# MEMORIA DEL PROYECTO DEL PROCESADOR

Proyecto de Ingeniería de Computadores

31/05/2023

Víctor Mena Doz
Rubén Juánez Jiménez

# Índice

**1. Introducción..... 3**

1.1. Finalidad del proyecto..... 3

**2. Entorno de desarrollo y recursos necesarios..... 5**

**3. Etapas de la planificación..... 6**

3.1. Etapa 1: Procesador Base..... 6

3.2. Etapa 2.1: Procesador Multiciclo..... 6

3.3. Etapa 2.2: Controlador de Memoria..... 7

3.4. Etapa 3: Unidad Aritmeticológica (ALU)..... 8

3.5. Etapa 4: Instrucciones de Salto..... 8

3.6. Etapa 5: Entrada/Salida básica..... 9

3.7. Etapa 6: Controlador de teclado PS/2 y video VGA..... 9

3.8. Etapa 7.1: Interrupciones..... 9

3.9. Etapa 7.2: Excepciones..... 11

3.10. Etapa 7.3: Modo privilegiado y Llamadas a Sistema..... 12

3.11. Etapa 8: TLB (Translation Lookaside Buffer)..... 13

**4. Puntos destacables del procesador..... 14**

4.1. Partes ventajosas de la implementación..... 14

4.2. Partes mejorables de la implementación..... 15

**5. Conclusión..... 16**

**6. Posibles ampliaciones del Proyecto..... 17**

**7. Bibliografía..... 18**

**8. Anexos..... 19**

Figura 1. Placa DE1 Terasic..... 19

Figura 2. Codificación de las 55 instrucciones SISA..... 20

Figura 3. Protocolo de lectura controlador de memoria..... 21

Figura 4. Protocolo de lectura controlador de memoria..... 22

Figura 5. Cambios en las señales durante el estado SYSTEM..... 22

Figura 6. TLBs dentro del camino de datos..... 23

# 1. Introducción

## 1.1. Finalidad del proyecto

En los últimos años, han emergido diferentes tecnologías para la creación de prototipos de procesadores. La metodología más usada hoy en día es hacer uso de chips programables para poder poner a prueba el diseño de nuestro SoC antes de pasar a la producción física del mismo.

El objetivo de la realización de este proyecto es mostrar la creación de un prototipo de un computador, desde cero, usando lenguaje de programación VHDL y otras herramientas que se comentan a lo largo de esta memoria.

Los alumnos que cursamos la asignatura de Proyecto de Ingeniería de Computadores adquirimos conocimientos muy importantes de cara al mundo laboral relacionado con la implementación de hardware a partir de lenguajes como VHDL.

Al finalizar el cuatrimestre y por ende el proyecto, se obtiene soltura con la sintaxis de VHDL, el uso de una FPGA para hacer pruebas, técnicas de debug de señales y protocolos a implementar dentro del procesador, mejora en el desarrollo de código legible y capacidades personales como ahora el trabajo en equipo.

El proyecto se ha planificado en diversas etapas en las que se plantean diferentes objetivos para ir avanzando en la implementación del procesador de una forma ordenada. La duración del mismo ha sido aproximadamente de unos 3 meses y medio. La estructura de desarrollo de cada etapa ha sido aproximadamente la siguiente:

- **Etapa 1:** una sesión de dos horas y horas sueltas.
- **Etapa 2.1:** una semana.
- **Etapa 2.2:** una semana.
- **Etapa 3:** una sesión de dos horas y horas sueltas.
- **Etapa 4:** una sesión de dos horas y horas sueltas.
- **Etapa 5:** una semana.
- **Etapa 6:** una semana.
- **Etapa 7.1:** dos semanas.
- **Etapa 7.2:** dos semanas.
- **Etapa 7.3:** una semana.
- **Etapa 8:** una semana y media.

Durante el proceso de implementación de nuestro procesador hemos creado y comprobado diversos juegos de prueba que nos han ayudado a comprobar si el funcionamiento de nuestro SoC era el esperado. Además, hemos hecho servir otras herramientas, como el Modelsim, que nos permitían visualizar los valores de nuestras señales en todo momento.

Después de dedicar muchas horas a la comprobación del funcionamiento de nuestra implementación, podemos asegurar que todo el desarrollo de nuestro código tiene el comportamiento esperado.

La estructura de esta memoria es bastante sencilla. Primero se especificará cuál ha sido el entorno de desarrollo y qué recursos han sido necesarios para llevar a cabo el diseño. A continuación se explican decisiones de implementación por cada etapa del proyecto y los pertinentes protocolos que se han tenido que implementar. Más adelante se hace autocrítica de nuestro diseño, destacando los puntos mejorables y los que creemos que nuestra implementación es francamente beneficiosa. Seguirá con ciertas conclusiones extraídas después de estos cuatro meses de trabajo y preparación. Finalmente, se expondrán algunas posibles ampliaciones del proyecto que se han planteado durante este tiempo.

# 2. Entorno de desarrollo y recursos necesarios

Para la realización de este proyecto, que ha consistido en desarrollar un prototipo de un computador desde cero, hemos necesitado usar diferentes programas que nos han ayudado a lo largo de la implementación.

La principal herramienta que hemos utilizado para la implementación del software del sistema, usando lenguaje VHDL, es el software de diseño FPGA de Intel, también conocido como *Quartus II* en la versión *13.0sp1*. Dentro de este programa se ha creado el proyecto del procesador y se han ido añadiendo todos los archivos nuevos necesarios para la implementación de nuevos módulos que requería el procesador. Además, se ha usado para compilar el diseño, examinar diagramas RTL y configurar el dispositivo de destino con el programador.

Junto con el software mencionado anteriormente, también se ha usado ModelSim que es un entorno multilingüe para la simulación de lenguajes de descripción de hardware. Ha permitido hacer un análisis temporal de los valores que tomaban en cada momento todas las señales del procesador cuando se ejecutaba un código ensamblador concreto.

Por otra parte, Quartus también permite trabajar con *SignalTap II Logic Analyzer*, el cual permite recoger muestras durante la ejecución a partir del uso de *triggers*, es decir, cambios de señales selectas. Es altamente configurable y permite observar qué es lo que está sucediendo en la placa en todo momento.

Por último, comentar que se ha usado una placa de desarrollo *DE1* del fabricante *Terasic*, que incorpora una FPGA de *Altera* de la familia *Cyclone II*. Además, incorpora una gran cantidad de componentes. En este proyecto se ha hecho uso de la SRAM, los pulsadores, los interruptores, los LEDs, los LCD Displays y el puerto de VGA, aparte de la propia FPGA. En los anexos se encuentra una imagen de la placa mostrando alguna de sus características.

