Tiempos, Simulación y Verificación

David Alejandro González Márquez

Programación de softcores en FPGAs
Programa de Profesoras/es Visitantes
Departamento de computación
Universidad de Buenos Aires

Clase disponible en: `https://github.com/fokerman/fpgaSoftcoreProgrammingCourse`

Tiempos, Simulación y Verificación

1 - Temporización de circuitos combinacionales

- Propagación de señales.
- Aparición de *glitches*.

2 - Temporización de circuitos secuenciales

- Tiempo entre cambios, tiempo de espera y tiempo en que se mantienen los datos.
- Determinación de la velocidad máxima de operación.

3 - Verificación de circuitos

- Verificación funcional y temporal.
- Construcción de tests.

Concesiones en el diseño de circuitos

Área

- El área utilizada es **proporcional al costo** del dispositivo, cuanta más área requerida, más costoso de construir es el circuito.

Speed/Througput

- Circuitos más rápidos y con más capacidad, implican **más componentes**, más complejidad, mayor consumo.

Power/Energy

- Buscamos circuitos que consuman energía de forma **eficiente** para la tarea a realizar.
- Dispositivos de alto rendimiento deben poder **disipar** mucha energía.
- Dispositivos móviles tienen una cantidad **limitada** de energía disponible.

Design Time

- Los diseños más complejos requieren más **tiempo de diseño**, repercutiendo en costos.
- Además el diseño debe considera el *time to market* del circuito como **producto**.

Tiempos en un circuito (*Circuit Timming*)

Timing

Consiste en el análisis de la temporización de las operaciones que se suceden dentro de un circuito digital.

Busca responder

¿Qué tan rápido puede ser mi circuito?
¿Puedo hacer correr aun más rápido mi circuito?
¿Qué sucede si supero el límite de mi circuito?

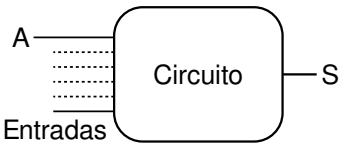
Un diseño puede ser logicamente correcto, pero puede fallar en una implementación por problemas fisicos de llevar este circuito a la vida real. **El *timing* es uno de estos problemas**.

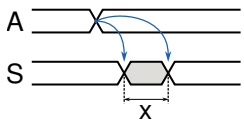
Modelo de los circuitos digitales

Podemos modelar problemas considerando que los cambios en las entradas de las compuertas se ven inmediatamente reflejados en la salida.
Esta es una muy conveniente abstracción, pero dista de modelar que sucede en el mundo físico.

En la vida real tenemos retardos (*delay*):

- Las salidas demoran tiempo en **exponer los cambios** de las entradas.
- Los transistores toman un **tiempo finito** en cambiar.
- Las señales demoran en **propagarse por los cables**.





x: Rango de tiempo en que esperamos recibir la respuesta. Desde que comienza a cambiar hasta que efectivamente cambia.

Variaciones del *delay* en un circuito

Las causas del *delay* son:

- La **capacitancia** y **resistencia** en el circuito.
- La **velocidad de la luz** (no es tan rápida en la escala de los nanosegundos).

Cualquier elemento que afecte estas causas afecta el *delay*

- Reloj, **activación por flanco** ascendente o descendente.
- Las **diferentes entradas** tienen diferentes *delays*.
- El circuito se resuelve por **diferentes caminos**.
- Cambios en el entorno, como la **temperatura**.
- **Edad** del circuito, cambia las capacidades y resistencias de los componentes.

Tenemos un gran rango de posibles retardos entre la entrada y la salida de cualquier circuito.
Queremos entonces obtener el máximo y el mínimo retardo para generar una salida.

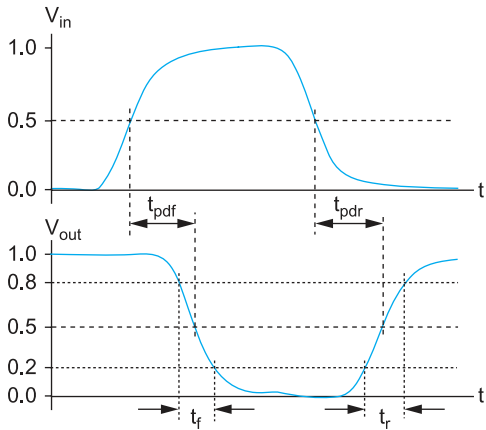
Definiciones

Propagation delay time
t_{pd} = máximo tiempo entre que la entrada cruzo el 50 % y la salida cruzo el 50 %.

Contamination delay time
t_{cd} = mínimo tiempo entre que la entrada cruzo el 50 % y la salida cruzo el 50 %.

Rise time
t_r = Tiempo que toma la señal entre subir de un 20 % a un 80 % del valor estable.

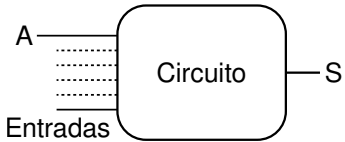
Fall time
t_f = Tiempo que toma la señal entre bajar de un 80 % a un 20 % del valor estable.

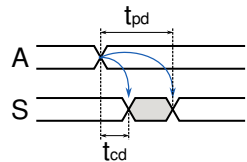


Físicamente las señales no son perfectas, respetan un modelo de comportamiento **no lineal**

Retardos entre el *Input* y el *Output*

En lo que sigue vamos a simplificar el modelo, considerando un comportamiento lineal.





- **Propagation delay** (t_{pd}): Retardo hasta que el cambio de S termina.
- **Contamination delay** (t_{cd}): Retardo desde que S comienza a cambiar.

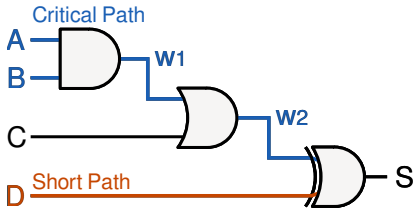
Vamos a tener que calcular los caminos en el circuito, tanto el más corto, como el más largo.

Calculo del mayor y menor camino

Nos interesa conocer el mayor (longest delay) y menor (shortest delay) retardo en el camino dentro del circuito.

