Controlador de Ascensor de 4 Plantas con Sistema de Emergencia en VHDL

Sistemas Electrónicos Digitales

(2025-2026)

A-408 GRUPO 16

- Julia Magallares Verde (56449)
- Pablo Muñoz Moreno (56507)
- Víctor Malumbres Muñoz (56451)

➢ Tutor: Rubén Núñez

ÍNDICE

1. Introducción

En esta memoria presentamos el desarrollo del proyecto final para la asignatura de Sistemas Electrónicos Digitales del curso 2025-2026. El trabajo consiste en el diseño, implementación y programación en VHDL de un controlador para un ascensor de cuatro plantas con modificaciones especiales.

Hemos elegido la propuesta "6. Ascensor" del listado de trabajos de la asignatura porque nos parecía uno de los retos más completos: hay que controlar el movimiento vertical del ascensor y gestionar las peticiones de los usuarios. Sin embargo, al empezar a diseñar, nos dimos cuenta de que podíamos ir un paso más allá. No queríamos limitarnos a lo básico, así que hemos ampliado el proyecto añadiendo funcionalidades que no se pedían explícitamente, como audio, movimiento horizontal y protocolos de seguridad.

Para la implementación del diseño hemos utilizado la placa de desarrollo Nexys 4 DDR (basada en la FPGA Artix-7), aprovechando la gran variedad de periféricos que ofrece para crear una interfaz de usuario rica y funcional. El principal reto ha sido diseñar un sistema de control para algo físico (motores, puertas) usando solo lógica digital, dividiendo el problema en bloques que operan de manera simultánea.

El sistema desarrollado contiene las siguientes características principales:

- **Control Lógico Central:** El núcleo del diseño es una Máquina de Estados Finitos (FSM) de Moore, encargada de tomar decisiones en función de las entradas de los usuarios y el estado de los sensores.
- **Gestión de Movimiento Avanzada:** Aparte de subir y bajar, hemos simulado que el ascensor puede moverse lateralmente a distintas "habitaciones" en cada planta, imitando un sistema de logística automático
- **Interfaz (HMI):** Usamos los switches y pulsadores de la placa, esto incluye la sincronización de entradas manuales (pulsadores y switches) para evitar rebotes y la visualización de datos en los displays de 7 segmentos y LEDs RGB.
- **Simulación de Planta Física:** Como no tenemos una maqueta física del ascensor, hemos programado una entidad que simula las leyes físicas. Este bloque recibe las señales de los motores y devuelve la posición actual, permitiéndonos verificar el funcionamiento del controlador en un entorno cerrado y controlado.
- **Audio y Seguridad:** Hemos incluido avisos sonoros (PWM) y un sistema que bloquea el ascensor si detecta sobrecarga o una emergencia.

Todo el proyecto se basa en un diseño síncrono, utilizando el reloj de 100 MHz de la placa como referencia única, y validando cada módulo mediante bancos de pruebas (testbenches) antes de su integración final en la entidad superior (`TOP_ASCENSOR`).

A continuación, detallaremos los objetivos específicos, la arquitectura interna del sistema, la descripción de los componentes VHDL desarrollados y los resultados obtenidos tras las pruebas de simulación e implementación física.

2. Objetivos y Especificaciones del Diseño

El objetivo de este proyecto es tener un sistema de control de ascensor totalmente funcional, validado y cargado en la FPGA. A continuación, explicamos qué requisitos nos venían impuestos y cuáles hemos añadido nosotros.

2.1. Objetivos Generales y Requisitos Técnicos

El diseño debe cumplir con una serie de normas técnicas que garantizan su viabilidad y buen funcionamiento en un entorno real:

- **Diseño Síncrono:** Todo el circuito se mueve al ritmo del reloj principal (100 MHz).Para ello, hemos implementado divisores de frecuencia que generan señales de habilitación (*enables*) para los distintos subsistemas.
- **Gestión de Entradas y Salidas:** Utilizamos pulsadores y switches como entradas de control manual, y LEDs y displays de 7 segmentos para la visualización de datos.
- **Robustez y Sincronización:** Es un requisito indispensable sincronizar todas las entradas asíncronas externas. Hemos diseñado etapas de sincronización para los pulsadores y switches con el fin de eliminar la metaestabilidad y filtrar los rebotes mecánicos antes de que las señales lleguen a la lógica de control.
- **Control mediante Máquina de Estados:** La lógica de control se basa en una Máquina de Estados Finitos (FSM). Hemos escogido una máquina de Moore, donde las salidas dependen únicamente del estado actual, lo que facilita la estabilidad de las señales de control.
- **Reset:** El sistema incorpora una señal de *RESET* global (conectada al pulsador `CPU_RESETN`) que reinicia todo el sistema a un estado seguro.

2.2. Especificaciones Funcionales del Ascensor

Las especificaciones funcionales que hemos implementado son las siguientes:

- **Control de Desplazamiento Vertical:** El sistema gestiona el movimiento de una cabina entre 4 pisos (del 0 al 3). El controlador recibe como entradas el piso actual (desde sensores simulados) y el piso objetivo (desde los botones de llamada).
- **Motores del Ascensor:** El circuito genera señales de salida para controlar el motor de movimiento (2 bits: parado, subir, bajar) y el motor de la puerta (1 bit: abrir/cerrar).
- **Algoritmo de Control:**
    - Al recibir una llamada, el ascensor se desplaza hacia el piso indicado.
    - Una vez en el destino, se activa la señal de apertura de puertas, manteniéndolas abiertas durante un tiempo antes de volver al estado de reposo o atender otra llamada.
    - **Prioridad de Tarea:** Se ha implementado una lógica de bloqueo tal que, si el ascensor ya se encuentra en movimiento realizando un servicio, ignora nuevas pulsaciones de botones hasta que finaliza la tarea actual, evitando cambios bruscos de dirección o estados indeterminados.

- **Visualización de Estado:** Se utilizan los displays de 7 segmentos para mostrar en todo momento el piso actual y el estado del sistema (ej. "S" para subiendo, "b" para bajando, "P" para puertas abiertas), cumpliendo con el requisito de visualización de información.

2.3. Ampliación de Funcionalidades y Mejoras

Hemos diseñado e integrado las siguientes mejoras para el ascensor que expanden las funcionalidades básicas del enunciado:

- **Simulador de Planta Física en FPGA:** Dado que no disponemos de una maqueta de ascensor real con sensores y motores físicos, hemos implementado una entidad (`Simulador_Planta`) que emula el comportamiento físico del ascensor. Este módulo calcula la posición teórica del ascensor en tiempo real, cerrando el lazo de control dentro de la propia FPGA.
- **Movimiento Horizontal (Sistema Multi-Eje):** Hemos añadido una dimensión extra al movimiento. Además de subir y bajar plantas, el sistema simula que el ascensor puede desplazarse horizontalmente hacia 4 "habitaciones" o posiciones distintas en cada piso. Esto añade complejidad a la FSM, que ahora debe coordinar dos motores (vertical y horizontal) para llevar la cabina a las coordenadas (Piso, Habitación) exactas seleccionadas por el usuario mediante switches.
- **Sistema de Audio:** Hemos implementado un controlador de audio PWM (`Controlador_Audio`) que utiliza la salida de audio de la placa para proporcionar retroalimentación al usuario. El sistema genera diferentes patrones sonoros:
  - Melodía dinámica durante el movimiento.
  - Tono de aviso al abrir puertas.
  - Señal de alarma en caso de emergencia.
  - Aviso grave en caso de sobrecarga.
- **Protocolos de Seguridad:**
  - **Detección de Sobrecarga:** Mediante un switch dedicado, simulamos un sensor de peso excesivo. Si se activa, el ascensor entra en estado de bloqueo, abre puertas, muestra un aviso visual ("L" de *Load*) y emite una alerta sonora, impidiendo el movimiento hasta que se retira la carga.
  - **Parada de Emergencia:** Se ha configurado un mecanismo que, mediante un switch determinado, fuerza al ascensor a un estado de seguridad, deteniendo motores y activando un parpadeo visual de alerta en todos los indicadores (LEDs y Displays).

3. Metodología de Diseño y Arquitectura

En nuestro proyecto hemos seguido una metodología de diseño jerárquica y modular. En lugar de intentar programar una única entidad que gestione todas las señales, hemos dividido el código en bloques funcionales independientes, facilitando la depuración de errores y la verificación individual de cada componente antes de su integración final.

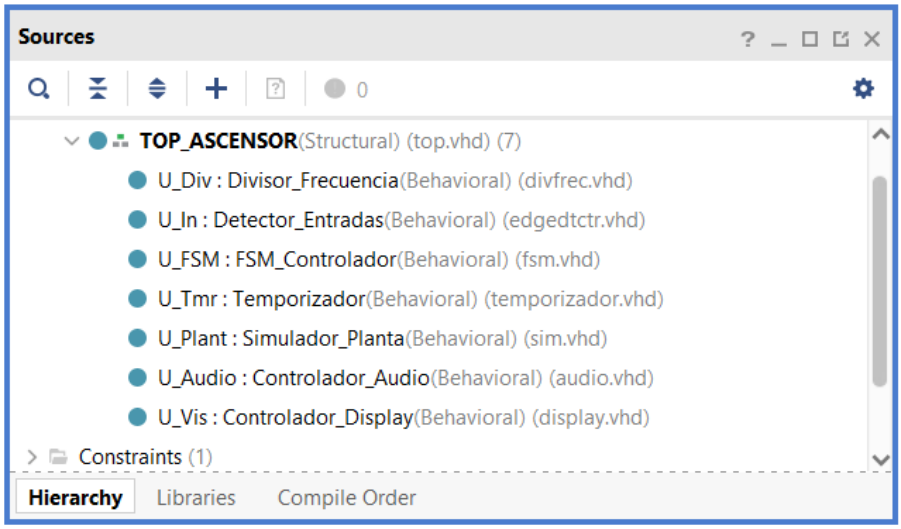

Figura 3.1: Jerarquía de módulos en Vivado. Se muestra la estructura modular del proyecto bajo la entidad TOP_ASCENSOR.