# 3. Etapas de la planificación

## 3.1. Etapa 1: Procesador Base

Para iniciar el proyecto se tuvieron que crear todos los módulos indispensables para la creación de un procesador que se usaría como base. Se precisaba de mucha documentación para entender cada una de las partes que se desarrollaron.

Una vez hecho el estudio previo, se empezó por la creación de un banco de registros con 8 registros de 16 bits en el archivo **regfile.vhd**.

Seguidamente, se añadió una unidad aritmeticológica (**alu.vhd**) que, de momento, solo tendría dos instrucciones: MOVI y MOVHI. Para que pusieran pasar los datos entre la ALU y el banco de registros se requiere del **datapath.vhd** que interconecta los puertos de cada módulo.

Por último, se requería de una unidad de control que decodificase las instrucciones y crease las señales de control. Este módulo se divide en dos archivos: **unidad_control.vhd**: que hace el control del pc actual y el siguiente, además de pasar la instrucción al componente control_l (**control_l.vhd**) para que este decodifique cada campo de la misma.

## 3.2. Etapa 2.1: Procesador Multiciclo

Una vez el procesador base es desarrollado y funciona correctamente, ha pasado todos los juegos de prueba que comprueban su funcionamiento, es hora de convertirlo en multiciclo.

Primero de todo, se añade toda la lógica de lectura y escritura de memoria en el procesador. Aunque todavía no hay controlador de memoria y es por esto que sigue siendo necesario hacer las pruebas desde el ModelSim. Para hacer esto posible, el procesador tiene que ser multiciclo: un primer ciclo de *fetch* (F) y un segundo ciclo donde se hace el *decode*, el acceso a memoria y la ejecución de la instrucción (DEMW). Toda esta lógica está especificada en el archivo **multi.vhd** que se situa dentro de la unidad de control.

Al implementar las nuevas instrucciones ST, LD, STB y LDB se requieren cambios en casi todos los módulos del procesador. El más afectado es el **control_l.vhd** donde se hace la decodificación de las instrucciones y se calculan los inmediatos.

## 3.3. Etapa 2.2: Controlador de Memoria

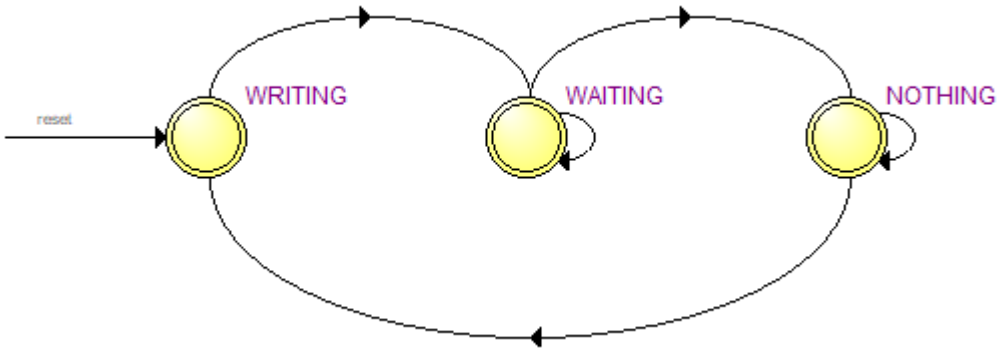
Para la creación del controlador de memoria que usa la placa DE1 es necesario recabar información sobre los protocolos de escritura y lectura especificados por el fabricante de la placa en cuestión. En la figura 3 y 4 de los anexos se encuentran los diagramas temporales de los protocolos usados. Una vez adquiridos los conocimientos requeridos, se toman las siguientes decisiones de implementación:

- El grafo de estados estará formado por tres estados: NOTHING, WRITING y WAITING.

  En NOTHING es cuando se realiza una lectura o no se requiere de ninguna operación de entrada o salida de la memoria.

  Para pasar a WRITING la señal WR tiene que estar a uno, en otras palabras, se debe solicitar la escritura. En este estado solo se estará un ciclo de memoria y saltaremos rápidamente al estado WAITING. En este último es hasta que la señal de WR vuelva a valer 0.

  Inicialmente, no se usaba el estado WAITING, pero se decide añadir para así conseguir hacer la escritura de una forma mucho más rápida, de tal forma que solo se necesita una cuarta parte de un ciclo para realizarla. Esta ventaja será usada seguramente para la parte final del proyecto, pudiendo mejorar la velocidad de ejecución de instrucciones o escribir más seguido.




- Para hacer las escrituras se usará de media un ciclo, aunque realmente solo se requiera una cuarta parte de este y para las lecturas un ciclo también. Esto es debido a que la memoria tiene un clock que va a una frecuencia 8 veces superior:
  - CLOCK PROCESADOR: 6,25 MHz
  - CLOCK MEMORIA: 50 MHz

Todos estos cambios se aplican en un nuevo archivo nombrado **MemoryController.vhd**.

## 3.4. Etapa 3: Unidad Aritmeticológica (ALU)

En esta etapa se pide codificar muchas nuevas instrucciones y diferenciarlas en la ALU. Es por esto que se decide crear un archivo auxiliar **const_alu.vhd** donde se hace una especificación de todos los *opcodes* que se han requerido para diferenciar las operaciones dentro de la ALU, así se pueden estructurar de una forma más legible en el código. En el mismo archivo se han añadido los resultados que se pueden obtener de las operaciones de comparación.

De las modificaciones necesarias, existe una que es añadir una señal en la ALU para especificar si la salida es un cero o no lo es. La señal, llamada *z*, sale de la comparación de *0* con el bus *y*. Se ha implementado de esta forma porque hace reducir el tiempo que se tarda en tomar la decisión de hacer un salto.

Existen un gran número de juegos de prueba debido a que hay muchos cambios que añadir en la unidad de control, para decodificar la instrucción, y en la ALU. Todos se han observado rigurosamente y han sido de utilidad para encontrar fallos.

## 3.5. Etapa 4: Instrucciones de Salto

En esta siguiente etapa se requiere la implementación de saltos. Al igual que en la anterior, hace falta cambios en la **ALU** y en **control_l**. Además, hace falta un nuevo multiplexor en el ciclo de *fetch* para escoger correctamente el PC en caso de que pueda haber algún tipo de salto.

Para realizar estas nuevas instrucciones se usarían dos nuevas señales: *tknbranch* y *in_d*. En la implementación no se han usado y en su lugar se han creado dos funciones, implementadas en el archivo *func_ayuda_control_pkg* que son las que dictaminan si se requiere alterar la secuencia de ejecución de instrucciones y de qué forma. Debido a esto, los respectivos cálculos de las direcciones de salto se hacen en el archivo **unidad_control.vhd** justo después de saber los valores de retorno de una de las dos funciones.