- Critical Path: Camino más largo (t_{pd})
- Short Path: Camino más corto (t_{cd})

Ejemplo:



Critical Path

$$t_{pd} = t_{pd_{and}} + t_{pd_{or}} + t_{pd_{xor}}$$

Short Path

$$t_{cd} = t_{cd_{xor}}$$

El retardo de cada componente se toma de la **libreria de celdas** utilizada.

Ejemplo de *datasheet*

TOSHIBA TC74HC00AP/AF/AFN

TOSHIBA CMOS DIGITAL INTEGRATED CIRCUIT SILICON MONOLITHIC

TC74HC00AP, TC74HC00AF, TC74HC00AFN

QUAD 2 – INPUT NAND GATE

The TC74HC00A is a high speed CMOS 2-INPUT NAND GATE fabricated with silicon gate C²MOS technology.
It achieves the high speed operation similar to equivalent LSTTL while maintaining the CMOS low power dissipation.
The internal circuit is composed of 3 stages including buffer output, which provide high noise immunity and stable output.
All inputs are equipped with protection circuits against static discharge or transient excess voltage.

FEATURES :

- High Speed ····· t_{pd} = 6ns(typ.) at V_{CC} = 5V
- Low Power Dissipation ····· I_{CC} = 1 μA(Max.) at Ta = 25°C
- High Noise Immunity ····· $V_{NIH} = V_{NIL}$ = 28% V_{CC} (Min.)
- Output Drive Capability ····· 10 LSTTL Loads
- Symmetrical Output Impedance ··· $|I_{OH}| = I_{OL}$ = 4mA(Min.)
- Balanced Propagation Delays ···· $t_{pLH} = t_{pHL}$
- Wide Operating Voltage Range ···· V_{CC} (opr.) = 2V ~ 6V
- Pin and Function Compatible with 74LS00

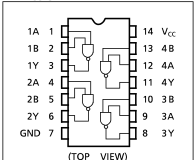
(Note) The JEDEC SOP (FN) is not available in Japan.

P (DIP14-P-300-2.54)
Weight : 0.96g (Typ.)

F (SOP14-P-300-1.27)
Weight : 0.18g (Typ.)

FN (SOL14-P-150-1.27)
Weight : 0.12g (Typ.)

PIN ASSIGNMENT



(TOP VIEW)

TOSHIBA TC74HC00AP/AF/AFN

AC ELECTRICAL CHARACTERISTICS (C_L = 15pF, V_{CC} = 5V, Ta = 25°C, Input $t_r = t_f$ = 6ns)

PARAMETER	SYMBOL	TEST CONDITION	MIN.	TYP.	MAX.	UNIT
Output Transition Time	t_{TLH} t_{THL}		—	4	8	ns
Propagation Delay Time	t_{pLH} t_{pHL}		—	6	12	

AC ELECTRICAL CHARACTERISTICS (C_L = 50pF, Input $t_r = t_f$ = 6ns)

PARAMETER	SYMBOL	TEST CONDITION	V_{CC}(V)	Ta = 25°C MIN.	TYP.	MAX.	Ta = −40−85°C MIN.	MAX.	UNIT
Output Transition Time	t_{TLH} t_{THL}		2.0	—	25	75	—	95	ns
			4.5	—	7	15	—	19	
			6.0	—	6	13	—	16	
Propagation Delay Time	t_{pLH} t_{pHL}		2.0	—	27	75	—	95	
			4.5	—	9	15	—	19	
			6.0	—	8	13	—	16	
Input Capacitance	C_{IN}			—	5	10	—	10	pF
Power Dissipation Capacitance	C_{PD} (1)			—	20	—	—	—	

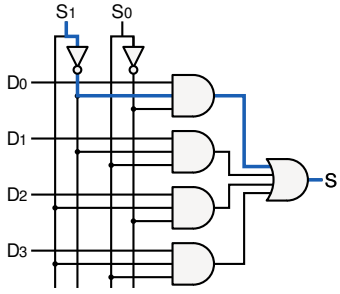
Note (1) C_{PD} is defined as the value of the internal equivalent capacitance which is calculated from the operating current consumption without load.
Average operating current can be obtained by the equation :

$$I_{CC}(opr) = C_{PD} \cdot V_{CC} \cdot f_{IN} + I_{CC} / 4 \text{ (per Gate)}$$

Ejemplo: Implementaciones de un multiplexor

Supongamos las siguientes implementaciones de un multiplexor y sus posibles delays.

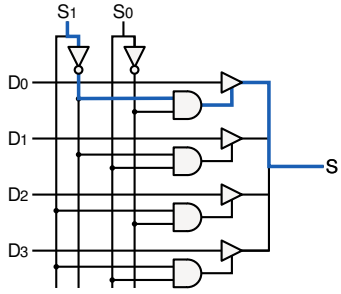
Implementación 1



$t_{pd} = t_{pd_{not}} + t_{pd_{and3}} + t_{pd_{or4}}$

$t_{cd} = t_{cd_{and3}} + t_{cd_{or4}}$

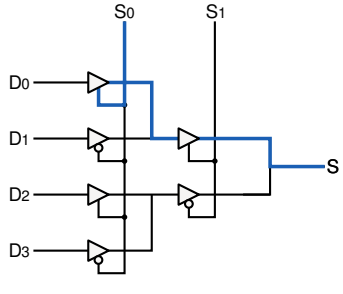
Implementación 2



$t_{pd} = t_{pd_{not}} + t_{pd_{and2}} + t_{pd_{buffSel}}$

$t_{cd} = t_{cd_{buff}}$

Implementación 3



$t_{pd} = t_{pd_{buffSel}} + t_{pd_{buff}}$

$t_{cd} = t_{cd_{buff}} + t_{cd_{buff}}$

Estamos ignorando los **delay de los cables**, pero para altas frecuencias los deberíamos considerar.

