DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Diseño de Sistemas Digitales con FPGA

Verificación de hardware

Primer Cuatrimestre 2023

Diseño de Sistemas Digitales con FPGA
DC - UBA

Introducción

Verificación de hardware

Hoy vamos a presentar los formalismos y herramientas necesarias para verificar especificaciones de circuitos y protocolos.

Verificación de hardware

El enfoque que se presentará hoy (basado en prueba de propiedades formales) no sólo introduce una perspectiva destinada a mejorar la calidad del hardware sino que también nos acerca a varios modelos (sean explícitos, formales o mentales) para razonar sobre nuestros diseños.

Verificación de hardware

El objetivo va a ser **probar propiedades sobre los estados y las ejecuciones del sistema**.

Estructura de la clase

En la clase hoy vamos a ver:

- Ejemplo motivacional **AXI+FIFO**.

Estructura de la clase

En la clase hoy vamos a ver:

- Ejemplo motivacional **AXI+FIFO**.
- Construcción de un modelo abstraco de nuestro sistema (Kripke).

Estructura de la clase

En la clase hoy vamos a ver:

- Ejemplo motivacional **AXI+FIFO**.
- Construcción de un modelo abstraco de nuestro sistema (Kripke).
- Lógicas temporales.

Estructura de la clase

En la clase hoy vamos a ver:

- Ejemplo motivacional **AXI+FIFO**.
- Construcción de un modelo abstraco de nuestro sistema (Kripke).
- Lógicas temporales.
- Model checking.

Estructura de la clase

En la clase hoy vamos a ver:

- Ejemplo motivacional **AXI+FIFO**.
- Construcción de un modelo abstraco de nuestro sistema (Kripke).
- Lógicas temporales.
- Model checking.
- Assertion based verification (`SVA`, `PSL`, `SystemVerilog`).

Estructura de la clase

En la clase hoy vamos a ver:

- Ejemplo motivacional **AXI+FIFO**.
- Construcción de un modelo abstraco de nuestro sistema (Kripke).
- Lógicas temporales.
- Model checking.
- Assertion based verification (`SVA`, `PSL`, `SystemVerilog`).
- Un ejemplo de verificación con PSL.

Motivación - AXI+FIFO

Imaginemos que queremos diseñar un componente que nos permita intercambiar información entre dos partes sin necesidad de que la parte receptora consuma inmediatamente el dato. Para esto debemos definir:

- El mecanismo de control de flujo, que permita señalizar **cuándo** es posible intercambiar la información (**handshake**).

Imaginemos que queremos diseñar un componente que nos permita intercambiar información entre dos partes sin necesidad de que la parte receptora consuma inmediatamente el dato. Para esto debemos definir:

- El mecanismo de control de flujo, que permita señalizar **cuándo** es posible intercambiar la información (**handshake**).
- La forma en la que vamos a almacenar los datos sin procesar (**buffer**).

Vamos a presentar primero el mecanismo de control de flujo de datos del protocolo `AXI`. Lo definimos del siguiente modo:

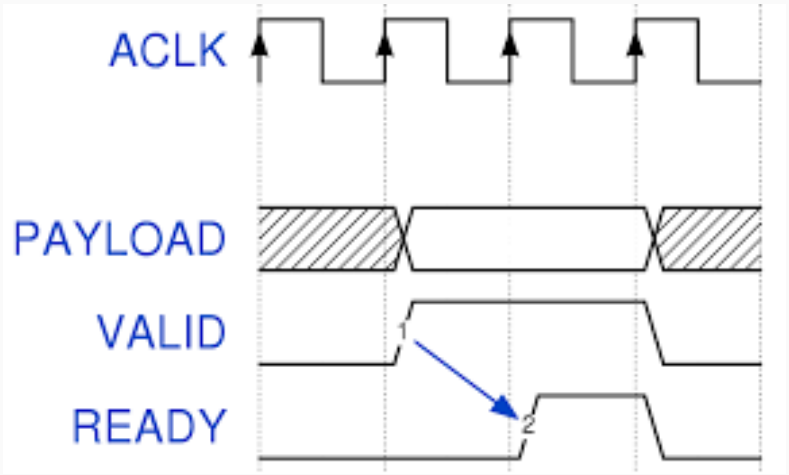
- Para cada dirección de un canal se define una señal, una señal **ready** que depende del receptor y una señal **valid** que depende del emisor.

Vamos a presentar primero el mecanismo de control de flujo de datos del protocolo `AXI`. Lo definimos del siguiente modo:

- Para cada dirección de un canal se define una señal, una señal **ready** que depende del receptor y una señal **valid** que depende del emisor.
- Cuando el emisor considera que el dato presente en el canal es válido levanta la señal **ready**.

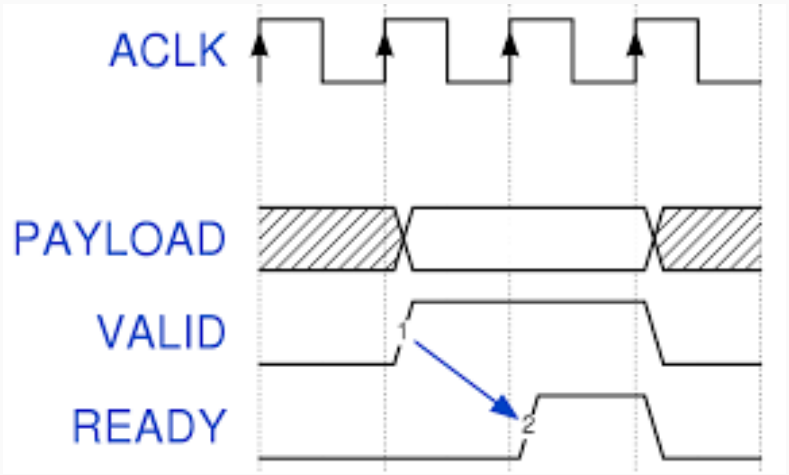
Vamos a presentar primero el mecanismo de control de flujo de datos del protocolo `AXI`. Lo definimos del siguiente modo:

- Para cada dirección de un canal se define una señal, una señal **ready** que depende del receptor y una señal **valid** que depende del emisor.
- Cuando el emisor considera que el dato presente en el canal es válido levanta la señal **ready**.
- Cuando el receptor está en condiciones de leer el dato levanta la señal **ready**.



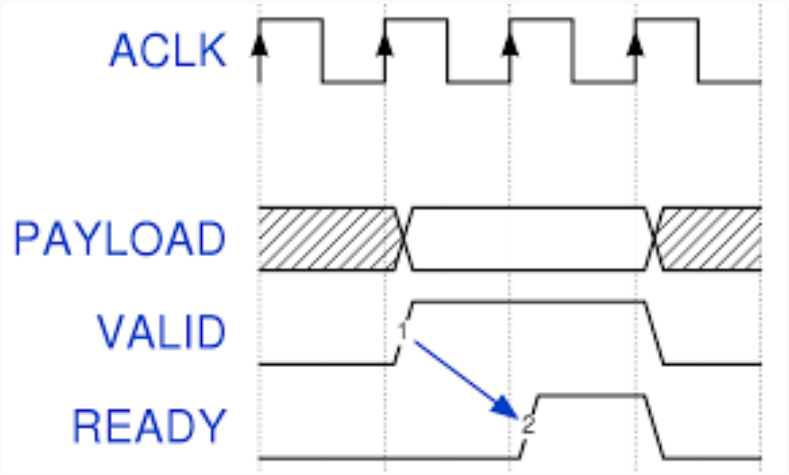
Notemos que:

- Ambas partes están sincronizadas por reloj.



