UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

FLUJO DE DISEÑO DE CIRCUITOS INTEGRADOS DIGITALES APLICADO AL DESARROLLO DE UN CONTROLADOR USB 2.0

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELECTRICISTA

DANIEL ALFONSO DÍAZ PÉREZ

PROFESOR GUÍA:
VÍCTOR GRIMBLATT HINZPETER

MIEMBROS DE LA COMISIÓN:
HÉCTOR AGUSTO ALEGRÍA
NICOLÁS BELTRÁN MATURANA

SANTIAGO DE CHILE
MAYO 2010

RESUMEN DE LA MEMORIA
PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELECTRICISTA
POR: DANIEL ALFONSO DÍAZ PÉREZ
FECHA: 7 DE MAYO DE 2010
PROF. GUÍA: SR. VÍCTOR GRIMBLATT H.

"FLUJO DE DISEÑO DE CIRCUITOS INTEGRADOS DIGITALES APLICADO AL DESARROLLO DE UN CONTROLADOR USB 2.0"

El presente trabajo de título tiene como objetivo principal la exposición del flujo de procesos involucrado en el diseño de un circuito integrado digital. Como consecuencia de ésto, se espera poder demostrar y motivar a explorar esta área en Chile, partiendo por el entorno más cercano: el Departamento de Ingeniería Eléctrica de la Facultad de Ciencias Físicas y Matemáticas de la Universidad de Chile.

Para conseguir este objetivo, se diseña un controlador USB 2.0 desde la asimilación de su protocolo para el modelamiento de una solución, hasta el esquemático de la disposición física del circuito resultante. El flujo de diseño consiste principalmente en: a partir de una idea, generada por una necesidad del mercado o bien por una razón académica, determinar sus especificaciones eléctricas y dividir sus funcionalidades; caracterizar el circuito utilizando un lenguaje de descripción de *hardware*, para esta memoria, Verilog HDL; traducir la descripción del circuito a una lista de compuertas lógicas reales interconectadas; y finalmente, utilizando la representación física de cada compuerta y conexiones, producir un plano físico del circuito. El desarrollo del controlador es guiado y facilitado por las herramientas de diseño de Synopsys®.

El resultado final es un informe que reúne los principales conceptos comprometidos en cada etapa del diseño, junto con su aplicación al modelamiento de un controlador USB 2.0. Aunque el uso de las herramientas de diseño no es detallado, los pasos realizados para cada proceso están basados en los flujos de diseño específicos de cada una de ellas. Respecto a lo anterior, se entrega como resultado un CD que contiene los módulos Verilog del controlador, la lista de compuertas interconectadas y archivos *guión* que contienen las principales instrucciones ejecutadas en las herramientas de Synopsys®.

USB es un protocolo de comunicación de mediana complejidad que requiere de un equipo de diseñadores para desarrollarlo con profundidad en todos sus aspectos. A su vez, el flujo de diseño de circuitos integrados posee una gran cantidad de conceptos que deben ser manejados para realizar un trabajo de nivel profesional, probablemente con especialistas en cada etapa. Se concluye que el diseño de circuitos integrados, sin considerar la fabricación, es posible de realizar completamente en Chile, considerando la existencia de las herramientas adecuadas y profesionales especializados o con interés en hacerlo.

A mi familia que me entregó las herramientas, y a aquéllos que se arriesgan por nuevos horizontes.

Agradecimientos

Agradezco a mi familia, que siempre me ha entregado todo el cariño para superar las adversidades de la vida y por darme las herramientas y oportunidades para alcanzar cada unos de mis objetivos. Sin todo su apoyo jamás habría logrado estar donde estoy, terminando mi carrera luego de muchos años de felicidad junto a ellos.

Agradezco también al profesor Victor Grimblatt, a Jorge Ramírez y a Dan Landau por guiarme en el trabajo de esta memoria, que prácticamente es un mundo nuevo para mi, de conceptos, metodologías y herramientas de software.

Finalmente quiero agradecer a todos mis compañeros de eléctrica y primer año que, a veces sin saberlo, me ayudaron a salir de problemas con los que me encontré durante mi vida universitaria.

Índice General

Índice de Figuras

Índice de Tablas

Capítulo 1

Presentación

1.1. Introducción

El presente documento corresponde a la memoria del Trabajo de Título realizado por el autor para optar al título de Ingeniero Civil Electricista en la Universidad de Chile. El principal objetivo es mostrar, a través del desarrollo de un controlador USB 2.0, el flujo de procesos involucrados en el diseño de un circuito integrado digital. Como consecuencia de ésto se espera motivar y demostrar la factibilidad del desarrollo de circuitos integrados en Chile.

Un circuito integrado es un trozo delgado de, generalmente, silicio que tiene un pequeño circuito electrónico grabado en su superficie [1]. En sus inicios, los sistemas electrónicos estaban construidos con circuitos compuestos de elementos discretos, lo cual, para sistemas más complejos resultaba en dispositivos de enorme tamaño y consumo de potencia. Un circuito integrado puede contener millones de elementos microscópicos (de hecho, nanométricos), incluyendo transistores, resistencias, condensadores y conductores, con un bajo consumo de potencia.

El avance tecnológico en la fabricación de circuitos integrados ha permitido duplicar la cantidad de transistores en un mismo trozo de silicio cada dos años[1], lo cual se puede traducir en que una misma funcionalidad se implemente en un dispositivo más pequeño, en que se puede introducir más funcionalidades en un mismo espacio y en que los costos disminuyen en términos de capacidad por unidad de área del circuito. Como consecuencia de la disminución de los tamaños en la fabricación de circuitos integrados, los métodos y herramientas para su diseño se han visto obligados a evolucionar también. El proceso de diseño desarrollado en esta memoria se basa en los mostrados en [3] y [4], pero consistente con las herramientas de diseño de Synopsys®.

El proceso de desarrollo de circuitos integrados actual permite separar las etapas de

[1] Ley de Moore [2].

diseño y fabricación, de manera que la primera puede ser realizada sin la necesidad de invertir en infraestructura ni tecnología especializada para su manufactura. Los requerimientos básicos serían computadores lo suficientemente equipados para correr las herramientas de diseño y mantener, al menos, un diseño de tamaño medio en memoria. De ésto se desprende que el diseño puede ser realizado en Chile, en función de los requerimientos que el mercado pueda presentar.

Lo anterior genera interés a crear en un nuevo espacio en el mercado chileno dedicado al diseño de circuitos integrados. La CORFO[1], en el marco del proyecto *InnovaChile*, indica que una de las razones para innovar es que ésto "abre nuevos mercados, pues no sólo permite mejorar la competitividad y expandir negocios actuales de una empresa, sino que además potencia el desarrollo de nuevos mercados, creando nuevos negocios diferentes de los actuales e incluso generando mercados antes inexistentes, a través de nuevos bienes o servicios" [5]. Además, señala que [6]:

$$Innovación = conocimiento + acción$$

El *conocimiento* existe, pero requiere experiencia y profundización. El interés en la materia en cuestión es difícil de cuantificar, pero aún con interés, falta *acción* para llevar acabo la innovación. Por su parte, la financiación existe[2], y en particular el "Consejo Nacional de Innovación para la Competitividad" se propuso como estrategia nacional la meta central de duplicar hacia el año 2021 el ingreso per cápita[3], pasando del lugar 45 al 27 en el factor de innovación dentro del índice de competitividad en los principales países del mundo [7].

Como se mencionó al inicio, esta memoria busca acercar el *conocimiento* y motivar la *acción*.

1.2. Objetivos

Los objetivos de este trabajo de título se presentan a continuación:

1.2.1. Objetivos Generales

El objetivo principal es la presentación de las metodologías y herramientas de diseño aplicadas al desarrollo de un circuito integrado digital de pequeña a mediana complejidad, un controlador USB 2.0. Se espera entregar un documento que recorra los principales conceptos sobre el diseño de circuitos integrados, además de una solución práctica a cada uno de los problemas y metodologías de cada etapa del diseño.

[1]Corporación de Fomento de la Producción.
[2]La CORFO propone muchas líneas de apoyo financiero además del proyecto InnovaChile.
[3]Lo que significa llegar a 25.000 dólares en ese año

Como consecuencia de lo anterior, se espera demostrar y motivar el diseño de circuitos integrados en Chile, partiendo por el entorno más cercano: el Departamento de Ingeniería Eléctrica de la Facultad de Ciencias Físicas y Matemáticas de la Universidad de Chile.

1.2.2. Objetivos Específicos

Los objetivos específicos se describen a continuación:

- Determinar y dividir las funciones que realiza un controlador USB 2.0 en base a su especificación [8].
- Generar un modelo del controlador en base a las funciones determinadas, lo cual implica la definición de módulos, interacciones entre ellos e interfaces externas.
- Implementar a nivel RTL[1] el modelo, utilizando el lenguaje de descripción de hardware Verilog.
- Verificar la funcionalidad de la descripción RTL del modelo, realizando simulaciones coherentes con el diseño.
- A partir de la descripción RTL ya validada, generar una lista de compuertas lógicas basadas en una tecnología específica y que cumpla con la funcionalidad y restricciones que afectan su desempeño.
- Realizar el proceso de traducción de la lista de compuertas a un plano físico, el cual está compuesto por la representación física de cada compuerta y sus conexiones, entre otros elementos.

1.3. Estructura de la Memoria

La memoria está estructurada de manera que los capítulos deben ser analizados secuencialmente, ya que cada uno tiene información relevante y necesaria para el posterior. A continuación se describen los capítulos sin considerar el de Presentación:

- **Antecedentes:** Este capítulo describe principalmente un flujo estándar de diseño de circuitos integrados, además de las características generales del protocolo de comunicación USB 2.0. Su objetivo es dar un respaldo informativo a los siguientes capítulos.
- **Descripción del Sistema Implementado:** Este capítulo contiene, por una parte, las características específicas del protocolo USB que deben ser consideradas en el controlador, y por otra, se describe parcialmente el modelo de la solución realizado para la implementación del controlador.

[1]Siglas en inglés de *Register Transfer Level* o Nivel de Transferencia de Registros.

- **Diseño del Controlador USB 2.0:** En este capítulo, para cada etapa del diseño, se describe el proceso realizado, se muestran resultados intermedios y se explican los principales conceptos asociados a cada una de ellas.
- **Discusión y Conclusiones:** En este capítulo se realiza la conclusión del trabajo respecto a los objetivos planteados, y se deja claro el trabajo a futuro que se puede realizar sobre el diseño.

Esta memoria no contiene una sección dedicada a los resultados y sus análisis, ya que el desarrollo los requiere e incluye para cada una de sus etapas, además, este documento es un resultado de por sí, dado los objetivos ya descritos.

Capítulo 2

Antecedentes

Este capítulo se divide en dos partes: En la primera se explica el flujo de diseño de circuitos integrados digitales, mientras que en la segunda se describe a *grosso modo* el sistema USB basado en la especificación de la versión 2.0. En capítulos posteriores se muestran detalles de diseño asociados a la aplicación en cuestión.

2.1. Flujo de Diseño

En esta sección se describe el flujo de diseño y sus etapas así como las herramientas de Synopsys® involucradas en cada una de ellas.

2.1.1. Evolución

El diseño de circuitos integrados ha evolucionado rápidamente los últimos 25 años. Los primeros circuitos eran diseñados con tubos de vacío y transistores. Luego se inventaron los circuitos integrados, en los cuales se colocan compuertas lógicas dentro de un chip [4]. Desde ese momento se creó mucha terminología para referirse a los distintos niveles de integración [3], es decir, a la cantidad de transistores contenidos en un solo chip. Se partió con SSI[1], luego se pasó a MSI[2] y cuando se llegó a LSI[3] el proceso de diseño empezó a ser muy complicado, por lo cual los diseñadores sintieron la necesidad de automatizar los procesos, y las llamadas técnicas de Automatización de Diseño Electrónico[4] empezaron a evolucionar. En este punto, los diseñadores comenzaron a utilizar herramientas de simulación para verificar la funcionalidad de los bloques de un orden de aproximadamente 100 transistores. Los circuitos

[1]Sigla en inglés de Escala Pequeña de Integración.
[2]Sigla en inglés de Escala Mediana de Integración.
[3]Sigla en inglés de Escala Grande de Integración.
[4]EDA por sus siglas en inglés.

aún se probaban en *protoboards* y el *layout*[1] se hacía a mano en papel o en un computador con herramientas gráficas [4].

Cuando se llegó a VLSI[2], se podían colocar más de 100.000 transistores en un solo chip, de manera que los circuitos ya no podían ser probados en un protoboard. Entonces, las técnicas computacionales de verificación y diseño se volvieron críticas, y los programas computacionales para hacer automáticamente el layout de *place&route* se hicieron populares. Ahora los diseñadores construían los circuitos a nivel de compuerta en terminales gráficos. Partían construyendo bloques pequeños, para luego hacer bloques más grandes a partir de ellos. El proceso seguía hasta que llegaban al bloque del nivel más alto (*top-level block*), lo cual responde la definición de diseño *bottom-up*. Luego aparecieron los simuladores lógicos para verificar la funcionalidad de estos bloques antes de mandar a fabricar el chip [4].

A medida que la geometría de los semiconductores sub-micro se hace más pequeña, los métodos tradicionales de diseño de circuitos integrados se hace cada vez más difícil. Además, se colocan más y más transistores en un mismo *die size* [3], de manera que la validación del diseño es extremadamente difícil, a veces imposible. Aún más, bajo presión de *time to market*[4], el ciclo de diseño de circuitos integrados se ha mantenido igual, o ha sido constantemente reducido. Para contrarrestar estos problemas, nuevos métodos y herramientas han evolucionado para facilitar la metodología de diseño de ASIC's[5] [3].

2.1.1.1. Aparición HDL's

Los primeros Lenguajes de Descripción de Hardware[6] fueron desarrollados alrededor del año 1977 y fueron llamados ISP y KARL, aunque eran más bien lenguajes de programación que describían las relaciones entre entradas y salidas de un diseño [9].

Los ahora populares **Verilog** y **VHDL** HDL's se originaron a mediados de los años 1980's en Gateway Design Automation y DARPA respectivamente. A diferencia de los lenguajes de programación, los HDL's permiten realizar operaciones concurrentes además de secuenciales. En un comienzo se utilizaban básicamente para documentar y simular diseños de circuitos que habían sido hechos de otra manera, generalmente a través de esquemáticos. Con la aparición de la síntesis lógica a finales de los años 1980's cambiaron el método de diseño drásticamente [4]. Ahora bastaba con realizar la descripción del circuito en HDL para que luego la herramienta de síntesis llevara el diseño a un nivel de *gate net-list*[7].

En particular, Verilog es aceptado como un estándar IEEE desde 1995, siendo IEEE 1364-1995 el original. La última versión del estándar es el IEEE 1364-2001, el cual tiene

[1] Layout para este caso es el mapa de interconexiones entre compuertas lógicas.
[2] Sigla en inglés de Escala Muy Grande de Integración.
[3] Espacio que utiliza un chip en una oblea de silicio.
[4] Tiempo que toma desarrollar un producto desde la idea inicial a las ventas iniciales en el mercado.
[5] Sigla en inglés de Circuitos Integrados de Aplicación Específica.
[6] En inglés, *Hardware Description Language* (HDL).
[7] Este nivel es una descripción del circuito en términos de compuertas y conexiones entre ellas.

cambios significativos respecto de la versión original.

2.1.1.2. Ventajas de los Lenguajes de Descripción de Hardware

En un comienzo el diseño de circuitos integrados a través de HDL's y herramientas de síntesis no daba resultados tan óptimos como el diseño directo con esquemáticos, aunque de todos modos era más rápido. Con la evolución de las herramientas de diseño el proceso mejoró, manteniendo sus ventajas originales. A continuación se listan las principales atributos de los HDL's:

- Diseño en nivel abstracto, independiente de teconología.
- Permite simular y eliminar casi completamente errores de funcionalidad antes de hacer el chip.
- Permite un mejor entendimiento de un diseño que su representación esquemática, aún más en diseños complejos.
- En el caso de Verilog, es un lenguaje de descripción de hardware de propósito general y similar al lenguaje de programación C, por lo cual fácil de aprender y usar.

2.1.2. Descripción del Flujo de Diseño

Hay dos maneras de abordar el diseño de un circuito integrado [10]:

- **Bottom-up:** Se comienza con el diseño de partes pequeñas, para a partir de éstas formar bloques más grandes, de mayor complejidad, hasta llegar al bloque *top-level*. Se recomienda para optimizar los bloques más pequeños que se vayan a utilizar.
- **Top-down:** Esta técnica consiste en partir del nivel más complejo e ir descomponiéndolo progresivamente para llegar a lo simple. Se utiliza para el diseño de sistemas más complejos en general.

A continuación se muestra una figura explicativa de ambos enfoques [10]:

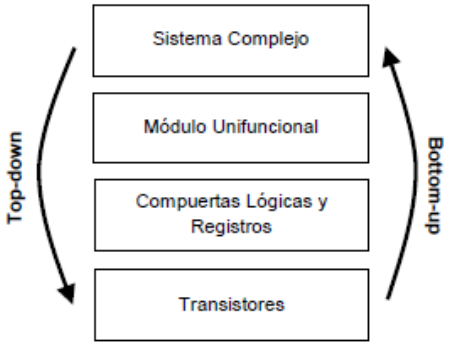


Figura 2.1: Enfoques de Diseño

En el diseño top-down es fundamental el concepto de síntesis, el cual se refiere al proceso de traducción de un modelo desde un nivel de abstracción mayor a otro equivalente en función, pero de nivel inferior. En cada una de las etapas de diseño del circuito, se deben optimizar ciertos objetivos de diseño (como área, velocidad y disipación de potencia), satisfaciendo una serie de restricciones que se le imponen (tasas de muestreo, instrucciones por segundo, etc.) [10].

El flujo de diseño mostrado en la figura 2.2, es usado típicamente por diseñadores que usan Lenguajes de Descripción de Hardware [4]. Se observa que se parte en un nivel alto de abstracción y se termina en el nivel inferior pasando por una serie de etapas de abstracción intermedia, verificación y retroalimentación para la optimización del circuito. Otros flujos de diseño similares se pueden encontrar en [3], donde se hacen explícitas más etapas de verificación para la inserción del *clock tree*[1].

2.1.2.1. Especificación y Descripción RTL

El diseño de un chip comienza con la concepción de una idea proveniente de una necesidad del mercado. Luego estas ideas se traducen en especificaciones eléctricas y arquitecturales. Las especificaciones arquitecturales definen la funcionalidad y partición del chip en varios bloques manejables[2], mientras que las especificaciones eléctricas describen las relaciones entre estos bloques en términos de la información de sincronización[3] [3].

Las especificaciones deben ser implementadas luego de definidas. Ésto se realiza utilizando HDL's, como Verilog y VHDL siendo los más populares de hoy en día. Existen 3 niveles de abstracción que pueden ser utilizados para describir el diseño: Comportamental[4], RTL[5] y Estructural[6] [3].

[1]Es la distribución de la señal de clock en el circuito, dada las restricciones de éste [11].
[2]Bloques relativamente simples en función e interfaz para comunicación con otros.
[3]Del inglés *timing*.
[4]Del inglés *Behavioral*.
[5]Sigla en inglés de Nivel de Transferencia de Registros.
[6]Del inglés *Structural*.

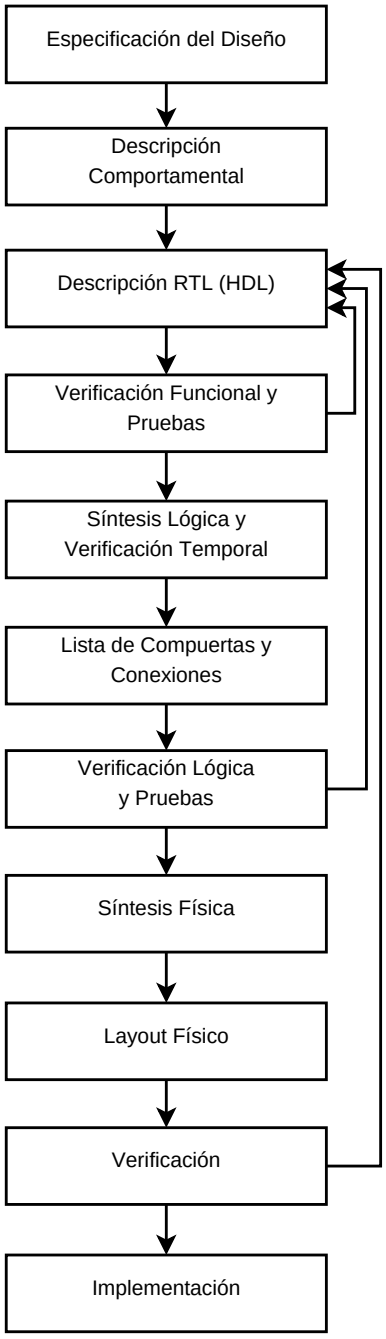


Figura 2.2: Flujo de Diseño Típico [4]

El nivel comportamental se utiliza básicamente para lograr simular el diseño inicial, y así verificar su factibilidad. Recientemente están apareciendo compiladores que sintetizan este nivel de abstracción (el más alto) [3]. La descripción en RTL se utiliza ampliamente para describir las especificaciones y luego se sintetiza para pasar a un *net-list* de nivel de compuerta. En sí, RTL es una mezcla entre formas[1] de *Data Flow*[2] y comportamentales [4].

[1]Traducido del inglés *construct* en un contexto de lenguajes de programación.
[2]Descripción de la parte combinacional entre registros de un circuito [4].

2.1.2.2. Simulación Dinámica

Luego de descrito un circuito en HDL se debe simular el RTL para monitorear si la función es efectivamente la que se quiere. Todos los simuladores actuales son capaces de simular tanto el nivel comportamental como RTL [3]. Es importante notar que lo que se verifica en este paso es solo la funcionalidad del circuito sin consideración de restricciones.

En este punto aparece lo que se conoce como *test bench*, el cual genera los estímulos (señales de entrada) al código RTL a simular y monitorea las salidas. El *test bench* se escribe usualmente a nivel comportamental para simular RTL, y es una pieza fundamental para lograr un cubrimiento adecuado de todos los posibles casos en que el circuito pueda estar.

2.1.2.3. Síntesis Lógica

Una vez verificada la funcionalidad del circuito a través de simulaciones, se debe sintetizar. Ésto significa reducir el RTL a un nivel de interconexiones de compuertas[1].

El proceso de sintetizar un diseño es iterativo y comienza con la definición de las restricciones (de tiempo, área y potencia) para cada bloque del circuito [3]. Estas restricciones indican la relación entre cada señal con respecto al reloj de entrada. Además se debe especificar la librería de celdas o librería de tecnología a utilizar durante la síntesis. Estas librerías implementan una representación de las compuertas utilizadas en el diseño, es decir, definen las características temporales y físicas de las compuertas de una cierta tecnología, de manera de realizar una síntesis y optimización con los elementos que se van a usar al fabricar el circuito.

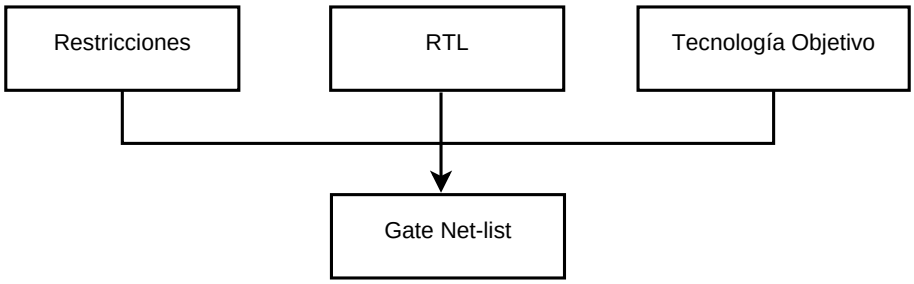


Figura 2.3: Etapas de Síntesis Lógica

Luego de sintetizar el diseño es necesario realizar un *Análisis Estático de Tiempo* [3]. En caso de que no todos los requerimientos temporales se cumplan puede ser necesario volver a modificar el RTL antes de continuar con el diseño.

[1]Del inglés *gate-level netlist*.

2.1.2.4. Verificación Formal

Las técnicas Formales de verificación realizan la validación de un diseño utilizando métodos matemáticos sin necesidad de consideraciones temporales y efectos físicos. Comparan las funciones lógicas de un diseño con las de un diseño referencia [3].

La principal ventaja de esta verificación es que, a diferencia de la simulación dinámica, permite cubrir todo el diseño al comparar toda la estructura y funcionalidad lógica en vez de revisar casos particulares. De esta manera, la verificación formal se puede utilizar para verificar, por ejemplo, si hubo cambios en la funcionalidad al pasar de RTL a compuertas, o luego de compuertas al *layout*.

2.1.2.5. Análisis Estático de Tiempo

Este análisis es muy importante, ya que además de revisar los caminos críticos después de la síntesis (en la llamada etapa *pre-layout*[1] permite obtener de manera ordenada los caminos críticos en la etapa *post-layout*. El informe suele contener además información como el *fanout*[2] y carga capacitiva de cada *net*.

Si la información que se obtiene es aceptable, entonces ésta se puede utilizar como entrada para la herramienta de *layout*, lo que acelera el proceso de optimización al partir más cerca de un punto óptimo. La información resultando de la etapa *post-layout* contiene nueva información de las interconexiones en términos de la capacitancia y retardos[3] RC. Ésto puede utilizarse para volver a una etapa anterior, realizar análisis estático de tiempo, y ver si se siguen cumpliendo las restricciones.

El proceso es puramente iterativo en la búsqueda de un óptimo que satisfaga las restricciones temporales y de área definidas.

2.1.2.6. Síntesis Física

El objetivo de la etapa de síntesis física en el diseño de un circuito integrado es, a partir de la lista de compuertas lógicas interconectadas obtenidas de la síntesis lógica, restricciones temporales y físicas, generar un mapa o *layout* de la ubicación de todas las compuertas, conexiones y *pads* implementadas a través de distintas capas de metal. La síntesis física posee principalmente 3 sub-etapas: planificación del diseño o *floorplan*, ubicación y ruteo.

[1] Es decir antes del *place and route*.
[2] Número de compuertas lógicas que se conecta a la salida de otra del mismo tipo [12].
[3] Del inglés *delay*.

La sub-etapa de *floorplan* o plano de planta[1] consiste en decidir donde se ubicarían todos los elementos (compuertas) del circuito dentro de un espacio limitado, definiendo zonas de bloqueo[2], ubicación de celdas de mayor tamaño (como memorias RAM), malla de alimentación y ubicación de los puertos de entrada y salida. Esta sub-etapa es más crítica que la de ruteo, ya que una ubicación óptima de las celdas no solo acelera el proceso de ruteo si no que se obtienen mejores características de tiempo [3].

La sub-etapa de ubicación de celdas consiste en ubicar óptimamente todas las compuertas del circuito en las zonas definidas en el *floorplan*. El objetivo principal es evitar la congestión, lo que se traduce en evitar que existan lugares en el circuito donde al rutear se tendrían muchas conexiones.

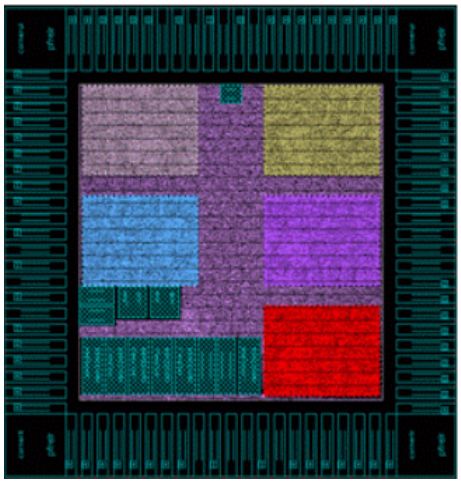


Figura 2.4: *Floorplan* + Ubicación de Celdas de un Circuito

En general, previo al ruteo del diseño, se realiza la inserción del árbol de reloj o *clock tree*, lo cual consiste en el ruteo óptimo del reloj del circuito, de manera de evitar el *skew*[3] de él. Este proceso puede agregar o modificar elementos utilizados con el propósito de balancear los retardos en todas las ramas del árbol de reloj, por lo tanto el *clock tree* se debe insertar también en la *netlist* original y verificar formalmente si se mantiene la funcionalidad.

En la sub-etapa de ruteo se realiza el "alambrado" entre todos los componentes [13]. En general, las herramientas de *layout* realizan el ruteo en dos etapas: global y detallada. Un ruteo global se utiliza para revisar la calidad de la ubicación de las celdas y además para obtener una aproximación de los retrasos reales que se obtendrían después de un ruteo detallado. Esta información se puede utilizar para mejorar el diseño en alguna etapa anterior incluyendo el *floorplan* y luego seguir con el ruteo detallado cuando se cumplan las restricciones.

[1]Traducido literalmente de *floorplan*, es un esquemático con la ubicación tentativa de los bloques principales del circuito [14].

[2]Lugares donde no puede ir ninguna compuerta o celda.

[3]Diferencia en el tiempo de llegada del reloj a distintas compuertas.

2.1.3. Herramientas de Diseño

A continuación se especifican las herramientas de Synopsys® asociadas a las principales etapas del flujo de diseño:

- **Simulación Dinámica y Pruebas:** Verilog Coded Simulator (VCS).
- **Síntesis Lógica y Análisis Temporal:** Design Compiler® (DC).
- **Verificación Formal:** Formality.
- **Planificación, ubicación y ruteo:** Integrated Circuit Compiler (ICC).

Tanto Design Compiler® como IC Compiler poseen herramientas para realizar análisis temporal antes y después de cada una de las etapas de diseño asociadas a estas herramientas.

2.2. USB 2.0

En esta sección se describe de manera general las características de *Universal Serial Bus 2.0*. Para todos los efectos, la información aquí mostrada está basada en su totalidad en [8].

2.2.1. Descripción del Sistema USB

El bus utiliza un protocolo basado en *token*[1], el cual es administrado por el host, para comunicarse con los periféricos conectados a él. El bus permite conectar, sacar, configurar y usar periféricos mientras otros de ellos están en operación.

Un sistema USB está descrito por 3 áreas:

- Interconexión USB
- Dispositivos USB
- Host USB

La interconexión USB es la manera en la cual los dispositivos estan conectados y se comunican con el host. Esto incluye lo siguiente:

- Topología del Bus: Modelo de conexión entre dispositivos y host.
- Relaciones entre Capas: Las tareas que son ejecutadas en cada capa del sistema.
- Modelos de Flujo de Datos: La manera en que los datos se mueven en el sistema USB.
- Programa USB: Organización temporal para el acceso al bus de manera de permitir transferencias isocrónicas[2] y eliminar arbitraje de overhead.

2.2.1.1. Topología del Bus

La interconexión USB física es de estrella jerarquizada, donde el centro de la estrella es el *hub* raíz. Se permiten hasta 7 niveles de jerarquización incluyendo el nivel raíz y no pudiendo colocar hubs en la última capa.

[1]Un token o téstigo[15] es un paquete de datos que viaja de nodo a nodo para indicar permiso para transmitir.

[2]En inglés *Isochronous*. Se refiere a los procesos donde los datos deben ser transmitidos bajo ciertas restricciones de tiempo.

2.2.1.2. Host USB

Hay un solo host en cualquier sistema USB. Se le llama Controlador Host a la interface USB que se comunica con el sistema del computador host. Éste puede ser implementado como un conjunto de hardware, firmware y software. Al host también se le agrega un hub raíz, de manera de permitir más puntos de acceso USB en el centro de la topología.

2.2.1.3. Dispositivos USB

Los dispositivos USB pueden ser:

- Hubs, los cuales proveen puntos adicionales de conexión USB.
- *Funciones* (así llamados por el estándar), las cuales proveen capacidades al sistema, como una conexión ISDN (Red Digital de Servicios Integrados), un *joystick* digital o parlantes.

Los dispositivos USB presentan una interface USB estándar en términos de lo siguiente:

- Su comprensión del protocolo USB.
- Su respuesta frente a operaciones USB estándar, como configuración y reset.
- Su información descriptiva de capacidades estándar.

2.2.2. Interfaz Eléctrica

USB transmite la señal y alimentación a través de un cable de 4 alambres como se muestra en la figura 2.5. La señalización ocurre sobre dos alambres en cada segmento punto-a-punto.

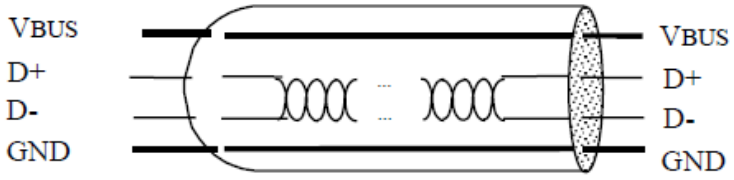


Figura 2.5: Cable USB

Existen 3 tasas transmisión de datos:

- *High-speed* a 480 Mb/s.
- *Full-speed* a 12 Mb/s.
- *Low-speed* a 1.5 Mb/s.

El modo low-speed permite un número limitado de dispositivos de bajo ancho de banda (como el *mouse*), ya que un uso más general degradaría la utilización del bus.

El *clock* (reloj) es enviado junto con los datos diferenciales. Se utiliza codificación NRZI con relleno de bit para asegurar captura y transiciones adecuadas. Un campo SYNC precede a cada paquete para que el receptor pueda sincronizar su reloj de recuperación de bit.

El cable también transporta los alambres de V_{BUS} y GND para entregar la potencia necesaria a los dispositivos. V_{BUS} tiene un valor nominal de +5 [V] en la fuente.

2.2.2.1. Distribución de Potencia

Cada segmento USB proporciona de una cantidad limitada de potencia a través del cable. El host suministra la potencia a los dispositivos que están directamente conectados. Además, cada dispositivo puede tener su propia fuente de alimentación para funcionar.

2.2.2.2. Administración de Potencia

Un host USB puede tener un sistema de administración de potencia independiente de USB. El software del sistema USB interactúa con el sistema de administración de potencia del host para manejar los eventos de potencia del sistema, como *suspender* o *resumir*. Por otra parte, los dispositivos USB típicamente implementan características adicionales de administración de potencia para que los temas de potencia sean administrados por el software de sistema.

Las características distribución y administración de potencia de USB permiten que este sea diseñado para sistemas que son críticos en potencia, como en notebooks basados en baterías.

2.2.3. Protocolo del Bus

USB es un bus *polled*, lo que quiere decir que el controlador Host inicia todas las transferencias (en todas direcciones).

La mayoría de las transacciones del bus implican la transmisión de hasta tres paquetes. Cada transacción comienza cuando el Controlador Host, de manera programada, envía un token paquete describiendo el tipo y sentido de la transacción, la dirección del dispositivo USB al que va dirigido, y un número de *endpoint* (ver sección 3.2.3.5).

El dispositivo USB direccionado en el paquete se selecciona así mismo decodificando el campo de dirección. En una transacción dada, los datos son transferidos ya sea desde el host al dispositivo o viceversa. Entonces, el origen de la transacción envía un paquete de datos o indica que no tiene datos para transferir. El destino, en general, responde con un paquete de *handshake*[1] para indicar si la transferencia fue exitosa.

El modelo USB de transferencia de datos entre origen o destino en el host y un endpoint en un dispositivo se denomina *pipe*. Hay dos tipos de pipes: *stream* y *message*. Los datos stream no tienen una estructura USB definida, mientras que los datos del tipo message si. Por otro lado, los pipes tienen atributos de ancho de banda de datos, tipo de servicio de transferencia, y características de endpoint como direccionalidad y tamaños de buffer.

La mayoría de los pipes son creados cuando el dispositivo USB está configurado. Hay un pipe del tipo message que siempre existe desde que el dispositivo es encendido, este es el pipe de Control por defecto. El pipe de control por defecto provee acceso a la configuración del dispositivo, al estado e información de control.

La programación de la transacción permite control de flujo para algunos *stream pipes*. A nivel de hardware, esto previene un sub-desbordamiento [16] o desbordamiento de los buffers al usar un NAK[2] handshake para "estrangular" la tasa de transmisión. Las transacciones son re-intentadas cuando hay tiempo disponible en el bus. El mecanismo de control de flujo permite la construcción de una pogramación flexible de manera de acomodar el servicio concurrente de una mezcla heterogénea de stream pipes. De esta manera se pueden ejecutar muchos stream pipes a diferentes intervalos de tiempo y con paquetes de distinto tamaño.

2.2.3.1. Detección de Errores

Cuando se requiere integridad de datos, como con los dispositivos de datos sin pérdidas, se puede utilizar un procedimiento de recuperación desde el hardware o software.

El protocolo incluye CRC's (cyclic redundancy checks) separados para los campo de control y datos de cada paquete. Si un CRC falla se considera que el paquete está corrupto. Esto entrega un 100 % de cobertura en los errores de uno y dos bits.

[1] Handshake en español es un apretón de manos, lo cual para este caso un paquete handshake se refiere a un tipo de respuesta determinada por la situación.

[2] Not Acknowledge, se refiere a una respuesta que indica que los datos están erróneos o no pueden ser aceptados.

2.2.3.2. Manejo de Errores

El protocolo permite que el manejo de los errores sea realizado a través de hardware o software. Si se hace a través del *hardware*, esto incluye reportar el error y reintentar las transferencias fallidas. El Controlador Host inentará retransmitir 3 veces antes de informar al *software* cliente de la falla. Este último puede recuperarse de una manera esepecífica a su implementación.