3.1. Estrategia de Diseño

La implementación se ha basado en los siguientes tres principios, siguiendo las recomendaciones de diseño digital:

- **Diseño Totalmente Síncrono:** Todos los procesos secuenciales del sistema son disparados por el flanco de subida del reloj maestro de 100 MHz (`CLK100MHZ`).
- **Separación entre Control y Datos:** Hemos aislado la lógica de toma de decisiones (la FSM) de los bloques de gestión de datos y periféricos (contadores, simulador físico, controladores de display).
- **Máquina de Estados de Moore:** Para el controlador central, hemos optado por una topología de Moore, donde las salidas dependen exclusivamente del estado actual.

3.2. Arquitectura RTL del Sistema

La arquitectura del proyecto se estructura en torno a una entidad superior denominada `TOP_ASCENSOR`. Esta entidad actúa como un contenedor estructural que instancia las diferentes entidades y define las señales internas que los conectan.

A continuación, se presenta el Diagrama de Bloques RTL (*Register Transfer Level*) que representa la estructura interna del diseño y el flujo de señales.

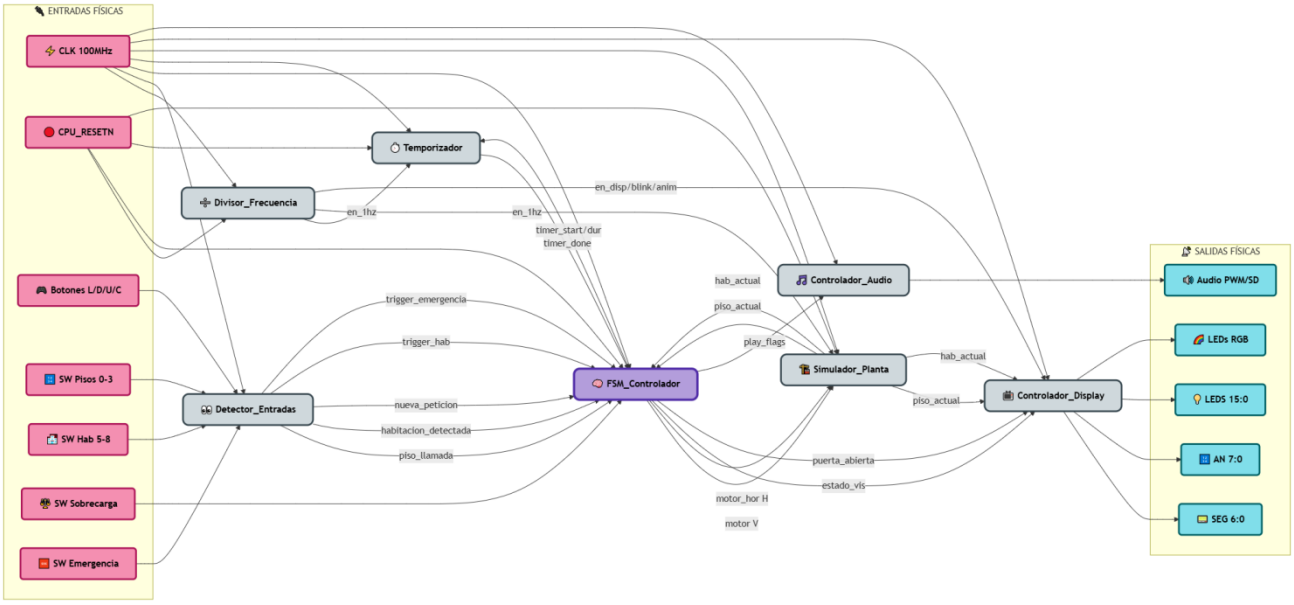

Figura 3.2: Diagrama de Bloques y Arquitectura del Sistema.

Como se observa en la Figura 3.2, la arquitectura presenta un flujo de información claro de izquierda a derecha:

- **Entradas Físicas (Magenta):** A la izquierda se sitúan las señales provenientes del mundo exterior (pulsadores, switches). Estas entran al bloque `Detector_Entradas`, que actúa como barrera de protección y sincronización.
- **Núcleo de Control (Centro/Lila):** El bloque `FSM_Controlador` se destaca visualmente como el "cerebro" del sistema. Recibe las peticiones procesadas y coordina al resto de módulos (Gris) mediante señales de control.
- **Realimentación (Feedback Loop):** Un aspecto crítico es la conexión bidireccional con el `Simulador_Planta`. La FSM envía órdenes de motor y recibe la posición actual (`piso_actual`, `hab_actual`), cerrando el lazo de control automático.
- **Salidas Físicas (Cian):** A la derecha, los módulos (`Controlador_Display`, `Controlador_Audio`) traducen el estado interno a señales físicas para los displays de 7 segmentos, LEDs y PWM.

3.3. Gestión de Reloj y Tiempos

Es importante destacar en la arquitectura la distribución de la señal de reloj. Como se aprecia en el diagrama, la señal `CLK 100MHz` alimenta a todos los bloques secuenciales al mismo tiempo.

La gestión temporal no se realiza dividiendo este reloj, sino mediante el módulo auxiliar `Divisor_Frecuencia` (visible junto al reloj), que genera pulsos de habilitación (`en_1hz`, `en_disp`) que se distribuyen al resto de bloques para marcar el ritmo de operaciones lentas, como el movimiento del motor o el refresco de pantalla.

4. Descripción Detallada de Bloques (Hardware Description)

El diseño del sistema se ha realizado siguiendo una arquitectura modular, donde cada archivo `.vhd` corresponde a una entidad con una función específica todas estas se conectan en el archivo principal `TOP_ASCENSOR`. A continuación, describimos el funcionamiento interno de cada uno de estos bloques.

4.1. Divisor de Frecuencia

La placa Nexys 4 DDR posee un reloj interno que oscila a una velocidad de 100 MHz (10^8 ciclos por segundo). Esta velocidad es perfecta para el procesamiento de datos, pero es excesivamente rápida para la percepción humana o para el movimiento mecánico de un ascensor.

Por ello, hemos diseñado el módulo `Divisor_Frecuencia`. Su función no es crear un nuevo reloj, sino generar señales de habilitación (*enables*) que actúan como "metrónomos" para marcar el ritmo de los demás componentes. Funciona mediante contadores internos que se reinician al llegar a un valor límite.

Hemos generado cuatro bases de tiempo distintas:

- **`en_1hz`:** Un pulso cada segundo. Lo usamos para contar cuánto tiempo está la puerta abierta o cuánto tarda en subir un piso.
- **`en_disp`:** Una señal rápida (~1 kHz) para el refresco de los displays de 7 segmentos.
- **`en_blink`:** Una señal de parpadeo visible (~2 Hz) para las luces de emergencia.
- **`en_anim`:** Una señal intermedia para suavizar las transiciones de los LEDs de las puertas.

4.2. Acondicionamiento de Entradas (Switches y Botones)

Uno de los primeros problemas que tuvimos fue el ruido mecánico, cuando pulsamos un botón mecánico, la señal eléctrica no cambia de '0' a '1' de forma limpia instantáneamente; suele haber pequeñas fluctuaciones o "rebotes". Si conectásemos el botón directamente a la lógica de control, el ascensor podría interpretar una sola pulsación como múltiples llamadas seguidas.

El bloque `Detector_Entradas` soluciona esto y actúa como "traductor" entre el usuario y la máquina. Realiza dos funciones vitales:

1. **Sincronización:** Pasa las señales externas a través de registros internos (`_sync`, `_prev`) sincronizados con el reloj, eliminando la metaestabilidad.
2. **Detección de Flancos:** No nos importa si el botón está pulsado, sino el momento exacto en el que se pulsa. El sistema detecta la transición de 0 a 1.

Ejemplo de funcionamiento:

Imaginemos que el usuario activa el interruptor de "Emergencia". El detector compara el valor actual con el valor del ciclo de reloj anterior. Si son diferentes (sw_res_sync /=

sw_res_prev), genera un pulso único llamado trigger_emergencia que dura exactamente un ciclo de reloj. Este pulso es la "orden limpia" que recibe el cerebro del sistema.

4.3. Máquina de Estados Finitos (FSM)

Este es el módulo más complejo del ascensor (`FSM_Controlador`). Hemos implementado una máquina de estados de tipo Moore, lo que significa que las salidas (motores, luces, audio) dependen únicamente del estado en el que nos encontremos, garantizando estabilidad.

La máquina gestiona una serie de estados lógicos: `IDLE` (reposo), `CERRANDO`, `SUBIENDO`, `BAJANDO`, `MOVIENDO_HOR` (habitaciones), `LLEGADA`, `ABRIENDO` y los estados de emergencia `EMERG_WAIT` y `SOBRECARGA`.

Ejemplo Práctico de la Lógica de Prioridades:

Para entender la complejidad de la FSM, planteamos la siguiente situación:

El ascensor está esperando en la Planta 0, Habitación 1. Un usuario quiere ir a la Planta 2, Habitación 3.

1. El usuario pulsa el botón del Piso 2 y activa el switch de Habitación 3.
2. La FSM recibe ambas peticiones, pero está programada para dar prioridad vertical.
3. Primero, entra en estado `SUBIENDO`. Aunque la habitación destino (3) es distinta a la actual (1), el ascensor ignora el movimiento horizontal mientras sube.
4. Al llegar al Piso 2, la FSM evalúa si es la planta correcta solo cuando esta es correcta pasa al siguiente estado (habitación).
5. Automáticamente, transita al estado `MOVIENDO_HOR`. Ahora activa el motor horizontal hasta llegar a la Habitación 3.
6. Solo cuando ambas coordenadas (Piso 2, Hab 3) coinciden, pasa al estado `ABRIENDO` puertas.

Este comportamiento secuencial asegura que el ascensor nunca intente realizar movimientos diagonales erráticos.