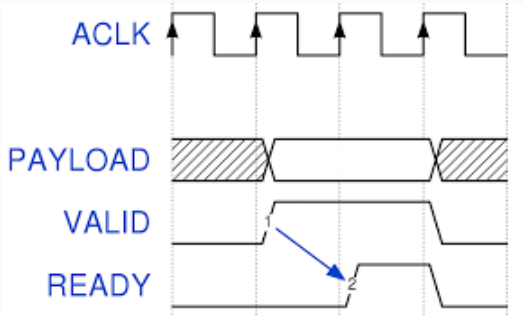
Notemos que:

- Ambas partes están sincronizadas por reloj.
- El mecanismo de control de flujo se replica para cada dirección de cada canal que adhiera al protocolo.



Notemos que:

- Ambas partes están sincronizadas por reloj.
- El mecanismo de control de flujo se replica para cada dirección de cada canal que adhiera al protocolo.
- Solamente intercambian información cuando **ready** y **valid** están arriba.



Para la entrada de un canal la entidad se vería así:

```
entity axi_channel is
  generic (data_width : natural);
  port (
    clk: in std_logic;
    rst: in std_logic;
    in_ready: out std_logic;
    in_valid: in std_logic;
    in_data: in std_logic_vector(data_width - 1 downto 0);
  );
end axi_channel;
```

Ring Buffer

Ahora veremos como almacenar la información. Para esto empleamos un **buffer en anillo (ring buffer)**:

- Los datos se escriben sobre una estructura lineal (`block RAM`).

Ring Buffer

Ahora veremos como almacenar la información. Para esto empleamos un **buffer en anillo (ring buffer)**:

- Los datos se escriben sobre una estructura lineal (`block RAM`).
- Hay un registro **head** que indica dónde comienzan los datos.

Ring Buffer

Ahora veremos como almacenar la información. Para esto empleamos un **buffer en anillo (ring buffer)**:

- Los datos se escriben sobre una estructura lineal (`block RAM`).
- Hay un registro **head** que indica dónde comienzan los datos.
- Hay un registro **tail** que indica dónde terminan los datos.

Ring Buffer

Ahora veremos como almacenar la información. Para esto empleamos un **buffer en anillo (ring buffer)**:

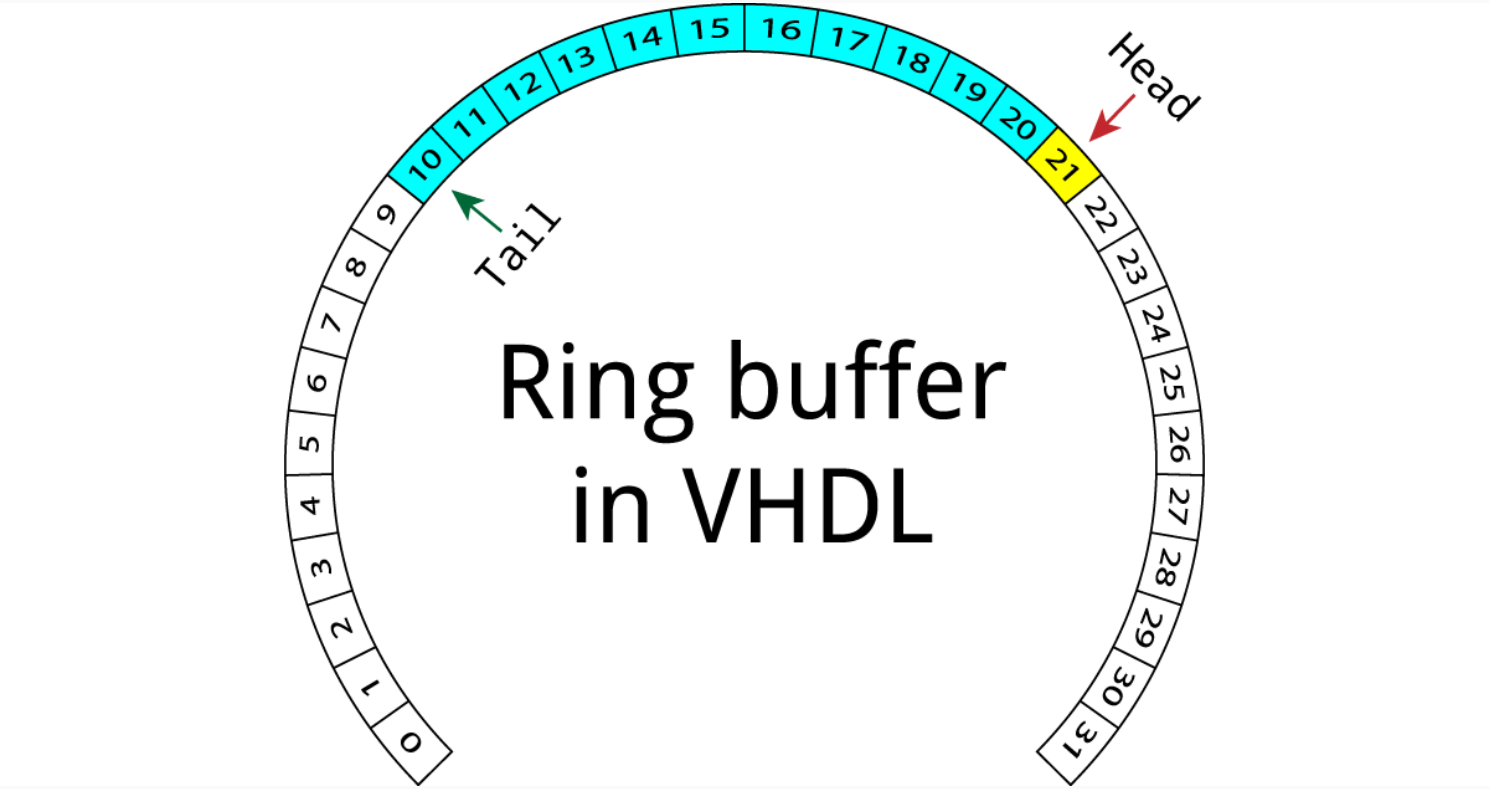
- Los datos se escriben sobre una estructura lineal (`block RAM`).
- Hay un registro **head** que indica dónde comienzan los datos.
- Hay un registro **tail** que indica dónde terminan los datos.
- Al **leer** del buffer, si el valor de **tail** es mayor al largo de la estructura lineal y aún hay datos disponibles, se vuelve al comienzo.

Ring Buffer

Ahora veremos como almacenar la información. Para esto empleamos un **buffer en anillo (ring buffer)**:

- Los datos se escriben sobre una estructura lineal (`block RAM`).
- Hay un registro **head** que indica dónde comienzan los datos.
- Hay un registro **tail** que indica dónde terminan los datos.
- Al **leer** del buffer, si el valor de **tail** es mayor al largo de la estructura lineal y aún hay datos disponibles, se vuelve al comienzo.
- Al **escribir**, si el valor de **head** es mayor al largo de la estructura lineal y aún hay espacio disponible, se vuelve al comienzo.

Ring Buffer


Head
Tail
Ring buffer
in VHDL
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31

Ahora tomemos la **perspectiva formal**, ¿qué observadores queremos derivar del protocolo?