2.2.3.3. Tipos de Flujo de Datos

Las transferencias de datos se realizan entre el *software* del *host* y *endpoint* (ver sección 3.2.3.5) particular en un dispositivo. Tales asociaciones entre software *host* y *endpoints* del dispostivo son llamadas *pipes*[1]. En general, el movimiento de datos por un *pipe* es independiente del flujo de datos de cualquier otro *pipe*. Un solo dispositivo USB puede tener muchos *pipes*, como por ejemplo podría tener un *endpoint* que soporte un *pipe* para transportar datos al dispositivo y otro *endpoint* que soporte un *pipe* para transportar datos desde el dispositivo.

USB 2.0 posee cuatro tipos de transferencia básicos (un *pipe* puede soportar solo uno de estos tipos para una configuración dada):

- **Transferencias de Control**[2]
  Este modo es utilizado para configurar un dispositivo en el momento en que se conecta y también puede ser usado para otros propósitos específicos de un periférico, incluyendo el control de otros *pipes* en el dispositivo (por ejemplo para levantar la condición de detención). La entrega de los datos es sin pérdidas.

- **Transferencias Voluminosas de Datos**[3]
  Este modo típicamente consiste en grandes cantidades de datos secuenciales, como en el caso de las impresoras y scanners. El intercambio confiable de datos es asegurado a nivel de hardware usando detección de errores e invocando un número limitado de re-intentos de transmisión. Además, el ancho de banda puede variar dependiendo de las otras actividades del bus. También es sin pérdidas.

- **Transferencias Interrumpidas de Datos**[4]
  La transferencia de este modo desde o hacia un dispositivo tiene latencia limitada. Los datos son presentados para transmitirlos por un dispositivo en cualquier momento y son entregados por USB a una velocidad no menor que la especificada por el periférico. Este tipo de datos típicamente consisten en notificaciones de eventos, caracteres o coordinadas que son organizados en uno o más bytes. Un ejemplo de estos datos son las coordinadas de un dispositivo puntero.

[1]Un *pipe* corresponde a un tubo en español.
[2]Del inglés, *Control Transfers*.
[3]Del inglés, *Bulk Transfers*.
[4]Del inglés, *Interrupt Transfers*.

- **Transferencias Isocrónicas de Datos**[1]

  Este modo utiliza una cantidad pre-negociada de ancho de banda USB y latencia de entrega. Los datos isocrónicos son continuos y de tiempo real en términos de su creación, entrega y consumo. La información relacionada con la coordinación temporal es consecuencia de una tasa de transmisión segura, a la cual los datos iscrónicos son recibidos y transferidos. Para los *pipes* isocrónicos, el ancho de banda requerido es basado típicamente en las características de *sampling* de la función (dispositivo) asociada. La latencia requerida está relacionada con la capacidad de los *buffers*[2] en cada *endpoint*. Estos datos son especificados al configurar el dispositivo.

  La entrega oportuna de los datos isocrónicos es asegurada en desmedro de potenciales pérdidas transientes en el flujo de datos. En otras palabras, cualquier error eléctrico en la transmisión no es corregido por el hardware. Se le reserva una porción del ancho de banda dedicada a los flujos de datos isocrónicos, de manera de asegurar que éstos puedan ser entregados a la velocidad deseada.

USB reserva ancho de banda para algunos *pipes* cuando uno de ellos es establecido. Se requiere que los dispositivos provean de *buffers* para los datos. Se asume que los dispositivos que requieran más ancho de banda son capaces de proveer buffers más grandes. El objetivo de la arquitectura USB es asegurar que el retardo producido por los *buffers* esté limitado a unos cuantos milisegundos.

La capacidad del ancho de banda USB puede ser reservado para varios flujos de datos diferentes, lo que permite un amplio rango de dispositivos conectarse al bus. Aún más, dispositivos de diferentes velocidades con un amplio rango dinámico pueden ser soportados concurrentemente.

2.2.4. Host USB: Hardware y Software

El host USB interactúa con los dipositivos USB a través del Controlador Host. El host es responsable de lo siguiente:

- Detectar conexión y extracción de dispositivos.
- Administrar flujo de control entre host y dispositivos.
- Administrar flujo de datos entre host y dispositivos.
- Obtener estadísticas de estado y actividad.
- Proveer alimentación a los dispositivos conectados.

[1] Del inglés, *Isochronous Transfers*.
[2] En este caso se refiere a un lugar de almacenamiento de datos.

El Software de Sistema USB en el host administra las interacciones entre dispositivos USB y software de dispositivos basados en el host. Hay 5 áreas de interacciones entre el Software de Sistema USB y el *software* del dispositivo:

- Enumeración y configuración del dispositivo.
- Transferencias isocrónicas.
- Transferencias asíncronas.
- Adminitración de energía.
- Información de administración de dispositivo y bus.

2.2.5. Dispositivos USB

Los dispositivos USB están divididos en clases como hub, interfaz humana, impresora, imagenología o almacenamiento. Existen en realidad 2 divisiones mayores de clases de dispositivos: hubs y funciones. Los hubs son los que permiten tener mayor cantidad de puertos USB, mientras que las funciones proveen de capacidades adicionales al host.

Todos los dispositivos USB son accedidos a través de una dirección USB que es asignada cuando se conecta y enumera el dispositivo. Adicionalmente, cada dispositivo puede tener uno o más *pipes* para que el host se pueda comunicar. Todos los dispositivos deben proveer de un *pipe* especial en el *endpoint* cero a través del cual el pipe de control del dipositivo será conectado. Todos los dipositivos tienen un mecanismo común de acceso para acceder a la información a través de el pipe de control mencionado anteriormente.

La información asociada al *pipe* de control en el *endpoint* cero es requerida para describir completamente el dispositivo USB. Esta información se categoriza de la siguiente manera (además de la información de control y estado):

- Estándar: Es información que es común a todos los dispositivos USB e incluye datos como la identificación del fabricante, la clase de dispositivo y la capacidad de administración de potencia.
- Clase: Este tipo de información varía dependiendo de la clase del dispositivo.
- Fabricante USB: El fabricante puede colocar la información que quiera en este tipo. El formato no es definido por la especificación.

A continuación se describen las clases de dispositivos:

- **Hubs**

  Los hubs son concentradores cableados y permiten la conexión de multiples dispositivos. Cada hub convierte un punto de conexión en múltiples puntos de conexión,

permitiéndose también la conexión en cadena de muchos hubs. El hub se conecta al host a través del puerto *upstream*, y a las funciones u otros hubs a través del puerto *downstream*.

Un hub USB 2.0 tiene tres partes: el Controlador Hub, el Repetidor Hub y el Traductor de Transacciones. El repetidor hub es un conmutador controlado por protocolo entre los puertos *upstream* y *downstream*. También tiene soporte por *hardware* para las señalizaciones de reinicio (reset) y suspender/reanudar. El Controlador Host provee la comunicación hacia y desde el host. Comandos específicos del hub de estado y control permiten al host configurar un hub y monitorear y controlar sus puertos. El Traductor de Transacciones provee mecanismos que soportan dispositivos de velocidad completa/lenta conectados al hub, mientras que transmite los datos del dispositivo entre el host y hub a alta velocidad.

- **Funciones**

  Una función es un dispositivo USB que es capaz de transmitir o recibir datos o información de control sobre el bus. Cada función contiene información de configuración que describe sus capacidades y requerimientos de recursos. Antes de que una función pueda ser usada, ésta debe ser configurada por el host. Esta configuración incluye la reservación de ancho de banda USB y la selección de las opciones específicas de la función.

2.2.5.1. Conexión de los Dispositivos USB

Todos los dispositivos USB se conectan a los puertos USB de unos dispositivos especializados conocidos como hubs. Éstos tienen bits de estado que son usados para reportar la conexión o extracción de un dispositivo USB de alguno de sus puertos. El host le pide al hub que recupere los bits de estado. En caso de conexión de un dispositivo, el host habilita el puerto y le da una dirección al periférico a través de su pipe de control en la dirección por defecto.

El host le asigna una dirección única al dispositivo y luego determina si es un hub o una función. El host establece su final del pipe de control para el dispositivo USB usando la recién asigana dirección y el endpoint número cero.

Si el dispositivo conectado resulta ser un hub y tiene otros perfiféricos conectados a él, entonces el procedimiento descrito anteriormente se repite para cada uno de sus puertos. En cambio si es una función, entonces las notificaciónes de conexión serán manejadas por el software del host que es apropiado para esa función.

2.2.5.2. Extracción de Dispositivos USB

Cuando se extrae un dispositivo USB de uno de los puertos de un hub, el hub deshabilita ese puerto y proporciona una indicación al host de que el dispositivo se sacó. Luego, esta indicación es procesada por un software USB de sistema apropiado. En el caso en que se extraiga un hub, el software debe manejar el hecho de sacar ese hub y todos los periféricos que estaban conectados a él.

2.2.5.3. Enumeración del Bus

Enumeración del bus es la actividad que identifica a cada dispositivo (conectado) con una dirección única. Como los eventos de conectar o extraer un dispositivo puede ocurrir en cualquier momento, la enumeración del bus es una actividad contínua del software de sistema USB. Esta característica también incluye la detección y procesamiento de extracción.

Capítulo 3

Descripción del Sistema Implementado

En esta sección se presentan de manera detallada los principales requerimientos y características implementadas en base a la especificación de USB 2.0. Además, se describen las principales máquinas de estados e interfaz diseñadas para cumplir con el protocolo USB. En otras palabras, este capítulo corresponde al paso previo a la descripción RTL: modelamiento y especificación del diseño.

El controlador desarrollado posee algunas simplificaciones respecto de los ya existentes, las cuales también se describen en este apartado.

3.1. Organización Temporal de Transacciones

USB 2.0 utiliza un sistema de marcos[1] o ventanas de tiempo para organizar las transacciones de datos en el bus. Un *frame* (como será utilizado de ahora en adelante) o marco corresponde a $1[ms]$, además se definen los llamados *micro-frames* que ocurren cada $125[\mu s]$, en otras palabras, se tienen 8 micro-frames en cada frame. Al comienzo de cada micro-frame se transmite un paquete especial para identificarlo a través de un número (contador) que se incrementa para cada frame.

En High-speed se utilizan micro-frames para organizar las transacciones. Se puede reservar hasta un 80 % de un micro-frame para transacciones periódicas (interrupciones e isocrónicas). El resto del espacio se utiliza para las transacciones de volumen y control (no críticas en requerimientos de ancho de banda y retardos[2]). Ninguna transacción debe ejecutarse si es que no fuese a terminar antes del punto denominado EOF1[3], lo que corresponde

[1] Del inglés, *Frames*.
[2] Del inglés, *delay*.
[3] *End of (micro)Frame 1*, fin del (micro)marco.

a 560 tiempos de bit en High-speed.

El responsable de organizar las transacciones de cada micro-frame es el software (driver) del controlador USB, por lo tanto debe ser capaz de asegurar que las transacciones periódicas no superen el 80 % de cada micro-frame. El controlador se encarga de evitar que se ejecuten transacciones en los límites del micro-frame.

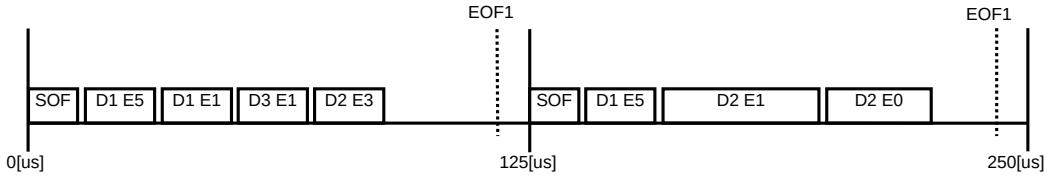


Figura 3.1: Ejemplo Organización micro-Frames

En la figura anterior se muestra al comienzo de cada micro-frame un paquete SOF (explicado más adelante), y luego el tiempo utilizado por distintas transacciones pertenecientes a distintos dispositivos (Dx) y endpoints (Ex).

3.2. Protocolo USB

En este apartado se describen las características que el controlador debe poseer para ser compatible con la especificación USB 2.0, salvo por las simplificaciones ya mencionadas.

3.2.1. Little-Endian

El orden en que se envían los datos en el bus es desde el bit menos (lsb) hasta el bit más significante (msb). Este orden no se aplica al paquete completo, sino por tramos, más específicamente a cada campo de datos que pertenece al paquete. Estos campos no son necesariamente de 8 bits (1 byte) de largo como se verá más adelante.

3.2.2. Tipos de Paquetes

La tabla 3.1 muestra los distintos tipos de paquetes implementados para llevar a cabo las funciones High-Speed de USB 2, clasificados por tipo de *PID* o Identificador de Paquete.

Tipo de PID	**Nombre del PID**	**PID[3:0]**	**Descripción**
Token	OUT	0001	Identificador para una transacción desde host a dispositivo.
	IN	1001	Identificador para una transacción desde dispositivo a host.
	SOF	0101	Indica el comienzo de un Frame o micro-Frame.
	SETUP	1101	Identificador para una el comienzo de una transacción de control.
Data	DATA0	0011	Identificador para paquetes de datos.
	DATA1	1011	Identificador para paquetes de datos.
	DATA2	0111	Identificador especial para paquetes de datos.
	MDATA	1111	Identificador especial para paquetes de datos.
Handshake	ACK	0010	Indica recepción correcta de datos, acuse de recibo.
	NAK	1010	Indica que no se pueden recibir o enviar datos, no acuse de recibo.
	STALL	1110	Indica que el dispositivo está imposibilitado de realizar transacciones.
	NYET	0110	Indica que aún el dispositivo no tiene espacio o tiempo para recibir más datos.
Especial	PING	0100	Se utiliza para verificar si un dispositivo es capaz de recibir datos.

Tabla 3.1: Tipos de Paquetes

3.2.3. Campos de Paquetes

Cada paquete está constituido por una serie de campos de distinto largo de bit. Todos los paquetes comienzan con un campo llamado SYNC, continúan con el identificador del paquete PID[1] (ver sección 3.2.2) y finalizan con otro denominado EOP[2].

3.2.3.1. SYNC

La función de este campo es sincronizar la recepción de los paquetes con el reloj local. Una vez convertido en código NRZI, es una secuencia de 15 pares de 0's y 1's terminando con dos 0's (32 bits en total) para indicar el fin del campo, lo que es utilizado como una señal de reloj de referencia para fijar un circuito extractor de reloj en la recepción, y así luego poder rescatar adecuadamente los datos que le siguen a este campo.

3.2.3.2. PID

Este campo es de 8 bits de largo y como ya se ha mencionado anteriormente, se encarga de identificar al paquete.

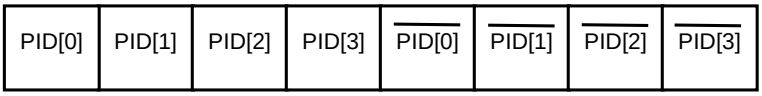


Figura 3.2: Estructura Campo PID

En la figura anterior los primeros cuatro bits corresponden a los que representan al PID como se muestra en la tabla 3.1, pero ordenados desde el bit menos significativo (PID[0]) al más significativo (PID[3]). Los siguientes 4 bits corresponden al campo de verificación del PID que simplemente son el opuesto de los bits anteriores en el mismo orden. Por ejemplo, el campo PID de un paquete SETUP sería 11010010.

3.2.3.3. EOP

Este campo indica el fin del paquete al forzar un error de rellenado de bits[3]. Ésto se realiza enviando al final del paquete la secuencia de bits 01111111, de manera que al codificar, el primer bit adquiere el valor inverso al del último bit del paquete para luego continuar con otros 7 bits con ese mismo valor. Por ejemplo si el bit codificado antes del EOP es un 0, entonces el EOP sería un byte de 1's.

[1]Del inglés, *Packet Identifier*.
[2]Fin de paquete, del inglés *End of Packet*.
[3]Del inglés, *bit stuffing*.

El EOP de un paquete SOF es distinto al de todos los demás paquetes. Éste utiliza una secuencia de 40 bits en vez de 8: 0111111111111111111111111111111.

3.2.3.4. Dirección

Este campo, de 7 bits, contiene la dirección del dispositivo al que va dirigida la transacción actual. Las direcciones se asignan en un proceso de enumeración que se realiza para cada dispositivo que se conecta al bus. Se pueden direccionar 127 dispositivos, ya que la dirección 0 está reservada como dirección por defecto.

El campo de dirección va en los paquetes OUT, IN, SETUP y PING, y se envía en orden *little-endian* (ver sección 3.2.1).

3.2.3.5. Endpoint

Un *endpoint* es el punto final[1] en una comunicación entre host y dispositivo. Este campo permite mayor flexibilidad para el direccionamiento de un dispositivo. Es de 4 bits por lo tanto cada dispositivo puede tener hasta 16 endpoints considerando que 0 corresponde al endpoint de control por defecto. Cada endpoint tiene una dirección (todas desde el punto de vista del host):

- In: Para transacciones que requieran mover datos desde un dispositivo al host.
- Out: Para transacciones que requieran mover datos desde el host a un dispositivo.
- Bidireccional: Para transacciones de control.

El campo de endpoint va en los paquetes OUT, IN, SETUP y PING, después del campo de dirección, y se envía en orden little-endian (ver sección 3.2.1).

3.2.3.6. Número de *Frame*

En un paquete SOF (ver sección 3.2.2), se indica a través de este campo de 11 bits el número del frame actual. El número se repite para cada uno de los 8 micro-frames y cambia al comienzo de cada frame.

[1]Traducción directa de *endpoint*.

3.2.3.7. Datos

El campo de datos corresponde, como lo dice su nombre, a todos los datos (limitado por la máxima cantidad de bytes para el tipo de transacción utilizado) que se requiere enviar en la transacción. Su largo va de 0 a 1024 bytes y cada byte se ordena de bit menos significante y bit más significante.

3.2.3.8. CRC de Paquetes Token

CRC o Comprobación de Redundancia Cíclica[1] es una función que se aplica sobre datos para obtener un código que permite revisar la integridad de ellos (detección de errores). El código es el residuo de la división polinomial de los datos con un polinomio particular.

El campo CRC para el caso de los paquetes Token es de 5 bits, se calcula sobre la dirección y endpoint (también se calcula sobre el número de frame en un paquete SOF), y luego se coloca después del endpoint (después del número de frame de un paquete SOF).

El polinomio utilizado para el CRC de 5 bits es el siguiente:

$$G(X) = X^5 + X^2 + 1$$

El patrón de bits que lo representa es 00101. El polinomio también se caracteriza por el residual que, para este caso, es la secuencia 01100. Este residual se utiliza para la verificación de que el campo de dirección, endpoint y CRC estén libres de errores.

En la recepción de un paquete Token, se calcula el CRC de los campos de dirección, endpoint y CRC recibidos. Si el resultado es igual que el residual descrito anteriormente entonces no existe ningún error en los campos protegidos por el código.

3.2.3.9. CRC de Paquetes de Datos

El modo de operar del CRC de un paquete de datos es el mismo que el de un paquete token, salvo que en este caso el residuo es de 16 bits, se utiliza otro polinomio y por lo tanto se tiene otro residual, y se calcula sobre todo el campo de datos. El polinomio utilizado para el CRC de 16 bits es el siguiente:

$$G(X) = X^{16} + X^{15} + X^2 + 1$$

El patrón de bits que lo representa es 1000000000000101, y el residual es 1000000000001101.

Ambos campos CRC no requieren cambiar el orden de los bits, ya que se calculan sobre datos que se encuentran en el orden correcto.

[1]Del inglés, *Cyclic Redundancy Check*.

3.2.4. Estructura de Paquetes

Cada tipo de paquete está formado por algunos de los campos mencionados en la sección anterior. Considerar lo siguiente para las figuras que vienen a continuación:

- Un paquete PING tiene la misma estructura que un token.
- ADDR se refiere al campo de dirección.
- ENDP se refiere al campo endpoint.
- CRC5 y CRC16 se refieren a los campos CRC de 5 y 16 bits respectivamente.
- FRAME_N se refiere al campo número de frame.

Figura 3.3: Estructura Paquete SoF

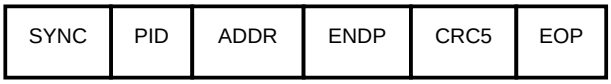


Figura 3.4: Estructura Paquete Token

Figura 3.5: Estructura Paquete de Datos

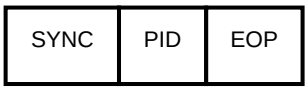


Figura 3.6: Estructura Paquete de Handshake

3.2.5. Tipos de Transferencias

Los cuatros tipos de transferencias fueron descritos en la sección 2.2.3.3. A continuación se presentan en una tabla las principales características de cada tipo de transferencia:

	Control	**Bulk**	**Interrupciones**	**Isocrónicas**
Tamaño Máximo [B]	64	512	1024	1024
Máxima Tasa de Transferencia [kB/s]	15.872	53.248	49.152	57.344
Dirección de la Transferencia	IN y OUT	IN o OUT	IN o OUT	IN o OUT
Corrección de Errores	si	si	si	no
Ancho de Banda Garantizado	no	no	no	si
Máxima Latencia Garantizada	no	no	si	si
Utiliza Protocolo Ping	si	si	no	no
Sincronización por Data PID	si	si	si	no

Tabla 3.2: Características de Tipos de Transferencias

La máxima tasa de transferencia para cada tipo está calculada para el caso más optimista en cada uno de ellos. Ésto significa que:

- Para las transferencias de control y bulk el micro-frame está libre para el uso completo por cada uno de ellos. En el caso de la transferencia de control se pueden realizar un máximo de 31 transacciones [8], y para bulk un máximo de 13 (ambos con tamaño de paquete máximo).
- Para las transferencias interrumpidas e isocrónicas se considera que para un endpoint con alta exigencia de ancho de banda se pueden realizar hasta 3 transacciones de 1024 bytes de datos cada una.

3.2.6. Estructura de Transferencias

Una transferencia desde el host a un endpoint (o viceversa) se compone de una o más transacciones. A su vez, cada transacción consiste de uno, dos o tres paquetes, los cuales corresponden a las fases de token, handshake o data. Una transacción posee como mínimo una fase token, y opcionalmente una de handshake y/o data. Luego, cada paquete se estructura como se describe en la sección 3.2.4.

A continuación se muestra esquemáticamente la estructura de las transferencias USB:

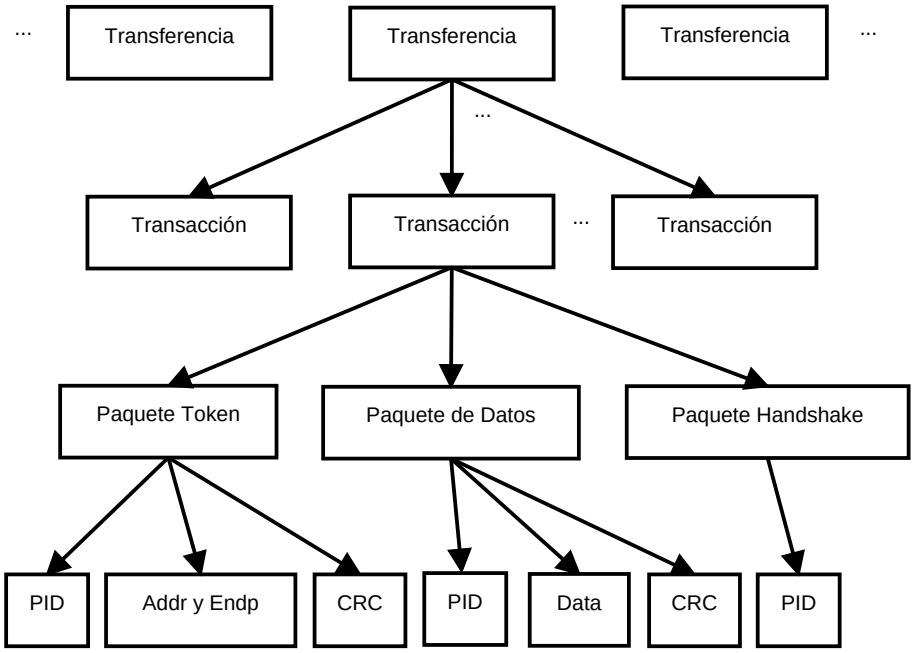


Figura 3.7: Estructura de Transferencias

En las transferencias de tipo Control se tienen tres etapas consistentes de una o más transacciones cada una de ellas. La primera etapa es la de SETUP, la segunda es opcional y es la de DATA, y la última es la de STATUS. Los demás tipos de transferencias poseen una sola etapa y es la de DATA. Es importante notar la diferencia entre fase y etapa, de manera que una etapa está constituida de una a tres de las fases mencionadas anteriormente.

3.2.7. Tiempo entre Paquetes y Transacciones

La especificación de USB 2.0 [8] señala que debe existir como mínimo 88 tiempos de bit[1] entre paquetes continuos de una misma transacción enviados por el host, y como máximo 192 bits. No se define un tiempo máximo entre paquetes de transacciones distintas (el mínimo es de 88 bits también).

Cuando el host responde a un paquete enviado por un dispositivo, el tiempo entre paquetes debe ser de mínimo 8 bits y máximo 192 bits.

3.2.8. Protocolo Ping y Sincronización por PID de Datos

El protocolo Ping es utilizado para evitar enviar datos y utilizar tiempo disponible del bus si es que el endpoint objetivo no tiene espacio o tiempo para recibir los datos. Este procedimiento se aplica solo en transferencias Bulk OUT y Control (etapa de datos OUT).

[1]Es decir, 88 ciclos de reloj a 480 [Mb/s].

Cuando un dispositivo responde a un paquete token OUT con un NAK[1], el host entiende que el endpoint no puede recibir datos momentáneamente. Al recibir el NAK pasa a un estado en que envía paquetes PING para consultar al dispositivo si ya puede recibir datos. En caso que el dispositivo responda con un NYET al paquete token OUT, significa que recibió los datos pero no está listo para recibir más, por lo cual debe enviar paquetes PING en la próxima transacción. Si es que el dispositivo envía un ACK como respuesta al PING, entonces puede recibir datos los cuales son enviados inmediatamente. Si recibe un NAK debe continuar enviando paquetes PING.

Por otra parte, las transacciones Bulk, Control e Interrupt utilizan un procedimiento de sincronización de los paquetes recibidos a través del PID que va en los paquetes de datos. Se utilizan los paquetes DATA0 y DATA1 para llevar a cabo la sincronización. El procedimiento permite forzar el reenvío de un paquete si es que llegó corrupto (con errores) o no llegó al receptor y además mantener la sincronización en caso de que un paquete con acuse de recibo llegue con errores.

3.2.9. Máquinas de Estado de Transacciones

En este apartado se describen las máquinas de estado, implementadas en base a las sugerencias en [8] y [17], de manera de aclarar el funcionamiento de cada tipo de transferencia. Solo se explican las señales relevantes para la caracterización de cada transferencia en términos de las distintas secuencias de estados en función de las entradas.

Los diagramas mostrados en las siguientes secciones corresponden a una versión simplificada donde se omite la información de las salidas. Hay algunas salidas que dependen solo del estado, otras dependen también de las entradas y algunas más particulares dependen además de su valor anterior.

3.2.9.1. Bulk OUT y Control OUT

Las transacciones que van desde host a dispositivo del tipo Bulk y Control funcionan bajo la misma secuencia de eventos. Ambos generan un paquete token OUT, luego un paquete de datos y finalmente esperan alguna respuesta del destino de la transacción. Además, ambos (excepto para la etapa de SETUP) deben soportar el protocolo ping y sincronización a través del PID de datos.

[1] *Not Acknowlegment*, o no acuse de recibo.

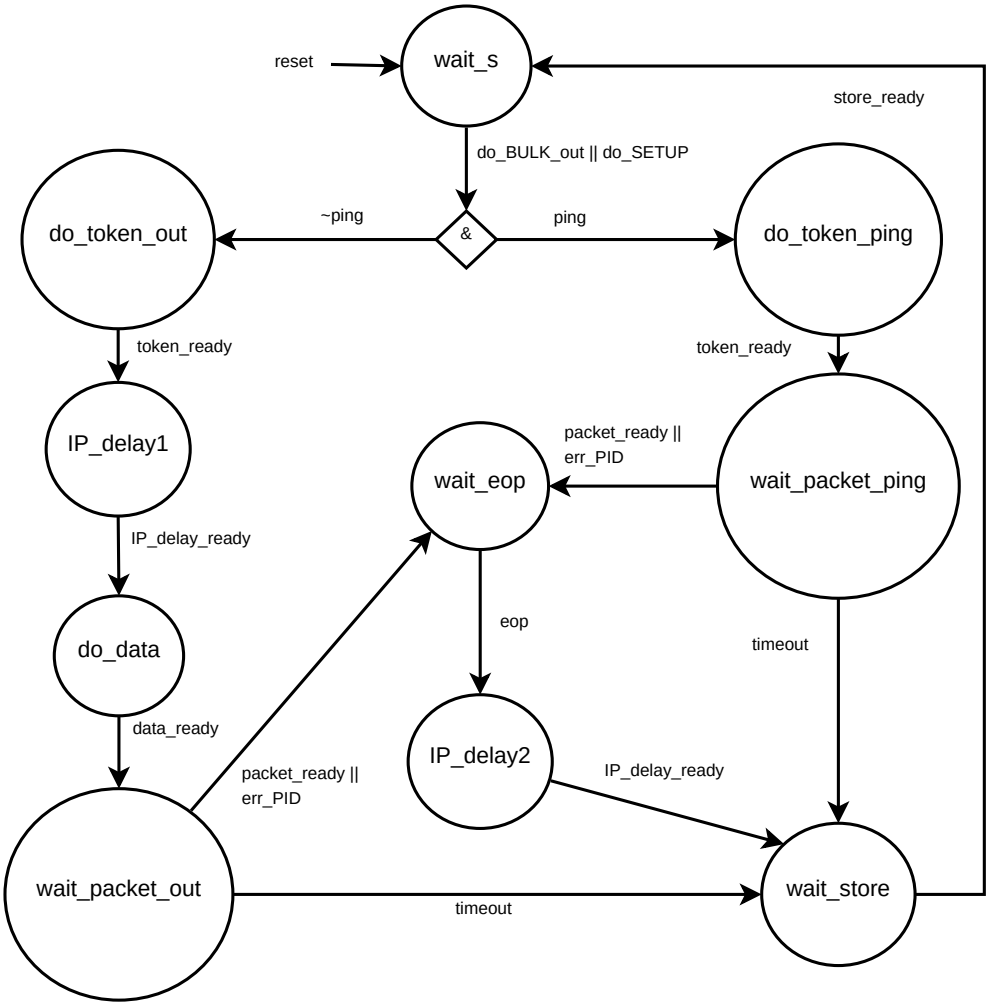


Figura 3.8: Máquina de Estados Bulk Out y Control Out

El símbolo & en el diagrama simplificado de la máquina de estados significa que se requiere de la señal que le llega y alguna de las que le sale para llegar a otro estado, en otras palabras, es un operador lógico *AND*[1]. Por ejemplo, para llegar al estado *do_token_ping* desde *wait_s* se requiere de (*do_BULK_out* **o** *do_SETUP*) **y** *ping*.

La descripción de las entradas a la máquina de estados se muestra en la tabla 3.3 y la de los estados en la tabla 3.4.

[1]Es el operador lógico que requiere que todas sus entradas sean verdaderas para que su salida también lo sea.

Señal	Descripción
reset	Reinicia todas las máquinas de estados y registros a sus valores iniciales.
do_BULK_out	Indica que la transacción a ejecutar es del tipo Bulk en dirección OUT.
do_SETUP	Indica que la transacción a ejecutar es la etapa Setup de una transferencia de control.
ping	Si está en alto entonces se debe generar un paquete ping en vez del solicitado.
token_ready	Generación del paquete token terminada.
data_ready	Generación del paquete de datos terminada.
IP_delay_ready	Tiempo entre paquetes completado.
packet_ready	PID de paquete recibido detectado y decodificado.
err_PID	PID de paquete recibido desconocido o incorrecto para la transacción.
timeout	Se superó el tiempo de espera de un paquete.
eop	Fin del paquete detectado.
store_ready	Resultados y estado de transacción almacenados.

Tabla 3.3: Descripción Entradas Bulk y Control Out

Estado	Descripción
wait_s	Estado inicial donde se mantienen los valores por defecto de salidas y se espera alguna señal para comenzar una transacción.
do_token_out	Generación de paquete token OUT.
do_token_ping	Generación de token para transacción PING.
IP_delay_x	Generación de tiempo entre paquetes.
wait_packet_ping	Estado para espera de respuesta a paquete ping (handshake).
do_data	Generación de paquete de datos.
wait_packet_out	Estado para fase de Handshake.
wait_eop	Espera detección de final del paquete.
wait_store	Espera a que los resultados y estado de transacción sean almacenados.

Tabla 3.4: Descripción Estados Bulk y Control Out

3.2.9.2. Bulk IN, Control IN e Interrupt IN

Las transacciones que van desde un dispositivo al host del tipo Bulk, Control e Interrupt funcionan bajo la misma secuencia de eventos. Primero generan un paquete token IN, luego esperan por un paquete de datos, handshake o un *timeout*[1], y luego generan un ACK (ver tipos de paquetes 3.1) si es que se recibieron los datos correctamente. En el caso en que los datos no se recibieron o están erróneos, se fuerza un *timeout* en el dispositivo al no responder nada.

[1]Es la condición de desconexión por tiempo si es que una respuesta no llega en menos de 736 bits[8] de tiempo.

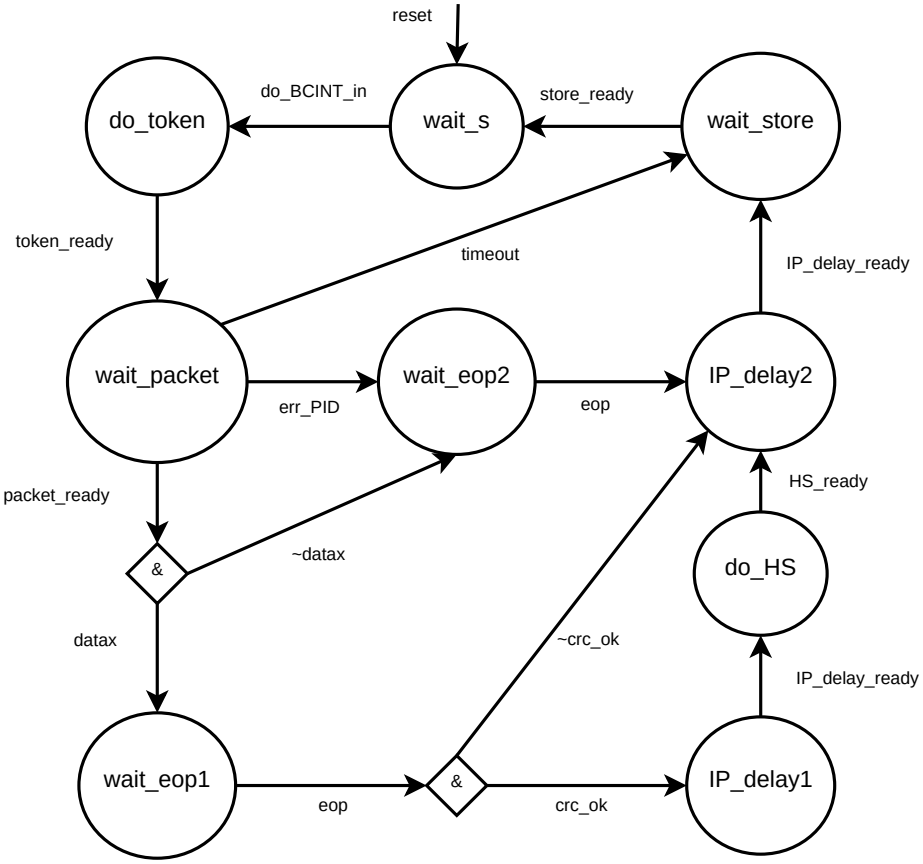


Figura 3.9: Máquina de Estados para Bulk, Control e Interrupt IN

La descripción de las entradas a la máquina de estados se muestra en la tabla 3.5 y la de los estados en la tabla 3.6.

Señal	Descripción
reset	Reinicia todas las máquinas de estados y registros a sus valores iniciales.
do_BCINT_in	Comienza la ejecución de una transacción Bulk, Control o Interrupt con dirección IN.
token_ready	Generación del paquete token terminada.
IP_delay_ready	Tiempo entre paquetes completado.
packet_ready	PID de paquete recibido detectado y decodificado.
datax	Para efectos de la máquina de estados simplificada mostrada, representa las señales data0 y data1, las cuales a su vez indican que uno de los dos paquetes de datos fue recibido.
crc_ok	Indica si el campo CRC que viene en el paquete de datos es coherente con ellos.
HS_ready	Generación y transmisión de paquete ACK completada.
err_PID	PID de paquete recibido desconocido o incorrecto para la transacción.
timeout	Se superó el tiempo de espera de un paquete.
eop	Fin del paquete detectado.
store_ready	Resultados y estado de transacción almacenados.

Tabla 3.5: Descripción Entradas Bulk, Control e Interrupt In

Estado	Descripción
wait_s	Estado inicial donde se mantienen los valores por defecto de salidas y se espera alguna señal para comenzar una transacción.
do_token	Generación de paquete token IN.
IP_delay_x	Generación de tiempo entre paquetes.
do_HS	Generación de paquete ACK en caso de que se haya recibido correctamente un paquete de datos.
wait_packet	En este estado se espera un paquete de datos, handshake o timeout.
wait_eopx	Espera detección de final del paquete.
wait_store	Espera a que los resultados y estado de transacción sean almacenados.

Tabla 3.6: Descripción Estados Bulk, Control e Interrupt In

3.2.9.3. Interrupt OUT

Las transacciones del tipo Interrupt OUT son similares a Bulk y Control OUT salvo que no implementan el protocolo ping y son parte de las transacciones periódicas.