Por último, comentar que no se ha requerido la ALU para hacer el paso de la dirección absoluta de salto que usa cada instrucción de salto, sino que directamente se ha añadido una señal *jump_dir* que viene directamente del banco de registros a la unidad de control para modificar el PC. Así el dato llega de una forma más directa y se invierte menos tiempo.

## 3.6. Etapa 5: Entrada/Salida básica

El procesador requiere soporte para las instrucciones que permiten la comunicación con los dispositivos de entrada y salida. Para hacer posible se crea un banco de registros que servirá para esta comunicación explicada.

Los registros de este banco están mapeados directamente hacia los dispositivos de entrada y salida, como los LEDs y los visores de *7-segmentos*. La especificación de este banco se encuentra en el archivo **controladores_IO.vhd**.

Los controladores de los LEDs y los visores de *7-segmentos* se han reciclado de prácticas hechas anteriormente. Algunos de los archivos son **driver7Segmentos.vhd** y **driverSegmentos.vhd**.

## 3.7. Etapa 6: Controlador de teclado PS/2 y video VGA

Para que fuese el desarrollo de una computadora requería tener controladores para permitir el uso de teclado y pantalla. Ambos controladores ya estaban implementados y los reciclamos. Los únicos cambios que se debieron realizar fueron para conectarlos adecuadamente al procesador.

El controlador de teclado PS2 funciona por encuesta y no funciona por interrupción. Para su correcto funcionamiento era necesario gestionar y conectar bien las señales *read_char*, *clear_char* y *data_ready*. La gestión de las mismas se realiza en **controladores_IO.vhd** donde se hace uso de los registros creado en la anterior etapa.

El controlador VGA para el SISA tiene una complejidad más elevada. Se compone por cuatro módulos distintos: **vga_controller**, **vga_ram_dual**, **vga_font_rom**, **vga_sync**. En el controlador de memoria se añaden nuevas señales y se reserva el rango 0xA000 - 0xBFFF para la memoria de video. Este controlador es añadido en el **sisa.vhd**.

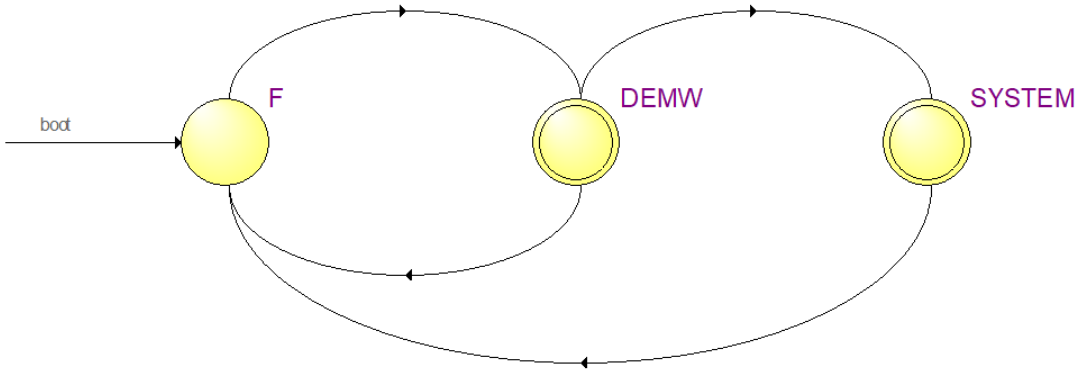
Existen diferentes juegos de prueba para esta etapa, pero el más interesante es el *snake*. Donde cuando todo funciona puedes correr el juego de la serpiente con el procesador implementado hasta esta etapa.

## 3.8. Etapa 7.1: Interrupciones

Para poder habilitar esta funcionalidad, se ha añadido un nuevo estado que prepara al procesador para saltar a la rutina de atención a interrupciones, el estado SYSTEM. Tal y como se puede ver en el grafo, a este estado solo se puede llegar desde el estado de DEMW y tiene como condición de salto que la señal *intr* este activa.

En el estado SYSTEM, únicamente se debe poder modificar el valor del registro pc**,** y el valor de los registros de sistema, los cuales se modifican a través de las operaciones

del banco de registros, por tanto, todas las otras señales de permiso de escritura o que permitan modificar el estado del procesador han estado inhibidas.




Para poder gestionar el salto a este nuevo estado, se ha creado el archivo **controlador_interrupciones.** Este módulo se sitúa dentro del controlador de entrada y salida. Esto es debido a que las interrupciones son generadas por los dispositivos de entrada y, por tanto, las interrupciones que puedan generar son parte del sistema de entrada y salida de estos dispositivos.

Para la implementación de este módulo se ha decidido usar un registro de desacoplo que memoriza el *iid*, del dispositivo que causa la interrupción. Este, está sincronizado con los flancos ascendentes de *inta*. Cabe destacar que *inta* es una señal síncrona, que solo se genera durante el estado de DEMW, cuando se ejecuta GETIID.

El hecho de tener el *iid* síncrono con *inta*, garantiza que solo se envía el *inta* correspondiente del módulo más prioritario que ha generado la interrupción antes de empezar a tratarla.

En caso de que no se hiciese esta sincronización, si se recibiera una nueva petición de interrupción, mientras que *inta* estuviera activo y diera el que esta petición sea de más prioridad de la que se estuviera tratando, tanto el dispositivo de más prioridad como el que se estaba tratando recibirían *inta*, pero solo se devolvería el *iid* de la interrupción más prioritaria y, por tanto, solo se trataría por software, la más prioritaria.

Para poder generar las interrupciones de los interruptores y pulsadores, se ha creado un nuevo módulo: **int_controller.** Este archivo es un genérico que tiene como entrada un bus de un determinado tamaño, el cual se especifica a través de *IN_VEC_SIZE***.** Este módulo memoriza el valor de la entrada, para cuando se produzca un cambio, activar la señal *intr*, que no se bajará hasta que no se active *inta*.

Por otra banda, para implementar la interrupción que se tiene que recibir cada 1/20 segundos, se ha creado el módulo **timmer.** Este módulo genera una señal cuadrada a la que se le puede programar la frecuencia a través de la entrada genérica *numero_inicial*. El periodo de esta señal es igual a 2**numero_inicial.*

A continuación, los registros de sistema se han añadido en **regfile.** Este es el encargado de almacenar todos datos relacionados con el salto a la rutina de atención de interrupción y la vuelta al punto donde se interrumpió la ejecución. Para poder hacer los cambios pertinentes sobre estos registros, se puede usar la señal *reg_op* para hacer varias operaciones en un mismo ciclo, esto es útil para las instrucciones EI, DI, GETIID y los cambios a realizar en el estado SYSTEM. También se puede escribir a estos registros usando la dirección y bus de datos que se usan con los registros que ya teníamos, siempre y cuando se active la señal de permiso de escritura de registro de sistema *wr_sys*.