¿Qué señales y elementos con estado observamos del protocolo?

- La señal `clk`.

¿Qué señales y elementos con estado observamos del protocolo?

- La señal `clk`.
- La señal `rst`.

¿Qué señales y elementos con estado observamos del protocolo?

- La señal `clk`.
- La señal `rst`.
- La señal `in_ready`.

¿Qué señales y elementos con estado observamos del protocolo?

- La señal `clk`.
- La señal `rst`.
- La señal `in_ready`.
- La señal `in_valid`.

¿Qué señales y elementos con estado observamos del protocolo?

- La señal `clk`.
- La señal `rst`.
- La señal `in_ready`.
- La señal `in_valid`.
- El dato `in_data`.

Ring Buffer

Ahora observemos al buffer.

Ring Buffer

¿Qué señales y elementos con estado observamos del buffer?

- El valor de `head`.

Ring Buffer

¿Qué señales y elementos con estado observamos del buffer?

- El valor de `head`.
- El valor de `tail`.

Ring Buffer

¿Qué señales y elementos con estado observamos del buffer?

- El valor de `head`.
- El valor de `tail`.

¿Hay propiedades que nos puede venir bien derivar de éstos?

Ring Buffer

¿Hay propiedades que nos puede venir bien derivar de éstos?

- El atributo `count` como diferencia entre `head` y `tail`.

Ring Buffer

¿Hay propiedades que nos puede venir bien derivar de éstos?

- El atributo `count` como diferencia entre `head` y `tail`.
- Podríamos exponer las señales `full` y `empty` para indicar que el buffer se llenó o se encuentra vacío, respectivamente.

Ring Buffer

¿Hay propiedades que nos puede venir bien derivar de éstos?

- El atributo `count` como diferencia entre `head` y `tail`.
- Podríamos exponer las señales `full` y `empty` para indicar que el buffer se llenó o se encuentra vacío, respectivamente.
- ¿Sobre los datos almacenados queremos razonar?

Ring Buffer

Por lo general solemos intentar probar propiedades sobre el estado de las señales y registros internos que no tienen que ver con los datos que se están procesando.

Repaso de señales, estado y atributos:

- Del protocolo tenemos: `clk, rst`, `in_ready`, `in_valid`, `in_data`, `out_ready`, `out_valid`, `out_data` (agregamos la dirección de salida).

Repaso de señales, estado y atributos:

- Del protocolo tenemos: `clk, rst`, `in_ready`, `in_valid`, `in_data`, `out_ready`, `out_valid`, `out_data` (agregamos la dirección de salida).
- Del buffer `head`, `tail`.

Repaso de señales, estado y atributos:

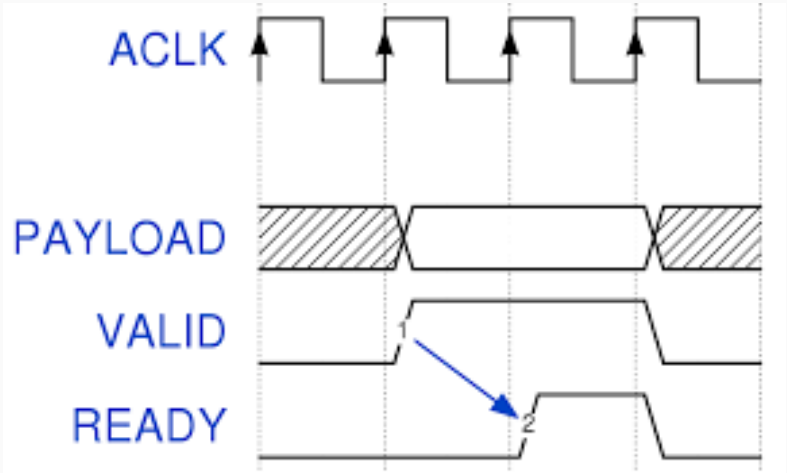
- Del protocolo tenemos: `clk, rst`, `in_ready`, `in_valid`, `in_data`, `out_ready`, `out_valid`, `out_data` (agregamos la dirección de salida).
- Del buffer `head`, `tail`.
- Derivando de los anteriores `count`, `full`, `empty`

Repaso de señales, estado y atributos:

- Del protocolo tenemos: `clk, rst`, `in_ready`, `in_valid`, `in_data`, `out_ready`, `out_valid`, `out_data` (agregamos la dirección de salida).
- Del buffer `head`, `tail`.
- Derivando de los anteriores `count`, `full`, `empty`

Enumeremos propiedades que nos parezcan pertinentes. ¿Cómo las escribimos?

Enumeremos propiedades que nos parezcan pertinentes. ¿Cómo las escribimos?



Notemos que:

- Ambas partes están sincronizadas por reloj.
- El mecanismo de control de flujo se replica para cada dirección de cada canal que adhiera al protocolo.
- Solamente intercambian información cuando **ready** y **valid** están arriba.

Ring Buffer - Machete

Enumeremos propiedades que nos parezcan pertinentes. ¿Cómo las escribimos?

- Los datos se escriben sobre una estructura lineal (`block RAM`).
- Hay un registro **head** que indica dónde comienzan los datos.
- Hay un registro **tail** que indica dónde terminan los datos.
- Al **leer** del buffer, si el valor de **tail** es mayor al largo de la estructura lineal y aún hay datos disponibles, se vuelve al comienzo.
- Al **escribir**, si el valor de **head** es mayor al largo de la estructura lineal y aún hay espacio disponible, se vuelve al comienzo.

AXI + FIFO, propiedades

Enumeremos propiedades que nos parezcan pertinentes. ¿Cómo las escribimos?

- Por ejemplo siempre que `rst` esté alto debería implicar que `in_ready`,`out_valid` estén bajo y `count` en 0.

AXI + FIFO, propiedades

Enumeremos propiedades que nos parezcan pertinentes. ¿Cómo las escribimos?

- Por ejemplo siempre que `rst` esté alto debería implicar que `in_ready`,`out_valid` estén bajo y `count` en 0.
- Siempre que `count` sea 0 debe valer que `out_valid` esté bajo.

AXI + FIFO, propiedades

Enumeremos propiedades que nos parezcan pertinentes. ¿Cómo las escribimos?

- Por ejemplo siempre que `rst` esté alto debería implicar que `in_ready`,`out_valid` estén bajo y `count` en 0.
- Siempre que `count` sea 0 debe valer que `out_valid` esté bajo.
- Siempre que valga `in_valid` y no `in_ready` y no `rst` los valores de `in_valid` y `in_data` deben quedar inalterados.

AXI + FIFO, propiedades

Escribamos lo anterior en lógica proposicional.

- Siempre $rst \implies \neg in\_ready \wedge out\_valid \wedge count = 0$.

AXI + FIFO, propiedades

Escribamos lo anterior en lógica proposicional.

- Siempre $rst \implies \neg in\_ready \land out\_valid \land count = 0$.
- Siempre $count = 0 \implies \neg out\_valid$.

AXI + FIFO, propiedades

Escribamos lo anterior en lógica proposicional.

- Siempre $rst \implies \neg in\_ready \land out\_valid \land count = 0$.
- Siempre $count = 0 \implies \neg out\_valid$.
- Siempre $in\_valid \land \neg in\_ready \land \neg rst \implies in\_valid_t = in\_valid_{t+1} \land in\_data_t = in\_data_{t+1}$.

