Instituto Tecnológico y de Estudios Superiores de Occidente

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación del 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática

Maestría en Diseño Electrónico

REPORTE DE FORMACIÓN COMPLEMENTARIA EN ÁREA DE CONCENTRACIÓN DE SISTEMAS DIGITALES

TRABAJO RECEPCIONAL que para obtener el **GRADO** de
MAESTRO EN DISEÑO ELECTRÓNICO

Presenta: **FRANCISCO JAVIER DELGADILLO CASAS**

Asesor: **OMAR HUMBERTO LONGORIA GÁNDARA**

Tlaquepaque, Jalisco. 26 de junio de 2019.

RESUMEN DE CONTENIDO

El presente trabajo contiene proyectos enfocados en el diseño de sistemas digitales utilizando principalmente herramientas utilizadas en la industria, tales como: OVM, UVM, SystemVerilog, ModelSim, QuestaSim y FPGAs. Los proyectos presentados a continuación se enfocan en arquitectura de computadoras modernas. Se desarrolló un sistema RISC, basado en arquitectura MIPS segmentado (*pipeline*), predictor de saltos (*Jump Predictor Unit*), detector de errores (*Hazzard Unit*), unidad de acarreo (*Fordward Unit*) y con un sistema de memoria temporal (*cache*). También se implementó el protocolo MESI para mantener la coherencia de la memoria cache con la memoria principal (*RAM*).

Además cada proyecto se elaboró en base a estándares usados en la industria. Incluyendo el orden en el proceso de diseño de un circuito integrado: la generación de especificaciones de diseño (*Hardware Architecture Specification HAS*), creación de un plan de pruebas (*test plan*), creación de un ambiente de verificación formal en System Verilog (*test bench*), creación de las pruebas, ejecución de las pruebas y por último detección de errores en el diseño y corrección de los mismos.

Contenido

1. Introducción.

Los sistemas digitales son la base de la tecnología moderna, la mayoría de los dispositivos electrónicos actuales son digitales. Se estima que para el 2020 habrá 3 dispositivos conectados a internet por cada ser humano, dato que significa que se alcanzarán los 25 mil millones de dispositivos en internet para 2020 (Charlton, 2014). Esta cifra muestra la cantidad de dispositivos digitales en operación sólo para las aplicaciones que requieran estar conectadas a internet. Las cifras rebasan por mucho la cantidad de dispositivos en aplicaciones embebidas, que van desde aplicaciones aeronáuticas hasta dispositivos personales como: audífonos, relojes inteligentes o bocinas inalámbricas. Con esta cantidad de dispositivos, la generación de información crecerá de forma exponencial, por lo que dicho volumen de información deberá procesarse a velocidades mayores y con dispositivos digitales más especializados.

Estas son algunas de las razones por las cuáles el área de diseño de sistemas digitales es cada vez más relevante en la era de la información. Por lo tanto, el enfoque del presente trabajo y del campo de estudio profesional que se presenta, es el diseño de sistemas digitales, tomando en cuenta las áreas de concentración que se acreditaron en la maestría, tanto en diseño de sistemas digitales como en diseño de circuitos integrados digitales. A continuación, se listan las asignaturas cursadas y aprobadas relacionadas con el diseño de sistemas digitales:

DISEÑO DE SISTEMAS DIGITALES	SP - AREA DE CONCENTRACION EN CIRCUITOS INTEGRADOS DIGITALES
DISEÑO DE MICROPROCESADORES	SP - AREA DE CONCENTRACION EN SISTEMAS DIGITALES
VERIFICACION DE SISTEMAS DIGITALES	SP - AREA DE CONCENTRACION EN CIRCUITOS INTEGRADOS DIGITALES
ARQUITECTURA DE MICROPROCESADORES	SP - AREA DE CONCENTRACION EN SISTEMAS DIGITALES
DISEÑO E IMPLEMENTACION DE SISTEMAS OPERATIVOS	SP - AREA DE CONCENTRACION EN SISTEMAS DIGITALES

Tabla 1-1 Asignaturas cursadas y aprobadas relativas al diseño de sistemas digitales.

Como parte de las asignaturas cursadas, se realizaron proyectos enfocados al desarrollo y verificación de la arquitectura de microprocesadores. Se procuró que en cada proyecto se hiciera uso de herramientas de diseño actualmente empleadas en la industria, tales como: SystemVerilog, las metodologías estándar de validación OVM y UVM, tarjetas de desarrollo basadas en FPGAs de última generación y los simuladores ModelSim y QuestaSim. También se utilizó la denominación estándar de las distintas etapas para el desarrollo de un proyecto de ingeniería, tales como: definición de especificación, desarrollo del RTL model, definición del plan de pruebas, desarrollo de la cama de pruebas, la realización de las pruebas en sí y la depuración del diseño.

2. Resumen de los proyectos realizados.

Los proyectos consignados en el presente reporte corresponden al ciclo completo de diseño y validación de un sistema digital en la etapa conocida como pre-silicio (antes de su manufactura). Se tomó como caso de estudio el diseño de un microprocesador segmentado basado en el conjunto de instrucciones del microprocesador MIPS, con una arquitectura Harvard y un sistema de memoria caché basado en MESI.

- Desarrollo de un microprocesador MIPS segmentado con soporte de recursividad: Implementado en el curso de DISEÑO DE MICROPROCESADORES impartido por el Dr. José Luis Pizano Escalante. En él, se revisan conceptos avanzados de arquitectura de microprocesadores, así como su implementación y depuración. Además del MIPS, durante el desarrollo del proyecto se generó un programa recursivo en lenguaje ensamblador, se creó una unidad de predicción de saltos y se síntetizó el RTL para ser probado en una plataforma de desarrollo basada en un FPGA de la familia Cyclone IV de Altera. Estas actividades muestran cómo funcionan los procesadores actuales, además de su proceso de diseño, incluidas las etapas de simulación y depuración.
- Verificación formal de la Dispatch Unit de un MIPS superescalar. Desarrollado en el curso de VERIFICACION DE SISTEMAS DIGITALES, impartido por el Mtro. Carlos Vázquez Tello. En él, se realizó la implementación de un ambiente de verificación basado en SystemVerilog para validar la *Dispatch Unit* de un MIPS superescalar (con múltiples unidades de ejecución). Además del ambiente de pruebas, se revisó la arquitectura de un procesador superescalar con varias unidades de procesamiento, se desarrolló el plan de

pruebas para la verificación formal del sistema y se introdujeron las metodologías de verificación OVM y UVM. La importancia de estas actividades es su estrecha relación con el proceso de verificación formal que actualmente se sigue en la industria para el diseño de sistemas digitales.

- Implementación de un sistema con memoria Cache basado en MESI. Desarrollado en el curso de ARQUITECTURA DE MICROPROCESADORES impartido por el Dr. Héctor R. Sucar Sucar. En él, se implementó un sistema de cómputo de caches con SystemVerilog, el cual consta de un controlador de memoria principal, un CPU y un componente periférico. Además del sistema, se crearon: una especificación tipo HAS (*Hardware Architecture Specification*, por sus siglas en inglés), una cama de pruebas y se implementó el protocolo MESI tanto en el procesador como en el controlador de memoria. Todos los elementos empleados en el diseño que se reporta se utilizan actualmente en el diseño y fabricación de los microprocesadores que cuentan con múltiples nucleos de ejecución y que requieren acceso a memoria compartida.

2.1. Desarrollo de un microprocesador MIPS segmentado con soporte de recursividad.

2.1.1 Introducción.

En la actualidad, los procesadores con arquitectura RISC (Reduced Instruction Set Computer), como es el caso de procesadores MIPS y ARM, son utilizados en la mayoría de los dispositivos móviles, tales como: tabletas, teléfonos inteligentes y relojes inteligentes, entre otros. El estudio de los procesadores con arquitectura RISC, como es el caso del MIPS, es fundamental para el diseño, validación y fabricación de dichos sistemas.

2.1.2 Antecedentes.

En el año de 1981, un equipo de trabajo en la universidad de Stanford comenzó el trabajo de lo que hoy se conoce como el primer procesador MIPS. Este procesador fue concebido desde

el principio como un sistema con un pipeline de ejecución (segmentación mediante registros), que le permitía alcanzar un mayor desempeño (*throughput* – cantidad de instrucciones ejecutadas por unidad de tiempo), sin embargo, existe una versión del microprocesador MIPS que ejecuta una instrucción por cada ciclo de reloj. En el presente reporte se abordan ambas arquitecturas. En la primera parte se muestra una arquitectura capaz de ejecutar una instrucción por ciclo de reloj y en la segunda parte se muestra el diseño de una versión que incluye un pipeline de ejecución para mejorar el rendimiento del microprocesador.

2.1.3 Solución desarrollada.

Se implementó un microprocesador MIPS segmentado que soporta las siguientes microinstrucciones: ADD, ADDI, SUB, OR, ORI, AND, ANDI, LUI, NOR, SLL, SRL, LW, SW, BEQ, BNE, J, JAL y JR. El diseño del pipeline de ejecución se hizo diferenciando 5 etapas principales: *Instruction Fetch* (IF), *Instruction Decode* (ID), *Instruction Execution* (EX*), Memory Access* (MEM) y *Write-Back* (WB).

La solución desarrollada tomó como base el diseño presentado en el libro *COMPUTER ARCHITECTURE: A Quantitative Approach* de David A. Patterson y John L. Hennessy, el cual contiene un diseño simple del MIPS. A dicho diseño se agregaron las unidades de control, de avance (*Fordward Unit*) y de predicción de saltos como se muestra en la Figura 2-1. En el diseño realizado, se aligeró parte de la complejidad requerida de lado del compilador al incluir una *Hazard Unit* e introducir instrucciones de NOP cuando son requeridas. La implementación se realizó en Verilog para su posterior simulación en ModelSim y síntesis en Quartus II, además de ser desarrollada a partir del diagrama presentado y sin ningún tipo de implementación base. Es importante mencionar que la arquitectura RISC empleada para el conjunto de instrucciones del MIPS puede extrapolarse para implementar el conjunto de instrucciones correspondiente a un microprocesador ARM y otros actualmente en uso en los dispositivos electrónicos comeciales.

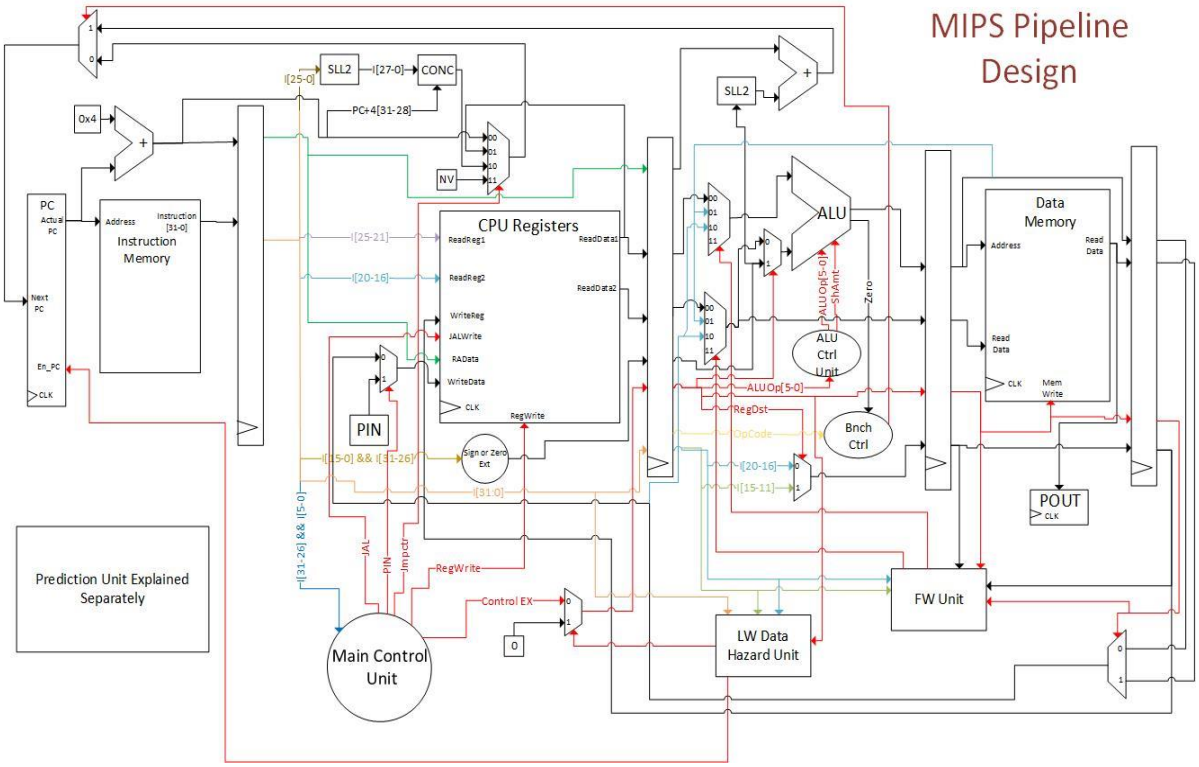


Figura 2-1 Diseño general de MIPS segmentado con *Forward Unit*, *Hazard Unit* y *Branch Prediction Unit*

2.1.4 Análisis de resultados.

Durante el diseño del sistema se generaron tres prototipos, los cuales fueron probados exitosamente tanto en simulación como en el FPGA. El primero contenía todas las instrucciones que soporta el procesador diseñado, el segundo hacía uso intensivo de la *Forward Unit* y de la *Control Hazard Unit*, el tercero proporcionaba el soporte de recursividad necesario para resolver problemas de cálculo complejos (para su prueba se empleó un programa en lenguaje ensamblador que resolvía el problema de las torres de Hanói). Se implementaron de forma correcta todas las funcionalidades descritas en los requerimientos, lo que permitió simular y sintetizar programas en lenguaje ensamblador que hacían uso de recursividad.

2.1.5 Conclusiones.

Dado que el proyecto se realizó de forma individual, todas las actividades y decisiones de diseño fueron tomadas por el ingeniero a cargo, lo cual contribuyó de forma significativa a la comprensión de las distintas etapas del ciclo de diseño de un producto. También se introdujeron algunas actividades simples relacionadas con la verificación del sistema. Y se desarrollaron programas de pruebas en lenguaje ensamblador que contribuyeron a una comprensión más detalladas y profunda de la arquitectura que emplean los microprocesadores comerciales.

Las pruebas se ejecutaron tanto en simulación, empleando el ambiente ModelSim, como directamente en un FPGA. Se obtuvieron resultados positivos de los tres prototipos después de depurar los errores encontrados en el diseño. También se elaboraron baterias básicas de prueba con el diseño implementado en un FPGA (emulación) con salidas y entradas limitadas y en simulación con visibilidad completa de las señales del diseño.

2.2. Verificación formal de la *Dispatch Unit* de un MIPS superescalar.

2.2.1 Introducción.

La creciente complejidad de los diseños digitales hacen necesaria su verificación formal en la etapa de pre-silicio, para ello se emplean diversas técnicas que crean baterías de pruebas (*testbenches*) modulares y reutilizables. Para este trabajo se emplean herramientas tales como SystemVerilog y las metodologías de verificación OVM y UVM. Verificar los diseños digitales permite cumplir con tres objetivos muy importantes: fortalecer el ciclo de diseño al disponer de información sobre el comportamiento del sistema bajo diversos tipos de condiciones estresantes, garantizar a los clientes, dentro de límites razonables, que los sistemas que se le ofrecen no fallarán una vez que sean puestos en operación bajo las condiciones para las cuáles fueron diseñados y ahorrar a las empresas millones de dólares debido a procesos tardíos de rediseño por errores que pudieron ser detectados con facilidad en etapas tempranas.

2.2.2 Antecedentes.

Es común entre las empresas del sector de los semiconductores, que la liberación de nuevas versiones de sus sistemas digitales conplejos presenten retardos de entre tres y seis meses, lo que representa costos colaterales de millones de dólares. Evitar este tipo de situaciones es un gran incentivo para que las compañías realicen, en etapas tempranas de diseño, procesos de validación de sus sistemas a fin de garantizar su correcta operación. Es importante señalar que los esfuerzos de validación deben ser conducidos por equipos de trabajo altamente especializados y que la incorporación de herramientas tales como OVM y UVM, aseguran que la verificación funcional se realiza de forma sistemática y exhaustiva.

2.2.3 Solución desarrollada.

El proyecto que se presenta consiste en la validación modular de un microprocesador MIPS superescalar. Se eligió la *Dispatch Unit* como módulo para ser verificado. A continuación, se describe de forma general la operación de este módulo del MIPS superescalar:

1. Se lee la instrucción entregada por el módulo encargado de realizar el proceso de fetch.
2. Se decodifica la instrucción.
3. Se calculan las direcciones de salto para cierto tipo de instrucciones que así lo requieren.
4. Se actualiza el *Register File*, y
5. Se actualiza el *Register Status Table*.

La primera etapa del proceso de verificación del diseño es la creación de un plan de pruebas (*Test Plan*) en el cual se describen el tipo y la cantidad de pruebas que se le realizarán al diseño (*device under test*). Un elemento clave del plan de pruebas es la descripción de todas las características del sistema que serán cubiertas.

En el caso específico de la *Dispatch Unit* del MIPS superescalar, aquí presentado, se definió un plan de pruebas basado en la funcionalidad del sistema y se creó una batería de pruebas que

cumpliera con lo señalado en el plan. En la creación de la batería de pruebas se emplearon conceptos de la metodología OVM tales como la creación y uso de *drivers*, monitores, *checkers* y *mailboxes*. La figura 2-2 muestra un esquema general de la batería de prueba que se desarrolló para estresar el sistema y validar su funcionamiento.

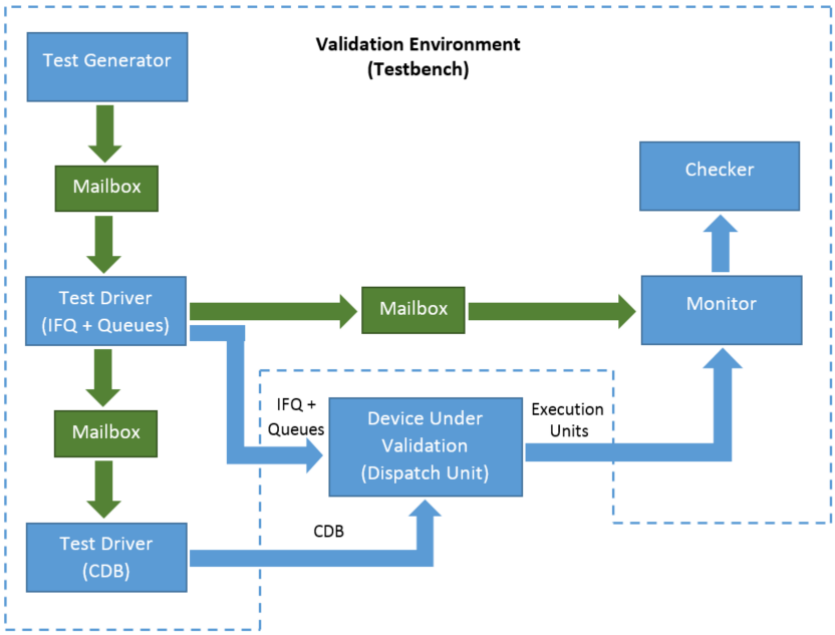


Figura 2-2 Esquema de la batería de pruebas desarrollada para la *Dispatch Unit* de un MIPS superescalar.

2.2.4 Análisis de resultados.

Se diseñó e implementó una batería de pruebas adecuada para estresar la funcionalidad completa de la *Dispatch Unit*. Se implementó con System Verilog y se ejecutó empleando el simulador QuestaSim. Se encontraron diversos errores, varios de ellos graves si se tratará de un diseño comercial:

- Las instrucciones para realizar multiplicaciones y divisiones (MULT y DIV) no se ejecutan correctamente, debido a que omiten los 32 bits más significativos del resultado.
- El código de operación de la instrucción para realizar una multiplicación no corresponde con el señalado por la especificación del microprocesador. En la

Dispatch Unit validada la instrucción MULT tiene un código de operación 0x19, pero la especificación señala que debe ser 0x18.

- Existe una gran vulnerabilidad en el diseño, debido a que no se limita el acceso a registros importantes como el Stack Pointer.
- La instrucción LUI no es soportada.
- La instrucción Branch no coloca en alto la señal "dispatch_en_integer", la cual es muy importante en el flujo de control del microprocesador.
- La instrucción SLTIU debe realizar sobre el dato involucrado una operación de "*Sign Extended Immediate*" y en su lugar realiza una operación de "*Zero Extended Immediate*".
- La ejecución del módulo CDB continúa a pesar de que los "*tags*" disponibles han sido asignados por completo.

Estos errores encontrados son un buen ejemplo del tipo de errores que pueden encontrarse en diseños de mayor complejidad o aquellos que tienen la intención de ser fabricados. Es una práctica común contar con un documento de referencia para la realización de la verificación, dicho documento es normalmente una especificación del funcionamiento del sistema llamada HAS (*Hardware Architecture Specification*). Una vez realizadas las pruebas señaladas en la batería y obtenidos sus resultados, se proporciona todo el detalle de los resultados obtenidos al equipo de diseño para que se hagan cargo de documentarlos y corregirlos.

2.2.5 Conclusiones.

El aporte personal a este proyecto se enfoca en el diseño del plan de pruebas (*Test Plan*) y el desarrollo de la batería de pruebas (*testbench*), que incluyó la codificación de los módulos *driver*, *monitor* y el *checker*. Se realizaron las pruebas en equipos de dos personas para dividir el trabajo equitativamente. Al encontrarse los errores de diseño se realizó, como en la industria, un triaje de pruebas y una verificación lógica de diseño para comprobar errores de RTL reales antes de generar el reporte correspondiente. Todo el trabajo se enfocó en construir una experiencia de cómo se realizan las verificaciones de diseños de sistemas digitales reales en la industria; además

se realizó todo el ciclo de validación, desde el diseño y aprobación del plan de pruebas, la creación de la batería de pruebas, la ejecución de las pruebas en lo individual y la documentación y reporte de los hallazgos hechos.

Como un elemento adicional de la betería de pruebas, se incorporaron algunos *mailboxes* que son un equivalente de los llamados TLMs en las metodologías estándar de validación tales como OVM y UVM. Estos módulos comunican todos los procesos entre sí, dándole a la batería de pruebas la inteligencia necesaria para sincronizar las distintas etapas de las pruebas.

2.3. Implementación de un sistema con memoria inmediata (*cache*) basado en MESI.

2.3.1 Introducción.

Los nuevos microprocesadores comerciales se caracterizan por el incremento en el número de núcleos que incorporan y sus altas frecuencias de operación; sin embargo, un aspecto igualmente importante en el desempeño general de un equipo de cómputo es su sistema de manejo de memoria. Los actuales sistemas de manejo de memoria han incrementado su complejidad, respecto de aquellos usados a principios de la década. En los últimos años, se desarrollaron diversos protocolos de coherencia de memoria inmediata (*cache*) y éstos pueden agruparse en dos categorías: aquellos basados en la invalidez de datos y los basados en la actualización de datos.

2.3.2 Antecedentes.

Para el desarrollo de este proyecto, se tomó como punto de partida el trabajo realizado en dos asignaturas previas (diseño de microprocesadores y validación de sistemas digitales), que consistieron en el diseño e implementación de un MIPS segmentado y la validación de la *Dispatch Unit* de un MIPS superescalar. Se partió de un sistema de memoria accedido por un sólo núcleo de procesamiento, pero rápidamente se demostró que esta aproximación no era realista si se consideraban los actuales diseños comerciales. Los actuales diseños comerciales cuentan con una memoria compartida por varios núcleos de procesamiento, además de varios componentes periféricos que también tienen acceso a la memoria del sistema. Debido a esta potencial fuente de

conflictos e inconsistencias en la información, se necesita implementar un protocolo de coherencia de memoria para siempre tener acceso a los datos actualizados y ejecutar los programas de manera correcta.

2.3.3 Solución desarrollada.

Para este proyecto se seleccionó el protocolo MESI, a fin de mantener la coherencia necesaria para un sistema de básico de procesamiento. La figura 2-3 muestra un sistema básico que cuenta con lo necesario para implementar el protocolo MESI de coherencia de memoria. El sistema consta de una Unidad Central de Procesamiento, su respectiva memoria inmediata (*cache*) conectada al bus del sistema, un periférico que se conecta al sistema mediante una interfaz tipo serial, un controlador de memoria que hace las veces de un árbitro que da acceso a la memoria y, por supuesto, la memoria principal del sistema. Cabe resaltar que la RAM tiene 4 páginas de 256 líneas con palabras de 32 bits y que la memoria *cache* en el microprocesador consta de 1 página de 256 líneas con palabras de 32 bits.

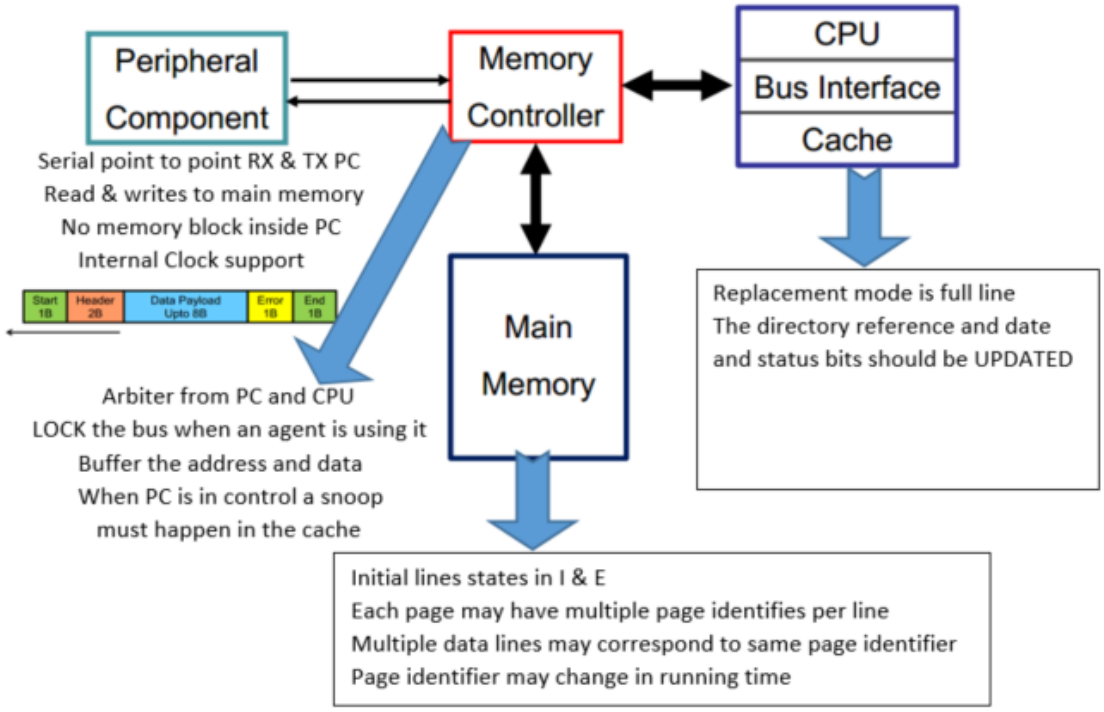


Figura 2-3 Sistema básico para implementar un protocolo de coherencia de memoria basado en MESI. Nótese que el CPU y un periférico comparten el acceso a la RAM.

Las direcciones de memoria son de 24 bits, con lo cual es posible direccionar hasta 64 MB (téngase en cuenta que las líneas de memoria son de 32 bits). El componente periférico puede acceder a la memoria y hacer lecturas y escrituras, sin embargo, estas deben apegarse al protocolo de coherencia, es decir, mantener un acceso de sólo lectura para el dato más actualizado. El CPU soporta la lectura y escritura de una copia de memoria en su propia *cache*, en dicha memoria inmediata se pueden tener líneas de diferentes páginas de RAM, facilitando entonces los aciertos en *cache*. Para ambos componentes que acceden a la memoria se definen flujos de operación para lecturas, escrituras, escrituras con eventos de acierto y fallo (*hit* y *miss*), lecturas con eventos de acierto y fallo (*hit* y *miss*), además de un protocolo espía (*snoop*) para invalidar memoria no actualizada.

2.3.4 Análisis de resultados.

Posterior a la implementación del sistema de coherehcia de memoria ya descrito, se generaron un plan (*Test Plan*) y una batería de pruebas (*Testbench*) con la finalidad de encontrar posibles fallos en el sistema y corregirlos. Como parte del plan de pruebas se generaron los siguientes escenarios:

- Lectura por parte del CPU con un evento de fallo en la *cache* sin transacción por parte del componente periférico.
- Lectura del CPU con un evento de acierto en la cache (con dato limpio o modificado) a la vez que se genera una transacción del componente periférico.
- Transacción de escritura del componente periférico hacia la RAM con un evento de acierto o fallo.
- Transacción de lectura del componente periférico a la RAM con un evento de acierto o fallo.
- Transacción de escritura del componente periférico a la RAM con un evento de acierto y el dato modificado.
- Transacción de lectura del cmponente periférico a la RAM con un evento de acierto y el dato modificado.

- Error en la transacción desde el componente periférico y reprocesamiento de la misma.
- Revisión de coherencia de datos entre la RAM y el componente periférico.

Al ejecutar el plan de pruebas se encontraron varios errores en la implementación, los cuales se detallan en el Apéndice C; dichos errores se corrigieron.

2.3.5 Conclusiones.

En este proyecto, se revisaron los protocolos de coherencia de momoeria utilizados actualmente en la industria, se eligió uno y se implementó utilizando un sistema mínimo con la capacidad de generar diversos escenarios de prueba.

Posteriormente, se validó el sistema, se encontraron errores de implementación y éstos fueron ciorregidos. Como parte de la validación del sistema, se creó el plan de pruebas, la batería de pruebas (los distintos módulos que la conforman) y la documentación. Se ejecutó el plan de pruebas planeado, se hallaron errores de implementación y se corrigieron dichos errores en el sistema, completando con ello el ciclo de vida del proyecto en fases reducidas pero similares a las ocurridas en la industria. El aporte personal se enfocó principalmente en el desarrollo de las pruebas, la depuración de errores y la arquitectura del sistema en general, además de escribir parte importante de la documentación.

3. Conclusiones.

El área de desarrollo profesional especializado que se eligió fue: diseño y verificación de sistemas digitales, con un enfoque particular en el ciclo de vida que los proyectos siguen en la vida real. Desde la definición de especificaciones de funcionamiento hasta el desarrollo de pruebas y depuración de errores encontrados en la implementación. Para el desarrollo de los proyectos, se procuró hacer uso de métodos de diseño y validación utilizados actualmente en la industria, tales como las como metodologías de validación OVM y UVM, cuyo uso es estándar en la industria de los semiconductores para la verificación de ASICs (*Application Specific Integrated Circuits*), microprocesadores de propósito general, microcontroladores (empleados en sistemas embebidos) y sistemas digitales reconfigurables implantados mediante un FPGA. También, tal como ocurre en

el ámbito industrial, en los proyectos se implementaron algunas características nuevas, que se definieron después de comenzar el proyecto, que modificaron de forma radical la implementación.

4. Bibliografía.

- "Internet Connected Devices Will Outnumber Humans Three to One by 2020" Charlton, Alistair 2014 URL: https://www.ibtimes.co.uk/internet-connected-devices-will-outnumber-humans-by-three-one-by-2020-1474216
- COMPUTER ARCHITECTURE: A Quantitative Approach, John L. Hennessy David A. Patterson, Fifth edition, 2012

Apéndices

A. REPORTE DE DESARROLLO DE UN MICROPROCESADOR MIPS SEGMENTADO CON SOPORTE DE RECURSIVIDAD.

Introducción

En el año de 1981, un equipo de trabajo en la universidad de Stanford comenzó el trabajo de lo que hoy se conoce como el primer procesador MIPS. Este procesador fue desde el principio concebido como un procesador que incluye pipeline, lo cual genera un mayor desempeño, sin embargo, el concepto del MIPS es que se ejecute una instrucción por un ciclo de reloj. Esto puede generar un desglose del diseño en dos partes: la parte de ejecutar una instrucción por ciclo de reloj y la segunda parte, posterior a la primera, el diseño de pipeline que pueda generar el rendimiento deseado del procesador.

Se podrá entender a un nivel más profundo el diseño de Pipeline en el procesador MIPS, los retos de diseño, comenzando con la primera parte de separar el MIPS de un ciclo en varias hasta poder implementar unidades de control para evitar Hazard tanto de control como de datos. Esto refuerza el conocimiento de la arquitectura RISC MIPS en este caso y da un entendimiento más profundo de los retos que los diseñadores se enfrentan en los sistemas digitales actuales.

Requerimientos de implementación

En general, el objetivo es implementar un procesador MIPS pipeline que soporte las siguientes instrucciones: ADD, ADDI, SUB, OR, ORI, AND, ANDI, LUI, NOR, SLL, SRL, LW, SW, BEQ, BNE, J, JAL y JR. Dichas instrucciones deben estar totalmente apegadas con la hoja de datos del MIPS. Además de eso la implementación que se diseñe debe ser capaz de soportar recursividad.

Para efectos del Pipeline se debe realizar en 5 etapas: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM) y WriteBack (WB). Además se debe implementar la unidad de forwarding para eliminar o reducir el número de burbujas que se necesitan insertar cuando se presenta un Hazard de datos en instrucciones de tipo aritmético-lógico y la unidad de data Hazard para reducir el número de burbujas que se necesitan insertar cuando se presenta un Hazard de control en instrucciones de brinco (BNE y BEQ), salto (J, JAL, JR) y de carga de datos a registros (LW). Para los saltos se debe implementar un predictor de brincos de 1 bit con una BTH y un BTB de 8 localidades y se debe comparar los registros en la etapa 3 de Execute (EX).