Ejemplo: Implementaciones de un multiplexor

Analizando las implementaciones de un multiplexor y sus posibles delays obtenemos:

Implementación 1

$t_{pd} = t_{pd_{not}} + t_{pd_{and3}} + t_{pd_{or4}}$

$t_{pd} = 30 + 80 + 90 = 200$ ns

Implementación 2

$t_{pd} = t_{pd_{not}} + t_{pd_{and2}} + t_{pd_{buffSel}}$

$t_{pd} = 30 + 60 + 50 = 140$ ns

Implementación 3

$t_{pd} = t_{pd_{buffSel}} + t_{pd_{buff}}$

$t_{pd} = 50 + 40 = 90$ ns

Suponiendo los siguientes retardos para las compuertas.

Compuerta	t_{pd}
not	30 ns
and2	60 ns
and3	80 ns
or4	90 ns
buffSel	50 ns
buff	40 ns
buffNegSel	40 ns
buffNeg	30 ns

Calcular *long and short paths*

No es simple determinar los caminos de máximos y mínimos retardos.

- No todas las **entradas afectan las salidas**.
- Pueden existir **múltiples caminos** desde una entrada a una salida.

No todos los circuitos son construidos de la misma forma.

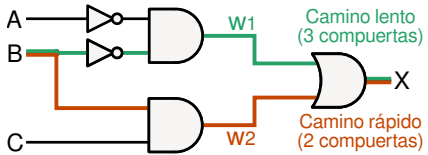
- Diferentes instancias de la misma compuerta pueden tener **diferentes retardos**.
- Los **cables no tiene retardo cero**, el retardo es proporcional a su longitud.
- La **temperatura** y el **voltaje** afecta la velocidad de los circuitos
  - No todos los elementos en el circuito se afectan de la misma forma.
  - Incluso el *critical path* puede variar dependiendo de estos factores.

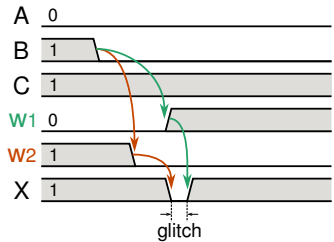
Los diseñadores asumen las **peores condiciones** y terminan ejecutando muchas simulaciones para balancear una solución **conservadora** vs de **alta performance**.

Glitches en las salidas

glitch: Un cambio en una entrada produce múltiples transiciones a la salida.

Ejemplo: Cambio en B





¡Aparecen dos caminos, uno rápido y otro lento!

Los *glitches* no son importantes si se diseñan correctamente los circuitos.

Excepto que se trabaje a frecuencias muy altas.

El problema se manifiesta cuando se toma una lectura del estado del circuito en el momento de glitch. Mientras estemos seguros de diseñar el circuito y que eso nunca suceda, entonces el *glitch* no debería ser un problema.

Como evitar los *glitches*

La idea básica es utilizar **circuitos redundantes** que impidan que se generen los *glitches*, logrando evitar que el camino más rápido haga cambiar la salida antes de tiempo.

Sin embargo, los circuitos redundantes:

- Aumentan los **costos** de diseño.
- Aumentan el **área**.
- Requieren más **energía**.
- Además **no (necesariamente) mejoran** el desempeño del circuito.

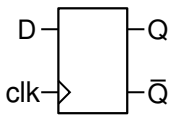
En general el diseñador debe decidir si ignorar los *glitches*, y los ignora.

En cualquier caso es necesario tenerlos en cuenta en las **simulaciones**, para entender que pueden suceder y que no representan un problema.

Timming en circuitos secuenciales

Para analizar la temporización de circuitos secuenciales, vamos a tomar uno de los elementos que nos permite **almacenar datos**.

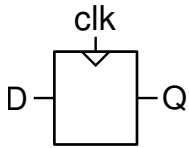
Un Flip-flop D es circuito secuencial básico.
Guarda el valor de un bit leyendolo de su entrada D y exponiendolo en su salida Q.
Su activación es por el cambio de flanco en la señal de reloj.

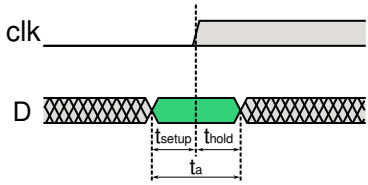


clk	D	Q_n	Q_{n+1}
1	0	Q_n	0
1	1	Q_n	1
0	×	Q_n	Q_n

Tiempos en un Flip-flop D (*input*)

La entrada D debe ser **estable** en el momento que es leída para ser copiada en el estado del circuito (activación por flanco).





Setup time (t_{setup}):

Tiempo antes del cambio de flanco, donde el dato de entrada debe ser estable.

Hold time (t_{hold}):

Tiempo luego del cambio de flanco, donde el dato debe ser estable.

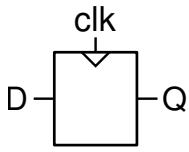
Aperture time (t_a):

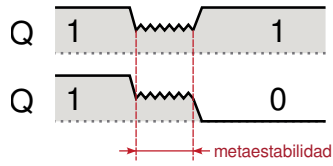
Tiempo durante el cambio de flanco, donde el dato debe ser estable ($t_{setup} + t_{hold}$).

Metaestabilidad: ¿Qué sucede si no respetamos los tiempos?

Si se violan los tiempos mínimos: **La entrada D cambia durante el tiempo de apertura.**

Se genera *metaestability*.





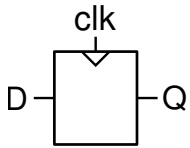
La salida Q toma un valor entre 0 y 1 durante un tiempo antes de tomar el valor definitivo.

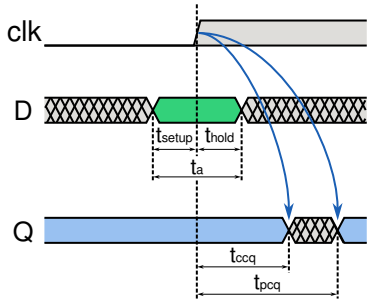
El resultado será no determinístico. Puede quedar tanto en 0 como en 1.

La temporización de los circuitos se diseña para
jamas entrar en un escenario de metaestabilidad

Tiempos en un Flip-flop D (*output*)

La salida Q tomará su **valor final** luego de un tiempo dado, y comenzará a variar desde un tiempo antes.