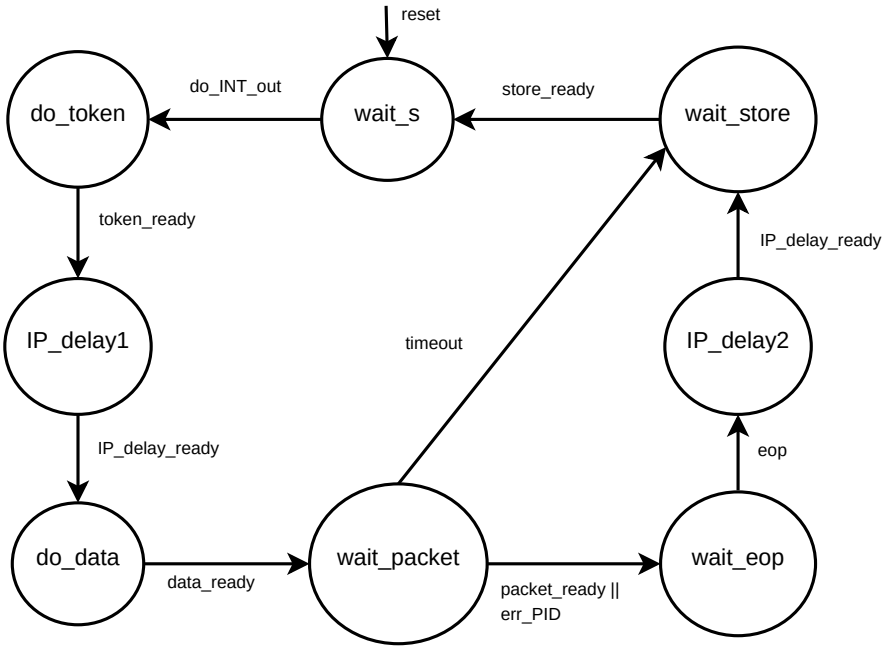


Figura 3.10: Máquina de Estados Interrupt Out

La descripción de las entradas a la máquina de estados se muestra en la tabla 3.7 y la de los estados en la tabla 3.8.

Señal	**Descripción**
reset	Reinicia todas las máquinas de estados y registros a sus valores iniciales.
do_INT_out	Comienza la ejecución de una transacción Interrupt con dirección OUT.
token_ready	Generación del paquete token terminada.
IP_delay_ready	Tiempo entre paquetes completado.
data_ready	Generación del paquete de datos terminada.
packet_ready	PID de paquete recibido detectado y decodificado.
err_PID	PID de paquete recibido desconocido o incorrecto para la transacción.
timeout	Se superó el tiempo de espera de un paquete.
eop	Fin del paquete detectado.
store_ready	Resultados y estado de transacción almacenados.

Tabla 3.7: Descripción Entradas Interrupt Out

Estado	Descripción
wait_s	Estado inicial donde se mantienen los valores por defecto de salidas y se espera alguna señal para comenzar una transacción.
do_token	Generación de paquete token OUT.
IP_delay_x	Generación de tiempo entre paquetes.
do_data	Generación de paquete de datos.
wait_packet	En este estado se espera un paquete de datos, handshake o timeout.
wait_eop	Espera detección de final del paquete.
wait_store	Espera a que los resultados y estado de transacción sean almacenados.

Tabla 3.8: Descripción Estados Interrupt Out

3.2.9.4. Isocrónica IN

Las transacciones isocrónicas IN como no implementan métodos de corrección de errores ni acuse de recepción, simplemente generan el paquete token IN y luego esperan por los datos o timeout.

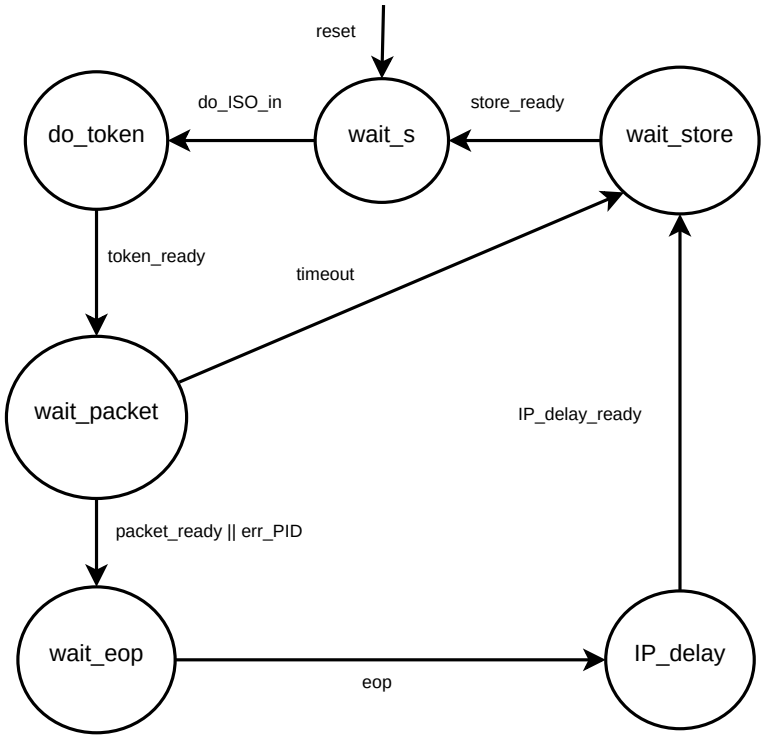


Figura 3.11: Máquina de Estados Isocrónica In

La descripción de las entradas a la máquina de estados se muestra en la tabla 3.9 y la de los estados en la tabla 3.10.

Señal	**Descripción**
reset	Reinicia todas las máquinas de estados y registros a sus valores iniciales.
do_ISO_in	Comienza la ejecución de una transacción isocrónica con dirección IN.
token_ready	Generación del paquete token terminada.
IP_delay_ready	Tiempo entre paquetes completado.
packet_ready	PID de paquete recibido detectado y decodificado.
err_PID	PID de paquete recibido desconocido o incorrecto para la transacción.
timeout	Se superó el tiempo de espera de un paquete.
eop	Fin del paquete detectado.
store_ready	Resultados y estado de transacción almacenados.

Tabla 3.9: Descripción Entradas Isocrónica In

Estado	**Descripción**
wait_s	Estado inicial donde se mantienen los valores por defecto de salidas y se espera alguna señal para comenzar una transacción.
do_token	Generación de paquete token IN.
IP_delay	Generación de tiempo entre paquetes.
wait_packet	En este estado se espera un paquete de datos, handshake o timeout.
wait_eop	Espera detección de final del paquete.
wait_store	Espera a que los resultados y estado de transacción sean almacenados.

Tabla 3.10: Descripción Estados Isocrónica In

3.2.9.5. Isocrónica OUT

Las transacciones isocrónicas OUT no implementan protocolo ping ni requieren esperar de un acuse de recibo, por lo tanto generan un paquete token OUT y luego generan el paquete de datos.

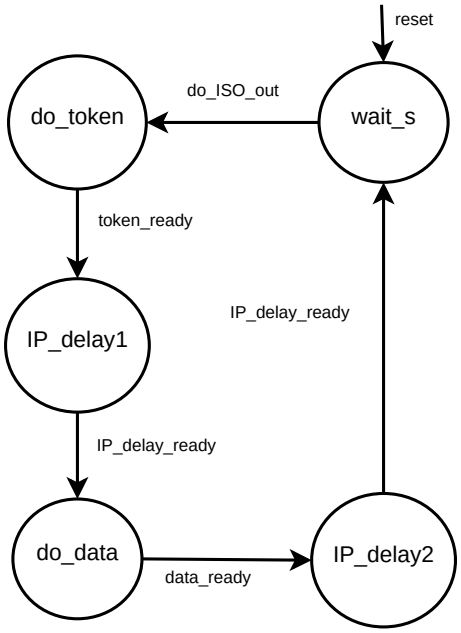


Figura 3.12: Máquina de Estados Isocrónica Out

La descripción de las entradas a la máquina de estados se muestra en la tabla 3.11 y la de los estados en la tabla 3.12.

Señal	**Descripción**
reset	Reinicia todas las máquinas de estados y registros a sus valores iniciales.
do_ISO_out	Indica que la transacción a ejecutar es del tipo Isocrónica en dirección OUT.
token_ready	Generación del paquete token terminada.
IP_delay_ready	Tiempo entre paquetes completado.
data_ready	Generación del paquete de datos terminada.

Tabla 3.11: Descripción Entradas Isocrónica Out

Estado	**Descripción**
wait_s	Estado inicial donde se mantienen los valores por defecto de salidas y se espera alguna señal para comenzar una transacción.
do_toke	Generación de paquete token OUT.
IP_delay_x	Generación de tiempo entre paquetes.
do_data	Generación de paquete de datos.

Tabla 3.12: Descripción Estados Isocrónica Out

3.3. Tratamiento de Señal para TX y RX

El protocolo USB requiere de un tratamiento adicional de la señal que contiene el paquete a ser enviado. Ésto es para asegurar una correcta transmisión y recepción a través del bus. A continuación se describen las características a implementar para USB 2.0 (High-speed).

3.3.1. NRZI

Al momento de transmitir se debe considerar utilizar codificación NRZI (Non Return to Zero Inverted, [8] y [18]). En esta codificación, un "1" es representado por una señal en nivel constante, mientras que un "0" por una transición. Por lo tanto, si se tienen muchos "1's" continuados se tendría una señal sin transiciones por un largo periodo, mientras que muchos "0's" implican que la señal codificada en NRZI cambia de valor por cada bit. En la siguiente figura se puede apreciar gráficamente su funcionamiento:

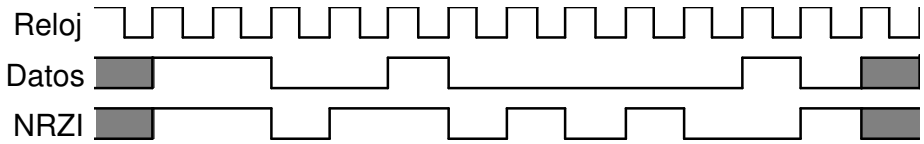


Figura 3.13: Codificación NRZI

Al recibir un paquete éste debe ser decodificado utilizando el procedimiento inverso, en otras palabras, un 0 implica una transición mientras que un 1 significa que la señal se mantenga en el valor anterior.

3.3.2. Bit stuff

Para asegurar una transición en la señal, de manera que el circuito que captura el reloj por el lado de la recepción se mantenga fijo, se debe agregar un "0" cada seis "1's" continuados en el paquete. Estos se aplica desde el SYNC, pasando por el campo PID hasta (incluído) el campo CRC.

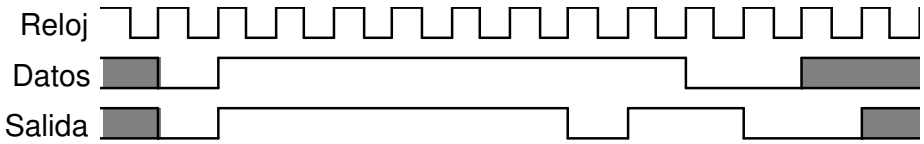


Figura 3.14: Rellenado de Bits

Como se observa en la figura 3.14, cada vez que se requiere agregar un 0 a la señal, implica retrasar todo el proceso en un bit de tiempo. Se requieren *buffers* ([19]) para absorber estos retrasos y continuar el procesamiento de la señal a través de sus distintas etapas.

3.4. Interfaz para Comunicación con Sistema Externo

En esta sección se describen los elementos considerados para que el controlador pueda llevar a cabo las transacciones requeridas y organizadas por el sistema conectado a él. El software controlador conoce las características temporales de todos los tipos de transacciones, por lo cual es capaz de generar ordenadamente listas de tareas a ejecutar.

3.4.1. Memorias Externas

Se considera que el sistema posee dos memorias separadas: una que contiene la descripción de las tareas (transacciones) a ser ejecutadas por el controlador USB; y otra que contiene los datos a ser transmitidos y espacio para almacenar los recibidos.

La memoria de instrucciones tiene ancho de palabras de 32 bits para poder contener la descripción de cada transacción a ejecutar (explicado en 3.4.3). Por otro lado la memoria de datos tiene ancho de palabra de 8 bits para hacer lecturas y escrituras equivalente al número de bytes enviados o recibidos.

3.4.2. Organización de Transacciones

La organización de las transacciones está basada en la Especificación EHCI para USB [20]. El software debe crear listas de transacciones periódicas (para transferencias del tipo Isocrónicas y de Interrupciones) y asíncronas (Voluminosas y de Control).

3.4.2.1. Lista Periódica

Una lista periódica es una estructura de datos que contiene una serie de elementos base que apuntan hacia una lista de descriptores de transacciones isocrónicas y/o *interrupt*, en otras palabras, es una lista de direcciones a otras listas de elementos.

La lista de direcciones es recorrida secuencialmente, una dirección en cada micro-frame, por lo tanto todos los elementos base deben estar en direcciones continuas en memoria. De esta manera se ejecuta una lista de descriptores al comienzo de cada micro-frame. El software

debe asegurar de que cada lista de descriptores no vaya a tomar un periodo mayor al 80 % del micro-frame. La figura 3.15 muestra la estructura de una lista periódica.

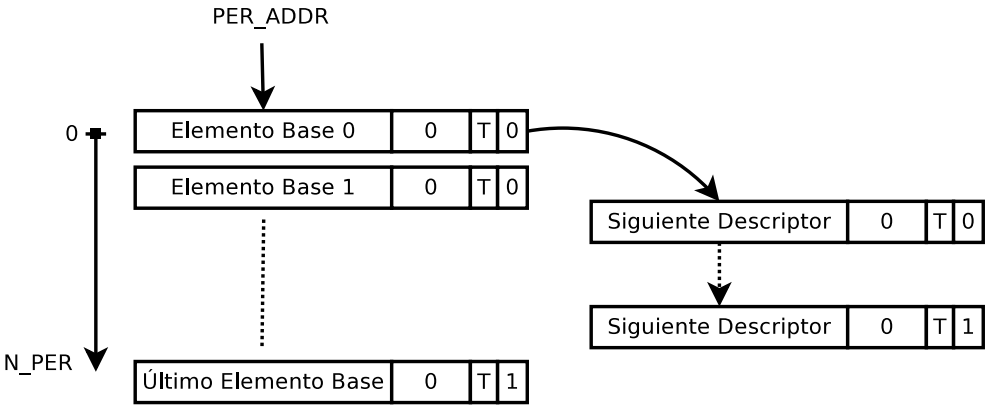


Figura 3.15: Formato Lista Periódica

Los parámetros PER_ADDR y N_PER se explican en la tabla 3.19. Las direcciones a descriptores y elementos bases son de 12 bits, lo cual permite direccionar hasta 4K (4096) elementos. El campo de dirección del último elemento base puede poseer cualquier valor, ya que no se utiliza. Es importante que el campo *Tipo* del último elemento de cada lista que "cuelga" de un elemento base debe estar en 0, para indicar que el siguiente descriptor pertenece a la lista asíncrona y no es isocrónico.

En caso de querer modificar la lista periódica mientras se está ejecutando, se debe deshabilitar colocando en 0 el registro de configuración PER_EN. Luego se debe esperar a que el registro PER_ST también esté en 0 para asegurarse de que el controlador no esté utilizando la lista.

3.4.2.2. Lista Asíncrona

Una lista asíncrona es una estructura de datos que contiene una serie cerrada (el último elemento apunta al primero) de descriptores de transacciones voluminosas y/o de control. En la figura 3.16 se muestra la estructura de una lista asíncrona.

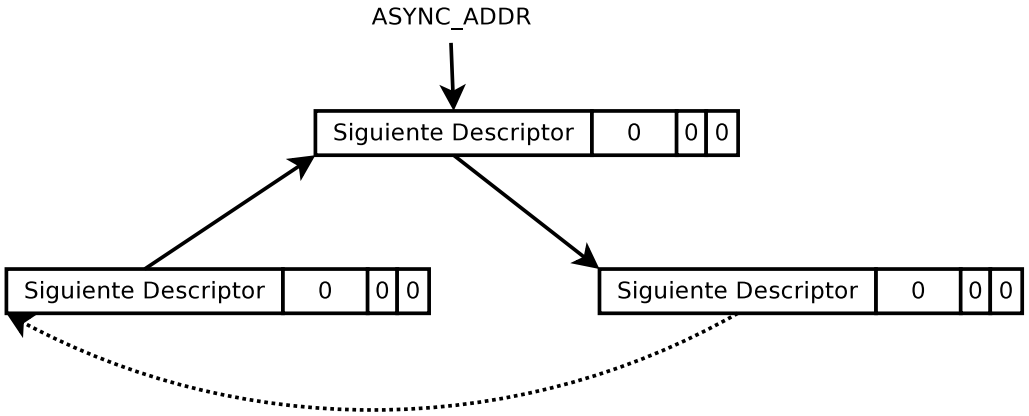


Figura 3.16: Formato Lista Asíncrona

El parámetro ASYNC_ADDR se explica en la tabla 3.19.

En caso de querer modificar la lista asíncrona mientras se está ejecutando, se debe deshabilitar colocando en 0 el registro de configuración ASYNC_EN. Luego se debe esperar a que el registro ASYNC_ST también esté en 0 para asegurarse de que el controlador no esté utilizando la lista.

3.4.3. Formato de Descriptores de Transferencias

El formato de las instrucciones o descriptores de transferencias está basado en la Especificación EHCI para USB [20]. Dada las simplificaciones consideradas para el diseño (sección 3.5), los descriptores son muchos más simples, además de que se implementó otro método para el direccionamiento de los datos de una transferencia a ejecutar.

Los descriptores están formados por palabras de 32 bits, las cuales deben estar en posiciones de memoria continuas para una correcta lectura.

3.4.3.1. Elemento Base

A continuación se muestra el elemento base de una lista periódica:

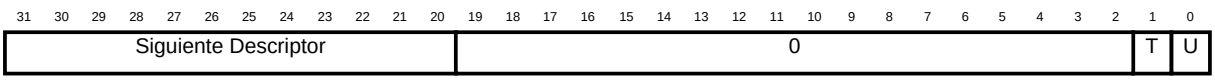


Figura 3.17: Elemento Base

El campo T indica el tipo de descriptor que está apuntando el elemento base. Un “1”

indica que el descriptor es de transferencias isocrónicas, mientras que un “0” es para el resto. El último elemento base debe tener el campo U igual a “1”.

3.4.3.2. Descriptor para Transacciones Bulk, Control e Interrupt

Un descriptor para una transacción *bulk*, *control* o *interrupt* (BCINTd) posee un encabezado (fig. 3.18) y un cuerpo (fig. 3.19). Un encabezado puede tener más de un cuerpo para realizar transferencias de tamaño mayor a 32 [kB] a un mismo *endpoint*, y en otra dirección para el caso Control.

Encabezado del Descriptor

El encabezado mantiene los datos que son comunes a todas las transacciones que se pueden hacer con el descriptor actual.

Figura 3.18: Encabezado Transferencias BCINT

El significado de cada campo del encabezado se muestra en la siguiente tabla:

Campo	Descripción
Siguiente Descriptor	Indica la dirección del siguiente BCINTd o ISOd.
T (Tipo)	Tipo de descriptor de la siguiente transacción. 1: ISOd, 0: BCINTd.
U (Último)	Indica si es el último descriptor de la lista (1) o no (0). Generalmente no se utiliza en listas asíncronas.
Dirección Dispositivo	Dirección del dispositivo al que va dirigida la transacción.
Endpoint	Endpoint al que va dirigida la transacción.
DT (Data Toggle)	Indica el valor del bit de sincronización por PID (sección 3.2.8). Un 1 indica que el PID de los datos en la siguiente transacción debe ser DATA1, un 0 implica DATA0.
P (Ping)	Un 1 indica que se debe enviar un paquete PING (sección 3.2.8) en la siguiente transacción, mientras que un 0 envía los datos.
Descriptor Actual	Indica la dirección del BCINTd actual, de manera de poder retomar la transferencia en cualquier otro momento.
Tamaño Máximo Paquete	Es el tamaño máximo en bytes que soporta el endpoint.
# Err	Es el número de errores acumulados del endpoint. Vuelve a 0 automáticamente si es que hay alguna transacción exitosa hacia el endpoint. Si se acumulan tres errores el endpoint pasa a estado de detención (halt) cambiando el valor del campo halt a 1.
H (Halt)	Si está en 1 el endpoint está detenido y no realiza ninguna transacción. El software puede cambiar de valor este campo.

Tabla 3.13: Descripción Campos Encabezado

Cuerpo

El cuerpo del descriptor posee las características propias de la transacción actual.

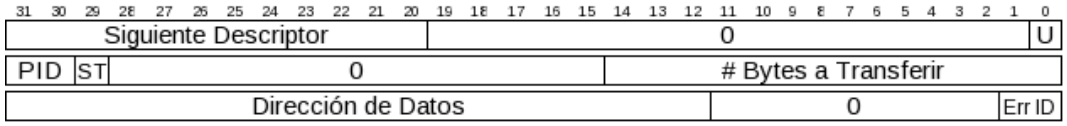


Figura 3.19: Cuerpo Transferencias BCINT

El significado de cada campo del cuerpo se muestra en la siguiente tabla:

Campo	Descripción
Siguiente Descriptor	Indica la dirección del siguiente descriptor BCINTd.
U (Último)	Indica si es el último descriptor de la transferencia y encabezado actual (1) o no (0).
PID	Indica el PID de la transacción actual. 00: OUT, 01: IN, 10: SETUP, 11: Reservado.
ST (Status)	Indica si la transacción corresponde a la etapa de STATUS de una transferencia de Control, de manera de forzar un paquete IN.
# Bytes a Transferir	Indica el número de bytes a transferir. Se pueden enviar hasta 32 [kB] de datos usando un solo descriptor.
Dirección de Datos	Corresponde a la dirección de los datos a enviar o donde se van a almacenar. Se puede direccionar hasta 1 [MB] de datos
Err ID	Contiene la identificación del último error detectado en la ejecución del descriptor. 00: no hubo error, 01: timeout, 10: error de CRC, 11: PID desconocido.

Tabla 3.14: Descripción Campos Cuerpo

3.4.3.3. Descriptor para Transacciones Isocrónicas

Este descriptor (ISOd) permite la ejecución de transacciones isocrónicas en ambas direcciones, señalando el PID de datos correspondiente para la transmisión, almacenando el PID de datos recibido y dejando constancia de los errores detectados en caso de que el software requiera tomar alguna decisión.

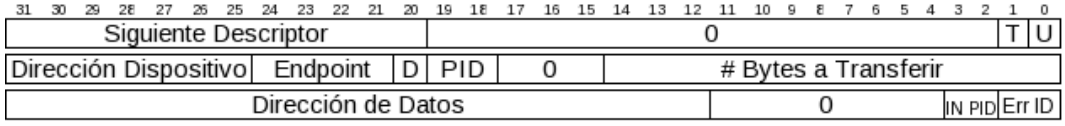


Figura 3.20: Descriptor Transferencias Isocrónicas

El significado de cada campo del descriptor se muestra en la siguiente tabla:

Campo	Descripción
Siguiente Descriptor	Indica la dirección del siguiente BCINTd o ISOd.
T (Tipo)	Tipo de descriptor de la siguiente transacción. 1: ISOd, 0: BCINTd.
U (Último)	Indica si es el último descriptor de la lista (1) o no (0). Generalmente no se utiliza en listas asíncronas.
Dirección Dispositivo	Dirección del dispositivo al que va dirigida la transacción.
D (Dirección)	Señala la dirección de la transacción. 0: OUT, 1: IN.
PID	Corresponde al PID del paquete de datos a utilizar para transmitir. 00: DATA0, 01: DATA1, 10: DATA2, 11: MDATA.
# Bytes a Transferir	Indica el número de bytes a transferir. El software debe asegurarse de no permitir más de 1024 bytes.
Dirección de Datos	Corresponde a la dirección de los datos a enviar o donde se van a almacenar. Se puede direccionar hasta 1 [MB] de datos
IN PID	En este campo se almacena el PID de los datos recibidos. 00: DATA0, 01: DATA1, 10: DATA2, 11: Reservado.
Err ID	Contiene la identificación del último error detectado en la ejecución del descriptor. 00: no hubo error, 01: timeout, 10: error de CRC, 11: PID desconocido.

Tabla 3.15: Descripción Campos ISOd

3.4.4. Máquina de Estados Principal

La máquina de estados que se presenta en este apartado realiza la función de controlar todo el resto de máquinas de estado, activandolas en la secuencia adecuada para que las transferencias sean llevadas a cabo de acuerdo a los requerimientos de USB. La máquina de estados se describe de manera general, no se detallan todas sus señales.

La máquina de estados principal se muestra por partes en las figuras 3.21, 3.22, 3.23 y 3.24, ya que es de mayor complejidad. Las figuras están separadas por grupos de estados origen (de acuerdo a la tabla 3.16), mostrándose así las señales requeridas para alcanzar los demás estados.

Grupo	Estados Origen
1	wait_s, init, wait_sof, fetch_base_element, advance_next y fetch_ISO_desc.
2	fetch_BCINT_head y next_BCINT_head.
3	fetch_BCINT_desc
4	execute, store_s, advance_BCINT y next_BCINT_desc.

Tabla 3.16: Grupos de Estados Origen

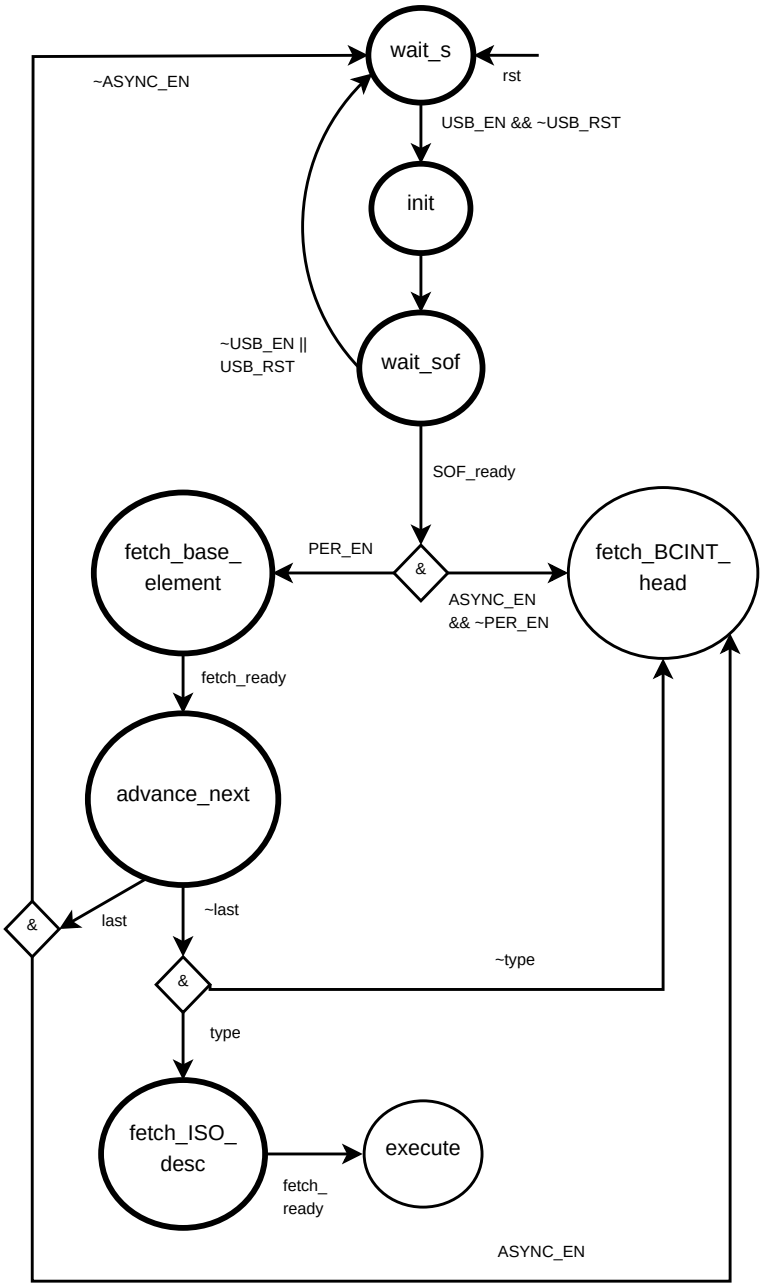


Figura 3.21: Grupo 1 de Estados Origen

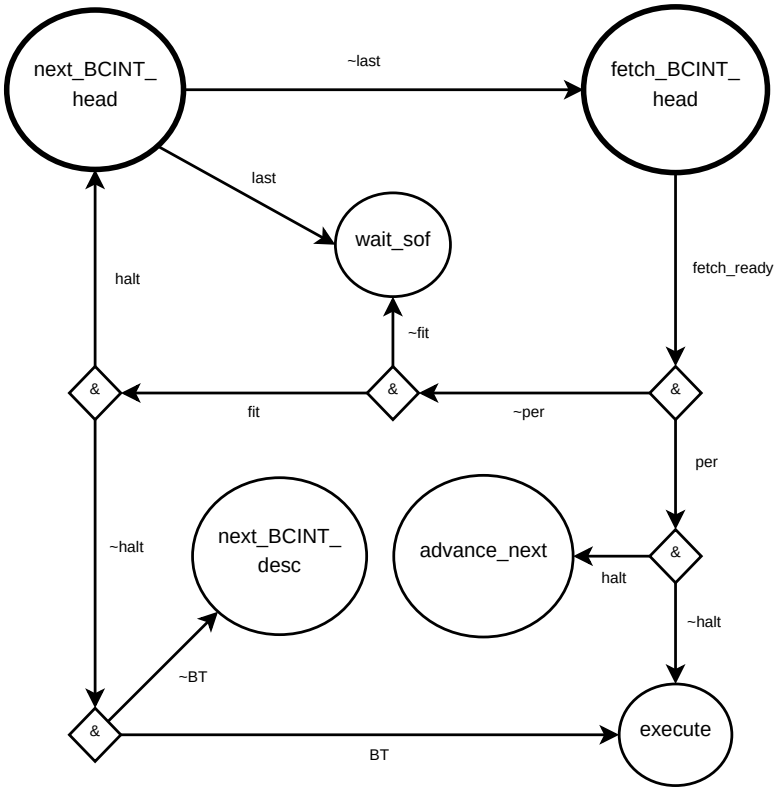


Figura 3.22: Grupo 2 de Estados Origen

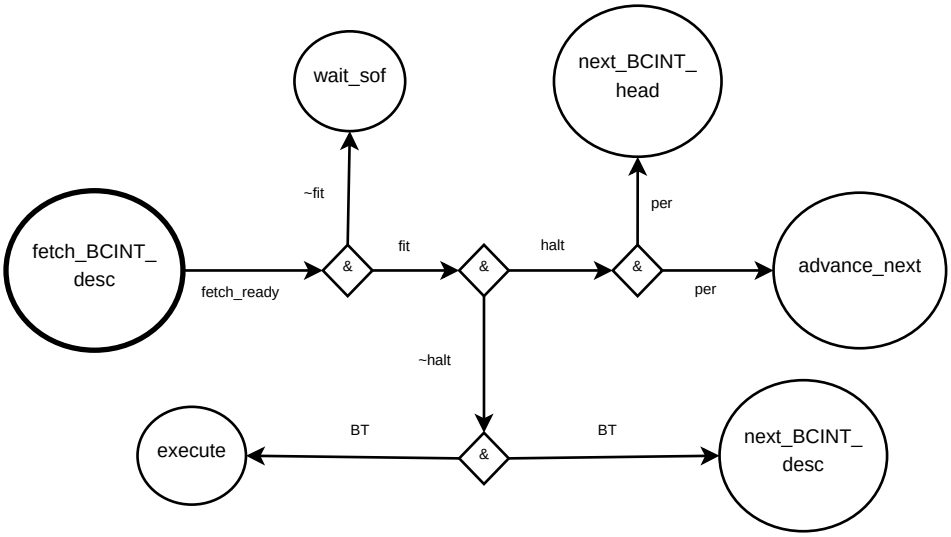


Figura 3.23: Grupo 3 de Estados Origen

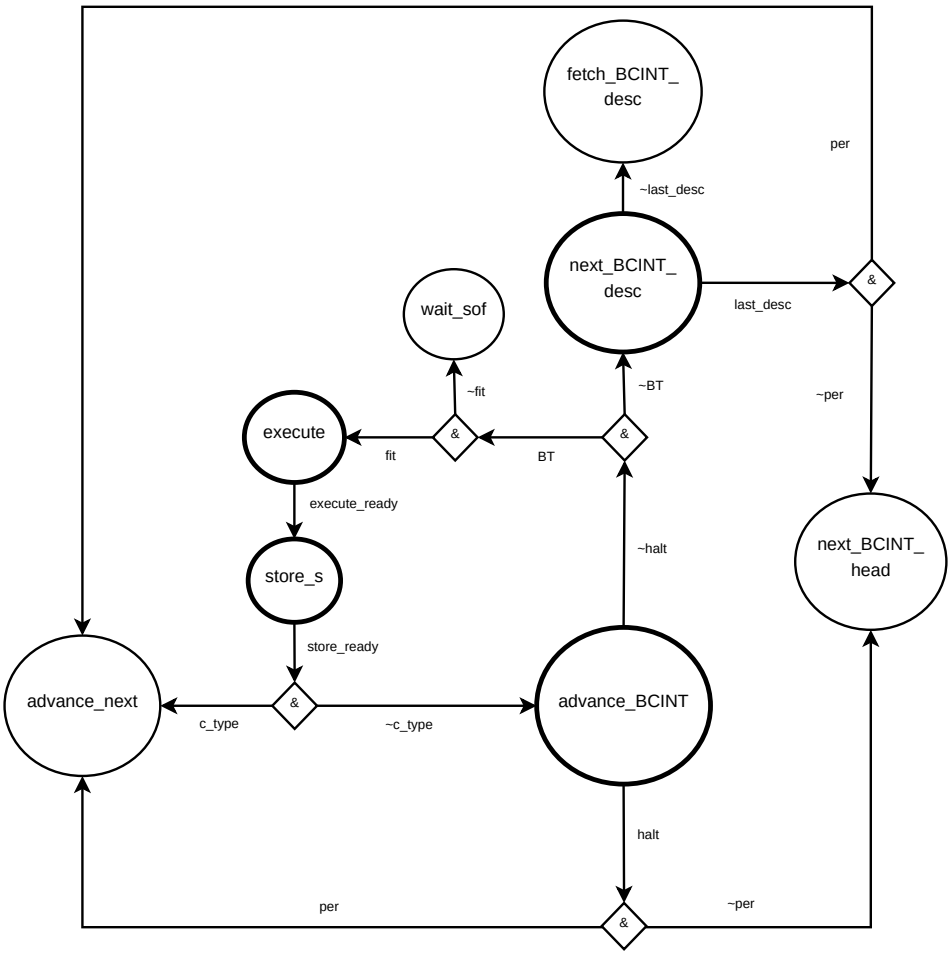


Figura 3.24: Grupo 4 de Estados Origen

La descripción de las entradas a la máquina de estados se muestra en la tabla 3.17 y la de los estados en la tabla 3.18.

Señal	**Descripción**
rst	Reinicia todas las máquinas de estados y registros a sus valores iniciales.
USB_EN	Habilita el controlador USB.
USB_RST	Reinicia el controlador USB.
SOF_ready	Generación de paquete SOF (comienzo de frame) lista.
PER_EN	Habilita lista periódica.
ASYNC_EN	Habilita lista asíncrona.
fetch_ready	Lectura y escritura (en registros locales) de descriptor lista.
last	Indica si es el último elemento de la lista actual.
type	Indica el tipo del siguiente descriptor. 1: isocrónico, 0: el resto.
halt	Indica si la transacción actual está en estado de detención.
fit	Señala si queda suficiente espacio para ejecutar la transacción actual.
per	Registro local que indica si se está en la lista periódica (1) o asíncrona (0).
BT	Cantidad de bytes que queda por transferir en el descriptor actual.
last_desc	Indica si es el último descriptor para el encabezado actual.
c_type	Indica el tipo del descriptor actual. 1: isocrónico, 0: el resto.
execute_ready	Ejecución de transacción actual lista.
store_ready	Almacenamiento de resultados y estado de transacción actual listo.

Tabla 3.17: Descripción Entradas Máquina Estados Principal

Estado	**Descripción**
wait_s	Estado inicial donde se mantienen los valores por defecto de salidas y registros internos y se espera que se salga de condición reset y se habilite el controlador.
init	Suponiendo que los registros de configuración ya están correctamente establecidos, se almacenan en registros locales.
wait_sof	Se espera a que la máquina generadora de comienzos de frame indique que recién terminó un SOF para poder empezar a ejecutar transacciones organizadamente en cada micro-frame. Si está habilitada, se comienza por la lista periódica.
fetch_base_element	Realiza la lectura de un elemento base en la memoria de instrucciones.
advance_next	Verifica si el descriptor (o elemento base) leído anteriormente indica que es el último de la lista. Si lo es, pasa a ejecutar la lista asíncrona si está habilitada, sino, sigue ejecutando la lista periódica realizando lecturas de acuerdo al tipo de descriptor.
advance_BCINT	Analiza si el endpoint actual se encuentra detenido (condición halt), si es que ya se transfirieron todos los bytes para el descriptor, y si es que queda suficiente tiempo para la transmisión de la información que queda.
next_BCINT_desc	Analiza si el descriptor actual es el último para el encabezado. Si lo es, pasa a un estado que analiza el siguiente elemento para el tipo de lista actual, sino, pasa a un estado que realice la lectura del siguiente descriptor para el encabezado actual.
next_BCINT_head	Al terminar de ejecutar un descriptor en una lista asíncrona, verifica si quedan más elementos a ejecutar, si no quedan vuelve al estado wait_sof.
fetch_ISO_desc	Realiza la lectura de un descriptor para transacciones isocrónicas.
fetch_BCINT_head	Realiza la lectura de un descriptor (completo) para transacciones voluminosas, control o de interrupciones.
fetch_BCINT_desc	Realiza la lectura de un descriptor para el encabezado actual.
execute	Ejecuta el descriptor actual.
store_s	Almacena los resultados y estado de la transacción del descriptor actual.