Para poder seleccionar que salgan los registros de sistema por el bus *a*, *sys_a* debe valer '1' y *reg_op* debe valer "000". Al igual que cuando se ejecute una operación de banco de registros que use el bus *a*, *sys_a* debe ser igual a '0'.

Finalmente, para poder generar todas las señales pertinentes cuando se entra en el ciclo de sistema, tal y como se muestra en la figura 5, la comprobación que determina si el procesador está en el ciclo SYSTEM, debe de ser la primera de todas en cualquier señal que tenga que tomar un valor determinado durante este estado.

Si por algún casual, no se siguiera este esquema, el resultado puede ser impredecible. Esto sucede a causa de que aunque en el ciclo de SYSTEM no se debe ejecutar ninguna instrucción, el registro de instrucción siempre tendrá un valor, el cual puede ser interpretado por el procesador y causar cambios en él.

## 3.9. Etapa 7.2: Excepciones

Durante esta fase, se ha implementado la capacidad de detectar tanto excepciones lanzadas por el procesador, como por el código que se ejecuta. Básicamente, se han sentado las bases para poder añadir todas las excepciones pertinentes en las siguientes fases de desarrollo, además de empezar a tratar algunas excepciones como se describe en el manual del procesador.

Para poder gestionar las excepciones, se ha creado el módulo **exception_controller.** Este módulo tiene la función de activar la señal de excepción e informar al procesador que excepción se ha producido.

Para poder funcionar correctamente, este módulo tiene 4 buses de salida. *exception_id* tiene el valor de la excepción que se debe tratar, en caso de que haya alguna. *excep_UP_F* y *excep_UP*, son las señales que indican que el procesador tiene que pasar al ciclo SYSTEM. La primera es la encargada de la transición desde el estado *fetch* (F), mientras que la segunda es la encargada de la transición desde el estado de DEMW. Por último, la señal *stop_execution* le indica al procesador que debe bloquear las señales que permitan que se puedan hacer cambios en el estado de este.

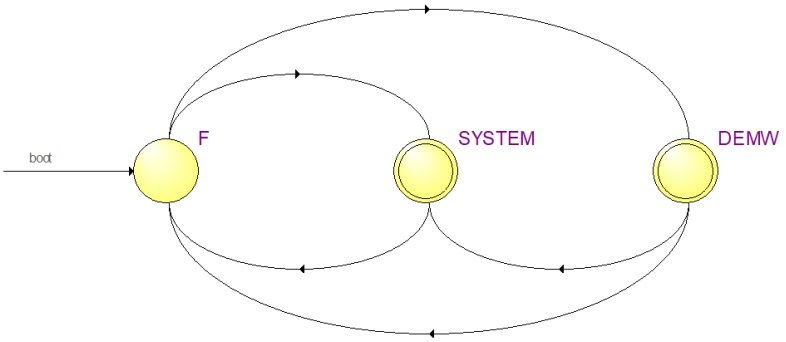
Para poder generar todas estas señales de salida, este módulo recoge un conjunto de señales de entrada que indican el estado del procesador y de los módulos que pueden generar excepciones.

A continuación se detalla de dónde salen las señales necesarias para poder determinar si se ha producido una de las excepciones que se han implementado en esta etapa:

· Excepción 0: Instrucción ilegal. **contro_l**: *illegal_inst*.
· Excepción 1: Alineación impar. **memory_controller**: *invalid_address*.
· Excepción 4: División por cero. **alu:** *div_zero*.
· Excepción 15: Interrupción. **controlador_interrupciones:** *intr*.

Cabe destacar que en la enumeración anterior, solo se explican las señales que informan de una posible excepción. Después es el **exception_controller** quien determina si lo que está ocurriendo es realmente una excepción dependiendo del estado en el que se encuentre el procesador.

Finalmente, en el módulo **multi_l,** se ha añadido la posibilidad de hacer transiciones del estado *fetch* (F) al de SYSTEM. Esto ha sido necesario dado que las excepciones que indican que el pc es erróneo, solo aparecen en el estado F, que es cuando se extraen las instrucciones de memoria.




## 3.10. Etapa 7.3: Modo privilegiado y Llamadas a Sistema

En esta etapa se añade la diferenciación entre modo usuario y modo sistema. Este cambio implica ligeras modificaciones en las operaciones de banco de registro, y la agregación de nuevas excepciones para limitar las acciones que puede hacer el usuario en el modo sistema. Además de una nueva instrucción para poder saltar del modo usuario al modo de sistema.

Para poder ejecutar la instrucción CALLS, se ha decidido usar el ciclo de DEMW para guardar en S4 el valor del registro codificado en *Ra*. Se ha tomado esta decisión dado que durante el ciclo de SYSTEM ya se está usando la entrada *d* para poder guardar en el registro S2 el pc de retorno de la RSG. En caso de querer hacer todas estas operaciones en el estado SYSTEM, se tendría que añadir una nueva entrada en el banco de registros.

A continuación se detalla de dónde salen las señales necesarias para poder determinar si se ha producido una de las excepciones que se han implementado en esta etapa:

. Excepción 0: Instrucción ilegal, *call* en modo sistema. **control_l:** *sys_call*.
**regfile**: *system_mode***.**
· Excepción 11: Acceso memoria protegida. **memory_controller:** *system_address*
· Excepción 13: Excepción de protección. **control_l:** *system_ins*.
· Excepción 14: Llamada al sistema. **control_l:** *sys_call*.

## 3.11. Etapa 8: TLB (*Translation Lookaside Buffer*)

Finalmente, en esta etapa se han añadido dos TLBs para hacer la traducción de direcciones lógicas a físicas. Estos dos módulos se encuentran en el archivo de **datapath**, dado que así no hay que cambiar ninguna señal externa a este, tal y como se muestra en la figura 6.

Para poder generar la dirección completa, el módulo **datapath** se encarga de unir la salida de cada TLB a los bits de offset de cada dirección. Las TLBs traducen direcciones de forma constante, y es **datapath** a través de la señal *ins_dad* que decide qué dirección es enviada al **MemoryController**.

Para esta etapa se ha creado un nuevo módulo **tlb**, para inicializar los valores tal y como se describe en el manual, este utiliza la señal de *boot* sincronizada con el flanco ascendente del reloj para poner todos sus registros a estos valores.

La traducción de direcciones se realiza de forma asíncrona, en cambio, la escritura de los registros de este módulo se hace de forma síncrona utilizando la señal *tlb_op*, que determina qué operación se tiene que efectuar.