Simulación

Se debe incluir los resultados de simulación para 3 programas. El primero un programa que ejercite todas las instrucciones para verificar que el Pipeline funcione de manera adecuada, en el cual se podrán insertar NOP. El segundo es la transpuesta de una matriz que mueva una matriz de 4, 3, 2 a 3, 4, 2. Al finalizar correr el programa de HANOI para verificar la recursividad no haya sido afectada y para comparar el rendimiento.

FPGA

Incluir en IC, CPI, ClkRate y CPI time para implementación del MIPS corriendo los programas e incluir la comparación de velocidades. En el caso del IC se tiene que especificar el porcentaje de instrucciones de tipo R I y J. A demás se debe generar una salida de los programas por medio de UART.

Diseño MIPS

La arquitectura propuesta fue documentada en Visio y tiene como principales cambios los mencionados en la sección de requerimientos. Sin embargo cabe mencionar las más importantes decisiones del diseño:

- Para la instrucción JAL se cambió el RegisterFile para poder soportar dos escrituras de dos registros al mismo tiempo. Para simplificar el diseño, el segundo registro solo se puede escribir a la dirección de $RA (0x1F).
- Todos los saltos incondicionales, incluyendo JR, JAL y J, no necesitan insertar burbujas en el pipe, ya que se calcula en medio ciclo la nueva dirección en la segunda etapa de ID y se manda el valor inmediato para sobre escribir el PC con el nuevo valor.
- El control de saltos condicionales, incluyendo BNE y BEQ, se movió a un nuevo archivo de BranchControl.v que se conecta en la etapa de EX para la detección en la etapa requerida.
- Para implementar el Hazard de datos se implementó la señal que se había quitado de MemRead, lo cual ayuda a detectar los LW de una manera anticipada reusando la señal.

Todo lo anterior permite ejecutar los mismos programas de en el MIPS pipeline y asegurar la coherencia de datos intentando la mayor eficiencia posible en el PIPE y recortando lo más posible la ruta crítica de datos.

Los cambios en el control principal se muestran a continuación en el siguiente arreglo, como se puede observar el tamaño se conservó igual con algunos cambios para tener algo de compatibilidad con los módulos de la versión anterior:

```
                //{RegDst,ALUSrc,MemtoReg,JCtrl, Jal,
MemRead,MemWrite,RegWrite,ALUOp,PIN_EN}
        JR:     rControl = { 1'h1,  1'h0,  1'h0,   JMP_R, 1'h0, 1'h0,   1'h0,    1'h0, R_TYP,
1'h0};
        ROP:    rControl = { 1'h1,  1'h0,  1'h0,   PC_P4, 1'h0, 1'h0,   1'h0,    1'h1, R_TYP,
1'h0};
        BEQ:    rControl = { 1'h0,  1'h0,  1'hx,   PC_P4,  1'h0, 1'h0,   1'h0,    1'h0,  SUB_R,
1'h0};
        BNE:    rControl = { 1'h0,  1'h0,  1'hx,   PC_P4,  1'h0, 1'h0,   1'h0,    1'h0,  SUB_R,
1'h0};
        SW:     rControl = { 1'h0,  1'h1,  1'hx,   PC_P4,  1'h0, 1'h0,   1'h1,    1'h0,  ADD_R,
1'h0};
        LW:     rControl = { 1'h0,  1'h1,  1'h1,   PC_P4,  1'h0, 1'h1,   1'h0,    1'h1,  ADD_R,
1'h0};
```

LUI: rControl = { 1'h0, 1'h1, 1'h0, PC_P4, 1'h0, 1'h0, 1'h0, 1'h1, SLL_R, 1'h0};

ADDI: rControl = { 1'h0, 1'h1, 1'h0, PC_P4, 1'h0, 1'h0, 1'h0, 1'h1, ADD_R, 1'h0};

ORI: rControl = { 1'h0, 1'h1, 1'h0, PC_P4, 1'h0, 1'h0, 1'h0, 1'h1, OR_R , 1'h0};

ANDI: rControl = { 1'h0, 1'h1, 1'h0, PC_P4, 1'h0, 1'h0, 1'h0, 1'h1, AND_R, 1'h0};

JMP: rControl = { 1'h0, 1'hx, 1'hx, JMP_JAL, 1'h0, 1'h0, 1'h0, 1'h0, ZER_R, 1'h0};

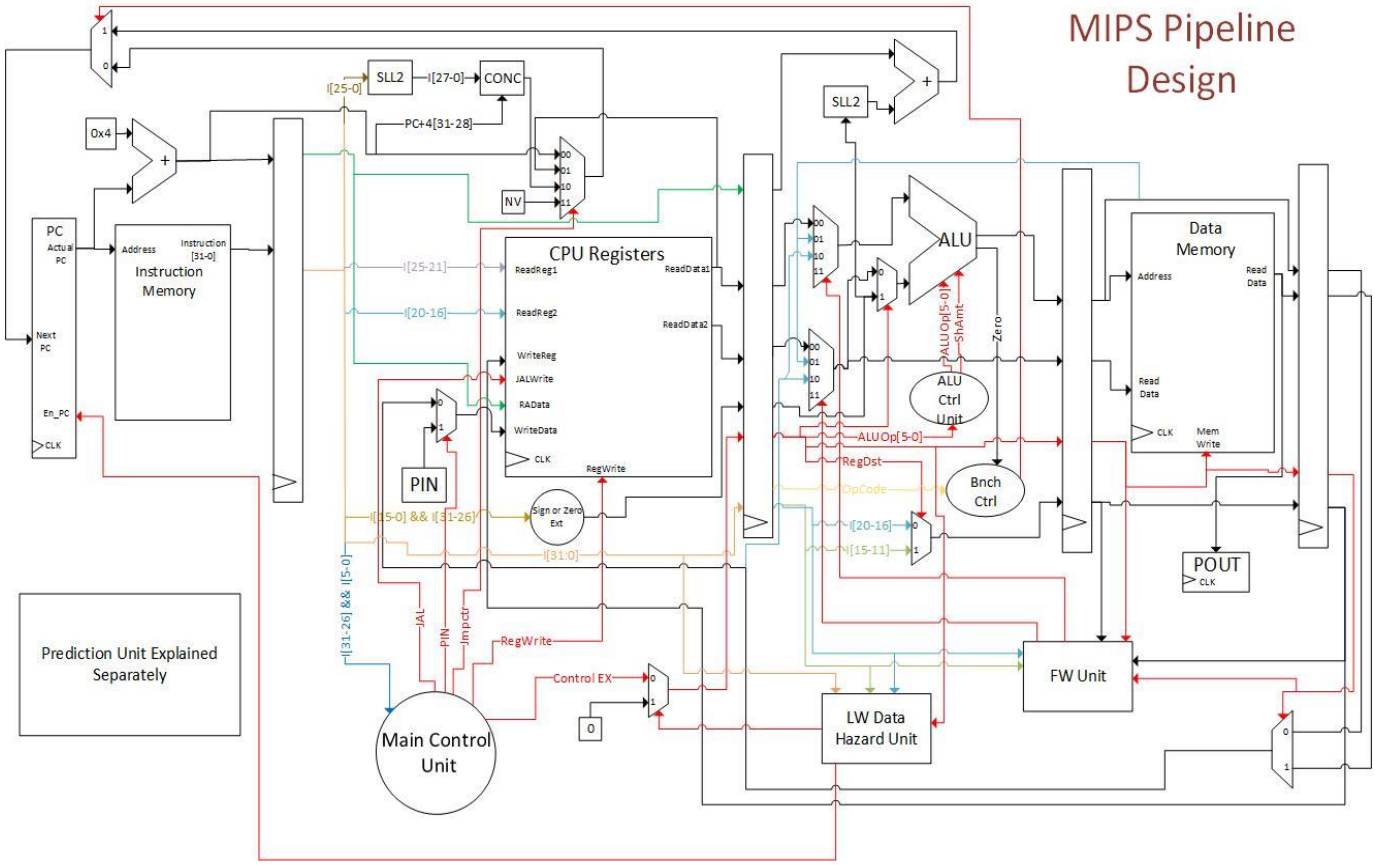
JAL: rControl = { 1'h0, 1'hx, 1'hx, JMP_JAL, 1'h1, 1'h0, 1'h0, 1'h0, ZER_R, 1'h0};

PIN: rControl = { 1'h0, 1'hx, 1'hx, PC_P4, 1'h0, 1'h0, 1'h0, 1'h1, ZER_R, 1'h1};

POUT: rControl = { 1'h0, 1'h1, 1'hx, PC_P4, 1'h0, 1'h0, 1'h0, 1'h0, ADD_R, 1'h0};

default: rControl = 16'hx;

A continuación se muestra el diseño por partes del MIPS pipeline que soporta todas las instrucciones en los requerimientos. Además de la ruta de datos que en su mayoría es de color negro y de 31 bits, también se agregó en el diagrama la etapa de control con líneas de color rojo:



Los archivos Verilog se anexan a la carpeta comprimida, en su caso solo se agregaron pocos archivos de Verilog conteniendo los pipes, control de saltos condicionales, unidad de FW y los Hazard incluyen control, datos y predicción de saltos condicionales. Para más información revisar los archivos adjuntos.

Programas de Prueba

Se implementaron 3 códigos de prueba para verificar la funcionalidad del procesador MIPS:

- El primer programa es un programa que contiene todas las instrucciones que soporta el MIPS que fue diseñado, incluidos los BNE y BEQ. En este programa al menos se espera que no se use ningún NOP por parte del compilador y principalmente se utiliza en la simulación para probar el funcionamiento de FWUnit, la unidad de Hazard de control para lw, bne y beq, asi como la unidad de predicción de saltos.

- El segundo programa diseñado para probar que el MIPS se encuentre funcionando de una manera correcta para probar intensivamente las unidades tanto de FW como de Control Hazard es la matriz transpuesta que a continuación se propone:

```
//Programa que calcula la matriz conjugada transpuesta de una matriz A
//Estas arreglos multidimensionales (matrices) deben ubicarse en memoria de datos.
//El arreglo B contendrá el resultado de la operación
int A[4][3][2] = {{{1,-3},{5,7},{9,0}},
                  {{2,-4},{6,8},{1,3}},
                  {{-1,0},{1,0},{0,0}},
                  {{-5,7},{9,0},{1,0}}};
int B[3][4][2];

//Estas variables se pueden asignar a registros
int i;
int j;

for(i=0; i<4; i++)
{
    for(j=0; j<3; j++)
    {
        B[j][i][0] = A[i][j][0];    //Copia la parte real de Aij a Bji
        B[j][i][1] = -A[i][j][1];   //Copia la parte imaginaria negada de Aij a Bji
    }
}
```

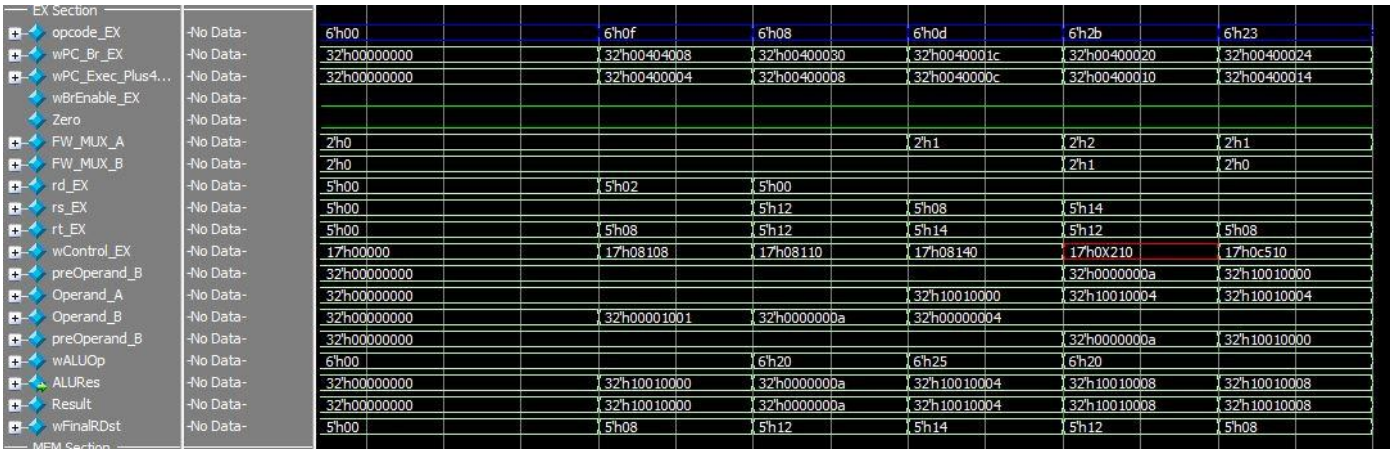
El código ensamblador se encuentra en los anexos para mayor información.

- Por último se utilizo es el mismo código de las torres de Hanói que se utiliza para probar el procesador uni-ciclo para comparar el desempeño de ambos procesadores con métricas como el CPI y el tiempo de ejecución. Cabe mencionar que solo se utilizó esa métrica, para programas diferentes el comportamiento del MIPS puede mejorar ya que el CPI es variable.

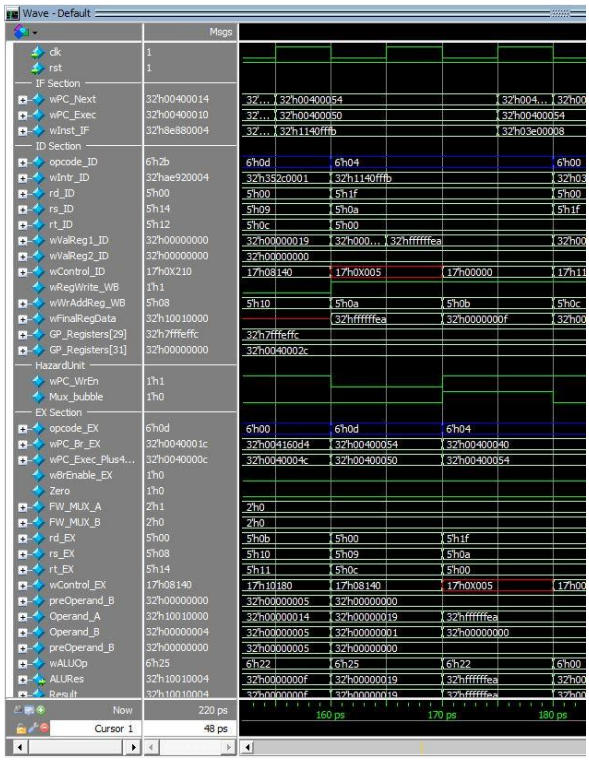
Simulación MODELSIM

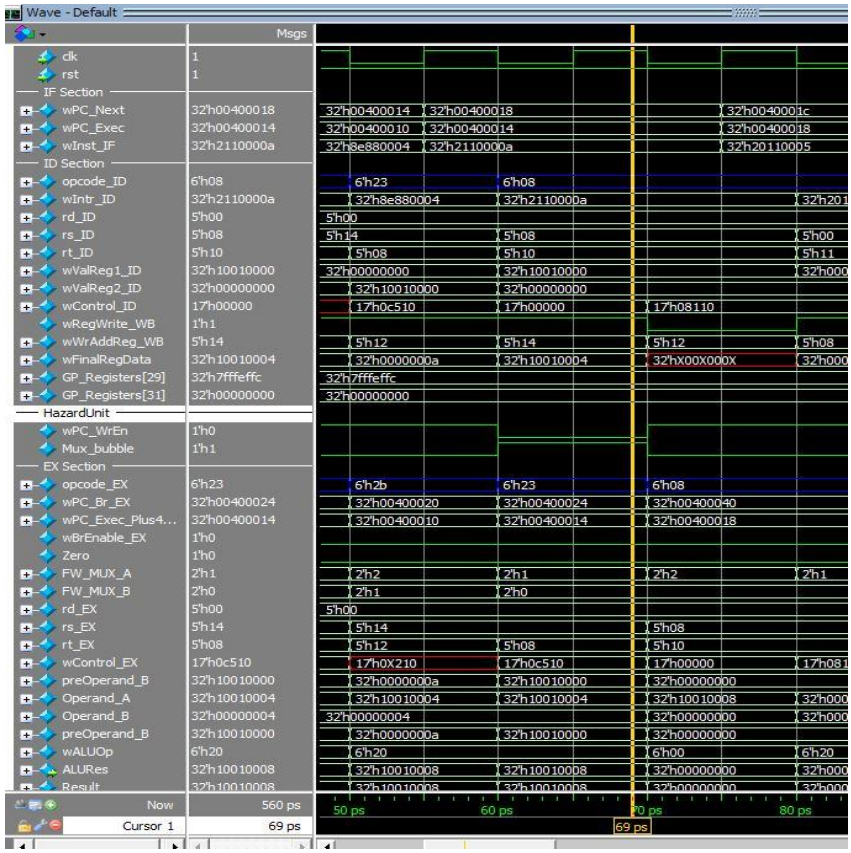
El programa que se eligió para probar toda la funcionalidad de MIPS es uno que prueba todas las instrucciones y es simplemente el programa de la tarea 5 con instrucciones extra para poder probar la funcionalidad completa. En las siguientes imágenes se muestra como funcionan la unidad de FW y la unidad de Hazard para Branch y LW:

- En la figura de abajo se puede fácilmente apreciar como funciona la unidad de FW para ambos datos al mismo tiempo, se aprecia como FW_MUX_A cambia de 1 a 2 y otra vez a 1. Al mismo tiempo el FW_MUX_B cambia a 1 sin que haya ninguna interferencia.



- En la imagen de la derecha se aprecia cómo funciona la unidad de control de Hazard cuando se detecta un branch en este caso con Opcode 4. Se puede observar que el Opcode se extiende por dos ciclos así como el valor del registro PC_Exec, controlado con la señal de Enable para el Program Counter.

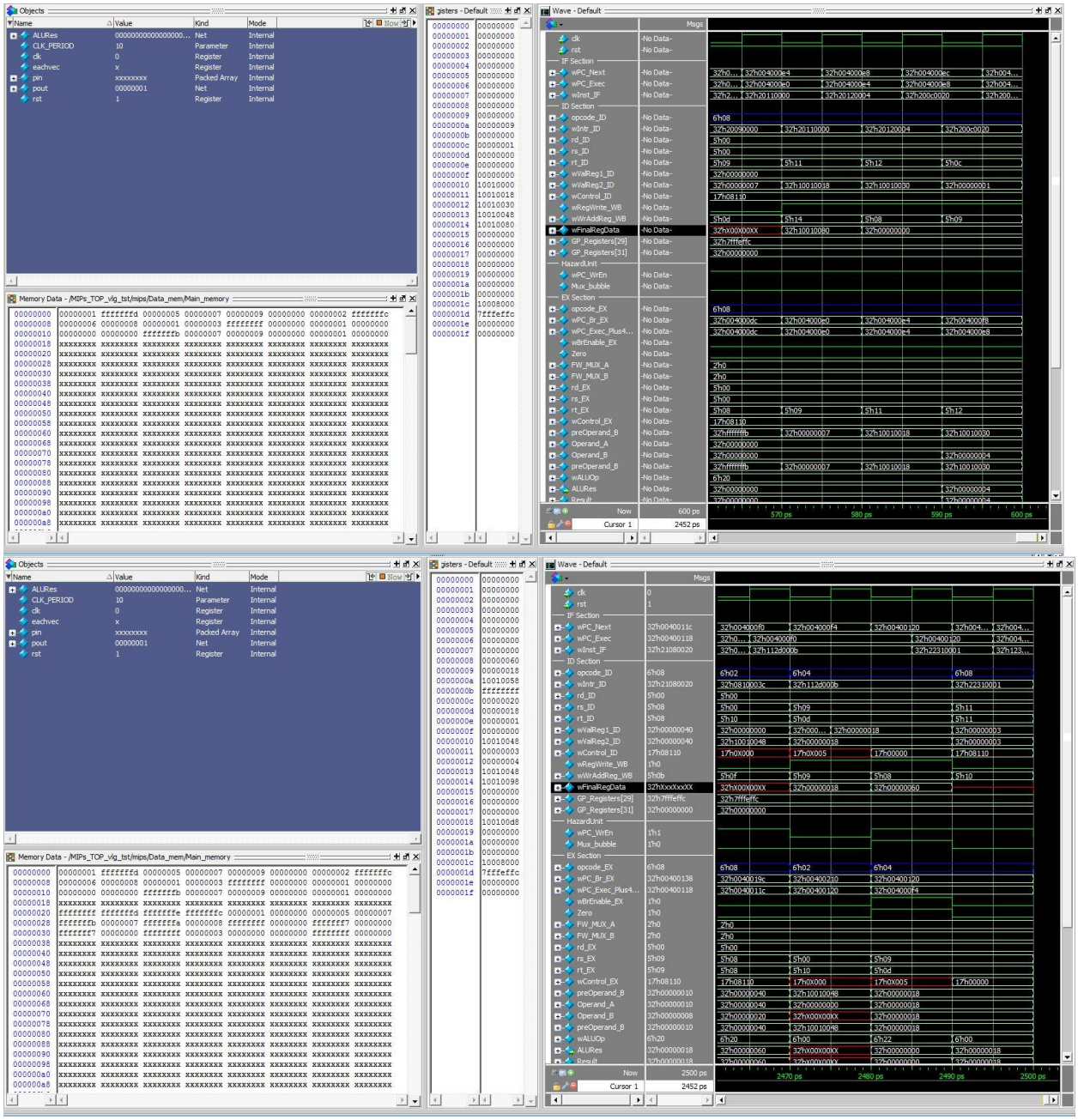




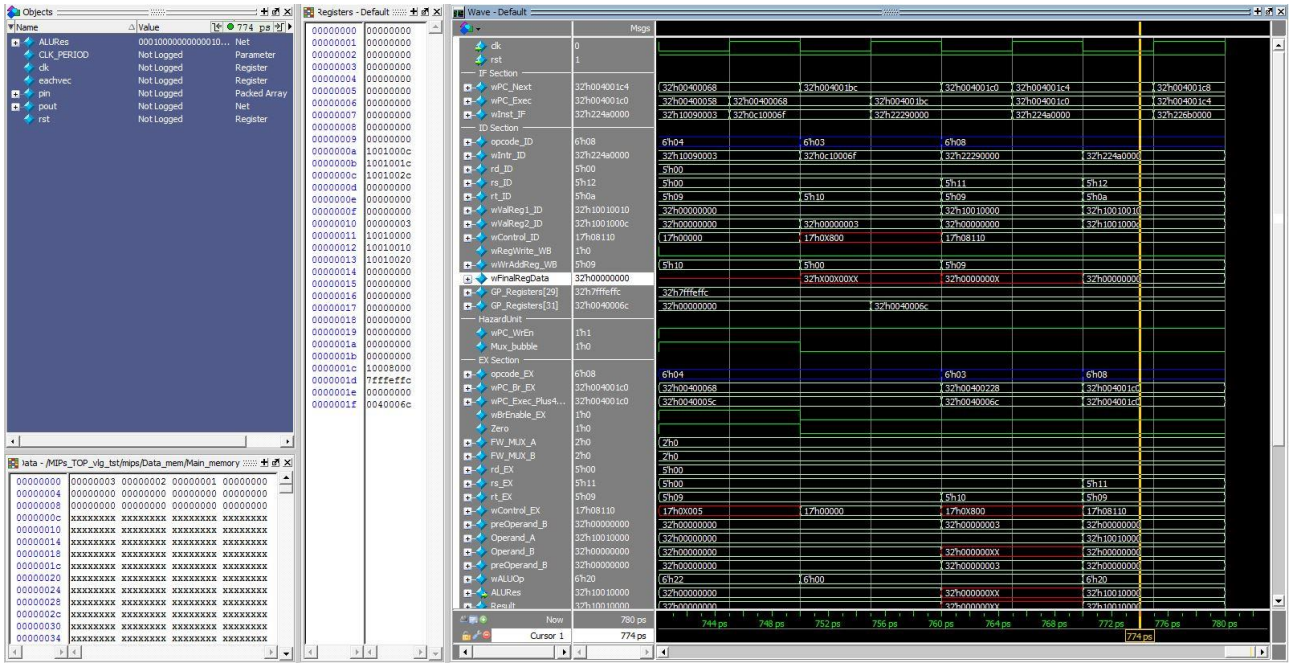
- En la imagen de la izquierda se puede observar la misma unidad de control de detección de Hazard pero en este caso con la instrucción LW (Opcode 0x23).

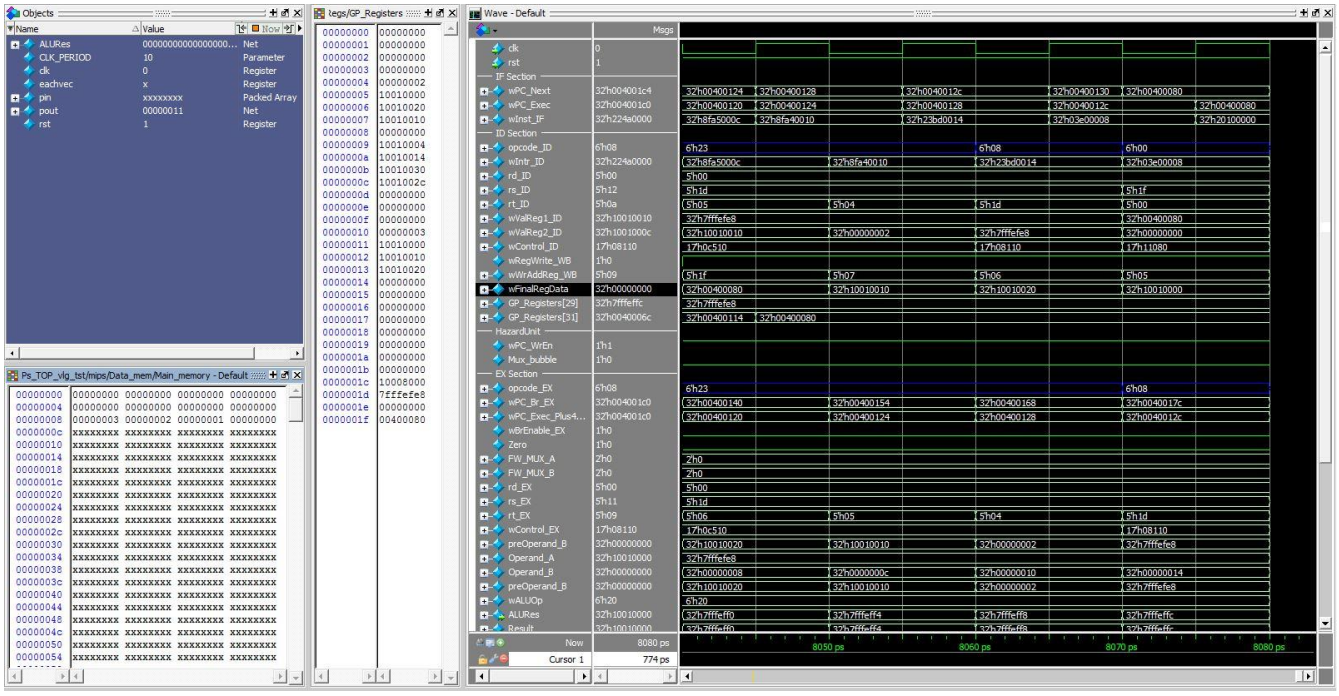
En esta se puede ver como se detecta el Hazard hasta que la instrucción viaja a la etapa de ejecución que es donde se inyectan los ceros en el control y la siguiente instrucción se alarga un ciclo para esperar el dato.

A continuación se muestra como el programa de la Matriz transpuesta funciona a través de cómo se carga la matriz y en la siguiente se carga transpuesta en una dirección más abajo.

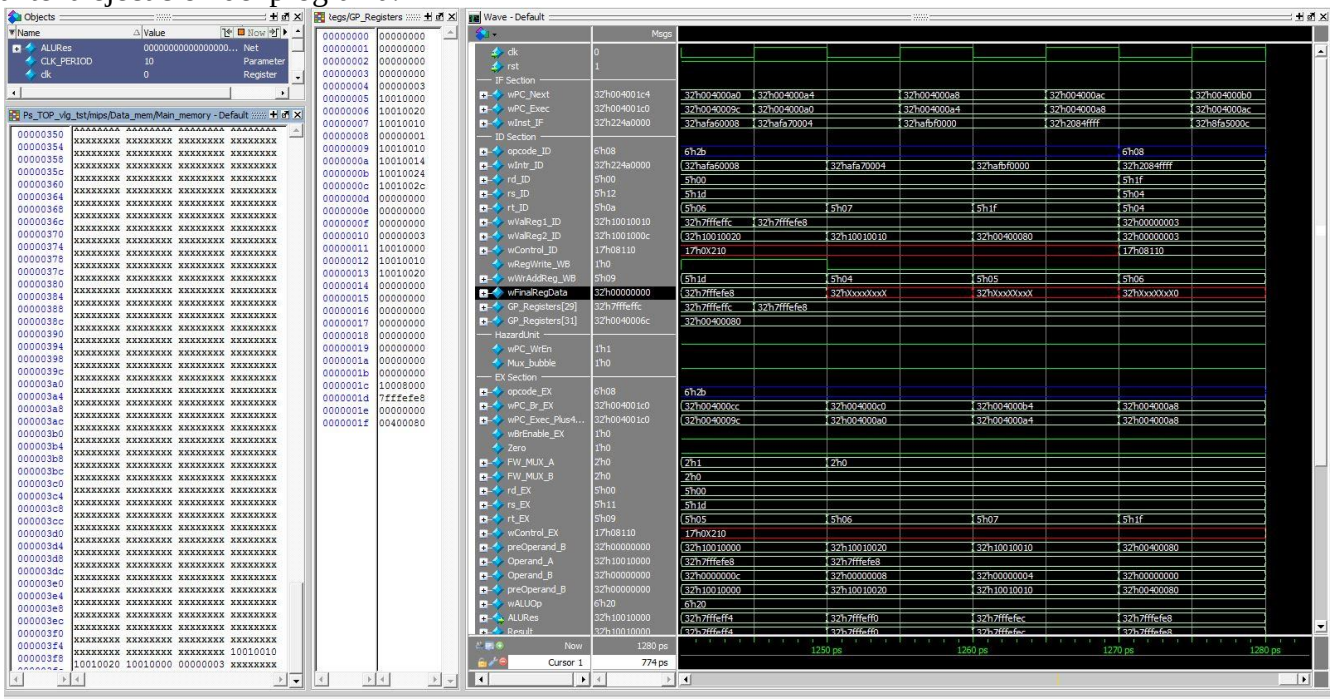


Después de esto se finalizó con las torres de Hannoi mencionado anteriormente, con unas pequeñas modificaciones insertando algunos NOP.


Objects
Registers - Default
Wave - Default
Wave - Default

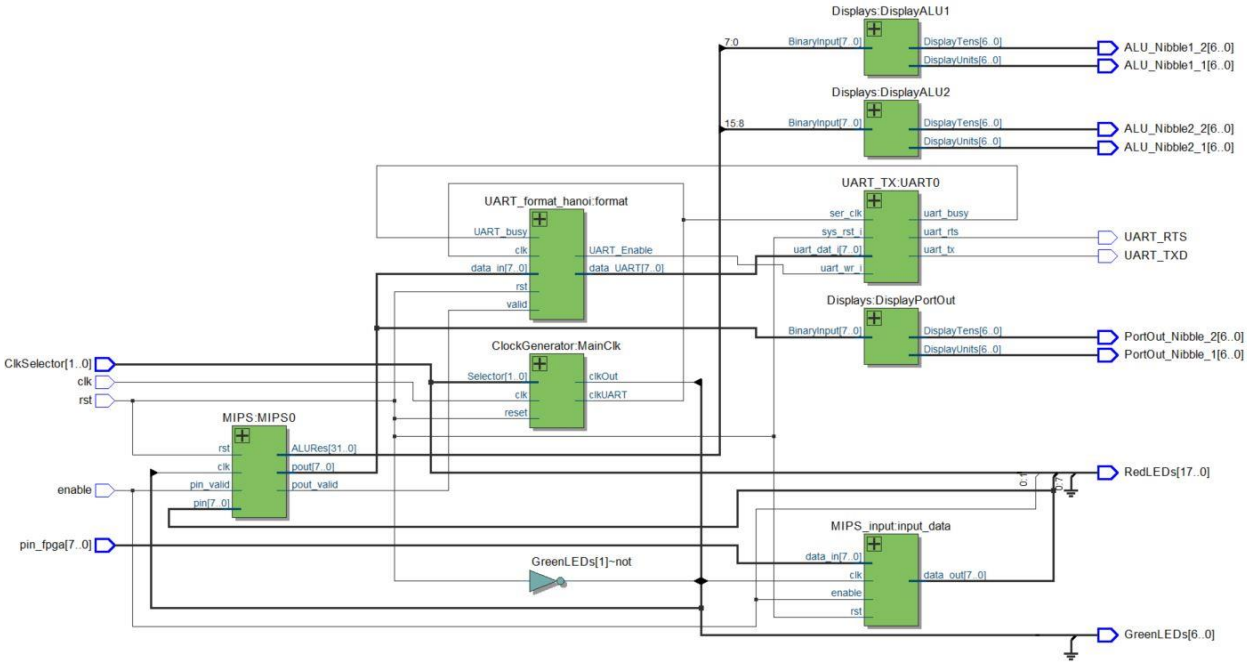

Objects
Wave - Default

Para confirmar que el stack siga funcionando como se debe se revisó a profundidad durante la ejecución del programa.