Contamination delay clock-to-q (t_{ccq}):

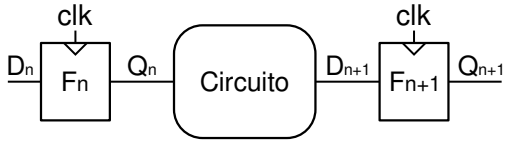
Tiempo entre el cambio de flanco y que Q comienza a cambiar (inestable).

Propagacion delay clock-to-q (t_{pcq}):

Tiempo entre el cambio de flanco y que Q deja de cambiar (estable).

Temporización de sistemas secuenciales

Un sistema secuencial se puede ver como **múltiples flip-flops interconectados** mediante circuitos combinatorios puros.



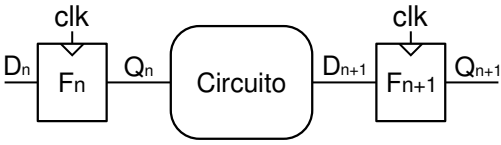
Para asegurar el correcto funcionamiento debemos verificar que se cumplen los **requerimientos de tiempo**, tanto para F_n como para F_{n+1}.

Para asegurar que F_{n+1} tome el valor correcto:
D_{n+1}, debe ser **estable** durante el tiempo de apertura de F_{n+1}.

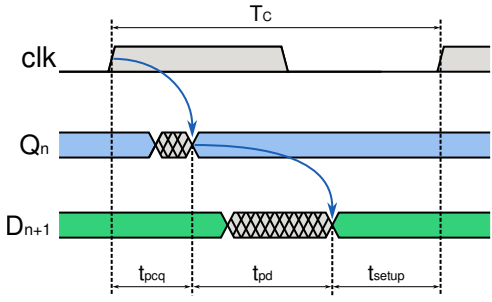
- Si la lógica combinacional es **muy rápida**, entonces el ***hold time* no se cumple**.
- Si la lógica combinacional es **muy lenta**, entonces el ***setup time* no se cumple**.

Limitaciones del tiempo de *setup*

Para asegurar el tiempo se debe **limitar el retardo máximo** entre F_n y F_{n+1}.
La entrada de F_{n+1} debe ser **estable** al menos t_{setup} antes del cambio de clock.



$$T_c > t_{pcq} + t_{pd} + t_{setup}$$

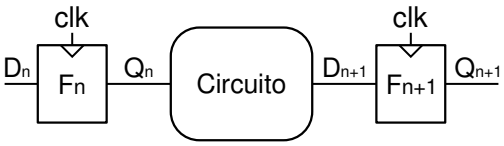


Donde,

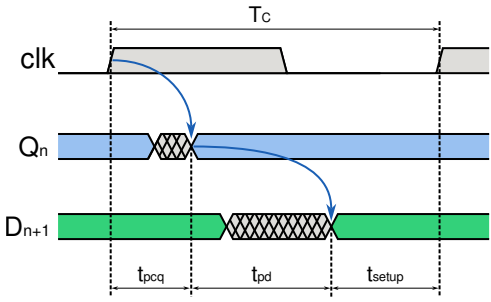
- T_c → Tiempo del ciclo de reloj.
- t_{pd} → Tiempo útil del circuito.
- t_{pcq} → Tiempo perdido para escribir el próximo dato.
- t_{setup} → Tiempo perdido para leer el próximo dato.

El tiempo desperdiciado en garantizar la secuencialidad de los eventos se denomina **Secuencial Overhead**.
($t_{pcq} + t_{setup}$)

Limitaciones del tiempo de *setup*



$$T_c > t_{pcq} + t_{pd} + t_{setup}$$

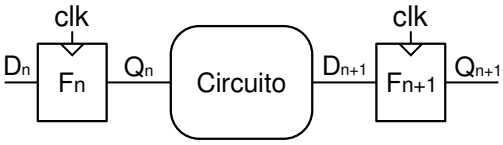


El rendimiento del diseño está determinado por el *critical path* (t_{pd}).
Este permite determinar el mínimo *clock period*, o **frecuencia máxima** de operación.

- Si el *critical path* es muy **grande**, el diseño resulta **muy lento**.
- Si el *critical path* es muy **chico**, en cada ciclo estamos haciendo **poco trabajo útil** (la mayor parte del ciclo se pierde en el secuencial overhead).

Limitaciones del tiempo de *hold*

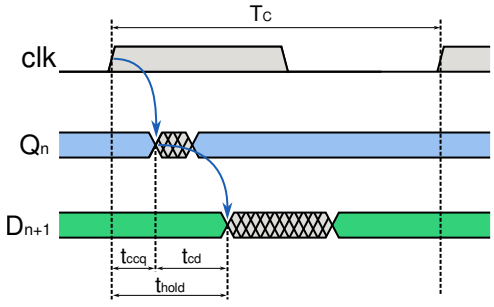
Para asegurar el tiempo se debe **limitar el retardo mínimo** entre F_n y F_{n+1}
D_{n+1} debe ser **estable** al menos t_{hold} luego del cambio de clok.



$$t_{ccq} + t_{cd} > t_{hold}$$

o

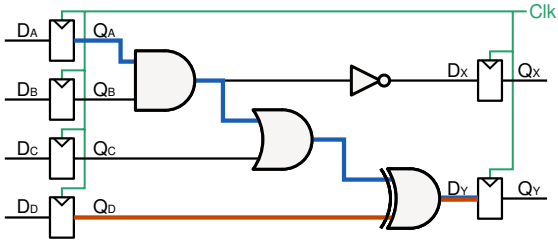
$$t_{cd} > t_{hold} - t_{ccq}$$



El *contamination delay* (t_{cd}) del circuito combinacional no puede ser cero,
debe tener un mínimo de tiempo de al menos $t_{hold} - t_{ccq}$.

No depende del reloj (T_c), depende del diseño del circuito.