Tabla 3.18: Descripción de los Estados

La máquina de estados requiere de los siguientes registros de configuración:

Registro	**Descripción**
USB_EN	Habilita (1 lógico) o deshabilita (0 lógico) el controlador.
PER_EN	Habilita (1) o deshabilita (0) la lista de elementos periódicos (explicado más adelante).
ASYNC_EN	Habilita (1) o deshabilita (0) la lista de elementos asíncronos.
PER_ADDR	Indica la dirección en memoria externa del primer elemento de la lista base periódica.
ASYNC_ADDR	Indica la dirección en memoria del externa primer elemento de la lista asíncrona.
USB_RST	Reset del controlador por software.
PER_ST	Registro que indica si efectivamente (1) se está trabajando con la lista periódica o no (0).
ASYNC_ST	Registro que indica si efectivamente (1) se está trabajando con la lista asíncrona o no (0).
N_PER	Indica la cantidad de elementos base de la lista periódica.

Tabla 3.19: Registros de Configuración

3.5. Simplificaciones Realizadas

A continuación se listan las simplificaciones que se realizaron para hacer un diseño más compacto, de menor complejidad y que permitiese terminar en un tiempo adecuado:

- **Solo High Speed.**
  Se limitó la funcionalidad del controlador a solo aceptar dispositivos y hubs que trabajen en High Speed, es decir, no se implementó compatibilidad con las versiones anteriores de USB. Esta simplificación permite evitar el uso de los paquetes PRE, ERR y SPLIT. Además se simplifican las máquinas de estado que representan los modos de transferencia, ya que en general el protocolo no es el mismo para Low/Full Speed respecto a High Speed.

- **No se incluye el Hub de raíz.**
  La implementación del hub de raíz se dejó como propuesta, de manera de concentrar los esfuerzos de diseño en el controlador en sí. Es bueno dejar constancia de que como el sistema objetivo para el controlador diseñado es High Speed, entonces el diseño del hub es mucho más simple, porque no se deben administrar distintas velocidad con paquetes especiales para los distintos tramos del bus USB.

- **No se especifica circuito análogico asociado a la interfaz con el bus.**
  Como el objetivo es mostrar el flujo de diseño de un circuito digital de mediana complejidad, entonces la parte analógica no se consideró. Las funciones que el chip analógico debería cumplir son básicamente las siguientes:
  - Extracción del clock de los paquetes.
  - Detección de paquetes (envolvente).

  - Conversión de voltajes para realizar una señalización diferencial Full Duplex en el bus.
  - Tranmisión de VBUS y GND para proveer la potencia adecuada al bus.

- **Interfaz para configuración del controlador no especificada.**

  La manera de acceder a los registros para configurar y conocer el estado el controlador se dejó abierta, de manera de poder integrarse a cualquier sistema objetivo. Los registros de configuración y control se describen en la tabla 3.19.

  Además, no se especifica ningún protocolo externo para el acceso en términos de lectura/escritura de los registros de configuración. Las simulaciones consideran que los registros ya se encuentran escritos o se modifican sus valores mediante el archivo de simulación.

- **Una memoria para datos de palabras de ancho 8 bits, y una para instrucciones de 32 bits.**

  Con el fin de simplificar y acortar el tiempo de diseño, se supuso que el sistema objetivo consta con memorias de instrucciones y datos separadas. La memoria de instrucciones debe poseer un ancho de 32 bits, mientras que la de datos 8 bits. La especificación de la cantidad de datos a enviar siempre es en cantidad de bytes, por lo cual con una memoria de datos de palabras de 8 bits permite obtener esos datos realizando la misma cantidad de lecturas que cantidad de bytes. Algo similar ocurre al recibir paquetes de datos, donde la recepción se realiza separando el paquete en 8 bytes y almacenándolos inmediatamente en la memoria. Para ambas memorias la lectura se realiza levantando una señal durante un periodo de reloj, y se espera que los datos estén disponibles en el siguiente periodo.

Capítulo 4

Diseño del Controlador USB 2.0

En esta sección se describe el flujo de diseño mostrado en la figura 2.2, aplicado al desarrollo de un controlador USB 2.0 simplificado de acuerdo a las especificaciones y modelamiento descrito en el capítulo anterior. Primero se muestra el diagrama de bloques general que representa las principales funciones que implementan el controlador, para luego describir las etapas del diseño hasta la obtención del *layout* final del circuito.

4.1. Diagrama de Bloques

El diagrama de bloques (Fig. 4.1) que se presenta en este apartado corresponde a una representación funcional general del diseño del controlador. El diagrama mostrado fue construido en base a los módulos Verilog finales, no es un modelo previo a la descripción RTL. El modo a proceder fue básicamente rescatar las funciones esenciales que el controlador debe realizar[1] y desarrollar un diagrama de bloques bosquejo para tener un punto de partida para el paso de la descripción RTL del diseño.

El "Sistema Objetivo" corresponde a la plataforma a la cual se integraría el controlador, mientras que el circuito analógico es aquel que implementa las funciones descritas en la sección 3.5, además de otras mencionadas en la especificación [8]. Ninguno es caracterizado de manera profunda en el desarrollo de esta memoria.

[1]Las funciones fueron extraídas de la Especificación USB 2.0 [8].

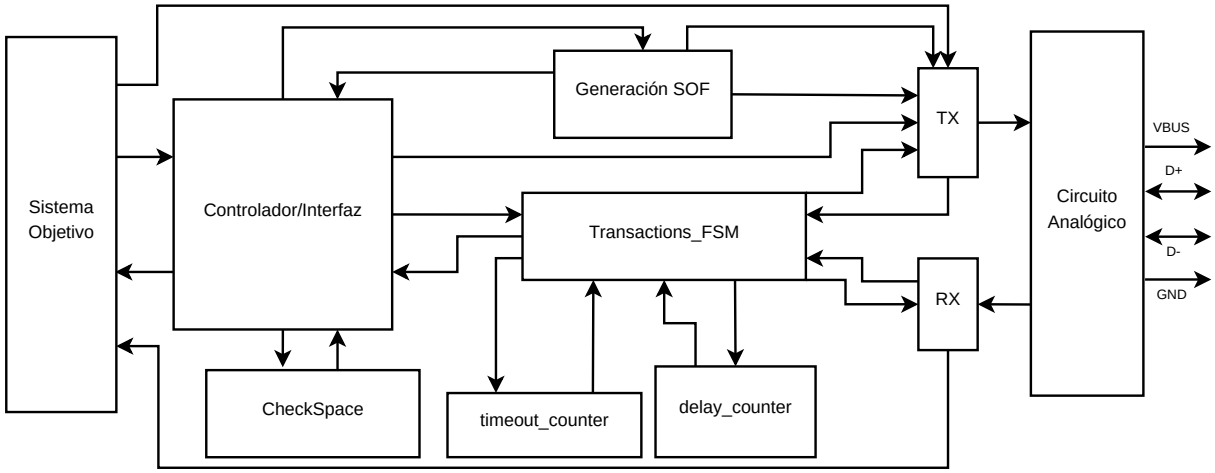


Figura 4.1: Diagrama de Bloques del Diseño

Las funciones realizadas por cada bloque se describen a continuación.

Controlador/Interfaz

Este bloque se encarga de:

- Comunicarse con el sistema objetivo para lectura y escritura en memorias.
- Ejecutar transacciones en el orden definido por el software, procurando cumplir los requerimientos de un micro-frame.
- Obtener información necesaria de los descriptores de transacciones para ejecutarlas y almacenar resultados coherentes con el tipo de transferencia.

CheckSpace

La función de este bloque es determinar si queda espacio en el micro-frame actual para realizar otra transacción de la lista asíncrona de transferencias, basado en el tiempo que queda y el máximo que podría ocupar la transacción.

Generación SOF

Este bloque realiza las siguientes funciones:

- Mantener un contador que indique el comienzo de un nuevo micro-frame y el tiempo que queda de él.
- Generar el número del frame actual.
- Generar las señales necesarias para la transmisión de un paquete SOF al comienzo del micro-frame, con el número de frame correspondiente.

Timeout_counter

Este bloque se encarga de contar el tiempo que pasa después de la transmisión de un paquete token IN. En caso de superar el tiempo de 736 bits se indica que ocurrió un *timeout*.

Delay_counter

Este bloque se encarga de evitar que se genere un paquete antes de que 88 bits de tiempo (ver sección 3.2.7) hayan ocurrido con respecto al paquete anterior.

Transactions_FSM

Este bloque está constituido por todas las máquinas de estado que controlan los distintos tipos de transferencias, por lo tanto su función es generar, de acuerdo al protocolo USB, las señales necesarias para la transmisión de paquetes y respuestas adecuadas respecto a la descripción de una transacción y la respuesta de un dispositivo. En otras palabras, este bloque es responsable de tomar los datos entregados por el bloque "Controlador/Interfaz" y ejecutar la siguiente transacción.

TX

La función de este bloque es generar los paquetes a transmitir solicitados por el bloque "Transactions_FSM". Ésto consiste en crear cada campo que pertenece al paquete, para luego procesarlo al agregar los bits de rellenado (sección 3.3), EOP y codificación NRZI.

RX

Este bloque se encarga de la recepción de los paquetes enviados por los dispositivos conectados al bus. Entre sus funciones están las siguientes:

- Decodificación NRZI del paquete.
- Decodificación del PID.
- Quitar bits de relleno.
- Verificar campo CRC en caso que sea un paquete de datos.
- Ordenar datos en bytes para correcto almacenamiento en memoria.

4.2. Descripción RTL

Esta etapa corresponde a la descripción de la funcionalidad del diseño en términos de la transferencia de datos entre registros. Se utilizó el lenguaje de descripción de hardware Verilog para este nivel de abstracción en el diseño del controlador.

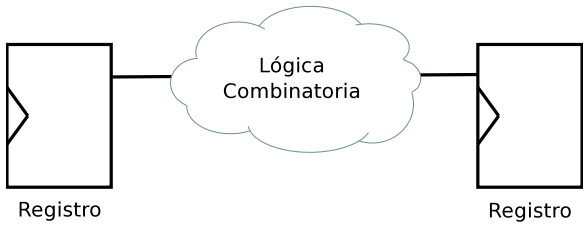


Figura 4.2: Nivel de Transferencia de Datos entre Registros

La funcionalidad de cada uno de los bloques mostrados en el diagrama de la sección anterior es implementada en varios módulos que se explican en este apartado. Todos los módulos están contenidos en otro, denominado *Top* o módulo maestro.

Para algunos módulos se muestra pseudo-código, a modo de ejemplificar su implementación a nivel RTL y dar una visión de lo que significa esta etapa.

4.2.1. Controlador/Interfaz

Este bloque está compuesto por los módulos que se explican a continuación:

4.2.1.1. Fetch

La función de este módulo es generar las señales requeridas para la lectura de descriptores de transacciones en memoria, y luego su escritura en el banco de registros.

Para realizar una lectura se levanta la señal *read* durante un ciclo de reloj, y en el ciclo siguiente se levanta la señal *write* para escribir[1] la palabra en los registros locales. Dependiendo del tipo de descriptor a leer es el número de veces que se levantan las señales mencionadas anteriormente. Por ejemplo, para leer un descriptor BCINT completo se levantan ambas señales 6 veces (6 palabras de 32 bits), y para el cuerpo de un descriptor BCINT, 3 veces como se muestra en la figura 4.3.

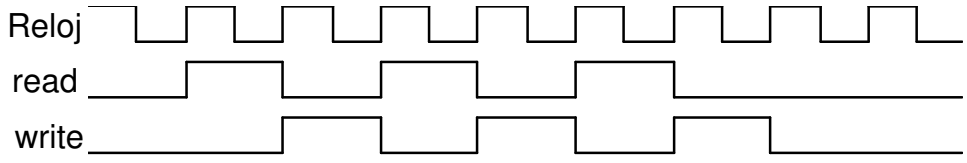


Figura 4.3: Fetch de Cuerpo de Descriptor BCINT

Las direcciones a la memoria de instrucciones y banco de registros también son manejadas por este módulo, dada una dirección base como entrada.

4.2.1.2. Execute

Este módulo genera la información (señales), en base a la instrucción recién leída, para una correcta formación de paquetes para la transacción a ejecutar. Una vez generada la información, se comienza la máquina de estados correspondiente al tipo de transferencia del descriptor y la lista que se está ejecutando. Cuando termina la ejecución de la transacción, ya sea con éxito o errores, se actualiza el valor de bytes a transferir (BT) y la dirección de la memoria de datos en donde se debería realizar la siguiente lectura.

La información generada es:

- Token PID, dependiendo del tipo de descriptor y sus campos de dirección y PID.
- Data PID, dependiendo del tipo de descriptor y sus campos PID, DT y *Status*.
- Número de bytes a transferir, dependiendo del tipo de descriptor y sus campos BT y tamaño máximo de paquete.

La máquina de estados que implementa este módulo es la siguiente:

[1]suponiendo que la información está disponible en ciclo siguiente de levantar la señal *read*.

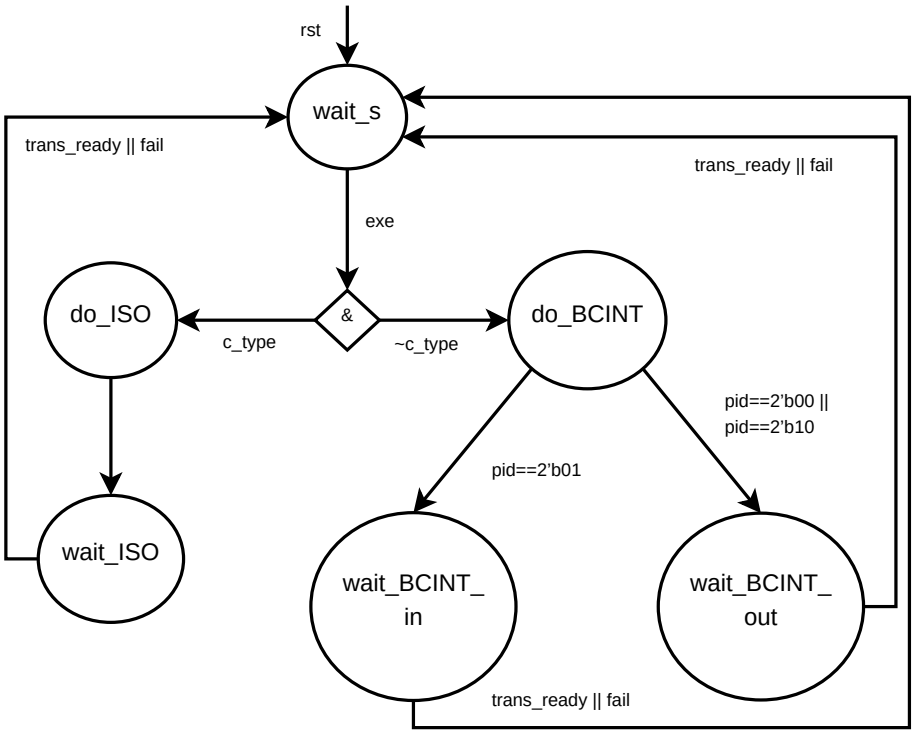


Figura 4.4: Máquina Estados *execute*

Las principales salidas en función de los estados y entradas se describen en la tabla 4.1. El estado y entradas mostradas para cada salida son las requeridas para llevarlas a un "1" lógico.

Salida	**Estado**	**Entradas**	**Descripción**
do_ISO_in	do_ISO	$dir = 1$	Comienza una transacción isocrónica dirección IN.
do_ISO_out	do_ISO	$dir = 0$	Comienza una transacción isocrónica dirección OUT.
do_INT_out	do_BCINT	$pid = 00, per = 1$	Comienza una transacción de interrupción dirección OUT.
do_BULK_out	do_BCINT	$pid = 00, per = 0$	Comienza una transacción voluminosa o de control dirección OUT.
do_SETUP	do_BCINT	$pid = 10$	Comienza una transacción de control etapa Setup.
do_BCINT_in	do_BCINT	$pid = 01$	Comienza una transacción voluminosa, de control o interrupciones dirección IN.

Tabla 4.1: Principales Salidas Máquina Estado *execute*

4.2.1.3. Store

Este módulo se encarga de generar las direcciones a la memoria de instrucciones, para almacenar correctamente las palabras del descriptor que recién se ejecutó, que contengan campos de resultados y estado de la transferencia. Las palabras que se guardan corresponden a la tercera en el caso de un ISOd, y de la segunda a sexta en un BCINTd. Se levanta una

señal *write* durante el proceso para indicarle a la memoria que se quiere almacenar los datos que se le presentan en la entrada en la dirección indicada.

4.2.1.4. Main_FSM

Este módulo corresponde a la descripción RTL de la máquina de estados (Mealy[1]) descrita en la sección 3.4.4. Ésta hace uso principalmente de los módulos *fetch*, *execute* y *store*, para una ejecución de acuerdo a la organización planteada por el *software*.

Las principales señales que permiten recorrer las listas de acuerdo a lo que ocurra en tiempo real y lo requerido por el software son:

- c_type: Indica el tipo del descriptor cargado en los registros.
- type: Indica el tipo del siguiente descriptor en la lista.
- fit: Indica si queda tiempo o no disponible para la ejecución de una transacción.
- last: Indica si es el último descriptor de una lista.
- BT: Cantidad de bytes que quedan por transferir.

La máquina de estados también se encarga de actualizar el valor de las direcciones base a descriptores (encabezado y cuerpo para el caso de un BCINTd), para así poder almacenar y continuar la ejecución de descriptores que tienen más de un cuerpo y/o requieran transferir datos de tamaño superior al tamaño máximo.

Las principales salidas se describen en la siguiente tabla:

Salida	**Descripción**
per	Un "1" lógico indica que la lista que se está ejecutando es la periódica, mientras que un "0", la lista asíncrona.
fetchx	Comienza la lectura de algún descriptor. x=1: elemento base, x=2: ISOd, x=3: BCINTd, x=4: cuerpo BCINTd.
en_sof	Habilita el módulo *SOFcounter*.
exe	Comienza ejecución de una transacción.
save	Almacena los resultados y estado de la transacción recién ejecutada.
load_data_addr	Carga una dirección en el bus de direcciones de la memoria de datos.

Tabla 4.2: Salidas Máquinas de Estados de Principal

[1]Una máquina de estados Mealy es aquélla donde las salidas dependen del estado actual y de las entradas.

4.2.1.5. Register_bank

El módulo *register_bank* es un banco de 6 registros de palabras de 32 bits para poder guardar el descriptor que se va a ejecutar. Además, realiza la decodificación del descriptor al generar señales de salida correspondientes a cada campo de él (excepto de aquellos que son resultados de transacción de interés particular del software, como IN PID).

Los ISOd's se almacenan en los primeros tres registros al igual que un encabezado de un BCINTd, mientras que el cuerpo se guarda en los últimos tres.

4.2.1.6. Data_ram_bus

La función de este módulo es cargar una dirección en el bus de direcciones de la memoria de datos una vez que se haya leído un descriptor o la máquina de estados principal lo requiera (por ejemplo cuando se vuelve intentar una transacción). Además, se encarga de avanzar el cursor o bus de direcciones en una posición más adelante cada vez que se escribe o lee un byte de datos.

4.2.2. CheckSpace

El diseño de este módulo está basado en el algoritmo de aproximación a mejor ocupación de un micro-frame sugerido en [20]. El algoritmo toma en cuenta el peor caso de transacción, donde se tiene la mayor cantidad de tiempo utilizado en campos que no son datos, tiempo entre paquetes y de espera de respuesta. Se utilizan esos valores de peor caso además del tiempo que agrega cada bit de llenado en el peor caso (aproximadamente 4 bits de relleno cada 3 bytes de 1's) más el tamaño de paquete máximo para el endpoint para determinar el tiempo que tomaría ejecutar la transacción y así poder compararlo con los bytes que quedan en el micro-frame.

4.2.3. Generación SOF

La implementación de la funcionalidad de este bloque está dividida en dos módulos: El primero está compuesto por contadores para la cantidad de bits que quedan en el micro-frame y el número de frame; el segundo es la máquina de estados que genera el paquete SOF.

4.2.3.1. SOFcounter

Las funciones que realiza este módulo son las siguientes:

- Indica el tiempo restante del micro-frame en bits.
- Indica el comienzo de un nuevo micro-frame (señal *new_uF*) cuando el contador anterior llega a 0.
- Entrega el número del frame actual durante sus 8 micro-frames.

SOFcounter es habilitado por la máquina de estados principal cuando ésta pasa a al estado *wait_sof* (ver sección 3.4.4), y deshabilitado en los estados *wait_s* e *init*.

4.2.3.2. SOF_FSM

Esta máquina de estados (Mealy) se activa cuando el módulo *SOFcounter* levanta la señal *new_uF*. Luego pasa al estado *do_SOF* en donde, además de activar el generador de paquetes token en el módulo TX, señala que el *token* a crear es del tipo SOF y que debe utilizar el número de frame en vez de un número de dispositivo y *endpoint*. Después de haber generado el paquete SOF, se fuerza el tiempo entre paquetes en el estado *IP_delay* (Fig. 4.5).

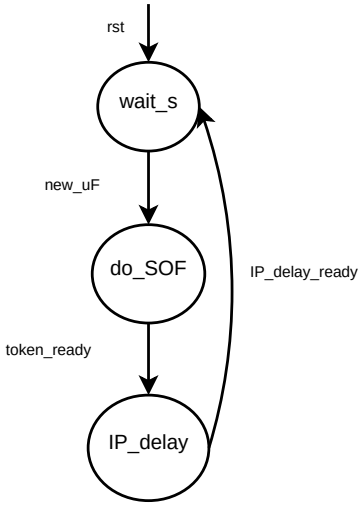


Figura 4.5: Máquina Estados Paquete SoF

4.2.4. Timeout_counter

Este módulo es básicamente un contador que inicialmente tiene un valor de 735 y cuando es habilitado comienza una cuenta regresiva hasta 0, momento en el que levanta la señal *timeout*. El módulo *Transactions_FSM* es el encargado de habilitar el contador cuando espera un paquete desde algún dispositivo.

4.2.5. Delay_counter

En la transmisión de información a través del bus se debe respetar cierto tiempo entre paquetes como se explica en la sección 3.2.7. Este módulo se encarga de generar ese tiempo, cuando es habilitado, a través de un contador hacia atrás (cuenta regresiva) desde un valor inicial de 88. De esta manera se asegura de que el tiempo entre paquetes esté dentro del rango descrito en la especificación USB 2.0 [8].

4.2.6. Transactions_FSM

Este módulo consiste en 5 máquinas de estado (todas Mealy), de acuerdo a la sección 3.2.9, que controlan los distintos tipos de transferencias. Se encarga de levantar las señales que inician la generación y configuración de todos los tipos de paquetes, tiempo entre paquetes, registros que almacenan el estado y caracterización de la transacción actual y fin de transacción.

Cada máquina de estados se inicia con una señal diferente, las cuales son generadas por el bloque *execute*. Las principales salidas de este módulo se describen en la tabla 4.3.

Salida	**Descripción**
make_token	Comienza la generación de un paquete *token*.
make_data	Inicia la generación de un paquete de datos.
en_IP_delay	Habilita el módulo *delay_counter* para generar el tiempo entre paquetes (sección 3.2.7).
make_HS_sel	Comienza la generación de un paquete *handshake* ACK.
sel	En conjunto con la señal *sof* y *make_HS_sel* selecciona el PID adecuado y los siguientes 11 bits para el paquete token a generar.
tod	Selecciona los datos que entran al módulo *PHY_TX* dependiendo si al paquete a generar es *token* (1) o de datos (0).
do_halt	Indica si el *endpoint* se debe colocar o no en estado de detención.
change_dt	Si tiene valor "1" lógico, invierte el valor del campo DT del descriptor actual.
save_data	Indica si los datos recibidos deben ser almacenados o no.
en_RX	Habilita el módulo *timeout_counter* para determinar condición de *timeout*.
trans_ready	Indica que la transacción terminó exitosamente.
fail	Indica que la transacción terminó con errores.
set_ping	Indica el valor del campo Ping de un BCINTd que debería almacenarse cuando termine la transacción.
err_count_out	Indica el número de errores al terminar la transacción.
err_ID	Indica el tipo de error con el que terminó la transacción.
ISOin_pid	Indica el PID de los datos recibidos en una transacción isocrónica dirección IN.

Tabla 4.3: Salidas Principales Máquinas de Estados de Transacciones

Las salidas son activadas en distintas secuencias dependiendo del tipo de transacción, descriptor del *endpoint* y del comportamiento del dispositivo objetivo, de acuerdo a las máquinas de estado presentadas en la sección 3.2.9. No se detallan las salidas en las máquinas de estado en este documento, pero la implementación se puede ver en el código Verilog (en el CD adjunto) de ellas.

4.2.7. TX

Este bloque se encarga de la elaboración de todos los tipos de paquetes como sea requerido por las máquinas de estado de las transferencias, y luego transmitirlas de acuerdo a lo descrito en el Capítulo 3.

Las distintas funciones que debe implementar este bloque para la transmisión de paquetes son divididas en sub-bloques, como se muestra en el diagrama de la figura 4.6.

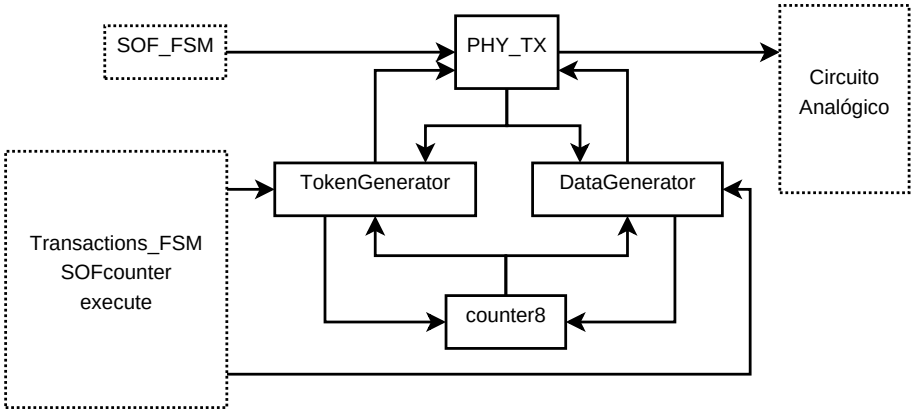


Figura 4.6: Diagrama de Bloques Transmisión

A su vez, cada uno de los bloques a veces requiere ser implementado en más de un módulo para separar claramente las distintas etapas del proceso, como se explica en los siguientes apartados.

4.2.7.1. TokenGenerator

Este módulo se encarga de la generación de los paquetes tipo token (incluído el paquete SOF) y handshake ACK, en función de las características que estos deberían tener según el descriptor que se está ejecutando. Se utiliza máquina de estados separadas para los paquetes token y ACK. En sí, este módulo está constituído por otros 5 que se explican a continuación.

4.2.7.1.1. TokenFSM

La generación de paquetes token y SOF se lleva acabo por la máquina de estados (Mealy) codificada en este módulo. La máquina de estados está en espera de la señal *make_token* para salir del estado inicial, para luego cargar el campo PID, dirección de dispositivo y *endpoint* (o número de frame) y el campo CRC (de 5 bits) en un banco de registros de 8 bits. Una vez cargados los campos, éstos son extraídos en orden (partiendo por los 4 bytes de SYNC) y enviados al módulo *PHY_TX* para procesarlos bit a bit.

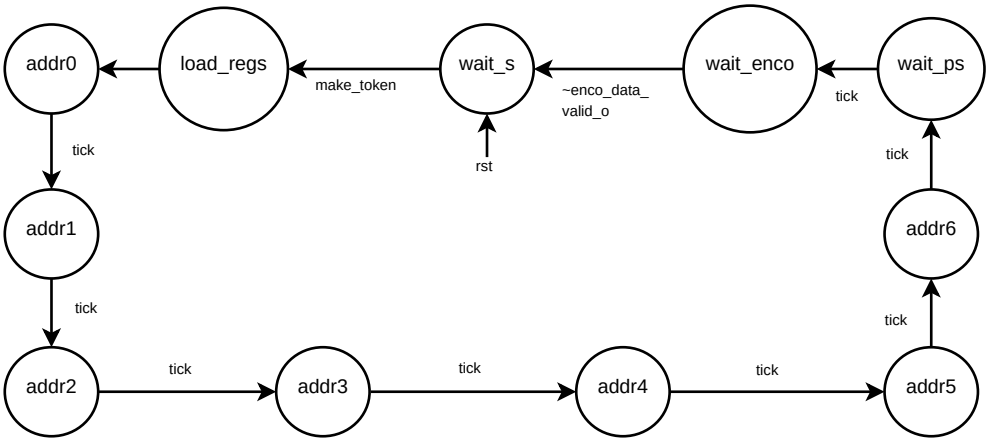


Figura 4.7: Máquina de Estados Generación Paquete Token

La señal *tick* es la encargada de realizar una lectura de un registro de 8 bits del banco en la dirección indicada en cada estado. Esta señal se levanta durante un ciclo cada 8 ciclos de reloj de manera de darle tiempo al módulo *PHY_TX* para procesar cada bit de los registros.

4.2.7.1.2. HsFSM

La generación del paquete de acuse de recibo ACK es llevada acabo por una máquina de estados Mealy similar a la anterior, salvo que no posee los estados addr5 y addr6, debido a que solo requiere de un campo PID y no de los siguientes 2 bytes (direcciones y CRC).

4.2.7.1.3. CRC5

La determinación del campo CRC de 5 bits para los campos de dirección y endpoint del dispositivo (o número de frame) es llevada a cabo por dos módulos: CRC5 y crc5FSM. El módulo CRC5, cuando es habilitado, comienza un cálculo cíclico sobre 11 bits de datos cargados previamente. El algoritmo de cálculo de CRC que se describe a continuación está basado en [8] y [21]:

1. Cargar un registro de 5 bits, *crc5*, con "1's".

2. Si el resultado de un XOR[1] entre el bit más significativo del registro crc5 y de los datos es:
   - 1, entonces cargar crc5 con el resultado de un XOR entre el desplazamiento en 1 bit hacia la izquierda del registro crc5 y el patrón de bits 00101.
   - 0, desplazar el registro crc5 hacia la izquierda en 1 bit.
3. Desplazar los datos un bit hacia la izquierda. Si ya se recorrieron los 11 bits de datos, el campo CRC corresponde al inverso del registro crc5, si es que no, volver al paso 2.

El pseudo-código que implementa al módulo CRC5 se muestra a continuación:

```
crc5 = 31; // Equivalente a 5 bits en 1
data = data_in; // Carga datos

while(en)
{
      data = data ≪ 1;
      if(crc5[4] ∧ data[10])
           crc5 = (crc5 ≪ 1) ∧ 5;
      else
           crc5 = crc5 ≪ 1;
}

crc5_out = ∼crc5;
```

4.2.7.1.4. Crc5FSM

Este módulo es una máquina de estados Moore[2] que se encarga de habilitar al bloque CRC5 durante los 11 bits de datos, y luego indicarle al banco de registros que el CRC calculado es válido y por lo tanto se debe almacenar. Ésto es necesario ya que CRC5 no tiene como saber cuando terminó de recorrer todos los datos.

4.2.7.1.5. Packet_reg

Se utiliza un banco de registros de 8 bits para almacenar temporalmente el paquete a transmitir. Los primeros 4 registros corresponden al campo SYNC del paquete. El quinto registro se utiliza para almacenar el PID, el sexto y séptimo para la dirección, *endpoint* (o número de frame) y CRC. La función principal de este módulo, dentro de todo el proceso de generación del paquete, es uniformar la velocidad en que están disponibles sus campos y aprovechar mejor el tiempo al transmitir de inmediato el SYNC mientras se determina el resto de ellos.

[1]Operación de ó-exclusivo, *exclusive or*.
[2]Una máquina de estado Moore es aquélla donde las salidas solo dependen del estado actual.

4.2.7.2. DataGenerator

Este módulo se encarga de la generación de los paquetes de datos, dado el PID indicado por el bloque *execute* y los datos en la dirección indicada por el descriptor de la transacción. Al igual que la generación de un paquete *token*, un paquete de datos requiere de varios módulos para su elaboración, en particular, también utiliza un banco de registros aparte, pero distinto.

4.2.7.2.1. DataFSM

La máquina de estados (Mealy) que genera el paquete de datos es similar a la de un *token* (Fig. 4.7), salvo que posee más estados para recorrer más registros y que en cada registro para datos debe estar atenta al termino del campo de datos para luego enviar el CRC.

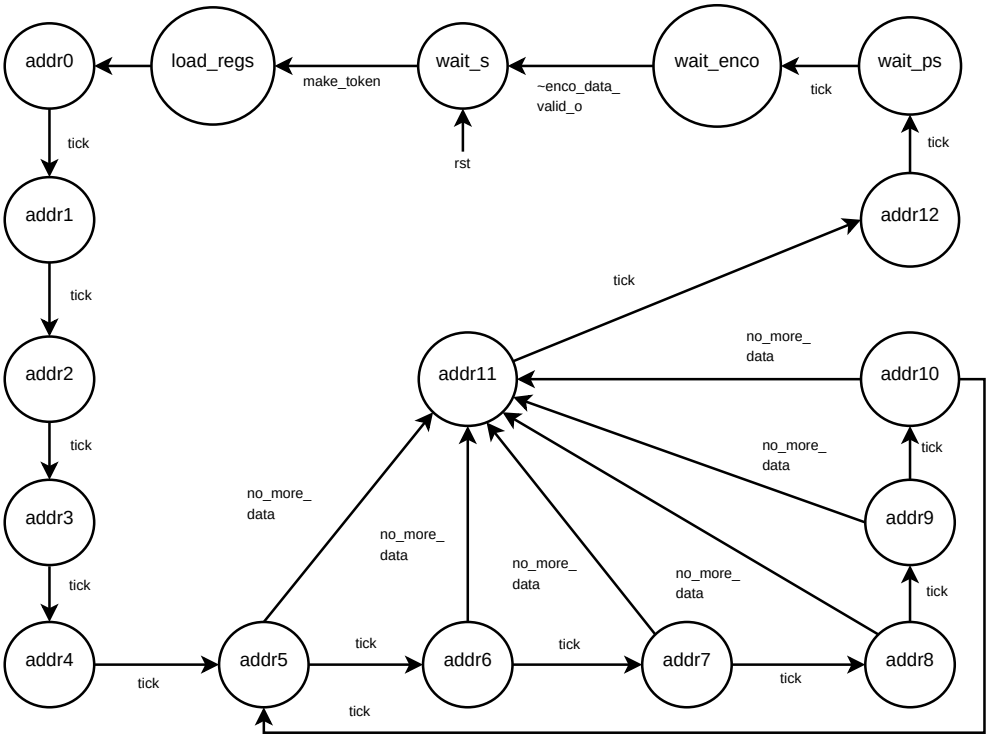


Figura 4.8: Máquina de Estados Generación Paquete de Datos

Se utilizan más estados para permitir enviar completamente el SYNC y PID antes de volver a escribir el primer registro de datos en caso de que la cantidad de bytes a enviar sea superior a 5. Los últimos dos registros son representados por los estados addr11 y addr12, los cuales contienen los 16 bits de CRC. Con esta implementación, el CRC es calculado al mismo tiempo que los datos son enviados al módulo *PHY_TX*.

La señal *no_more_data* es generada por el módulo *DataCounter*, de manera de indicar

que ya se realizó realizó la lectura de todos los datos almacenados en el banco de registros.

4.2.7.2.2. DataCounter

Este módulo se encarga de indicar si se debe seguir leyendo datos de la memoria o no, en función de la cantidad a datos a transferir según el módulo *execute*. Ésto lo realiza utilizando un contador que contiene inicialmente el valor de bytes a transferir más cinco, ya que se incluye los registros de SYNC y PID. De esta manera también se puede indicar exactamente con este contador en qué momento se debe comenzar y terminar la lectura de los datos almacenados en el banco de registros, debido a que los datos desde memoria no son enviados inmediatamente al módulo *PHY_TX*.

Además, este módulo se encarga de habilitar el bloque que realiza el cálculo del CRC de 16 bits de los datos al mismo tiempo en que éstos son leídos desde memoria. El CRC válido corresponde al valor en el ciclo siguiente al término de lectura de datos, lo cual es informado, a través de la señal *load_crc16*, al banco de registros para incorporarlo inmediatamente.

4.2.7.2.3. CRC16

El cálculo del CRC de 16 bits de los datos es llevado acabo por este módulo. El algoritmo es muy similar al utilizado para determinar el CRC de 5 bits, y se describe a continuación:

1. Cargar un registro de 16 bits, *crc16*, con "1's", e iniciar un contador de 3 bits en su máximo valor.
2. Si el resultado de un XOR entre el bit más significativo del registro crc16 y el bit de la entrada apuntado por el contador es:
   - 1, entonces cargar crc16 con el resultado de un XOR entre el desplazamiento en 1 bit hacia la izquierda del registro crc16 y el patrón de bits 1000000000000101.
   - 0, desplazar el registro crc16 hacia la izquierda en 1 bit.
3. Restar 1 al contador. Volver al paso 2 mientras el módulo esté habilitado.