AXI + FIFO, propiedades

Escribamos lo anterior en lógica proposicional.

- Siempre $rst \implies \neg in\_ready \land out\_valid \land count = 0$.
- Siempre $count = 0 \implies \neg out\_valid$.
- Siempre $in\_valid \land \neg in\_ready \land \neg rst \implies in\_valid_t = in\_valid_{t+1} \land in\_data_t = in\_data_{t+1}$.

Vamos a querer formalizar toda la expresión.

¿Qué son **siempre**, in_valid_t y in_valid_{t+1}?

AXI + FIFO, propiedades

- $\square(rst \implies \neg in\_ready \wedge out\_valid \wedge count = 0)$.

AXI + FIFO, propiedades

- $\square(rst \implies \neg in\_ready \wedge out\_valid \wedge count = 0)$.
- $\square(count = 0 \implies \neg out\_valid)$.

AXI + FIFO, propiedades

- $\square(rst \implies \neg in\_ready \wedge out\_valid \wedge count = 0)$.
- $\square(count = 0 \implies \neg out\_valid)$.
- $\square(in\_valid \wedge \neg in\_ready \wedge \neg rst \implies stable(in\_valid) \wedge stable(in\_data))$.

Formalizando propiedades

Necesitamos extender nuestras propiedades más allá de la lógica proposicional, que si nos permite razonar sobre los estados individuales del sistema no alcanza para expresa la relación de progreso entre ellos, que llamaremos **comportamiento** y será expresado con **lógicas temporales**.

Abstracción del modelo

Abstracción

Para poder comprender como escribir y leer nuestras propiedades debemos:

- Formalizar las definiciones involucradas.

Abstracción

Para poder comprender como escribir y leer nuestras propiedades debemos:

- Formalizar las definiciones involucradas.
- Construir una abstracción de nuestro sistema.

Definiciones informales

Repasemos definiciones:

- Un **estado** es una asignación de valores a las señales y registros de nuestro sistema.

Definiciones informales

Repasemos definiciones:

- Un **estado** es una asignación de valores a las señales y registros de nuestro sistema.
- Una **traza** es una secuencia de estados con un estado inicial y un orden total.

Definiciones informales

Repasemos definiciones:

- Un **estado** es una asignación de valores a las señales y registros de nuestro sistema.
- Una **traza** es una secuencia de estados con un estado inicial y un orden total.
- Una **traza pertenece a un sistema** si el estado inicial y todos los estados generados son compatibles con el sistema.

Definiciones informales

En nuestro ejemplo (observando solamente *clk*, *rst*, *in_ready*, *in_valid*) de **AXI + FIFO** decimos que:

- $clk \wedge \neg rst \wedge \neg in\_ready \wedge in\_valid$ es un estado inicial válido del sistema.

Definiciones informales

En nuestro ejemplo (observando solamente *clk*, *rst*, *in_ready*, *in_valid*) de **AXI + FIFO** decimos que:

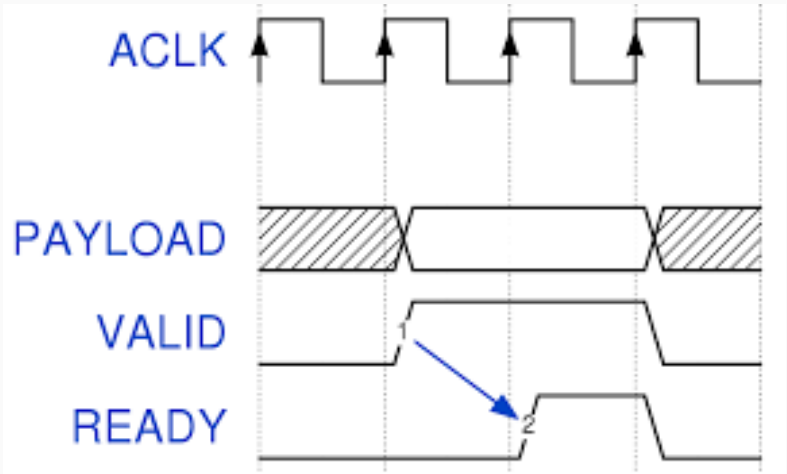
- $clk \wedge \neg rst \wedge \neg in\_ready \wedge in\_valid$ es un estado inicial válido del sistema.
- $t = (clk \wedge \neg rst \wedge \neg in\_ready \wedge in\_valid),(clk \wedge \neg rst \wedge in\_ready \wedge in\_valid),\ldots$ es una traza.

Definiciones informales

En nuestro ejemplo (observando solamente *clk*, *rst*, *in_ready*, *in_valid*) de **AXI + FIFO** decimos que:

- $clk \wedge \neg rst \wedge \neg in\_ready \wedge in\_valid$ es un estado inicial válido del sistema.
- $t = (clk \wedge \neg rst \wedge \neg in\_ready \wedge in\_valid),(clk \wedge \neg rst \wedge in\_ready \wedge in\_valid),\ldots$ es una traza.
- $t = (clk \wedge \neg rst \wedge \neg in\_ready \wedge \neg in\_valid),(clk \wedge \neg rst \wedge in\_ready \wedge \neg in\_valid),\ldots$ no es válida para nuestro sistema porque no podemos levantar *in_ready* sin observar *in_valid*.

Trazas en AXI



Notemos que $t = (clk \wedge \neg rst \wedge \neg in\_ready \wedge \neg in\_valid), (clk \wedge \neg rst \wedge in\_ready \wedge \neg in\_valid), \ldots$ no es válida para nuestro sistema porque no podemos levantar *in_ready* sin observar *in_valid*.

Definiciones formales

Vamos a dar las definiciones formales de lo anterior.

- AP son los elementos binarios pertenecientes a nuestro sistema (señales y memoria) y serán llamados **átomos proposicionales**.

Definiciones formales

Vamos a dar las definiciones formales de lo anterior.

- AP son los elementos binarios pertenecientes a nuestro sistema (señales y memoria) y serán llamados **átomos proposicionales**.
- Un **estado** s pertenece a 2^{AP}, lo que equivale a decir que es una valuación de nuestros átomos proposcionales (asignación de valores de verdad a cada variables binaria).

Definiciones formales

Vamos a dar las definiciones formales de lo anterior.

- AP son los elementos binarios pertenecientes a nuestro sistema (señales y memoria) y serán llamados **átomos proposicionales**.
- Un **estado** s pertenece a 2^{AP}, lo que equivale a decir que es una valuación de nuestros átomos proposcionales (asignación de valores de verdad a cada variables binaria).
- Una secuencia $s_1, s_2, \ldots$ donde $s_i \in 2^{AP}$ es denominada **traza**.

Construcción del modelo

Con estas definiciones vamos a construir un modelo basado en autómatas como **abstracción de nuestro sistema**.

Estructura de Kripke

Podemos vincular a nuestros estados para describir ejecuciones posibles en un sistema con un tipo de autómata llamado **estructura de Kripke**. Supongamos que $M = (S, I, R, L)$ es una estructura de Kripke

- S es un conjunto de estados.

Estructura de Kripke

Podemos vincular a nuestros estados para describir ejecuciones posibles en un sistema con un tipo de autómata llamado **estructura de Kripke**. Supongamos que $M = (S, I, R, L)$ es una estructura de Kripke

- S es un conjunto de estados.
- $I \subseteq S$ es el conjunto de estados iniciales.