4.4. Temporizador Programable

Para evitar detener el procesador con bucles de espera ineficientes, hemos decidido incorporar el módulo Temporizador.

La FSM es la que da órdenes: le dice al temporizador que le avise cuando pase un tiempo determinado. La FSM se queda esperando en un estado (por ejemplo, con la puerta abierta) hasta que el temporizador devuelve la señal timer_done. Esto permite modificar los tiempos de espera del ascensor (tiempo de puerta, tiempo de emergencia) cambiando solo una línea de código, sin alterar la lógica de control.

4.5. Simulador de Planta Física

Este es uno de los aportes originales más importantes de nuestro grupo. Dado que no disponemos de un hueco de ascensor real en el laboratorio, hemos creado una "Planta Virtual" dentro de la FPGA (`Simulador_Planta`).

Este bloque se comporta físicamente según las leyes del movimiento que hemos programado:

- Tiene dos registros internos: `p_reg` (posición vertical) y `h_reg` (posición horizontal).
- Si recibe la señal `motor = "01"` (subir), incrementa el contador de pisos cada cierto tiempo. Si recibe `motor = "10"`, lo decrementa.
- Lo mismo ocurre con el motor horizontal y las habitaciones.

Gracias a este módulo, el controlador "cree" que está moviendo un ascensor real. Si desconectáramos este módulo y conectáramos los cables a una maqueta física con sensores reales, el controlador funcionaría exactamente igual sin cambiar ni una línea de código de la FSM.

4.6. Controlador de Periféricos de Salida (Display y LEDs)

Este bloque (`Controlador_Display`) se encarga de traducir el estado interno del sistema a información visual comprensible para el usuario, utilizando técnicas de multiplexación.

- **Multiplexación de Displays:** La placa tiene 8 displays de 7 segmentos, mostrando información distinta: los displays de la izquierda muestran la Habitación (ej: `H-01`) y los de la derecha el estado y piso (ej: `S-P2` para "Subiendo al Piso 2").
- **Código de Colores (Semáforo):** Utilizamos los LEDs RGB para dar un feedback instantáneo:
  - **Verde:** Reposo / Llegada.
  - **Rojo:** En movimiento (Peligro).
  - **Amarillo:** Sobrecarga.
  - **Azul Parpadeante:** Emergencia.
- **Audio (PWM):** Aunque es un módulo independiente (`Controlador_Audio`), forma parte de la salida. Genera ondas cuadradas de frecuencia variable mediante contadores. Por ejemplo, para hacer un tono agudo, el contador se reinicia muy rápido; para un tono grave, más lento. Combinando estas frecuencias, logramos efectos sonoros distintos para la alarma o la llegada a planta.

5. Diagrama de Estados (FSM)

El comportamiento secuencial del ascensor está gobernado por una Máquina de Estados Finitos (FSM) modelada bajo la arquitectura de Moore. Esto implica que las salidas de control (motores, leds, audio) se definen estáticamente dentro de cada estado, lo que simplifica la depuración y evita condiciones de carrera en las señales de salida.

A continuación, describimos la lógica de transición implementada, la cual se divide en tres ramas operativas: funcionamiento normal, gestión de sobrecarga y protocolo de emergencia.

5.1. Definición de Estados

La máquina cuenta con 10 estados definidos en el tipo enumerado `estados_t` dentro del código VHDL:

1. **IDLE_OPEN (Reposo):** Estado inicial (Verde). El ascensor está quieto en una planta, con la puerta abierta y esperando peticiones.
2. **CERRANDO:** Estado de transición (Amarillo). Se apaga la señal de puerta abierta y se activa un temporizador de seguridad antes de arrancar los motores.
3. **SUBIENDO / BAJANDO:** Estados de movimiento vertical (Azul). Se activa el motor principal y el audio de movimiento.
4. **MOVIENDO_HOR:** Estado exclusivo de esta implementación (Morado). Se activa si el piso es correcto pero la habitación no. Controla el segundo motor para el desplazamiento lateral.
5. **LLEGADA:** Breve pausa técnica (Amarillo) al alcanzar el destino antes de iniciar la maniobra de apertura.
6. **ABRIENDO:** Se activa la señal de puerta abierta y el sonido de notificación de llegada.
7. **SOBRECARGA:** Estado de bloqueo (Rojo). Se activa por el switch de peso, mostrando un aviso de error.
8. **EMERG_WAIT y EMERG_BLINK:** Secuencia de emergencia (Rojo). Primero muestra una señal estática ("E") y luego pasa a un parpadeo de alerta total.

5.2. Representación Gráfica del Sistema

La siguiente figura muestra el diagrama de estados completo generado a partir de la lógica del controlador:

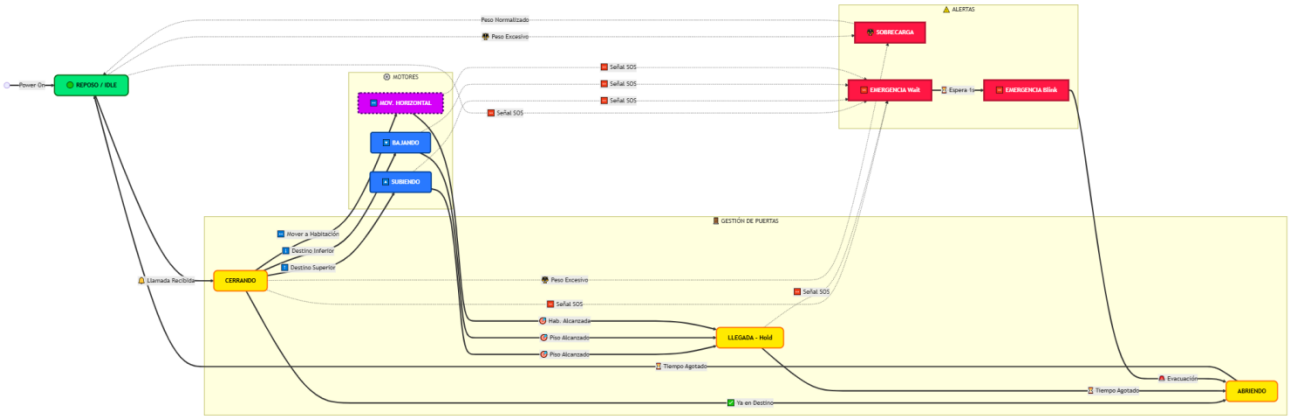

Figura 5.1: Diagrama de Estados del Controlador del Ascensor.

Como se aprecia en el diagrama, hemos utilizado una codificación de colores para agrupar visualmente los subsistemas lógicos:

- 🟢 **Bloque de Reposo (Verde):** Punto de inicio y espera del sistema.
- 🔵 **Bloque de Motores (Azul/Morado):** Agrupa los estados de movimiento vertical y horizontal (`SUBIENDO`, `BAJANDO`, `MOV. HORIZONTAL`).
- 🟡 **Bloque de Puertas (Amarillo):** Gestiona la seguridad de entrada y salida (`CERRANDO`, `LLEGADA`, `ABRIENDO`).
- 🔴 **Bloque de Seguridad (Rojo):** Estados críticos (`SOBRECARGA`, `EMERGENCIA`) que tienen prioridad sobre el funcionamiento normal.

5.3. Lógica de Transiciones (Flujo del Diagrama)

El flujo de estados sigue una jerarquía de prioridades estricta para garantizar la seguridad: Emergencia > Sobrecarga > Movimiento Normal.

A. Ciclo de Movimiento Normal (Bucle Principal):

1. **Inicio:** Desde `IDLE`, si hay una "Llamada Recibida", el sistema pasa a `CERRANDO`.
2. **Decisión:** Al terminar de cerrar, la FSM evalúa el destino:
   - Si el destino es superior/inferior → Transita a `SUBIENDO` o `BAJANDO`.
   - Si el piso es correcto pero la habitación distinta → Transita a `MOV. HORIZONTAL`.
   - Si ya está en el sitio → Salta directamente a `ABRIENDO`.
3. **Llegada:** Todos los estados de movimiento convergen en `LLEGADA` cuando los sensores detectan el objetivo ("Piso Alcanzado" o "Hab. Alcanzada").
4. **Retorno:** Tras los temporizadores de `ABRIENDO`, el sistema regresa a `REPOSO`, completando el ciclo.

B. Gestión de Sobrecarga: Si se activa el interruptor de peso (`sw_sobrecarga`) desde el reposo o cerrando, el sistema transita incondicionalmente a `SOBRECARGA`, impidiendo el movimiento hasta que se normalice el peso.

C. Protocolo de Emergencia: Cualquier señal de `SOS TRIGGER` provoca una transición inmediata al bloque de emergencia (`EMERG_WAIT`), sin importar el estado actual. La secuencia finaliza forzando una evacuación (`EMERG_BLINK` → `ABRIENDO`) para garantizar que los pasajeros nunca queden atrapados.

6. Implementación Física y Restricciones (.xdc)

La síntesis del diseño y su implementación en la FPGA no dependen únicamente del código VHDL. Para que el "bitstream" final funcione correctamente en la placa Nexys 4 DDR, es necesario definir un archivo de restricciones físicas (XDC - *Xilinx Design Constraints*).

Este archivo cumple dos funciones vitales en nuestro proyecto: mapear los puertos lógicos de la entidad `TOP_ASCENSOR` a los pines físicos del chip Artix-7 y definir las restricciones temporales para garantizar que el circuito cumpla con los requisitos de velocidad.

6.1. Mapeo de E/S (Pinout)

Dada la gran cantidad de entradas y salidas que gestiona nuestro sistema (más de 40 señales entre pulsadores, switches, LEDs y displays), hemos diseñado una distribución más sencilla para facilitar su uso durante la demostración.