El byte de datos presentados en la entrada del módulo deben mantenerse por 8 ciclos de reloj para que el contador pueda recorrer todos los bits. El siguiente byte de datos debe estar disponible inmediatamente después del anterior, ya que el módulo sigue calculando el CRC (mientras está habilitado) con los datos que estén en su entrada.

El pseudo-código que implementa al módulo CRC16 se muestra a continuación:

```
crc5 = 65535; // Equivalente a 16 bits en 1

while(en)
{
      for(counter = 7; counter == 0; counter−−)
      {
            if(crc16[15] ∧ data_in[counter])
                  crc16 = (crc16 ≪ 1) ∧ 32773;
            else
                  crc16 = crc16 ≪ 1;
      }
}

crc16_out = ∼crc16;
```

4.2.7.2.4. DPMem

Este módulo es el banco de registros utilizado para almacenar los campos y datos del paquete a transmitir, y cumple la misma función que *packet_reg*. Posee 13 registros de 8 bits, de los cuales los primeros cuatro corresponden al SYNC, el quinto al PID, del sexto al undécimo a datos y los últimos dos al CRC.

Los datos son almacenados a medida que están disponibles de manera secuencial desde el sexto registro al undécimo y luego vuelve al sexto si es que se requiere enviar más de 5 bytes. La lectura de los datos almacenados en los registros se realiza mientras la señal *data_read_en* esté en 1, lo cual es controlado por la máquina de estados DataFSM.

4.2.7.2.5. Data_write

Este bloque genera la señal *load_data*, la cual le indica al banco de registros que almacene los datos que están en su entrada *data_in*. La posición en que se almacenan los datos en generada internamente por el módulo *DPMem*.

4.2.7.3. Counter8

Este módulo se encarga de generar la señal *tick* cada 8 ciclos de reloj para así poder organizar el resto de los módulos en tiempos de bytes. Está implementado sobre un contador de 3 bits que debe ser habilitado para llevar acabo su función.

4.2.7.4. PHY_TX

El bloque *PHY_TX* realiza las 4 etapas finales[1] en la transmisión de un paquete: serialización de los datos leídos desde el banco de registros, rellenado de bits, agregar EOP dependiendo del tipo de paquete (ver sección 3.2.3.3) y finalmente codificar en NRZI.

4.2.7.4.1. Parallel2Serial

Este módulo realiza la serialización de los datos que provienen del bloque *TokenGenerator* o *DataGenerator*. Junto a los datos serializados entrega una señal (*data_valid_o*) que indica en que momento éstos son válidos, de manera que el siguiente módulo pueda procesarlos sin perder ni agregar un bit.

4.2.7.4.2. BitStuffer

Los datos provenientes del módulo *Parallel2Serial* son procesados por este bloque mientras la señal de datos válidos se encuentre en un "1" lógico. *BitStuffer* realiza la función de agregar un bit 0 por cada seis 1's consecutivos en los datos de entrada.

El ingreso de un bit 0 a los datos implica un retraso en un ciclo de reloj de los datos originales, por lo cual se requiere de un elemento de memoria para ir almacenando los datos de entrada. En particular se utiliza un registro FIFO[2] para guardar y enviar los datos.

La implementación de este bloque posee mayor complejidad debido a que dependiendo del estado del registro FIFO y de la señal *data_valid* proveniente del módulo serializador, aparecen distintos casos que se deben abordar:

- Cuando el FIFO está vacío y la señal es 1, la salida es igual a la entrada de datos, y se analizan los datos de entrada.
- Cuando el FIFO no está vacío y la señal es 1, se deben almacenar los datos de entrada, los datos de salida provienen del FIFO y deben ser analizados.
- Cuando el FIFO no está vacío y la señal es 0, los datos de salida provienen del FIFO y deben seguir siendo analizados.

Con analizar los datos se refiere a contar los 1's consecutivos e insertar un 0 cuando se llegue a seis. Este módulo también genera una señal, *data_valid_o*, que indica en que momento los datos de salida son válidos para que así el siguiente bloque sepa cuando procesarlos.

Otros registros que se definen en este módulo se explican a continuación:

[1] Antes del circuito analógico.
[2] *First In First Out*, registro que envía datos en el orden en que llegan.

- *n_data*: Se utiliza para determinar si el FIFO está lleno o no, de manera que si el registro es mayor a 0 entonces el FIFO tiene uno o más elementos.
- *fifo_read*: Corresponde a un puntero a la posición de un dato del FIFO que debe ser leído a continuación.
- *fifo_write*: Corresponde a un puntero a la posición del FIFO en que se debe almacenar el siguiente dato.
- *bit_count*: Almacena la cuenta de 1's consecutivos.
- *data_out*: Bit de salida del módulo.

La implementación de este módulo es representada por el pseudo-código mostrado a continuación. Notar que *data_in* corresponde a los datos de entrada provenientes de *Parallel2Serial*.

```
while(data_valid or n_data or (bit_count is 6))
{
      data_valid_o = 1;
      if(bit_count is 6) {
           data_out = 0;
           bit_count = 0;
           if(data_valid) {
                n_data = n_data + 1;
                fifo[fifo_write] = data_in;
           }
      }
      else {
           if(n_data > 0) {
                if(fifo[fifo_read])
                     bit_count = bit_count + 1;
                else
                     bit_count = 0;
                data_out = fifo[fifo_read];
                fifo_read = fifo_read + 1;
                if(data_valid) {
                     fifo[fifo_write] = data_in;
                     fifo_write = fifo_write + 1;
                }
                else
                     n_data = n_data - 1;
           }
           else {
                if(data_in)
                     bit_count = bit_count + 1;
```

```
                    else
                        bit_count = 0;
                    data_out = data_in;
                }
            }
        }
```

4.2.7.4.3. EOP

Este módulo realiza la función de insertar el campo EOP correspondiente al tipo de paquete. Se utiliza un registro de 40 bits que contiene el eop, el cual cambia de valor en función de la entrada *sof*. Esta última es levantada durante la generación de un paquete SOF.

La señal *data_valid_o* proveniente del módulo *BitStuffer* ayuda a determinar el fin de los datos para así insertar el EOP. Al igual que los bloques anteriores, éste también genera una señal de validez de datos.

4.2.7.4.4. NRZIenco

La codificación NRZI (ver sección 3.3.1) del paquete a enviar es llevada a cabo por este módulo. La codificación es fácilmente implementada en Verilog basándose en el siguiente algoritmo:

1. Inicializar un registro de datos codificados, *encoded_data*, en 1.
2. Mientras los datos sean válidos, y el bit entrante es:
   - 1, mantener el valor del registro encoded_data.
   - 0, invertir el valor del registro.

Este módulo también genera una señal para indicarle al circuito analógico que los datos codificados que se están enviando son válidos.

4.2.8. RX

Este módulo se encarga de la recepción de los datos que provienen del circuito analógico transpondedor[1] conectado al bus. Las funciones que realiza son básicamente identificar que tipo de paquete es el recibido, que parte son datos, si el CRC está correcto y generar señales de control correspondientes para que el resto del controlador sepa que debe hacer a

[1]Del inglés *tranceiver*.

continuación. Además, en caso de ser un paquete de datos, éstos los envía a la memoria de datos en secuencias de a 1 byte sin el campo CRC.

Este módulo se divide en los bloques mostrados en la figura 4.9 para realizar sus funciones.

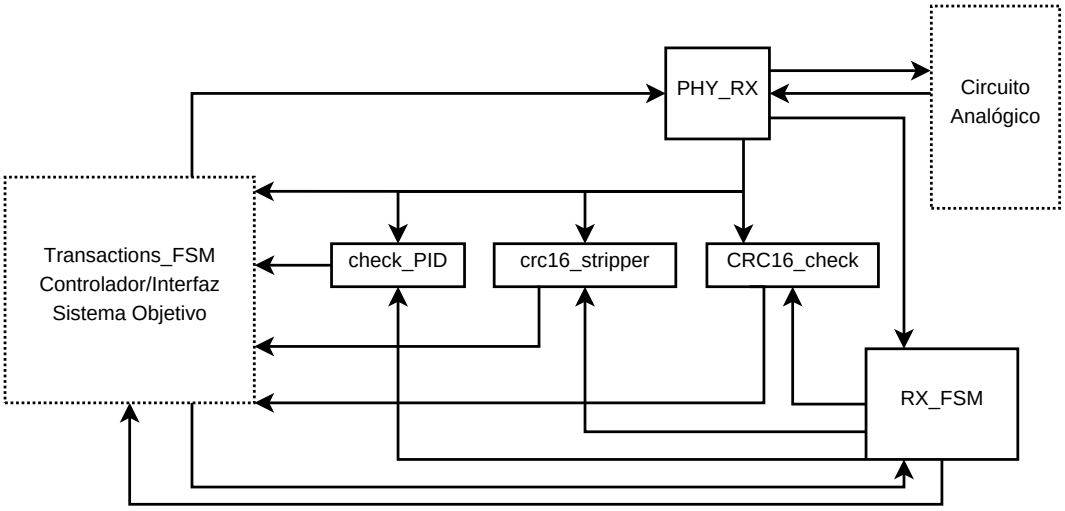


Figura 4.9: Diagrama de Bloques Recepción

4.2.8.1. PHY_RX

Este módulo realiza las funciones básicas y necesarias para todos los paquetes recibidos. Está compuesto por 3 módulos que se explican en los siguientes apartados.

Además del reloj del controlador, este bloque utiliza el reloj capturado por el circuito análogo (clk_bus) para la sincronización de los datos. Una señal *squelch* (que se explica más adelante) permite reconocer el paquete luego de haber sido ser procesado por la etapa análoga para volverlo a valores digitales. Como salida se envían de a 8 bits los valores del paquete recibido, una señal *data_ready* que indica que los 8 bits anteriores son nuevos datos y válidos, y la señal *eop* que indica el fin del paquete.

4.2.8.1.1. Elasticity_buffer

Este bloque permite absorber las diferencias entre el reloj del controlador y la velocidad en que llegan los datos en el bus. Es básicamente un buffer de 24 bits que se llena (a la velocidad del bus) hasta un umbral de 12 bits, para luego comenzar a enviar los datos almacenados a la velocidad del controlador. La dimensión del buffer y umbral está basada en [22] y [23], donde se tomó en consideración para el cálculo la precisión del reloj pedida por la especificación de USB 2.0. de ± 500 [ppm], el *jitter* o variaciones de fase y frecuencia, y el paquete de datos más largo (1024 bytes de datos). De esta manera se asegura de que si

los datos están llegando más rápido o lento no se pierdan y puedan ser enviados a la misma velocidad en que trabajan los módulos del controlador.

El circuito receptor analógico debe poseer un detector de envolvente, el cual indica a través de una señal de *squelch* cuando los datos entrantes son inválidos. En otras palabras, cuando la señal *squelch* está en un "1" lógico indica que no hay un paquete válido, mientras que un "0" indica que si hay. De acuerdo a la especificación, el detector debe ser lo suficientemente rápido como para poder extraer el reloj del paquete entrante, y lograr detectar por lo menos los últimos 12 bits del SYNC del paquete. La señal squelch es utilizada como habilitador del buffer.

Además de la función de amortiguación, este módulo entrega una señal de *data_valid* que indica en que momento hay datos válidos para los siguientes bloques.

4.2.8.1.2. NRZIdeco

Este módulo realiza la función de decodificar los datos entrantes (codificados en NRZI). Utiliza la señal *data_valid* proveniente del bloque *elasticity_buffer* para habilitar la decodificación de la entrada de datos.

El módulo está básicamente diseñado con dos registros: Uno para mantener el bit anterior de datos, y el otro que corresponde a la salida que mantiene el valor calculado como un XNOR[1] entre el dato anterior y el actual. Esta implementación se presenta con el siguiente pseudo-código:

```
data_out = 1;
old_data = 0;

while(en)
{
      data_out = old_data ∧ ∼ data_in;
      old_data = data_in;
}
```

4.2.8.1.3. Syncstrip

Este bloque se encarga de generar una señal que indica que parte del paquete recibido es el campo SYNC y cual no, en otras palabras, elimina el campo SYNC para que luego otros módulos procesen desde el campo PID hasta el EOP.

Para realizar su función, el módulo implementa una máquina de 4 estados consecutivos como se muestra en la figura 4.10. Los estados se codificaron en código gray [24] para

[1]Operador lógico que calcula un ó-exclusivo negado.

minimizar área (cantidad de flip-flops) y carreras críticas.

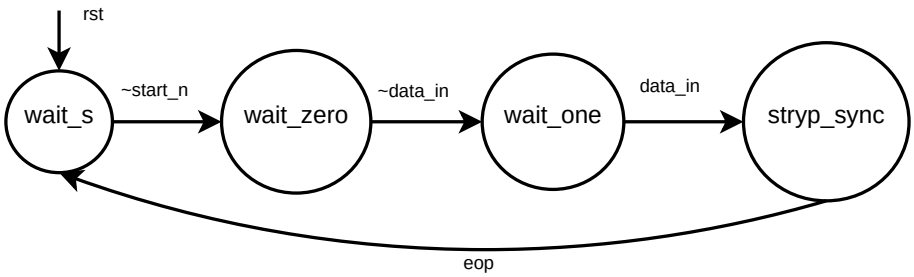


Figura 4.10: Máquina de Estados para Quitar el SYNC

La señal *start* le indica a la máquina de estados que un paquete está siendo recibido, por lo tanto debe salir del estado de espera general. Se utiliza la señal *data_valid* proveniente del bloque *elasticity_buffer* como *start*, con la cual se pasa a otro estado de espera, pero esta vez buscando el momento en que el bit de entrada tenga valor 0.

Los datos de entrada corresponden a la señal ya decodificada, por lo tanto lo que se debería esperar es un "1" que indicaría el fin del campo SYNC. Ésto sería siempre correcto si es que la porción del campo SYNC codificado que se recibiese partiera siempre en un "1", de manera que al decodificar se tendría un "0" como primer bit decodificado. Sin embargo, no necesariamente se van a recibir la misma cantidad de bits de SYNC, por lo cual la señal decodificada puede partir en cualquier valor. En el caso en que ésta tuviese un "1" como bit inicial, se cometería un error al considerar ese bit como fin del campo SYNC, por lo cual se debe omitir ese bit y esperar al siguiente "1". Ésto se logra asegurando esperar el momento en que la señal vuelva a "0" para cualquiera de los dos casos, y luego esperar el "1" que indica el fin del SYNC.

Una vez detectado el comienzo del campo PID, se pasa al estado sync_strip, donde se levanta la señal *data_en_o* para indicar la porción del paquete que los demás módulos deben procesar. El fin del paquete es detectado por otro bloque, para lo cual levanta la señal *eop*.

4.2.8.1.4. BitUnstuffer

La función realizada por este módulo es la de quitar los "0's" agregados en los datos debido a la necesidad de producir una transición en los datos codificados, como se vió en la sección 3.3.2. Además detecta el fin del paquete o bien un error al contar siete "1's" consecutivos. Este bloque es habilitado por el módulo syncstrip durante los campos PID hasta antes del EOP del paquete.

El funcionamiento consiste básicamente en ir almacenando los bits de entrada en un registro de 8 bits al mismo tiempo que se cuentan los "1's" consecutivos. Cuando suman seis, se verifica el valor del siguiente bit de entrada: si es 1, entonces se produjo un error o se llegó

al fin del paquete, por lo cual se levanta la señal *eop*; si es 0, se vuelve a 0 la cuenta y no se almacena. Cuando ya se almacenaron 8 bits en el registro se asigna a la salida y se levanta la señal *data_ready* para indicar que un nuevo byte del paquete se encuentra disponible para los demás módulos.

Cabe señalar que al comienzo del proceso se genera un *data_ready* que está demás (debido al diseño del bloque), y que no debe ser considerado, lo cual es llevado acabo por la máquina de estados de recepción.

La implementación de este módulo se representa por el siguiente pseudo-código:

```
data_out = 1;
old_data = 0;
eop = 0;
ptr = 7;

while(en)
{
      if(ptr is 7) {
           data_out = data;
           data_ready = 1;
      }
      else
           data_ready = 0;
      if(bit_count is 6) {
           if(data_in is 1)
                eop = 1;
           else
                bit_count = 0;
      }
      else {
           if(data_in is 1)
                bit_count = bit_count + 1;
           else
                bit_count = 0;
           data[ptr] = data_in;
           ptr = ptr - 1;
      }
}
```

El registro *ptr* se utiliza como puntero a la posición del registro de 8 bits *data*, donde se almacenan temporalmente los datos de entrada. Cuando se ya se escribieron 8 bits, es decir cuando *ptr* es 7, *data* se asigna al registro de salida *data_out* y se indica que son válidos al levantar *data_ready*.

4.2.8.2. Check_PID

Este módulo se encarga de decodificar el campo PID del paquete recibido. Tiene una salida por cada tipo de paquete posible de recibir[1] más una que indica error, ya sea por un campo de verificación de PID incorrecto o por un PID desconocido. Los posibles paquetes a recibir son los siguientes: data0, data1, data2, ack, nak, stall y nyet.

Se debe levantar la entrada *check* en el momento en que los datos en la entrada *PID* correspondan al PID a revisar. Una señal de salida *err_PID* le indica al resto del controlador de un posible error en el campo PID del paquete recibido.

4.2.8.3. RX_FSM

Esta máquina de estados tipo Mealy controla todos los eventos en la etapa de recepción de una transmisión USB. Esto se traduce en lo siguiente:

- Identificar en qué momento se debe activar el bloque que revisa el PID del paquete entrante.
- Si no hay errores en el PID, indicar que se identificó el paquete correctamente, a través de la señal *packet_ready*.
- Si es que es un paquete de datos, activar la verificación de CRC, quitar campos CRC y realizar conteo de bytes.
- Esperar que se guarden datos y resultados del proceso de recepción para volver al estado inicial.

En la figura 4.11 se puede observar la máquina de estados implementada. Los estados, junto con algunas de sus salidas y entradas, son descritas en la tabla 4.4.

[1]Un dispositivo no debería enviar nunca un paquete de datos MDATA o paquetes tipo token.

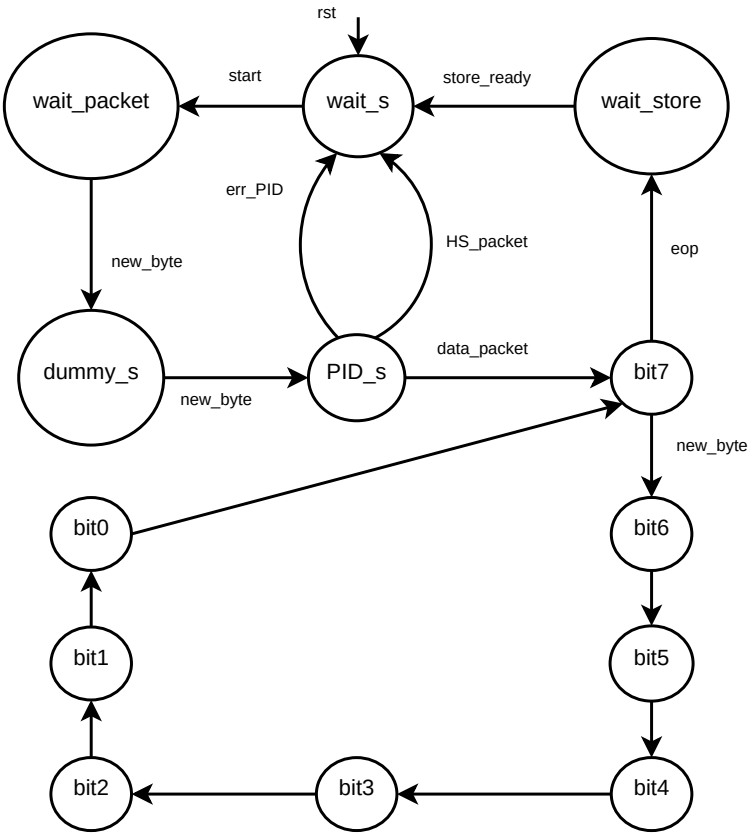


Figura 4.11: Máquina de Estados para Recepción

Estado	**Descripción**
wait_s	Estado inicial donde se mantienen los valores por defecto de salidas y registros internos y se espera que se salga de condición reset y se habilite el controlador.
wait_packet	Estado en que se espera que aparezca un paquete, lo cual es expresado a través de la señal *new_byte* proveniente de del módulo *BitUnstuffer*.
dummy_s	Estado creado para absorber el *new_byte* que está demás, como se explicó en la sección 4.2.8.1.4.
PID_s	En este estado se revisa el PID del paquete recibido, y se toma una decisión de acuerdo al tipo de paquete. Si hay error (*err_PID*) o es un paquete handshake (*HS_packet*) , se vuelve al estado inicial. Si es un paquete de datos (*data_packet*) se pasa una serie de estados que supervisan la recepción de datos bit a bit.
wait_store	Espera a que los resultados y estado de transacción sean almacenados.
bit7-0	En estos estados se habilita el chequeo de CRC, se determina la cantidad de bytes recibidos y se generan 2 señales en distintos tiempos: Una para indicarle al bloque *crc16_stripper* que los datos son válidos, y la otra para almacenar los datos en memoria. La señal *eop* indica el fin del paquete y se vuelve al estado inicial

Tabla 4.4: Descripción Estados RX_FSM

4.2.8.4. CRC16_check

Este módulo verifica si el campo CRC que viene en un paquete de datos es coherente con los datos. Esto se realiza calculando el CRC de 16 bits (de igual manera que en la sección 4.2.7.2.3) sobre todos los datos y campo CRC en el orden en que llegan. Si el resultado es igual al residual 1000000000001101, entonces datos y campo crc se encuentran sin errores, por lo cual se levanta la señal *crc16_ok* para indicar que está bien.

4.2.8.5. Crc16_stripper

La función de este bloque es evitar que se envíe el campo CRC como parte de los datos hacia la memoria donde se van a almacenar. Como no es posible predecir en que momento comienza el campo CRC, pero si se sabe que son los últimos dos bytes (luego de quitar el EOP), entonces la manera de proceder es retrasar en 2 ciclos el envío de datos hacia memoria, de manera que en el momento en que se llega al campo CRC ya no exista la señal *new_byte*[1].

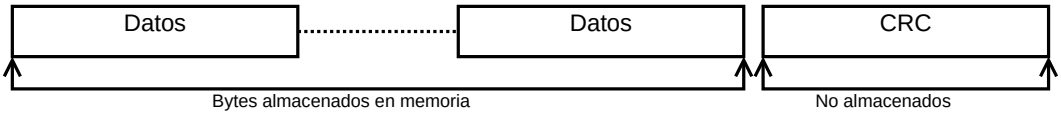


Figura 4.12: Objetivo del Módulo *crc16_stripper*

[1]Este señal indica que los datos presentados son nuevas y válidos, por lo tanto se deberían considerar para procesarlos. Se utiliza, junto a otra señal, para almacenar los datos.

4.3. Verificación Funcional

La verificación funcional del diseño escrito en Verilog se realizó a través de simulaciones con las herramientas de Synopsys® VCS®(para la compilación) y DVE®(para la visualización).

La mayor parte de los módulos fueron probados por separado, de manera de simplificar el proceso de búsqueda de errores al momento de verificar grupos de módulos. En particular, los módulos contenidos en el bloque Controlador/Interfaz fueron probados con el diseño completo, es decir, con el *Top*.

4.3.1. Testbench

La verificación funcional requiere de la creación de un *testbench* para el DUT[1], donde se especifiquen todas las entradas del diseño en los momentos deseados para así poder analizar las salidas en el tiempo.

El *testbench* fue implementado utilizando Verilog a nivel comportamental, esto implica usar directivas que no son sintetizables pero que simplifican la verificación funcional.

Para comprobar si la funcionalidades del diseño fueron implementadas correctamente, se creó un dispositivo simple, que solo se dedicase a responder con paquetes predefinidos, pero escogidos aleatoriamente para el tipo de transferencia del descriptor de transacción cargado en memoria. Además del dispositivo virtual, se implementaron a nivel comportamental las memorias de instrucciones y datos, de manera que contengan su respectiva información para probar el funcionamiento completo del controlador, es decir, tanto comunicación con el sistema objetivo como ejecución del protocolo USB.

4.3.1.1. Direcciones y Endpoints

Se definieron una serie de direcciones y endpoints de dispositivos para una verificación más completa del diseño realizado, de manera que todos los tipos de transferencia sean cubiertos. La asignación de direcciones a dispositivos es un paso previo a las transferencias probadas, pero que se diferencian solo en que la dirección y endpoint que se utilizan son las por defecto (0), por lo tanto la suposición que ya están asignados no perjudica la calidad del *testbench*.

En la tabla 4.5 se muestran todas las direcciones definidas junto con sus *endpoints* y características. Un tipo de *endpoint BI* significa que es bidireccional.

[1] *Design Under Test*, o Diseño Bajo Prueba.

En base a los dispositivos virtuales caracterizados en la tabla 4.5 se definieron listas de transacciones que se describen más adelante.

Dirección	Endpoint	Tipo Endpoint	Tipo Transferencia
5	0	BI	Control
	5	OUT	Bulk
	6	IN	
7	0	BI	Control
	1	OUT	Interrupt
	10	IN	
10	0	BI	Control
	3	OUT	Isochronous
	5	IN	
51	0	BI	Control
	5	IN	Isochronous
	9	OUT	
66	0	BI	Control
	1	OUT	Bulk
	2	IN	
85	0	BI	Control
	1	OUT	Interrupt
	2	IN	

Tabla 4.5: Direcciones y Endpoints Virtuales

4.3.1.2. Dispositivo Virtual

El dispositivo virtual, implementado para la verificación funcional del diseño, está compuesto por distintos módulos para llevar a cabo sus dos principales funciones que son recibir un paquete y transmitir una respuesta acorde a éste. Este dispositivo representa a todas las direcciones y endpoints definidos anteriormente.

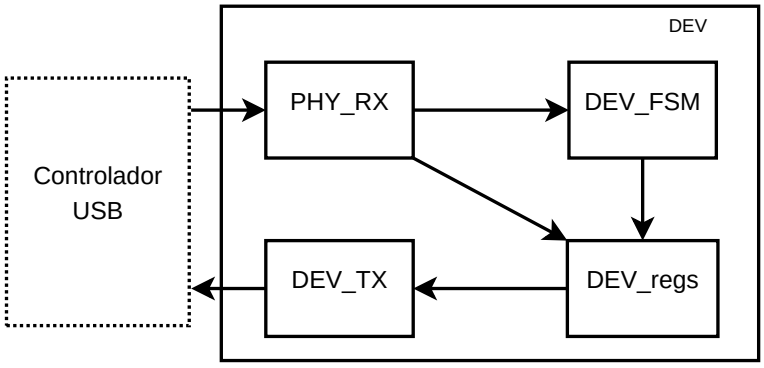


Figura 4.13: Diagrama de Bloques Dispositivo Virtual

Como se observa en la figura 4.13, se utilizó el mismo módulo *PHY_RX* (sección

4.2.8.1) del controlador para la recepción de los paquetes.

La máquina de estados *DEV_FSM* está basada en *RX_FSM* (sección 4.2.8.3) con algunas modificaciones, ya que no requiere prestar atención al campo de datos ni tampoco indicarle a otro módulo que decodifique el PID. Su función es básicamente indicarle a un banco de registros de 8 bits, *DEV_regs*, que almacene cada byte del paquete a medida que lo va recibiendo.

El banco de registros, a su vez, le entrega al módulo *DEV_TX* los primeros tres bytes recibidos, los cuales corresponden a:

- En el caso de un paquete token y PING: PID, dirección y endpoint del dispositivo respectivamente.
- En el caso de un paquete de datos o handshake solo interesa el primer byte que es el PID.

4.3.1.2.1. DEV_TX

Este módulo se encarga de generar las respuestas en las transacciones iniciadas por el host/controlador. Para ésto, decodifica el PID, dirección y endpoint para identificar el tipo de transferencia y por lo tanto el modo de proceder.

Para cada tipo de transferencia y etapa de una transacción se definen un conjunto de respuestas, que puede contener entre otras:

- Paquetes handshake ACK, NAK, NYET y STALL.
- Paquetes de datos DATA0 y DATA1 con campo CRC correcto e incorrecto.
- Desconexión por *timeout*.
- Paquetes con PID erróneos.

La respuesta es escogida aleatoriamente dentro del conjunto. Cada vez que el fin del paquete es detectado[1] se generan dos números pseudo-aleatorios. El primero permite escoger el tipo de respuesta (handshake, datos, timeout o PID erróneo), mientras que el segundo define el paquete de datos en caso que el tipo de paquete sea de datos.

4.3.1.3. Listas de Transacciones

Para verificar la funcionalidad de la descripción RTL del controlador, se implementaron listas de transacciones periódicas y asíncronas, suponiendo que el software las escribió en

[1]Lo cual es indicado por la señal *eop* proveniente del módulo *PHY_RX*.

memoria. Los descriptores de las listas están basados en los dispositivos virtuales planteados en la sección 4.3.1.1.

En las figuras 4.16 y 4.15 se muestra la lista asíncrona y periódica respectivamente, en términos de la secuencia de descriptores a ejecutar. Cada elemento tiene el siguiente formato:

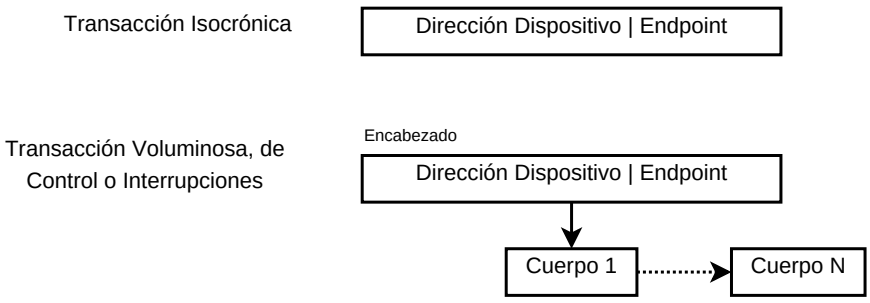


Figura 4.14: Formato Elementos de Listas

Un elemento de lista que representa un descriptor isocrónico no requiere especificar el tipo de paquete token que va a utilizar, ya que solo puede generar transacciones en la dirección en que está definido el endpoint de acuerdo al descriptor. Por otro lado, un descriptor BCINTd puede especificar el paquete token que va a utilizar en cada uno de sus cuerpos, por lo cual se debe evidenciar el campo PID de cada cuerpo.

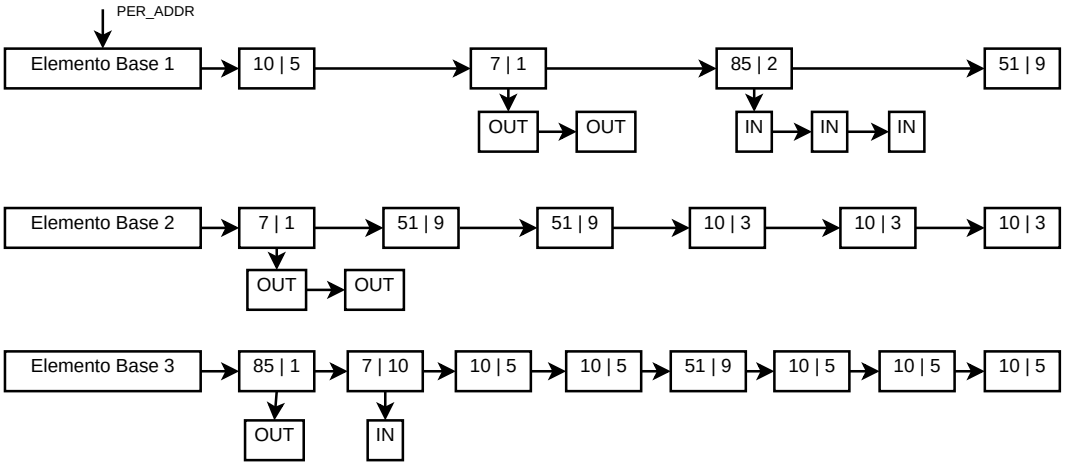


Figura 4.15: Lista Periódica Implementada en Testbench

La lista periódica está compuesta de 3 elementos base, por lo tanto hay 3 secuencias de descriptores a ejecutar en 3 distintos micro-frames. La lista comienza en el elemento base apuntado por el registro de configuración *PER_ADDR*.

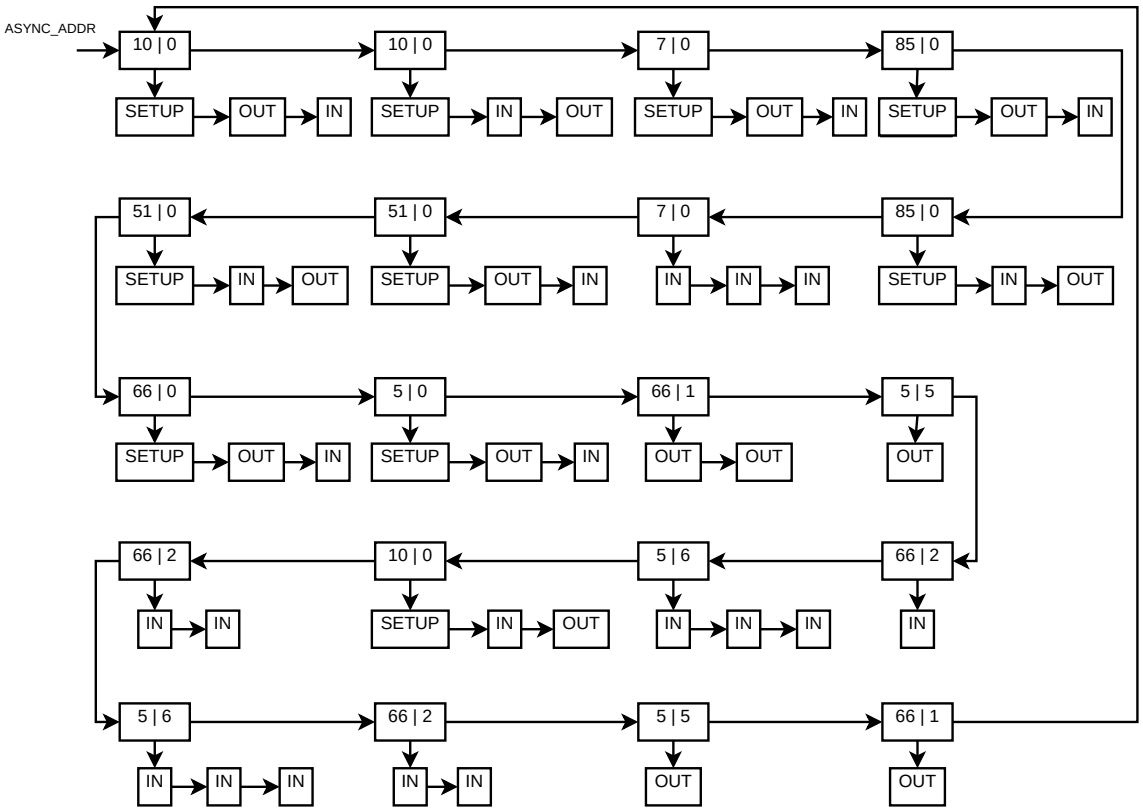


Figura 4.16: Lista Asíncrona Implementada en Testbench

La lista asíncrona implementada es cerrada, de manera de poder reanudar transacciones detenidas, y contiene todo los tipos de transferencia asíncronas. El comienzo de la lista es indicado por el registro de configuración *ASYNC_ADDR*.

4.3.1.4. Registros de Configuración

Como se constató en la sección 3.5, no se definió un protocolo para el acceso a los registros de configuración del controlador. Éstos son asignados directamente en el *Top* del diseño, de manera que sean considerados a nivel comportamental junto al resto del testbench.

Inicialmente los registros *USB_RST*, *PER_EN* y *USB_EN* están en 0 y la lista asíncrona se encuentra activada, simulando el hecho de que los dispositivos son configurados antes de poder generar transacciones periódicas. Los registros de dirección se encuentran asignados de acuerdo a la posición del inicio de las listas en memoria, mientras que el número de elementos base, *N_PER*, es 3.

El dispositivo es habilitado asignando un 1 al registro *USB_EN* ciclo y medio de reloj después de comenzada la simulación. La lista periódica es habilitada para el segundo micro-frame de la simulación.

4.3.2. Simulación

La validación del diseño[1] se llevó acabo utilizando la herramienta de Synopsys® DVE®, la cual permite una visualización de las formas de onda de todas las señales de los módulos pertenecientes al *Top*. La simulación da como resultado los valores en el tiempo de las señales internas y salidas en función de las entradas definidas en los apartados anteriores.

Dada la mediana complejidad de la implementación del controlador USB, se tienen muchas señales que requieren ser revisadas en términos de su comportamiento comparado con lo que se espera según el diseño y con la especificación de USB 2.0 [8]. La cantidad de señales a revisar complica el proceso de validación funcional, por lo cual gran parte de los módulos fueron simulados por separado, y luego en conjunto con los más cercanos. Por ejemplo, cada uno de los 4 módulos pertenecientes a *PHY_TX* se probaron y corrigieron errores hasta que su funcionalidad fuese llevada a cabo correctamente, y luego se juntaron para verificar la funcionalidad completa y corregir los errores de comunicación entre ellos. Es bueno notar que el proceso de diseño de circuitos digitales es iterativo en cada una de sus etapas y entre una y otra también, ya que la corrección de un error o recopilación de nueva información en una etapa posterior puede llevar a re-diseñar otros módulos.