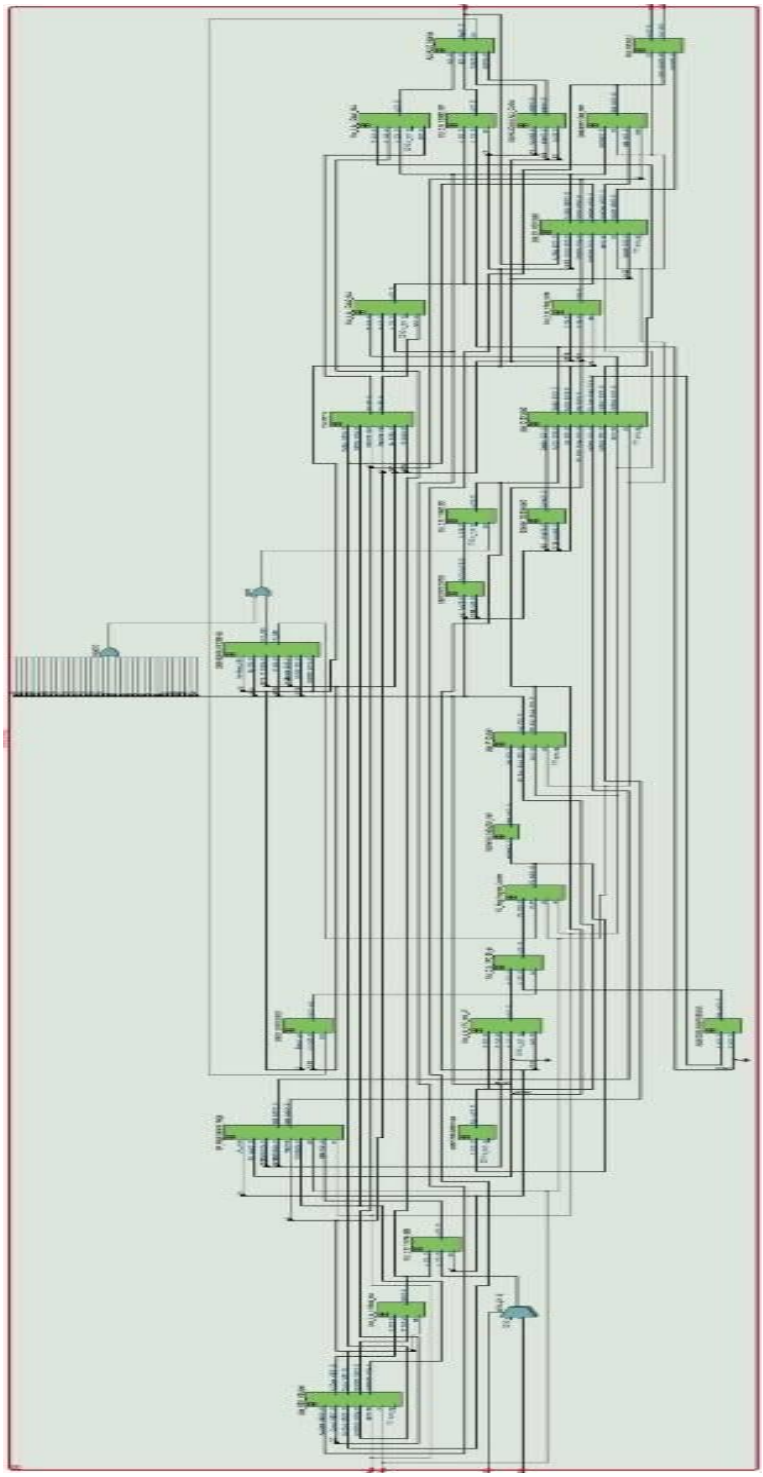


RTL MIPS Quartus

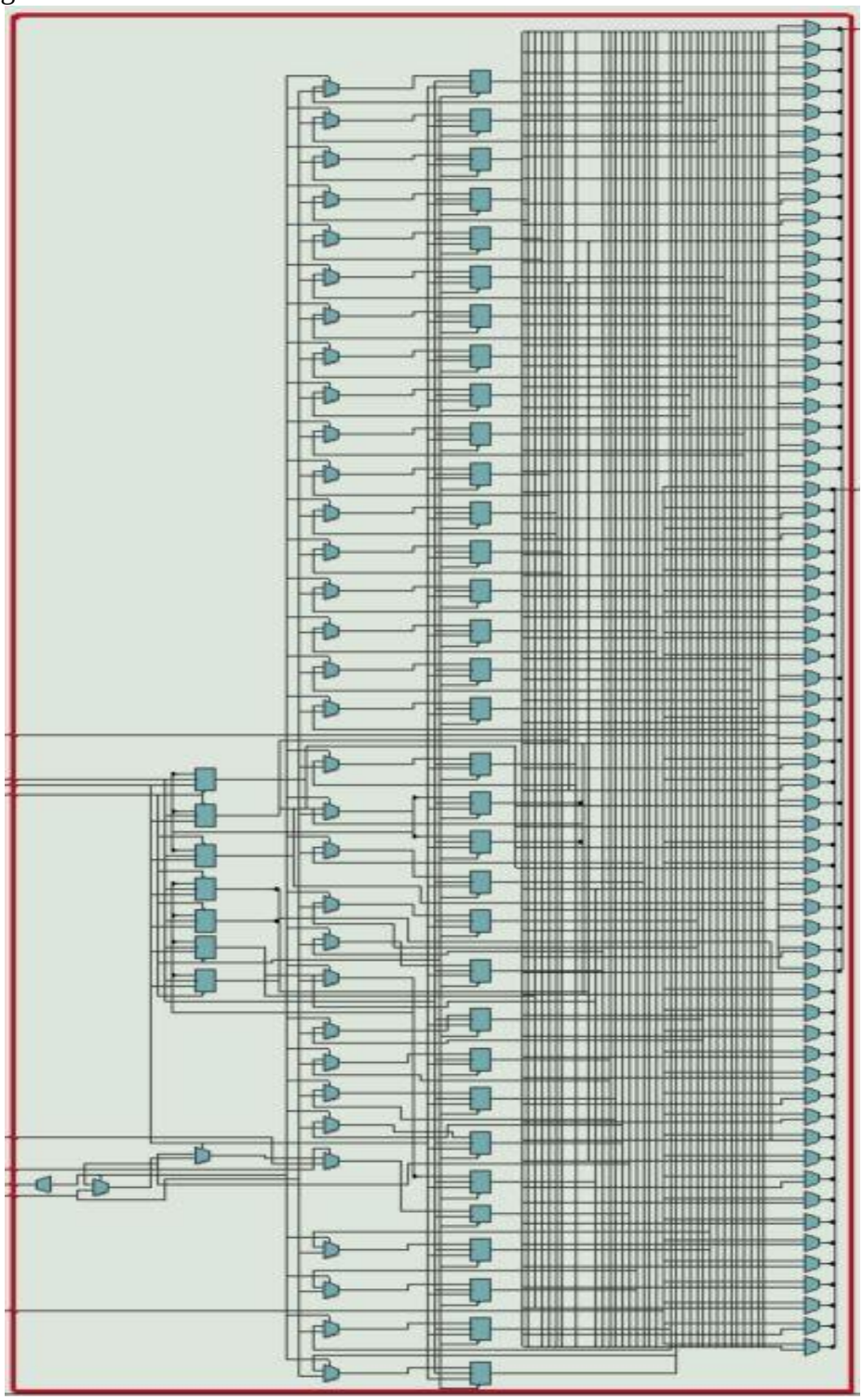
A continuación se muestra el resultado del RTL generado por Quartus, se hace notar que este se hace un desglose de Top-Down comenzando con el RTL del modelo general el cual tiene su instancia del MIPS:



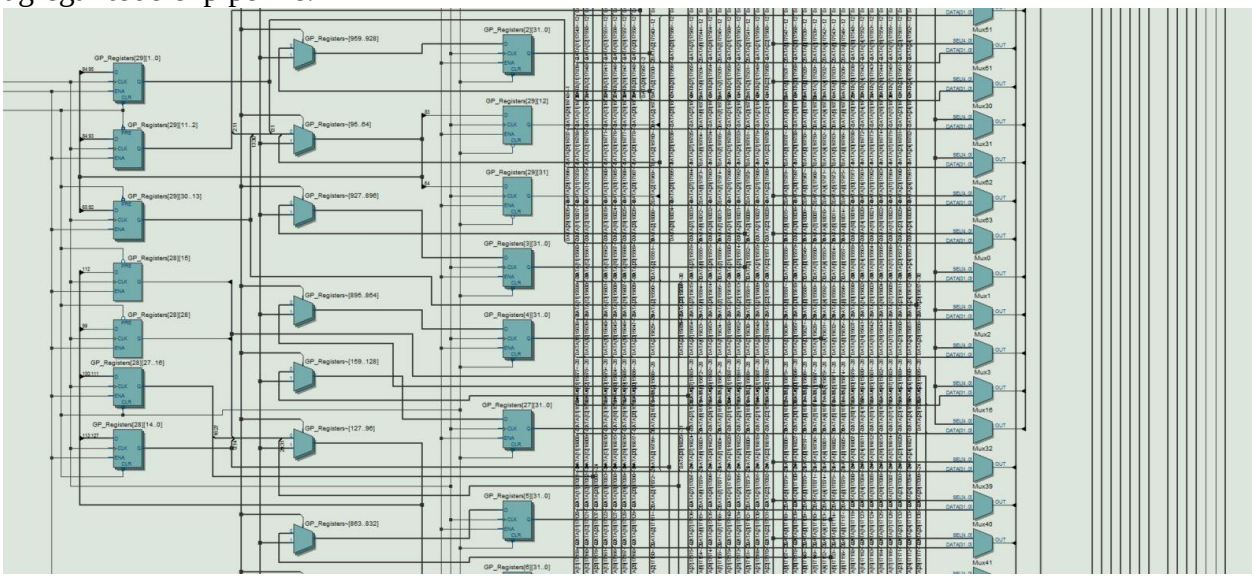
Partiendo del RTL anterior se puede abrir el RTL de la instancia del MIPS, y se puede observar como la complejidad aumenta y tan solo en una sola vista es difícil apreciar la cantidad de módulos y conexiones nuevas que se han creado a partir del modelo original. A continuación se muestra un aumento y desglose de todo el MIPS. Para más información consultar los archivos anexos.



Teniendo una perspectiva más amplia se puede observar que incluso el RegisterFile es bastante más complejo simplemente con un cambio que se agregó, teniendo un buen pedazo de lógica extra:



Para finalizar el análisis del RTL se hace un acercamiento al RegisterFile para ver la lógica extra que se agregó con un sencillo cambio al código de Verilog, lo cual provoca un cambio bastante considerable en el RTL (parte izquierda) podríamos imaginar el enorme cambio al agregar todo el pipeline.



Implementación FPGA

Para la implementación se logró implementar las Torres de Hanói y la transpuesta de una matriz con algunas complicaciones sobre todo de formato de salidas y de entradas así como el cambio de programación dentro del FPGA.

Se usan los mismos métodos que en el procesador pasado tanto de acceso como de lectura. A continuación se explica brevemente los métodos.

Entradas MIPS y PIN

Como entrada se usaron principalmente las señales de Reset, Enable, ClkSelector y 8 bits de data in para seleccionar el número de torres una vez el programa finalice correctamente. También se tiene como entrada un clk de 50MHz que se divide y estabiliza con un PLL.

Las señales de Enable y data in se usaron en conjunto, siendo la señal de Enable el habilitador de un registro de 8 flip-flops para tener una entrada más estable al diseño del MIPS. El ClkSelector se usa para escoger una frecuencia de 1Hz o 5Hz. Por último la señal de Reset manda todo el procesador a su estado inicial o de default donde el programa se ejecuta desde el comienzo con los números de discos cargados en el programa original (3).

La instrucción PIN fue diseñada para poder escribir a cualquier registro interno del procesador sin importar el inmediato ni el registro destino, ya que usa un multiplexor que inyecta el dato cuando se requiere. Esta instrucción es usada solamente para el registro $s0 que es el que contiene el número de discos que se van a mover. El formato de instrucción es el siguiente, y se usó un formato tipo I pero con unidad de control diferente:

Opcode [31-26] 0x1F	[25-21] XXXXX	Rt [20-16] DesReg	[15-0] XXXXXXXXXX

Al usar esta instrucción se tuvo que manualmente cambiar la operación introducida por la de PIN. A continuación se muestra el pedazo de código HEX que se cambió para darle soporte:

```
34100000#####Change instruction for PIN to write $s0 ###7c100000
```

Salidas MIPS y POUT

Las salidas principalmente son LEDS, DISPLAY de 7 segmentos y la salida serial UART que imprime las torres. Estas salidas se configuraron de diferente manera para poder depurar y también para arrogar la mayor información posible. Los LEDS muestran algunas de las señales internas de la interfaz así como del MIPS para poder identificar fácilmente el estado del diseño.

Se implementaron 6 salidas de DISPLAY de 7 segmentos, de las cuales 4 contienen los primeros 16 bits del resultado de la ALU y los otros 2 contienen el POUT que se espera de la salida. Cabe mencionar que este POUT no es válido a menos que se encienda el LED POUT_valid, esta verificación se tuvo que agregar para controlar el UART así poder saber cuál dato mandar.

La implementación de la UART fue un reto debido al control de datos que fluye desde el MIPS. Al final se implementó un control donde el último digito de cada torre sea un cero. De esta manera no se imprimen todos los ceros sino solo el último de cada torre y se continúa con la siguiente. También debe considerarse el aumento de IC por estas instrucciones es considerable, ya que por esa estructura se debe leer cada vez que se cambia un disco con funciones de recorrido de arreglos (Aproximadamente un 60% de incremento en el IC).

Por último se agregó la instrucción POUT, que es muy similar a PIN, la cual sigue la misma estructura de una instrucción tipo I. En esta instrucción los 8 bits menos significativos del dato leído de memoria se almacenan directamente en un registro de 8 flip-flops y de ahí a la salida en el siguiente ciclo de reloj. El formato es el siguiente:

Opcode [31-26] 0x1E	[25-21] RS SrcReg	[20-16] XXXXX	[15-0] Immediate_operand

Al usar esta instrucción se tuvo que manualmente cambiar la operación introducida por la de POUT. A continuación se muestra el pedazo de código HEX que se cambió para darle soporte:

```
20000000#####Change instruction for POUT to read $t1 ###79200000
```

MIPS Performance (Pipeline vs Single Cycle)

Posteriormente a la implementación con los módulos de entradas y salidas se calculó la frecuencia máxima que puede soportar los diseños implementados con el programa de Quartus de Altera.

El Procesador MIPS Pipeline está en el tope con un peor caso de 45.57MHz:

Slow 1200mV 0C Model Fmax Summary

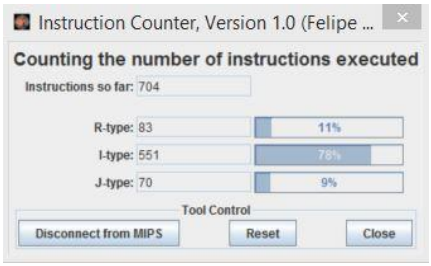
	Fmax	Restricted Fmax	Clock Name
1	45.57 MHz	45.57 MHz	ClkSelector[0]
2	241.72 MHz	241.72 MHz	MainClk|PLL0|altpll_component|auto_generated|pll1|clk[0]
3	321.44 MHz	321.44 MHz	ClockGenerator:MainClk|ClockDivider:ClockDivider_9600Hz|Toggle:FF_Toggle|Shot_reg

El Procesador MIPS de ciclo simple está un poco abajo con un peor caso de 43.26MHz:

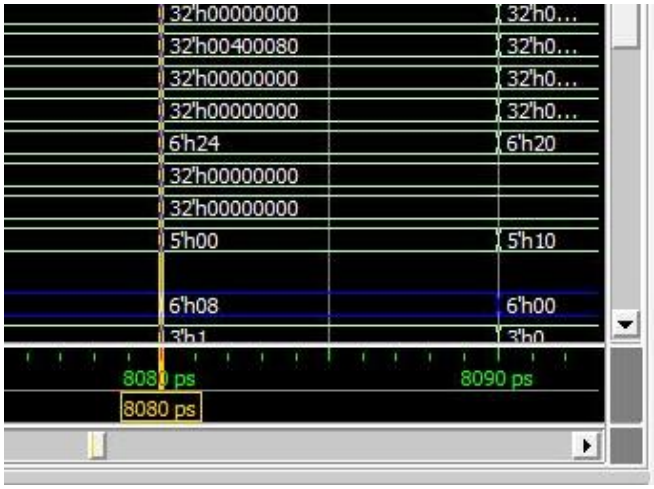
Slow 1200mV 0C Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name
1	43.26 MHz	43.26 MHz	ClkSelector[0]
2	206.06 MHz	206.06 MHz	MainClk|PLL0|altpll_component|auto_generated|pll1|clk[0]
3	365.63 MHz	365.63 MHz	ClockGenerator:MainClk|ClockDivider:ClockDivider_9600Hz|Toggle:FF_Toggle|Shot_reg

Con estos datos además del IC como se hizo notar en la sección anterior se dispara por el uso de PIN y POUT. A continuación se muestran los nuevos IC para el último programa de las torres de Hanói generado:



Para el procesador Pipeline el Instruction Count se realiza con ModelSim para tener una aproximación más exacta de cuantos ciclos de reloj se requieren para terminar el programa de torres de Hanói de 3 torres:



Con estos datos se puede calcular el performance de los procesadores MIPS diseñados previamente para el programa de las torres de Hanói para 3 torres con la información de Quartus y de ModelSim:

MIPS ciclo sencillo	MIPS Pipeline
IC = 704 CPI = 1 ClkRate = 43.26MHz CPU Time = (704*1)/43.26M= 16.27uS	IC = 622 CPI = 808/704 = 1.14 ClkRate = 45.57 CPU Time = (704*1.14)/45.57M= 17.61uS

Este performance es esperado por el CPI más alto del MIPS Pipeline además de la poca mejora en frecuencia que se obtuvo. Aunado a que el FPGA está compuesto de celdas lógicas que tienen un retardo en el reloj generado externamente y a través de un Multiplexor, si se hace notar el reloj interno tiene una ganancia más notable de 40MHz.

Resultados

Se logró implementar correctamente el diseño de un procesador MIPS pipeline en la tarjeta DSi-150 de altera de una manera correcta. A pesar que el funcionamiento interno del procesador es correcto también es bastante mejorable, la mayoría de los problemas encontrados fueron dependencias de datos y sobre todo en la parte de predicción dinámica de saltos. A continuación se listan las mejoras que se podrían implementar en MIPS pipeline:

- Hazard de datos con instrucciones como JR al escribir a $RA por la inviabilidad del FW Unit de proveer la dirección correcta, causa bucles infinitos o causa repeticiones indeseables de lazos. Para resolver este problema se necesita insertar NOPS, lo cual causa ineficiencias.

- Escribir con JAL la dirección de retorno al mismo tiempo que el WB está escribiendo al mismo registro $RA provoca corrupción de datos y comportamiento indefinido del valor de este registro. A pesar que este problema no se presentó en los programas podría llegar a darse muy comúnmente en programas más grandes.

- Se podría implementar además predicción de datos mapeada como memoria cache para identificar de una manera más eficiente la predicción y solo equivocarse máximo 2 veces en un bucle, que sería a la entrada y la salida del mismo. Se intentó realizar este tipo de predicción de datos, pero debido a cuestiones de tiempo y complejidad se decidió no realizarlo.

- Se podría acortar los Paths de datos para poder tener un ClkRate mucho más grande, esto dejando que las instrucciones de Salto (J) se ejecuten en la fase de IF y las ramificación (B) en la fase de ID. Esto mejoraría dramáticamente el desempeño y además recortaría la complejidad del diseño en gran medida.

Conclusiones

A pesar de la evidente victoria del procesador de un ciclo, el objetivo de acortar el retraso en las señales se logró satisfactoriamente en las imágenes posteriores (primero la de MIPS Pipeline, segundo MIPS ciclo simple), las cuales muestran un recorte del tiempo entre etapas de la mitad.

Slow 1200mV 85C Model Setup: 'ClkSelector[0]'

	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	-11.840	MIPS:MIPS0|pipe_ID_EX:pipe2|rInst_EX[19]	MIPS:MIPS0|pipe_EX_MEM:pipe3|rALURes_MEM[14]	ClkSelector[0]	ClkSelector[0]	1.000	-0.085	12.753
2	-11.743	MIPS:MIPS0|pipe_MEM_WB:pipe4|rWrAddReg_WB[0]	MIPS:MIPS0|pipe_EX_MEM:pipe3|rALURes_MEM[14]	ClkSelector[0]	ClkSelector[0]	1.000	-0.085	12.656
3	-11.658	MIPS:MIPS0|pipe_MEM_WB:pipe4|rWrAddReg_WB[4]	MIPS:MIPS0|pipe_EX_MEM:pipe3|rALURes_MEM[14]	ClkSelector[0]	ClkSelector[0]	1.000	-0.085	12.571
4	-11.654	MIPS:MIPS0|pipe_EX_MEM:pipe3|rDestReg_MEM[2]	MIPS:MIPS0|pipe_EX_MEM:pipe3|rALURes_MEM[14]	ClkSelector[0]	ClkSelector[0]	1.000	-0.084	12.568
5	-11.572	MIPS:MIPS0|pipe_ID_EX:pipe2|rInst_EX[16]	MIPS:MIPS0|pipe_EX_MEM:pipe3|rALURes_MEM[14]	ClkSelector[0]	ClkSelector[0]	1.000	-0.085	12.485
6	-11.551	MIPS:MIPS0|pipe_ID_EX:pipe2|rInst_EX[19]	MIPS:MIPS0|PC_Reg:Program_Counter|PC[17]	ClkSelector[0]	ClkSelector[0]	0.500	0.396	12.445
7	-11.547	MIPS:MIPS0|pipe_ID_EX:pipe2|rInst_EX[19]	MIPS:MIPS0|PC_Reg:Program_Counter|PC[14]	ClkSelector[0]	ClkSelector[0]	0.500	0.396	12.441

Slow 1200mV 85C Model Setup: 'ClkSelector[0]'

	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
58	-23.844	MIPS:MIPS0|PC_Decoder:Program_Counter|PC[3]	MIPS:MIPS0|GP_Regs:General_Regs|GP_Registers[27][1]	ClkSelector[0]	ClkSelector[0]	1.000	-0.106	24.736
59	-23.843	MIPS:MIPS0|PC_Decoder:Program_Counter|PC[6]	MIPS:MIPS0|GP_Regs:General_Regs|GP_Registers[22][30]	ClkSelector[0]	ClkSelector[0]	1.000	-0.097	24.744
60	-23.840	MIPS:MIPS0|PC_Decoder:Program_Counter|PC[6]	MIPS:MIPS0|GP_Regs:General_Regs|GP_Registers[18][30]	ClkSelector[0]	ClkSelector[0]	1.000	-0.097	24.741
61	-23.834	MIPS:MIPS0|PC_Decoder:Program_Counter|PC[5]	MIPS:MIPS0|GP_Regs:General_Regs|GP_Registers[4][1]	ClkSelector[0]	ClkSelector[0]	1.000	-0.091	24.741
62	-23.833	MIPS:MIPS0|PC_Decoder:Program_Counter|PC[5]	MIPS:MIPS0|GP_Regs:General_Regs|GP_Registers[12][1]	ClkSelector[0]	ClkSelector[0]	1.000	-0.091	24.740
63	-23.823	MIPS:MIPS0|PC_Decoder:Program_Counter|PC[4]	MIPS:MIPS0|GP_Regs:General_Regs|GP_Registers[10][30]	ClkSelector[0]	ClkSelector[0]	1.000	-0.124	24.697
64	-23.815	MIPS:MIPS0|PC_Decoder:Program_Counter|PC[3]	MIPS:MIPS0|GP_Regs:General_Regs|GP_Registers[21][1]	ClkSelector[0]	ClkSelector[0]	1.000	-0.091	24.722

Otro aspecto a considerar es que la implementación en FPGA varía mucho con respecto a un ASIC, donde la optimización como el árbol de relojes y la optimización de la síntesis con la misma herramienta pero con licencias.

Bibliografía

- Patterson, D.A.; Hennesy, J.L..Computer Organization and Design: The Hardware/Software interface. Morgan Kaufman, 5th Edition.
- https://en.wikipedia.org/wiki/MIPS_instruction_set

B. REPORTE DE VERIFICACIÓN FORMAL DE LA DISPATCH UNIT DEL MIPS SUPERSCALAR

Introducción.

La llamada revolución del silicio[1] desborda evidencias de su abundancia y ubicuidad a cada paso que damos. Es común ver a personas en las calles con sus teléfonos móviles en mano (a veces más de uno), o en las escuelas y oficinas ver a todo mundo hacer uso de tabletas, computadoras portátiles, computadoras de escritorio y equipos de comunicación y en casa también es común que se disponga de televisores inteligentes, computadoras, consolas de videojuegos, sistemas de TV por cable o DTH y algún sistema para acceder a internet. Todos estos equipos hacen uso de sistemas digitales basados en chips construidos a partir de semiconductores y año con año surgen, e incorporamos a nuestra cotidianidad, nuevos dispositivos que buscan facilitarnos el trabajo o hacernos la vida más cómoda y placentera.

Este dinamismo demanda constantemente nuevos diseños de hardware y adaptaciones de los ya existentes. Pero se enfrenta a un impedimento que no tiene que ver con la fase de diseño en sí misma, sino con la verificación de los diseños.

Costos de millones de dólares y retardos de entre tres y seis meses para liberar nuevas versiones de complicados y enormes controladores, procesadores o circuitos integrados de aplicación específica, son comunes en las empresas del sector de semiconductores. Esto constituye un gran incentivo para las compañías para concentrar sus esfuerzos en hacer sus sistemas (chips) correctamente desde el principio y ello ha dado pie a que en las empresas de diseño y fabricación de semiconductores se implanten los procesos de verificación como esfuerzos sistemáticos y de equipo que aseguran que sus chips funcionan como se espera. El proceso de verificación desde hace varios años y hasta ahora se hace en paralelo al proceso de diseño y son procesos complementarios que se refuerzan mutuamente.

Todo inicia con la especificación del dispositivo, su descripción en algún lenguaje HDL en la modalidad RTL, una metodología de verificación a la cual apegarse y una herramienta para la creación de un ambiente de verificación (un testbench en su forma más estructurada y elaborada).

A partir de este punto, se desarrollan un conjunto de tareas cuyo propósito es estresar el chip y poner a prueba todos los aspectos funcionales señalados en su especificación bajo todas las posibles combinaciones de entradas, configuraciones y condiciones de operación; e incluso pueden llevarse las cosas hasta el punto de probar aspectos que no están descritos en la especificación pero que pueden presentarse como resultado de un uso no esperado del dispositivo por parte del usuario final. Todo esto con el único propósito de lograr un alto nivel de confiabilidad en la funcionalidad que el sistema (chip) ofrece.

Se buscarán errores de lógica, se someterá el sistema a una variedad de casos de funcionamiento incluidos aquellos que por sí mismos constituyen errores (y con ello se validará el manejo que el sistema hace de ellos). También se asegurará que el sistema (chip) alcanza los objetivos de desempeño y que funciona bajo condiciones que son el resultado de combinaciones inusuales de parámetros (los llamados casos esquina), a fin de confirmar que

[1] Moore, Gordon. *Cramming more components onto integrated circuits*. Electronics. Volume 38, number 8, april 19th, 1965.

sus elementos y características funcionales, tales como: registros, manejo de interrupciones, accesos a memoria y manejo de periféricos trabajan como se espera.
Los errores más fáciles de encontrar son que se presentan en los bloques más elementales del sistema (aquellos que normalmente sólo realizan una tarea). Los errores que les siguen en nivel de complejidad son los errores debidos a las interacciones entre dos o más bloques del sistema; muchos problemas pueden surgir cuando dos bloques simples se conectan entre sí para realizar alguna tarea más compleja. Y finalmente existen los errores más complicados de rastrear que son los errores a nivel general y que emergen cuando el sistema completo está en operación y recibe estímulos desde el exterior y debe generar una respuesta acorde con la especificación.
El principio más importante que debe respetarse como ingenieros de validación es: "Los errores (bugs) son buenos". Como señala *Chris Spear* en su libro **SystemVerilog for verification:** "No sean tímidos al encontrar el siguiente error, no duden en sonar la campana cada vez que descubran alguno, y más aún, siempre rastrea los detalles de cada error encontrado"[2]. Este ha sido el objetivo del actual proyecto.
Y parafraseando una vez más a *Chris Spear*: "Nunca puedes probar que no existen errores remanentes, así que necesitas regresar constantemente con nuevas tácticas de verificación"[3].
Así que este reporte es el resumen de nuestras varias incursiones con diversas tácticas para encontrar los posibles errores funcionales del sistema que nos fue encomendado.
En este proyecto se ha empleado SystemVerilog debido a las muchas ventajas que tiene. Tal vez la más relevante de ellas es que puede verse como cinco lenguajes en uno: un conjunto de declaraciones sintetizables, un lenguaje de aserciones, un lenguaje de restricciones, un lenguaje de cobertura y un lenguaje orientado a objetos (que resulta muy útil para implantar las abstracciones necesarias en las estructuras de datos y métodos que se utilizarán en el ambiente de verificación). Cada uno de ellos con su propia sintaxis, semántica y características pero integrados a partir del uso en común de varias palabras reservadas y estructuras de programación fáciles de comprender.
Y como herramienta para el desarrollo del ambiente de verificación se decidió emplear el simulador Questa versión 10.4 de la empresa Mentor Graphics.

[2] Spear, Chris. *SystemVerilog for Verification. A Guide to Learning the Testbench Language Features*. Third Edition. Springer. New York, 2012. p. 1.
[3] *Ibidem*. p. 4.

Descripción del Device Under Validation – Dispatch Unit.

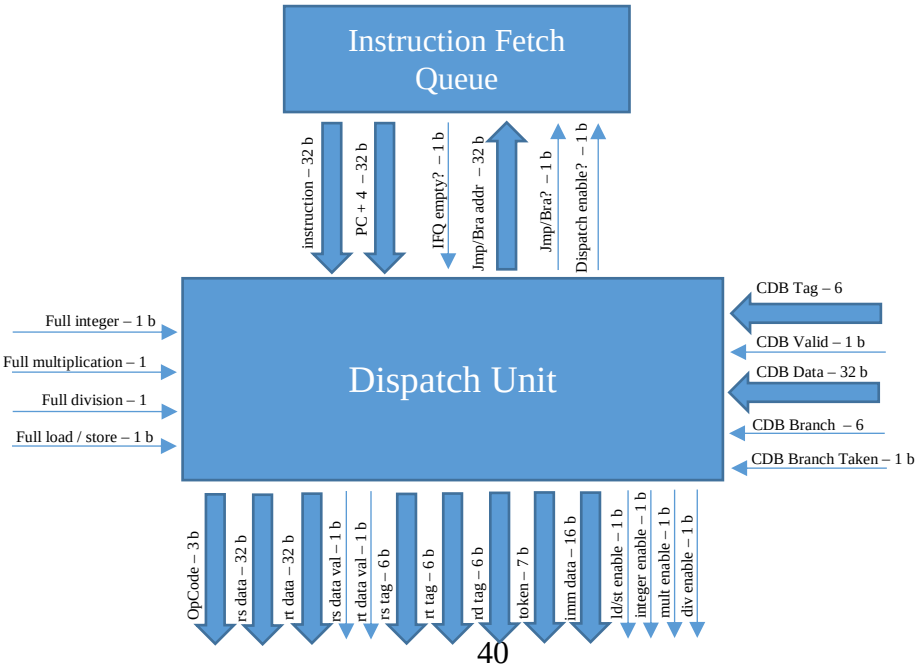
La Dispatch Unit es la responsable de leer las instrucciones que le son enviadas por la unidad Instruction Fetch Queue (IFQ) y de enviarlas a sus respectivas unidades de ejecución. Dentro de la Dispatch Unit cada instrucción es procesada una por una y siguiendo el orden del programa. La Dispatch Unit debe realizar las siguientes tareas:

- Leer la instrucción entregada por la Instruction Fetch Queue.
- Decodificar cada instrucción.
- Despachar la instrucción y sus operandos a la Execution Unit Queue que le corresponde.
- Actualizar el Register Status Table (RST) y el Register File (RF).
- Calcular la dirección de salto para las instrucciones del tipo Jump y Branch.
- Ejecutar las instrucciones del tipo Jump.
- Detener la entrega de instrucciones por parte de la IFQ cuando una instrucción del tipo Branch es ejecutada.

La Dispatch Unit es el corazón del sistema e interactúa con la IFQ, las cuatro unidades de ejecución (UE) y el Common Data Bus (CDB). En el presente documento serán utilizadas las siguientes abreviaciones DU para Dispatch Unit, IFQ para Instruction Fetch Queue, EU para Execution Unit y CDB para Common Data Bus.
El banco de registros, el RST y el TAG FIFO se encargan de implementar el mecanismo de renombramiento de registros (REGISTER RENAMING). La unidad de decodificación funciona igual que en la implementación de un pipeline de 5 etapas. Para todas las instrucciones, sean o no del tipo Branch o Jump, se calculan las direcciones de salto. En el caso de las instrucciones del tipo Branch se genera la lógica que detiene la IFQ hasta que la instrucción Branch no sea resuelta (lo que significa que debe ser publicada en el CDB). Finalmente, la Dispatch Unit toma los resultados publicados en el CDB para actualizar el banco de registros, actualizar el RST y seleccionar los datos que serán enviados como operandos a las colas de ejecución, una vez resuelta esta etapa, se encarga de ensamblar las señales y transmitirlas a las colas de ejecución.
La Dispatch Unit es capaz de hacer el renombramiento de registros, decodificar la instrucción y mandar la instrucción a su respectiva cola de ejecución en un solo ciclo. Logra esto mediante la realización de diversas operaciones en paralelo, muchas de las cuáles no serán necesarias para la ejecución correcta de todas las instrucciones pero en la implementación bajo validación se sacrificó el consumo de área y de energía a cambio de un mejor desempeño.

