
        cnt = wcnt,
        a = a_i,
        b = b_i,
        c = c_i
        ) |=>
            ((##[0:$] (valid_o && ready_i & (rcnt==cnt))) |->
            (result_o == a + a + b - c));
endproperty
```

# Banc de tests vs assertions

## Lignes de code nécessaires

| Projet | Lignes TB | Lignes Assertions |
|---|---|---|
| Compteur | 200 | 4 |
| FIFO | 200 | 30 |
| FIFO multiple | 300 | 45 |
| Calculateur | 350 | 50 |

Lignes *utiles*

# Design pour vérification

- L'usage de paramètres génériques autant que possible est intéressant
  - Conception modulaire et réutilisation de code (standard)
  - Permet de réduire l'espace de recherche pour la preuve formelle

$\Rightarrow$ Utilisez des paramètres génériques autant que possible
Ou des constantes dans un paquetage

# Méthode

## TbGenerator

- Avec QuestaFormal, il faut un wrapper qui instancie le DUV et le *bind* au module contenant les assertions
  - Code peu intéressant à écrire

  $\Rightarrow$ Automatisation
- `https://gitlab.com/reds-public/TbGenerator`
  - Génération de squelettes via des scripts Python
    - Banc de test VHDL
    - Banc de test SystemVerilog
    - Wrapper pour preuve formelle
    - Scripts de compilation/simulation/preuve

N'hésitez pas à participer !

# Quelques limitations

## Liées à QuestaFormal

```
always_ff @(posedge clk, posedge rst) begin
    if (rst == 1) begin
        count <= 0;
        internal_count <= 0;
    end
    else begin
        internal_count <= internal_count + 1;
        if (load == 1)
            count <= load_value;
        else begin
            if (en == 1) begin
                if (up_ndown == 1)
                    count <= count + 1;
                else
                    count <= count - 1;
                if (internal_count == 10000000)
                    count <= count -1;
            end
        end
    end
end
```

# Vérification à la HEIG-VD/institut REDS

- Enseignement
  - Au niveau bachelor
    - Conception de bancs de tests en VHDL
  - Au niveau master
    - Conception de bancs de tests en SystemVerilog/TLM
    - Légère introduction aux assertions
- Validation des travaux d'étudiants
  - Jusqu'à il y a une année : bancs de tests
  - Maintenant : complément avec vérification formelle
- Projets
  - Bancs de tests VHDL ou SystemVerilog
  - Preuve formelle pour certains cas

# Conclusion

- Courbe d'apprentissage assez raide
- Ensuite grande efficacité
- Peu de lignes de code
- Efficace pour les chemins de contrôle
- Plus délicat pour les chemins de données
- Manière de penser différente que le design

$\Rightarrow$ A encourager !

# Synthèse de propriétés

- Certaines propriétés peuvent être synthétisées
  - Pour la vérification
    - Embarquées dans un émulateur
  - Pour le design
    - Génération d'un design qui satisfait les propriétés

# Synthèse d'un design

## Approche par automate

- Projet européen PROSYD (`http://www.prosyd.org/`), 2004-2006
- Pour des propriétés PSL, construction d'une machine de Mealy qui satisfait les propriétés
  1. Un *jeu à deux joueurs* entre le système et un environnement
  2. La structure du jeu est un multi-graphe qui représente les variables d'entrée et les variables d'état
     - Chaque noeud représente une combinaison valide des entrées et des variables d'état
     - Chaque flèche représente une transition d'un ensemble d'entrées et de variables d'états vers un autre ensemble valide d'entrées et de variables d'état.
  3. Si l'environnement gagne, alors la spécification n'est pas *réalisable*, et si le système gagne, alors on extrait un BDD (Binary Decision Diagram) représentant le système
  4. Dans le cas où la spécification est réalisable, le BDD correspondant est synthétisé.
- Use cases : Un buffer généralisé et un arbitre de bus AMBA
- Problèmes : Taille du design généré (et applicabilité)

# Synthèse d'un design

## Approche modulaire

- Dominique Borrione et son équipe (Grenoble, France)
- Basé sur une bibliothèque de moniteurs et de générateurs
  - Correspond aux opérateurs PSL
- Les moniteurs détectent les comportements corrects et incorrects
- Les générateurs produisent des séquences
- Permet d'interpéter des assertions LTL-style et SERE-style
- Largement plus efficace en termes de taille que la solution précédente