Estructura de Kripke

Podemos vincular a nuestros estados para describir ejecuciones posibles en un sistema con un tipo de autómata llamado **estructura de Kripke**. Supongamos que $M = (S, I, R, L)$ es una estructura de Kripke

- S es un conjunto de estados.
- $I \subseteq S$ es el conjunto de estados iniciales.
- $R \subseteq S \times S$ es la función de transición que satisface $\forall s \in S \exists s' \in S$ tal que $(s, s') \in R$ (o sea que todo estado tiene sucesor).

Estructura de Kripke

Podemos vincular a nuestros estados para describir ejecuciones posibles en un sistema con un tipo de autómata llamado **estructura de Kripke**. Supongamos que $M = (S, I, R, L)$ es una estructura de Kripke

- S es un conjunto de estados.
- $I \subseteq S$ es el conjunto de estados iniciales.
- $R \subseteq S \times S$ es la función de transición que satisface $\forall s \in S \exists s' \in S$ tal que $(s, s') \in R$ (o sea que todo estado tiene sucesor).
- $L : S \to 2^{AP}$ es la función de etiquetado (que indica la valuación del estado).

AXI - Kripke

En el ejemplo simplificado de **AXI**, donde el `clk` indica el avance entre estados, y el `rst` el regreso a ciertos estados determinados, nuestra estructura quedaría $M_{AXI} = (S, I, R, L)$ es una estructura de Kripke

- $S = \{0, 1, 2\}$

AXI - Kripke

En el ejemplo simplificado de **AXI**, donde el `clk` indica el avance entre estados, y el `rst` el regreso a ciertos estados determinados, nuestra estructura quedaría $M_{AXI} = (S, I, R, L)$ es una estructura de Kripke

- $S = \{0, 1, 2\}$
- $I = \{0\}$

AXI - Kripke

En el ejemplo simplificado de **AXI**, donde el `clk` indica el avance entre estados, y el `rst` el regreso a ciertos estados determinados, nuestra estructura quedaría $M_{AXI} = (S, I, R, L)$ es una estructura de Kripke

- $S = \{0, 1, 2\}$
- $I = \{0\}$
- $R = \{(0,0), (0,1), (1,0), (1,1), (1,2), (2,2), (2,0)\}$

AXI - Kripke

En el ejemplo simplificado de **AXI**, donde el `clk` indica el avance entre estados, y el `rst` el regreso a ciertos estados determinados, nuestra estructura quedaría $M_{AXI} = (S, I, R, L)$ es una estructura de Kripke

- $S = \{0, 1, 2\}$
- $I = \{0\}$
- $R = \{(0,0), (0,1), (1,0), (1,1), (1,2), (2,2), (2,0)\}$
- $L = \{(0, (\overline{in\_ready}, in\_valid)),$
  $(1, (\overline{in\_ready}, in\_valid)), (2, (in\_ready, in\_valid))\}$.

AXI - Kripke

Damos una representación visual de la estructura de Kripke del protocolo simplificado.

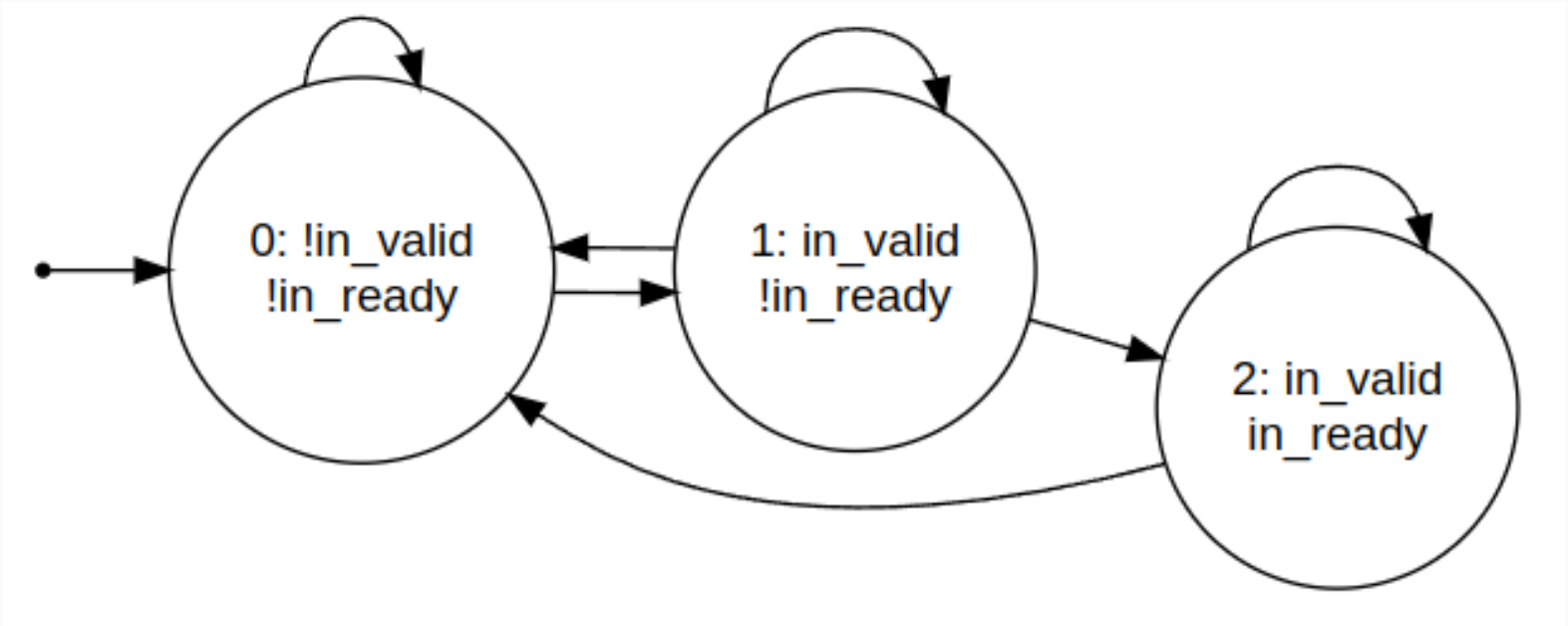
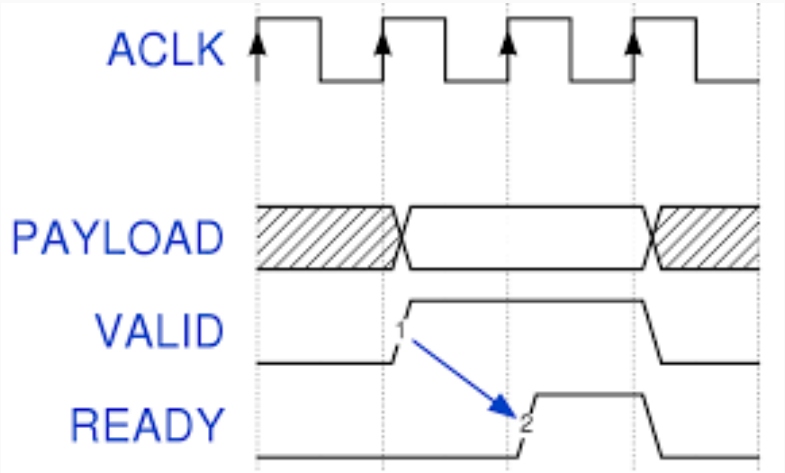
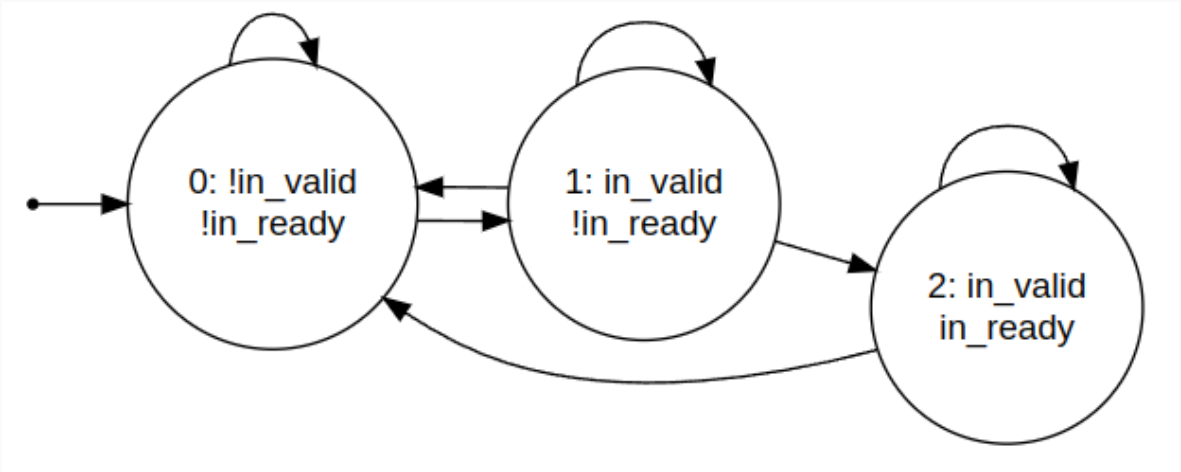