El flujo de operaciones, luego de recibida una instrucción proveniente de la IFQ es el siguiente:

1. Se leen los registros Rs y Rt tanto del banco de registros como de la RST. Se asume que todas las operaciones requieren de estos operandos.
2. Se calculan las direcciones de salto tanto para el caso de una instrucción del tipo Branch como una del tipo Jump. Se asume que todas las instrucciones son del tipo Branch o Jump.
3. A los 16 bits del campo inmediato se les hace la extensión de signo. Se asume que todas las instrucciones son del tipo I.
4. Se decodifica la instrucción.
    a. Dependiendo del tipo de instrucción (resultado de la decodificación), del valor de los estados de los registros leídos (resultado RST) y de los resultados publicados en el CDB (si fuera necesario alguno) se generan y seleccionan las señales que deben de ser propagadas a las colas de ejecución.
    b. En caso de que la instrucción no pueda ser despachada (es decir, que la cola de ejecución destino se encuentre llena) se tienen que retener los resultados de la decodificación.
    c. En caso de que la instrucción si pueda ser despachada se disparan las siguientes operaciones: actualización del RST y se manda a traer la próxima instrucción.
    d. En caso que la instrucción sea una del tipo Branch, se dispara la operación de detención del IFQ hasta que el resultado del Branch sea resuelto.
    e. Cada vez que el CDB publique un nuevo resultado se disparan las siguientes operaciones: actualización del RST y del banco de registros.



Arquitectura del ambiente de prueba.

El propósito del ambiente de pruebas es determinar el adecuado funcionamiento del dispositivo bajo validación (DUV). Y esto puede lograrse si se cumplen con las características mínimas necesarias que son:

- Generación de estímulos.
- Aplicación de los estímulos al Dispositivo bajo Validación.
- Captura de la respuesta.
- Checar la validez de la respuesta.
- Cuantificar el progreso frente a los objetivos completos de verificación.

Los aspectos básicos de nuestro ambiente de pruebas son:

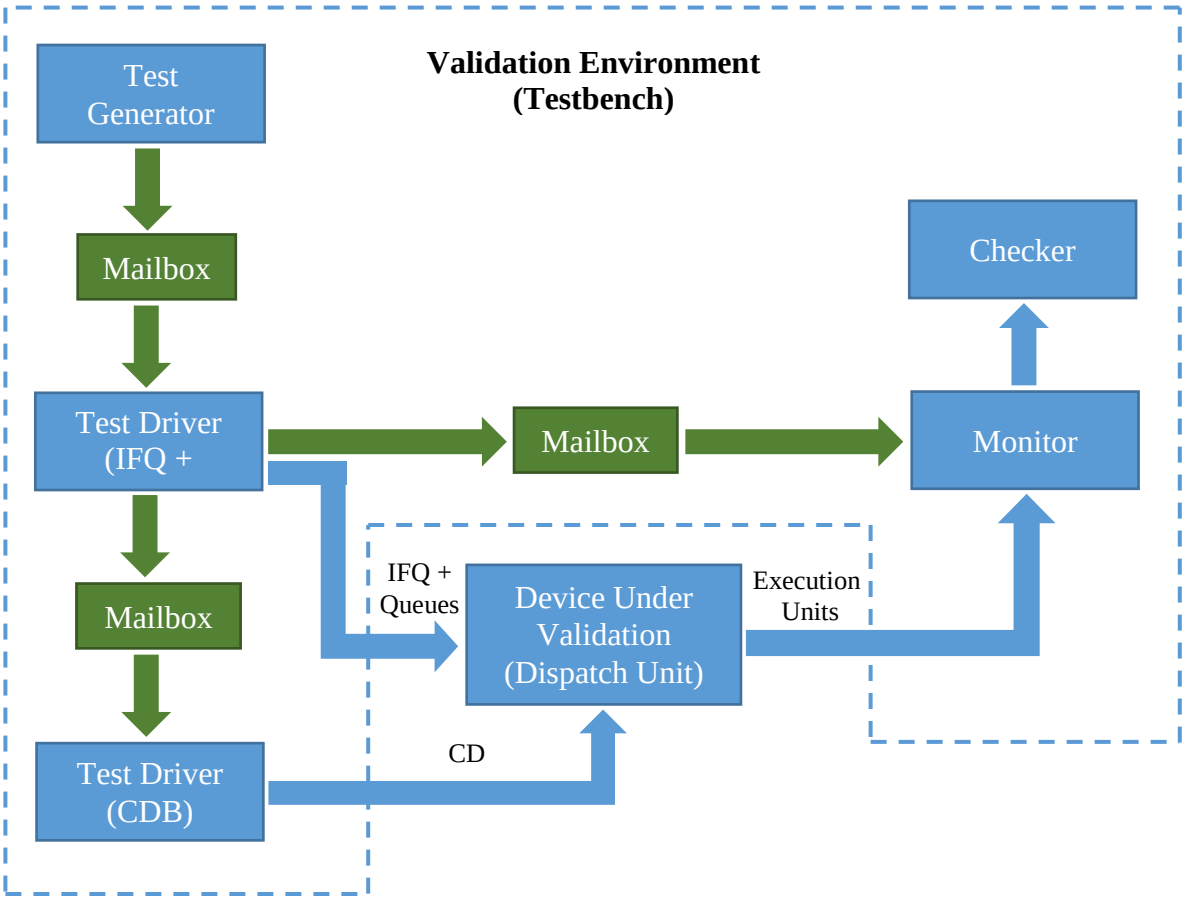
- Generación aleatoria de los estímulos. Y empleo de restricciones para dirigir parcialmente el alcance de los estímulos.
- Estructuración por capas y uso de estructuras (mailboxes) para transferencia de información entre ellas.
- Aspectos comunes para todas las pruebas.
- Pruebas específicas para comprobar ciertos aspectos funcionales.

Los estímulos aleatorios son útiles para encontrar más fácil y rápidamente errores que no se anticipan. La separación en capas de la arquitectura ayuda a mantener la complejidad del ambiente dentro de niveles manejables.

Los bloques que constituyen en general nuestro ambiente de pruebas son:

- Test Generator.
- Test Driver (para las señales provenientes del IFQ).
- Test Driver (para las señales provenientes del CDB).
- Monitor.
- Checker.
- Mailboxes.

A continuación se muestra la arquitectura del ambiente de pruebas que se desarrolló para la Dispatch Unit.



A continuación se listan los módulos de código que forman parte del ambiente de verificación para la Dispatch Unit del procesador MIPS SuperScalar.

1. Testbench (Dispatch_tb.sv)
2. TestDefinitions (TB_defs.sv)
3. TestGenerator (Dispatch_TestGenerator.sv)
4. TestDriver (Dispatch_Driver.sv)
5. TestMonitor (Dispatch_Monitor.sv)
6. TestChecker (Dispatch_Checker.sv)

Código del módulo Top del ambiente de pruebas.

```
//////////////////////////////////////////////////////////////////////////////////
// Authors:      Francisco Delgadillo y Cesar Gómez Cruz
// Create Date:  May, 2017
// Modified Date:
// File Name:    Dispatch_tb.sv
// Description:
//       Test bench for Dispatch unit for super scalar MIPS chip
//
// Revision:     0.1
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
`timescale 1 ps/ 1 ps
`include "./Dispatch Driver.sv"
`include "./Dispatch Monitor.sv"
`include "./Dispatch_TestGenerator.sv"
`include "./TB_defs.sv"

//++++++++++++++++++++++++++++++++++++++++++++++++++
//        Main Test Bench
//++++++++++++++++++++++++++++++++++++++++++++++++++
module Dispatch_tb();

        // TB Signals
        reg clk = 1'b1;
        reg rst = 1'b0;

        //Clock generator with a configurable period
        always #(CLK_PERIOD/2) clk = ~clk;

        // Initial Conditions for Design: reset.
        initial begin
                #(CLK_PERIOD+CLK_PERIOD/5) rst = 1'b1;
                $display("Reset Done!");
        end

        //+++++++++Interface TB connected to clk and rst+++++++
        //TB Interfaces
        intfc_ifq con_ifq(clk, rst);
        intfc_cdb con_cdb(clk, rst);
        intfc_queues con_queues(clk, rst);

        //RTL Interfaces
        iqf_disp_if             iqf_if(clk, rst);               // IF with Instruction Queue Fifo
        disp_issueq_if   issueq_if(clk, rst);     // IF with the Issue Queues
        cdb_if                  cdb_if(clk, rst);               // IF with Common Data Bus

        //+++++++++Instance Driver, Monitor, Generator and Checker+++++++
        Dispatcher_dr    tb_driver;
        Dispatcher_mon   tb_monitor;
        Dispatcher_Testgen tb_testgen;

        //+++++++++Instance mailboxes+++++++
```

```
mailbox e_tests;
mailbox m_tests;
mailbox m_check;

//+++++++++CONNECTIONS FROM TB TO DUT+++++++
// Logic from Dispatch to IQF
assign con_ifq.dispatch_ren = iqf_if.RdEn;
assign con_ifq.Dispatch_Jmp_br_addr = iqf_if.Jmp_Br_Addr;
assign con_ifq.dispatch_jmp = iqf_if.Jmp_Br_Valid;

// Logic from IQF to Dispatch
assign iqf_if.PC_disp = con_ifq.ifetch_pc_plu_four;
assign iqf_if.Instr = con_ifq.ifetch_instruction;
assign iqf_if.Empty = con_ifq.ifetch_empty_flag;

// Common signals for issue queues
assign con_queues.dispatch_rd_tag = issueq_if.Rd_tag;
assign con_queues.dispatch_rs_data = issueq_if.Rs_data;
assign con_queues.dispatch_rs_data_val = issueq_if.Rs_data_val;
assign con_queues.dispatch_rs_tag = issueq_if.Rs_tag;
assign con_queues.dispatch_rt_data = issueq_if.Rt_data;
assign con_queues.dispatch_rt_data_val = issueq_if.Rt_data_val;
assign con_queues.dispatch_rt_tag = issueq_if.Rt_tag;

// Signals to/from integer issue queue
assign con_queues.dispatch_en_integer = issueq_if.En_int;
assign con_queues.dispatch_ALU_opcode = issueq_if.OpCode_int;
assign issueq_if.Int_full = con_queues.issueque_full_integer;

// Signals to/from LdSt issue queue
assign con_queues.dispatch_en_ld_st = issueq_if.En_ld_st;
assign issueq_if.Ld_st_full = con_queues.issueque_full_ld_st;
assign con_queues.dispatch_ld_st_opcode = issueq_if.OpCode_ld_st;
assign con_queues.dispatch_imm_ld_st = issueq_if.Imm_ld_st;

// Signals to/from mult issue queue
assign con_queues.dispatch_en_mul = issueq_if.En_mult;
assign issueq_if.Mult_full = con_queues.issueque_full_mul;

// Signals to/from mult issue queue
assign con_queues.dispatch_en_div = issueq_if.En_div;
assign issueq_if.Div_full = con_queues.issueque_full_div;

//CDB CONNECTIONS
assign cdb_if.Cdb_tag  = con_cdb.Cdb_tag;
assign cdb_if.Cdb_val  = con_cdb.Cdb_valid;
assign cdb_if.Cdb_data = con_cdb.Cdb_data;
assign cdb_if.Cdb_brch = con_cdb.Cdb_branch;
assign cdb_if.Cdb_brch_tkn = con_cdb.Cdb_branch_taken;

//+++++++++ DUT Instance+++++++
dispatch dispatcher (
.clk, .rst,
.tst_addr(),
.tst_reg(),
.iqf_if (iqf_if.dispatch),         // IF with Instruction Queue Fifo
.issueq_if(issueq_if.dispatch),   // IF with the Issue Queues
.cdb_if(cdb_if.slv)               // IF with Common Data Bus
);
```

```
        //++++ Test 2 Run 100 Random supported instructions with no CDB response ++++
        initial @(rst == 1)
        begin
                //Initializing all mailboxes
                e_tests = new();
                m_tests = new();
                m_check = new();

                //Initializing all TB elements
                tb_testgen = new(e_tests, m_tests);
                tb_driver = new(con_ifq.dr, con_cdb.dr, con_queues.dr, e_tests, m_check);
                tb_monitor = new(con_queues.mon, con_cdb.mon, con_ifq.mon, m_tests, m_check);

                //Setting Tests to be run
                tb_testgen.execute_test(RANDOM_NOCDB);
                $display("Test Starting!");

                fork
                        //tb driver.check regfile();
                        tb_driver.send_test();
                        tb_monitor.monitor_test();
                join

                $display("Tests Done!");
                $stop;
        end

endmodule
```

Definiciones generales del ambiente de pruebas

```
//////////////////////////////////////////////////////////////////////
// Authors:      Francisco Delgadillo y Cesar Gómez Cruz
// Create Date:  May, 2017
// Modified Date:
// File Name:    TB_defs.sv
// Description:
//       Defines for TB general environment
//
// Revision:     0.1
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////
`ifndef TB_DEFS
        `define TB_DEFS

         `include "./Source/disp_defs.sv"

        parameter R_INST = 0, I_INST = 1, J_INST = 2, LW_INST = 3, SW_INST = 4;
        parameter B_INST =  5, INVAL_INST = 6;
        parameter NO_TEST = 0, RANDOM_NOCDB = 1, RANDOM_CDB = 2, RANDOM_FULL_QUEUE = 3;
        parameter EMPTY_IFETCH =  4, RANDOM_NOVALID = 5;
        parameter INT_QUEUE = 0, MUL_QUEUE = 1, DIV_QUEUE = 2, LS_QUEUE = 3;

        parameter CLK_PERIOD = 10;
```

```
parameter PASS = 0, FAIL = 1;

//MACRO to Check If randomization failed
`define SV_RAND_CHECK(r) \
        do begin \
                if (!r) begin \
                        $display ("%s:%0d: Randomization  \
                                   Failed!!",`__FILE__,`__LINE__); \
                        $stop; \
                end \
        end while (0)

//Packed Instruction structure
typedef struct packed{
        bit [5:0] opcode;
        bit [4:0] rs;
        bit [4:0] rt;
        bit [4:0] rd;
        bit [4:0] sha;
        bit [5:0] fun;
} inst;

//Class for Decoded Instruction for CDB Responses
class decoded_instr;
        //Instruction fields generated by the driver
        inst                inst_val;
        bit [15:0]          immediate;
        bit [31:0]          sign_ext_imm;
        bit [31:0]          zero_ext_imm;
        bit [25:0]          jmp_addr;
        bit [31:0]          jmp_addr_ext;
        bit                 valid_inst;
        int                 inst_type;
        logic           ifetch_empty_flag;          //INPUT
        logic  [31:0]   ifetch_pc_plu_four;         //INPUT
        logic           issueque_full_integer;              //INPUT
        logic           issueque_full_ld_st;                //INPUT
        logic           issueque_full_mul;          //INPUT
        logic           issueque_full_div;          //INPUT

        //QUEUE generated TAGS and values by Distpatcher
        logic           dispatch_rs_data_val;
        logic           dispatch_rt_data_val;
        logic  [31:0]   dispatch_rs_data;
        logic  [31:0]   dispatch_rt_data;
        logic  [5:0]    dispatch_rs_tag;
        logic  [5:0]    dispatch_rt_tag;
        logic  [5:0]    dispatch_rd_tag;

        //IFQ signals generated by Distpatcher
        logic           dispatch_jmp;
        logic           dispatch_ren;
        logic  [31:0]   Dispatch_Jmp_br_addr;

        //Signals from Dispatcher to Execution Units
        logic               dispatch_en_integer;
        logic               dispatch_en_ld_st;
        logic           dispatch_en_mul;
        logic               dispatch_en_div;
```

```
        //Used to calculate address and immediate
        task calculate_inst();
                immediate = {inst_val.rd, inst_val.sha, inst_val.fun};
                jmp_addr  = {inst_val.rs, inst_val.rt, immediate};
                sign_ext_imm = (immediate[15] == 1) ? \
                                {16'hFFFF,immediate}:{16'h0000,immediate};
                zero_ext_imm = {16'h0000,immediate};
                jmp_addr_ext = {ifetch_pc_plu_four[31:28
                ],jmp_addr, 2'h0};
        endtask;
endclass : decoded_instr

//Class for Random Instruction
class Rand_Instr;

        rand bit [5:0] op_code;
        rand bit [4:0] reg_rs;
        rand bit [4:0] reg_rt;
        rand bit [4:0] reg_rd;
        rand bit [4:0] shamnt;
        rand bit [5:0] funct;

        inst instr;

        function void build_inst();
                instr.opcode = op_code;
                instr.rs = reg_rs;
                instr.rt = reg_rt;
                instr.rd = reg_rd;
                instr.sha = shamnt;
                instr.fun = funct;
        endfunction

        constraint valop {(op_code == OP_SPE  )||
                                        (op_code == OP_ADDI )||
                                        (op_code == OP_ADDIU)||
                                        (op_code == OP_ANDI )||
                                        (op_code == OP_BEQ  )||
                                        (op_code == OP_BNE  )||
                                        (op_code == OP_J    )||
                                        (op_code == OP_LW   )||
                                        (op_code == OP_LUI  )||
                                        (op_code == OP_ORI  )||
                                        (op_code == OP_SLTI )||
                                        (op_code == OP_SLTIU)||
                                        (op_code == OP_SW   );
                                       }
        constraint invalop {(op_code != OP_SPE  )&&
                                        (op_code != OP_ADDI )&&
                                        (op_code != OP_ADDIU)&&
                                        (op_code != OP_ANDI )&&
                                        (op_code != OP_BEQ  )&&
                                        (op_code != OP_BNE  )&&
                                        (op_code != OP_J    )&&
                                        (op_code != OP_LW   )&&
                                        (op_code != OP_LUI  )&&
                                        (op_code != OP_ORI  )&&
```

```
                                        (op_code != OP_SLTI )&&
                                        (op_code != OP_SLTIU)&&
                                        (op_code != OP_SW   );
                                        }
                constraint valfnt{(funct == FN_ADD)||
                                        (funct == FN_ADDU)||
                                        (funct == FN_AND )||
                                        (funct == FN_NOR )||
                                        (funct == FN_OR  )||
                                        (funct == FN_SLT )||
                                        (funct == FN_SLTU)||
                                        (funct == FN_SUB )||
                                        (funct == FN_SUBU)||
                                        (funct == FN_MULT)||
                                        (funct == FN_DIV );
                                        }
                constraint valregs{reg_rs inside {[2:27]};
                                        reg_rt inside {[2:27]};
                                        reg_rd inside {[2:27]};
                                        }
        endclass: Rand_Instr

`endif
```

Código del módulo TestGenerator.

```
//////////////////////////////////////////////////////////////////////
// Author:              Francisco Delgadillo / Cesar Gómez Cruz
// Create Date:         May, 2017
// Modified Date:
// File Name:           Dispatch_TestGenerator.sv
// Description:
//      Test generator for Distpatch unit using Mailbox
//
// Revision:            0.1
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////

`include "./TB_defs.sv"

//++++++++++++++++++++++++++++++++++++++++++++++++
// Define the driver for IFQ
//++++++++++++++++++++++++++++++++++++++++++++++++
class Dispatcher_Testgen;

        mailbox exe_tests;
        mailbox mon_tests;

        function new(mailbox exe_tests, mailbox mon_tests);
                this.exe_tests = exe_tests;
                this.mon_tests = mon_tests;
        endfunction : new

        task execute_test(int test_to_execute);
                exe_tests.put(test_to_execute);
```

```
            mon_tests.put(test_to_execute);
        endtask : execute_test


endclass
```

Código del TestDriver.

```
////////////////////////////////////////////////////////////////////////////////
// Author:                          Francisco Delgadillo / Cesar Gómez
// Create Date:           May, 2017
// Modified Date:
// File Name:             Dispatch_Driver.sv
// Description:
//                Test bench for Dispatch unit for super scalar MIPS chip
//
// Revision:              0.1
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

`include "./TB_defs.sv"
`include "./Source/disp_defs.sv"

//++++++++++++++++++++++++++++++++++++++++++++++++++
// Define the driver for IFQ
//++++++++++++++++++++++++++++++++++++++++++++++++++
class Dispatcher_dr;

        logic [31:0] PC;
        virtual intfc_cdb.dr cdb_dr;
        virtual intfc_ifq.dr ifq_dr;
        virtual intfc_queues.dr queues_dr;

        Rand_Instr inst;
        decoded_instr curr_inst;
        decoded_instr cdb_inst;
        mailbox e_tests;
        mailbox m_instcheck;
        mailbox m_instructions;
        int test;

        function new(virtual intfc_ifq.dr ifq_dr, virtual intfc_cdb.dr cdb_dr, virtual intfc_queues.dr queues_dr, mailbox e_tests, mailbox m_instcheck);
                this.cdb_dr = cdb_dr;
                this.ifq_dr = ifq_dr;
                this.queues_dr = queues_dr;
                this.e_tests = e_tests;
                this.m_instcheck = m_instcheck;
                PC = 32'h00400004;
                inst = new();
                m_instructions = new();
        endfunction : new

        //TASK WITH INSTRUCTION WITH RANDOM FULL QUEUES
        task send_instr_isr(int cycles_inst, int queue_type);
```

```
bit wait_cycles = 0;
int disabled_cycles = 0;
inst.valop.constraint_mode(1);
inst.invalop.constraint_mode(0);
while(cycles_inst>0) begin
        //Negedge setting all signals for the dispatcher to read them on posedge
        @(negedge queues_dr.clk);
        if((ifq_dr.dispatch_ren) || (disabled_cycles>10)) begin
                //Store instruction in mailbox to send Cdb response
                curr_inst = new();
                if(queue_type == INT_QUEUE) begin
                        `SV_RAND_CHECK (inst.randomize() with {op_code == OP_SPE; funct == FN_ADD;});
                        $display ("%s:%0d:%0d ps: Generating new Instruction for INT QUEUE!",`__FILE__,`__LINE__, $time);
                        if (disabled_cycles<10) begin
                                curr_inst.issueque_full_integer = 1'b1;
                        end else begin
                                curr_inst.issueque_full_integer = 1'b0;
                        end
                        curr_inst.issueque_full_ld_st = 1'b0;
                        curr_inst.issueque_full_mul = 1'b0;
                        curr_inst.issueque_full_div = 1'b0;
                        curr_inst.inst_type = INT_QUEUE;
                end else if(queue_type == MUL_QUEUE) begin
                        `SV_RAND_CHECK (inst.randomize() with {op_code == OP_SPE; funct == FN_MULT;});
                        $display ("%s:%0d:%0d ps: Generating new Instruction for MULT QUEUE!",`__FILE__,`__LINE__, $time);
                        curr_inst.issueque_full_integer = 1'b0;
                        curr_inst.issueque_full_ld_st = 1'b0;
                        if (disabled_cycles<10) begin
                                curr_inst.issueque_full_mul = 1'b1;
                        end else begin
                                curr_inst.issueque_full_mul = 1'b0;
                        end
                        curr_inst.issueque_full_div = 1'b0;
                        curr_inst.inst_type = MUL_QUEUE;
                end else if(queue_type == DIV_QUEUE) begin
                        `SV_RAND_CHECK (inst.randomize() with {op_code == OP_SPE; funct == FN_DIV;});
                        $display ("%s:%0d:%0d ps: Generating new Instruction for DIV QUEUE!",`__FILE__,`__LINE__, $time);
                        curr_inst.issueque_full_integer = 1'b0;
                        curr_inst.issueque_full_ld_st = 1'b0;
                        curr_inst.issueque_full_mul = 1'b0;
                        if (disabled_cycles<10) begin
                                curr_inst.issueque_full_div = 1'b1;
                        end else begin
                                curr_inst.issueque_full_div = 1'b0;
                        end
                        curr_inst.inst_type = DIV_QUEUE;
                end else if(queue_type == LS_QUEUE) begin
                        `SV_RAND_CHECK (inst.randomize() with {op_code == OP_LW;});
                        $display ("%s:%0d:%0d ps: Generating new Instruction for LW/SW QUEUE!",`__FILE__,`__LINE__, $time);
                        curr_inst.issueque_full_integer = 1'b0;
                        if (disabled_cycles<10) begin
                                curr_inst.issueque_full_ld_st = 1'b1;
                        end else begin
                                curr_inst.issueque_full_ld_st = 1'b0;
                        end
                        curr_inst.issueque_full_mul = 1'b0;
                        curr_inst.issueque_full_div = 1'b0;
                        curr_inst.inst_type = LS_QUEUE;
                end else begin
                        cycles_inst = 0;
```

```
                $display ("%s:%0d:%0d ps: ERROR no QUEUE supported!! please select a supported QUEUE!",`__FILE__,`__LINE__, $time);
        end
        //Build the random instruction before sending it
        PC = PC + 32'h4;
        inst.build_inst();
        curr_inst.inst_val = inst.instr;
        curr_inst.valid_inst = 1;
        curr_inst.ifetch_empty_flag = 0;
        curr_inst.ifetch_pc_plu_four = PC;

        wait_cycles = 0;
        cycles_inst = cycles_inst - 1;
end else begin
        $display ("%s:%0d:%0d ps: SAVING instruction for CDB & Monitor Response: %h",`__FILE__,`__LINE__, $time, curr_inst.inst_val);
        m_instructions.put(curr_inst);
        m_instcheck.put(curr_inst);
        wait_cycles = 1;
        disabled_cycles = disabled_cycles + 1;
end
//Driving Quueues full signals
queues_dr.issueque_full_integer = curr_inst.issueque_full_integer;
queues_dr.issueque_full_ld_st = curr_inst.issueque_full_ld_st;
queues_dr.issueque_full_mul = curr_inst.issueque_full_mul;
queues_dr.issueque_full_div = curr_inst.issueque_full_div;

#(CLK_PERIOD/10);
//Driving IFQ signals on posedge clk
ifq_dr.ifetch_instruction = curr_inst.inst_val;
ifq_dr.ifetch_pc_plu_four = curr_inst.ifetch_pc_plu_four;

//Driving fetch empty flag
ifq_dr.ifetch_empty_flag = curr_inst.ifetch_empty_flag;

//Posedge + some time reading values from the dispatcher
@(posedge queues_dr.clk);
#(CLK_PERIOD/10);
//Saving all values generated by the instructions decode
curr_inst.dispatch_rs_data_val = queues_dr.dispatch_rs_data_val;
curr_inst.dispatch_rt_data_val = queues_dr.dispatch_rt_data_val;
curr_inst.dispatch_rs_data = queues_dr.dispatch_rs_data;
curr_inst.dispatch_rt_data = queues_dr.dispatch_rt_data;
curr_inst.dispatch_rs_tag = queues_dr.dispatch_rs_tag;
curr_inst.dispatch_rt_tag = queues_dr.dispatch_rt_tag;
curr_inst.dispatch_rd_tag = queues_dr.dispatch_rd_tag;
curr_inst.dispatch_jmp = ifq_dr.dispatch_jmp;
curr_inst.dispatch_ren = ifq_dr.dispatch_ren;
curr_inst.Dispatch_Jmp_br_addr = ifq_dr.Dispatch_Jmp_br_addr;
//
curr_inst.dispatch_en_integer = queues_dr.dispatch_en_integer;
curr_inst.dispatch_en_ld_st = queues_dr.dispatch_en_ld_st;
curr_inst.dispatch_en_mul = queues_dr.dispatch_en_mul;
curr_inst.dispatch_en_div = queues_dr.dispatch_en_div;
curr_inst.calculate_inst();

if(ifq_dr.dispatch_ren && !wait_cycles) begin
        $display ("%s:%0d:%0d ps: SAVING instruction for CDB & Monitor Response: %h",`__FILE__,`__LINE__, $time, curr_inst.inst_val);
        m_instructions.put(curr_inst);
```

```
                        m_instcheck.put(curr_inst);
                        wait_cycles = 1;
                end
        end

endtask:send_instr_isr

task ifq_empty(int cycles_inst);

        int third = cycles_inst/3;
        inst.valop.constraint_mode(1);
        inst.invalop.constraint_mode(0);


        while(cycles_inst > 0) begin
                @(negedge queues_dr.clk);
                `SV_RAND_CHECK (inst.randomize() with {(op_code != OP_BNE); (op_code != OP_BEQ);});
                //Build the random instruction before sending it
                $display ("%s:%0d:%0d ps: Generating full random Valid instruction number %0d",`__FILE__,`__LINE__, $time, cycles_inst);
                PC = PC + 32'h4;
                inst.build_inst();

                //Store instruction in mailbox to send Cdb response
                curr_inst = new();
                curr_inst.inst_val = inst.instr;
                curr_inst.valid_inst = 1'h1;

                if ((cycles_inst > third) && (cycles_inst < 2*third))
                        begin
                                curr_inst.ifetch_empty_flag = 1'h1;
                        end
                else
                        begin
                                curr_inst.ifetch_empty_flag = 1'h0;
                        end

                curr_inst.ifetch_pc_plu_four = PC;
                curr_inst.issueque_full_integer = 1'h0;
                curr_inst.issueque_full_ld_st = 1'h0;
                curr_inst.issueque_full_mul = 1'h0;
                curr_inst.issueque_full_div = 1'h0;

                //Driving IFQ signals on posedge clk
                ifq_dr.ifetch_instruction = curr_inst.inst_val;
                ifq_dr.ifetch_pc_plu_four = curr_inst.ifetch_pc_plu_four;

                //Driving fetch empty flag
                ifq_dr.ifetch_empty_flag = curr_inst.ifetch_empty_flag;
                //Driving Quueues full signals
                queues_dr.issueque_full_integer = curr_inst.issueque_full_integer;
                queues_dr.issueque_full_ld_st = curr_inst.issueque_full_ld_st;
                queues_dr.issueque_full_mul = curr_inst.issueque_full_mul;
                queues_dr.issueque_full_div = curr_inst.issueque_full_div;

                //Posedge + some time reading values from the dispatcher
                @(posedge queues_dr.clk);
                #(CLK_PERIOD/10);
                //Saving all values generated by the instructions decode
                curr_inst.dispatch_rs_data_val = queues_dr.dispatch_rs_data_val;
                curr_inst.dispatch_rt_data_val = queues_dr.dispatch_rt_data_val;
```

```
                curr_inst.dispatch_rs_data = queues_dr.dispatch_rs_data;
                curr_inst.dispatch_rt_data = queues_dr.dispatch_rt_data;
                curr_inst.dispatch_rs_tag = queues_dr.dispatch_rs_tag;
                curr_inst.dispatch_rt_tag = queues_dr.dispatch_rt_tag;
                curr_inst.dispatch_rd_tag = queues_dr.dispatch_rd_tag;
                curr_inst.dispatch_jmp = ifq_dr.dispatch_jmp;
                curr_inst.dispatch_ren = ifq_dr.dispatch_ren;
                curr_inst.Dispatch_Jmp_br_addr = ifq_dr.Dispatch_Jmp_br_addr;
                //
                curr_inst.dispatch_en_integer = queues_dr.dispatch_en_integer;
                curr_inst.dispatch_en_ld_st = queues_dr.dispatch_en_ld_st;
                curr_inst.dispatch_en_mul = queues_dr.dispatch_en_mul;
                curr_inst.dispatch_en_div = queues_dr.dispatch_en_div;
                curr_inst.calculate_inst();

                m_instcheck.put(curr_inst);

                cycles_inst = cycles_inst - 1;
        end
endtask: ifq_empty

//TASK WITH INSTRUCTION SEND
task send_instr(int cycles_inst, bit valid);
        bit wait_br = 0;
        if(valid) begin
                inst.valop.constraint_mode(1);
                inst.invalop.constraint_mode(0);
        end else begin
                inst.valop.constraint_mode(0);
                inst.invalop.constraint_mode(1);
        end

        while(cycles_inst>0) begin

                //Negedge setting all signals for the dispatcher to read them on posedge
                @(negedge queues_dr.clk);
                `SV_RAND_CHECK (inst.randomize());
                if(ifq_dr.dispatch_ren) begin
                        //Build the random instruction before sending it
                        $display ("%s:%0d:%0d ps: Generating full random Valid instruction number %0d",`__FILE__,`__LINE__, $time, cycles_inst);
                        PC = PC + 32'h4;
                        inst.build_inst();

                        //Store instruction in mailbox to send Cdb response
                        curr_inst = new();
                        curr_inst.inst_val = inst.instr;
                        curr_inst.valid_inst = valid;
                        curr_inst.ifetch_empty_flag = 1'h0;
                        curr_inst.ifetch_pc_plu_four = PC;
                        curr_inst.issueque_full_integer = 1'h0;
                        curr_inst.issueque_full_ld_st = 1'h0;
                        curr_inst.issueque_full_mul = 1'h0;
                        curr_inst.issueque_full_div = 1'h0;

                        if (inst.instr.opcode == OP_SPE) begin
                                curr_inst.inst_type = R_INST;
                                $display ("%s:%0d:%0d ps: Generating R_type Valid instruction opcode: 0x%h",`__FILE__,`__LINE__, $time, inst.instr.opcode);
```

```
        end else if (inst.instr.opcode == OP_J) begin
                curr_inst.inst_type = J_INST;
                $display ("%s:%0d:%0d ps: Generating J_type Valid instruction opcode: 0x%h",`__FILE__,`__LINE__, $time, inst.instr.opcode);
        end else if ((inst.instr.opcode == OP_BEQ)||(inst.instr.opcode== OP_BNE)) begin
                curr_inst.inst_type = B_INST;
                $display ("%s:%0d:%0d ps: Generating B_type Valid instruction opcode: 0x%h",`__FILE__,`__LINE__, $time, inst.instr.opcode);
        end else if (inst.instr.opcode == OP_LW) begin
                curr_inst.inst_type = LW_INST;
                $display ("%s:%0d:%0d ps: Generating LW_type Valid instruction opcode: 0x%h",`__FILE__,`__LINE__, $time, inst.instr.opcode);
        end else if (inst.instr.opcode== OP_SW) begin
                curr_inst.inst_type = SW_INST;
                $display ("%s:%0d:%0d ps: Generating SW_type Valid instruction opcode: 0x%h",`__FILE__,`__LINE__, $time, inst.instr.opcode);
        end else begin
                curr_inst.inst_type = I_INST;
                $display ("%s:%0d:%0d ps: Generating I_type Valid instruction opcode: 0x%h",`__FILE__,`__LINE__, $time, inst.instr.opcode);
        end
        wait_br = 0;
        cycles_inst = cycles_inst - 1;
end     else begin
        $display ("%s:%0d:%0d ps: Waiting for Branch Response value %h!!",`__FILE__,`__LINE__, $time, wait_br);
        if (!wait_br) begin
                $display ("%s:%0d:%0d ps: SAVING instruction for CDB & Monitor Response: %h",`__FILE__,`__LINE__, $time, curr_inst.inst_val);
                m_instructions.put(curr_inst);
                m_instcheck.put(curr_inst);
        end
        wait_br = 1;
end

//Driving IFQ signals on posedge clk
ifq_dr.ifetch_instruction = curr_inst.inst_val;
ifq_dr.ifetch_pc_plu_four = curr_inst.ifetch_pc_plu_four;

//Driving fetch empty flag
ifq_dr.ifetch_empty_flag = curr_inst.ifetch_empty_flag;
//Driving Quueues full signals
queues_dr.issueque_full_integer = curr_inst.issueque_full_integer;
queues_dr.issueque_full_ld_st = curr_inst.issueque_full_ld_st;
queues_dr.issueque_full_mul = curr_inst.issueque_full_mul;
queues_dr.issueque_full_div = curr_inst.issueque_full_div;

//Posedge + some time reading values from the dispatcher
@(posedge queues_dr.clk);
#(CLK_PERIOD/10);
//Saving all values generated by the instructions decode
curr_inst.dispatch_rs_data_val = queues_dr.dispatch_rs_data_val;
curr_inst.dispatch_rt_data_val = queues_dr.dispatch_rt_data_val;
curr_inst.dispatch_rs_data = queues_dr.dispatch_rs_data;
curr_inst.dispatch_rt_data = queues_dr.dispatch_rt_data;
curr_inst.dispatch_rs_tag = queues_dr.dispatch_rs_tag;
curr_inst.dispatch_rt_tag = queues_dr.dispatch_rt_tag;
curr_inst.dispatch_rd_tag = queues_dr.dispatch_rd_tag;
curr_inst.dispatch_jmp = ifq_dr.dispatch_jmp;
curr_inst.dispatch_ren = ifq_dr.dispatch_ren;
curr_inst.Dispatch_Jmp_br_addr = ifq_dr.Dispatch_Jmp_br_addr;
//
curr_inst.dispatch_en_integer = queues_dr.dispatch_en_integer;
curr_inst.dispatch_en_ld_st = queues_dr.dispatch_en_ld_st;
curr_inst.dispatch_en_mul = queues_dr.dispatch_en_mul;
curr_inst.dispatch_en_div = queues_dr.dispatch_en_div;
curr_inst.calculate_inst();
```

```
                if(ifq_dr.dispatch_ren && !wait_br) begin
                        $display ("%s:%0d:%0d ps: SAVING instruction for CDB & Monitor Response: %h",`__FILE__,`__LINE__, $time, curr_inst.inst_val);
                        m_instructions.put(curr_inst);
                        m_instcheck.put(curr_inst);
                        if (curr_inst.inst_type == B_INST) wait_br = 1;
                end
        end

endtask: send_instr

//TASK WITH CDB RESPONSE for all instructions
task send_cdb_response(int cycles_cdb);
        cdb_dr.Cdb_valid = 1'h0;
        cdb_dr.Cdb_tag = 6'h0;
        cdb_dr.Cdb_data = 32'h0;
        cdb_dr.Cdb_branch = 1'h0;
        cdb_dr.Cdb_branch_taken = 1'h0;
        cdb_dr.Cdb_valid = 0;
        //Waiting time before sending the first CDB Response emulating the processing time
        #(CLK_PERIOD);
        while(cycles_cdb>0) begin
                #(CLK_PERIOD);
                if(m_instructions.try_get(cdb_inst)) begin
                        $display ("%s:%0d: Cycle %0d for CDB Response",`__FILE__,`__LINE__, cycles_cdb);
                        //Negedge setting all signals for the dispatcher to read them on posedge
                        @(negedge queues_dr.clk);
                        if (cdb_inst.inst_type == I_INST) begin
                                $display ("%s:%0d:%0d ps: GET NEW INSTR Type I Sending Response!",`__FILE__,`__LINE__, $time);
                                cdb_dr.Cdb_valid = 1'h1;
                                cdb_dr.Cdb_tag = cdb_inst.dispatch_rt_tag;
                                cdb_dr.Cdb_data = {26'h0, cdb_inst.dispatch_rt_tag};
                                cdb_dr.Cdb_branch = 1'h0;
                                cdb_dr.Cdb_branch_taken = 1'h0;
                        end else if (cdb_inst.inst_type == R_INST) begin
                                $display ("%s:%0d:%0d ps: GET NEW INSTR Type R Sending Response!",`__FILE__,`__LINE__, $time);
                                cdb_dr.Cdb_valid = 1'h1;
                                cdb_dr.Cdb_tag = cdb_inst.dispatch_rd_tag;
                                cdb_dr.Cdb_data = {26'h0, cdb_inst.dispatch_rd_tag};
                                cdb_dr.Cdb_branch = 1'h0;
                                cdb_dr.Cdb_branch_taken = 1'h0;
                        end else if (cdb_inst.inst_type == LW_INST) begin
                                $display ("%s:%0d:%0d ps: GET NEW INSTR Type LoadWord Sending Response!",`__FILE__,`__LINE__, $time);
                                cdb_dr.Cdb_valid = 1'h1;
                                cdb_dr.Cdb_tag = cdb_inst.dispatch_rt_tag;
                                cdb_dr.Cdb_data = {26'h0, cdb_inst.dispatch_rt_tag};
                                cdb_dr.Cdb_branch = 1'h0;
                                cdb_dr.Cdb_branch_taken = 1'h0;
                        end else if (cdb_inst.inst_type == B_INST)begin
                                $display ("%s:%0d:%0d ps: GET NEW INSTR Type Branch Sending Response!",`__FILE__,`__LINE__, $time);
                                cdb_dr.Cdb_valid = 1'h0;
                                cdb_dr.Cdb_tag = cdb_inst.dispatch_rd_tag;
                                cdb_dr.Cdb_data = {26'h0, cdb_inst.dispatch_rd_tag};
                                cdb_dr.Cdb_branch = 1'h1;
                                cdb_dr.Cdb_branch_taken = $random();
                        end else begin
                                $display ("%s:%0d:%0d ps: GET NEW INSTR Type J or SW and NOT Sending Response!",`__FILE__,`__LINE__, $time);
```

```
                                cdb_dr.Cdb_valid = 1'h0;
                                cdb_dr.Cdb_tag = 6'h0;
                                cdb_dr.Cdb_data = 32'h0;
                                cdb_dr.Cdb_branch = 1'h0;
                                cdb_dr.Cdb_branch_taken = 1'h0;
                        end
                        cycles_cdb = cycles_cdb - 1;
                end else begin
                        $display ("%s:%0d:%0d ps: NOT message posted Sending void Response!",`__FILE__,`__LINE__, $time);
                        cdb_dr.Cdb_valid = 1'h0;
                        cdb_dr.Cdb_tag = 6'h0;
                        cdb_dr.Cdb_data = 32'h0;
                        cdb_dr.Cdb_branch = 1'h0;
                        cdb_dr.Cdb_branch_taken = 1'h0;
                end
        end
endtask : send_cdb_response

//TASK WITH CDB RESPONSE for brch only
task send_cdb_br();
        int cycles_br = 10;
        cdb_dr.Cdb_valid = 1'h0;
        cdb_dr.Cdb_tag = 6'h0;
        cdb_dr.Cdb_data = 32'h0;
        cdb_dr.Cdb_branch = 1'h0;
        cdb_dr.Cdb_branch_taken = 1'h0;
        cdb_dr.Cdb_valid = 0;
        #(CLK_PERIOD);
        while(cycles_br>0) begin
                if(m_instructions.try_get(cdb_inst)) begin
                        $display ("%s:%0d:%0d ps: Cycle %0d for CDB Response",`__FILE__,`__LINE__, $time, cycles_br);
                        //Negedge setting all signals for the dispatcher to read them on posedge
                        @(negedge queues_dr.clk);
                        if (cdb_inst.inst_type == B_INST)begin
                                $display ("%s:%0d:%0d ps: GET NEW INSTR Type Branch Sending Response!",`__FILE__,`__LINE__, $time);
                                cdb_dr.Cdb_valid = 1'h0;
                                cdb_dr.Cdb_tag = cdb_inst.dispatch_rd_tag;
                                cdb_dr.Cdb_data = {26'h0, cdb_inst.dispatch_rd_tag};
                                cdb_dr.Cdb_branch = 1'h1;
                                cdb_dr.Cdb_branch_taken = $random();
                        end else begin
                                $display ("%s:%0d:%0d ps: GET NEW INSTR Type X NOT Sending Response!",`__FILE__,`__LINE__, $time);
                                cdb_dr.Cdb_valid = 1'h0;
                                cdb_dr.Cdb_tag = 6'h0;
                                cdb_dr.Cdb_data = 32'h0;
                                cdb_dr.Cdb_branch = 1'h0;
                                cdb_dr.Cdb_branch_taken = 1'h0;
                        end
                        cycles_br = 10;
                end else begin
                        $display ("%s:%0d:%0d ps: NOT message posted Sending void Response!",`__FILE__,`__LINE__, $time);
                        cdb_dr.Cdb_valid = 1'h0;
                        cdb_dr.Cdb_tag = 6'h0;
                        cdb_dr.Cdb_data = 32'h0;
                        cdb_dr.Cdb_branch = 1'h0;
                        cdb_dr.Cdb_branch_taken = 1'h0;
                        cycles_br = cycles_br - 1;
                end
                #(CLK_PERIOD);
        end
```

```
endtask : send_cdb_br

//TASK WITH ALL TEST DEFINITION
task send_test();
        e_tests.get(test);
        case(test)
        RANDOM_NOCDB: begin
                $display("####Random 1000 Valid Instruction without CDB Response: Test Starting!####");
                fork
                        send_instr(1000,1);
                        send_cdb_br();
                join
                $display("####Random 1000 Valid Instruction without CDB Response: Test Finished!####");
        end
        RANDOM_CDB: begin
                $display("####Random 1000 Valid Instruction with CDB Response: Test Starting!####");
                fork
                        send_instr(1000,1);
                        send_cdb_response(1000);
                join
                $display("####Random 1000 Valid Instruction with CDB Response: Test Finished!####");
        end
        RANDOM_FULL_QUEUE: begin
                $display("#### Random execution fetch queue full for all units:  Test Starting!####");
                $display("----------SENDING INT FULL QUEUE TEST---------------");
                fork
                        send_instr_isr(40, INT_QUEUE);
                        send_cdb_br();
                join
                        $display("----------SENDING MULT FULL QUEUE TEST---------------");
                        send_instr_isr(40, MUL_QUEUE);
                        $display("----------SENDING DIV FULL QUEUE TEST---------------");
                        send_instr_isr(40, DIV_QUEUE);
                        $display("----------SENDING LOAD/STORE FULL QUEUE TEST---------------");
                        send_instr_isr(40, LS_QUEUE);
                $display("#### Random execution fetch queue full for all units:  Test Finished!####");
        end
        EMPTY_IFETCH: begin
                $display("#### Test ifetch empty signal toggling:  Test Starting!####");
                fork
                        ifq_empty(18);
                        send_cdb_br();
                join
                $display("#### Test ifetch empty signal toggling:  Test Finished!####");
        end
        RANDOM_NOVALID: begin
                $display("#### Random 1000 Non Valid Instructions:  Test Starting!####");
                fork
                        send_instr(1000,0);
                        send_cdb_br();
                join
                $display("#### Random 1000 Non Valid Instructions:  Test Finished!####");
        end
        default: begin
                $display("ERROR!:No Test Case defined for that selection!");
                $stop;
                end
```

```
                endcase
        endtask: send_test
endclass
```