Ejemplo de análisis de tiempo



Valor	Tiempo
t_{ccq}	30 ps
t_{pcq}	50 ps
t_{setup}	60 ps
t_{hold}	70 ps
t_{pd}	35 ps (all gates)
t_{cd}	25 ps (all gates)

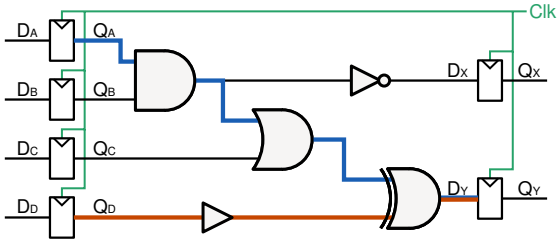
$t_{pd} = 35 + 35 + 35\ \text{ps} = 105\ \text{ps}$
$t_{cd} = 25\ \text{ps}$

Validación *setup time*:
$T_c > t_{pcq} + t_{pd} + t_{setup}$
$T_c > 50 + 105 + 60$
$T_c > 215$
$\text{Frec}_{max} = 1/T_c = 4{,}65\ \text{GHz}$

Validación *hold time*:
$t_{ccq} + t_{cd} > t_{hold}$
$35 + 25 > 70$
$55 > 70$
¡No verifica!

Debemos alterar el circuito para que el camino mínimo sea más largo.

Ejemplo de análisis de tiempo



Valor	Tiempo
t_{ccq}	30 ps
t_{pcq}	50 ps
t_{setup}	60 ps
t_{hold}	70 ps
t_{pd}	35 ps (all gates)
t_{cd}	25 ps (all gates)

$t_{pd} = 35 + 35 + 35\ \text{ps} = 105\ \text{ps}$
$t_{cd} = 25 + 25\ \text{ps} = 50\ \text{ps}$

Validación *setup time*:

$T_c > t_{pcq} + t_{pd} + t_{setup}$
$T_c > 50 + 105 + 60$
$T_c > 215$
$\text{Frec}_{max} = 1/T_c = 4{,}65\ \text{GHz}$

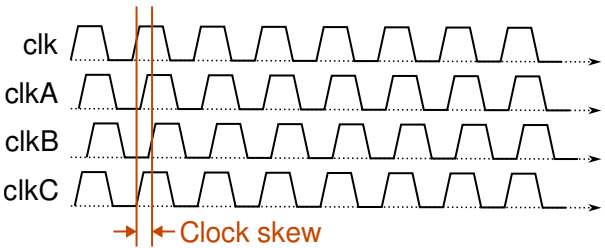
Validación *hold time*:

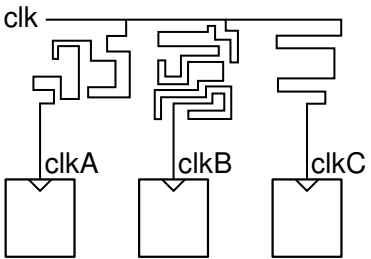
$t_{ccq} + t_{cd} > t_{hold}$
$35 + 50 > 70$
$85 > 70$
¡Verifica!

Clock Skew: El reloj también tiene retardo

La señal de reloj no llega a todos los rincones del circuito al mismo tiempo.
No todos los componentes secuenciales ven el reloj en el **mismo momento.**

Clock Skew: es la diferencia entre dos cambios de flanco del mismo reloj en lugares diferentes.



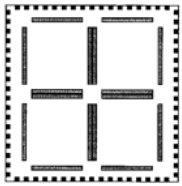


Este problema hace el calculo aun más complejo,
el reloj puede llegar antes o despues que el reloj de la próxima etapa.

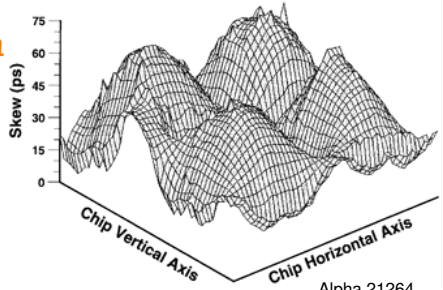
Clock Skew: Ejemplo procesador DEC Alpha

Alpha 21064
(1993): 200MHz

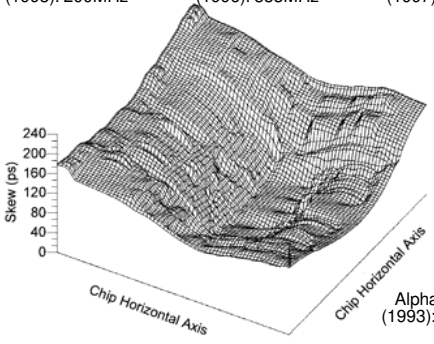
Alpha 21164
(1996): 333MHz



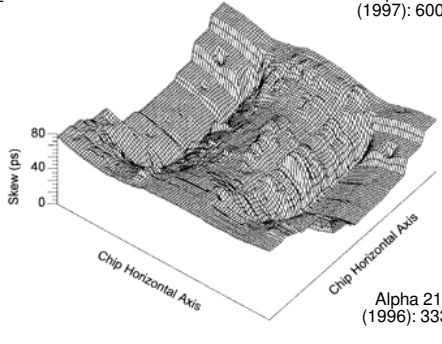
Alpha 21264
(1997): 600MHz



Alpha 21264
(1997): 600MHz



Alpha 21064
(1993): 200MHz

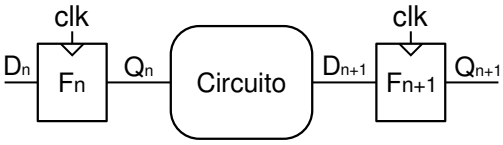


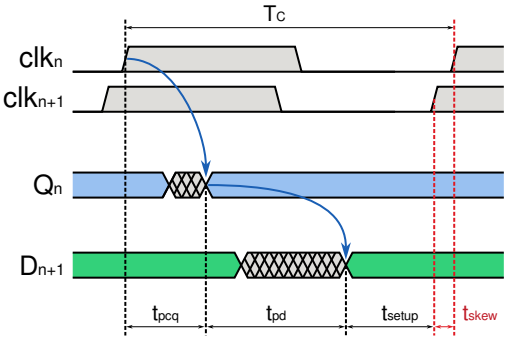
Alpha 21164
(1996): 333MHz

"High-Performance Microprocessor Design" P. E. Gronowski, et. al. IEEE JOURNAL OF SOLID-STATE CIRCUITS, VOL. 33, NO. 5, MAY 1998