Las asignaciones en el fichero `.xdc` se han realizado siguiendo esta lógica:

- **Reloj y Reset:**
  - `CLK100MHZ`: Asignado al pin **E3**, correspondiente al oscilador de cristal de 100 MHz de la placa.
  - `CPU_RESETN`: Asignado al botón rojo dedicado (pin **C12**). Al ser lógica negativa (activo a nivel bajo), en el VHDL lo invertimos (`rst <= not CPU_RESETN`) para trabajar con lógica positiva internamente.
- **Interfaz de Movimiento Vertical (Pisos):** Hemos agrupado los controles de piso en la parte derecha de la placa para operarlos con una mano.
  - **Llamadas (Botones):** Los pulsadores en cruz (`BTNU`, `BTND`, `BTNL`, `BTNC`) se utilizan para llamar a los pisos, aprovechando su disposición geográfica intuitiva.
  - **Selector de Piso Actual (Switches):** Los interruptores `SW(0)` a `SW(3)` permiten al usuario (o al profesor) forzar manualmente la detección del piso actual si se deshabilita el simulador.
- **Interfaz de Movimiento Horizontal (Habitaciones):** Para diferenciar claramente esta funcionalidad extra, hemos separado sus controles en el banco de interruptores central.

    - `SW(5)` a `SW(8)`: Actúan como selectores de la habitación destino (1 a 4). Esta separación física evita que el usuario confunda seleccionar un piso con seleccionar una habitación.
- **Sistemas de Seguridad:** Para evitar activaciones accidentales, hemos situado las funciones críticas en interruptores aislados o extremos:
    - **Sobrecarga:** Asignada al `SW(4)`, separando los controles de piso de los de habitación.
    - **Emergencia Total:** Asignada al `SW(15)` (el último a la izquierda). Al ser un interruptor de palanca, permite dejar el ascensor en estado de emergencia permanente hasta que se baje manualmente.
- **Salidas Visuales y Auditivas:**
    - **Displays:** Se han mapeado los cátodos (`SEG`) y ánodos (`AN`) para utilizar los 8 dígitos. Los 4 de la izquierda muestran la habitación (`H-XX`) y los 4 de la derecha el estado del piso (`P-XX`).
    - **LEDs RGB:** Los dos LEDs tricolores (`LED16` y `LED17`) se usan como semáforos de estado (Verde=Libre, Rojo=Ocupado, Azul=Emergencia).
    - **Audio:** El puerto `AUD_PWM` se conecta a la salida de audio mono y, muy importante, hemos asignado el puerto `AUD_SD` (*ShutDown*) a '1' lógico para habilitar el amplificador de la placa, que por defecto está apagado.

6.2. Restricciones Temporales (Timing Constraints)

Además de la ubicación de los pines, el fichero XDC incluye la definición del reloj principal. Esta línea es crucial para que la herramienta de síntesis (Vivado) sepa que el sistema debe funcionar a 100 MHz (periodo de 10 ns).

```
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK100MHZ}];
```

Gracias a esta restricción, el sintetizador analiza todos los caminos lógicos entre registros (Flip-Flops). Durante la implementación, Vivado verifica que las señales lleguen a tiempo (Setup Time) antes del siguiente flanco de reloj. En nuestro reporte de *Timing Summary*, confirmamos que el WNS (*Worst Negative Slack*) es positivo, lo que garantiza que el diseño es robusto y no sufrirá fallos de sincronización ni metaestabilidad interna.

6.3. Configuración Eléctrica

Todos los pines de E/S se han configurado bajo el estándar LVCMOS33 (Low Voltage CMOS 3.3V), que es el nivel de tensión nativo de los periféricos de la Nexys 4 DDR, asegurando la compatibilidad eléctrica y protegiendo la FPGA.

7. Simulación y Verificación (Testbench)

Siguiendo las recomendaciones de la metodología de diseño, antes de sintetizar el diseño en la FPGA, hemos procedido a la verificación funcional de cada bloque mediante simulación comportamental. Esta etapa es imprescindible, ya que la

depuración sobre el hardware físico es limitada (solo vemos LEDs y displays), mientras que la simulación nos permite inspeccionar señales internas, variables de estado y cronogramas exactos.

Para ello, hemos utilizado el simulador integrado en Vivado, creando ficheros de prueba (*Testbenches*) específicos para cada entidad.

Cabe destacar la modificación de la frecuencia de las variables en la entidad divfrec para que las simulaciones se realicen en un tiempo razonable, no en los elegidos para el funcionamiento en la placa.

7.1. Estrategia de Verificación (Bottom-Up)

Nuestra estrategia se ha dividido en dos fases:

1. **Pruebas Unitarias:** Verificación de cada entidad por separado para asegurar que cumplen su función específica.
2. **Prueba de Integración:** Validación del sistema completo (`TOP_ASCENSOR`) para comprobar la interconexión entre bloques.

7.2. Pruebas Unitarias

Hemos desarrollado bancos de pruebas para los siguientes componentes clave:

- Testbench de la FSM (FSM_Controlador_TB):

  Al ser el núcleo lógico, este fue el test más exhaustivo. Simulamos estímulos de entrada manuales (sin depender de botones físicos) para forzar transiciones.

  - *Caso de Prueba 1:* Ciclo normal de llamada. Verificamos que la máquina pasa correctamente por `IDLE` ⇒ `CERRANDO` ⇒ `SUBIENDO` ⇒ `LLEGADA` ⇒ `ABRIENDO`.
  - *Caso de Prueba 2:* Prioridad de movimiento. Comprobamos que si se activa `sw_sobrecarga` mientras el ascensor está `SUBIENDO`, la máquina completa el viaje hasta el piso más cercano y se bloquea allí, en lugar de detenerse en seco entre plantas.
- Testbench del Simulador de Planta (Simulador_Planta_TB):

  Validamos que la física del ascensor virtual es correcta.

  - Inyectamos señales de motor (`"01"` y `"10"`) durante tiempos prolongados y verificamos mediante la forma de onda que los registros de posición (`piso_actual`, `hab_actual`) se incrementan y decrementan secuencialmente, respetando los límites (no bajar del piso 0 ni subir del 3).
- Testbench de Visualización (Controlador_Display_TB):

  Comprobamos el correcto funcionamiento de la multiplexación en el tiempo. En la simulación, observamos cómo la señal de ánodos (AN) rota cíclicamente

activando un display a la vez, y verificamos que los segmentos (SEG) decodifican correctamente los caracteres especiales diseñados (letras 'P', 'H', 'E', 'L', etc.).

7.3. Simulación del Sistema Completo (`TOP_ASCENSOR_TB`)

Una vez validados los componentes, ejecutamos el testbench general `TOP_ASCENSOR_TB`. Este archivo emula el comportamiento de un usuario real interactuando con la placa completa.

1. **Inicialización:** Se aplica un pulso de `CPU_RESETN` y se comprueba que el sistema arranca en Piso 0, Habitación 1, con la puerta abierta (Estado `IDLE`).
2. **Viaje Vertical:** Simulamos la pulsación del botón `BTND` (Piso 2). Observamos en el cronograma cómo se activan los motores verticales, cambia el estado a `SUBIENDO` y, tras un tiempo, el piso actual llega a 2.
3. **Desplazamiento Horizontal:** Estando en el Piso 2, activamos el Switch de Habitación 3. Verificamos que el sistema entra en estado `MOVIENDO_HOR` y ajusta su posición lateral.
4. **Prueba de Fallo (Sobrecarga):** Activamos el switch de sobrecarga (`SW4`). Confirmamos que el LED RGB cambia a amarillo, el display muestra 'L' y los motores se desactivan.
5. **Prueba Crítica (Emergencia):** Finalmente, activamos el switch de emergencia (`SW15`). La simulación muestra cómo la FSM interrumpe cualquier operación, pasa a estado de espera y luego inicia el parpadeo de emergencia, validando la prioridad absoluta de la seguridad.

7.4. Resultados y Visualización

A continuación, se presenta una captura del cronograma obtenido tras la ejecución del testbench completo (`TOP_ASCENSOR_TB`). En ella se pueden apreciar las transiciones de la máquina de estados y la respuesta síncrona de las señales de control:

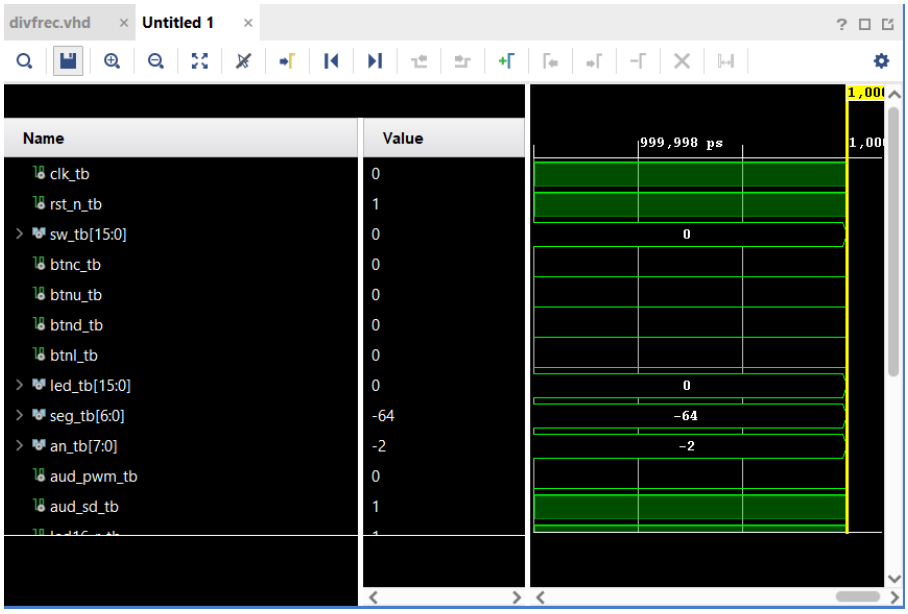

Figura 7.1: Cronograma de la simulación funcional en Vivado. Se observa la evolución de las señales de estado y motores ante los estímulos de prueba.