La detección de excepciones, también se hace de forma síncrona, dado que como se ha explicado anteriormente, en el módulo **exception_controller** ya hace de filtro y detecta si realmente la TLB que genera la excepción está siendo usada. Además, que el módulo **multi** también filtra, dado que solo salta si coincide que está activa la señal de transición al estado SYSTEM desde el estado que determina cada señal.

A continuación se detalla de dónde salen las señales necesarias para poder determinar si se ha producido una de las excepciones que se han implementado en esta etapa. :

· Excepción 6: Miss en TLB de instrucciones. **tlb_ins:** *miss_tlb_I.*
· Excepción 7: Miss en TLB de datos. **tlb_dat:** *miss_tlb_D*.
**control_l**: *data_memory_acces*.
· Excepción 8: Página inválida al TLB de instrucciones. **tlb_ins:** *invalid_page_I*
· Excepción 9: Página inválida al TLB de datos. **tlb_dat:** *invalid_page_D*.
**control_l**: *data_memory_acces*.
· Excepción 10: Página protegida al TLB de instrucciones.
**memory_controller:** *system_address*. **control_l**: *data_memory_acces*.
· Excepción 12: Página de solo lectura. **tlb_dat:** *read_only_page_D*. **control_l:** *isST*.

# 4. Puntos destacables del procesador

## 4.1. Partes ventajosas de la implementación

La implementación del procesador presenta una serie de puntos ventajosos que son importantes de destacar.

En primer lugar, cabe destacar el módulo de excepciones por su estructura clara y sencilla, lo que facilita su comprensión y escalabilidad.

Además, cada módulo en la implementación cumple una única función clara y precisa. Por ejemplo, el código de lógica de control se encuentra encapsulado en el módulo **control_l**, evitando así la dispersión del código en diferentes archivos. Esta organización modular simplifica los cambios y mejoras en el procesador, ya que se pueden realizar modificaciones específicas en cada módulo sin afectar otros componentes.

Otra ventaja destacable es la facilidad de reemplazo de los módulos. Tal y como se acaba de explicar, cada uno tiene una función específica, lo que permite realizar mejoras o actualizaciones en el diseño sin tener que reescribir o modificar código repetido por todo el procesador. Esta flexibilidad facilita el desarrollo iterativo y permite adaptar el procesador a diferentes requisitos o futuras actualizaciones.

Además, se ha hecho uso de comentarios para facilitar el entendimiento del código, especialmente en los cambios realizados en las últimas etapas.

Un punto adicional a destacar es el módulo de control de memoria, el cual permite hacer escrituras en 4 ciclos. Esta característica es relevante, ya que podría permitir que se redujera a la mitad el tiempo de ciclo del procesador, siempre y cuando lo permitieran los demás módulos.

## 4.2. Partes mejorables de la implementación

Una vez acaba la implementación del procesador, se han identificado varios puntos que podrían mejorarse para lograr un diseño más sólido, eficiente y robusto.

En primer lugar, no se ha hecho un uso adecuado de constantes en la programación del procesador. La falta de uso de constantes ha llevado a una necesidad de agregar comentarios en el código para explicar su propósito, lo que dificulta la legibilidad y mantenibilidad del mismo. La incorporación de constantes adecuadas hubiese simplificado el código y lo hubiese hecho más claro y fácil de entender.

En segundo lugar, no se ha respetado la arquitectura típica o lógica de un procesador. Esto ha podido venir dado por la manera de implementar este. El hecho de que la implementación se haya realizado de manera progresiva, ha hecho que se hayan desarrollado módulos sin tener en cuenta los aspectos futuros del diseño. Esto ha resultado en un diseño fragmentado y potencialmente menos eficiente. Habría sido importante tener una visión clara de la arquitectura completa del procesador y planificar las implementaciones de los módulos de acuerdo con ese diseño global.

Otra cuestión a mejorar, es la conexión entre módulos. Se podrían haber usado buses entre etapas del procesador. Esta falta de uso de buses ha llevado a una estructura de cables compleja y poco clara. Los buses hubiesen proporcionado una forma organizada y eficiente de conectar los diferentes componentes del procesador, además de haber reducido la cantidad de cables necesarios y, por tanto, haber mejorado la legibilidad del código.

Por último, habría sido beneficioso realizar más juegos de prueba desde el principio del proyecto. Aunque se han realizado juegos de prueba, estos han sido generales de cada etapa. Hubiese sido mucho mejor probar cada módulo por separado de una forma exhaustiva. Esto hubiera facilitado la comprensión de cada módulo y hubiera ayudad a verificar el correcto funcionamiento del procesador y detectar posibles errores o fallos en las etapas iniciales.

En resumen, para mejorar la implementación de este procesador, se recomienda añadir constantes, restituir los módulos de forma que quede más claro que hace cada uno de estos y emplear menos señales y en agregación a esto, utilizar buses entre etapas y probar de manera unitaria cada módulo. Estas mejoras contribuirán a obtener un diseño más eficiente, legible y confiable.

# 5. Conclusión

La implementación de este procesador ha sido un proyecto desafiante que nos ha exigido dedicación y horas de trabajo para completar todas las etapas. A lo largo del proceso, nos hemos enfrentado a la complejidad inherente de diseñar y desarrollar un procesador desde cero sin tener prácticamente conocimientos de programación en VHDL.

Sin embargo, esta experiencia ha sido sumamente enriquecedora, ya que nos ha permitido adquirir conocimientos profundos sobre el diseño de protocolos, además de consolidar los conocimientos adquiridos durante las asignaturas donde se nos ha explicado, de forma teórica, las diferentes partes que conforman un procesador.

Hemos aprendido a manejar conceptos complejos y a aplicarlos en la práctica, lo cual nos brinda una sólida base de conocimiento y nos capacita para enfrentar desafíos similares en el futuro.

Aunque el diseño de un procesador sigue siendo una tarea compleja, ahora contamos con el conocimiento y la experiencia necesarios para abordar proyectos similares en el futuro. Y en caso de querer profundizar y ampliar el procesador hecho durante esta asignatura o otro con una ISA diferente, nos sentimos confiados en nuestra capacidad para aplicar los conceptos aprendidos y llevar a cabo diseños eficientes y robustos.

El hecho de enfrentarnos a este reto en equipo, aunque al principio fue un desafío, nos ha brindado la oportunidad de hacer que nos tengamos que adaptar a distintos tipos de programación y sobre todo, acostumbrarnos a tener que documentar todo lo que hacemos a cada instante para facilitar la comprensión de los demás integrantes del equipo.