Clock Skew: Limitaciones del tiempo de *setup*

Supongamos el peor caso para el *Clock Skew*. **La señal de reloj llega a F_{n+1} antes que a F_n.**
Esto reduce el tiempo que disponemos para resolver el circuito combinatorio.





$$T_c > t_{pcq} + t_{pd} + t_{setup} + t_{skew}$$

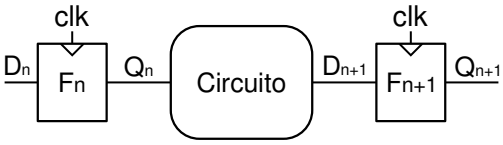
Luego,

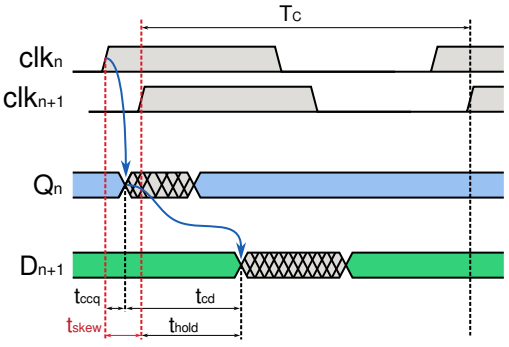
$$t_{pd} < T_c - (t_{pcq} + t_{setup} + t_{skew})$$

Considerando *skew time*, esté se materializa como un aumento en el *setup time*.
Limita aun más el tiempo de propagación máximo del circuito combinatorio (t_{pd}).

Clock Skew: Limitaciones del tiempo de *hold*

Supongamos el peor caso para el *Clock Skew*. **La señal de reloj llega a F_{n+1} despues que a F_n.**
Esto incrementa el tiempo mínimo que debe tener como retardo el circuito combinatorio.





$$t_{cd} + t_{ccq} > t_{hold} + t_{skew}$$

Luego,

$$t_{cd} > t_{hold} + t_{skew} - t_{ccq}$$

Considerando *skew time*, esté se materializa como un aumento en el *hold time*.
Limita aun más el tiempo mínimo del circuito combinatorio (t_{cd}).

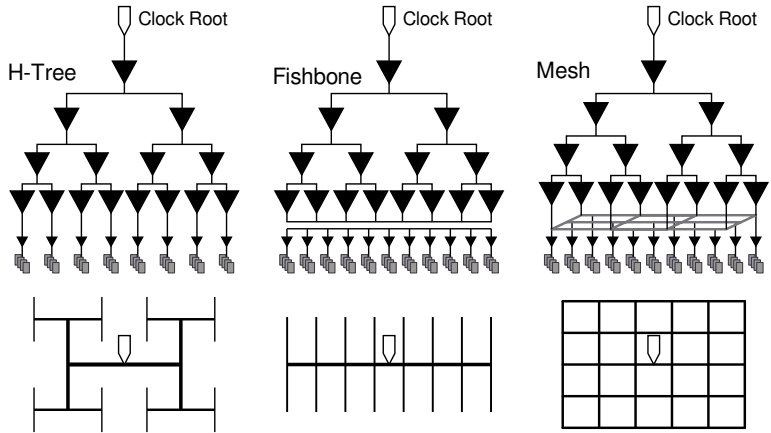
Clock Skew

El *Clock Skew* incrementa los tiempos de t_{setup} y t_{hold}.
Aumenta el **secuencial overhead**, dejando menos tiempo para el trabajo útil por ciclo de reloj.

Se buscan reducir al mínimo el *Clock Skew*.

Diseñando inteligentes formas de distribuir el reloj a lo largo del circuito.

Pre-drivers, *Clock-tree*, *Cross-links*, *Mesh*, *Sinks*, son algunas de las técnicas que se utilizan.



Verificación de circuitos

¿Cómo sabemos que un circuito funciona correctamente?

Incluso si es logicamente correcto, puede que el hardware donde lo vamos a implementar no pueda respetar todas las limitaciones de tiempos del diseño.

Tipos de comprobaciones:

- Testeo **funcional**: Respeta el comportamiento descripto por el HDL.
- Testeo **temporal**: Respeta las limitaciones de tiempo del hardware.

Herramientas de simulación:

- **Verificación formal**: Modelo matemático (ej. SAT solvers).
- **Simulación desde HDL**: Simulación de alto nivel (ej. Vivado).
- **Simulación del circuito**: Simulación de bajo nivel (ej. SPICE).

Verificación de circuitos: El problema de los diseños muy grandes

La verificación del diseño puede llevar mucho tiempo.

- Todos los **caminos lógicos** del circuito son funcionalmente correctos.
- Todos los **elementos del circuito** se usan dentro de los límites de tiempo y energía.

La comprobación entonces se separa en dos responsabilidades:

1. **Chequeo solo de la funcionalidad a alto nivel, sobre el circuito desde el HDL.**
   - Más rápido que a nivel de circuitos, permite **cubrir mayor cantidad de posibilidades**.
   - **Fácil de escribir** y correr comprobaciones.
2. **Chequeo solo de tiempo, energía y límites a bajo nivel, sobre el circuito.**
   - **No se realiza un chequeo funcional** del modelo a bajo nivel.
   - Se chequea un **modelo funcional equivalente** al de alto nivel, que resulta más simple que comprobar el funcionamiento completo a este nivel.

La simulación del circuito (bajo nivel) es más lenta que la simulación desde HDL (alto nivel).

Verificación de circuitos: El problema de los diseños muy grandes

Tenemos herramientas para manejar **diferentes niveles de verificación**.

- **Logical Syntesis tools**: Comprobar equivalencia entre la lógica de alto nivel y la sintetización a nivel de descripción del circuito.
- **Timming verification tools**: Comprobar la temporización del circuito en base a una tecnología de implementación.
- **Design rule checks**: Comprobar que la implementación del circuito sea físicamente realizable.