El análisis de estas formas de onda valida el éxito de la simulación, demostrando que:

- **Integridad de Señal:** No existen conflictos de bus ni señales indefinidas ('X') en las salidas críticas.
- **Sincronismo:** Los tiempos de *Setup* y *Hold* se cumplen correctamente; todas las transiciones ocurren alineadas con el flanco de subida del reloj.
- **Determinismo:** La lógica de control responde de manera predecible ante todas las entradas planteadas en el guion de pruebas descrito en el apartado 7.3.

8. Problemas Encontrados y Soluciones Adoptadas

Durante el desarrollo del proyecto nos enfrentamos a diversos desafíos técnicos, tanto en la fase de simulación como en la implementación física. A continuación, describimos los problemas más significativos y las soluciones de ingeniería que aplicamos para resolverlos.

8.1. Rebotes y Metaestabilidad en las Entradas

Problema: En las primeras pruebas con la placa, detectamos un comportamiento errático: al pulsar un botón de llamada una sola vez, el ascensor a veces registraba múltiples peticiones o cambiaba de estado de forma impredecible. Identificamos que esto se debía al ruido mecánico (rebotes) de los pulsadores y a la asincronía de la señal de entrada respecto al reloj de la FPGA.

Solución: Implementamos un módulo específico, `Detector_Entradas`, que actúa como filtro digital.

- Añadimos registros de desplazamiento (`in_pisos_sync`, `in_pisos_prev`) para sincronizar las señales externas con el reloj de 100 MHz.
- Diseñamos una lógica de detección de flancos que solo genera un pulso de control (`nueva_peticion`) cuando la señal es estable, ignorando el ruido transitorio.

8.2. Gestión de Múltiples Relojes (Clock Domains)

Problema: Inicialmente, consideramos la opción de dividir la frecuencia del reloj principal para crear relojes más lentos (ej. 1 Hz) y alimentar con ellos la FSM. Sin embargo, descubrimos que el uso de "Gated Clocks" o relojes derivados en FPGAs no es una buena práctica, ya que introduce problemas de *skew* (desfase) y dificulta el análisis de tiempos.

Solución: Optamos por un diseño totalmente síncrono. Mantenemos el reloj de 100 MHz (`CLK100MHZ`) como única fuente de reloj para todos los Flip-Flops del sistema. Para los procesos lentos (movimiento del motor, parpadeo), utilizamos el módulo

`Divisor_Frecuencia` para generar señales de habilitación (*clock enables*) de un solo ciclo de duración. Esto mantiene todo el sistema perfectamente sincronizado.

8.3. Verificación sin Hardware Externo

Problema: El enunciado requiere controlar un ascensor, pero al trabajar en el laboratorio o en casa, no siempre disponíamos de una maqueta física con motores y sensores reales conectados a la placa. Esto hacía imposible verificar si la lógica de control ("detenerse al llegar al piso") funcionaba correctamente, ya que el controlador se quedaba esperando infinitamente una señal de sensor que nunca llegaba.

Solución: Desarrollamos el módulo `Simulador_Planta`. Este bloque de hardware emula el comportamiento físico: recibe las órdenes de los motores y actualiza contadores internos que representan la posición. De esta forma, "engañamos" al controlador haciéndole creer que está moviendo un ascensor real, lo que nos permitió depurar toda la lógica compleja (incluido el movimiento horizontal) sin salir del entorno de desarrollo Vivado.

8.4. Conflicto de Prioridades (Vertical vs Horizontal)

Problema: Al añadir la funcionalidad extra de las habitaciones, surgió un conflicto lógico: ¿qué debía hacer el ascensor si el usuario pedía cambiar de piso y de habitación simultáneamente? En las primeras versiones, la máquina intentaba ajustar ambos parámetros a la vez, entrando en estados inconsistentes.

Solución: Rediseñamos la FSM para que fuese estrictamente secuencial. Establecimos una jerarquía de prioridades donde el error de piso (vertical) siempre se corrige antes que el error de habitación (horizontal). Esto simplificó las transiciones del diagrama de estados y eliminó los comportamientos indeseados.

9. Manual de Usuario

Este apartado describe cómo interactuar con el sistema implementado en la placa Nexys 4 DDR. El diseño utiliza una combinación de pulsadores, interruptores y displays para controlar y visualizar el estado del ascensor.

9.1. Puesta en Marcha

1. Conectar la placa Nexys 4 DDR a la alimentación (USB o externa) y encenderla.
2. Cargar el *bitstream* generado (`TOP_ASCENSOR.bit`) mediante Vivado Hardware Manager.
3. **Estado Inicial:** Al arrancar, es recomendable pulsar el botón rojo CPU_RESETN. El sistema se iniciará en el Piso 0, Habitación 1, con la puerta abierta (Display mostrando `H-01` y `O-F0`). El LED RGB derecho estará en Verde.

9.2. Control de Movimiento Vertical (Pisos)

Para enviar el ascensor a una planta distinta, utilice los pulsadores direccionales situados a la derecha de la placa:

- **BTNC (Centro):** Llamada al Piso 0.
- **BTNU (Arriba):** Llamada al Piso 1.
- **BTND (Abajo):** Llamada al Piso 2.
- **BTNL (Izquierda):** Llamada al Piso 3.

Observación: Al pulsar un botón, si el piso es distinto al actual, las puertas se cerrarán (LEDs apagándose), el indicador mostrará el movimiento (ej. `S` para Subiendo) y se escuchará una melodía si hay altavoces conectados.

9.3. Control de Movimiento Horizontal (Habitaciones)

Para desplazar el ascensor lateralmente dentro de una planta, utilice el banco de interruptores central (SW5 - SW8). Solo debe haber uno activo a la vez para evitar confusiones, aunque el sistema detectará el último cambio realizado.

- **SW5:** Selecciona Habitación 1.
- **SW6:** Selecciona Habitación 2.
- **SW7:** Selecciona Habitación 3.
- **SW8:** Selecciona Habitación 4.

Nota: Si se solicita un cambio de piso y de habitación simultáneamente, el ascensor viajará primero al piso destino y, una vez allí, se desplazará a la habitación indicada.

9.4. Funciones de Seguridad y Test

- **Simulación de Sobrecarga (SW4):** Si se activa el interruptor SW4, el sistema simula un exceso de peso.
    - *Efecto:* El ascensor se detiene, abre puertas, el display muestra `L` (Load), suena una alarma grave y el LED RGB cambia a Amarillo.
    - *Solución:* Desactivar SW4 para recuperar el control.
- **Parada de Emergencia (SW15):** El interruptor situado más a la izquierda (SW15) actúa como seta de emergencia.
    - *Efecto:* Bloqueo inmediato del sistema. Se muestra una `E` fija durante un segundo, seguida de un parpadeo total de todos los displays y LEDs (aviso visual crítico) acompañado de una alarma sonora intermitente.
    - *Recuperación:* Al desactivar SW15, el sistema realiza un protocolo de seguridad (apertura de puertas) antes de volver a estar operativo.

9.5. Guía Visual (Displays 7-Segmentos)

Los 8 dígitos de la placa se dividen en dos secciones:

- **Izquierda (Habitación):** Formato `H-0X` (donde X es 1, 2, 3 o 4).
- **Derecha (Estado y Piso):**

- `O-FX`: Puerta Abierta (**O**pen) en Piso X.
- `C-FX`: Puerta Cerrada (**C**lose) en Piso X.
- `S-FX`: **S**ubiendo hacia o pasando por Piso X.
- `b-FX`: **B**ajando hacia o pasando por Piso X.
- `=-FX`: Moviendo horizontalmente en Piso X.

10. Conclusiones y Líneas Futuras

10.1. Conclusiones

La realización de este proyecto nos ha permitido consolidar los conocimientos teóricos sobre diseño digital y sistemas empotrados. Tras completar el desarrollo y las pruebas, extraemos las siguientes conclusiones principales:

1. **Robustez del Diseño Síncrono:** La decisión de utilizar un único reloj maestro y gestionar los tiempos mediante *enables* ha sido fundamental para evitar problemas de estabilidad. El sistema se comporta de manera predecible y no presenta fallos por condiciones de carrera.
2. **Importancia de la Simulación:** La implementación del módulo `Simulador_Planta` nos sirvió para comprobar la importancia de las herramientas de simulación.
3. **Modularidad y Escalabilidad:** La arquitectura basada en bloques independientes (FSM separada del control de periféricos) ha facilitado enormemente la adición de funcionalidades extras, como el movimiento horizontal, sin necesidad de reescribir el código base del controlador vertical.

Como validación final de la implementación hardware, la siguiente imagen muestra el resultado de la síntesis física en Vivado, donde se aprecia la ocupación real de recursos lógicos dentro de la matriz de la FPGA:

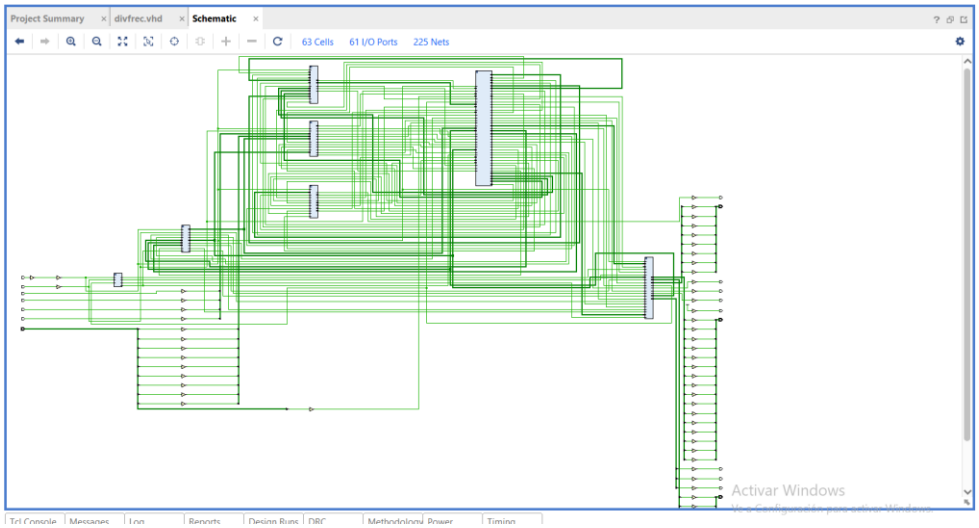

Figura 10.1: Vista del diseño implementado (Implemented Design). Se observa la distribución de recursos y rutas dentro del chip Artix-7.