Código del módulo TestMonitor.

```
////////////////////////////////////////////////////////////////////////
// Author:       Francisco Delgadillo / Cesar Gómez Cruz
// Create Date:  March 19, 2017
// Modified Date:
// File Name:    Dispatch_Monitor.sv
// Description:
//       Test bench for Dispatch unit for super scalar MIPS chip
//
// Revision:     0.1
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////

`include "./TB_defs.sv"
`include "./Dispatch_Checker.sv"

//++++++++++++++++++++++++++++++++++++++++++++++++
// Define the MONITOR for Dispatcher
//++++++++++++++++++++++++++++++++++++++++++++++++
class Dispatcher_mon;

        virtual intfc_queues.mon queues_mon;
        virtual intfc_cdb.mon cdb_mon;
        virtual intfc_ifq.mon ifq_mon;

        Dispatcher_checker tb_checker;
        decoded_instr d_inst;
        mailbox m_tests;
        mailbox m_instcheck;
        mailbox cdb_responses;

        function new(virtual intfc_queues.mon queues_mon, virtual intfc_cdb.mon cdb_mon, virtual intfc_ifq.mon ifq_mon, mailbox m_tests, mailbox m_instcheck);
                this.queues_mon = queues_mon;
                this.cdb_mon = cdb_mon;
                this.ifq_mon = ifq_mon;
                this.m_tests = m_tests;
                this.m_instcheck = m_instcheck;
                cdb_responses = new();
                tb_checker = new(cdb_responses);
        endfunction

        //
        task monitor_inst();
                int cycles = 10;
                while (cycles > 0) begin
                  if(m_instcheck.try_get(d_inst)) begin
                          if(tb_checker.check_ds_response(d_inst)) begin
                                  $display("%s:%0d:%0d ps: ERROR!: Instruction Logic RESPONSE FAILED!!!!",`__FILE__,`__LINE__,$time);
                                  $stop;
                          end
```

```
                        cycles = 10;
                    end else begin
                        $display("%s:%0d:%0d ps: Instruction not received yet wait for another %d clk cycles",`__FILE__,`__LINE__, $time, cycles);
                        cycles = cycles - 1;
                    end
                    #(CLK_PERIOD);
            end
        endtask: monitor_inst

        task monitor_ifq_empty();
            int cycles = 10;
            while (cycles > 0) begin
                if(m_instcheck.try_get(d_inst)) begin
                    if (d_inst.ifetch_empty_flag == 1'b0)
                        begin
                        if(tb_checker.check_ds_response(d_inst)) begin
                            $display("%s:%0d:%0d ps: ERROR!: Instruction Logic RESPONSE FAILED!!!!",`__FILE__,`__LINE__,$time);
                            $stop;
                        end
                        end
                    else
                        begin
                        if(tb_checker.check_ifq_empty(d_inst)) begin
                            $display("%s:%0d:%0d ps: ERROR!: Instruction Logic RESPONSE FAILED!!!!",`__FILE__,`__LINE__,$time);
                            $stop;
                        end
                        end
                    cycles = 10;
                end else begin
                    $display("%s:%0d:%0d ps: Instruction not received yet wait for another %d clk cycles",`__FILE__,`__LINE__, $time, cycles);
                    cycles = cycles - 1;
                end
                #(CLK_PERIOD);
            end
        endtask: monitor_ifq_empty

        //NICE TO HAVE!!!! TO CHECK Register File coherency
        task monitor_cdb();
        //Pull all signals from CDB interface so no packet is lost
            //cdb_mon

        endtask: monitor_cdb

        task monitor_queues();
            int cycles = 20;
            while (cycles > 0) begin
                if(m_instcheck.try_get(d_inst)) begin
                    if((d_inst.issueque_full_ld_st == 1'b1)||(d_inst.issueque_full_div == 1'b1)||(d_inst.issueque_full_mul == 1'b1)||
(d_inst.issueque_full_integer == 1'b1)) begin
                        if(tb_checker.check_queue_full(d_inst))begin
                            $display("%s:%0d:%0d ps: ERROR!: Instruction Logic RESPONSE FAILED!!!!",`__FILE__,`__LINE__,$time);
                            $stop;
                        end
                    end else begin
                        if(tb_checker.check_int_queue(d_inst))begin
                            $display("%s:%0d:%0d ps: ERROR!: Instruction Logic RESPONSE FAILED!!!!",`__FILE__,`__LINE__,$time);
                            $stop;
```

```
                                        end
                                end
                                cycles = 10;
                        end else begin
                                $display("%s:%0d:%0d ps: Intruction not recieved yet wait for another %d clk cycles",`__FILE__,`__LINE__, $time, cycles);
                                cycles = cycles - 1;
                        end
                        #(CLK_PERIOD);
                end
        endtask: monitor_queues


        task monitor_test();
                int test;
                m_tests.get(test);
                case(test)
                RANDOM_NOCDB: begin
                        monitor_inst();
                end
                RANDOM_CDB: begin
                        fork
                                monitor_inst();
                                monitor_cdb();
                        join
                end
                RANDOM_FULL_QUEUE: begin
                        monitor_queues();
                end
                EMPTY_IFETCH: begin
                        monitor_ifq_empty();
                end
                RANDOM_NOVALID: begin
                        monitor_inst();
                end
                default: $display("ERROR!:No Test Case to Monitor!");
                endcase

        endtask: monitor_test

endclass
```

Código del módulo TestChecker.

```
//////////////////////////////////////////////////////////////////////
// Author:       Francisco Delgadillo y Cesar Gómez Cruz
// Create Date:  March 19, 2017
// Modified Date:
// File Name:    Dispatch_Checker.sv
// Description:
//       Test bench for Dispatch unit for super scalar MIPS chip
//
// Revision:     0.1
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////

`include "./TB_defs.sv"
```

```
//+++++++++++++++++++++++++++++++++++++++++++++++++
// Define the CHECKER for Dispatcher
//+++++++++++++++++++++++++++++++++++++++++++++++++
class Dispatcher_checker;
        mailbox cdb_responses;

        function new(mailbox cdb_responses);
                this.cdb_responses = cdb_responses;
        endfunction: new

        function bit check_queue_full (decoded_instr d_inst);
                if (~d_inst.dispatch_jmp && ~d_inst.dispatch_ren) begin
                        $display("%s:%0d:%0d ps: Full queue signal handled Correctly!",`__FILE__,`__LINE__, $time);
                end else begin
                        $display("%s:%0d:%0d ps: ERROR!: FULL queue sending bad signaling",`__FILE__,`__LINE__, $time);
                        return FAIL;
                end
                return PASS;
        endfunction      : check_queue_full

        function bit check_ifq_empty(decoded_instr d_inst);
                if (~d_inst.dispatch_en_integer && ~d_inst.dispatch_en_ld_st && ~d_inst.dispatch_en_mul && ~d_inst.dispatch_en_div && ~d_inst.dispatch_jmp) begin
                        $display("%s:%0d:%0d ps: Oki Doki!",`__FILE__,`__LINE__, $time);
                end else begin
                        $display("%s:%0d:%0d ps: ERROR!: IFQ EMPTY FLAG not working!",`__FILE__,`__LINE__, $time);
                        return FAIL;
                end
                return PASS;
        endfunction: check_ifq_empty


        function bit check_br_inst(decoded_instr d_inst);
                if (d_inst.dispatch_en_integer && ~d_inst.dispatch_en_ld_st && ~d_inst.dispatch_en_mul && ~d_inst.dispatch_en_div && ~d_inst.dispatch_jmp) begin
                        $display("%s:%0d:%0d ps: Branch instruction issued to integer queue Correctly!",`__FILE__,`__LINE__, $time);
                end else begin
                        $display("%s:%0d:%0d ps: ERROR!: KNOWN BUG 02: BRANCH instruction issued failed to be sent to INT ALU",`__FILE__,`__LINE__, $time);
                        //return FAIL;
                end
                return PASS;
        endfunction: check_br_inst

        function bit check_int_queue (decoded_instr d_inst);
                if((d_inst.inst_val.opcode != OP_SPE) && (d_inst.inst_val.opcode != OP_ANDI) && (d_inst.inst_val.opcode != OP_ORI))begin
                        if(d_inst.sign_ext_imm != d_inst.dispatch_rt_data) begin
                                $display("%s:%0d:%0d ps: ERROR!: On INT instruction RT data does not match the sign ext immediate sent 0x%h !=
                                        0x%h",`__FILE__,`__LINE__, $time, d_inst.dispatch_rt_data, d_inst.sign_ext_imm );
                                return FAIL;
                        end
                end     else if(d_inst.inst_val.opcode == OP_ANDI) begin
                        if(d_inst.zero_ext_imm != d_inst.dispatch_rt_data) begin
                                $display("%s:%0d:%0d ps: ERROR!: On ANDI instruction RT data does not match the zero ext immediate sent",`__FILE__,`__LINE__, $time);
                                return FAIL;
                        end
                end     else if(d_inst.inst_val.opcode == OP_ORI) begin
                        if(d_inst.zero_ext_imm != d_inst.dispatch_rt_data) begin
                                $display("%s:%0d:%0d ps: ERROR!: On ORI instruction RT data does not match the zero ext immediate sent",`__FILE__,`__LINE__, $time);
                                return FAIL;
```

```
                end
        end
        if (d_inst.dispatch_en_integer && ~d_inst.dispatch_en_ld_st && ~d_inst.dispatch_en_mul && ~d_inst.dispatch_en_div && ~d_inst.dispatch_jmp) begin
                $display("%s:%0d:%0d ps: Integer instruction issued to integer queue Correctly!",`__FILE__,`__LINE__, $time);
        end else begin
                $display("%s:%0d:%0d ps: ERROR!: INT instruction issued failed",`__FILE__,`__LINE__, $time);
                return FAIL;
        end
        return PASS;
endfunction: check_int_queue

function bit check_lui_queue (decoded_instr d_inst);
        if (d_inst.dispatch_en_integer && ~d_inst.dispatch_en_ld_st && ~d_inst.dispatch_en_mul && ~d_inst.dispatch_en_div && ~d_inst.dispatch_jmp) begin
                $display("%s:%0d:%0d ps: LUI instruction issued to integer queue Correctly!",`__FILE__,`__LINE__, $time);
        end else begin
                $display("%s:%0d:%0d ps: ERROR!: KNOWN BUG 01: LUI instruction not correctly supported",`__FILE__,`__LINE__, $time);
                //return FAIL;
        end
        return PASS;
endfunction: check_lui_queue

function bit check_sltui_queue (decoded_instr d_inst);
        if (d_inst.dispatch_en_integer && ~d_inst.dispatch_en_ld_st && ~d_inst.dispatch_en_mul && ~d_inst.dispatch_en_div && ~d_inst.dispatch_jmp) begin
                if(d_inst.sign_ext_imm != d_inst.dispatch_rt_data) begin
                        $display("%s:%0d:%0d ps: ERROR!: KNOWN BUG 03: SLTUI instruction using Zero ext immediate and no Sign ext immediate as spec is written
                                0x%h != 0x%h",`__FILE__,`__LINE__, $time, d_inst.dispatch_rt_data, d_inst.sign_ext_imm);
                        //return FAIL;
                end
                $display("%s:%0d:%0d ps: SLTUI instruction issued to integer queue Correctly!",`__FILE__,`__LINE__, $time);
        end else begin
                $display("%s:%0d:%0d ps: ERROR!:  SLTUI instruction not correctly supported",`__FILE__,`__LINE__, $time);
                return FAIL;
        end
        return PASS;
endfunction: check_sltui_queue

function bit check_mult_queue(decoded_instr d_inst);
        if (~d_inst.dispatch_en_integer && ~d_inst.dispatch_en_ld_st && d_inst.dispatch_en_mul && ~d_inst.dispatch_en_div && ~d_inst.dispatch_jmp) begin
                $display("%s:%0d:%0d ps: MULT Issued to Multiplication queue Correctly!",`__FILE__,`__LINE__, $time);
        end else begin
                $display("%s:%0d:%0d ps: ERROR!: MULT instruction issued failed",`__FILE__,`__LINE__, $time);
                return FAIL;
        end
        return PASS;
endfunction: check_mult_queue

function bit check_div_queue(decoded_instr d_inst);
        if (~d_inst.dispatch_en_integer && ~d_inst.dispatch_en_ld_st && ~d_inst.dispatch_en_mul && d_inst.dispatch_en_div && ~d_inst.dispatch_jmp) begin
                $display("%s:%0d:%0d ps: DIV Issued to Division queue Correctly!",`__FILE__,`__LINE__, $time);
        end else begin
                $display("%s:%0d:%0d ps: ERROR!: DIV instruction issued failed",`__FILE__,`__LINE__, $time);
                return FAIL;
        end
        return PASS;
endfunction: check_div_queue

function bit check_lwsw_queue(decoded_instr d_inst);
        if (~d_inst.dispatch_en_integer && d_inst.dispatch_en_ld_st && ~d_inst.dispatch_en_mul && ~d_inst.dispatch_en_div && ~d_inst.dispatch_jmp) begin
                $display("%s:%0d:%0d ps: LW/SW Issued to Memory queue Correctly!",`__FILE__,`__LINE__, $time);
        end else begin
```

```
                $display("%s:%0d:%0d ps: ERROR!: LW/SW instruction issued failed",`__FILE__,`__LINE__, $time);
                return FAIL;
        end
        return PASS;
endfunction: check_lwsw_queue

function bit check_jmp_queue(decoded_instr d_inst);
        if(d_inst.jmp_addr_ext != d_inst.Dispatch_Jmp_br_addr) begin
                $display("%s:%0d:%0d ps: ERROR!: JMP address incorrect 0x%h != 0x%h",`__FILE__,`__LINE__,
                        $time,d_inst.jmp_addr_ext,d_inst.Dispatch_Jmp_br_addr);
                return FAIL;
        end
        if (~d_inst.dispatch_en_integer && ~d_inst.dispatch_en_ld_st && ~d_inst.dispatch_en_mul && ~d_inst.dispatch_en_div && d_inst.dispatch_jmp) begin
                $display("%s:%0d:%0d ps: JMP instruction executed Correctly!",`__FILE__,`__LINE__, $time);
        end else begin
                $display("%s:%0d:%0d ps: ERROR!: JMP instruction issued failed",`__FILE__,`__LINE__, $time);
                return FAIL;
        end
        return PASS;
endfunction: check_jmp_queue

function bit check_ds_response(decoded_instr d_inst);
if(d_inst.valid_inst) begin
        case(d_inst.inst_val.opcode)
                OP_SPE:
                begin
                        case(d_inst.inst_val.fun)
                                FN_ADD:
                                        begin
                                                $display("%s:%0d:%0d ps: OpCode: %h and the function code for %s",`__FILE__,`__LINE__, $time,
                                                        d_inst.inst_val.opcode, d_inst.inst_val.fun, "ADD");
                                                if (check_int_queue(d_inst)) return FAIL;
                                        end
                                FN_ADDU:
                                        begin
                                                $display("%s:%0d:%0d ps: OpCode: %h and function code for %s",`__FILE__,`__LINE__, $time,
                                                        d_inst.inst_val.opcode, d_inst.inst_val.fun, "ADDU");
                                                if (check_int_queue(d_inst)) return FAIL;
                                        end
                                FN_AND:
                                        begin
                                                $display("%s:%0d:%0d ps: OpCode: %h and function code for %s",`__FILE__,`__LINE__, $time,
                                                        d_inst.inst_val.opcode, d_inst.inst_val.fun, "AND");
                                                if (check_int_queue(d_inst)) return FAIL;
                                        end
                                FN_NOR:
                                        begin
                                                $display("%s:%0d:%0d ps: OpCode: %h and function code for %s",`__FILE__,`__LINE__, $time,
                                                        d_inst.inst_val.opcode, d_inst.inst_val.fun, "NOR");
                                                if (check_int_queue(d_inst)) return FAIL;
                                        end
                                FN_OR:
                                        begin
                                                $display("%s:%0d:%0d ps: OpCode: %h and function code for %s",`__FILE__,`__LINE__, $time,
                                                        d_inst.inst_val.opcode, d_inst.inst_val.fun, "OR");
                                                if (check_int_queue(d_inst)) return FAIL;
                                        end
```

```
                FN_SLT:
                        begin
                                $display("%s:%0d:%0d ps: OpCode: %h and function code for %s",`__FILE__,`__LINE__, $time,
                                        d_inst.inst_val.opcode, d_inst.inst_val.fun, "SLT");
                                if (check_int_queue(d_inst)) return FAIL;
                        end
                FN_SLTU:
                        begin
                                $display("%s:%0d:%0d ps: OpCode: %h and function code for %s",`__FILE__,`__LINE__, $time,
                                        d_inst.inst_val.opcode, d_inst.inst_val.fun, "SLTU");
                                if (check_int_queue(d_inst)) return FAIL;
                        end
                FN_SUB:
                        begin
                                $display("%s:%0d:%0d ps: OpCode: %h and function code for %s",`__FILE__,`__LINE__, $time,
                                        d_inst.inst_val.opcode, d_inst.inst_val.fun, "SUB");
                                if (check_int_queue(d_inst)) return FAIL;
                        end
                FN_SUBU:
                        begin
                                $display("%s:%0d:%0d ps: OpCode: %h and function code for %s",`__FILE__,`__LINE__, $time,
                                        d_inst.inst_val.opcode, d_inst.inst_val.fun, "SUBU");
                                if (check_int_queue(d_inst)) return FAIL;
                        end
                FN_MULT:
                        begin
                                $display("%s:%0d:%0d ps: OpCode: %h and function code for %s",`__FILE__,`__LINE__, $time,
                                d_inst.inst_val.opcode, d_inst.inst_val.fun, "MULT");
                                if (check_mult_queue(d_inst)) return FAIL;
                        end
                FN_DIV:
                        begin
                                $display("%s:%0d:%0d ps: OpCode: %h and function code for %s",`__FILE__,`__LINE__, $time,
                                        d_inst.inst_val.opcode, d_inst.inst_val.fun, "DIV");
                                if (check_div_queue(d_inst)) return FAIL;
                        end
                default:
                        begin
                        $display("%s:%0d:%0d ps: ERROR! OpCode: %h With function %h is an Invalid Instruction!!",`__FILE__,`__LINE__, $time,
                                d_inst.inst_val.opcode, d_inst.inst_val.fun);
                        $stop;
                end
        endcase
        //$display("%s:%0d:%0d ps: The 's' register is: %h",`__FILE__,`__LINE__, $time, d_inst.inst_val.rs);
        //$display("%s:%0d:%0d ps: The 't' register is: %h",`__FILE__,`__LINE__, $time, d_inst.inst_val.rt);
        //$display("%s:%0d:%0d ps: The 'd' register is: %h",`__FILE__,`__LINE__, $time, d_inst.inst_val.rd);
        //$display("%s:%0d:%0d ps: The 'function' code is: %h",`__FILE__,`__LINE__, $time, d_inst.inst_val.fun);
end
OP_ADDI:
        begin
                $display("%s:%0d:%0d ps: OpCode: %h instruction: %s",`__FILE__,`__LINE__, $time, d_inst.inst_val.opcode, "ADDI");
                if (check_int_queue(d_inst)) return FAIL;
                //$display("%s:%0d:%0d ps: The 's' register is: %h",`__FILE__,`__LINE__, $time, d_inst.inst_val.rs);
                //$display("%s:%0d:%0d ps: The 't' register is: %h",`__FILE__,`__LINE__, $time, d_inst.inst_val.rt);
                //$display("%s:%0d:%0d ps: The immediate data is: %h",`__FILE__,`__LINE__, $time, d_inst.immediate);
        end
OP_ADDIU:
        begin
                $display("%s:%0d:%0d ps: OpCode: %h instruction: %s",`__FILE__,`__LINE__, $time, d_inst.inst_val.opcode, "ADDIU");
                if (check_int_queue(d_inst)) return FAIL;
```

```
        end
OP_ANDI:
        begin
                $display("%s:%0d:%0d ps: OpCode: %h instruction: %s",`__FILE__,`__LINE__, $time, d_inst.inst_val.opcode, "ANDI");
                if (check_int_queue(d_inst)) return FAIL;
        end
OP_BEQ:
        begin
                $display("%s:%0d:%0d ps: OpCode: %h instruction: %s",`__FILE__,`__LINE__, $time, d_inst.inst_val.opcode, "BEQ");
                if (check_br_inst(d_inst)) return FAIL;
        end
OP_BNE:
        begin
                $display("%s:%0d:%0d ps: OpCode: %h instruction: %s",`__FILE__,`__LINE__, $time, d_inst.inst_val.opcode, "BNE");
                if (check_br_inst(d_inst)) return FAIL;
        end
OP_J:
        begin
                $display("%s:%0d:%0d ps: OpCode: %h instruction: %s",`__FILE__,`__LINE__, $time, d_inst.inst_val.opcode, "JMP");
                $display("%s:%0d:%0d ps: The JMP address is: %h",`__FILE__,`__LINE__, $time, d_inst.jmp_addr);
                if (check_jmp_queue(d_inst)) return FAIL;
        end
OP_LW:
        begin
                $display("%s:%0d:%0d ps: OpCode: %h instruction: %s",`__FILE__,`__LINE__, $time, d_inst.inst_val.opcode, "LW");
                if (check_lwsw_queue(d_inst)) return FAIL;
        end
OP_LUI:
        begin
                $display("%s:%0d:%0d ps: OpCode: %h instruction: %s",`__FILE__,`__LINE__, $time, d_inst.inst_val.opcode, "LUI");
                if (check_lui_queue(d_inst)) return FAIL;
        end
OP_ORI:
        begin
                $display("%s:%0d:%0d ps: OpCode: %h instruction: %s",`__FILE__,`__LINE__, $time, d_inst.inst_val.opcode, "ORI");
                if (check_int_queue(d_inst)) return FAIL;
        end
OP_SLTI:
        begin
                $display("%s:%0d:%0d ps: The OpCode is: %h and correspond to the instruction: %s",`__FILE__,`__LINE__, $time,
                        d_inst.inst_val.opcode, "SLTI");
                if (check_int_queue(d_inst)) return FAIL;
        end
OP_SLTIU:
        begin
                $display("%s:%0d:%0d ps: OpCode: %h instruction: %s",`__FILE__,`__LINE__, $time, d_inst.inst_val.opcode, "SLTIU");
                if (check_sltui_queue(d_inst)) return FAIL;
        end
OP_SW:
        begin
                $display("%s:%0d:%0d ps: OpCode: %h instruction: %s",`__FILE__,`__LINE__, $time, d_inst.inst_val.opcode, "SW");
                if (check_lwsw_queue(d_inst)) return FAIL;
        end
default:
        begin
                $display("%s:%0d:%0d ps: ERROR! OpCode: %h Correspond to an Invalid Instruction and expecting a valid
                        one!!",`__FILE__,`__LINE__, $time, d_inst.inst_val.opcode);
```

```
                                        return FAIL;
                                end
                endcase
        end else begin
                if (~d_inst.dispatch_en_integer && ~d_inst.dispatch_en_ld_st && ~d_inst.dispatch_en_mul && ~d_inst.dispatch_en_div && ~d_inst.dispatch_jmp) begin
                        $display("%s:%0d:%0d ps: NOT SUPPORTED INST Handled correctly!",`__FILE__,`__LINE__, $time);
                end else begin
                        $display("%s:%0d:%0d ps: ERROR!: Some control line is up even with an invalid instruction",`__FILE__,`__LINE__, $time);
                        return FAIL;
                end
        end
        return PASS;
        endfunction: check_ds_response

endclass
```

Test Plan – Dispatch Unit.