Además de la cantidad de señales se tiene la complejidad del protocolo, que puede extender la ejecución de una transacción en gran parte de un micro-frame. Las listas implementadas en el testbench poseen muchos elementos como para constatar la ejecución de cada uno de ellos en este documento, por lo tanto se escogieron dos de ellos: Un descriptor isocrónico de la lista periódica y un descriptor de control de la lista asíncrona para abarcar la mayor parte del protocolo USB. Además se muestra la generación de un paquete SOF.

4.3.2.1. Comienzo de Micro-Frame

A continuación se muestra el primer micro-frame generado en la simulación (después del *reset*):

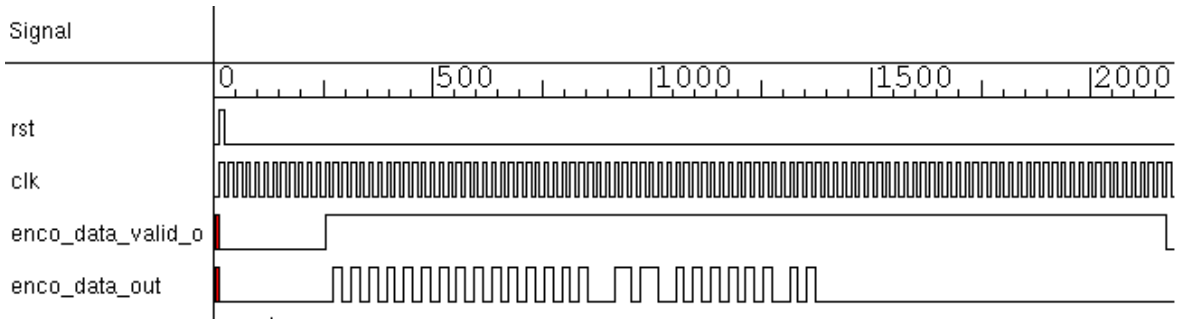


Figura 4.17: Primer Paquete SOF

[1]En la etapa de descripción RTL.

La señal *enco_data_valid_o* le indica al circuito analógico en que momento los datos generados son válidos. Los datos, *enco_data_out*, ya se encuentran codificados, por lo cual se observa que al inicio se tienen variaciones periódicas correspondientes al campo SYNC, mientras que en el final se mantiene constante por 40 ciclos, lo cual corresponde al campo EOP para un SOF.

La regla de tiempo que aparece en la figura 4.17 y en todas las simulaciones es tal que 20 unidades corresponden a un ciclo de reloj.

4.3.2.2. Descriptor de Control

El primer descriptor ejecutado en el *testbench* corresponde a uno de control. La razón de la elección de mostrar un descriptor de control y no uno de transferencia voluminosa de datos, es que éste abarca mayor parte del protocolo. Las características (o campos) del encabezado del descriptor se muestran en la tabla 4.6, mientras que la de los cuerpos en las tablas 4.7, 4.8 y 4.9. Los valores mostrados son los iniciales, ya que algunos pueden cambiar en función de lo que ocurra en el bus.

Característica	**Valor**
Dirección Dispositivo	10
Endpoint	0
Data Toggle	0
Ping	0
Tamaño Máximo	5
Número Errores	0
Halt	0

Tabla 4.6: Encabezado de Descriptor

Característica	**Valor**
PID	10 = setup
Status	0
N° Bytes a Transferir	5
Err ID	0

Tabla 4.7: Primer Cuerpo del Descriptor

Característica	**Valor**
PID	00 = out
Status	0
N° Bytes a Transferir	30
Err ID	0

Tabla 4.8: Segundo Cuerpo del Descriptor

Característica	**Valor**
PID	01 = in
Status	1
N° Bytes a Transferir	24
Err ID	0

Tabla 4.9: Último Cuerpo del Descriptor

Cada cuerpo corresponde a una etapa de la transacción de control: el primero cuerpo corresponde a la etapa de SETUP; el segundo a la de DATA o datos; y el último a la de STATUS.

Las próximas figuras muestran solo una parte de la simulación, de manera de poder aislar e identificar claramente las señales de interés. Por esta misma razón, se presentan solo los primeros 3 bytes de cada paquete recibido por el dispositivo virtual, los cuales se explican a continuación:

- *byte0*: Es el primer byte recibido del paquete enviado por el controlador, corresponde al PID en *little-endian* (ver sección 3.2.1).
- *byte1*: Es el segundo byte recibido. Dependiendo del tipo de paquete puede contener la dirección del dispositivo, parte del endpoint o datos.
- *byte2*: Es el tercer byte recibido. Dependiendo del tipo de paquete puede contener parte del endpoint, CRC o datos.

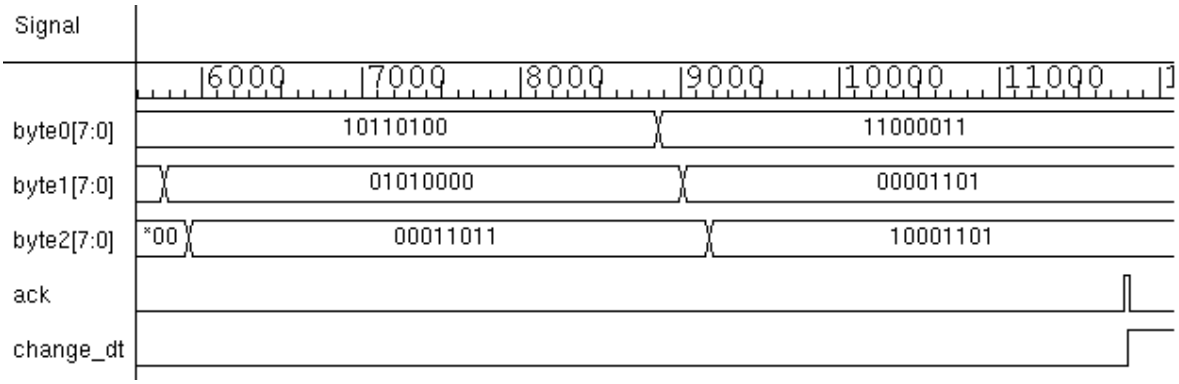


Figura 4.18: Etapa SETUP

En la figura 4.18 se muestra lo que ocurrió para la etapa de SETUP de la transacción. Se envió el paquete token y luegos los datos. En respuesta a ésto, el dispositivo virtual respondió con un ACK, lo que se refleja en la señal *ack*. Ésto implica recepción exitosa y que el bit de *data toggle* debe cambiar, lo cual es indicado por *change_dt*.

La siguiente etapa corresponde al envío de datos. Las señales mostradas en esta etapa se explican a continuación:

- *nak*: Indica recepción de un paquete NAK.
- *err_PID*: Indica que el PID del paquete recibido por el controlador no es conocido o tiene errores.
- *nyet*: Indica recepción de un paquete NYET.
- *DT*: Refleja el valor del campo *data toggle* del encabezado del descriptor.
- *set_ping*: Pone en "1" el campo *ping* del encabezado del descriptor.
- *ping*: Refleja el valor del campo *ping* del encabezado del descriptor.

Las señales *nak*, *err_PID* y *nyet* provocan el envío de un paquete PING en la siguiente transacción. Como el tamaño máximo del descriptor es de 5 bytes y la etapa de datos requiere enviar 30, entonces se necesitan 6 transacciones de datos (cambiando su PID de DATA0 a DATA1 y viceversa). El detalle de esta etapa junto con su respectiva simulación se encuentra en el Anexo A.

La etapa de STATUS (Figs. 4.19 y 4.20), de acuerdo al protocolo, corresponde a transacciones en el sentido opuesto a la etapa de datos y con PID DATA1 para los datos. En el primer intento de realizar la transacción ocurre una desconexión por *timeout*, lo cual aumenta el contador de errores. En el segundo intento el paquete recibido es un STALL para indicar que se debe suspender las transacciones a y desde el endpoint del descriptor.

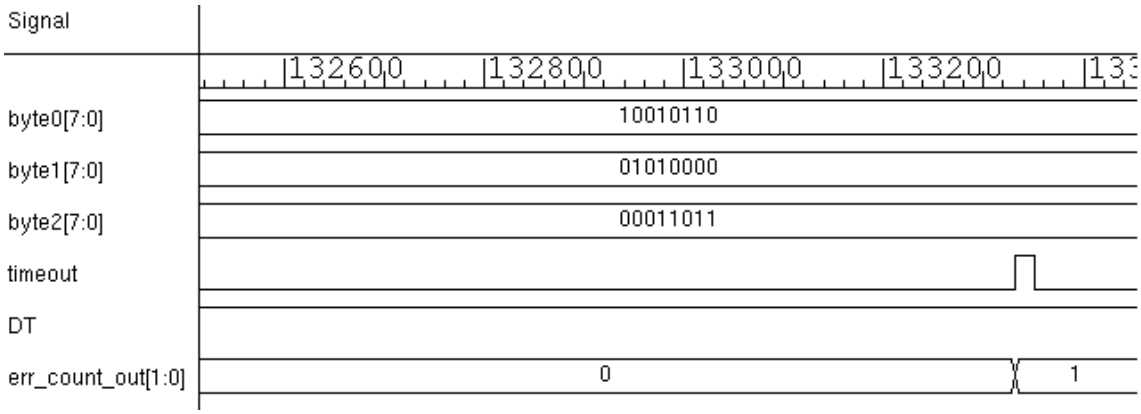


Figura 4.19: Etapa STATUS Primer Intento

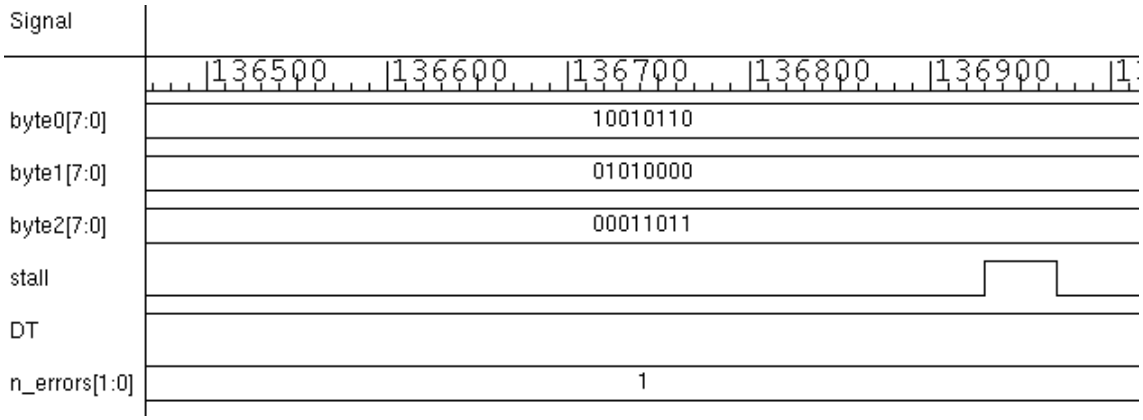


Figura 4.20: Etapa STATUS Segundo Intento

4.3.2.3. Descriptor Isocrónico

Las características del descriptor isocrónico documentado se muestran en la tabla 4.10

Característica	**Valor**
Dirección Dispositivo	51
Endpoint	9
Dirección	0
PID	01 = DATA1
N° Bytes a Transferir	10

Tabla 4.10: Características Descriptor Isocrónico

El descriptor corresponde entonces a una transacción de 10 bytes desde el controlador al dispositivo, con PID de datos DATA1. Este tipo de transferencia no tiene protocolo para asegurar recepción de dato, solo los envía. En la figura 4.21 se muestra la transmisión del paquete token y de datos, su señal validadora y los primeros 3 bytes recibidos por el dispositivo.

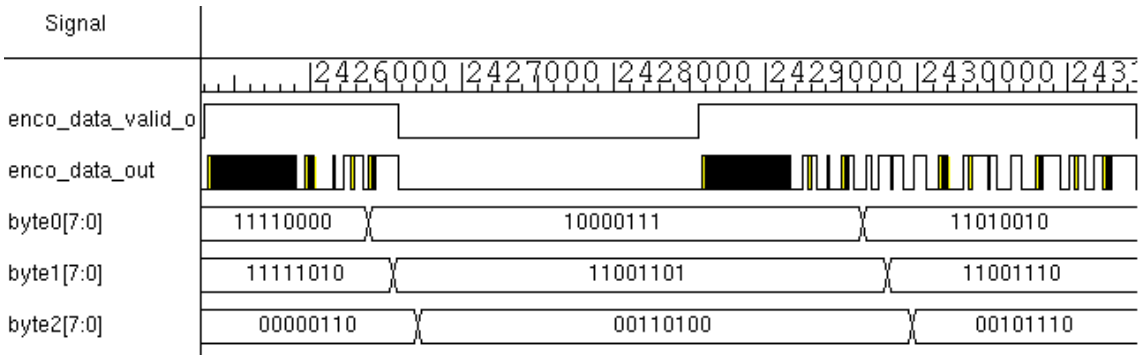


Figura 4.21: Simulación Transacción Isocrónica OUT

4.4. Síntesis Lógica

Esta etapa del flujo de diseño permite pasar del nivel de abstracción de la descripción RTL del circuito a una lista de compuertas interconectadas optimizada. De acuerdo a [25], la síntesis lógica es tal que:

$$Síntesis = Traducción + Optimización\ Lógica + Mapeo\ de\ Compuertas$$

La *traducción* corresponde al traspaso del código RTL a compuertas genéricas[1] que no poseen características de tiempo[2] ni carga[3]. En este nivel se agregan las restricciones de tiempo, área y potencia en caso de requerirse.

La *optimización de la lógica* y *mapeo de compuertas* se ejecutan en conjunto e iterativamente. La primera realiza optimizaciones (considerando las restricciones aplicadas) de las funciones lógicas que implementa el circuito, de manera de disminuir el número de compuertas que las implementan en términos de área, reducir los caminos críticos (explicados más adelante) y/o reducir consumo de potencia. La segunda cambia compuertas genéricas por otras pertenecientes a la librería de tecnología a utilizar, de manera de poseer información completa sobre las características de tiempo, área y potencia de la lista de compuertas interconectadas y optimizadas.

La herramienta utilizada para la síntesis lógica es Design Compiler® de Synopsys®. El flujo de diseño típico para esta etapa ([25]), en particular para la herramienta, se muestra a continuación:

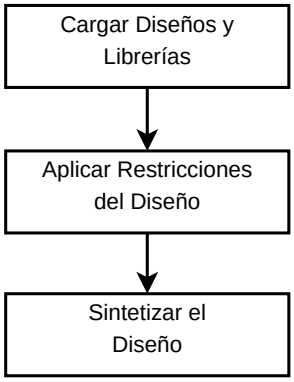


Figura 4.22: Diagrama de Flujo del Proceso de Síntesis Lógica

4.4.1. Especificación de Librerías

El primer paso en el proceso de síntesis lógica es definir las librerías de tecnología, *link* y *target*, las cuales contienen información acerca de las características[4] y funciones de cada

[1] Pertenecientes a la librería *GTECH*.
[2] Por ej., el tiempo que toma en cambiar la salida frente a un cambio en la entrada o tiempo de *setup*.
[3] Carga capacitiva en puertos de la compuerta.
[4] Nombre de celdas, de pines, arcos de retraso, carga de pines, reglas de diseño y condiciones de operación.

celda que provee la librería de un fabricante de semiconductores. Además se define la librería de símbolos, la cual contiene los símbolos esquemáticos para la visualización de las celdas de las librerías de tecnología.

La elección del fabricante de semiconductores y tecnología a utilizar en el chip es muy relevante, ya que cada uno tiene características diferentes y se puede adecuar de mejor manera al diseño realizado. Las principales características a considerar son las siguientes ([26]):

- Máxima frecuencia de operación.
- Restricciones físicas y de potencia.
- Restricciones de empaquetamiento.
- Implementación del *clock tree*.
- Planificación de plano[1].
- Soporte de retroalimentación de datos obtenidos.
- Soporte de diseño para librerías, *megacells* y memorias RAM.
- Areas núcleares (centrales) disponibles.
- Métodos de prueba disponibles.

La librería *target* es usada durante la síntesis (o también llamada compilación) para la creación de una lista de compuertas interconectadas pero utilizando las celdas contenidas en ella, por lo tanto, una lista con tecnología específica [25]. Design Compiler® selecciona durante la compilación las compuertas que mejor se ajusten de acuerdo a las restricciones impuestas.

La librería *link* permite resolver las referencias[2] de un diseño. Design Compiler® busca en las librerías de tecnología, indicadas por la librería *link*, los nombres de celdas coincidentes con la referencia.

4.4.1.1. Librerías Utilizadas

La elección de las librerías en el diseño del controlador se basó en, entre las disponibles para el autor, la que permitiese trabajar con mayor frecuencia de reloj. Una frecuencia de 480 [MHz] implica un periodo de reloj de aproximadamente 2.083 [ns], lo cual es bastante exigente para las librerías disponibles.

El método de elección fue explorativo, de manera que se realizó síntesis lógica y reporte de resultados de características de tiempo del diseño con distintas librerías, para escoger la

[1]Del inglés, *Floorplanning*.
[2]Una referencia es una compuerta, bloque o sub-diseño que es instanciada en el diseño [25].

más adecuada. Ninguna logró cumplir las restricciones para todos los módulos, pero una tuvo mejor resultado que el resto. Las librerías finalmente utilizadas se muestran en la tabla 4.11.

Librería *Target*	cb13fs120_tsmc_max
Librería *Link*	cb13fs120_tsmc_max
	cb13io320_tsmc_max
	ram16x128_max
Librería de Símbolos	sc.sdb

Tabla 4.11: Librerías Utilizadas

Las principales características de la librería *target* son las mostradas en la tabla 4.12.

Unidad de Tiempo	$1[ns]$
Unidad Carga Capacitiva	$1[pF]$
Unidad de Resistencia	$1[k\Omega]$
Unidad de Voltaje	$1[V]$
Unidad de Corriente	$1[\mu A]$
Unidad de Energía Dinámica	$1[pJ]$
Unidad de Pérdida de Potencia	$1[pW]$

Tabla 4.12: Características Librería *Target*

4.4.2. Lectura del Diseño

El segundo paso es la lectura del diseño, lo cual se realiza utilizando el comando `read_verilog` en la línea de comando de DC[1]. Design Compiler® utiliza HDL Compiler de Synopsys® para la traducción de descripciones HDL Verilog a representaciones internas y equivalentes de compuertas [27]. Esta representación interna es luego optimizada y transformada en una lista de compuertas pertenecientes a una tecnología específica. HDL Compiler además revisa la sintaxis de la descripción RTL del diseño y realiza optimización arquitectural.

4.4.3. Definición de Ambiente del Diseño

Antes de optimizar el diseño, se debe definir el ambiente en el cual el diseño debería operar. El ambiente se define al especificar las condiciones de operación, modelos de *wire load* y características de la interfaz del sistema [26].

[1] *DC Shell*, ambiente de línea de comandos de Design Compiler®.

4.4.3.1. Condiciones de Operación

Existen tres factores principales que pueden afectar de manera importante al desempeño[1] de un circuito, éstos son [26]:

- **Variaciones en la Temperatura de Operación:** Este factor es inevitable en la operación diaria de un diseño. Los efectos causados por la variación de la temperatura son a menudos tratados linealmente.
- **Variaciones en el Voltaje de Alimentación:** El voltaje de alimentación tampoco se mantiene constante en el día a día. Se realizan cálculos complejos en general, pero a nivel de la lógica del circuito se utiliza como un factor lineal.
- **Variaciones en el Proceso de Fabricación:** Son las diferencias que puede haber con respecto al producto esperado en el proceso de fabricación. Los efectos se cuantifican como una variación porcentual en el cálculo del desempeño.

La librería utilizada posee un sólo escenario de operación, por lo tanto no existe alternativa:

Nombre Condición de Operación	cb13fs120_tsmc_max
Proceso (adimencional)	1.20
Temperatura [°C]	125.00
Voltaje [V]	1.08
Modelo de Interconexión	*worst_case_tree*

Tabla 4.13: Condiciones de Operación de Librería

El modelo de interconexión o también llamado tipo de árbol[2] es el modelo utilizado para la estimación de la distribución de la resistencia y capacitancia de cada red [28]. En el *worst_case_tree*, o peor caso, se considera que una celda receptora está lo suficientemente lejos (físicamente) de la emisora, de manera que toda la resistencia de la red está entre la emisora y cargas capacitivas, lo que se traduce en menor corriente y mayores retrasos.

4.4.3.2. Modelos *Wire Load*

Los modelos *wire load* permiten estimar los efectos de la longitud y *fanout* de las redes de interconexión de compuertas, en su resistancia, capacitancia y área. Una vez estimados estos valores, Design Compiler® puede calcular los retrasos producidos por las redes.

[1] Velocidad a la que opera el circuito.
[2] Del inglés, *Tree Type*.

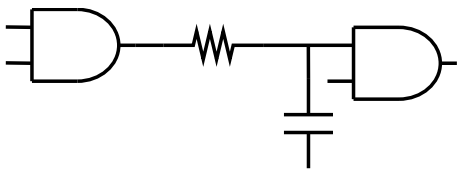


Figura 4.23: Modelo *Wire Load*

Las librerías poseen varios modelos *wire load* para distintos tamaños de diseño, muchas veces medido en cantidad de compuertas. Para el diseño del controlador se dejó que la elección del modelo sea realizado automáticamente por Design Compiler® durante la compilación.

4.4.3.3. Modelamiento de la Interfaz del Sistema

El modelamiento de la interfaz del sistema consiste en la especificación de las características en entradas y salidas del diseño que pudiesen afectar el desempeño de él. El objetivo es determinar el tiempo de transición en entradas y salidas.

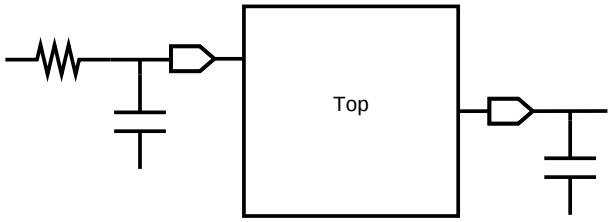


Figura 4.24: Modelamiento de Interfaz del Sistema

4.4.3.3.1. Características de Puertos de Entrada

El tiempo de transición de un puerto de entrada afecta directamente al retardo en la compuerta de entrada conectada a ella. Este tiempo puede ser especificado directamente o a través de la caracterización de los puertos de entrada.

En caso de caracterizar los puertos de entrada, el tiempo de transición se calcula como el producto entre la resistencia conductora[1] de entrada y su carga capacitiva [26], por lo tanto estos dos elementos deben ser definidos.

Se denomina "Fuerza Conductora" al inverso de la resistencia conductora. Una señal de entrada muy débil [10] es equivalente a una resistencia conductora de entrada muy grande, la cual debe ser amplificada utilizando *buffers*, los cuales a su vez ingresan más retrasos al diseño. A los puertos de entrada que tienen mucha carga, como el reloj, se les debe especificar una resistencia nula, de manera que Design Compiler® no llene de *buffers* toda la red [26].

[1]Del inglés, *Drive Resistance*.

Para el diseño no se especificaron estas restricciones, dado el desconocimiento de las características del sistema objetivo y el circuito analógico. Omitir este tipo de restricciones no es tan grave si es que otras se restringen más, pero el resultado es menos realista en términos del modelamiento del circuito.

4.4.3.3.2. Características de Puertos de Salida

El tiempo de transición de un puerto de salida afecta directamente al retraso de la primera compuerta de la celda de salida a la que se conecte. La carga capacitiva en un puerto de salida afecta directamente a este tiempo.

Además, se puede especificar la carga de *fanout*, un número adimensional que representa una contribución numérica al *fanout* total. El total considera tanto la cantidad de celdas que se espera que se conecten a la salida, como las que se conectan por dentro del diseño, y debe ser menor al límite establecido por las librerías y el diseño.

No se especificó la carga capacitiva, dado el desconocimiento del sistema objetivo y circuito analógico. Las salidas se restringieron a una carga de *fanaout* de 5 unidades.

4.4.4. Definición de Restricciones de Diseño

Las restricciones definen el objetivo del diseño en términos de las características medibles del circuito, como las temporales, área y capacitancia [26]. Existen dos tipos: restricciones de reglas de diseño y restricciones de optimización. Las primeras tienen prioridad sobre las segundas.

En este apartado se describen una serie características consideradas por Design Compiler® para un cálculo más realista de los distintos caminos de tiempo[1] y área para optimizar el diseño.

4.4.4.1. Reglas de Diseño

Las restricciones de reglas de diseño reflejan las restricciones específicas a la tecnología utilizada, y el diseño realizado las debe cumplir. Es posible hacer más restrictivas las reglas de diseño impuestas por las librerías, pero no menos.

Las reglas de diseño abarcan las siguientes características:

[1]Del inglés, *Timing Path*. Se refiere al tiempo que tomaría una señal en reflejar un cambio entre dos puntos distintos de un circuito.

4.4.4.1.1. Máximo Tiempo de Transición

El máximo de tiempo de transición para una red es el tiempo más largo requerido por su respectivo pin de origen para cambiar de valor lógico. Esta característica no se restringió más de lo que indica la librería.

4.4.4.1.2. Máximo *Fanout*

El máximo *fanout* pone una restricción en cada pin de salida que se conecta a una red, la cual a su vez está conectada a un número determinado de pines de entrada de distintas compuertas. La suma de cargas *fanout* de las compuertas conectadas a la red debe ser menor o igual al máximo especificado. Esta característica no se restringió más de lo que indica la librería.

4.4.4.1.3. Mínima y Máxima Capacitancia

La máxima capacitancia define el máximo valor de capacitancia a la que un pin de salida se puede conectar, en otras palabras, un pin de salida se puede conectar a una red, si la capacitancia total de ésta es menor al valor definido por la librería.

Por otro lado, la mínima capacitancia es el mínimo valor de capacitancia de una red a la que un pin de entrada se puede conectar.

4.4.4.1.4. Degradación de Celdas

La degradación de celdas es un valor similar al de máxima capacitancia. Éste también establece un valor máximo de capacitancia a un pin de salida, con menor que prioridad que máxima capacitancia, pero en función del tiempo de transición en las entradas de la celda. Las librerías pueden tener una tabla de valores máximos de capacitancia en función de los tiempos de transición.

4.4.4.2. Restricciones de Optimización

Las restricciones de optimización corresponden a los objetivos de velocidad y área del diseño. Como ya se mencionó, las reglas de diseño tienen mayor prioridad que las restricciones de optimización, pero dentro de éstas, las restricciones de velocidad (de tiempo o temporales) tienen mayor prioridad que las de área.

4.4.4.2.1. Reloj

En casi cualquier diseño la parte más crítica de la síntesis es la descripción del reloj [3]. En la etapa de síntesis lógica no sé realiza balance de *skew*[1] o síntesis del *clock tree*, en otras palabras, Design Compiler® por defecto no coloca *buffers* en la red del reloj para balancear todas sus bifurcaciones. Es por esto que se recomienda modelar el reloj, estimando su *skew*, latencia y tiempos de transición entre otros.

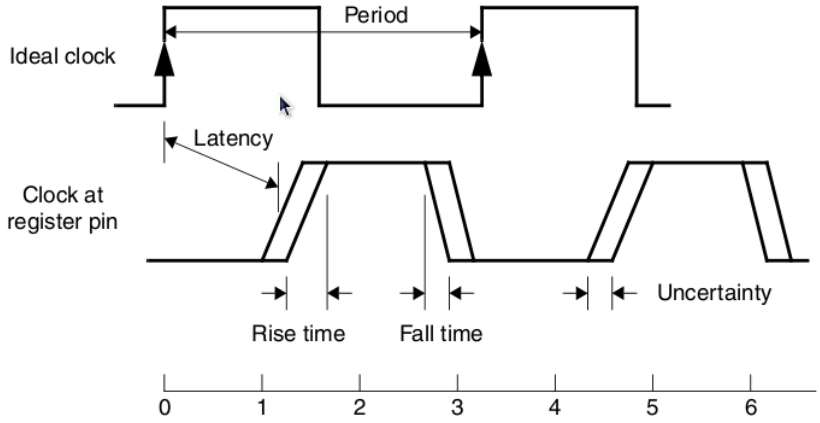


Figura 4.25: Características del *Clock* [28]

En la figura 4.25 se muestran las principales características a considerar en el modelamiento del reloj. La primera forma de onda muestra un reloj ideal a modo de referencia para la segunda forma de onda que es el reloj modelado. Las características se describen en la tabla 4.14.

Característica	**Descripción**
Period	Define el periodo (y frecuencia) del reloj. Las unidades dependen de la librería.
Latency	Define la latencia o retraso total del reloj que ingresa al sistema. En un diseño con un solo reloj se puede ignorar, ya que todos los registros son retrasados por la misma latencia [25].
Rise Time	Corresponde al tiempo de transición de 0 a 1 lógico (flanco de subida).
Fall Time	Corresponde al tiempo de transición de 1 a 0 lógico (flanco de bajada).
Uncertainty	La incertidumbre[2] modela el máximo retraso entre las distintas bifurcaciones de la red del reloj, pero puede incluir también el *jitter*[3] y efectos marginales [25].

Tabla 4.14: Características del *Clock*

Además de las características anteriores, es posible definir el ciclo de trabajo del reloj.

[1] Es la máxima diferencia o retraso entre las distintas ramas de la red del reloj [25].
[2] Del inglés, *uncertainty*.
[3] *Jitter* es la variación de la fase de una señal periódica.

En la tabla 4.15 se muestra el reloj creado para el diseño del controlador USB. [3] recomienda restringir en un 10 % más de lo necesario, de manera de evitar muchas iteraciones entre las etapas de síntesis lógica y física.

Característica	**Valor**	**Unidad**
Period	15	ns
Latency	0.2	ns
Rise Time	0.1	ns
Fall Time	0.1	ns
Uncertainty	0.2	ns
Ciclo Trabajo	50	%

Tabla 4.15: Definición del Reloj del Controlador

La tabla 4.16 muestra la definición del reloj proveniente del circuito analógico. Éste posee incerteza mayor para modelar la diferencia de frecuencia que puede existir entre el reloj de un dispositivo y el del controlador.

Característica	**Valor**	**Unidad**
Period	15	ns
Latency	0.2	ns
Rise Time	0.1	ns
Fall Time	0.1	ns
Uncertainty	0.3	ns
Ciclo Trabajo	50	%

Tabla 4.16: Definición del Reloj Circuito Analógico

El periodo del reloj se fijó finalmente en 15[ns] debido a que el requerido para operar a 480[MHz] resultó en caminos con retrasos muy cercanos al límite de las restricciones. Es posible seguir mejorando la descripción RTL del controlador, pero ésto queda propuesto para un trabajo futuro. También es posible trabajar a una frecuencia más lenta en el controlador y enviar y recibir datos a frecuencias mayores, lo cual requiere de algunas modificaciones a los módulos existentes y creación de módulos *buffer*, lo cual también queda propuesto. El objetivo principal de la memoria es constatar el diseño de un circuito integrado, por lo tanto el periodo del reloj se ajustó de manera de que las restricciones sean cumplidas con tiempo de reloj de sobra.

4.4.4.2.2. Restricciones de Optimización

Como ya se mencionó, las restricciones de optimización corresponden a los objetivos de velocidad y área del diseño. Para las primeras se definen cuatro tipos de caminos de tiempo (Fig. 4.26) los cuales pueden ser restringidos para luego optimizarlos.

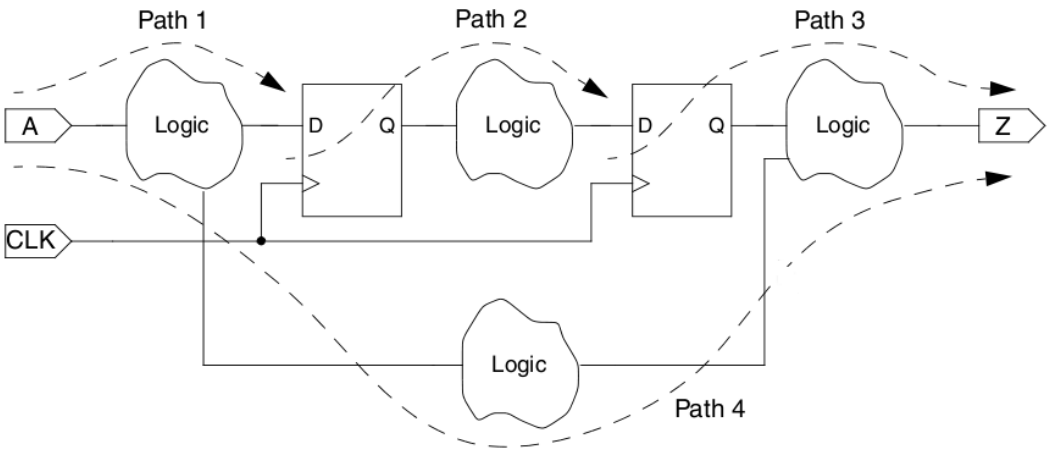


Figura 4.26: Caminos de Tiempo a Restringir [28]

Cada camino se identifica de acuerdo a su punto de inicio y fin en el diseño, como se aclara en la tabla 4.17. Un punto de inicio es un lugar en el diseño donde datos son gatillados por un flanco de reloj. Esos datos son propagados a través de lógica combinacional y luego capturados en un punto final por otro flanco de reloj [28].

Camino	**Denominación**
Path 1	Camino de Entrada
Path 2	Camino Registro-Registro
Path 3	Camino de Salida
Path 4	Camino Combinacional

Tabla 4.17: Identificación de Caminos de Tiempo

Restricción Camino de Entrada

Como se mencionó anteriormente, el retraso total de cada camino es calculado entre los flancos lanzadores y capturadores de dos elementos secuenciales. En el caso del camino de entrada se desconoce el tiempo de lanzamiento y lógica externa al diseño. La situación se ilustra en la figura 4.27.

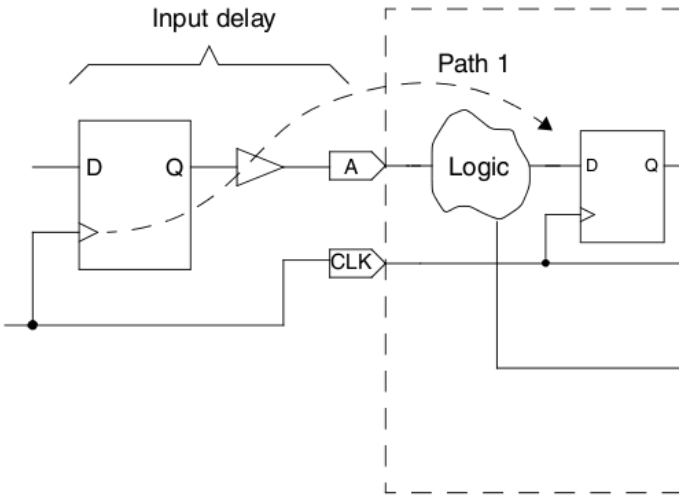


Figura 4.27: Camino de Entrada [28]

Lo que se requiere entonces es especificar el retraso desde el flanco de reloj de llegada de datos en el flip-flop externo hasta el puerto de entrada "A".

La restricción que se utilizó para todos los caminos de entrada al módulo maestro es de 1[ns].

Restricción Camino Registro-Registro

Un camino registro-registro es restringido directamente por el periodo del reloj. En el caso en que los registros no sean gatillados por un mismo tipo de flanco de reloj[1], también se considera el ciclo de trabajo del reloj.

Restricción Camino de Salida

La lógica exterior a un puerto de salida y flanco de captura no son del conocimiento del módulo a optimizar, similar a lo que ocurre con un camino de entrada, por lo tanto deben ser definidos para restringir el camino de salida. Ésto se ilustra en la figura 4.28.

[1]Flanco de subida o flanco de bajada.

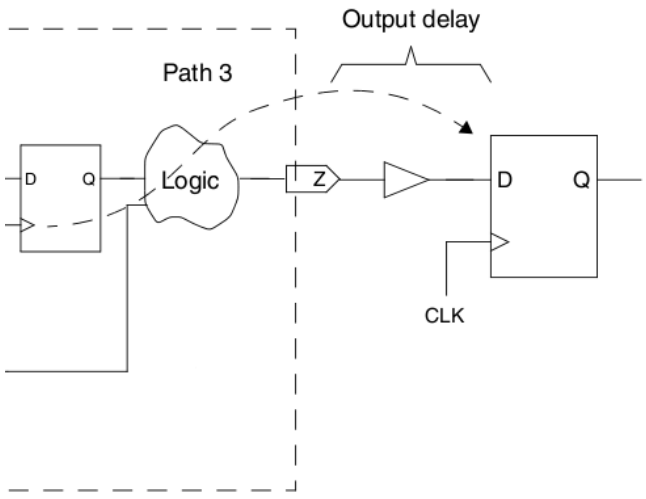


Figura 4.28: Camino de Salida [28]

Por lo tanto, se debe especificar el retraso existente entre el puerto de salida "Z" y el flanco capturador del elemento secuencial externo.

La restricción que se utilizó para todos los caminos de salida del módulo maestro es de 1[ns].

Restricción Camino Combinacional

La situación de un camino combinacional es una mezcla entre un camino de entrada y salida. Como no hay elementos secuenciales en el camino dentro del diseño, se deben especificar los retrasos entre el flanco de lanzamiento de datos del reloj del flip-flop externo de entrada hasta el puerto de entrada, y entre el puerto de salida y el flanco de captura del elemento secuencial de salida.

Como para todos los puertos de entrada y salida del diseño del controlador USB fueron restringidos de igual manera, los caminos combinacionales poseen las mismas restricciones.