En conclusión, el proyecto de implementación de este procesador ha sido un desafío gratificante que nos ha permitido adquirir conocimientos profundos y valiosos sobre el diseño de protocolos y las diferentes partes de un procesador. Aunque ha sido complejo, nos hemos superado a nosotros mismos y ahora nos sentimos preparados para llevar a la práctica todo lo aprendido.

# 6. Posibles ampliaciones del Proyecto

Una posible ampliación para el procesador, podría ser mejorar la velocidad del procesador mediante la segmentación en diversas etapas. Para poder realizar esta ampliación se tendría que hacer primero que el procesador fuera multiciclo para aprovechar y segmentar las etapas con mayor retraso. Esto permitiría que varias instrucciones se ejecutan simultáneamente y se superponen en su procesamiento. Esta ampliación tiene el potencial de acelerar significativamente la velocidad de ejecución de las instrucciones.

La implementación de un módulo de coma flotante también podría ser una forma de ampliar el procesador. El módulo de coma flotante permitiría realizar operaciones aritméticas más complejas y precisas con números de punto flotantes. Este tipo de módulo requeriría un nuevo banco de registros para los números de coma flotante y la creación de nuevas operaciones en la ALU.

Además, otra opción para ampliar el procesador sería cambiar el controlador de vía a modo gráfico. Esto implicaría cambiar el controlador de VGA y modificar el mapa de memoria porque esta ampliación requeriría una área mayor para controlar el contenido que se muestre por pantalla. Este tipo de ampliación sería especialmente relevante para aplicaciones que requieren una representación visual o interactiva, como un sistema operativo visual o poder mostrar imágenes por pantalla.

Otra posible ampliación, podría ser un gestor de archivos a través de la tarjeta SD, para así poder cargar programas a nuestro procesador a través de un sistema operativo, que también sería una ampliación bastante atractiva, tal y como explicaremos a continuación.

Por último, la forma más natural de ampliar el procesador sería la creación de un sistema operativo completo. Este sería capaz de administrar los recursos del sistema, como la memoria, los dispositivos de entrada/salida y la programación de tareas. La implementación de un sistema operativo permitirá ejecutar más de un proceso a través de cambios de contexto, además de poder gestionar las excepciones.

En resumen, las ampliaciones posibles sobre este procesador son muchas y muy variadas, pero estas son las que nos llaman más la atención y consideramos más prioritarias.

# 7. Bibliografía


- Departamento de arquitectura de computadores. (2013, 5 febrero). *PEC - Projecte d'Enginyeria de Computadors*. docencia.ac.upc.edu/FIB/grau/PEC. Recuperado 31 de mayo de 2023, de https://docencia.ac.upc.edu/FIB/grau/PEC/index.html

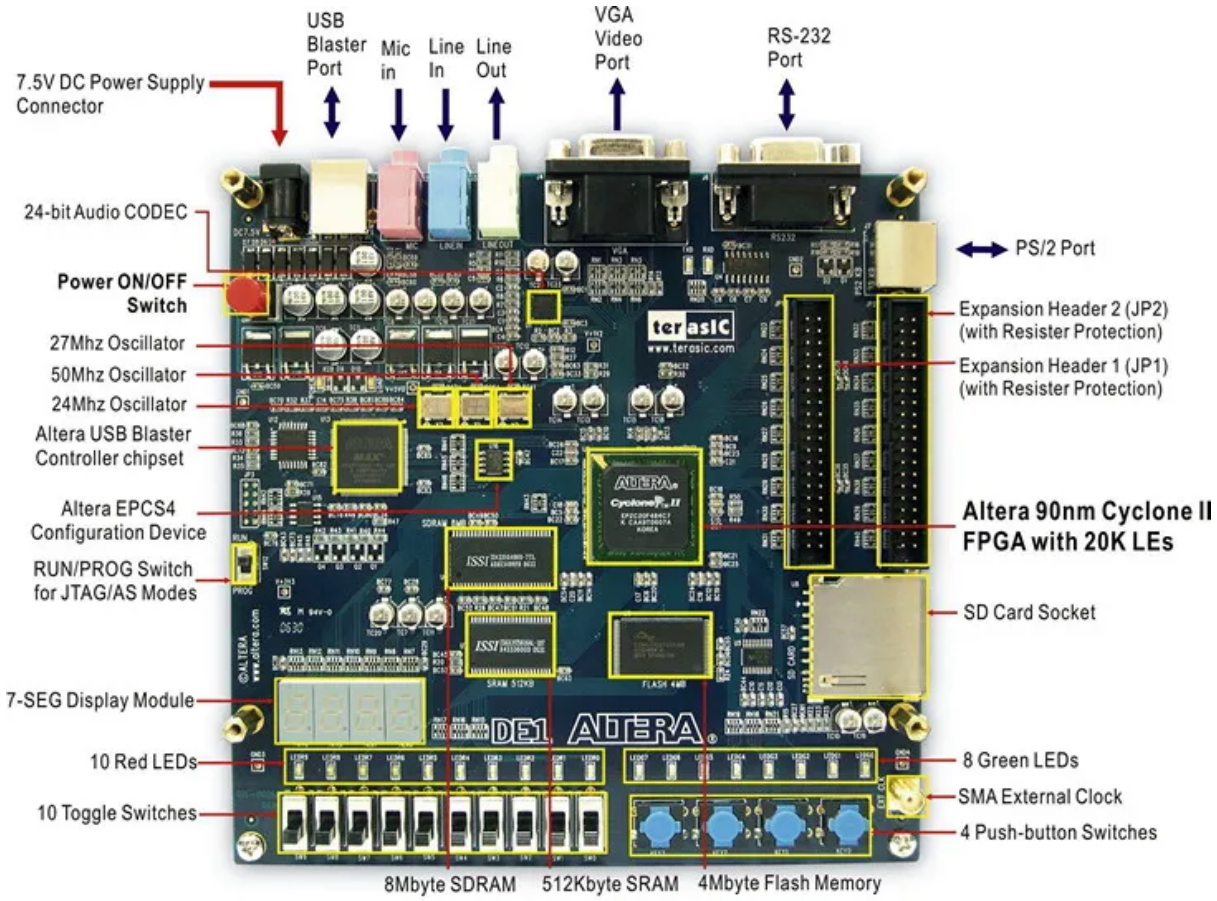
- Technologies, T. (s. f.). *Terasic - SoC Platform - Cyclone - DE1-SoC Board*. Copyright © 2001-2005. Recuperado 31 de mayo de 2023, de https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836

- ISSI. (2011b, diciembre). *Datasheet IS61LV25616AL*. issi.com. Recuperado 31 de mayo de 2023, de https://www.issi.com/ww/pdf/61lv25616al.pdf

- OpenCores. (s. f.). *OpenCores*. opencores.org. Recuperado 31 de mayo de 2023, de https://opencores.org/