Diagrama de tiempos - Kripke

El diagrama de tiempos induce una traza sobre la estructura de Kripke que derivamos del protocolo, en este caso $0, 1, 2, 0 \ldots$.





Lógicas temporales

Ahora podemos regresar a nuestras propiedades de **AXI + FIFO**.

- $\square(rst \implies \neg in\_ready \wedge out\_valid \wedge count = 0)$.

Ahora podemos regresar a nuestras propiedades de **AXI + FIFO**.

- $\square(rst \implies \neg in\_ready \wedge out\_valid \wedge count = 0)$.
- $\square(count = 0 \implies \neg out\_valid)$.

Ahora podemos regresar a nuestras propiedades de **AXI + FIFO**.

- $\square(rst \implies \neg in\_ready \wedge out\_valid \wedge count = 0)$.
- $\square(count = 0 \implies \neg out\_valid)$.
- $\square(in\_valid \wedge \neg in\_ready \wedge \neg rst \implies stable(in\_valid) \wedge stable(in\_data))$.

La abstracción del modelo se hace a través de un automáta y las propiedades se definen con lógicas temporales.

Las lógicas temporales son un tipo de **lógica modal**, que a su vez es una extensión de la **lógica proposicional**. Las lógicas modales cuentan con:

- Un **operador existencial** cuya satisfacción implica que existe alguna evaluación que satisface a la fórmula ligada al operador.

Las lógicas temporales son un tipo de **lógica modal**, que a su vez es una extensión de la **lógica proposicional**. Las lógicas modales cuentan con:

- Un **operador existencial** cuya satisfacción implica que existe alguna evaluación que satisface a la fórmula ligada al operador.
- Un **operador universal** cuya satisfacción implica que todas las evaluaciones satisfacen a la fórmula ligada al operador.

Las lógicas temporales son un tipo de **lógica modal**, que a su vez es una extensión de la **lógica proposicional**. Las lógicas modales cuentan con:

- Un **operador existencial** cuya satisfacción implica que existe alguna evaluación que satisface a la fórmula ligada al operador.
- Un **operador universal** cuya satisfacción implica que todas las evaluaciones satisfacen a la fórmula ligada al operador.

En el caso de la lógica temporal linear, que es la que vamos a ver, ambos operadores se interpretan sobre trazas.

Para la lógica temporal linear, dada una traza $t = (s_1, s_2, \ldots)$ perteneciente a las valuaciones de AP, la semántica de sus fórmulas φ se definen en base a si t las satisface ($t \models \varphi$).

- (Proposición) Para $p \in AP$, $s_i \models p$ sii $p \in s_i$.

Para la lógica temporal linear, dada una traza $t = (s_1, s_2, \ldots)$ perteneciente a las valuaciones de AP, la semántica de sus fórmulas φ se definen en base a si t las satisface ($t \models \varphi$).

- (Proposición) Para $p \in AP$, $s_i \models p$ sii $p \in s_i$.
- (Negación) $s_i \models \neg\varphi$ sii $s_i \not\models \varphi$.

Para la lógica temporal linear, dada una traza $t = (s_1, s_2, \ldots)$ perteneciente a las valuaciones de AP, la semántica de sus fórmulas φ se definen en base a si t las satisface ($t \models \varphi$).

- (Proposición) Para $p \in AP$, $s_i \models p$ sii $p \in s_i$.
- (Negación) $s_i \models \neg\varphi$ sii $s_i \not\models \varphi$.
- (Disjunción) $s_i \models \varphi \vee \psi$ sii $s_i \models \varphi \vee s_i \models \psi$.

Para la lógica temporal linear, dada una traza $t = (s_1, s_2, \ldots)$ perteneciente a las valuaciones de AP, la semántica de sus fórmulas φ se definen en base a si t las satisface ($t \models \varphi$).

- (Proposición) Para $p \in AP$, $s_i \models p$ sii $p \in s_i$.
- (Negación) $s_i \models \neg\varphi$ sii $s_i \not\models \varphi$.
- (Disjunción) $s_i \models \varphi \vee \psi$ sii $s_i \models \varphi \vee s_i \models \psi$.
- (Próximo estado) $s_i \models \bigcirc\varphi$ sii $s_{i+1} \models \varphi$.

Para la lógica temporal linear, dada una traza $t = (s_1, s_2, \ldots)$ perteneciente a las valuaciones de AP, la semántica de sus fórmulas φ se definen en base a si t las satisface ($t \models \varphi$).

- (Proposición) Para $p \in AP$, $s_i \models p$ sii $p \in s_i$.
- (Negación) $s_i \models \neg\varphi$ sii $s_i \not\models \varphi$.
- (Disjunción) $s_i \models \varphi \vee \psi$ sii $s_i \models \varphi \vee s_i \models \psi$.
- (Próximo estado) $s_i \models \bigcirc\varphi$ sii $s_{i+1} \models \varphi$.
- (Hasta, Until) $s_i \models \varphi \mathcal{U} \psi$ sii existe un k tal que $s_k \models \psi$ y para todos los $j : i \leq j \leq k$ vale $s_j \models \varphi$.

Derivamos dos operadores más de los anteriores.

- (Eventualmente) $\Diamond\varphi = \mathit{true}\mathcal{U}\varphi$.

Derivamos dos operadores más de los anteriores.

- (Eventualmente) $\Diamond\varphi = \mathit{true}\mathcal{U}\varphi$.
- (Siempre) $\Box\varphi = \neg\Diamond\neg\varphi$.

Derivamos dos operadores más de los anteriores.

- (Eventualmente) $\Diamond\varphi = \mathit{true}\mathcal{U}\varphi$.
- (Siempre) $\Box\varphi = \neg\Diamond\neg\varphi$.