4.4.4.2.3. Restricción de Área

El diseño del controlador es exigente en términos de la velocidad, por lo cual no se especificó ninguna restricción de área. Esto significa que Design Compiler® realiza esfuerzos para disminuir área siempre cuando pueda si es que esto no va en desmedro de la optimización de los retrasos del diseño.

4.4.5. Compilación del Diseño

Se denomina compilación o sintetización del diseño a la etapa de mapeo[1] del diseño a una combinación óptima de celdas de la librería *target*, basado en los requerimientos funcionales, de velocidad y área.

4.4.5.1. Proceso de Optimización

El proceso de optimización es llevado a cabo en tres niveles por Design Compiler®:

4.4.5.1.1. Optimización Arquitectural

Este nivel de optimización se realiza sobre la descripción HDL del diseño (sin mapear). Algunas de los tareas que lleva a cabo son identificar expresiones comunes para compartirlas, compartir recursos y reordenar operadores. El resultado es el diseño representado por una lista interconectada de compuertas genéricas, independientes de cualquier tecnología.

4.4.5.1.2. Optimización a Nivel Lógico

Este nivel de optimización se realiza sobre la lista de compuertas resultante de la optimización arquitectural. Se realizan dos procesos:

- **Estructuración:** Este proceso es utilizado principalmente para reducir área en caminos no críticos, basado en las restricciones. Se agregan variables y lógica intermedia al diseño.
- ***Flattening*:** Este proceso trata de convertir los caminos combinacionales del diseño en una representación de suma de productos de 2 niveles. No depende de las restricciones.

4.4.5.1.3. Optimización a Nivel de Compuertas

Este nivel de optimización trabaja sobre la lista de compuertas genérica, para producir una lista interconectada de compuertas pertenecientes a una tecnología específica. Los siguientes procesos son llevados a cabo:

- **Mapeo:** Utiliza compuertas de las librerías de la tecnología objetivo para generar una implementación a nivel de compuertas del diseño.
- **Optimización de Retrasos:** Trata de arreglar las violaciones de las restricciones de optimización introducidas (no incluídas las de área) por la fase de mapeo.

[1]Transformación de una representación del diseño a otra.

- **Corrección de Reglas de Diseño:** Corrige las violaciones de las reglas de diseño agregando buffers utilizando celdas de distinto tamaño. En caso de necesitarlo, genera violaciones de las restricciones de optimización.
- **Optimización de Área:** Trata de alcanzar las metas de área sin introducir violaciones de las otras restricciones.

4.4.5.2. Estrategias de Compilación

Existen tres estrategias básicas de compilación de un diseño jerarquizado:

4.4.5.2.1. Compilación *Top-Down*

En esta estrategia, el módulo de nivel más alto[1] es compilado junto con todos sus sub-módulos. Es utilizado para diseños que caben completos en la memoria del CPU. La ventaja de la compilación *top-down* es que es más directa (menos pasos) y se encarga automáticamente de las dependencias entre bloques.

4.4.5.2.2. Compilación *Bottom-Up*

La estrategia *bottom-up* es recomendada para diseños de tamaño medio a grande. Las principales ventajas son que utiliza el enfoque "dividir para conquistar" [26] y que requiere menos memoria que la estrategia *top-down*.

En esta estrategia cada sub-módulo es compilado por separado y después incorporado en el módulo maestro. Esto implica que las interfaces entre módulos deben ser especificadas. Al comienzo es probable que las restricciones para las interfaces entre bloques no sean precisas, por lo cual se requieran de más iteraciones de manera de ir mejorándolas y obtener resultados más realistas.

Una vez compilados los sub-módulos, estos se juntan con el módulo maestro, se aplican restricciones: si se satisfacen, no se requiere de más iteraciones; si no, se debe extraer información de los resultados para precisar mejor las dependencias entre bloques.

4.4.5.2.3. Compilación Mezclada

Esta estrategia es una combinación de las dos anteriores, de manera que se utilizan según se estime conveniente en cada sub-módulo.

[1] Módulo *Top* o maestro.

4.4.5.3. Estrategia Utilizada

Inicialmente, cada módulo fue compilado por separado de manera de aislar y corregir sus errores. Las principales tareas que se realizaron para cada módulo son:

- Verificar inferencias de *latches*[1]: Un *latch* es más dificil de restringir, requiere de métodos diferentes a los flip-flops. Se pueden evitar con una adecuada descripción HDL.
- Verificar señales con mismo comportamiento: En varios módulos se generaron señales que tenían el mismo comportamiento, por lo tanto se utiliza sólo una de ellas.
- Verificar existencia de *loops* combinacionales: Caminos cerrados de lógica combinacional. Pueden provocar errores graves en la funcionalidad del diseño, conocidos como *glitches* y *hazards* [29]. Se pueden romper con un elemento secuencial.

Una vez corregidos los errores anteriores, se utilizó la estrategia de compilación *top-down*, es decir se compiló todo junto.

4.4.5.3.1. Reporte de Retrasos

Design Compiler® permite realizar un análisis de todos los caminos de tiempo del diseño. Es posible obtener un reporte de los caminos críticos, lo cual demostró la existencia de caminos largos con mucha lógica combinatoria.

Design Compiler® define el *slack* de un camino como la diferencia entre el flanco de captura con respecto al tiempo de llegada de los datos. Un *slack* positivo significa que se cumplieron las restricciones, mientras que uno negativo que se violaron. Con un reloj de periodo de 2.0833[ns] correspondiente a 480[MHz] resultaron muchísimos caminos con *slack* negativo (Fig.[2] 4.29), en parte por la cantidad de lógica combinatoria y por las características de las celdas de la librería. Con un periodo de 4[ns] la mayor parte de los caminos resultaron (Fig. 4.30) con *slack* muy cercanos a 0, lo cual indica que es muy probable que en la etapa de síntesis física ya no se pueda cumplir con las restricciones. La principal solución adoptada fue romper los caminos más largos y ciclos de lógica combinatoria con elementos secuenciales.

[1]Elemento de memoria no secuencial.

[2]Un gráfico de *Endpoint Slack* muestra un histograma del *slack* medido para los caminos de tiempo del diseño.

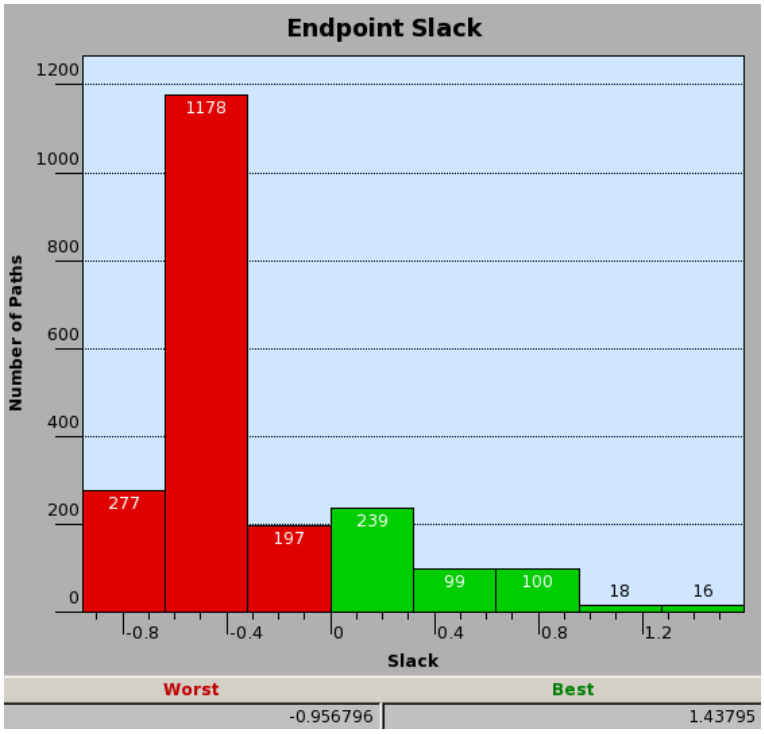


Figura 4.29: Gráfico de *Endpoint Slack* Periodo 2.0833[ns]

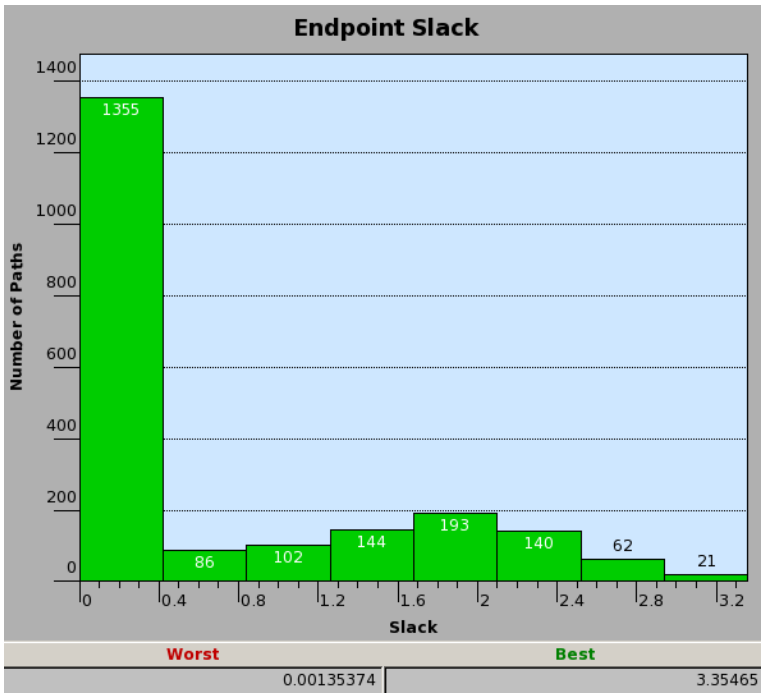


Figura 4.30: Gráfico de *Endpoint Slack* Periodo 4[ns]

Aún acortando gran parte de los caminos y rompiendo los ciclos de lógica combinatoria no fue posible eliminar todos los caminos críticos. Se probó con otros periodos de reloj hasta

que se fijó en 15[ns] como se explicó en la sección 4.4.4.2.1, con lo que se obtuvieron *slacks* mostrados en la figura 4.31.

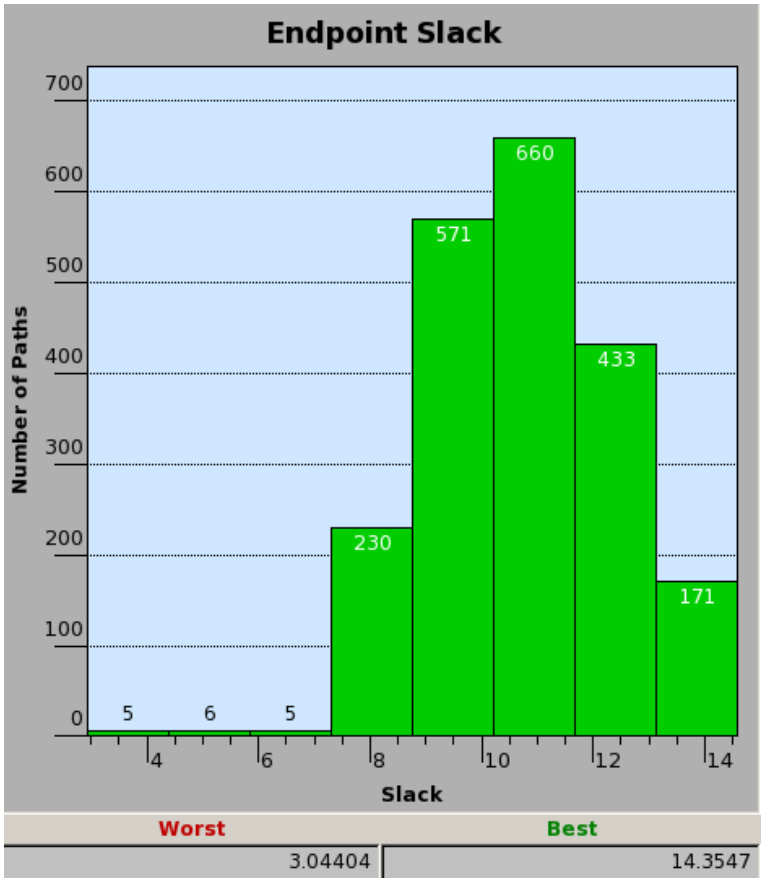


Figura 4.31: Gráfico de *Endpoint Slack* Periodo 15[ns]

4.4.5.3.2. Reporte de Área

Un reporte sobre el área y cantidad de elementos resultantes se muestra en la tabla 4.18

N° de Puertos	163
N° de Redes	2730
N° de Celdas	2475
N° de Referencias	75
Área Combinacional	9902,75[μm^2]
Área no Combinacional	11109,25[μm^2]
Área de Interconexiones	5440,66[μm^2]
Total Área de Celdas	21012,00[μm^2]
Área Total	26452,66[μm^2]

Tabla 4.18: Área del Diseño después de Síntesis Lógica

Se observa que el diseño posee de una gran cantidad de celdas. Cada celda está compuesta por una gran cantidad de transistores, por lo tanto se tiene un tamaño medio de diseño. Éste es almacenado en memoria sin problemas, por lo cual la estrategia *top-down* es aplicable.

4.5. Síntesis Física

Esta etapa del diseño de circuitos integrados permite pasar de la lista de compuertas interconectadas ya optimizada a un archivo GDSII[1]. La herramienta utilizada es IC Compiler®de Synopsys®. El diagrama de flujo del proceso de esta herramienta se muestra en la figura 4.32.

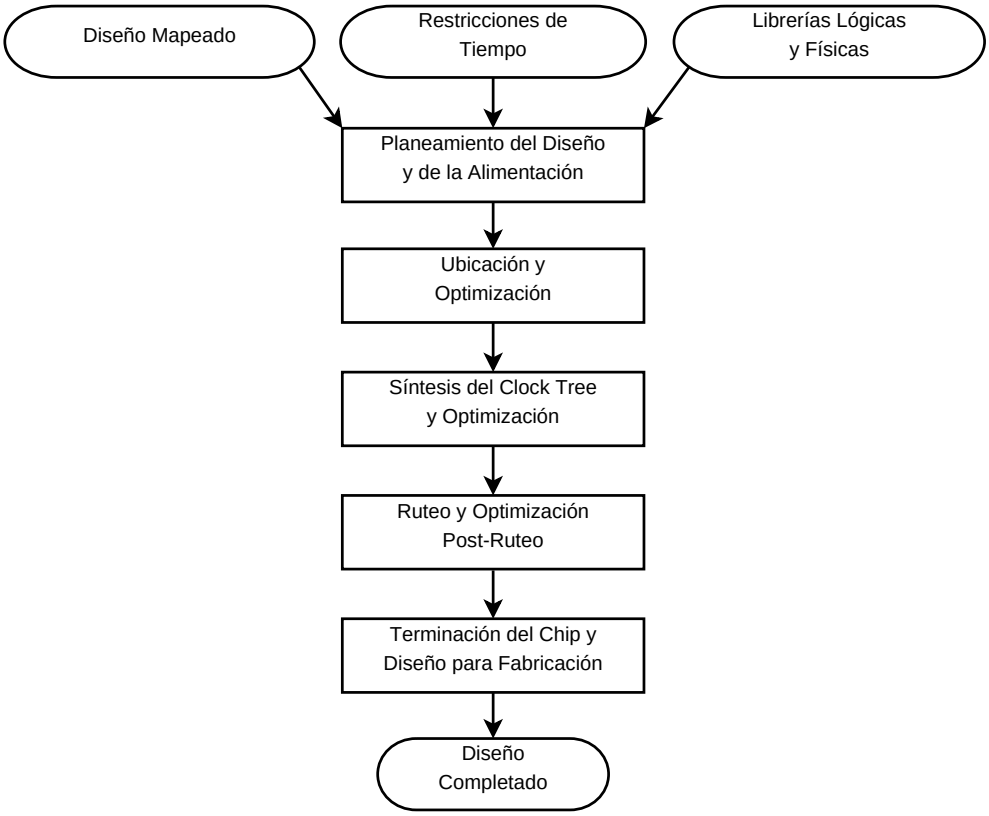


Figura 4.32: Diagrama de Flujo del Proceso de Síntesis Física [31]

4.5.1. Preparación del Diseño

IC Compiler® trabaja con una base de datos propia llamada *Milkyway*. En ésta librería se almacena el diseño (lista de compuertas), restricciones y las librerías asociadas. La

[1]GDSII es el formato estándar de archivo de base datos de *layouts* de circuitos integrados [30].

preparación del diseño consiste en la creación de esta base de datos, proporcionando toda la información necesaria.

4.5.1.1. Librerías

IC Compiler® usa librerías lógicas para obtener la información de retrasos y funcionalidad de todas las celdas estándar. Además, éstas pueden poseer la información de *hard macros*[1] como memorias RAM.

IC Compiler® también utiliza librerías Milkyway de referencia y archivos de tecnología para obtener la información de las características físicas de las celdas. Las librerías de referencia Milkyway contienen información sobre las características físicas de las celdas estándar y macro de la librerías de tecnología, además de definir la unidad de la teja[2] de ubicación. Los archivos de tecnología poseen la información específica de la tecnología utilizada, como los nombres y características de cada capa de metal.

Adicionalmente, se deben especificar los archivos *TLU*, los cuales poseen tablas especializadas para obtener los coeficientes RC, incluyendo los efectos del ancho, espacio, densidad y temperatura de las resistencias.

Las librerías lógicas utilizadas corresponden a las mismas de la síntesis lógica, mientras que las físicas son una versión especial de Milkyway de ellas.

4.5.1.2. Lectura del Diseño

Luego de especificadas las librerías y archivos necesarios se procede a cargar el diseño del controlador ya sintetizado, por lo tanto lo que se lee es una lista de compuertas optimizada y mapeada a una tecnología específica.

En este paso también se aplican las mismas restricciones de tiempo utilizadas en la etapa de síntesis lógica.

4.5.2. Planeamiento del Diseño

El planeamiento del diseño consiste principalmente en la definición del tamaño del área nuclear y periférica, ubicación general de celdas estándar y *hard macros*, y planeamiento de la malla de alimentación del chip.

[1] Celdas de mayor tamaño y con funcionalidades más complejas.
[2] Del inglés, *tile*.

La importancia de realizar el planeamiento del diseño es que permite revisar rápidamente la viabilidad de distintas estrategias en la implementación física del diseño [32]. Además, un buen planeamiento hace que las etapas siguientes de la síntesis física sean mucho más efectivas.

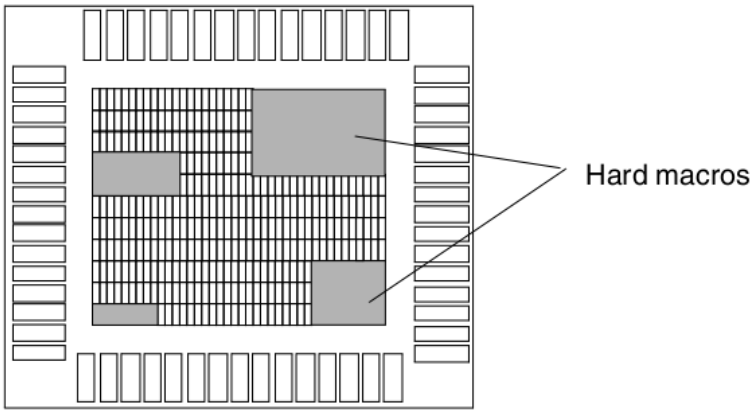


Figura 4.33: Planeamiento del Diseño [33]

4.5.2.1. Restricciones de I/O Pads

Antes de definir las características del chip explicadas en las siguientes secciones, se deben especificar restricciones físicas para los *pads*[1] de E/S (entrada/salida). Para cada pad se señala (no obligatoriamente) el lado del área nuclear en que debe estar, el orden con respecto a los otros pads, distancia a la que debe estar del borde del chip y distancia entre pads entre otros.

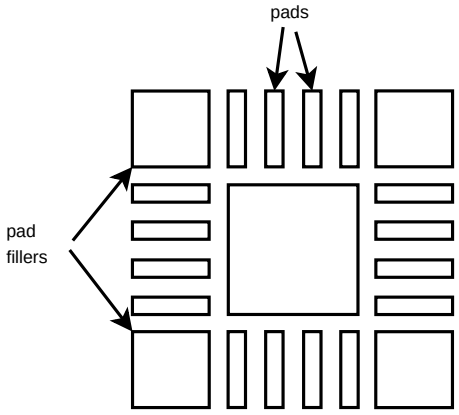


Figura 4.34: *Pads* y *Pads Fillers*

No se definió ningún pad en el diseño, ya que éste no está completo en términos de su interfaz con el sistema objetivo. La cantidad de señales de entrada y salida es muy grande como para considerar colocar pads y dar un carácter de chip completamente especificado.

[1]Es una superficie plana utilizada para realizar contacto eléctrico [34].

4.5.2.2. Definición del Área Nuclear

El área nuclear o *core* es lugar del chip donde se ubican todas las celdas. Se pueden utilizar distintas formas rectilíneas para el *core*, en particular, el rectángulo para este diseño.

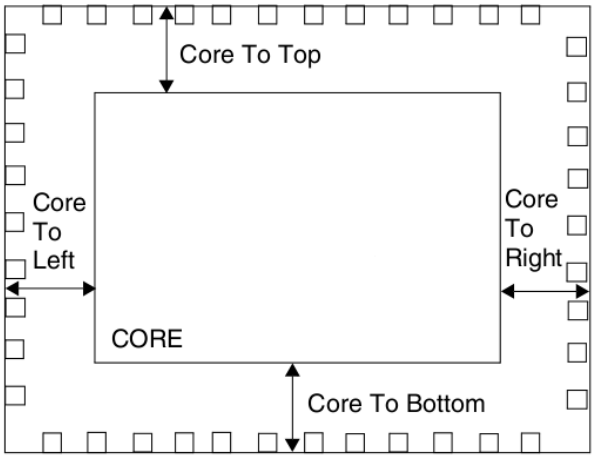


Figura 4.35: Área Nuclear [32]

El área del *core* puede ser especificada de las siguientes maneras:

- Razón de Aspecto: Alto dividido por el ancho del rectángulo.
- Alto y Ancho: Los valores exactos.
- Número de Filas[1].

Dependiendo del método escogido se debe especificar más información al respecto. Por ejemplo, en el caso de "razón de aspecto" se requiere de otro dato para determinar el área. En particular, se especificó la **Utilización del *Core***. Este valor es un porcentaje del área que utilizan las celdas del diseño con respecto al total, por lo tanto de esta manera se puede controlar fácilmente el porcentaje del área que se quiere dejar para el ruteo de las celdas. Se especificó un porcentaje de utilización de 70 %, de manera que se destinó el restante 30 % para rutear todas las redes.

La distancia desde el *core* al comienzo de los pads (Fig. 4.35) para cada lado también debe ser definida, en caso de que existan pads en el diseño.

[1]Una fila es el lugar, de ancho o alto de un *tile*, donde pueden ser ubicadas las celdas dentro del área nuclear [25].

4.5.2.3. Ubicación General de Celdas

Durante el planeamiento del diseño se pueden definir distintos tipos de áreas dentro del *core*, las principales son:

- Zona de Celdas Estándar: En estos lugares van todas las celdas estándar.
- Zona de *Hard Macro*: Lugar donde se ubican celdas de mayor tamaño como una memoria RAM.
- Área de Bloqueo: Lugar donde no se puede ubicar ningún tipo de celda.

En la figura 4.36 se muestra un ejemplo donde el área que no es de bloqueo ni RAM corresponde al lugar donde se ubicarían las celdas estándar.

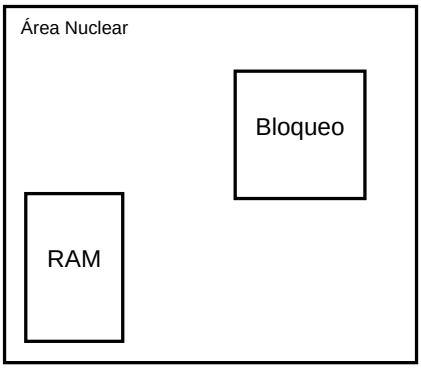


Figura 4.36: Definición de Áreas

Una buena definición de áreas puede contribuir a alcanzar mejores resultados de ruteo de celdas y reloj. Para el diseño del controlador no es necesario definir áreas, ya que se solo se tienen celdas estándar.

4.5.2.4. Planificación de la Alimentación

El planeamiento de la alimentación del chip consiste principalmente en la definición de anillos y bandas de metal para la alimentación y tierra del circuito. Previo a ésto, se deben conectar los pines de alimentación y tierra de las celdas y los pines de señales conectados a 1's y 0's lógicos, a las redes de alimentación y tierra.

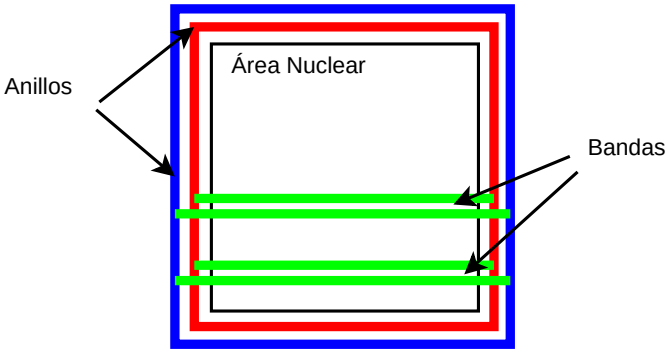


Figura 4.37: Malla de Alimentación

4.5.2.4.1. Conexión a Alimentación y Tierras

El diseño resultante de la síntesis lógica no posee la conexión explícita de los *pines* de alimentación y tierra de las compuertas a sus respectivas redes, como se muestra en la figura 4.38. La preparación del diseño considera un paso donde se realiza la conexión explícita, como se muestra en la figura 4.39.

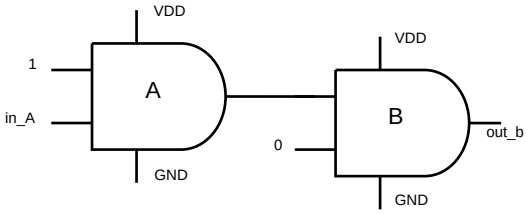


Figura 4.38: Celdas Desconectadas

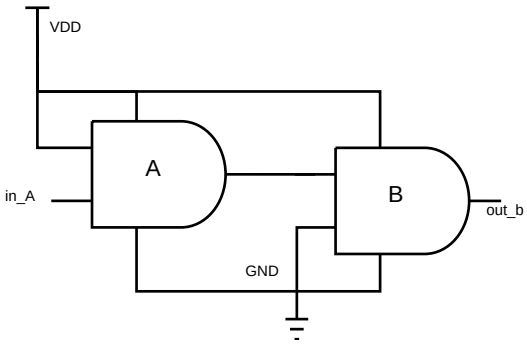


Figura 4.39: Celdas Conectadas

4.5.2.4.2. Creación de Anillos de Alimentación y Tierra

Después de la planificación base y conexión de pines de alimentación y tierra, se deben crear los anillos de alimentación y tierra alrededor del área nuclear (Fig. 4.37). Éstos permiten la conexión de las bandas de alimentación y tierra, además de las celdas cercanas.

Los anillos en el diseño fueron creados de manera que las franjas horizontales estuviesen en una capa de metal distinta que las verticales, tanto para la alimentación y la tierra.

4.5.2.4.3. Creación de Bandas de Alimentación y Tierra

Las bandas de alimentación y tierra cubren espaciadamente[1] (Fig. 4.37) el área nuclear, de manera de proporcionarle a las celdas lugares cercanos para conectarse.

Para el diseño se definieron 3 bandas verticales y 1 horizontal.

4.5.2.4.4. Pre-Ruteo de Alimentación y Tierra

Una buena práctica es rutear la alimentación y tierra previo al ruteo global y síntesis de la red del reloj. Ésto permite que la herramienta que realiza el ruteo global detecte obstrucciones [32]. Además de realizar el pre-ruteo, se crearon rieles horizontales de alimentación y tierra en las filas sin celdas, de manera de proveer lugares más cercanos para conectarse.

4.5.2.5. Diseño Planeado

El diseño ya planeado se muestra en la figura 4.40. Las objetos en los bordes del área nuclear corresponden a los puertos de entrada y salida. Hay una gran cantidad de estos puertos debido a que la interfaz con el sistema objetivo no esta debidamente definida.

[1] Puede ser uniforme o no uniformemente.

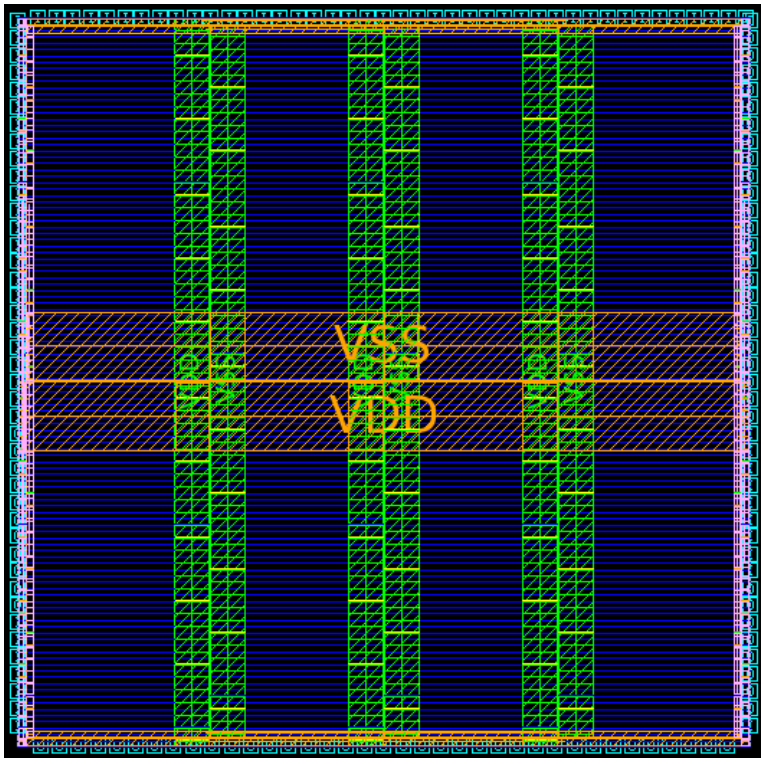


Figura 4.40: Diseño Planeado

4.5.3. Ubicación de Celdas

La ubicación de celdas en el área definida después de la planificación del diseño, es llevada a cabo practicamente en un solo paso por IC Compiler®.

El objetivo principal de este paso de la síntesis física es ubicar todas las celdas del diseño de manera óptima, evitando congestionar[1] algunos sectores y colocar celdas conectadas muy separadas. IC Compiler® utiliza un motor de emplazamiento directamente dirigido por las restricciones de tiempo del diseño [25].

La ubicación de las celdas se realizó con esfuerzo máximo y minimizando la congestión. En la figura 4.41 se muestra el mapa de congestión para ruteo global obtenido después de este proceso. El mapa muestra los bordes entre las celdas globales de ruteo (ver sección 4.5.5.1) con diferentes colores para denostar el nivel de sobre-flujo[2]. El orden creciente de nivel de congestión es representado por los colores azul (mínimo), celeste, verde, amarillo, naranjo y rojo (máximo). De la figura se observa que el nivel de congestión es mínimo para el diseño.

[1]La congestión es medida en función de la cantidad de conexiones que pasan por un mismo sector.

[2]Del inglés, *Overflow*. Es un valor que indica la cantidad de redes o conexiones que no tienen una pista disponible.

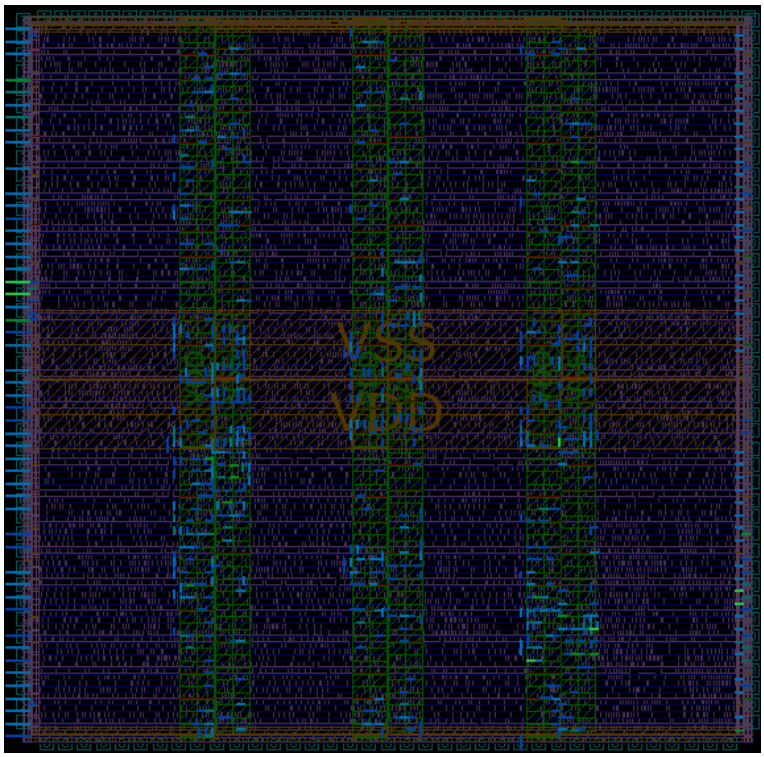


Figura 4.41: Mapa de Congestión para Ruteo Global

4.5.4. Síntesis del *Clock Tree*

Esta etapa tiene su mayor relevancia en el hecho de que es esencial controlar la latencia y *skew* del reloj [3]. La herramienta determina la mejor ubicación y estilo del *clock tree*, de manera de balancear y minimizar sus niveles.

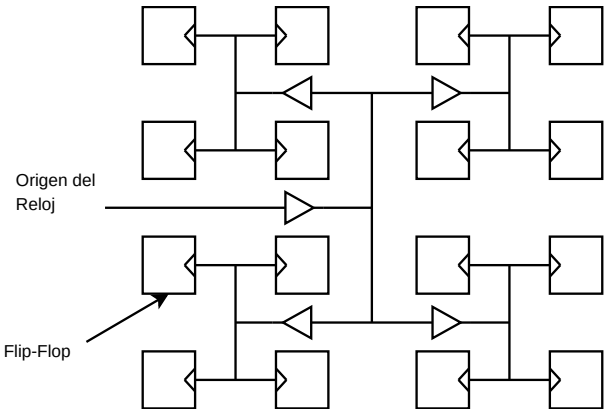


Figura 4.42: Ejemplo de *Clock Tree*

IC Compiler® sugiere que los siguiente requerimientos se cumplan antes de realizar la síntesis de la red del reloj [31]:

- Las celdas del diseño deben estar óptimamente ubicadas. En particular se deben tener resultados aceptables respecto a la congestión, restricciones de tiempo, capacitancia máxima y tiempo de transición máximo.
- Las redes de alimentación y tierra deben estar pre-ruteadas.
- Las redes con *fanout* alto deben estar sintetizadas con *buffers*.

Mediante un solo comando, `clock_opt`, se llevan a cabo los procesos de síntesis, optimización del *clock tree* y optimización física incremental, entre otros.

Antes de la síntesis del *clock tree*, IC Compiler® agranda y posiblemente mueve las compuertas ya existentes del reloj, lo cual puede mejorar la calidad de los resultados y reducir el número de niveles del *clock tree* [31]. Luego, la red del reloj es construida de manera de cumplir las reglas de diseño, al mismo tiempo de balancear la carga y minimizar el *skew*.

La optimización del *clock tree* utiliza técnicas como re-ubicación de *buffers* y compuertas, cambio de tamaño de *buffers* e inserción de retardos.

La síntesis de la red de reloj se realizó con esfuerzo máximo para el diseño, de manera de respetar las restricciones definidas. En la figura 4.43 se muestra el árbol de reloj sintetizado para el reloj del controlador. Se necesitaron cuatro niveles de ramificación.

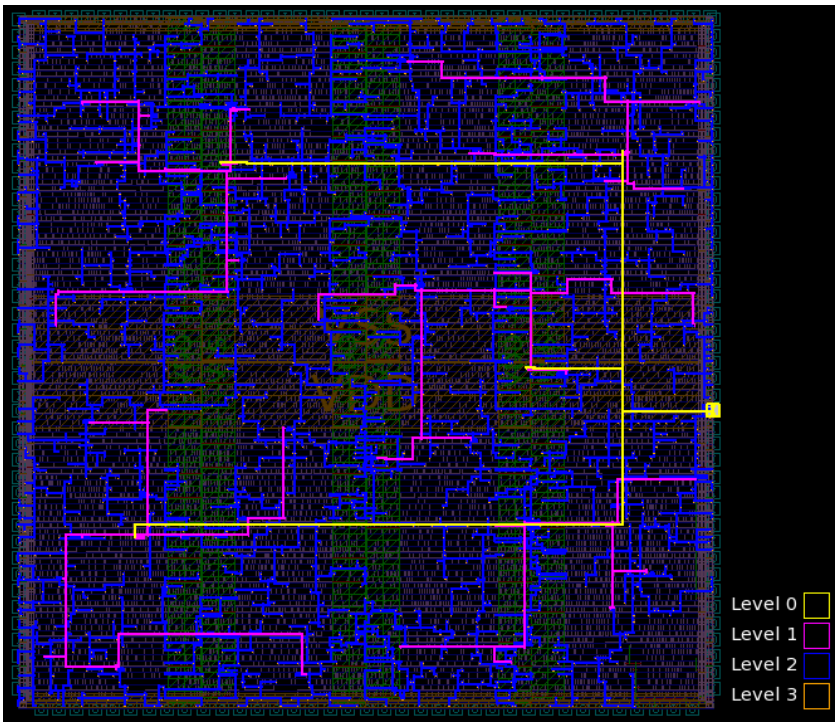


Figura 4.43: Árbol de Reloj Sintetizado

4.5.5. Ruteo

IC Compiler® realiza el ruteo de las conexiones sobre pistas de metal disponibles para ello. Si hay más conexiones que pistas disponibles en un área particular del diseño, se dice que esa área está congestionada [25]. En este caso, la herramienta tiene que rutear alrededor del área congestionada o se debe mejorar la etapa de ubicación de celdas. El ruteo del diseño es llevado a cabo en 3 etapas: ruteo global, asignación de pistas y ruteo detallado.