El objetivo del diseñador de la lógica del circuito es:

- Proveer **test funcionales** para comprobar la correctitud del diseño.
- Proveer **timing contraints**, por ejemplo, la deseable frecuencia de operación.

Los ingenieros de circuitos usando estas herramientas deciden si el diseño se puede construir.

Verificación Funcional

Consiste en un chequeo *booleano*. Se busca comprobar la correctitud encontrando un contraejemplo para su correcto funcionamiento.

Objetivo: Chequeo de la correctitud lógica del diseño

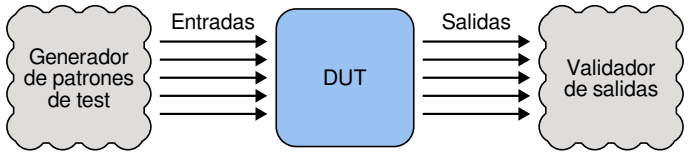
En este caso la **temporización es ignorada**.
Sin embargo, se comprueban chequeos simples para encontrar problemas básicos.

Dos enfoques principales:

- Simulación lógica (*rutinas de test*). → ¡esto es lo que vamos a hacer!
- Técnicas de verificación formal.

Verificación funcional basada en *Testbench*

Un *Testbench* es un módulo de Verilog creado específicamente para testear un diseño.
Usualmente llamado "device under test (DUT)"



Un *Testbench* provee **entradas** (patrones) al DUT.

- Generados manualmente.
- Generados automáticamente (secuencial o aleatorios).

Un *Testbench* chequea las **salidas** del DUT contra las correctas.

- Validadas manualmente.
- Validadas contra un modelo libre de errores ("*golden design*").

Verificación funcional basada en Testbench

Un *Testbench* puede ser:

- **Código HDL** escrito para testear otros modulos HDL.
- **Circuito esquemático** usado para testear otros diseños.

Un *Testbench* no esta diseñado para sintetizar en hardware:

- Ejecuta **solo en una simulación**.
  - Simulación HDL (Vivado Simulator).
  - Simulación del circuito (SIPICE).
- Usa herramientas **válidas solo dentro de la simulación**.
  - Marcas de espera por un tiempo fijo.
  - Indicaciones de voltajes y corrientes ideales.
  - No considera la construcción física del circuito.

Tipos de Testbench

Testbench	Generación de entradas y salidas	Chequeo de errores
Simple	Manual	Manual
Auto-Chequeo	Manual	Automático
Automático	Automático	Automático

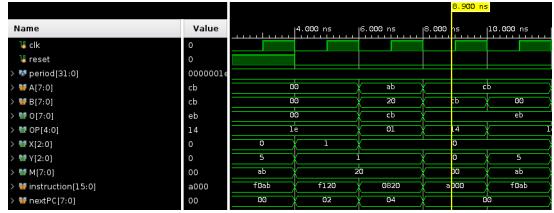
Ejemplo: Testbench Simple

```
module testbench_Simple();
    reg a, b, c;
    wire y;

    fun dut(.a(a), .b(b), .c(c), .y(y));

    initial begin
        a = 0; b = 0; c = 0;
        #10;
        c = 1;
        #10;
        b = 1; c = 0;
        #10
        a = 1; b = 0; c = 0;
        #10;
        ...
    end
endmodule
```

Reproduce un conjunto de señales que se envían al DUT cada un tiempo determinado.

La simulación registra el cambio de las señales. Idealmente se utilizan diagramas de señales para visualizar los cambios.

Para circuitos grandes, con muchas señales está solución no escala.



Testbench Simple

Pro:

- Simple de diseñar y hacer.
- Fácil testear, pocas entradas y casos borde.
- Útil para aprender.

Contras:

- No escala para circuitos grandes.
- Las salidas deben ser chequeadas **manualmente** fuera de la simulación.
  - Mirando las señales una por una.
  - Imprimiendo en un *log* de salida.

Ejemplo: Testbench Auto-Chequeo

```
module testbench_AutoChequeo();
    reg a, b, c;
    wire y;

    fun dut(.a(a), .b(b), .c(c), .y(y));

    initial begin
        a = 0; b = 0; c = 0;
        #10;
        if( y !== 1) $display("error 000");
        c = 1;
        #10;
        if( y !== 0) $display("error 001");
        b = 1; c = 0;
        #10
        if( y !== 0) $display("error 010");
        a = 1; b = 0; c = 0;
        #10;
        if( y !== 1) $display("error 100");
        ...
    end
endmodule
```

Agregando muy poca lógica es posible hacer chequeos en el código de test.

Los chequeos consisten en comparaciones con estados conocidos.

Para circuitos grandes, con muchos estados es muy complicado generar todos los casos de prueba y sus soluciones esperadas.

Testbench Auto-Chequeo

Pro:

- Simple de diseñar y hacer.
- Fácil testear pocas entradas y casos borde.
- El simulador indica cuando un error ocurre.

Contras:

- No escala para circuitos grandes.
- Es fácil cometer un error sobre los valores fijados como válidos.
- El proceso de *debug* es difícil, ya que se debe identificar si el error está en el test o en el DUT.

Testbench Auto-Chequeo usando `testvectors`

La idea es tener un archivo con todas las entradas y salidas a verificar.
El archivo se puede crear de forma manual o automática
a partir de un modelo de funcionamiento verificado.

Ejemplo: `input_output`

```
000 _1
001 _0
010 _1
011 _0
100 _1
101 _0
110 _1
111 _0
```

Usando una nueva señal de reloj para asignar entradas y leer salidas.

Ejecutamos una a una las entradas por cada ciclo de nuestro nuevo reloj.

- La señal sirve para separar las entradas de la lectura de salidas.
- Permite observar en los diagramas de señales las entradas y salidas.