Con esto van a poder entender todos los operadores de los lenguajes temporales de verificación de hardware.

Observación

Lo más importante es entender el significado de cada operador.

- (Próximo estado) $s_i \models \bigcirc\varphi$, en el siguiente pulso de reloj el estado de las señales y la memoria debe satisfacer φ.

Observación

Lo más importante es entender el significado de cada operador.

- (Próximo estado) $s_i \models \bigcirc\varphi$, en el siguiente pulso de reloj el estado de las señales y la memoria debe satisfacer φ.
- (Hasta, Until) $s_i \models \varphi\mathcal{U}\psi$, el estado de las señales y la memoria debe satisfacer φ hasta que llegue a un estado donde pueda satisfacer ψ.

Observación

Lo más importante es entender el significado de cada operador.

- (Próximo estado) $s_i \models \bigcirc\varphi$, en el siguiente pulso de reloj el estado de las señales y la memoria debe satisfacer φ.
- (Hasta, Until) $s_i \models \varphi\mathcal{U}\psi$, el estado de las señales y la memoria debe satisfacer φ hasta que llegue a un estado donde pueda satisfacer ψ.
- (Eventualmente) $\Diamond\varphi$, en algún momento el estado debe satisfacer φ.

Observación

Lo más importante es entender el significado de cada operador.

- (Próximo estado) $s_i \models \bigcirc\varphi$, en el siguiente pulso de reloj el estado de las señales y la memoria debe satisfacer φ.
- (Hasta, Until) $s_i \models \varphi\mathcal{U}\psi$, el estado de las señales y la memoria debe satisfacer φ hasta que llegue a un estado donde pueda satisfacer ψ.
- (Eventualmente) $\Diamond\varphi$, en algún momento el estado debe satisfacer φ.
- (Siempre) $\Box\varphi$, en todo estado de la ejecución debe valer φ.

En un último repaso, volvemos a las propiedades de nuestro buffer:

- $\square(rst \implies \neg in\_ready \wedge out\_valid \wedge count = 0)$.
- $\square(count = 0 \implies \neg out\_valid)$.
- $\square(in\_valid \wedge \neg in\_ready \wedge \neg rst \implies stable(in\_valid) \wedge stable(in\_data))$.

En un último repaso, volvemos a las propiedades de nuestro buffer:

- $\square(rst \implies \neg in\_ready \wedge out\_valid \wedge count = 0)$.
- $\square(count = 0 \implies \neg out\_valid)$.
- $\square(in\_valid \wedge \neg in\_ready \wedge \neg rst \implies stable(in\_valid) \wedge stable(in\_data))$.

Ya podemos interpretar el significa de $\square$, y podemos imaginar que *stable* indica que si una señal vale en un punto de la traza, debe quedar invariante en el inmediato siguiente ($s_i \models p \iff \bigcirc p$).

Model Checking

Probando propiedades

Volvamos al modelo que se construyó de nuestro sistema y veamos cómo se vincula con las propiedades a través de la relación de satisfacción.

Conformidad

La lógica temporal linear predica sobre trazas y vamos a decir que una traza $t = (s_1, s_2, \ldots)$ es **conforme** a una estructura M sí y sólo sí:

$$\forall s_i, s_{i+1} \in t : (\exists (S_i, S_{i+1}) \in R : (L(S_i) = s_i \wedge L(S_{i+1}) = s_{i+1}))$$

O sea, que las trazas conformes son las que se pueden generar siguiendo los caminos posibles en el modelo (Kripke).

Model checking

Vamos a decir que un modelo M satisface una fórmula φ, escrito como:

$$M \models \varphi$$

Sí y sólo sí toda traza t conforme a M que empieza en I un estado inicial, satisface a φ. O sea que todas las ejecuciones posibles del modelo satisfacen a la fórmula.

Al proceso de verificar que un modelo satisface una fórmula se lo conoce como model checking.

Assumption - Guarantee

En nuestros circuitos no podemos restringir las entradas, pero solemos suponer que se van a cumplir ciertas propiedades, por ejemplo que una señal va ponerse en el valor alto en algún momento o que dos señales de entrada nunca sucederán a la vez. Es por esto que suele ser muy común dividir las presunciones de entrada (ambiente) de los objetivos que tenemos que cumplir (sistema).

Assumption - Guarantee

Esto se suele conocer como `assume guarantee reasoning` y es una forma más de hacer uso del diseño por contratos. Nuestra versión de la fórmula de model checking puede escribirse en ese caso cómo:

$$M \models \varphi_e \implies \varphi_s$$

Assumption - Guarantee

Esto se suele conocer como `assume guarantee reasoning` y es una forma más de hacer uso del diseño por contratos. Nuestra versión de la fórmula de model checking puede escribirse en ese caso cómo:

$$M \models \varphi_e \implies \varphi_s$$

Dónde φ_e está restringiendo los casos en los que es necesario satisfacer φ_s, que si el antecedente no vale, la implicacón es vacuamente cierta.

Existe un formalismo, μ cálculo, que permite derivar algoritmos de satisfacción de forma mecánica a partir de una fórmula de lógica temporal linear. Con lo cuál podríamos completar las tres partes de la ecuación.

$$M \models \varphi$$

$$M \models \varphi$$

Aquí M es el modelo de Kripke que abstraemos de nuestro sistema, φ es la fórmula que nos interesa probar y $\models$ es el mecanismo de prueba, que puede ser un algoritmo escrito de forma manual o automática.

Assertion Based Verification

Assertion Based Verification

Assertion based verification (verificación basada en aserciones) es una definición que engloba un conjunto de técnicas y herramientas para permitir verificar sistemas digitales en base aserciones expresadas en algún tipo de lógica temporal.

Existen distintas versiones de lenguajes o bibliotecas que implementan ABV, por ejemplo:

- **Open Verification Library** (OVL) `https://www.accellera.org/activities/working-groups/ovl`

Existen distintas versiones de lenguajes o bibliotecas que implementan ABV, por ejemplo:

- **Open Verification Library** (OVL) `https://www.accellera.org/activities/working-groups/ovl`
- **SystemVerilog Assertions** (SVA) `https://www.systemverilog.io/verification/sva-basics/`

Existen distintas versiones de lenguajes o bibliotecas que implementan ABV, por ejemplo:

- **Open Verification Library** (OVL) `https://www.accellera.org/activities/working-groups/ovl`
- **SystemVerilog Assertions** (SVA) `https://www.systemverilog.io/verification/sva-basics/`
- **Property Specification Language** (PSL) `https://en.wikipedia.org/wiki/Property_Specification_Language`

En este curso veremos PSL, pero los fundamentos deberían ayudar a trabajar con el resto de las herramientas.

AXI+FIFO+PSL

Property Specification Language

- Vamos a revisar los archivos del proyecto de éste sitio: `https://vhdlwhiz.com/formal-verification-in-vhdl-using-psl`.

- Vamos a revisar los archivos del proyecto de éste sitio: `https://vhdlwhiz.com/formal-verification-in-vhdl-using-psl`.
- Aquí se explica con un poco de detalle el circuito del **AXI+FIFO**: `https://vhdlwhiz.com/axi-fifo/`.