A continuación se relacionan por completo las señales de entrada y salida de la Dispatch Unit hacia y desde la Instruction Fetch Queue (IFQ), las colas de las Unidades de Ejecución (EU Queues) y el Common Data Bus (CDB). Se indican sus nombres, el tipo de señal de la que se trata, el tamaño en bits y los valores considerados como válidos para la generación aleatoria.

Señales entre la IFQ y la DU:

Nombre	Tipo	Tamaño	Valores válidos
ifetch_pc_plus_four	Input (IFQ->DU)	[31:0]	<entre el valor mínimo y el valor máximo de la memoria de instrucciones>
ifetch_instruction	Input (IFQ->DU)	[31:0]	Ver tabla de instrucciones
ifetch_empty_flag	Input (IFQ->DU)	bit	0/1
dispatch_jmp_br_addr	Output (DU -> IFQ)	[31:0]	<entre el valor mínimo y el valor máximo de la memoria de instrucciones >
dispatch_jmp_br	Output (DU -> IFQ)	bit	0/1
dispatch_ren	Output (DU -> IFQ)	bit	0/1

Señales entre la DU y la Integer Execution Unit (IEU):

Nombre	Tipo	Tamaño	Valores válidos
dispatch_opcode	Output (DU -> IEU)	[5:0]	Ver tabla de instrucciones
dispatch_en_integer	Output (DU -> IEU)	bit	0/1
dispatch_rd_tag	Output (DU -> IEU)	[5:0]	Entre 0 y 63
dispatch_rs_data	Output (DU -> IEU)	[31:0]	Entre 0 y $(2^{32}-1)$
dispatch_rt_data	Output (DU -> IEU)	[31:0]	Entre 0 y $(2^{32}-1)$
dispatch_rs_tag	Output (DU -> IEU)	[5:0]	Entre 0 y 63
dispatch_rt_tag	Output (DU -> IEU)	[5:0]	Entre 0 y 63
issueque_full_integer	Input (IEU -> DU)	bit	0/1
dispatch_rs_data_val	Output (DU -> IEU)	bit	0/1
dispatch_rt_data_val	Output (DU -> IEU)	bit	0/1

Señales entre la DU y la Multiplication Execution Unit (MEU):

Nombre	Tipo	Tamaño	Valores válidos
dispatch_en_mul	Output (DU -> IEU)	bit	0/1
dispatch_rd_tag	Output (DU -> IEU)	[5:0]	Entre 0 y 63
dispatch_rs_data	Output (DU -> IEU)	[31:0]	Entre 0 y $(2^{32}-1)$
dispatch_rt_data	Output (DU -> IEU)	[31:0]	Entre 0 y $(2^{32}-1)$
dispatch_rs_tag	Output (DU -> IEU)	[5:0]	Entre 0 y 63
dispatch_rt_tag	Output (DU -> IEU)	[5:0]	Entre 0 y 63
issueque_full_mul	Input (IEU -> DU)	bit	0/1
dispatch_rs_data_val	Output (DU -> IEU)	bit	0/1
dispatch_rt_data_val	Output (DU -> IEU)	bit	0/1

Señales entre la DU y la Division Execution Unit (DEU):

Nombre	Tipo	Tamaño	Valores válidos
Name	Type	Size	Valid Values
dispatch_en_div	Output (DU -> IEU)	bit	0/1
dispatch_rd_tag	Output (DU -> IEU)	[5:0]	Entre 0 y 63
dispatch_rs_data	Output (DU -> IEU)	[31:0]	Entre 0 y (2^{32}-1)
dispatch_rt_data	Output (DU -> IEU)	[31:0]	Entre 0 y (2^{32}-1)
dispatch_rs_tag	Output (DU -> IEU)	[5:0]	Entre 0 y 63
dispatch_rt_tag	Output (DU -> IEU)	[5:0]	Entre 0 y 63
issueque_full_div	Input (IEU -> DU)	bit	0/1
dispatch_rs_data_val	Output (DU -> IEU)	bit	0/1
dispatch_rt_data_val	Output (DU -> IEU)	bit	0/1

Señales entre la DU y la Load/Store Execution Unit (LSEU):

Nombre	Tipo	Tamaño	Valores válidos
dispatch_opcode	Output (DU -> LSEU)	[5:0]	Ver tabla de instrucciones
dispatch_imm_ld_st	Output (DU -> LSEU)	[15:0]	Entre 0 y (2^{16}-1)
dispatch_en_ld_st	Output (DU -> LSEU)	bit	0/1
Token	Output (DU -> LSEU)	[6:0]	Entre 0 y 127
dispatch_rs_data	Output (DU -> LSEU)	[31:0]	Entre 0 y (2^{32}-1)
dispatch_rt_data	Output (DU -> LSEU)	[31:0]	Entre 0 y (2^{32}-1)
dispatch_rs_tag	Output (DU -> LSEU)	[5:0]	Entre 0 y 63
dispatch_rt_tag	Output (DU -> LSEU)	[5:0]	Entre 0 y 63
issueque_full_ld_st	Input (LSEU -> DU)	bit	0/1
dispatch_rs_data_val	Output (DU -> LSEU)	bit	0/1
dispatch_rt_data_val	Output (DU -> LSEU)	bit	0/1

Señales entre la DU y el Common Data Bus (CDB):

Nombre	Tipo	Tamaño	Valores válidos
Name	Type	Size	Valid Values
cdb_tag	Input (CDB ->DU)	[5:0]	Entre 0 y 63
cdb_valid	Input (CDB ->DU)	bit	0/1
cbd_data	Input (CDB ->DU)	[31:0]	Entre 0 y (2^{32}-1)
cdb_branch	Input (CDB ->DU)	bit	0/1
cdb_branch_taken	Input (CDB ->DU)	bit	0/1

Con esta configuración de entradas y salidas de la Dispatch Unit se plantean cinco casos de prueba generales que cubren todos los aspectos funcionales de acuerdo con las especificaciones que fueron entregadas. En todos los casos se hace uso de generación restringida o dirigida de señales aleatorias para los distintos campos de las señales de entrada a la Dispatch Unit.

Primer caso de prueba:
Ejecución de un número determinado de instrucciones válidas aleatorias sin respuesta del CDB.

1. Verificar los Tags sin repetición, la ejecución debe detenerse luego de que los Tags sean consumidos por completo dado que no existe respuesta del CDB.
2. Verificar que las instrucciones han sido enviadas a la correspondiente unidad de ejecución o se ha enviado la correspondiente dirección de salto.
3. En el caso de que se presente una instrucción de salto (J), verificar si es ejecutada incondicionalmente y la ejecución continúa.
4. En el caso de una instrucción de bifurcación (BNE o BEQ), el salto no debe ocurrir dado que no se ha recibido respuesta del CDB.

Segundo caso de prueba:
Ejecución de un número determinado de instrucciones válidas aleatorias con respuesta del CDB.

1. Verificar los Tags sin repetición, la ejecución no debería detenerse debido a que los Tags se están renovando porque existe respuesta del CDB.
2. Verificar que las instrucciones están siendo enviadas a la correspondiente unidad de ejecución o en su defecto, que ha sido enviada la correspondiente dirección de salto.
3. Verificar que los registros internos están siendo actualizados correctamente con las respuestas enviadas por el CDB (para propósitos de validación, el valor del dato será el mismo que el valor del Tag).
4. Verificar que todas las señales son válidas para rs, rt y rd.
5. En el caso de que se presente una instrucción de salto (J), verificar si se ejecuta de forma incondicional. La ejecución debería continuar sin contratiempos.
6. En el caso de que se presente una instrucción de bifurcación (BNE o BEQ), la respuesta del CDB puede ser aleatoriamente tomada o no. La ejecución debería continuar sin contratiempos.

Tercer caso de prueba.
Están activas todas las señales que indican que las colas de instrucciones de cada unidad de ejecución están llenas y no pueden aceptarse más instrucciones enviadas desde la Dispatch Unit.

1. Verificar que la Dispatch Unit se ha detenido.

Cuarto caso de prueba.
Se conmuta la señal que indica que la Instruction Fetch Queue está vacía.

2. Verificar que la Dispatch Unit ha detenido la ejecución de instrucciones hasta que la señal vuelva a nivel bajo.
3. Verificar que la ejecución de instrucciones continua de forma normal luego de que la señal es colocada a nivel bajo.

Quinto caso de prueba.

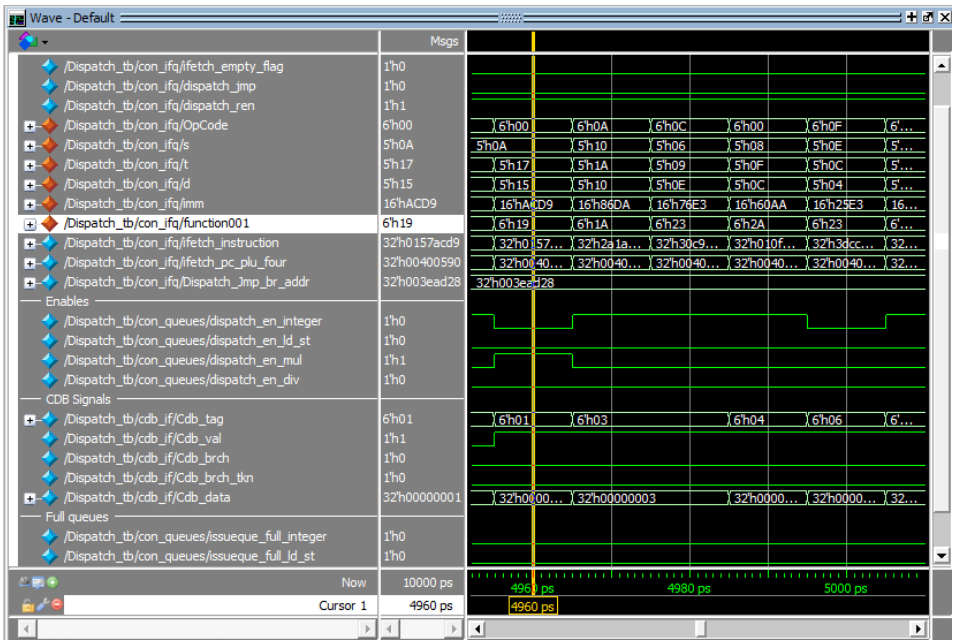
Ejecución de un número determinado de instrucciones aleatorias no válidas.

1. Verificar que ninguna de las instrucciones no válidas sea enviada a alguna de las colas de ejecución.
2. Verificar que ninguno de los registros sea afectado por tratarse de instrucciones no válidas.

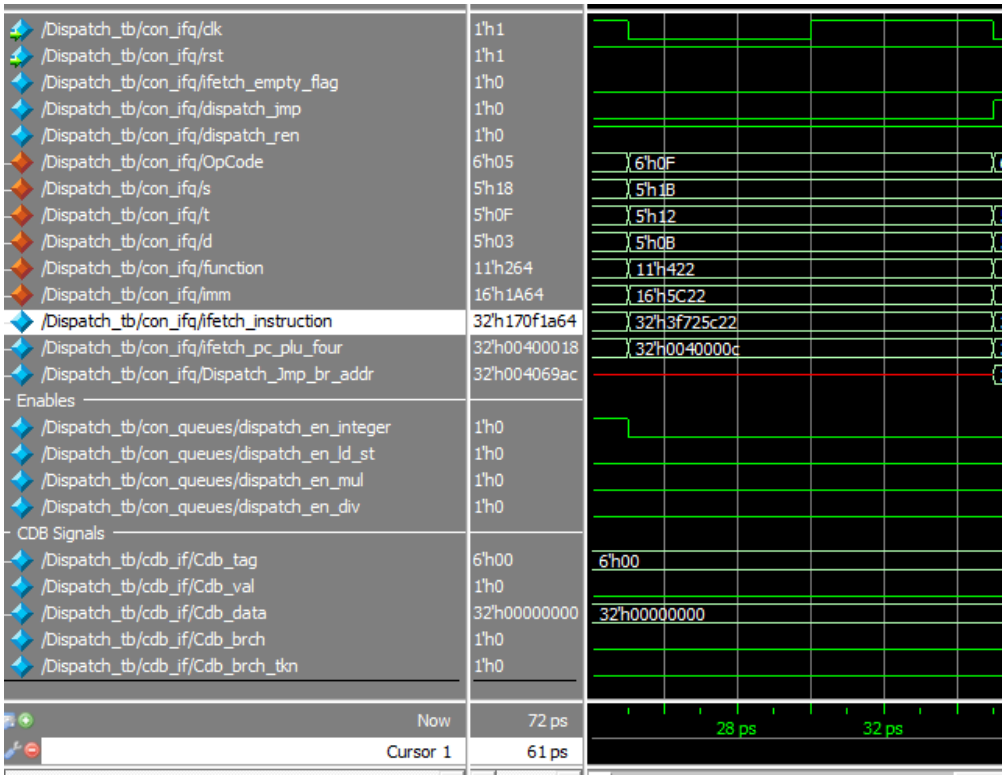
Resultados.

Luego de trabajar en la creación del ambiente de verificación para la "Dispatch Unit" del procesador MIPS Superescalar, hemos encontrado algunos errores:

1) Existe una discrepancia de implantación con respecto a la especificación de las instrucciones de multiplicación (mult) y división (div) para el MIPS. En la implantación no se utilizan los registros HI y LO para almacenar el resultado de estas operaciones. En cambio, en el formato de la instrucción se debe emplear un solo registro de 32 bits (en este caso, el registro "d") para almacenar el resultado. Esto hace perder los 32 bits más significativos del resultado en el caso de la multiplicación y el módulo en el caso de la división, lo cual constituye un error.
2) En la implantación se puede hacer uso directo por parte del usuario de cualquiera de los 32 registros del procesador, esto tampoco cumple con la especificación del MIPS; lo grave de ello es que se compromete la funcionalidad del micro al no limitar el acceso del usuario a registros importantes como el apuntador de la pila (stack pointer) y el registro de dirección de retorno (return address). Si estos registros se sobreescriben de forma intencional o accidental por parte del usuario, el procesador tendrá un comportamiento errático y con seguridad se entorpecerá la entrega de resultados coherentes.
3) Otra discrepancia que se encontró con respecto a la especificación estándar del MIPS y que no está documentada, corresponde al OpCode utilizado para identificar la operación MULT. La MIPS Reference Sheet establece que el opcode es 0x18 y la versión del MIPS con la que estamos trabajando utiliza el OpCode 0x19. Si el usuario no es advertido de esto y escribe un programa que haga uso de multiplicaciones, y posteriormente dicho programa es ensamblado mediante algún simulador que cumpla con la especificación estándar del MIPS como es el caso de MARS, las operaciones de multiplicación nunca se ejecutarán de forma adecuada.



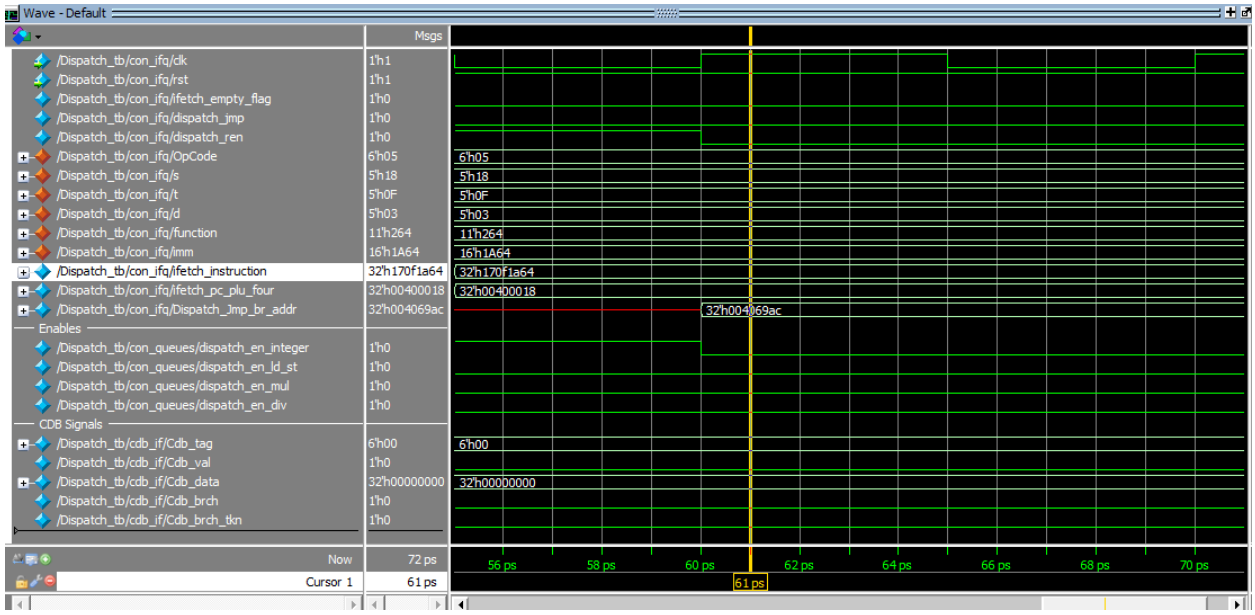
4) La instrucción LUI no es soportada por el DUV y dicha situación no se documenta en ninguna parte. Cuando el OpCode 0x0F se presenta, éste no se manda a ninguna cola de ejecución como se ve en la siguiente imagen. Debería mandarse a la Unidad de Ejecución de Enteros para su ejecución. Se puede observar en el flanco de reloj de subida a los 30 [ps] que no se envía a ninguna cola de ejecución dado que todas están vacías.



Por definición, LUI es R[rt]={imm,16'b0} lo cual no se está realizando.

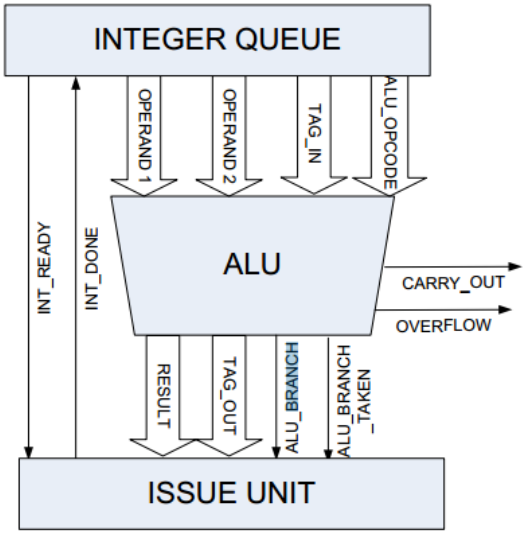
Load Upper Imm.	`lui`	I	R[rt] = {imm, 16'b0}	f_{hex}

5) La instrucción de Branch no manda la señal dispatch_en_integer, el cual según la especificación que revisamos en la clase debería hacerlo.



Como se observa en la cola de enteros, se tiene las líneas de ALU_BRANCH y de ALU_BRANCH_TAKEN, por lo cual se debería habilitar el dispatch_en_integer al menos por un ciclo de reloj para mandar a calcular si se salta o no.

Interface con la cola de ejecución de enteros y la ALU.

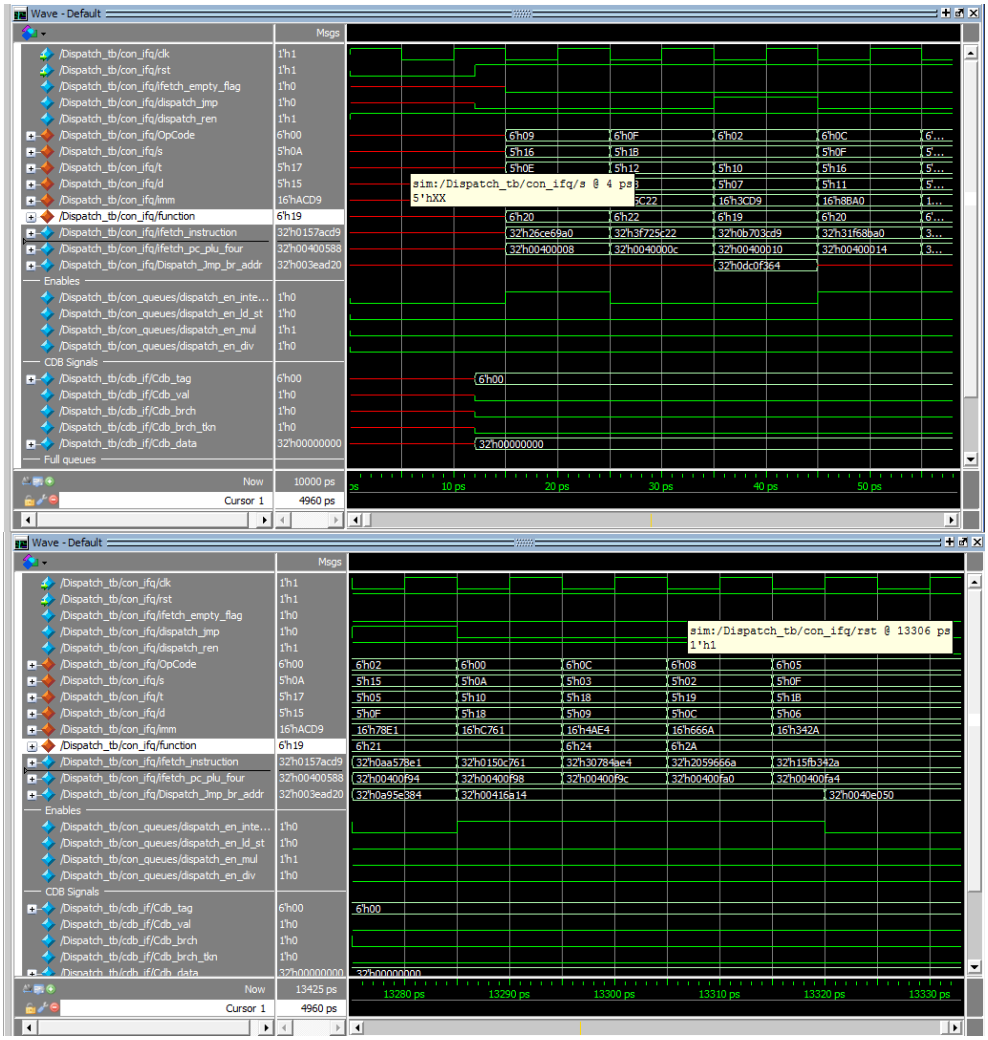


6) La ejecución de la operación SLTIU no es la misma de la especificación del MIPS, según el cual se debe usar un Sign Extended Immediate para la operación y se está usando un Zero Extended Immediate.

ong p	Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}

```
# ./Dispatch_Driver.sv:147:321 ps: SAVING instruction for CDB & Monitor Response: 2cd6a3e5
# ./Dispatch_Checker.sv:246:322 ps: OpCode: 0b instruction: SLTIU
# ./Dispatch_Checker.sv:40:322 ps: ERROR!: On INT instruction RT data does not match the sign ext immediate sent 0x0000a3e5 != 0xffffa3e5
# ./Dispatch_Monitor.sv:48: ERROR!: Instruction Logic RESPONSE FAILED!!!!
# ** Note: $stop    : ./Dispatch_Monitor.sv(49)
#    Time: 322 ps  Iteration: 0  Instance: /Dispatch_tb
```

7) La ejecución con no respuesta en CDB continua a pesar de que todos los Tags ya han sido asignados. La ejecución debería detenerse después de 64 Tags asignados.



Luego de más de mil instrucciones ejecutadas, la Dispatch Unit jamás se detuvo a pesar de que los Tags no eran renovados.

También encontramos algunos aspectos fácilmente mejorables tanto en la documentación como en la arquitectura:

1) En la documentación de la "Dispatch Unit" debería señalarse el set de instrucciones que esta versión del MIPS reconoce y ejecuta. Tal y como se encuentra actualmente la documentación es necesario revisar directamente el código en Verilog para enterarse de cuáles son las instrucciones reconocibles por esta versión del microprocesador MIPS.

2) Y dada la funcionalidad ya implantada en la arquitectura, ésta podría extenderse mediante la implantación de algunas instrucciones adicionales cuya inclusión representaría sólo algunas líneas extra de código, estas instrucciones son:

Instrucción	Operación	Instrucción	Operación
divu $s, $t	lo = $s / $t; hi = $s % $t	jr $s	pc = $s
multu $s, $t	hi:lo = $s * $t	lb $t, i($s)	$t = SE(MEM[$s+i]:1)
sllv $d, $t, $s	$d = $t << $s	lbu $t, i($s)	$t = ZE(MEM[$s+i]:1)
sra $d, $t, a	$d = $t >> a	lh $t, i($s)	$t = SE(MEM[$s+i]:2)
srav $d, $t, $s	$d = $t >> $s	lhu $t, i($s)	$t = ZE(MEM[$s+i]:2)
srlv $d, $t, $s	$d = $t >>> $s	sb $t, i($s)	MEM [$s + i]:1 = LB ($t)
xor $d, $t, $s	$d = $s ^ $t	sh $t, i($s)	MEM [$s + i]:2 = LH ($t)
xori $d, $t, i	$d = $s ^ ZE(i)		
bgtz $s, label	if ($s > 0) pc += i << 2		
blez $s, label	if ($s <= 0) pc += i << 2		
jalr $s	$31 = pc; pc = $s		

Conclusiones

El aprendizaje que se obtuvo durante el semestre fue enriquecedor tanto en el ámbito académico como en el industrial. En el ámbito académico destaca la revisión de los conceptos básicos relacionados con el desarrollo de ambientes de validación empleando SystemVerilog, y la creación de un plan de validación con todos los elementos indispensables. También es destacable que el proyecto final sea la validación de un procesador MIPS SuperScalar desarrollado en uno de los cursos previos de la maestría. En cuanto a los aprendizajes valorables desde el punto de vista industrial se tiene la revisión en clase de algunos aspectos de la metodología OVM que se emplea actualmente para validar diseños de sistemas digitales con fines comerciales. Algunos de los ejemplos que se vieron fueron reforzados con explicaciones y aplicaciones usadas en el ámbito laboral real, siempre dando ejemplos referentes a la validación de sistemas digitales actuales, lo que le dio un valor adicional al curso.

Por otra parte, el enfoque del curso nos permitió desarrollar habilidades y adquirir conocimientos sobre el uso de SystemVerilog, el manejo del simulador Questa Sim 64 de Mentor Graphics y el entendimiento de especificaciones que emulan las empleadas en el sector industrial. Gracias a ello fue posible implantar un ambiente básico de verificación desde cero y en el proceso comprender desde conceptos básicos relacionados con la programación orientada a objetos hasta conceptos muy elaborados como manejo de aserciones y cobertura del espacio de prueba. No obstante, algunos aspectos mejorable del curso son la profundización en algunos conceptos y metodologías a partir del desarrollado de código junto con el profesor, nos referimos en concreto al manejo de aserciones, el manejo de ‘coverage’ y el desarrollo de un ambiente de prueba empleando las librerías de OVM; es deseable desarrollar durante las primeras clases un ejemplo pequeño y completamente funcional de un ambiente de validación para mejorar la comprensión de los conceptos y facilitar el paso de la parte conceptual a la parte práctica; y a mediados del curso sería deseable actualizar ese mismo ambiente de validación empleando las librerías de OVM. También sería deseable la adopción de una herramienta que sea accesible fuera de los laboratorios de la universidad para ampliar el tiempo que puede dedicarse al desarrollo de código y la práctica del uso de la herramienta de desarrollo.

Para finalizar, estamos convencidos que el objetivo del curso se cumplió al desarrollar desde cero un ambiente de verificación para un sistema digital que es parte de un diseño más grande. Al implantar el ambiente fue necesario revisar las especificaciones de funcionamiento de manera planificada y se procuró cumplir con todos los requisitos propuestos al comienzo del proyecto. La manera como se ejecutó este proyecto sin lugar a dudas dio a los integrantes del equipo una visión real y actualizada sobre el diseño y verificación de sistemas digitales.

Anexo. Tabla de instrucciones.

Instrucciones del tipo R. Formato de la instrucción.

Campo	OpCode	Registro $s	Registro $t	Registro $d	Shift Amount + Function
Tamaño	[5:0]	[4:0]	[4:0]	[4:0]	Function: [10:0] o Shift Amount: [4:0] + Function: [5:0]

Instrucción	Operación	Sintaxis	OpCode	Registro $s	Registro $t	Registro $d	Shift Amount	Fun
ADD	$d = $s + $t; advance_pc (4);	add $d, $s, $t	0000 00	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)		000
ADDU	$d = $s + $t; advance_pc (4);	addu $d, $s, $t	0000 00	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)		000
AND	$d = $s & $t; advance_pc (4);	and $d, $s, $t	0000 00	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)		000
OR	$d = $s | $t; advance_pc (4);	or $d, $s, $t	0000 00	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)		000
SLT	if $s < $t $d = 1; advance_pc (4); else $d = 0; advance_pc (4);	slt $d, $s, $t	0000 00	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)		000
SLTU	if $s < $t $d = 1; advance_pc (4); else $d = 0; advance_pc (4);	sltu $d, $s, $t	0000 00	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)		000
SLL	$d = $t << h; advance_pc (4);	sll $d, $t, h	0000 00	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 0 y 2^5-1	00 0
SRL	$d = $t >> h; advance_pc (4);	srl $d, $t, h	0000 00	-- ---	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 0 y 2^5-1	00 0
SUB	$d = $s - $t; advance_pc (4);	sub $d, $s, $t	0000 00	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)		000

SUBU	$d = $s - $t; advance_pc (4);	subu $d, $s, $t	0000 00	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)		000

Instrucciones del tipo I. Formato de la instrucción.

Campo	OpCode	Registro $s	Registro $t	Dato inmediato
Tamaño	[5:0]	[4:0]	[4:0]	[15:0]

Instrucción	Operación	Sintaxis	OpCode	Registro $s	Registro $t	Dato inmediato
ADDI	$t = $s + imm; advance_pc (4);	addi $t, $s, imm	0010 00	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 0 y 2^{16}-1 (16 bits menos significativos)
ADDIU	$t = $s + imm; advance_pc (4);	addiu $t, $s, imm	0010 01	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 0 y 2^{16}-1 (16 bits menos significativos)
ANDI	$t = $s & imm; advance_pc (4);	andi $t, $s, imm	0011 00	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 0 y 2^{16}-1 (16 bits menos significativos)
LUI	$t = (imm << 16); advance_pc (4);	lui $t, imm	0011 11	-- ---	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 0 y 2^{16}-1 (16 bits menos significativos)
ORI	$t = $s | imm; advance_pc (4);	ori $t, $s, imm	0011 01	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 0 y 2^{16}-1 (16 bits menos significativos)
SLTI	if $s < imm $t = 1; advance_pc (4); else $t = 0; advance_pc (4);	slti $t, $s, imm	0010 10	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 0 y 2^{16}-1 (16 bits menos significativos)
SLTIU	if $s < imm $t = 1; advance_pc (4); else $t = 0; advance_pc (4);	sltiu $t, $s, imm	0010 11	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 0 y 2^{16}-1 (16 bits menos significativos)

Instrucciones del tipo J. Formato de la instrucción.