El sistema final cumple holgadamente con todos los requisitos del enunciado ("6. Ascensor") y demuestra capacidades superiores mediante la integración de audio, control multi-eje y protocolos de seguridad.

11. Anexos

A continuación se adjunta la descripción completa del hardware (RTL) y los bancos de pruebas utilizados en el proyecto.

11.1. Módulo Principal (Top Level)

Archivo: `TOP_ASCENSOR.vhd`

Descripción: Entidad superior que interconecta todos los submódulos.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity TOP_ASCENSOR is
    Port ( CLK100MHZ : in STD_LOGIC;
           CPU_RESETN : in STD_LOGIC;
           SW : in STD_LOGIC_VECTOR(15 downto 0);
           BTNC, BTNU, BTND, BTNL : in STD_LOGIC;
           LED : out STD_LOGIC_VECTOR(15 downto 0);
           SEG : out STD_LOGIC_VECTOR(6 downto 0);
           AN  : out STD_LOGIC_VECTOR(7 downto 0);
           AUD_PWM, AUD_SD : out STD_LOGIC;
           LED16_R, LED16_G, LED16_B, LED17_R, LED17_G, LED17_B : out
STD_LOGIC);
end TOP_ASCENSOR;

architecture Structural of TOP_ASCENSOR is
    signal rst : std_logic;
    signal t_1, t_d, t_a, t_b, n_p, t_h, t_e, tm_s, tm_d, d_o, mus,
d_s, alm, err : std_logic;
    signal p_o, p_a : integer range 0 to 3;
    signal h_o, h_a : integer range 1 to 4;
    signal st : integer range 0 to 8;
    signal tm_dur : integer;
    signal m_v, m_h : std_logic_vector(1 downto 0);
    signal rgb : std_logic_vector(5 downto 0);
    signal btns_vec : std_logic_vector(3 downto 0);
begin
    rst <= not CPU_RESETN;
    btns_vec <= BTNL & BTND & BTNU & BTNC;

    U_Div: entity work.Divisor_Frecuencia port map(CLK100MHZ, rst,
t_1, t_d, t_b, t_a);

    U_In: entity work.Detector_Entradas port map(
        clk => CLK100MHZ,
        sw_reset => SW(15),
        switches_pisos => SW(3 downto 0),
```

```
        switches_hab => SW(8 downto 5),
        botones => btns_vec,
        piso_llamada => p_o,
        habitacion_detectada => h_o,
        nueva_peticion => n_p,
        trigger_hab => t_h,
        trigger_emergencia => t_e
    );

    U_FSM: entity work.FSM_Controlador port map(
        clk => CLK100MHZ, reset => rst, piso_actual => p_a,
piso_llamada => p_o,
        hay_peticion => n_p, trigger_hab => t_h, habitacion_in => h_o,
hab_actual => h_a,
        sw_sobrecarga => SW(4), trigger_emergencia => t_e, timer_done
=> tm_d, motor => m_v,
        motor_hor => m_h, timer_start => tm_s, timer_dur => tm_dur,
estado_vis => st,
        puerta_abierta => d_o, play_musica => mus, play_puerta => d_s,
play_alarma => alm, play_error => err
    );

    U_Tmr: entity work.Temporizador port map(CLK100MHZ, rst, t_1,
tm_s, tm_dur, tm_d);
    U_Plant: entity work.Simulador_Planta port map(CLK100MHZ, rst,
t_1, m_v, m_h, p_a, h_a);
    U_Audio: entity work.Controlador_Audio port map(CLK100MHZ, mus,
d_s, alm, err, AUD_PWM, AUD_SD);
    U_Vis: entity work.Controlador_Display port map(CLK100MHZ, t_d,
t_b, t_a, p_a, h_a, st, d_o, SEG, AN, LED, rgb);

    LED17_R <= rgb(5); LED17_G <= rgb(4); LED17_B <= rgb(3);
    LED16_R <= rgb(2); LED16_G <= rgb(1); LED16_B <= rgb(0);
end Structural;
```

11.2. Controlador Lógico (FSM)

Archivo: `FSM_Controlador.vhd`

Descripción: Máquina de estados finitos que gobierna la lógica de control.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity FSM_Controlador is
    Port ( clk, reset : in STD_LOGIC;
           piso_actual, piso_llamada : in INTEGER range 0 to 3;
           hay_peticion, trigger_hab : in STD_LOGIC;
           habitacion_in, hab_actual : in INTEGER range 1 to 4;
           sw_sobrecarga, trigger_emergencia, timer_done : in
STD_LOGIC;
```

```
            motor, motor_hor : out STD_LOGIC_VECTOR(1 downto 0);
            timer_start : out STD_LOGIC;
            timer_dur : out INTEGER;
            estado_vis : out INTEGER range 0 to 8;
            puerta_abierta : out STD_LOGIC;
            play_musica, play_puerta, play_alarma, play_error : out
STD_LOGIC);
end FSM_Controlador;

architecture Behavioral of FSM_Controlador is
    type estados_t is (IDLE_OPEN, CERRANDO, SUBIENDO, BAJANDO,
MOVIENDO_HOR, LLEGADA, ABRIENDO, EMERG_WAIT, EMERG_BLINK, SOBRECARGA);
    signal estado : estados_t := IDLE_OPEN;
    signal obj_p : integer range 0 to 3 := 0;
    signal obj_h : integer range 1 to 4 := 1;
begin
    process(clk)
    begin
        if rising_edge(clk) then
            if reset = '1' then estado <= IDLE_OPEN;
            else
                timer_start <= '0'; play_musica <= '0'; play_puerta <=
'0'; play_alarma <= '0'; play_error <= '0';
                motor <= "00"; motor_hor <= "00"; timer_dur <= 2;

                if trigger_emergencia = '1' then
                    estado <= EMERG_WAIT;
                    timer_start <= '1';
                    timer_dur <= 1; -- 1 Segundo mostrando la 'E'
                else
                    case estado is
                        when IDLE_OPEN =>
                            estado_vis <= 3; puerta_abierta <= '1';
                            if sw_sobrecarga = '1' then estado <=
SOBRECARGA;
                            elsif hay_peticion = '1' and piso_llamada
/= piso_actual then
                                obj_p <= piso_llamada; estado <=
CERRANDO; timer_start <= '1';
                            elsif trigger_hab = '1' and habitacion_in
/= hab_actual then
                                obj_h <= habitacion_in; estado <=
CERRANDO; timer_start <= '1';
                            end if;
                        when SOBRECARGA =>
                            estado_vis <= 7; puerta_abierta <= '1';
play_error <= '1';
                            if sw_sobrecarga = '0' then estado <=
IDLE_OPEN; end if;
                        when CERRANDO =>
                            estado_vis <= 4; puerta_abierta <= '0';
                            if sw_sobrecarga = '1' then estado <=
SOBRECARGA;
                            elsif timer_done = '1' then
                                if piso_actual /= obj_p then
                                    if obj_p > piso_actual then estado
<= SUBIENDO; else estado <= BAJANDO; end if;
```

```
                                elsif hab_actual /= obj_h then estado
<= MOVIENDO_HOR;
                                else estado <= ABRIENDO; timer_start
<= '1'; end if;
                            end if;
                        when SUBIENDO =>
                            motor <= "01"; estado_vis <= 1;
play_musica <= '1';
                            if piso_actual = obj_p then estado <=
LLEGADA; timer_start <= '1'; end if;
                        when BAJANDO =>
                            motor <= "10"; estado_vis <= 2;
play_musica <= '1';
                            if piso_actual = obj_p then estado <=
LLEGADA; timer_start <= '1'; end if;
                        when MOVIENDO_HOR =>
                            estado_vis <= 8; play_musica <= '1';
                            if obj_h > hab_actual then motor_hor <=
"01"; else motor_hor <= "10"; end if;
                            if hab_actual = obj_h then estado <=
LLEGADA; timer_start <= '1'; end if;
                        when LLEGADA =>
                            estado_vis <= 0; if timer_done = '1' then
estado <= ABRIENDO; timer_start <= '1'; end if;
                        when ABRIENDO =>
                            estado_vis <= 3; puerta_abierta <= '1';
play_puerta <= '1';
                            if timer_done = '1' then estado <=
IDLE_OPEN; end if;
                        when EMERG_WAIT =>
                            estado_vis <= 5; -- Mostrar E
                            if timer_done = '1' then
                                estado <= EMERG_BLINK;
                                timer_start <= '1';
                                timer_dur <= 2;
                            end if;
                        when EMERG_BLINK =>
                            estado_vis <= 6; play_alarma <= '1';
                            if timer_done = '1' then estado <=
ABRIENDO; timer_start <= '1'; end if;
                    end case;
                end if;
            end if;
        end if;
    end process;
end Behavioral;
```

11.3. Módulos Auxiliares y Periféricos

Archivo: `Divisor_Frecuencia.vhd`

Descripción: Generador de señales de habilitación (enables).