Ejemplo: Testbench Auto-Chequeo usando `testvectors`

```
1  module testbench_AutoChequeo_TestVectors () ;
2      reg clk , reset ;
3      reg a, b, c;
4      wire y;
5      reg [31:0] vectornum, errors ;
6      reg [31:0] testvector [0:1000];
7
8       circuit dut (.a(a), .b(b), .c(c), .y(y));
9
10     always @(*)
11         clk = 1; #5; clk = 0; #5; // periodo 10ns
12
13     initial
14     begin
15         $readmemb("data.mem", testvector ) ;
16         vectornum = 0;
17          errors = 0;
18         reset = 1; #25; reset = 0;
19     end
20
21     always @(posedge clk)
22         {a, b, c, yexpected} = testvector [vectornum];
```

Línea 8: Declaración del DUT.
Línea 11: Definición del clock.
Línea 15: Lectura del archivo de `testvectors`.
Línea 22: Asignación de entradas al DUT.

```
24     always @(negedge clk)
25     begin
26         if (~ reset )
27         begin
28             if (y !== yexpected)
29             begin
30                 $display (" Error : inputs = %b", {a,b,c});
31                 $display ("          outputs = %b (expected = %b)", y, yexpected) ;
32                 errors = errors + 1;
33         end
34         vectornum = vectornum + 1;
35         if ( testvector [vectornum] === 4'bx)
36         begin
37             $display (" %d test completed with %d errors", vectornum, errors ) ;
38             $finish ;
39         end
40     end
41 endmodule
```

Línea 26: Si `reset` está en cero comienzo (línea 18).
Línea 28: Si el resultado no es el esperado se presenta el error.
Línea 34: Incremento el indice en el vector.
Línea 35: Si no tengo más datos en el vector, termino.

Testbench Auto-Chequeo usando `testvectors`

Pro:

- Simple de diseñar y hacer.
- Fácil testear pocas entradas y casos borde.
- El simulador indica cuando un error ocurre.
- No es necesario cambiar los valores fijos sobre diferentes tests.

Contras:

- Más escalable que hacer la tarea manual, pero aun así limitado a la lectura del archivo y construcción del mismo.
- Dependiendo como se construye el testvector, si es manual, se está expuesto a cometer errores fácilmente.

Golden Models

Un *Golden Model* representa el comportamiento del **circuito ideal**.

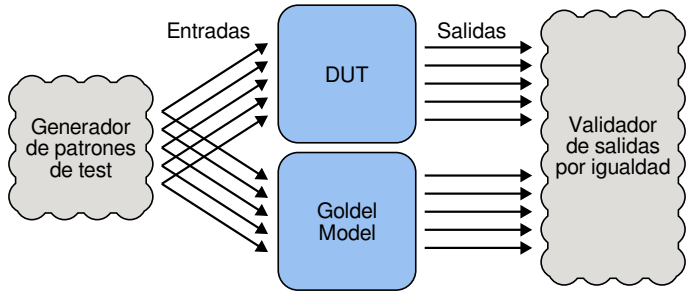
- Es difícil de desarrollar, ya que depende del **nivel de precisión del modelo**.
- Puede ser escrito en **diferentes lenguajes**, desde C hasta `Python` incluso en `verilog`.

El *Golden Model* debe ser:

- Más simple que un diseño a nivel de compuertas.
- Simple de entender, extender y modificar.
- Simple de verificar, comportandose de forma similar al DUT.

Testbench automático

En este *Testbench* la salida del DUP es **comparada contra el *golden model***.



Implica diseñar métodos para construir casos de entrada.

Es esperable que los casos cubran todo el espacio de entradas.
Sin embargo, esto no es posible en el caso general.

Testbench automático

Pro:

- La salida es chequeada automáticamente.
- Puede ser comparado incluso el *timming* contra el *golden model*.
- Es altamente escalable, mientras que la simulación se realice en un tiempo prudente.
  - Esto se contrapone a cubrir todo el espacio de entradas.
- Mejor separación de roles.
  - Equipos separados pueden trabajar en el DUT y en el *golden model*.
  - Se puede hacer foco en casos de test importantes y no en las salidas a chequear.

Contras:

- Crear un *golden model* puede ser muy difícil.
- Crear buenos casos de entrada puede ser muy difícil también.

Timing Verification

Simulación de alto nivel (Verilog)

- Es posible modelar tiempos usando las sentencias #x en el DUT.
- Se utiliza la jerarquía del modelo:
  - Se inserta delays en los FF, compuertas básicas, memorias, etc.
  - Se inserta delays en el diseño de alto nivel, componentes.
- No es preciso como la temporización a nivel del circuito.

Simulación a nivel del circuito (timming verification)

- Se requiere sintetizar el diseño a un circuito.
- No hay un camino general, depende del *workflow* de diseño específico.
- Utiliza la herramienta de la tecnología particular, ya sea FPGA/ASIC/etc.
  - Ejemplo: Xilinx Vivado.
  - Ejemplo: Synopsys/Cadence Tools (para VLSI *(very large-scale integration)*).

Buenas practicas de diseño

Ciclo de Reloj
El ciclo de reloj es determinado por el retardo máximo del componente combinatorio que podamos colocar sin violar ninguna limitación temporal.

Critical path
Minimizar el máximo retardo lógico implica maximizar el rendimiento.

Diseño Balanceado
Balancear el retardo lógico entre las distintas partes del sistema (entre diferentes componentes entre flip-flops). Evitar cuellos de botella y minimizar el tiempo perdido.

Optimizar el caso común
Pensar en optimizar para el caso más común de uso, pero estar seguro que los casos no comunes, no fuerzan el diseño. Evitarlos y hacer foco en casos realistas.

Bibliografía

- **“Digital Design and Computer Architecture”**, Second Edition
  David Money Harris, Sarah L. Harris - Morgan Kaufmann - 2013
  - Chapter 2 - Combinational Logic Design → 2.9 - Timing - Pag. 88-95
  - Chapter 3 - Sequential Logic Design → 3.5 - Timing of Sequential Logic - Pag. 141-161
  - Chapter 4 - Hardware Description Languages → 4.9 Testbenches - Pag. 220-225
- **“CMOS VLSI Design: A Circuits and Systems Perspective”**, Fourth Edition.
  Neil H. E. Weste, David Money Harris - Pearson - 2011
  - Chapter 10 - Sequential Circuit Design 10.2 →Sequencing Static Circuits - Pag. 376-391

¡Gracias!