# 8. Anexos

## Figura 1. Placa DE1 Terasic



# Figura 2. Codificación de las 55 instrucciones SISA

| 15 14 13 12 | 11 10 9 | 8 7 6 | 5 4 3 | 2 1 0 | | |
|---|---|---|---|---|---|---|
| 0 0 0 0 | Rd | Ra | f f f | Rb | Op. Lógicas y Aritméticas | AND, OR, XOR, NOT ADD, SUB, SHA, SHL |
| 0 0 0 1 | Rd | Ra | f f f | Rb | Comparación con y sin signo | CMPLT, CMPLE, -, CMPEQ CMPLTU, CMPLEU, -, - |
| 0 0 1 0 | Rd | Ra | n n n n n n | | Add inmediato | ADDI |
| 0 0 1 1 | Rd | Ra | n n n n n n | | Load | LD |
| 0 1 0 0 | Rb | Ra | n n n n n n | | Store | ST |
| 0 1 0 1 | Rd | 0 | n n n n n n n n | | Mover Inmediato | MOVI |
| | Ra Rd | 1 | | | | MOVHI |
| 0 1 1 0 | Rb | 0 | n n n n n n n n | | Salto condicional modo relativo al PC | BZ |
| | | 1 | | | | BNZ |
| 0 1 1 1 | Rd | 0 | n n n n n n n n | | Input | IN |
| | Rb | 1 | | | Output | OUT |
| 1 0 0 0 | Rd | Ra | f f f | Rb | Extensión aritmética | MUL, MULH, MULHU, - DIV, DIVU, -, - |
| 1 0 0 1 | Fd | Fa | f f f | Fb | Op/Cmp Float | ADDF, SUBF, MULF, DIVF CMPLT, CMPLEF, -, CMPEQF |
| 1 0 1 0 | Rb | Ra | 0 0 0 f f f | | Ruptura de secuencia modo registro | JZ, JNZ |
| | 0 0 0 | Ra | | | | -, JMP |
| | Rd | Ra | | | | JAL, -, - |
| | 0 0 0 | Ra | | | | CALLS |
| | | | f f f f f f | | Reservadas Fut. Ampl. | 42 códigos |
| 1 0 1 1 | Fd | Ra | n n n n n n | | Load Float | LDF |
| 1 1 0 0 | Fb | Ra | n n n n n n | | Store Float | STF |
| 1 1 0 1 | Rd | Ra | n n n n n n | | Load Byte (8 bits) | LDB |
| 1 1 1 0 | Rb | Ra | n n n n n n | | Store Byte (8 bits) | STB |
| 1 1 1 1 | | | 0 | | Reservadas Fut. Ampl. | 32 códigos |
| | 0 0 0 | 0 0 0 | 1 | f f f f f | Instrucciones especiales | EI, DI, -, - |
| | 0 0 0 | 0 0 0 | | | | RETI, -, -, - |
| | Rd | 0 0 0 | | | | GETIID, -, -, - |
| | Rd | Sa | | | | RDS, -, -, - |
| | Sd | Ra | | | | WRS, -, -, - |
| | Rb | Ra | | | | WRPI, WRVI, WRPD, WRVD |
| | 0 0 0 | Ra | | | | FLUSH, -, -, - |
| | 1 1 1 | 1 1 1 | | | | -, -, -, HALT |

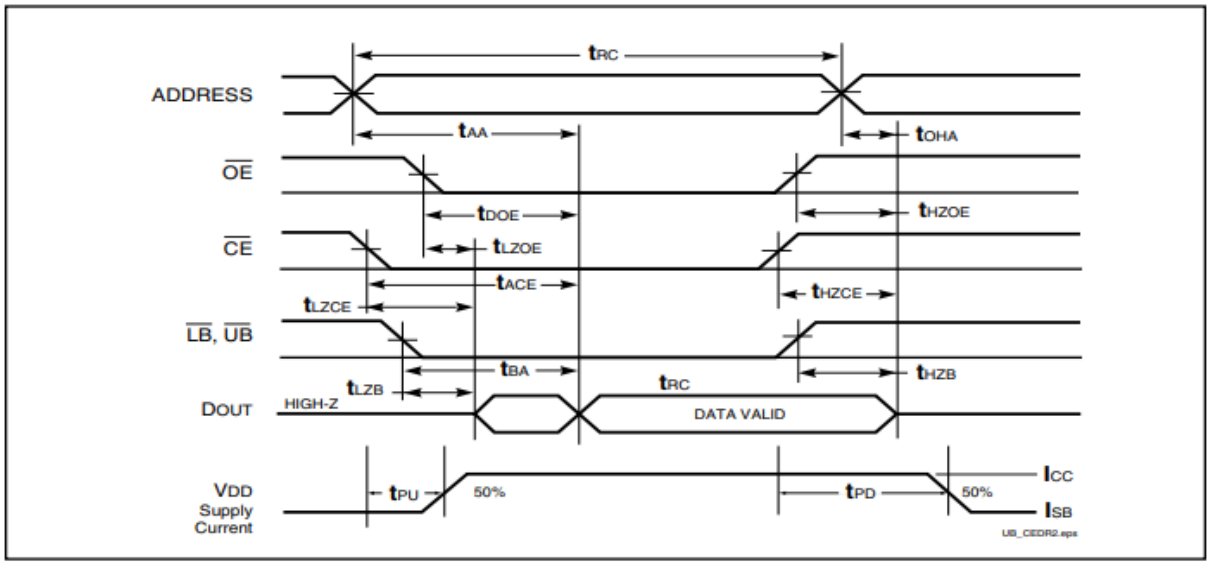
# Figura 3. Protocolo de lectura controlador de memoria

## READ CYCLE SWITCHING CHARACTERISTICS[1] (Over Operating Range)

| Symbol | Parameter | -10 Min. | -10 Max. | -12 Min. | -12 Max. | Unit |
|---|---|---|---|---|---|---|
| $t_{RC}$ | Read Cycle Time | 10 | — | 12 | — | ns |
| $t_{AA}$ | Address Access Time | — | 10 | — | 12 | ns |
| $t_{OHA}$ | Output Hold Time | 2 | — | 2 | — | ns |
| $t_{ACE}$ | $\overline{CE}$ Access Time | — | 10 | — | 12 | ns |
| $t_{DOE}$ | $\overline{OE}$ Access Time | — | 4 | — | 5 | ns |
| $t_{HZOE}$[2] | $\overline{OE}$ to High-Z Output | — | 4 | — | 5 | ns |
| $t_{LZOE}$[2] | $\overline{OE}$ to Low-Z Output | 0 | — | 0 | — | ns |
| $t_{HZCE}$[2] | $\overline{CE}$ to High-Z Output | 0 | 4 | 0 | 6 | ns |
| $t_{LZCE}$[2] | $\overline{CE}$ to Low-Z Output | 3 | — | 3 | — | ns |
| $t_{BA}$ | $\overline{LB}$, $\overline{UB}$ Access Time | — | 4 | — | 5 | ns |
| $t_{HZB}$[2] | $\overline{LB}$, $\overline{UB}$ to High-Z Output | 0 | 3 | 0 | 4 | ns |
| $t_{LZB}$[2] | $\overline{LB}$, $\overline{UB}$ to Low-Z Output | 0 | — | 0 | — | ns |
| $t_{PU}$ | Power Up Time | 0 | — | 0 | — | ns |
| $t_{PD}$ | Power Down Time | — | 10 | — | 12 | ns |