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Divisor_Frecuencia is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           en_1hz : out STD_LOGIC;
           en_disp : out STD_LOGIC;
           en_blink : out STD_LOGIC;
           en_anim : out STD_LOGIC);
end Divisor_Frecuencia;

architecture Behavioral of Divisor_Frecuencia is
    signal c_1hz : integer range 0 to 100000000 := 0;
    signal c_disp : integer range 0 to 100000 := 0;
    signal c_blink : integer range 0 to 20000000 := 0;
    signal c_anim : integer range 0 to 5000000 := 0;
begin
    process(clk)
    begin
        if rising_edge(clk) then
            if reset = '1' then
                c_1hz <= 0; c_disp <= 0; c_blink <= 0; c_anim <= 0;
            else
                if c_1hz = 99999999 then c_1hz <= 0; en_1hz <= '1';
                else c_1hz <= c_1hz + 1; en_1hz <= '0'; end if;

                if c_disp = 49999 then c_disp <= 0; en_disp <= '1';
                else c_disp <= c_disp + 1; en_disp <= '0'; end if;

                if c_blink = 9999999 then c_blink <= 0; en_blink <=
'1';
                else c_blink <= c_blink + 1; en_blink <= '0'; end if;

                if c_anim = 4999999 then c_anim <= 0; en_anim <= '1';
                else c_anim <= c_anim + 1; en_anim <= '0'; end if;
            end if;
        end if;
    end process;
end Behavioral;
```

Archivo: `Detector_Entradas.vhd`

Descripción: Sincronizador y detector de flancos.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Detector_Entradas is
    Port ( clk : in STD_LOGIC;
           sw_reset : in STD_LOGIC;
           switches_pisos : in STD_LOGIC_VECTOR(3 downto 0);
           switches_hab : in STD_LOGIC_VECTOR(3 downto 0);
           botones : in STD_LOGIC_VECTOR(3 downto 0);
```

```
            piso_llamada : out INTEGER range 0 to 3;
            habitacion_detectada : out INTEGER range 0 to 4;
            nueva_peticion : out STD_LOGIC;
            trigger_hab : out STD_LOGIC;
            trigger_emergencia : out STD_LOGIC);
end Detector_Entradas;

architecture Behavioral of Detector_Entradas is
    signal in_pisos_sync, in_pisos_prev : std_logic_vector(7 downto
0);
    signal in_hab_sync, in_hab_prev : std_logic_vector(3 downto 0);
    signal sw_res_sync, sw_res_prev : std_logic := '0';
begin
    process(clk)
    begin
        if rising_edge(clk) then
            in_pisos_sync <= switches_pisos & botones;
            in_pisos_prev <= in_pisos_sync;
            in_hab_sync <= switches_hab;
            in_hab_prev <= in_hab_sync;
            sw_res_sync <= sw_reset;
            sw_res_prev <= sw_res_sync;

            nueva_peticion <= '0'; trigger_emergencia <= '0';
trigger_hab <= '0';

            if sw_res_sync /= sw_res_prev then trigger_emergencia <=
'1'; end if;

            if (in_pisos_sync(7)/=in_pisos_prev(7)) or
(in_pisos_sync(3)='1' and in_pisos_prev(3)='0') then
                piso_llamada <= 3; nueva_peticion <= '1';
            elsif (in_pisos_sync(6)/=in_pisos_prev(6)) or
(in_pisos_sync(2)='1' and in_pisos_prev(2)='0') then
                piso_llamada <= 2; nueva_peticion <= '1';
            elsif (in_pisos_sync(5)/=in_pisos_prev(5)) or
(in_pisos_sync(1)='1' and in_pisos_prev(1)='0') then
                piso_llamada <= 1; nueva_peticion <= '1';
            elsif (in_pisos_sync(4)/=in_pisos_prev(4)) or
(in_pisos_sync(0)='1' and in_pisos_prev(0)='0') then
                piso_llamada <= 0; nueva_peticion <= '1';
            end if;

            if in_hab_sync(0) /= in_hab_prev(0) then
habitacion_detectada <= 1; trigger_hab <= '1';
            elsif in_hab_sync(1) /= in_hab_prev(1) then
habitacion_detectada <= 2; trigger_hab <= '1';
            elsif in_hab_sync(2) /= in_hab_prev(2) then
habitacion_detectada <= 3; trigger_hab <= '1';
            elsif in_hab_sync(3) /= in_hab_prev(3) then
habitacion_detectada <= 4; trigger_hab <= '1';
            end if;
        end if;
    end process;
end Behavioral;
```

Archivo: `Temporizador.vhd`

Descripción: Contador parametrizable.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Temporizador is
    Port ( clk, reset : in STD_LOGIC;
           en_1hz : in STD_LOGIC;
           start : in STD_LOGIC;
           duracion : in INTEGER;
           fin_tiempo : out STD_LOGIC);
end Temporizador;

architecture Behavioral of Temporizador is
    signal contador : integer range 0 to 10 := 0;
    signal activo : std_logic := '0';
begin
    process(clk)
    begin
        if rising_edge(clk) then
            if reset = '1' then
                contador <= 0; activo <= '0'; fin_tiempo <= '0';
            else
                fin_tiempo <= '0';
                if start = '1' then
                    contador <= 0; activo <= '1';
                elsif activo = '1' and en_1hz = '1' then
                    if contador >= duracion - 1 then
                        fin_tiempo <= '1'; activo <= '0';
                    else contador <= contador + 1; end if;
                end if;
            end if;
        end if;
    end process;
end Behavioral;
```

Archivo: `Simulador_Planta.vhd`

Descripción: Modelo físico del ascensor.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Simulador_Planta is
    Port ( clk, reset : in STD_LOGIC;
           en_1hz : in STD_LOGIC;
           motor : in STD_LOGIC_VECTOR(1 downto 0);
           motor_hor : in STD_LOGIC_VECTOR(1 downto 0);
           piso_actual : out INTEGER range 0 to 3;
           hab_actual : out INTEGER range 1 to 4);
end Simulador_Planta;
```

```
architecture Behavioral of Simulador_Planta is
    signal p_reg : integer range 0 to 3 := 0;
    signal h_reg : integer range 1 to 4 := 1;
begin
    process(clk)
    begin
        if rising_edge(clk) then
            if reset = '1' then p_reg <= 0; h_reg <= 1;
            elsif en_1hz = '1' then
                if motor = "01" and p_reg < 3 then p_reg <= p_reg + 1;
                elsif motor = "10" and p_reg > 0 then p_reg <= p_reg -
1;
                end if;
                if motor_hor = "01" and h_reg < 4 then h_reg <= h_reg
+ 1;
                elsif motor_hor = "10" and h_reg > 1 then h_reg <=
h_reg - 1;
                end if;
            end if;
        end if;
    end process;
    piso_actual <= p_reg; hab_actual <= h_reg;
end Behavioral;
```

Archivo: `Controlador_Display.vhd`

Descripción: Multiplexor para displays y LEDs.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Controlador_Display is
    Port ( clk, en_disp, en_blink, en_anim : in STD_LOGIC;
           piso : in INTEGER range 0 to 3;
           habitacion : in INTEGER range 1 to 4;
           estado_vis : in INTEGER range 0 to 8;
           puerta_abierta : in STD_LOGIC;
           seg : out STD_LOGIC_VECTOR(6 downto 0);
           an : out STD_LOGIC_VECTOR(7 downto 0);
           leds : out STD_LOGIC_VECTOR(15 downto 0);
           rgb_leds : out STD_LOGIC_VECTOR(5 downto 0));
end Controlador_Display;

architecture Behavioral of Controlador_Display is
    signal mux : integer range 0 to 7 := 0;
    signal led_level : integer range 0 to 16 := 0;
    signal blink_state : std_logic := '0';
    signal an_s    : std_logic_vector(7 downto 0);
    signal seg_s   : std_logic_vector(6 downto 0);
    signal rgb_s   : std_logic_vector(5 downto 0);
    signal leds_s  : std_logic_vector(15 downto 0);
begin
```

```
    process(clk)
    begin
        if rising_edge(clk) then
            if en_blink = '1' then blink_state <= not blink_state; end
if;
            if en_disp = '1' then
                if mux = 7 then mux <= 0; else mux <= mux + 1; end if;
            end if;
        end if;
    end process;

    process(clk)
    begin
        if rising_edge(clk) then
            if estado_vis = 6 then -- Emergencia Parpadeo
                if blink_state = '1' then
                    an_s <= "00000000"; seg_s <= "0000000";
                else
                    an_s <= "11111111"; seg_s <= "1111111";
                end if;
            else
                case mux is
                    when 0 => an_s <= "11111110";
                        case piso is
                            when 0 => seg_s <= "1000000"; when 1 =>
seg_s <= "1111001";
                            when 2 => seg_s <= "0100100"; when others
=> seg_s <= "0110000";
                        end case;
                    when 1 => an_s <= "11111101"; seg_s <= "0001110";
-- F
                    when 2 => an_s <= "11111011"; seg_s <= "0111111";
-- -
                    when 3 => an_s <= "11110111";
                        case estado_vis is
                            when 0 => seg_s <= "0001100"; -- P
                            when 1 => seg_s <= "0010010"; -- S
                            when 2 => seg_s <= "0000011"; -- b
                            when 3 => seg_s <= "1000000"; -- O
                            when 4 => seg_s <= "1000110"; -- C
                            when 5 => seg_s <= "0000110"; -- E
                            when 7 => seg_s <= "1000111"; -- L
                            when others => seg_s <= "0001001"; -- H
                        end case;
                    when 4 => an_s <= "11101111";
                        case habitacion is
                            when 1 => seg_s <= "1111001"; when 2 =>
seg_s <= "0100100";
                            when 3 => seg_s <= "0110000"; when others
=> seg_s <= "0011001";
                        end case;
                    when 5 => an_s <= "11011111"; seg_s <= "1000000";
-- 0
                    when 6 => an_s <= "10111111"; seg_s <= "0111111";
-- -
                    when 7 => an_s <= "01111111"; seg_s <= "0001001";
-- H
```

```
                    when others => an_s <= "11111111"; seg_s <=
"1111111";
                end case;
            end if;

            -- SEMÁFORO RGB
            case estado_vis is
                when 1 | 2 | 4 | 8 => rgb_s <= "100100"; -- ROJO
                when 5 | 6 => if blink_state = '1' then rgb_s <=
"001001"; else rgb_s <= "000000"; end if; -- AZUL
                when 7 => rgb_s <= "110110"; -- AMARILLO
                when others => rgb_s <= "010010"; -- VERDE
            end case;