La etapa de ruteo debe cumplir con los siguientes pre-requisitos:

- Las redes de alimentación y tierra deben estar ruteadas después de la planificación del diseño y antes de la ubicación de celdas.
- La síntesis del *clock tree* y su optimización ya deben haber sido realizadas.
- La congestión estimada debe estar en un nivel aceptable.
- El *slack* antes del ruteo debe ser aceptable.
- La máxima capacitancia y tiempo de transición estimadas deben cumplir con las restricciones.

4.5.5.1. Ruteo Global

El ruteo global no se basa en las restricciones de tiempo. Primero busca las redes que aún no han sido ruteadas, y luego realiza los siguientes procesos:

- Divide el chip en unidades cuadradas llamadas celdas globales de ruteo (o GRCs por su siglas en inglés).
- Se asigan a cada GRC las redes que pasan por ellas. La capacidad de ruteo de una GRC se calcula en función de las áreas de bloqueo, pines y pistas de ruteo dentro de ellas.
- Determina la demanda de pistas de ruteo (horizontales y verticales) para cada GRC en función de la ubicación de los pines de las celdas su respectiva conexión (no ruteada aún).
- Reporta el sobre-flujo para cada capa de metal.

4.5.5.2. Asignación de Pistas

El proceso de asignación de pistas tampoco considera las restricciones de tiempo. La función de este paso es la asignación de pistas de ruteo a cada red, lo que se realiza sobre todo el diseño de una sola vez. Con esto se tienen todas las redes ruteadas gruesamente.

4.5.5.3. Ruteo Detallado

El ruteo detallado utiliza la información sugerida por el ruteo global y asignación de pistas para rutear las redes. Además corrige las violaciones de reglas de diseño (físicas) introducidas por el paso anterior. La corrección de estas violaciones es realizada en varias iteraciones y a distintas escalas.

4.5.5.4. Diseño Ruteado

En la figura 4.44 se muestra el diseño ya ruteado.

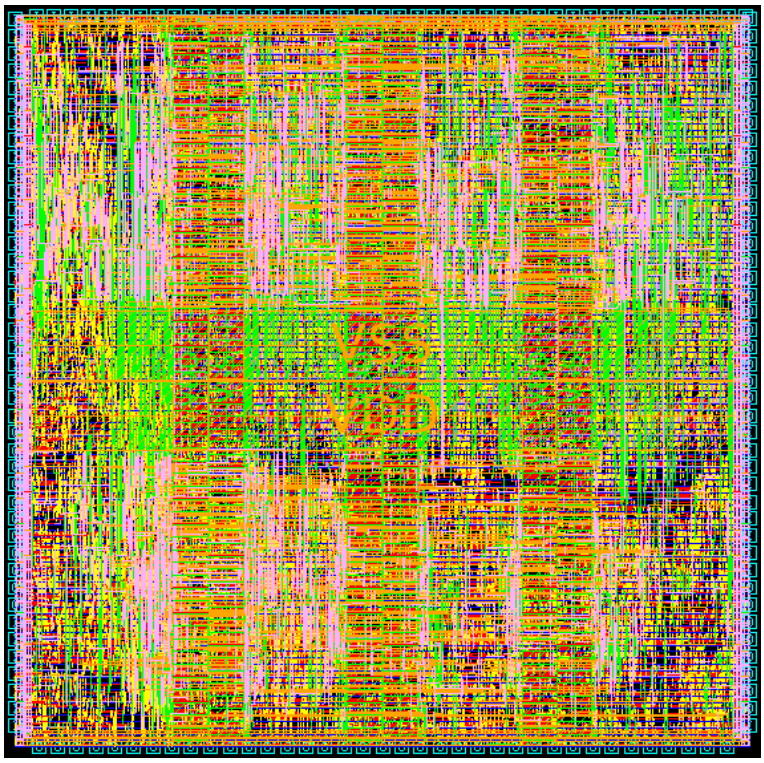


Figura 4.44: Controlador Ruteado

El peor *slack* reportado para el controlador es de 2.31 [ns]. El reporte de área final se detalla en la tabla 4.19.

N° de Puertos	163
N° de Redes	2774
N° de Celdas	2519
N° de Referencias	81
Área Combinacional	10232,75[μm^2]
Área no Combinacional	11110,25[μm^2]
Área de Interconexiones	5721,61[μm^2]
Total Área de Celdas	21343,00[μm^2]
Área Total	27064,61[μm^2]

Tabla 4.19: Área del Diseño después de Síntesis Física

4.5.6. Terminaciones del Chip

Una etapa adicional a las ya explicadas es definir las terminaciones del chip para dejar el diseño listo para su fabricación. Los procesos a ejecutar son llevados acabo a lo largo del flujo síntesis física, con objetivo de cuidar de temas particulares que aparecen durante la fabricación [31].

Las tareas de terminación de chip y preparación para la fabricación se muestran en el flujo de procesos de síntesis física modificada de la figura 4.45. El detalle de cada una de estas tareas se puede encontrar en [31].

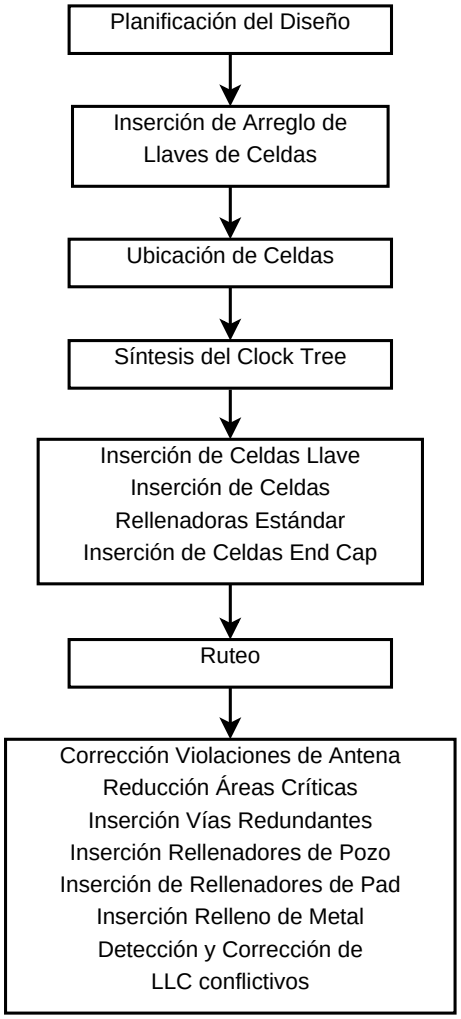


Figura 4.45: Terminaciones del Chip y Preparación para Fabricación

Capítulo 5

Discusión y Conclusiones

5.1. Conclusiones y Comentarios

El trabajo realizado finalmente consistió en aplicar al desarrollo de un controlador USB 2.0 el flujo de procesos de diseño de circuitos integrados:

1. Se modeló un controlador en base a la especificación de USB 2.0.
2. El modelo se describió utilizando Verilog a nivel RTL.
3. Se sintetizó una lista de compuertas interconectadas en base a la descripción de hardware.
4. Y finalmente se logró el plano físico del circuito.

Una de las principales conclusiones en base al trabajo realizado es que cada etapa puede y requiere ser más profundizada. Es posible que para un diseño profesional y comercial se necesite un equipo de personas especializadas para cada una de ellas. En base a ésto se listan las siguientes conclusiones:

- Una descripción a nivel comportamental, la cual no es siempre necesaria de realizar, puede ayudar de manera importante como guía a la descripción RTL del circuito. Se requiere un conocimiento acabado de la funcionalidad del diseño objetivo y de las técnicas de descripción comportamental.
- Un buen conocimiento de la herramienta compiladora de HDL y experiencia en la descripción RTL en circuitos permite obtener mejores resultados en la etapa de síntesis lógica. Tener una noción más cercana de lo que se obtiene en hardware al sintetizar un código HDL contribuye a un mayor control del diseño.

- Como en toda optimización, las restricciones tienen un rol protagónico y definen el sistema objetivo. Es importante tener un manejo de todas las características restringibles en un diseño, de manera de minimizar las diferencias entre diseño y circuito fabricado. Por ejemplo, cargas capacitivas, tiempos de transición y *skew* del reloj, entre otros.
- Es fundamental tener conocimiento del proceso de fabricación de circuitos integrados, de manera de manejar los conceptos que deben ser abordados en la síntesis física. La etapa de planificación del diseño es crítica para el resto del proceso, por lo tanto requiere atención especial.

El proceso de diseño, en el mejor de los casos, es puramente secuencial. Es cierto que cada etapa requiere de los resultados de la anterior, pero lo más probable es que en cada una se deba iterar más de una vez, y a veces regresar a etapas anteriores para volver a formular el diseño.

Respecto al protocolo USB 2.0, éste tiene suficientes aristas como para ser afrontado por un equipo de diseñadores. En particular se propone, para los lectores interesados, que el trabajo sea dividido en diseñar y re-diseñar las siguientes divisiones funcionales: interfaz con algún sistema objetivo; controlador de transferencias según su tipo; protocolo de transmisión y recepción; y, circuito analógico. Ésta conclusión se basa en el hecho de que el objetivo principal se hubiese llevado a cabo de mejor manera si el diseño se hubiese acotado a una de esas divisiones (sin considerar el circuito analógico).

A pesar de que el diseño no fue profundizado a nivel profesional, éste deja una guía clara de cómo proceder en el desarrollo de un circuito integrado digital. Los elementos utilizados fueron básicamente: un computador con acceso a la red de la Facultad de Ciencias Físicas y Matemáticas de la Universidad de Chile, de manera de poder conectarse al servidor que posee las herramientas proporcionadas por Synopsys®; documentación, principalmente de USB y del *software* de diseño; metodologías y herramientas nuevas de diseño para el marco de objetivos e intereses actuales del Departamento de Ingeniería Eléctrica. En conclusión, el diseño de circuitos integrados puede ser realizado en Chile, debido a que los elementos para su desarrollo existen, y su implementación en términos de fabricación puede ser realizada en otro país que posea la tecnología adecuada.

5.2. Trabajo Futuro

El proceso de diseño de circuito integrados posee etapas bien definidas, pero cada una de éstas tiene un gran nivel de profundidad. El diseño realizado puede ser aún más detallado y mejorado, por lo cual el trabajo que se puede realizar sobre él a futuro es considerable y se lista a continuación:

- Mejorar la descripción RTL del diseño, pensando siempre en el hardware que se sintetiza. Por ejemplo, se sugiere modificar el módulo *execute*, ya que es muy ineficiente en términos de la cantidad de compuertas que utiliza. También se puede tratar de dismi-

nuir la cantidad de máquinas de estado, por ejemplo unificando las que controlan las transacciones.

- Utilizar una memoria RAM en vez de bancos de registros, de manera de reducir la lógica sintetizada para cada registro y organizar de mejor manera el diseño.
- Considerar la utilización de solo una memoria RAM para intrucciones y datos, de manera de disminuir la cantidad de señales en la interfaz. En este sentido, se recomienda también el uso de buses bi-direccionales.
- Modificar módulos que dependan críticamente del reloj, como los contadores, de manera de poder operar a una frecuencia menor dentro del controlador. Este implica el diseño de módulos que permitan trabajar con los dos dominios de reloj: el del controlador y el de USB (480 [MHz]).
- Estudiar el uso de Design Compiler® en modo topográfico. Este modo permite realizar una estimación mucho más realista de los parámetros RC de las interconexiones, no se utilizan los *wire models*.
- Especificar más restricciones en la etapa de síntesis lógica, de manera de tener un modelo más realista. En particular, prestar atención a los caminos de tiempo mínimos, para evitar violaciones de tiempo de *hold*[1] de los flip-flops.
- Utilizar la herramienta Formality® para verificar que la funcionalidad se mantenga entre las etapas del diseño.
- Instanciar los pads para cada entrada y salida del módulo maestro, especificar restricciones para ubicación de pads e insertar rellenos de pads.
- Realizar análisis de caídas de voltaje en la malla de alimentación y tierra sintetizada.
- Profundizar en las tareas de terminaciones del chip, y en caso de ser necesario, aplicarlas.

[1]Es el tiempo que se debe mantener estable la señal de datos en un flip-flop después del flanco de captura del reloj.

Referencias

[1] IEEE Global History Network. *"Why Integrate a Circuit?"*. [En línea] <http://www.ieeeghn.org/wiki/index.php/Why_Integrate_a_Circuit%3F> [Consulta: 12 Marzo 2010]

[2] Intel. *"Excerpts from A Conversation with Gordon Moore: Moore's Law"*. 2005. [En línea] <ftp://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf> [Consulta: 12 Marzo 2010]

[3] Bhatnagar, H. *"Advanced Asic Chip Synthesis Using Synopsys® Design Compiler™ Physical Compiler™ and PrimeTime®"*. Kluwer Academic Publishers, 2002.

[4] Palnitkar, S. *"Verilog HDL, A Guide to Digital Design and Synthesis, IEEE 1364-2001 Compliant, Second Edition"*. Prentice Hall, 2008.

[5] Corfo. *"Por qué innovar"*. [En línea] <http://www.corfo.cl/acerca_de_corfo/innova_chile/por_que_innovar> [Consulta: 12 Marzo 2010]

[6] Corfo. *"Tipos de Innovación"*. [En línea] <http://www.corfo.cl/acerca_de_corfo/innova_chile/tipos_de_innovacion> [Consulta: 12 Marzo 2010]

[7] Consejo Nacional de Innovación para la Competitividad. *"Indicadores y Benchmarking"*. Marzo 26, 2008. [En línea] <http://bligoo.com/media/users/3/181209/files/18144/enic2008_anexo2.pdf> [Consulta: 12 Marzo 2010]

[8] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips. *"Universal Serial Bus Specification"*, Revision 2.0. Abril 27, 2000.

[9] Wikipedia contributors. *"Hardware description language"*, Wikipedia, The Free Encyclopedia. [En línea] <http://en.wikipedia.org/wiki/Hardware_description_language> [Consulta: 10 Octubre 2009]

[10] Sandolval, M. *"Diseño, con Fines Metodológicos, de un Transductor Digital utilizando Lenguaje de Descripción de Circuitos y Herramientas de Sínte-*

sis". Memoria (Ingeniería Civil Electricista). Santiago, Chile. Universidad de Chile, Facultad de Ciencias Físicas y Matemáticas, Octubre 6, 2006.

[11] Leblebici, Y. *"Introduction to VLSI Design: System Architecture and Timing Issues"*, Microelectronic Systems Laboratory, Diciembre, 1999.

[12] Wikipedia contributors. *"Fan-out"*, Wikipedia, The Free Encyclopedia. [En línea] <http://en.wikipedia.org/wiki/Fan-out> [Consulta: 07 Octubre 2009]

[13] Wikipedia contributors. *"Place and route"*, Wikipedia, The Free Encyclopedia. [En línea] <http://en.wikipedia.org/wiki/Place_and_route> [Consulta: 10 Octubre 2009]

[14] Wikipedia contributors. *"Floorplan (microelectronics)"*, Wikipedia, The Free Encyclopedia. [En línea] <http://en.wikipedia.org/wiki/Floorplan_(microelectronics)> [Consulta: 10 Octubre 2009]

[15] Contribuidores Wikipedia. *"Token Bus"*, Wikipedia, La Enciclopedia Libre. [En línea] <http://es.wikipedia.org/wiki/Token_Bus> [Consulta: 04 Febrero 2010]

[16] Contribuidores Wikipedia. *"Underrun"*, Wikipedia, La Enciclopedia Libre. [En línea] <http://es.wikipedia.org/wiki/Underrun> [Consulta: 15 Junio 2009]

[17] Compaq, Intel, Lucent, Microsoft, NEC. *"Errata for USB Revision 2.0 April 27, 2000"*, Diciembre 7, 2000.

[18] Wikipedia contributors. *"Non-return-to-zero"*, Wikipedia, The Free Encyclopedia. [En línea] <http://en.wikipedia.org/wiki/Non-return-to-zero> [Consulta: 05 Septiembre 2009]

[19] Wikipedia contributors. *"Data buffer"*, Wikipedia, The Free Encyclopedia. [En línea] <http://en.wikipedia.org/wiki/Data_buffer> [Consulta: 05 Febrero 2010]

[20] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips. *"Enhanced Host Controller Interface Specification for Universal Serial Bus"*, Revision 1.0. Marzo 12, 2002.

[21] USB Implementers Forums. *"Cyclic Redundancy Checks in USB"*, Draft.

[22] Intel. *"USB 2.0 Tranceiver Macrocell Interface (UTMI) Specification"*, Version 1.05. Marzo 29, 2001.

[23] Choate, Jim. *"USB 2.0 Hub Repeater"*, Presentación PDF. Junio 11, 2002.

[24] Cummings, Clifford. *"The Fundamentals of Efficient Synthesizable Finite State Machine Design using NC-Verilog and Buildgates"*, International Cadence Usergroup Conference. Septiembre 16-18, 2002.

[25] Synopsys® Customer Education Services. *"Design Compiler + IC Compiler"*, 2007.

[26] Synopsys®. *"Design Compiler User Guide"*, Junio, 2009.

[27] Synopsys®. *"HDL Compiler (Presto Verilog)"*, Manual de Referencia. Diciembre, 2003.

[28] Synopsys®. *"Timing Constraints and Optimization User Guide"*, Diciembre, 2009.

[29] San Diego State University, CS370 Course. *"Hazards/Glitches"*, Primavera, 2003. [En línea] <www-rohan.sdsu.edu/ subbaram/cs370/glitches.ppt> [Consulta: 20 Enero 2010]

[30] Wikipedia contributors. *"GDSII"*, Wikipedia, The Free Encyclopedia. [En línea] <http://en.wikipedia.org/wiki/GDSII> [Consulta: 07 Marzo 2010]

[31] Synopsys®. *"IC Compiler Implementation User Guide"*, Diciembre, 2009.

[32] Synopsys®. *"IC Compiler Design Planning User Guide"*, Diciembre, 2009.

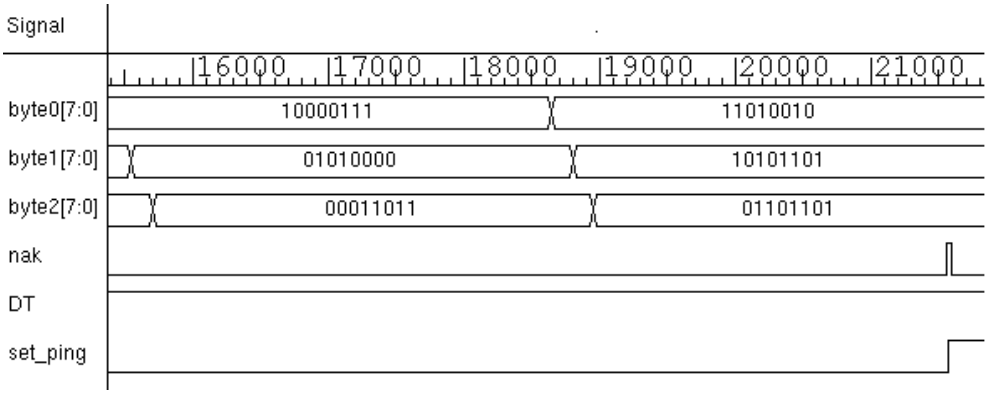
[33] Synopsys®. *"Physical Compiler User Guide User Guide"*, Volúmen 2. Diciembre, 2009.

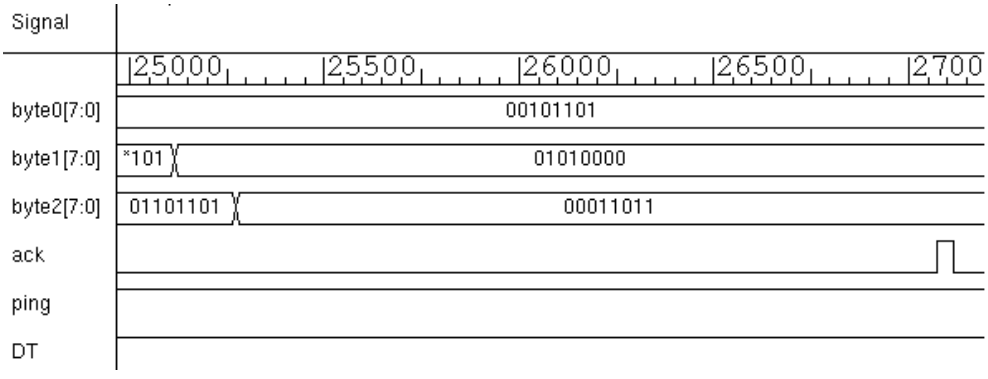
[34] Wikipedia contributors. *"Pad"*, Wikipedia, The Free Encyclopedia. [En línea] <http://en.wikipedia.org/wiki/Pad> [Consulta: 12 Marzo 2010]

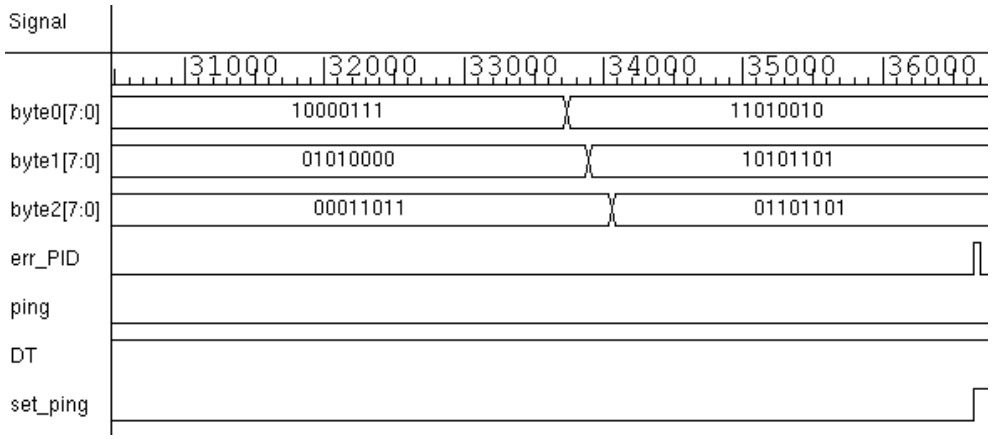
Anexo A

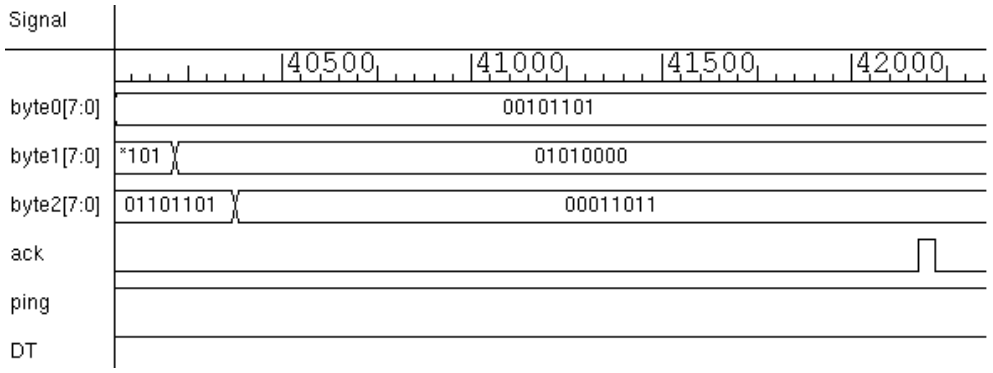
Etapa de Datos Simulación de Transacción de Control

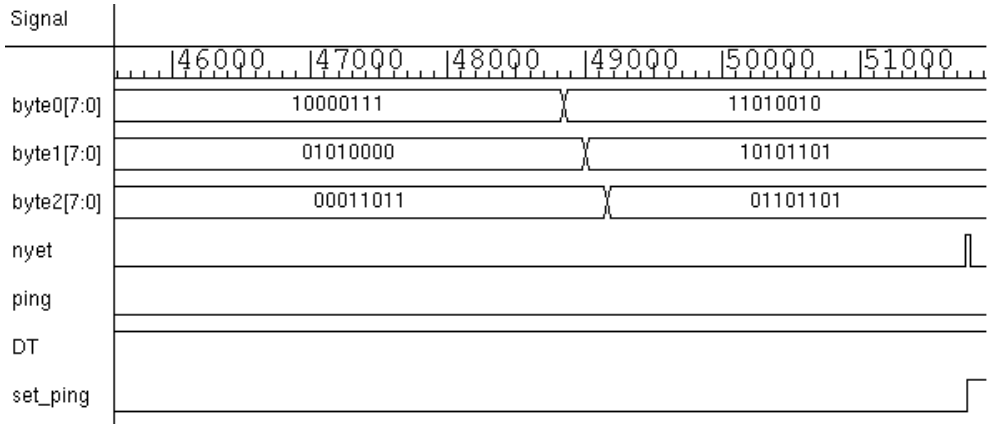
Las siguientes imágenes muestran la secuencia de transacciones pertenecientes a la simulación del descriptor explicado en la sección 4.3.2.2.

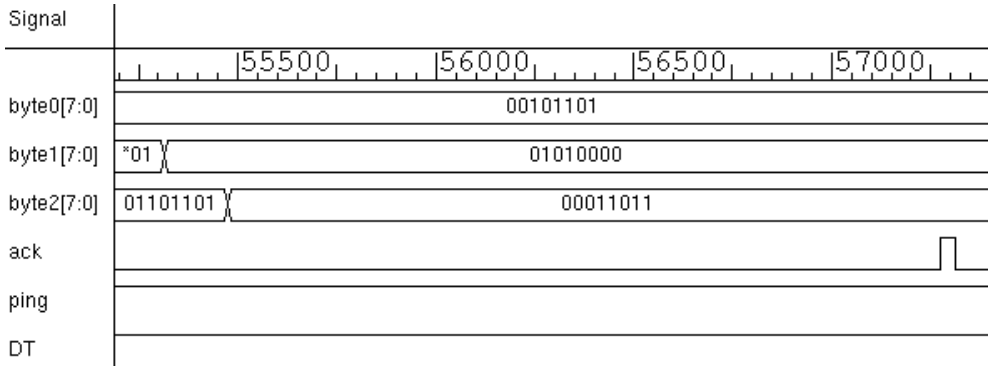


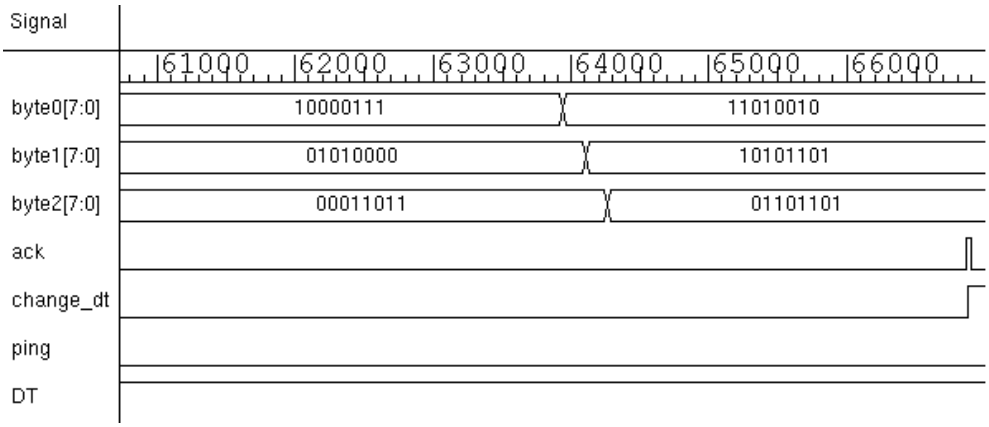


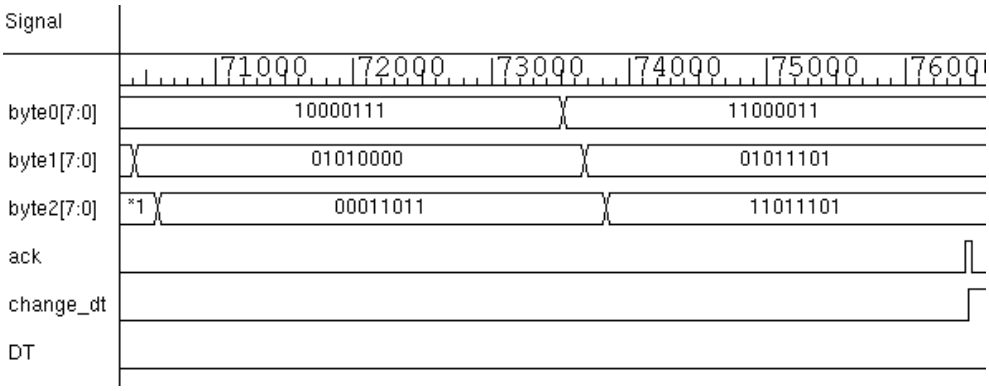

Signal
31000
32000
33000
34000
35000
36000
byte0[7:0]
10000111
11010010
byte1[7:0]
01010000
10101101
byte2[7:0]
00011011
01101101
err_PID
ping
DT
set_ping

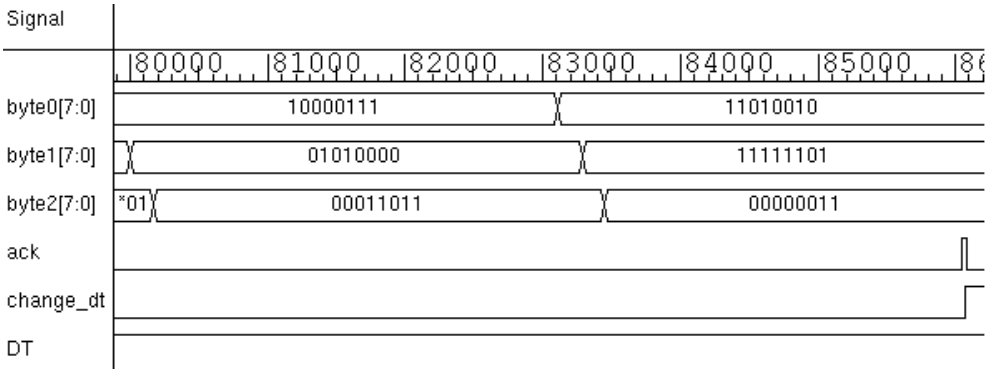

Signal
40500
41000
41500
42000
byte0[7:0]
00101101
byte1[7:0]
*101
01010000
byte2[7:0]
01101101
00011011
ack
ping
DT

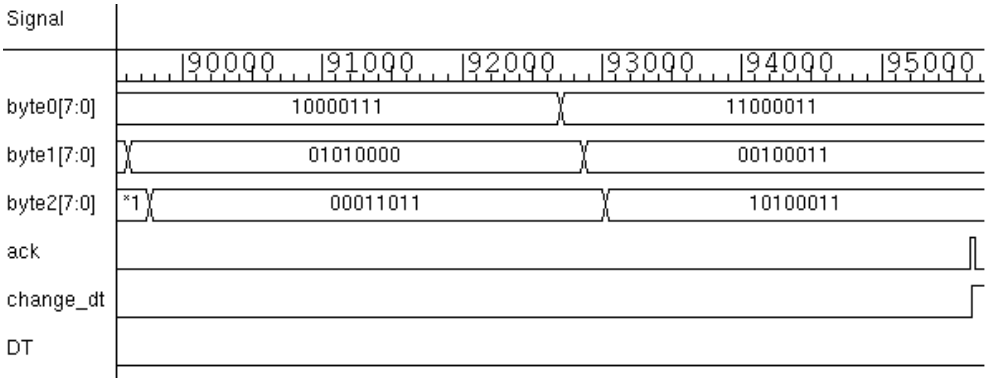

Signal
46000
47000
48000
49000
50000
51000
byte0[7:0]
10000111
11010010
byte1[7:0]
01010000
10101101
byte2[7:0]
00011011
01101101
nyet
ping
DT
set_ping

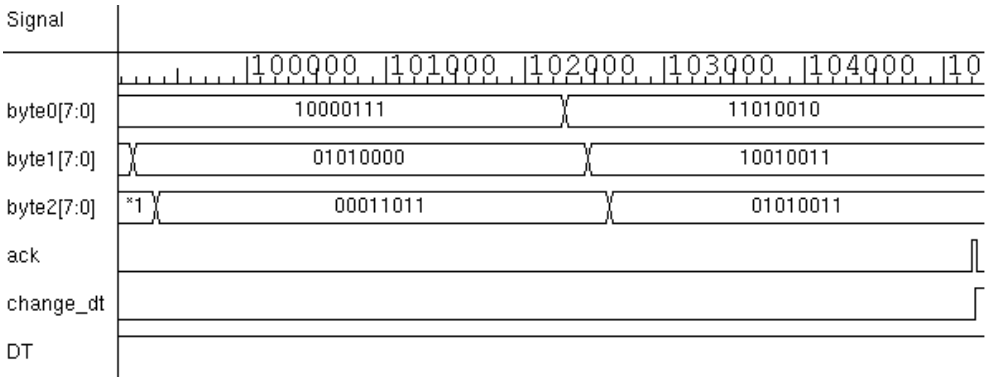

Signal
55500
56000
56500
57000
byte0[7:0]
00101101
byte1[7:0]
*01
01010000
byte2[7:0]
01101101
00011011
ack
ping
DT

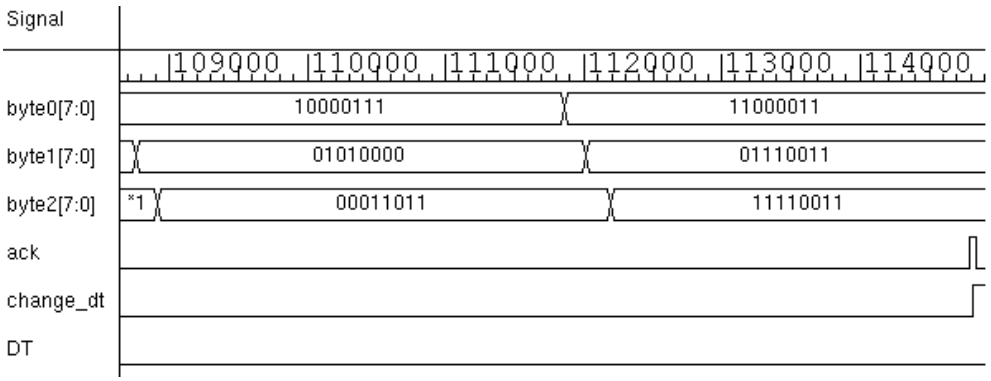

Signal
61000
62000
63000
64000
65000
66000
byte0[7:0]
10000111
11010010
byte1[7:0]
01010000
10101101
byte2[7:0]
00011011
01101101
ack
change_dt
ping
DT


Signal
71000
72000
73000
74000
75000
byte0[7:0]
10000111
11000011
byte1[7:0]
01010000
01011101
byte2[7:0]
*1
00011011
11011101
ack
change_dt
DT


Signal
80000
81000
82000
83000
84000
85000
byte0[7:0]
10000111
11010010
byte1[7:0]
01010000
11111101
byte2[7:0]
*01
00011011
00000011
ack
change_dt
DT


Signal
90000
91000
92000
93000
94000
95000
byte0[7:0]
10000111
11000011
byte1[7:0]
01010000
00100011
byte2[7:0]
*1
00011011
10100011
ack
change_dt
DT


Signal
100000
101000
102000
103000
104000
byte0[7:0]
10000111
11010010
byte1[7:0]
01010000
10010011
byte2[7:0]
*1
00011011
01010011
ack
change_dt
DT


Signal
109000
110000
111000
112000
113000
114000
byte0[7:0]
10000111
11000011
byte1[7:0]
01010000
01110011
byte2[7:0]
*1
00011011
11110011
ack
change_dt
DT

Anexo B

Material Incluído en CD

El CD que se adjunta al final de este documento está organizado de la siguiente manera:

- **Descripción RTL:** En esta carpeta se encuentran todos los módulos Verilog que describen el circuito diseñado. Además se encuentran los módulos utilizados para la validación a través de simulación.
- **Gate Net-List:** Esta carpeta posee la lista interconectada de compuertas, pertenecientes a la librería utilizada, que representa al circuito diseñado.
- **Scripts:** En esta carpeta se encuentran organizados los *scripts* que contienen las instrucciones de las herramientas utilizadas para cada etapa del diseño.

El detalle de los *scripts* se lista a continuación:

- **Compilación RTL:** El archivo contenido aquí permite la compilación conjunta de todos los módulos del diseño para luego poder visualizar la simulación con DVE®.
- **Síntesis Lógica:** Se adjunta un script para la lectura de las librerías (libs.tcl) y otro que ejecuta las instrucciones necesarias para la síntesis de la lista interconectada de compuertas (logic_synthesis.tcl). Además se encuentra el archivo con las restricciones del diseño (top.con).
- **Síntesis Física:** Los scripts contenidos en esta carpeta permiten la lectura de librerías(libs.tcl), creación de base de datos Milkyway (run.tcl) y planificación del diseño (floorplan.tcl). El resto de las etapas se sugiere realizar con la interfaz gráfica de ICC.