**Notes:**
1. Test conditions assume signal transition times of 3 ns or less, timing reference levels of 1.5V, input pulse levels of 0V to 3.0V and output loading specified in Figure 1.
2. Tested with the load in Figure 2. Transition is measured ±500 mV from steady-state voltage.

## READ CYCLE NO. 2[1,3]




**Notes:**
1. $\overline{WE}$ is HIGH for a Read Cycle.
2. The device is continuously selected. $\overline{OE}$, $\overline{CE}$, $\overline{UB}$, or $\overline{LB}$ = $V_{IL}$.
3. Address is valid prior to or coincident with $\overline{CE}$ LOW transition.

Figura 4. Protocolo de lectura controlador de memoria

**WRITE CYCLE SWITCHING CHARACTERISTICS**[1,3] (Over Operating Range)

| Symbol | Parameter | -10 Min. | -10 Max. | -12 Min. | -12 Max. | Unit |
|---|---|---|---|---|---|---|
| $t_{WC}$ | Write Cycle Time | 10 | — | 12 | — | ns |
| $t_{SCE}$ | $\overline{CE}$ to Write End | 8 | — | 8 | — | ns |
| $t_{AW}$ | Address Setup Time to Write End | 8 | — | 8 | — | ns |
| $t_{HA}$ | Address Hold from Write End | 0 | — | 0 | — | ns |
| $t_{SA}$ | Address Setup Time | 0 | — | 0 | — | ns |
| $t_{PWB}$ | $\overline{LB}$, $\overline{UB}$ Valid to End of Write | 8 | — | 8 | — | ns |
| $t_{PWE1}$ | $\overline{WE}$ Pulse Width | 8 | — | 8 | — | ns |
| $t_{PWE2}$ | $\overline{WE}$ Pulse Width ($\overline{OE}$ = LOW) | 10 | — | 12 | — | ns |
| $t_{SD}$ | Data Setup to Write End | 6 | — | 6 | — | ns |
| $t_{HD}$ | Data Hold from Write End | 0 | — | 0 | — | ns |
| $t_{HZWE}$[2] | $\overline{WE}$ LOW to High-Z Output | — | 5 | — | 6 | ns |
| $t_{LZWE}$[2] | $\overline{WE}$ HIGH to Low-Z Output | 2 | — | 2 | — | ns |

**Notes:**
1. Test conditions assume signal transition times of 3 ns or less, timing reference levels of 1.5V, input pulse levels of 0V to 3.0V and output loading specified in Figure 1.
2. Tested with the load in Figure 2. Transition is measured ±500 mV from steady-state voltage. Not 100% tested.
3. The internal write time is defined by the overlap of $\overline{CE}$ LOW and $\overline{UB}$ or $\overline{LB}$ and $\overline{WE}$ LOW. All signals must be in valid states to initiate a Write, but any one can go inactive to terminate the Write. The Data Input Setup and Hold timing are referenced to the rising or falling edge of the signal that terminates the write.

**WRITE CYCLE NO. 3** ($\overline{WE}$ Controlled. $\overline{OE}$ is LOW During Write Cycle) [1]

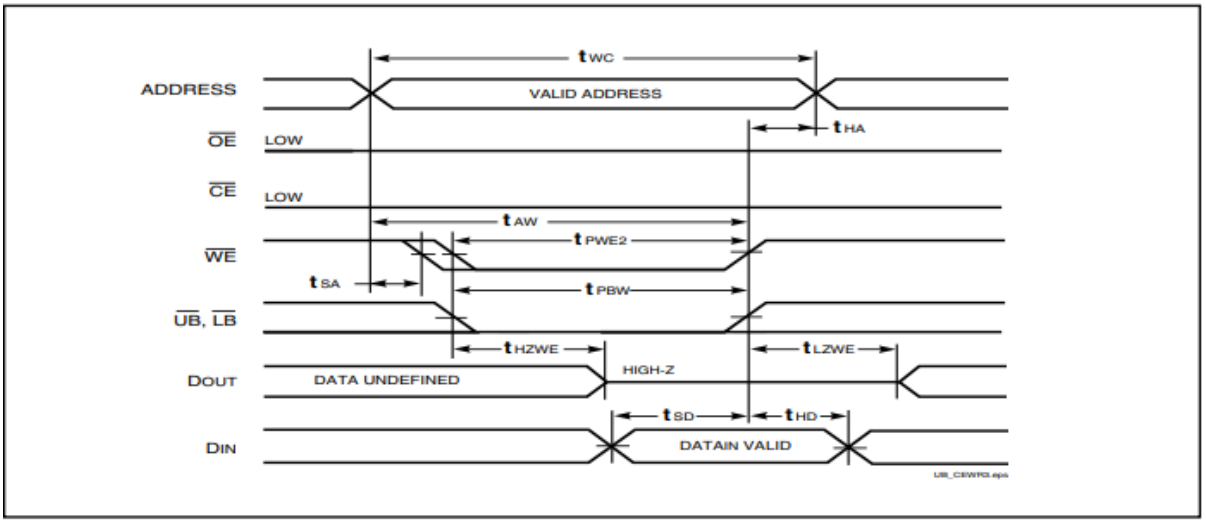



Figura 5. Cambios en las señales durante el estado SYSTEM

```
-- Entrada del banco de regitros: 001 : memoria, 000: alu, 010: pc+2
  in_d <="100" when system_act = '1' else
         "011" when ir(15 DOWNTO 12) = "1111" and ir(5 DOWNTO 0) = "101000" else --GETIID
         "001" when ir(15 DOWNTO 12) = "0011" else --Load
         "001" when ir(15 DOWNTO 12) = "1101" else --Load Byte
         "010" when ir(15 DOWNTO 12) = "1010" and ir(2 DOWNTO 0) = "100" else -- JAL
         "011" when ir(15 DOWNTO 12) = "0111" else -- IN
         "000" ;
```

Figura 6. TLBs dentro del camino de datos