            -- PUERTAS (LEDS)
            if en_anim = '1' then
                if puerta_abierta = '1' then
                    if led_level < 16 then led_level <= led_level + 1;
end if;
                else
                    if led_level > 0 then led_level <= led_level - 1;
end if;
                end if;
            end if;
            for i in 0 to 15 loop
                if i < led_level then leds_s(i) <= '1'; else leds_s(i)
<= '0'; end if;
            end loop;
        end if;
    end process;

    an <= an_s; seg <= seg_s; rgb_leds <= rgb_s; leds <= leds_s;
end Behavioral;
```

Archivo: `Controlador_Audio.vhd`

Descripción: Generador de tonos PWM.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Controlador_Audio is
    Port ( clk : in STD_LOGIC;
           enable_music, enable_door, enable_alarm, enable_overload :
in STD_LOGIC;
           audio_out, audio_sd : out STD_LOGIC);
end Controlador_Audio;

architecture Behavioral of Controlador_Audio is
    signal counter_freq, current_period, tempo_cnt : integer := 0;
    signal audio_toggle : std_logic := '0';
    signal note_index : integer range 0 to 3 := 0;
begin
    audio_sd <= '1';
```

```
    process(clk)
    begin
        if rising_edge(clk) then
            tempo_cnt <= tempo_cnt + 1;
            if enable_alarm = '1' then
                if tempo_cnt < 25000000 then current_period <= 250000;
                elsif tempo_cnt < 50000000 then current_period <= 0;
                else tempo_cnt <= 0; end if;
            elsif enable_overload = '1' then
                if tempo_cnt < 15000000 then current_period <= 450000;
                elsif tempo_cnt < 30000000 then current_period <= 0;
                else tempo_cnt <= 0; end if;
            elsif enable_door = '1' then
                if tempo_cnt < 25000000 then current_period <= 95556;
else current_period <= 127553; end if;
            elsif enable_music = '1' then
                if tempo_cnt > 10000000 then
                    tempo_cnt <= 0; if note_index = 3 then note_index
<= 0; else note_index <= note_index + 1; end if;
                end if;
                case note_index is
                    when 0 => current_period <= 191113; when 1 =>
current_period <= 151686;
                    when 2 => current_period <= 127553; when others =>
current_period <= 151686;
                end case;
            else current_period <= 0; tempo_cnt <= 0; end if;

            if current_period > 0 then
                counter_freq <= counter_freq + 1;
                if counter_freq >= current_period then counter_freq <=
0; audio_toggle <= not audio_toggle; end if;
                audio_out <= audio_toggle;
            else audio_out <= '0'; counter_freq <= 0; end if;
        end if;
    end process;
end Behavioral;
```

11.4. Simulación (Testbench)

Archivo: `TOP_ASCENSOR_TB.vhd`

Descripción: Banco de pruebas general.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity TOP_ASCENSOR_TB is
end TOP_ASCENSOR_TB;

architecture Behavioral of TOP_ASCENSOR_TB is
    component TOP_ASCENSOR
        Port (
```

```
            CLK100MHZ : in STD_LOGIC; CPU_RESETN : in STD_LOGIC; SW :
in STD_LOGIC_VECTOR(15 downto 0);
            BTNC, BTNU, BTND, BTNL : in STD_LOGIC; LED : out
STD_LOGIC_VECTOR(15 downto 0);
            SEG : out STD_LOGIC_VECTOR(6 downto 0); AN : out
STD_LOGIC_VECTOR(7 downto 0);
            AUD_PWM, AUD_SD : out STD_LOGIC;
            LED16_R, LED16_G, LED16_B, LED17_R, LED17_G, LED17_B : out
STD_LOGIC
        );
    end component;

    signal clk_tb, rst_n_tb : std_logic := '0';
    signal sw_tb : std_logic_vector(15 downto 0) := (others => '0');
    signal btnc_tb, btnu_tb, btnd_tb, btnl_tb : std_logic := '0';
    signal led_tb : std_logic_vector(15 downto 0);
    signal seg_tb : std_logic_vector(6 downto 0);
    signal an_tb : std_logic_vector(7 downto 0);
    signal aud_pwm_tb, aud_sd_tb : std_logic;
    signal led16_r_tb, led16_g_tb, led16_b_tb, led17_r_tb, led17_g_tb,
led17_b_tb : std_logic;

    constant CLK_PERIOD : time := 10 ns;
begin
    uut: TOP_ASCENSOR Port Map (
        CLK100MHZ => clk_tb, CPU_RESETN => rst_n_tb, SW => sw_tb,
        BTNC => btnc_tb, BTNU => btnu_tb, BTND => btnd_tb, BTNL =>
btnl_tb,
        LED => led_tb, SEG => seg_tb, AN => an_tb,
        AUD_PWM => aud_pwm_tb, AUD_SD => aud_sd_tb,
        LED16_R => led16_r_tb, LED16_G => led16_g_tb, LED16_B =>
led16_b_tb,
        LED17_R => led17_r_tb, LED17_G => led17_g_tb, LED17_B =>
led17_b_tb
    );

    clk_process : process
    begin
        clk_tb <= '0'; wait for CLK_PERIOD/2; clk_tb <= '1'; wait for
CLK_PERIOD/2;
    end process;

    stim_proc: process
    begin
        report "--- INICIO SIMULACION ---";
        rst_n_tb <= '0'; sw_tb <= (others => '0'); wait for 100 ns;
        rst_n_tb <= '1'; wait for 100 ns;

        report "Prueba 1: Piso 2"; btnd_tb <= '1'; wait for 200 ns;
btnd_tb <= '0'; wait for 2 us;
        report "Prueba 2: Hab 3"; sw_tb(7) <= '1'; wait for 200 ns;
sw_tb(7) <= '0'; wait for 2 us;
        report "Prueba 3: Sobrecarga"; sw_tb(4) <= '1'; wait for 1 us;
sw_tb(4) <= '0'; wait for 500 ns;
        report "Prueba 4: Emergencia"; sw_tb(15) <= '1'; wait for 500
ns; sw_tb(15) <= '0';
        wait for 5 us; report "--- FIN SIMULACION ---"; wait;
```

```
    end process;
end Behavioral;
```

Archivo: `Testbenches Unitarios` (Nota: Se incluyen en el archivo comprimido del proyecto para no extender el anexo innecesariamente).

11.5. Restricciones Físicas (Constraints)

Archivo: `Nexys4_DDR_Master.xdc`

Descripción: Asignación de pines físicos de la FPGA a los puertos lógicos del diseño.

```
## This file is a general .xdc for the NexysA7 100T Rev. C
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top level signal names in the project

## Clock signal
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { CLK100MHZ }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK100MHZ}];


##Switches

set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { SW[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { SW[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { SW[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { SW[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { SW[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { SW[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { SW[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports { SW[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports { SW[8] }]; #IO_L24N_T3_34 Sch=sw[8]
#set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports { SW[9] }]; #IO_25_34 Sch=sw[9]
#set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports { SW[10] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
#set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports { SW[11] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
#set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports { SW[12] }]; #IO_L24P_T3_35 Sch=sw[12]
#set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports { SW[13] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
#set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports { SW[14] }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports { SW[15] }]; #IO_L21P_T3_DQS_14 Sch=sw[15]


## LEDs

set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { LED[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { LED[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { LED[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { LED[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { LED[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { LED[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { LED[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports { LED[7] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { LED[8] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports { LED[9] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports { LED[10] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { LED[11] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports { LED[12] }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports { LED[13] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports { LED[14] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports { LED[15] }]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]

set_property -dict { PACKAGE_PIN R12   IOSTANDARD LVCMOS33 } [get_ports { LED16_B }]; #IO_L5P_T0_D06_14 Sch=led16_b
set_property -dict { PACKAGE_PIN M16   IOSTANDARD LVCMOS33 } [get_ports { LED16_G }]; #IO_L10P_T1_D14_14 Sch=led16_g
set_property -dict { PACKAGE_PIN N15   IOSTANDARD LVCMOS33 } [get_ports { LED16_R }]; #IO_L11P_T1_SRCC_14 Sch=led16_r
set_property -dict { PACKAGE_PIN G14   IOSTANDARD LVCMOS33 } [get_ports { LED17_B }]; #IO_L15N_T2_DQS_ADV_B_15 Sch=led17_b
set_property -dict { PACKAGE_PIN R11   IOSTANDARD LVCMOS33 } [get_ports { LED17_G }]; #IO_0_14 Sch=led17_g
set_property -dict { PACKAGE_PIN N16   IOSTANDARD LVCMOS33 } [get_ports { LED17_R }]; #IO_L11N_T1_SRCC_14 Sch=led17_r
```

```
##7 segment display

set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { SEG[0] }]; #IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports { SEG[1] }]; #IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports { SEG[2] }]; #IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports { SEG[3] }]; #IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { SEG[4] }]; #IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { SEG[5] }]; #IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports { SEG[6] }]; #IO_L4P_T0_D04_14 Sch=cg

#set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports { DP }]; #IO_L19N_T3_A21_VREF_15 Sch=dp

set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports { AN[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports { AN[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports { AN[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports { AN[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { AN[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports { AN[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports { AN[6] }]; #IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports { AN[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]


##Buttons

set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 } [get_ports { CPU_RESETN }]; #IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn

#set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { BTNC }]; #IO_L9P_T1_DQS_14 Sch=btnc
set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { BTNC }]; #IO_L4N_T0_D05_14 Sch=btnu Piso 0
set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports { BTNL }]; #IO_L12P_T1_MRCC_14 Sch=btnl Piso 3
set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports { BTNU }]; #IO_L10N_T1_D15_14 Sch=btnr Piso 1
set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports { BTND }]; #IO_L9N_T1_DQS_D13_14 Sch=btnd Piso 2
```