Property Specification Language

- Vamos a revisar los archivos del proyecto de éste sitio: `https://vhdlwhiz.com/formal-verification-in-vhdl-using-psl`.
- Aquí se explica con un poco de detalle el circuito del **AXI+FIFO**: `https://vhdlwhiz.com/axi-fifo/`.
- Y aquí hay una serie de ejemplos de `PSL` para `GHDL`: `https://github.com/tmeissner/psl_with_ghdl`.

Veamos cómo puede utilizarse `SymbiYosys` junto con `Yosys` para probar propiedades de nuestros sistemas.

- `Yosys` es un framework cuyo objetivo es permitir la síntesis de especificaciones `RTL`.

Veamos cómo puede utilizarse `SymbiYosys` junto con `Yosys` para probar propiedades de nuestros sistemas.

- `Yosys` es un framework cuyo objetivo es permitir la síntesis de especificaciones `RTL`.
- `SymbiYosys` extiende `Yosys` con capacidades de verificación.

Veamos cómo puede utilizarse `SymbiYosys` junto con `Yosys` para probar propiedades de nuestros sistemas.

- `Yosys` es un framework cuyo objetivo es permitir la síntesis de especificaciones `RTL`.
- `SymbiYosys` extiende `Yosys` con capacidades de verificación.

`https://yosyshq.readthedocs.io/projects/sby/en/latest`

Con `SymbiYosys` podemos:

- Hacer verificación acotada (en el largo de las trazas) de propiedades de safety ($\Box\neg in\_ready$).

Con `SymbiYosys` podemos:

- Hacer verificación acotada (en el largo de las trazas) de propiedades de safety ($\Box\neg \mathit{in\_ready}$).
- Hacer verificación no acotada (en el largo de las trazas) de propiedades de safety.

Con `SymbiYosys` podemos:

- Hacer verificación acotada (en el largo de las trazas) de propiedades de safety ($\Box\neg \mathit{in\_ready}$).
- Hacer verificación no acotada (en el largo de las trazas) de propiedades de safety.
- Generar testbenchs a partir de criterios de cobertura de nuestro diseño.

Con `SymbiYosys` podemos:

- Hacer verificación acotada (en el largo de las trazas) de propiedades de safety ($\Box\neg \mathit{in\_ready}$).
- Hacer verificación no acotada (en el largo de las trazas) de propiedades de safety.
- Generar testbenchs a partir de criterios de cobertura de nuestro diseño.
- Verificar propiedades de liveness, propiedades que siempre deberían poder ocurrir ($\Box\Diamond \mathit{in\_ready}$).

Para poder correr los ejemplos:

```
sudo apt-get install build-essential clang bison flex
    libreadline-dev gawk tcl-dev libffi-dev git graphviz
    xdot pkg-config python3 libboost-system-dev libboost-
    python-dev libboost-filesystem-dev zlib1g-dev
git clone https://github.com/YosysHQ/yosys
cd yosys
make config-gcc
make
sudo make install
```

```
sudo apt install gnat-8
git clone https://github.com/ghdl/ghdl
cd ghdl
./configure --prefix=/usr/local
make
sudo make install
cd ..
git clone https://github.com/ghdl/ghdl-yosys-plugin
cd ghdl-yosys-plugin
make
sudo cp ghdl.so /usr/local/share/yosys/plugins/ghdl.so
```

Proyecto con SymbiYosys

Nuestro proyecto típico tendrá:

- Un archivo vhdl `axi_fifo.vhdl`.

Proyecto con SymbiYosys

Nuestro proyecto típico tendrá:

- Un archivo vhdl `axi_fifo.vhdl`.
- Un testbench `axi_fifo_tb.vhdl`.

Proyecto con SymbiYosys

Nuestro proyecto típico tendrá:

- Un archivo vhdl `axi_fifo.vhdl`.
- Un testbench `axi_fifo_tb.vhdl`.
- Un archivo de especificación de propiedades PSL `axi_fifo.psl`.

Proyecto con SymbiYosys

Nuestro proyecto típico tendrá:

- Un archivo vhdl `axi_fifo.vhdl`.
- Un testbench `axi_fifo_tb.vhdl`.
- Un archivo de especificación de propiedades PSL `axi_fifo.psl`.
- Un archivo de configuración para indicar a SymbiYosys el tipo de verificación a realizar `axi_fifo.sby`.

Proyecto con SymbiYosys

Nuestro proyecto típico tendrá:

- Un archivo vhdl `axi_fifo.vhdl`.
- Un testbench `axi_fifo_tb.vhdl`.
- Un archivo de especificación de propiedades PSL `axi_fifo.psl`.
- Un archivo de configuración para indicar a SymbiYosys el tipo de verificación a realizar `axi_fifo.sby`.
- Un `Makefile` para ejecutar todo.

Proyecto con SymbiYosys

Nos vamos a concentrar solamente en los archivos `vhdl` y `psl` que son los que describen comportamiento y propiedades respectivamente.

Proyecto con SymbiYosys

Vamos a verlos en el editor.

Cierre

Repaso de hoy

En la clase hoy vimos:

- Ejemplo motivacional **AXI+FIFO**.

Repaso de hoy

En la clase hoy vimos:

- Ejemplo motivacional **AXI+FIFO**.
- Construcción de un modelo abstraco de nuestro sistema (Kripke).

Repaso de hoy

En la clase hoy vimos:

- Ejemplo motivacional **AXI+FIFO**.
- Construcción de un modelo abstraco de nuestro sistema (Kripke).
- Lógicas temporales.

Repaso de hoy

En la clase hoy vimos:

- Ejemplo motivacional **AXI+FIFO**.
- Construcción de un modelo abstraco de nuestro sistema (Kripke).
- Lógicas temporales.
- Model checking.

Repaso de hoy

En la clase hoy vimos:

- Ejemplo motivacional **AXI+FIFO**.
- Construcción de un modelo abstraco de nuestro sistema (Kripke).
- Lógicas temporales.
- Model checking.
- Assertion based verification (`SVA, PSL, SystemVerilog`).

Repaso de hoy

En la clase hoy vimos:

- Ejemplo motivacional **AXI+FIFO**.
- Construcción de un modelo abstraco de nuestro sistema (Kripke).
- Lógicas temporales.
- Model checking.
- Assertion based verification (`SVA, PSL, SystemVerilog`).
- Un ejemplo de verificación con PSL.

Referencias

Bibliografía fundamental:

- Synthesis of Reactive(1) Designs https://www.academia.edu/download/30755751/E_Allen_Emerson_Verification_Model_Checking_and.pdf#page=373
- On the Synthesis of a Reactive Module https://dl.acm.org/doi/pdf/10.1145/75277.75293
- Formal Verification of a Flash Memory Device Driver - an Experience Report https://koasas.kaist.ac.kr/bitstream/10203/23419/1/spin08.pdf
- Modal Mu-Calculi https://www.pure.ed.ac.uk/ws/files/16483207/Modal_Mu_Calculi.pdf

Referencias

Bibliografía complementaria:

- On Well-Separation of GR(1) Specifications `https://dl.acm.org/doi/pdf/10.1145/2950290.2950300`
- Compositional Reasoning in Model Checking `https://apps.dtic.mil/sti/pdfs/ADA339195.pdf`
- Temporal Logic and Fair Discrete Systems `https://scholar.archive.org/work/5pfi376nfngxhaplcdiap5eib4/access/wayback/http://pdfs.semanticscholar.org/1347/fc3eaf3d8fbd94bd0d7a32dc944cf4c663ad.pdf`

Verificación - Fin

Cierre y preguntas.