Campo	OpCode	Registro $s	Function + Dato inmediato
Tamaño	[5:0]	-- --- o [4:0]	Dato inmediato: [25:0] o Function [20:0]

Instrucción	Operación	Sintaxis	OpCode	Registro $s	Function	Dato inmediato
J	PC = nPC; nPC = (PC & 0xf0000000) | (target << 2);	j target	0000 10	-- ---		Valor entre 0 y 2^{26}-1 (26 bits menos significativos
JAL	$31 = PC + 8 (or nPC + 4); PC = nPC; nPC = (PC & 0xf0000000) | (target << 2);	jal target	0000 11	-- ---		Valor entre 0 y 2^{26}-1 (26 bits menos significativos

Instrucciones del tipo Branch. Formato de la instrucción.

Campo	OpCode	Registro $s	Registro $t	Dato inmediato (offset)
Tamaño	[5:0]	[4:0]	[4:0]	[15:0]

Instrucción	Operación	Sintaxis	OpCode	Registro $s	Registro $t	Dato inmediato
BEQ	if $s == $t advance_pc (offset << 2)); else advance_pc (4);	beq $s, $t, offset	0001 00	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 0 y 2^{16}-1 (16 bits menos significativos)
BNE	if $s != $t advance_pc (offset << 2)); else advance_pc (4);	bne $s, $t, offset	0001 01	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 0 y 2^{16}-1 (16 bits menos significativos)

Instrucciones para multiplicación y división. Formato de la instrucción.

Campo	OpCode	Registro $s	Registro $t	Function
Tamaño	[5:0]	[4:0]	[4:0]	[15:0]

Instrucción	Operación	Sintaxis	OpCode	Registro $s	Registro $t	Registro $d	Function
MULT	$d = $s * $t; advance_pc (4);	mult $s, $t	0000 00	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (0 0010 – 1 1001)	Valor entre 2 y 25 (0001 0 – 1100 1)	000 0001 1000
DIV	$d = $s / $t; advance_pc (4);	div $s, $t	0000 00	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (0 0010 – 1 1001)	Valor entre 2 y 25 (0001 0 – 1100 1)	000 0001 1010

Instrucciones para Load y Store. Formato de la instrucción.

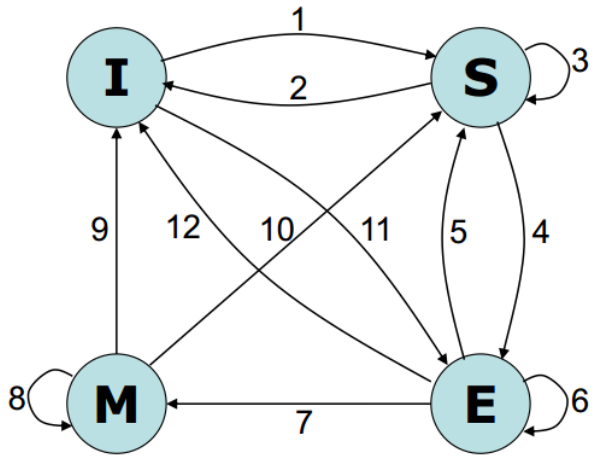
Campo	OpCode	Registro $s	Registro $t	Dato inmediato (offset)
Tamaño	[5:0]	[4:0]	[4:0]	[15:0]

Instrucción	Operación	Sintaxis	OpCode	Registro $s	Registro $t	Dato inmediato
LW	$t = MEM[$s + offset]; advance_pc (4);	lw $t, offset($s)	1000 11	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 0 y 2^{16}-1 (16 bits menos significativos)
SW	MEM[$s + offset] = $t; advance_pc (4);	sw $t, offset($s)	1010 11	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 2 y 25 (00 010 – 11 001)	Valor entre 0 y 216-1 (16 bits menos significativos)

C. REPORTE DE IMPLEMENTACIÓN DE UN SISTEMA CON MEMORIA INMEDIATA (CACHE) BASADO EN MESI

Summary

While the number of cores and its speed increases, the memory management increases in complexity as more and more cores and memory is added into the system. Over the last years there have been developed several cache coherence protocols, basically divided into two main categories: invalidation base protocols and update-based protocols. Citing an evaluation paper on several protocols "Invalidation based protocols invalidate all other cached copies of the data on a write, whereas update-based protocols broadcast the write so that other cached copies can be updated" (Suleman).
In the next implementation, MESI protocol will be develop as a basic invalidate-base protocol to maintain cache and main memory coherency. MESI is one of the most common protocol that can support write-back caches which is an advantage to free bus bandwidth, helping supporting the increment in cores into modern systems. The basic state machine that will be implemented is shown in the following figure and table.



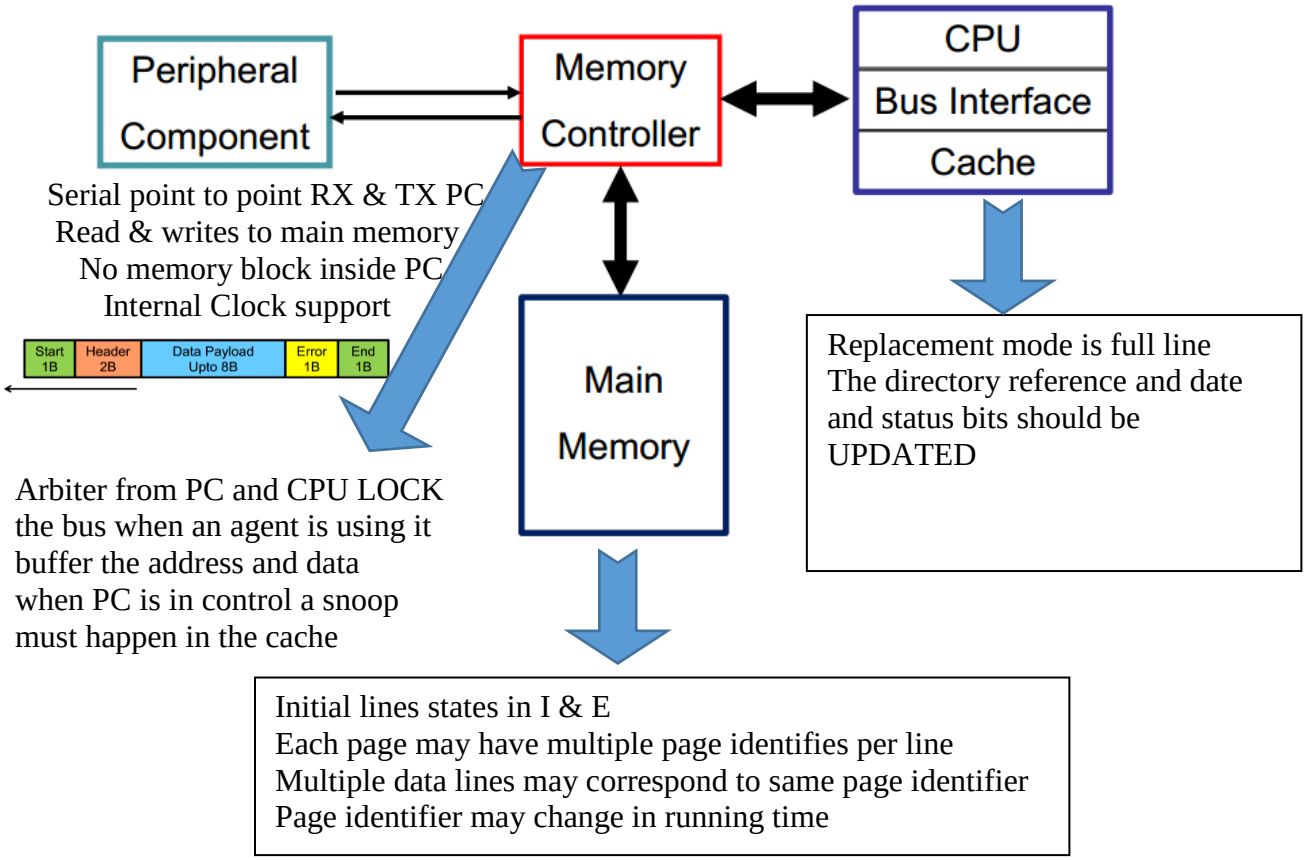
1. Read miss, line fill (WB/WT# = 0)
2. External snoop hit on write (INV=1) or internal snoop hit or FLUSH# or INVD or WBINVD
3. Write hit (WB/WT#=0) or read hit or external snoop hit on read
4. Write hit (WB/WT#=1)
5. External snoop hit on read
6. Read hit
7. Write hit
8. Read hit or write hit
9. Write back: external snoop hit on write (INV=1) or internal snoop hit or FLUSH# or INVD or WBINVD
10. Write back: external snoop hit on read
11. Read miss, line fill (WB/WT#=1)
12. External snoop hit on write (INV=1) or internal snoop hit or FLUSH# or INVD or WBINVD

Basically, each circle represents a state for a cache line. I state represents an invalid data state inside the specified line, which can be used to load new valid data. S state represents a shared clean data of the line, there may be several copies inside another cache cores or different cache levels. M state means the line has been modified by the core and is dirty data requiring a write to the main memory. E state is present in a line when it is only stored in one cache and the data is clean.

Depending on the system architecture the MESI protocol can be fully implemented or partially implemented. In this case, when the design only has one cache level and one core implemented, there is no need to implement it completely.

System design specifications

The main computing system block diagram is shown in the figure below:



General Design Specifications

In this section design limitations and constrains are shown:

- All design must be synchronized by a common clock connected to all main bocks.
- Memory controller connects with main memory and CPU through buses including address, data, and control signals. Bus direction must be set by control signals.
- Memory Addressing is 24-bit distributed in the following scheme.

23	page reference	8	7	memory line	0

- Page size is 256 lines each with 32-bit width, main memory should have at least two pages and cache is required to be only one page.

Design specific implementation

In this section all the specific design details are documented. The interfaces between modules and the verification mechanism to be used in the implementation.

Peripheral Component

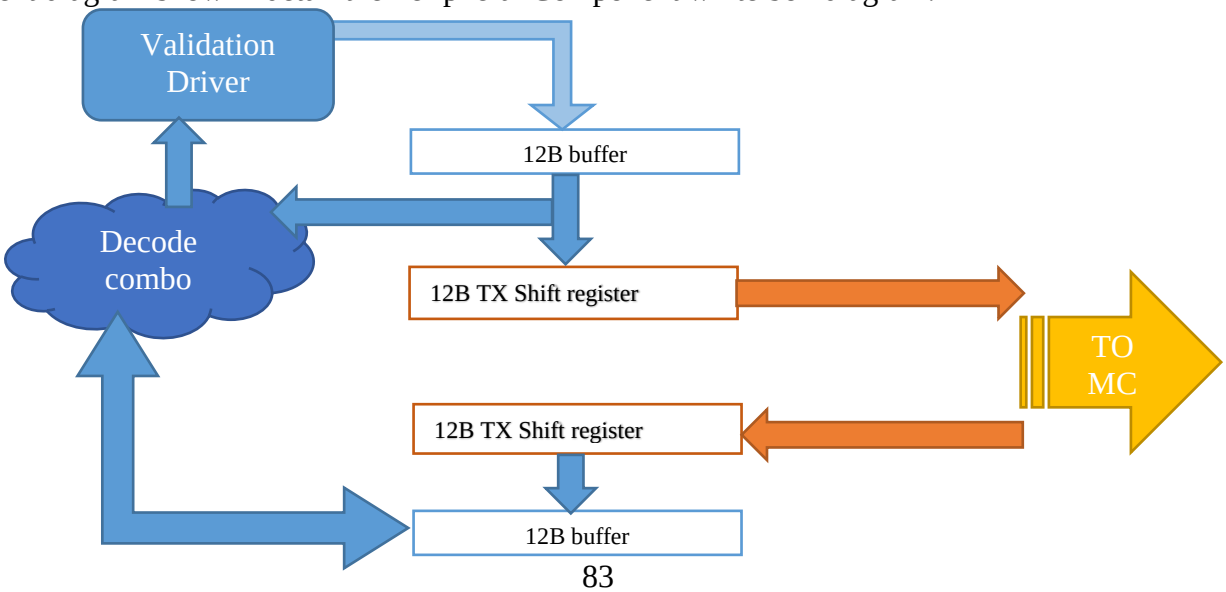
Peripheral component will be implemented the following defined protocol. RX and TX channels will use this same convention for all the packets that need to transmit. The full size on an entire packet is 12 Bytes long including the start and the stop bytes.

1B Stop	1B Error	7B Data Payload		2B Header		1B Start
8'h5A	8'h8A Error 8'h00 No Error	32'hX Line Data	24'hX Memory Address	8'hFF Invalid Data 8'hAA Valid Data	8'h3B Mem Read 8'h3A Mem Write 8'hA2 Acknowledge 8'hA5 Retry Pkt	8'hA5

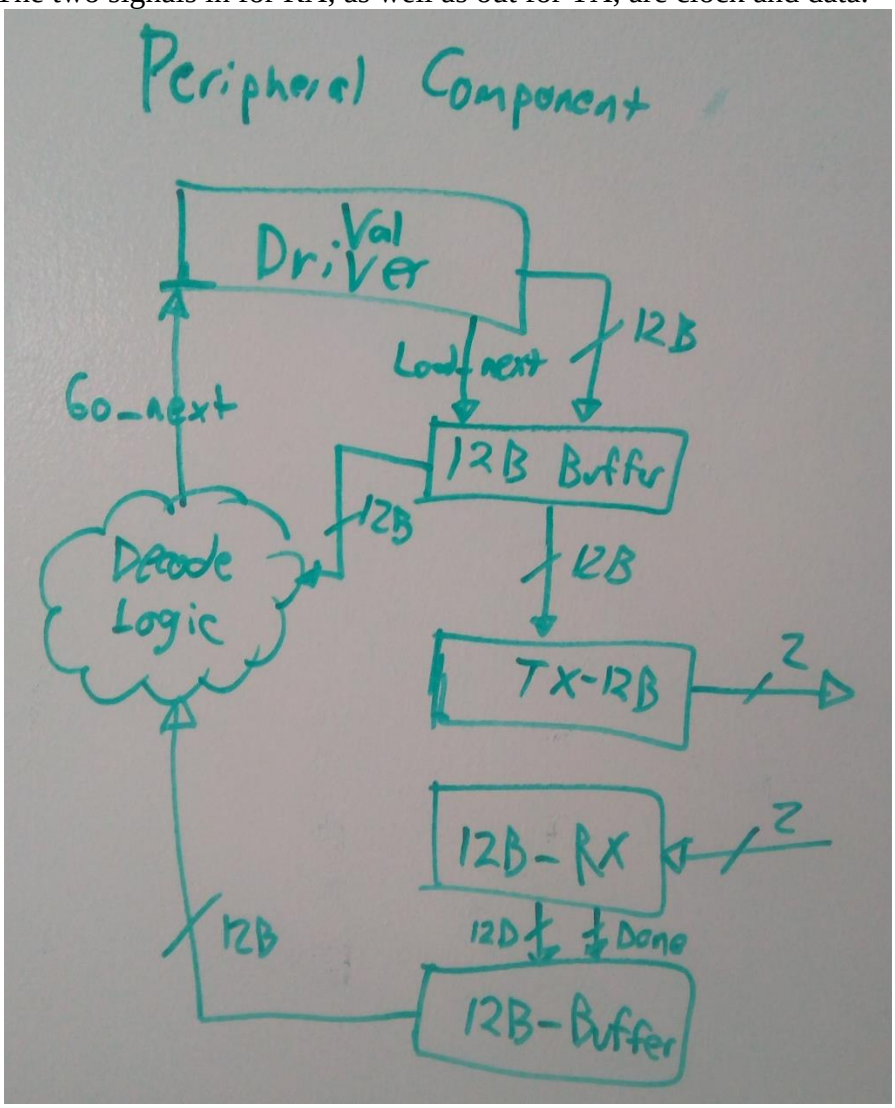
Simplifying CRC the errors will be injected randomly when generating the packets with errors. So basically it simulates the CRC implementation when an agent checks for a mismatch in a CRC module. Handling will when an error is detected by the MC or PC, then the corresponding module will send a Retry Pkt with the Invalid Data byte set and error field in No Error. In a real system if any endpoint detects an error, it will initiate the same transaction again.

MC must also send a Retry pkt if the bus is locked by the Cache even if the Error field is indicating no error is present. PC must keep sending the same transaction for each Retry received until it is finally accepted by the MC by an acknowledge message. PC also must wait until a response is received from the MC to send the next transaction as there will be no buffer implementation in MC.

Implementation will consist of a 12Bytes TX PISO shift register buffer, a 12Bytes RX SIPO shift register buffer and a 12 Bytes standard buffer to save the instruction to be executed. Also there will be combinational logic to decode when the instruction is processed or have an error. Next diagram show in detail the Peripheral Component white box diagram.



Next image is the schematic on the PC. It contains all signal names and directions from all components. The two signals in for RX, as well as out for TX, are clock and data.



Read flow

1. PC sends a pkt: Header field with MemRead/InvalidData value and Payload with valid address (in this case data field is irrelevant).
2. MC receives the pkt and check for errors. If no error is detected MC continues with read flow, if error is detected MC sends a Retry pkt. (Check error Handling)
3. If MC decides read can be handled, it locks cache from bus control and give it to PC.
4. Cache does a snoop while MC drives address bus and prepares to send a pkt: Header field with Acknowledge / ValidData or InvalidData, Payload with a valid address, data value and no error injected.
    a. If there is a miss or a hit clean in cache, MM drives the data bus to build the packet.
    b. If there is a hit modified in cache, Cache drives the data bus to build the packet.
5. Read is complete.

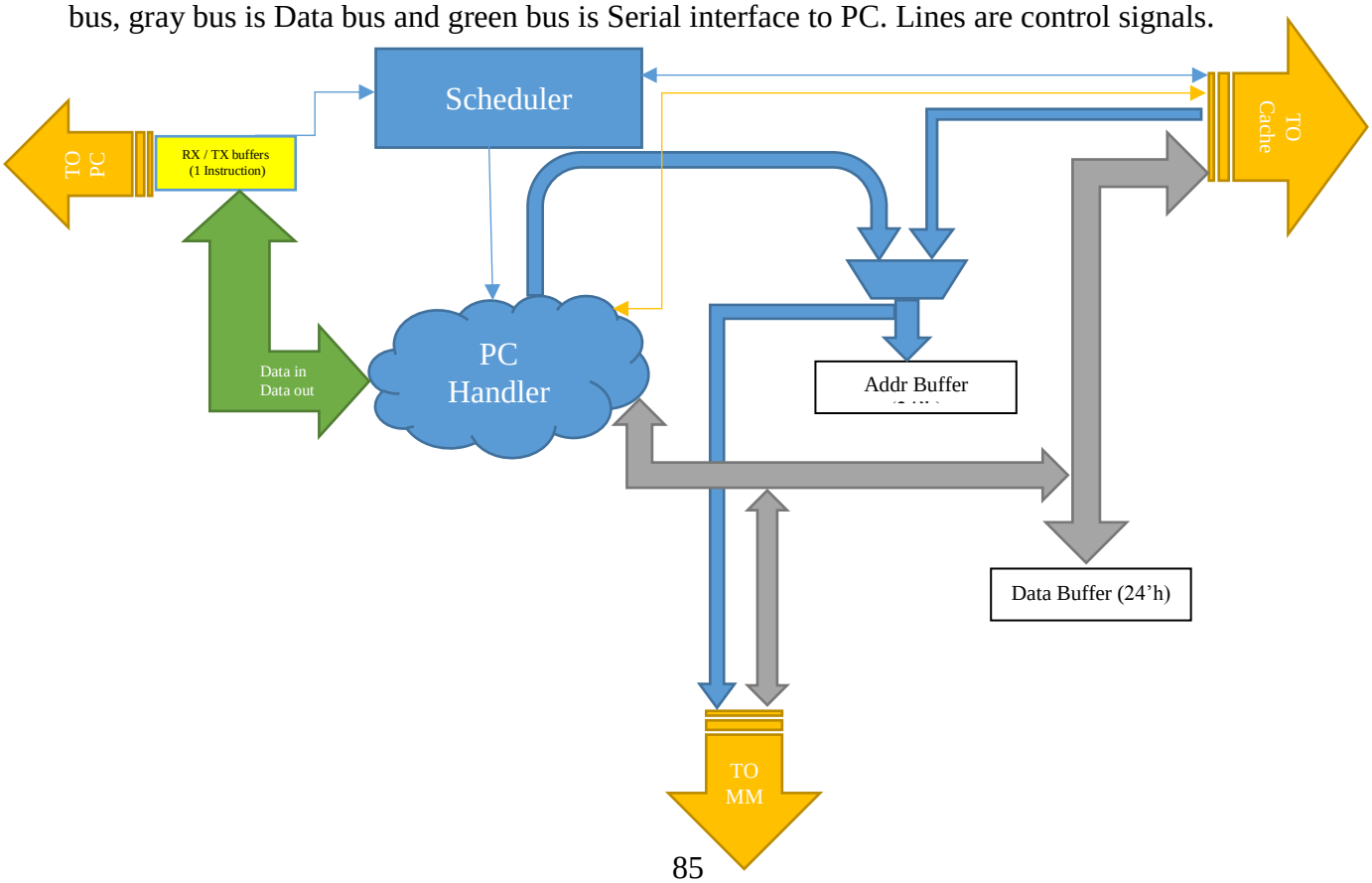
Write Flow

1. PC sends a pkt: Header field with MemWrite/ValidData value and Payload with valid address and data.
2. MC receives the pkt and check for errors. If no error is detected MC continues with write flow, if error is detected MC sends a Retry pkt. (Check error Handling)
3. If MC decides write can be handled, it locks cache from bus control and give it to PC.
4. Cache does a snoop while MC drives address and data bus and prepares to write to MM:
    a. If there is a miss or a hit clean in cache, MC writes data to MM and sends the packet: Header field with Acknowledge/InvalidData, Payload with a valid address and no error injected. Then cache puts the data in I state if there is a hit clean.
    b. If there is a hit modified in cache, MC sends a retry packet and then gives the control to cache to write in next clock cycle.
5. Write may be complete, depending on the MC response.

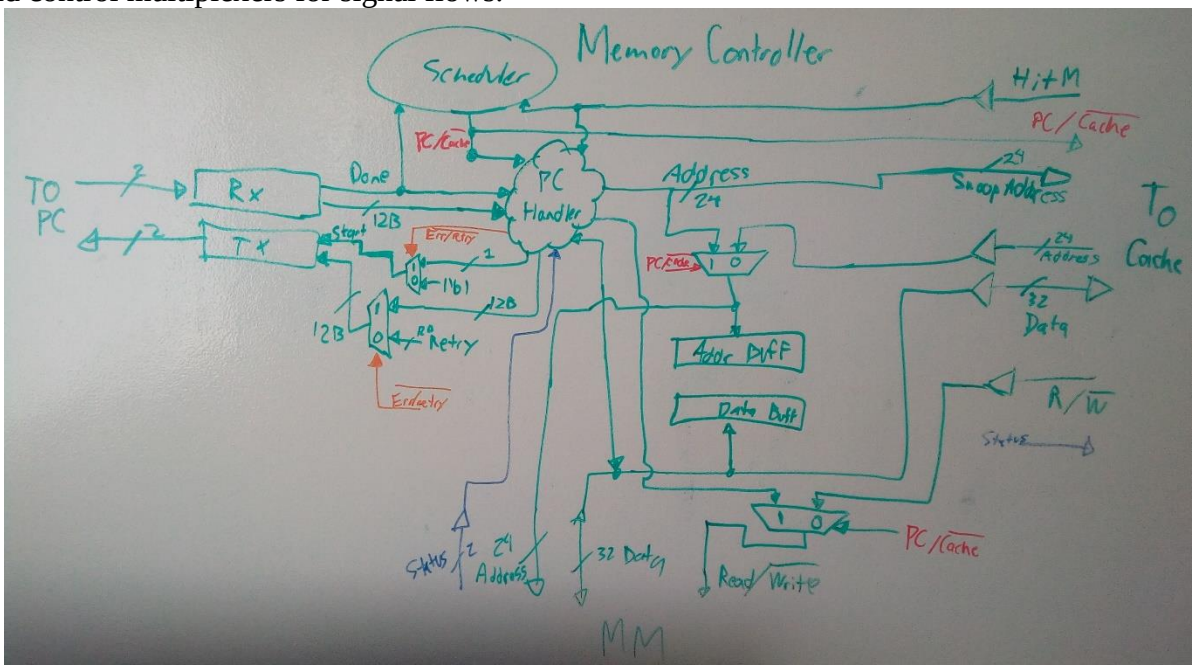
Memory Controller

Memory controller will consist on two main components that interact to route the data where it needs to be. First, the PC handler with all logic to decode and serve PC requests, it also drives Data bus when needed as well as Address and SnoopAddress signals when requested by the scheduler module. The second main component is the scheduler module acts as an arbiter between the Cache, PC and MM modules, it gives ownership to the data bus as well as signal driving for the correct Address to MM module.

A general high-level MC schematic to be implemented is shown below. Blue bus is address bus, gray bus is Data bus and green bus is Serial interface to PC. Lines are control signals.



Next image is the schematic on MC in details with all signal names and specifying direction and control multiplexers for signal flows.



MC to cache interface is mainly dominated with PC/Cache# signal which acts as an arbiter between PC and Cache modules to access main memory. If there is a hit modified the Cache module will notify the MC via HitM signal. Then the read or write flow inside the Peripheral Component chapter is triggered to complete the required write or read transaction.

MC to MM interface is dominated by R/W# signal, which controls the Data bus direction inside the MM module. The address input defines region that will be manipulated inside the MM, also it is controlled by a multiplexer with PC/Cache# signal.

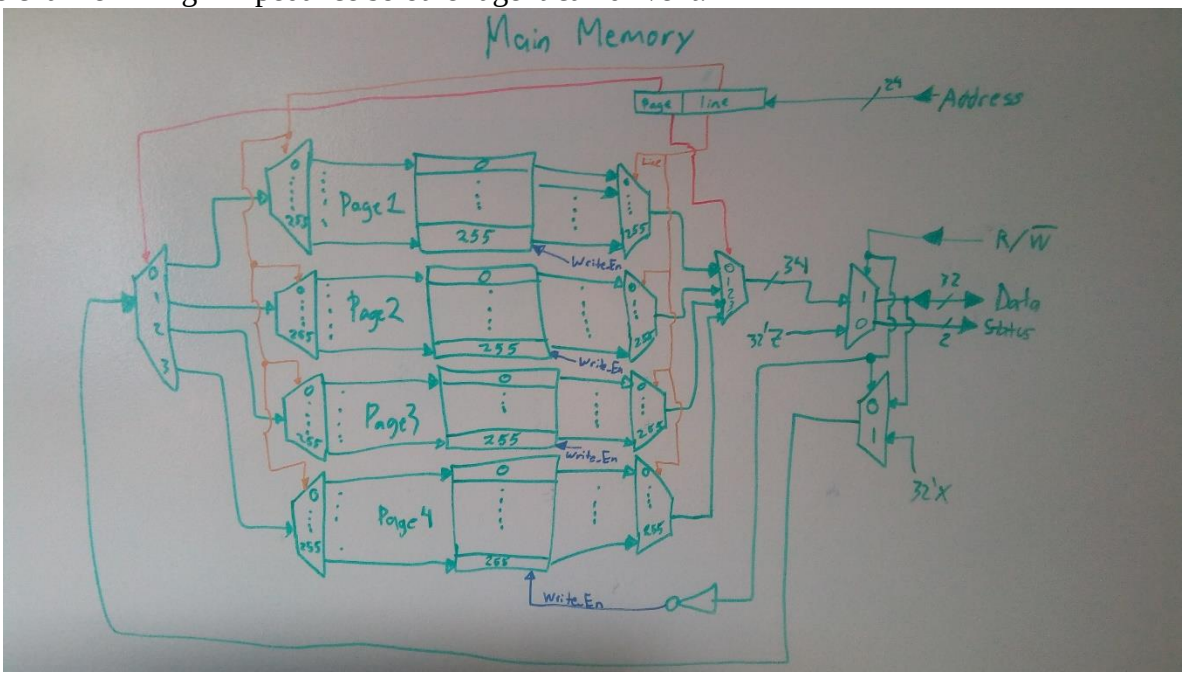
Same SIPO and PISO registers are used in RX and TX portion on the PC side. All read and write flows are explained in PC and Cache modules section.

Main Memory

Main Memory or MM implementation will consist on 4 pages of 1024 bytes arranged in 256 lines of 32-bits each. Align memory will be used and each byte will be mapped inside a 4-Byte array. In this implementation, all page lines will have different page identifier and they will not change in running time to make the implementation and verification simpler. There will be implementation to change page identifier during run time without verification. Memory coherency must not change as long as there is a HW mapped table with the physical address.

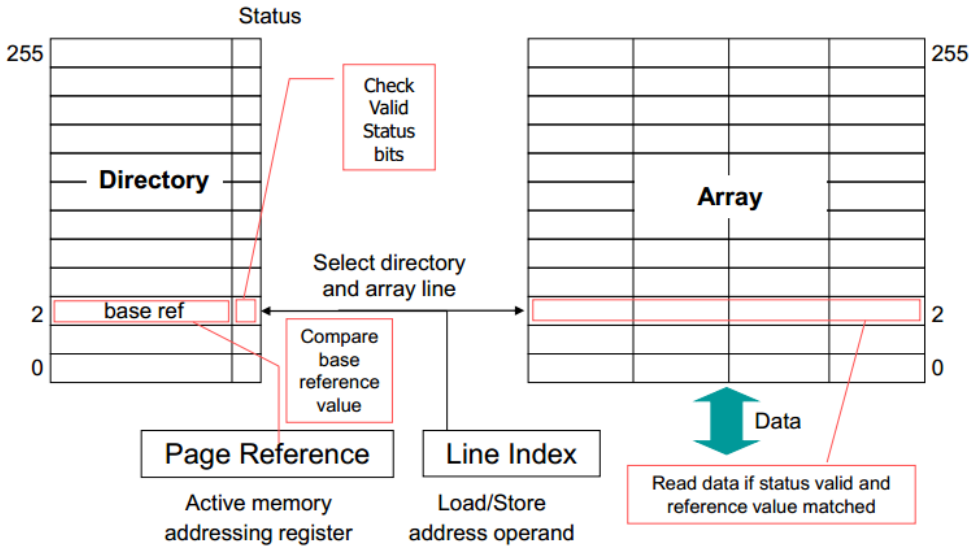
Initial line states in MM will be either I-state or E-state from MESI cache coherency protocol. The core can change the cache state lines to E-state when read an E-state or to M-state when reading an I-state line. Write back will only occur when the hitm signal is asserted due to a PC read or write request to that specific line.

Next image shows the simplified implementation diagram of a 4 page with dynamic page identifiers and MESI states. All write enables are directed to all pages, then selecting the one thought a page identifier multiplexer with the same Page value inside the address signal (page identifier and MESI state directories are not shown in the diagram). Each page can be instantiated as a separate Verilog module. R/W# signal is used to drive either a value of the address or to set the channel in high impedance so other agent can drive it.

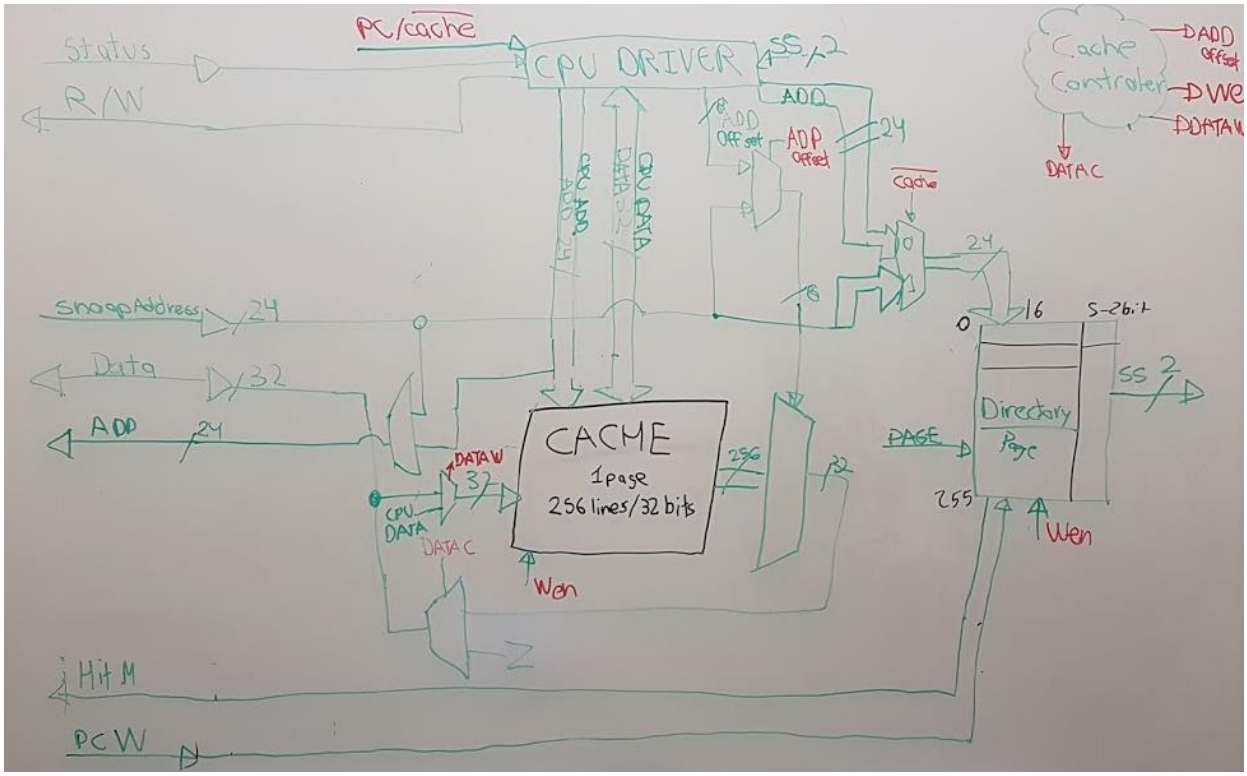


Cache

The CPU and Cache units are connected by the bus interface that could be control the MESI protocol. In these case we will simulate the CPU and execute the instructions and only take the operation by data.



Next is the image of CPU driver and cache. It constrains the signals and the possible directions and muxes for the different cases that we can have in the CPU and CACHE flow. Cache is set the initial state and all data cache status bits are initially set to the invalid state (I). All state elements are set to the reset state. The system may start a sequence of memory addressing and read operations.



Read flow with Cache MM access enabled

1. CPU receives a read instruction with the cache access enable.
2. Cache Directory receives the address and will send the status of the direction:
    a. Hit Modified: If we have a Hit modified cache will send data to the CPU driver. CPU driver receives the data and transaction is complete with no external access needed.
    b. Miss: If the address is missed, Cache needs to request the address to MC then Cache will receive it from DATAW Mux and then the CPU can read the address and Cache updates the directory with the new address.
    c. Hit Clean: If we have a Hit clean we well send data to the CPU driver. CPU driver receives the data and transaction is complete with no external access needed. If it is a write then state must be changed to Modified.

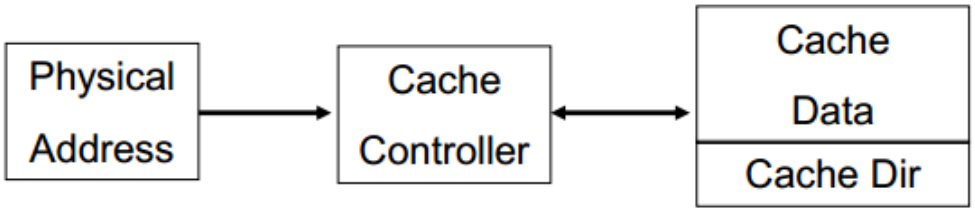
Write flow with Cache MM access enabled

1. Cache receives a write instruction with the MM access enabled.
2. Control cache generate the Write enable for cache and directory.
3. Directory looks for an I-state line then an E-state line, if there is none then it will issue a write back of a single M-state line inside the cache.
4. They would change the status of this address on the directory by modified. And write the data on the line found by the directory.

Snoop for PC Read/Write flow

1. If we have valid snoop address, Cache will have the PC signal asserted and cache will take the Snoopaddress signal.
2. Cache will send snoop address to directory and will try to find the address and the status. Depend of the status of address we can have the follow 3 cases:
    a. Hit Modified: If we have a Hit modified cache will assert the hitM signal and wait for the MC to give control over the data bus. When the data bus control is granted then cache sends data to the MC. MC receives the data and transaction is complete.
    b. Miss: If the address is missed, no further action is needed.
    c. Hit Clean: If the address is hit clean, no further action is needed. If PC writes to MM the line in cache must be invalidated checked with PCW signal.

Cache Functional Diagram



CPU driver flow

The CPU implementation need to have the follow characteristics:

1. Read and execute 1 instruction per clock.
2. Cache data operations may be managed by the CPU arbitrarily covering all MESI states as well as interaction with main memory.
3. Determine cache hit or miss.
4. Follow MESI protocol with status bit.

Interfaces

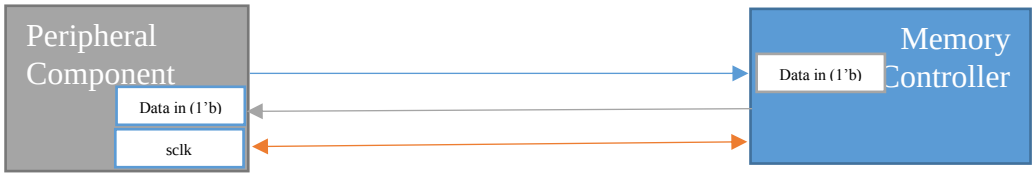
All interface signals summary are defined inside the table 1. NC meaning No Connection, I for Verilog INPUT and O for Verilog OUTPUT for each module and bus for Verilog INOUT.

	Signal	Size	Module				Description
			PC	MC	MM	Cache	
Clocks and reset	clk	1'b	I	I	I	I	Main System Clock
	sclk	1'b	I	I	NC	NC	Serial clock for PC
	rst	1'b	I	I	I	I	Main Reset signal
MC/PC signals	TX_PC_RX_MC	1'b	O	I	NC	NC	TX line from PC serial interface
	TX_MC_RX_PC	1'b	I	O	NC	NC	RX line from PC serial interface
Data bus	data_bus_wire	32'b	NC	Bus	Bus	Bus	Data to be written in memory
MC/MM signals	read_write_mcmm_wire	1'b	NC	O	I	NC	Write or read signal: 1 read, 0 write
	state_in_mcmm_wire	2'b	NC	O	I	NC	State to set into the line after read or write
	address_mcmm_wire	24'b	NC	O	I	NC	MM Address to Register for Read/Write
	state_mcmm_wire	2'b	NC	I	O	NC	Current state of the line read
	not_found_mcmm_wire	1'b	NC	I	O	NC	Signal indicating the data is not in MM: 1 Data not found, 0 Data found
MC/Cache signals	r_w_MC_Cache_wire	1'b	NC	I	NC	O	Read or write signal from the Cache: 1 read, 0 write
	hitm_MC_Cache_wire	1'b	NC	I	NC	O	Hit modified signal: 1 hitm occured, 0 no hitm detected
	cache_req_mm_MC_Cache_wire	1'b	NC	I	NC	O	Signal to indicate access to mm is needed from Cache: 1 clk requested, 0 noclk required
	address_MC_Cache_wire	24'b	NC	I	NC	O	Adress from the cache unit to read or write from MM
	state_cache_MC_Cache_wire	2'b	NC	I	NC	O	MM should send the MM line into this state while writing or reading
	SnoopAddress_MC_Cache_wire	24'b	NC	O	NC	I	Snoop address sent to the Cache when PC is in data bus control
	state_mm_MC_Cache_wire	2'b	NC	O	NC	I	Line state read from MM before writing the new value
	pc_cache_turn_MC_Cache_wire	1'b	NC	O	NC	I	Signal from MC to indicate cache PC or Cache turn: 1 PC turn, 0 Cache Turn
	nodata_mm_MC_Cache_wire	1'b	NC	O	NC	I	Data not found in MM only when Cache has control: 1 Data not found, 0 Data found
	invalidate_line_MC_Cache_wire	1'b	NC	O	NC	I	Invalidate line signal to cache

Table 1

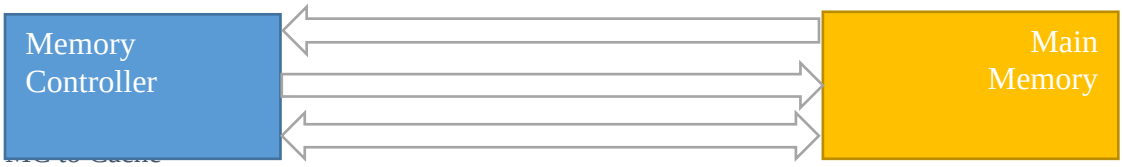
PC to MC

To make the design simpler there will be only 2 signals for RX and 2 for TX on the PC to MC interface. The clock will not be recovered from RX signal so there is a need to send two signals per serial interface clock and data. Blue lines are TX for PC and RX for MC and gray lines TX for MC and RX for MC.

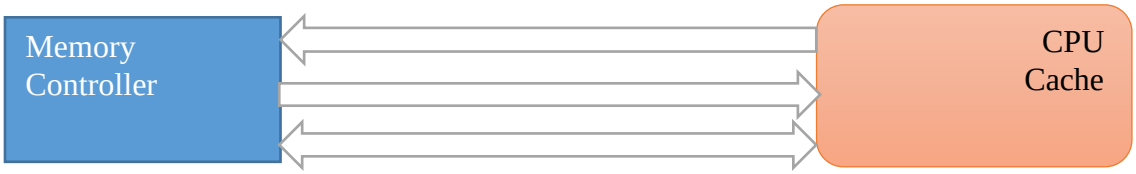


MC to MM

MC connects through an interface with the MM shown in the next image. All signal connections are in detail in the table 1.

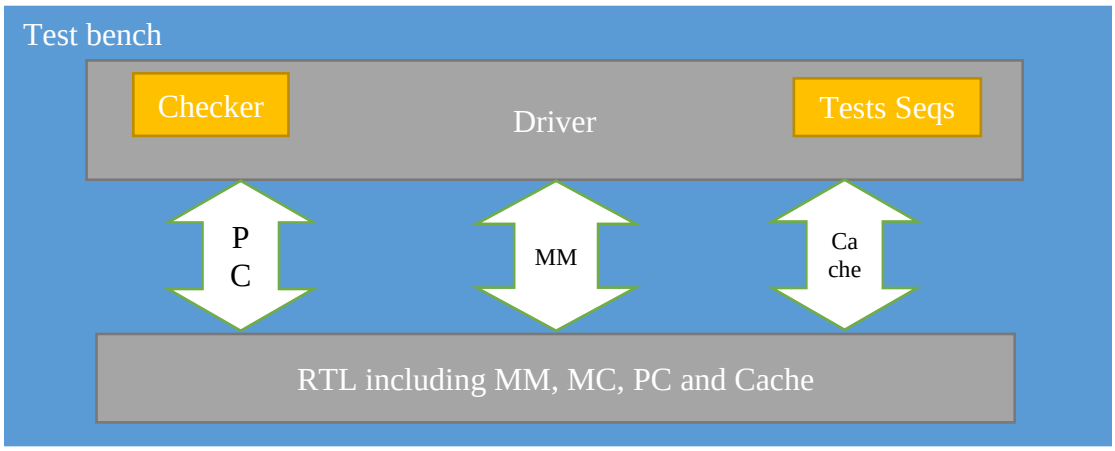


MC connects through an interface with the CPU shown in the next image. All signal connections are in detail in the table 1.



Test Plan

Test bench is based on a single system driver that send at the same time PC and CPU requests into the system. This system driver provides all necessary interface to validate data coherence, data path and PC and Cache read and writes. This will simplify the verification of all next scenarios.



- om the MM. This will be performed until the cache is full with valid data.

- ***CPU Read with cache hit clean or modified and optional PC transaction***
  CPU line reads with cache hit clean or modified so the cache send the data directly to the CPU and no MM transaction required. This test actually check the cache module is working properly and gives time to PC to take control over the data bus or not.

- ***PC Write Transaction to MM with hit clean or miss***
  PC Writes to MM while cache is doing a snoop, as data is clean or miss the PC should write MM with no problem. Test is checking the Data path and cache logic for snoop. If cache gets a hit clean the MC should send the PCW signal so the Cache will invalidate its line.

- ***PC Read Transaction to MM with hit clean or miss***
  PC Reads to MM while cache is doing a snoop, as data is clean or miss the PC should read MM with no problem. Test is checking the Data path and cache logic for snoop.

- ***PC Write Transaction to MM with hit modified***

PC writes to the MM while cache does a snoop, as data is modified then the write mechanism is activated for the Cache to write into MM. MC should send a retry to the PC. All this verifies the Write back mechanism works properly when the data is required.

- ***PC Read Transaction to MM with hit modified***
  PC reads to the MM while cache does a snoop, as data is modified then the MC read mechanism is activated for the Cache to write into MM. MC should send a retry to the PC. All this verifies the Write back mechanism works properly when the data is required.

- ***PC Error Retry test***
  PC retry test is aim to check the error retry mechanism inside the serial interface, this test verifies that an error injected transaction is retried and completed successfully.

- ***PC to MM Data Coherency check***
  PC to MM Data Coherency test function is to verify the PC reads and writes work correctly and is the last data in MM. Compare MM and PC arrival data checking is the latest in MM or in Cache.

Implementation Code Highlights

In this section, important parts of the implementation will be explained in detail.

Peripheral Component

Main structure of peripheral component where all messages are decoded and sent to the driver for coherence verification and data integrity. RTY packet is also used for error detection and correction. If no allowed combination is detected then a fatal response is issued.

```
//Get RX msg, decode it and send it to Driver
if(nw_msg && wait_rx_rsp) begin
    msg_rx_comp = msg_rx[7:0];
    msg_rx_sdata = msg_rx[15:8];
    msg_rx_err = msg_rx[79:72];
    drv_if.data_read = msg_rx[71:40];
    drv_if.addr_read = msg_rx[39:16];
    if (msg_rx_comp == ACK && msg_rx_sdata == VAL_DATA) begin
        drv_if.completion <= 1'h1;
        drv_if.valid_data <= 1'h1;
        drv_if.fatal_rsp <= 1'h0;
        retry <= 1'h0;
    end else if (msg_rx_comp == ACK && msg_rx_sdata == INV_DATA) begin
        drv_if.completion <= 1'h1;
        drv_if.valid_data <= 1'h0;
        drv_if.fatal_rsp <= 1'h0;
        retry <= 1'h0;
    end else if (msg_rx_comp == RTY_PKT) begin
        drv_if.completion <= 1'h0;
        drv_if.valid_data <= 1'h0;
        drv_if.fatal_rsp <= 1'h0;
        retry <= 1'h1;
    end else begin
        drv_if.completion <= 1'h1;
        drv_if.valid_data <= 1'h0;
        drv_if.fatal_rsp <= 1'h1;
        retry <= 1'h0;
    end
    wait_rx_rsp <= 1'b0;
end
```

PC module has the simpler signal interface on the design only serial lines, clocks. Interface to driver is connected as well in a single interface.

```
module PC(
    input   clk, sclk, rst,
    output  TX_line,
    input   RX_line,
    intfc_pc.pc drv_if
);
```

Main Memory

Main memory module connections are shown below. All of it is connected to Memory controller with the exception of the mm interfaces connected to the driver that actually works as a monitor for all the pages.

```
module MM(
    //INPUTs
    input clk,              //Main clock
    input read_write,       //Write Enable for write flow
    input [1:0] state_in,   //MESI state for memory for writes and or reads
    input [23:0] address,   //Address to Register for Read/Write
    inout [31:0] data,      //Data to be written or read from/to memory
    output [1:0] state,     //MESI state for line in Memory
    output not_found,       //Address is not inside MM
    //Driver_Monitor interfaces for all pages
    intfc_mm.mm page1,
    intfc_mm.mm page2,
    intfc_mm.mm page3,
    intfc_mm.mm page4
);
```

Below the main implementation of a simple page, here along with the driver signal, there is the read for data out and the state out. As shown, read implementation is combinational and write is done on clk positive edge.

```
always @(posedge clk) begin
    if (page_id == pag_identifier[line] && WriteEn) begin
            main_memory [line]  <= data_in;
    end
    if((page_id == pag_identifier[line]) && (state_in != I_STATE)) mem_state[line] <= state_in;
    if (pif.r_w_page) begin
        pag_identifier[pif.line_newid] <= pif.new_id;
        main_memory [pif.line_newid]  <= pif.data_dr;
        mem_state[pif.line_newid] <= E_STATE;
    end
end

//DUT output
assign data_out = (page_id == pag_identifier[line])? main_memory[line] : 32'hx;
assign state_out = (page_id == pag_identifier[line])? mem_state[line] : 2'hx;
assign found = (page_id == pag_identifier[line])? 1'h1 : 1'h0;
```

Memory Controller

Inside the Memory controller, the code for peripheral component handling messages is shown, implemented in a combinational manner it detects if any error is injected. All data and logic needed for answering any call from peripheral is in the code below.

```
always_comb begin
    if (msg_rx_req == MEM_READ && msg_rx_err == N_ER_DATA && msg_rx_sdata == INV_DATA) begin
        if (pc_cache && !hitm) begin
            if(status == I_STATE) new_tx_reg = {N_ER_DATA, data_in, msg_rx_addr, INV_DATA, ACK};
            else new_tx_reg = {N_ER_DATA, data_in, msg_rx_addr, VAL_DATA, ACK};
        end else begin
            new_tx_reg = {N_ER_DATA,56'h0, INV_DATA, RTY_PKT};
        end
    end else if (msg_rx_req == MEM_WRITE && msg_rx_err == N_ER_DATA && msg_rx_sdata == VAL_DATA) begin
        if (pc_cache && !hitm) begin
            new_tx_reg = {N_ER_DATA, 32'h0, msg_rx_addr, INV_DATA, ACK};
        end else begin
            new_tx_reg = {N_ER_DATA,56'h0, INV_DATA, RTY_PKT};
        end
    end else if (msg_rx_err == ERR_DATA && ((msg_rx_req == MEM_READ && msg_rx_sdata == INV_DATA)||(
    msg_rx_req == MEM_WRITE && msg_rx_sdata == VAL_DATA))) begin
        new_tx_reg = {N_ER_DATA,56'h0, INV_DATA, RTY_PKT};
    end else begin
        new_tx_reg = {ERR_DATA,56'h0, INV_DATA, FATAL_ERR};
    end
end
```

The main interface of the memory controller with the design is separated for better integration. MC is the heart of the design connecting all other modules. For any description on the signals check the interface Chapter on the Design specific implementation section.

```
module mc(
    //Clocks
    input   clk, sclk, rst,
    //PC interface
    output  TX_line,
    input   RX_line,
    //MM interface
    output read_write,
    output [1:0] state_in,
    output [23:0] address_to_mm,
    input [1:0] state,
    input not_found,
    //Cache interface
    input r_w,
    input hitm,
    input cache_req_mm,     //Si
    input [23:0] address,
    input [1:0] state_cache,
    output [23:0] SnoopAddress,
    output [1:0] state_mm,
    output pc_cache_turn,
    output nodata_mm,
    output invalidate_line,
    //common data bus
    inout [31:0] data
);
```

Shown in the below picture, the implementation of the data bus and all rerouting from the control signals to the main memory respect to the pc_cache signal which assigns the turn to the PC (1) or Cache (0). Snoop address and pc_cache signals are sent to the cache for doing the required snoop.

```
//Manage inout bus
assign data = (pc_cache && !rd_wr)? data_out: 32'hz; //PC turn and wr
assign data_in = data;
//Manage outputs for MM
assign read_write = (hitm)? 1'h1:(pc_cache)? rd_wr: r_w;
assign address_to_mm = (pc_cache)? PC_address: address;
assign state_in = (pc_cache)? pc_state:state_cache;
assign pc_state = (!rd_wr)? E_STATE: I_STATE; //IF invalid state is s
//Manage outputs for Cache
assign SnoopAddress = PC_address;
assign state_mm = (pc_cache)? 2'hx:state;
assign pc_cache_turn = pc_cache;
assign nodata_mm = (pc_cache)? 1'h0:not_found;
assign invalidate_line = (pc_cache && !rd_wr && !hitm) ? 1'h1: 1'h0;
```

Cache

Design for the cache shows a simple case on MESI exclusive state, in this state there is 3 main cases for the MESI state machine implemented. First the snoop hit invalidate the data if there is a write detected from peripheral component, second is the CPU write with hit inside the cache which sends the line to M state and then the rest of scenarios which maintains the E-state, cache needs to check if there is a hit or not inside cache.

```
E_STATE : begin  //E_state
    if (invalidate_line && hit_snoop) begin
        //Invalidate on snoop hit on write with 
        next_mesi_state = I_STATE;
        write_back = 1'h0;
        WriteEn_cache = 1'h0;
        WriteEn_pageid = 1'h0;
        wr_data = 32'hz;
        write_back_add = 24'hz;
        next_mesi_mm_state = state_mm;
    end else if (r_w == CPU_WRITE && hit) begin
        //Write and change the line to modified 
        next_mesi_state = M_STATE;
        write_back = 1'h0;
        WriteEn_cache = 1'h1;
        WriteEn_pageid = 1'h0;
        wr_data = cacheif.wr_data;
        write_back_add = 24'hz;
        next_mesi_mm_state = state_mm;
    end else begin
        next_mesi_state = E_STATE;
        write_back = 1'h0;
        wr_data = 32'hz;
        write_back_add = 24'hz;
        if(hit) begin
            WriteEn_cache = 1'h0;
            WriteEn_pageid = 1'h0;
            next_mesi_mm_state = state_mm;
        end else begin
            WriteEn_cache = 1'h1;
            WriteEn_pageid = 1'h1;
            next_mesi_mm_state = E_STATE;
        end
    end
end
M_STATE : begin  //M state
```

Also inside the cache there is a need for a write back logic when is required. The main implementation is shown for the main signals, the next_clk_wb signal is asserted when a snoop shows a hit modified exists while the peripheral is accessing main memory.

```
assign data = (next_clk_wb)? rd_data:32'hz;
assign address = (next_clk_wb)? wr_snoop_add:cacheif.address;
assign r_w = (next_clk_wb)? 1'h0:cacheif.rd_wr;
```

Shown in the below code is the main implementation of the cache directory, here the page identifier and the line state is saved. Depending on the turn of the PC or cache the state line is saved either to the snoop address or the address sent by the CPU.

```
always @(posedge clk) begin
    if(WriteEn_pageid) begin
        page_id [address[7:0]]  <= address[23:8];
    end
    if(pc_cache_turn) line_state [snoop_address[7:0]]  <= wr_line_state;
    else line_state [address[7:0]]  <= wr_line_state;
end
```

Inside the directory, all the calculation for hit and dirty signals are taking place. As shown in the code below there is a need for the page identifier to match only the exact same offset line as well as a specific state, for dirty should be equal to M-state and for hit all but the I-state. Offset is grabbed with [7:0] bits of the corresponding address.

```
assign hit = (page_id[address[7:0]] == address[23:8] && line_state[address[7:0]] != I_STATE)? 1'h1: 1'h0;
assign dirty = (page_id[address[7:0]] == address[23:8] && line_state[address[7:0]] == M_STATE)? 1'h1: 1'h0;

assign hit_snoop = (page_id[snoop_address[7:0]] == snoop_address[23:8] && line_state[snoop_address[7:0]] != I_STATE)? 1'h1: 1'h0;
assign dirty_snoop = (page_id[snoop_address[7:0]] == snoop_address[23:8] && line_state[snoop_address[7:0]] == M_STATE)? 1'h1: 1'h0;

assign rd_line_state =  (pc_cache_turn)? line_state[snoop_address[7:0]]:line_state[address[7:0]];
```

Main Driver

The main driver acts as a Monitor and Driver to avoid the use of mailboxes and message passing. Avoiding these TLM type of transactions the verification work diminishes and focus in the design improves. In the picture below we can check all verification interfaces are connected to the driver so it can do both monitor and driver functions.

```
function new(virtual intfc_pc.driver pc_dr,
             virtual intfc_mm.driver p1_dr,
             virtual intfc_mm.driver p2_dr,
             virtual intfc_mm.driver p3_dr,
             virtual intfc_mm.driver p4_dr,
             virtual intfc_core.driver core_dr);
    this.pcdr = pc_dr;
    this.pagedr[0] = p1_dr;
    this.pagedr[1] = p2_dr;
    this.pagedr[2] = p3_dr;
    this.pagedr[3] = p4_dr;
    this.coredr = core_dr;
endfunction : new
```

Next code shows a simple implementation of a verification test running a CPU read with hit modified, test needs to match the page loaded inside the cache and random offset.

```
CPU_RD_CH_HIT: begin
    $display("###################RUNNING TEST CPU_RD_CH_HIT#########################");
    r_addr.val_addr.constraint_mode(1);
    r_addr.inval_addr.constraint_mode(0);
    i = 0;
    while(i <10) begin
        $display("---------------Cycle %d ----------------------",i+1);
        `SV_RAND_CHECK (r_addr.randomize() with {page_id [15:8] == sel_page;});
        send_read_cache(r_addr.line, r_addr.page_id);
        i = i + 1;
    end
end
```

Next the code for a read cache sent by the CPU is executed. First step is to check if the cache is not doing a write back with the cache_busy signal. Next all signals are driven for the read to take place as well as some for debug purposes. Next a waiting period, including consideration of some control signals, should be consider before checking the reading was done. Finally the checking the vales seeing inside the interface with the checker built up inside a class.

```
task send_read_cache(logic [7:0] line, logic[15:0] page_id);
    if (!coredr.cache_busy) begin
        $display("%s:%0d:%0d ns: SENDING Core Read Transaction", `__FILE__,`__LINE__,$time);
        coredr.address = {page_id,line};
        coredr.wr_data = 32'h0;
        coredr.rd_wr = CPU_READ;
        coredr.dbg_address = coredr.address;
        #(CLK_PERIOD/10);
        while(!coredr.line_found && coredr.pc_cache_turn) @(posedge coredr.clk);
        @(posedge coredr.clk);
        check_mm_value(page_id,line,cache_mm_line);
        if(cache_mm_line.check_cache_value(coredr.rd_data, coredr.dbg_data, coredr.cache_line_state, coredr.
        line_found) != PASS) begin
            $stop;
        end
    end else begin
        $display("%s:%0d:%0d ns: Cache not available doing a write back",`__FILE__,`__LINE__,$time);
    end
endtask
```

Next task shows how the driver also works as a main memory monitor looking for a specific value and its state inside the main memory. In order to do the search there is a need for swapping all the four pages, this is done with a foreach loop with all interfaces, next the Value found message is printed with all the information. The information then will be used for the checker to see is the main memory value and state matches the ones shown for the cache and the PC.

```
task check_mm_value (logic[15:0] page_id, logic [7:0] line, ref memory_info current_mm_line);
    current_mm_line = new(line, page_id);
    foreach(pagedr[i]) begin
        pagedr[i].r_w_page = 0;
        pagedr[i].page_id = page_id;
        pagedr[i].line = line;
        #(0.1);
        if(pagedr[i].found) begin
            current_mm_line.found_count = current_mm_line.found_count + 1;
            current_mm_line.value = pagedr[i].data_mm;
            current_mm_line.valid_state = pagedr[i].state;
            $display("%s:%0d:%0d ns: Value found in MM DATA= 0x%h ADDRESS= 0x%h PAGE = 0x%h status = 0x%h",
            `__FILE__,`__LINE__,$time, current_mm_line.value, {page_id,line},i, current_mm_line.valid_state);
        end
    end
endtask
```

Checkers

Checker for the peripheral component is shown below, it compares first if the value was found inside the main memory, if not if just gives a warning as a specific address may or not be inside memory. Then it checks both state and value of the read PC transaction comparing them if not then it stops the simulation with an error message. Finally it checks if the value was found in memory on more than one page which also triggers an error and a simulation stop.

```
function bit check_pc_value(int comp_value, bit comp_state);
    bit result = FAIL;
    if (valid_state == I_STATE) begin
        state = 0;
    end else begin
        state = 1;
    end
    if (found_count == 0) begin
        $display("%s:%0d:%0d ns: WARNING! Value not found in Main Memory!",
            `__FILE__,`__LINE__,$time);
        result = PASS;
    end else if (found_count == 1) begin
        if (state == comp_state && value == comp_value) begin
            $display("%s:%0d:%0d ns: PASS Correct Data checked in correct state from MM ADDR= 0x%h VAL =
            0x%h",
            `__FILE__,`__LINE__,$time, {page_id,line}, value);
            result = PASS;
        end
        if ((state != comp_state) || (value != comp_value)) begin
            $display("%s:%0d:%0d ns: ERROR! SIMULATION STOP! Data MESI state not correct!! Address 0x%h
            status = 0x%h expected = 0x%h",
            `__FILE__,`__LINE__,$time, {page_id,line},comp_state,state);
            $display("%s:%0d:%0d ns: ERROR!! Incorrect Data detected 0x%h expected 0x%h",
            `__FILE__,`__LINE__,$time, value ,comp_value);
            result = FAIL;
        end

    end else if (found_count > 1) begin
        $display("%s:%0d:%0d ns: ERROR! SIMULATION STOP! More than one Valid Value found in Main Memory!!",
        `__FILE__,`__LINE__,$time);
        result = FAIL;
    end

    return result;
endfunction
```

Cache Checker is basically the same as PC checker but with an extra checker layer if the value is found inside the cache memory and checking also the full MESI states.

```
function bit check_cache_value(int main_cpu_data, int main_cache_data, logic [1:0] state, bit found_cache
);
    bit result = FAIL;
    if(found_cache) begin
        if (value == main_cpu_data && (state==S_STATE || state ==E_STATE)) begin
            $display("%s:%0d:%0d ns: PASS! Hit cache! Correct Data read from cache in E and S state ADDR=
            0x%h VAL = 0x%h",
            `__FILE__,`__LINE__,$time, {page_id,line}, value);
            return PASS;
        end else if(main_cache_data == main_cpu_data && state ==M_STATE) begin
            $display("%s:%0d:%0d ns: PASS!  Hit cache! Data line in MESI M state",
                `__FILE__,`__LINE__,$time);
                return PASS;
        end else if(state==I_STATE) begin
            $display("%s:%0d:%0d ns: Warning!! Data line in MESI I state, looking into MM next",
                `__FILE__,`__LINE__,$time);
                result = PASS;
        end else begin
            $display("%s:%0d:%0d ns: ERROR! SIMULATION STOP! Hit Cache! ! Check simulation State = 0x%h,
            CPU Value = 0x%h, MM Value = 0x%h Cache Value = 0x%h"
                ,`__FILE__,`__LINE__,$time,state, main_cpu_data, value, main_cache_data);
            return FAIL;
        end
    end
```

Test Bench

Test bench is just a big wrapper for connecting all the modules inside the design, the only signals driven by the test bench are the clocks and resets as shown below. The sclk signal is 60 times faster than the clk used by Cache and Main memory in order for PC to match its throughput.

```
// TB Signals
reg sclk=1, clk=0, rst;

// Initial Conditions for Design: reset.
initial begin
    rst = 1'b0;
    #(CLK_PERIOD/CLK_PERIOD) rst = 1'b1;
    $display("DUT Reset Done!");
    #(CLK_PERIOD/CLK_PERIOD) clk = ~clk;
end

//Clock generator with a configurable period
always #(CLK_PERIOD/2) clk = ~clk;
always #(CLK_PERIOD/CLK_PERIOD) sclk = ~sclk;
```

Next a single module connection, all the modules were connected either with wires or interfaces for verification. In the Case shown Memory controller doesn't have any interface connected to the Monitor/Driver as it doesn't require it.

```
mc mc_hw(
    //Clocks
    .clk, .sclk, .rst,
    //PC interface
    .TX_line(TX_MC_RX_PC),
    .RX_line(TX_PC_RX_MC),
    //MM interface
    .read_write(read_write_mcmm_wire),
    .state_in(state_in_mcmm_wire),
    .address_to_mm(address_mcmm_wire),
    .state(state_mcmm_wire),
    .not_found(not_found_mcmm_wire),
    //Cache interface
    .r_w(r_w_MC_Cache_wire),
    .hitm(hitm_MC_Cache_wire),
    .cache_req_mm(cache_req_mm_MC_Cache_wire),
    .address(address_MC_Cache_wire),
    .state_cache(state_cache_MC_Cache_wire),
    .SnoopAddress(SnoopAddress_MC_Cache_wire),
    .state_mm(state_mm_MC_Cache_wire),
    .pc_cache_turn(pc_cache_turn_MC_Cache_wire),
    .nodata_mm(nodata_mm_MC_Cache_wire),
    .invalidate_line(invalidate_line_MC_Cache_wire),
    //common data bus
    .data(data_bus_wire)
);
```

The main initial block is just an instance for the Driver/Monitor class connecting with all debug interfaces and launching a single task that runs a list of verification tests.

```
//++++++++++++++++++++++++++++++++++++++++++++++++++
//   Init Simulation
//++++++++++++++++++++++++++++++++++++++++++++++++++
initial @(rst == 1)
begin
    dr_inst = new (pcdr_inst.driver,
                   page1_inst.driver,
                   page2_inst.driver,
                   page3_inst.driver,
                   page4_inst.driver,
                   cdr_inst.driver);

    $display("Starting selected Tests!");
    dr_inst.launch_tests();
    $display("@@@@@@@@@@@@@@@ Tests Done!!!! PASS!!!!@@@@@@@@@@@@@@@@@@@@");
    $stop;
end
```

Simulation and Verification Results

Verification settings were specifically design to test the most of the possible values that the page identified, address and range of data values possible. In order to achieve it the design of a basic filegen.py script was made in order to inject the value of all 4 pages, the initial state as well as the page identifiers.

Next the algorithm used for the creation of the pages

```
import random

def write_pagid_file(i, max_num, times, file):
    offset = 0xA00
    while (i<=max_num):
        value = i*times+offset
        if (i==max_num):
            file.write('%02x'%value)
        else:
            file.write('%02x\n'%value)
        i = (i+1)*times
```

```
def write_memfile(i, max_num, times, file, offset):
    while (i<=max_num):
        value = i*times+offset
        if (i==max_num):
            file.write('%08x'%value)
        else:
            file.write('%08x\n'%value)
        i = (i+1)
```

It's important to notice that the Page identifier contains the line offset. The main purpose of this is to make the driver simpler so it can beforehand know which the page identifiers already loaded inside the pages are. Driver also can randomize between valid or invalid addresses in main memory according to this information.

For our verification scenarios, the page IDS are set to 16'h0A00 to 16'h0AFF for first page, 16'h0B00 to 16'h0BFF for second page, 16'h0C00 to 16'h0CFF for third page, 16'h0D00 to 16'h0DFF for forth and last page. Matching each last eight bits with the line offset.

For the memory it was initialized with offsets of 0, 1, 2 and 3 respectively, so all the values inside all pages are different and in a series of four.

```
fil = open("mem1.dat","w")
write_memfile(0,255,4,fil,0)
fil.close()

fil = open("mem2.dat","w")
write_memfile(0,255,4,fil,1)
fil.close()

fil = open("mem3.dat","w")
write_memfile(0,255,4,fil,2)
fil.close()

fil = open("mem4.dat","w")
write_memfile(0,255,4,fil,3)
fil.close()
```

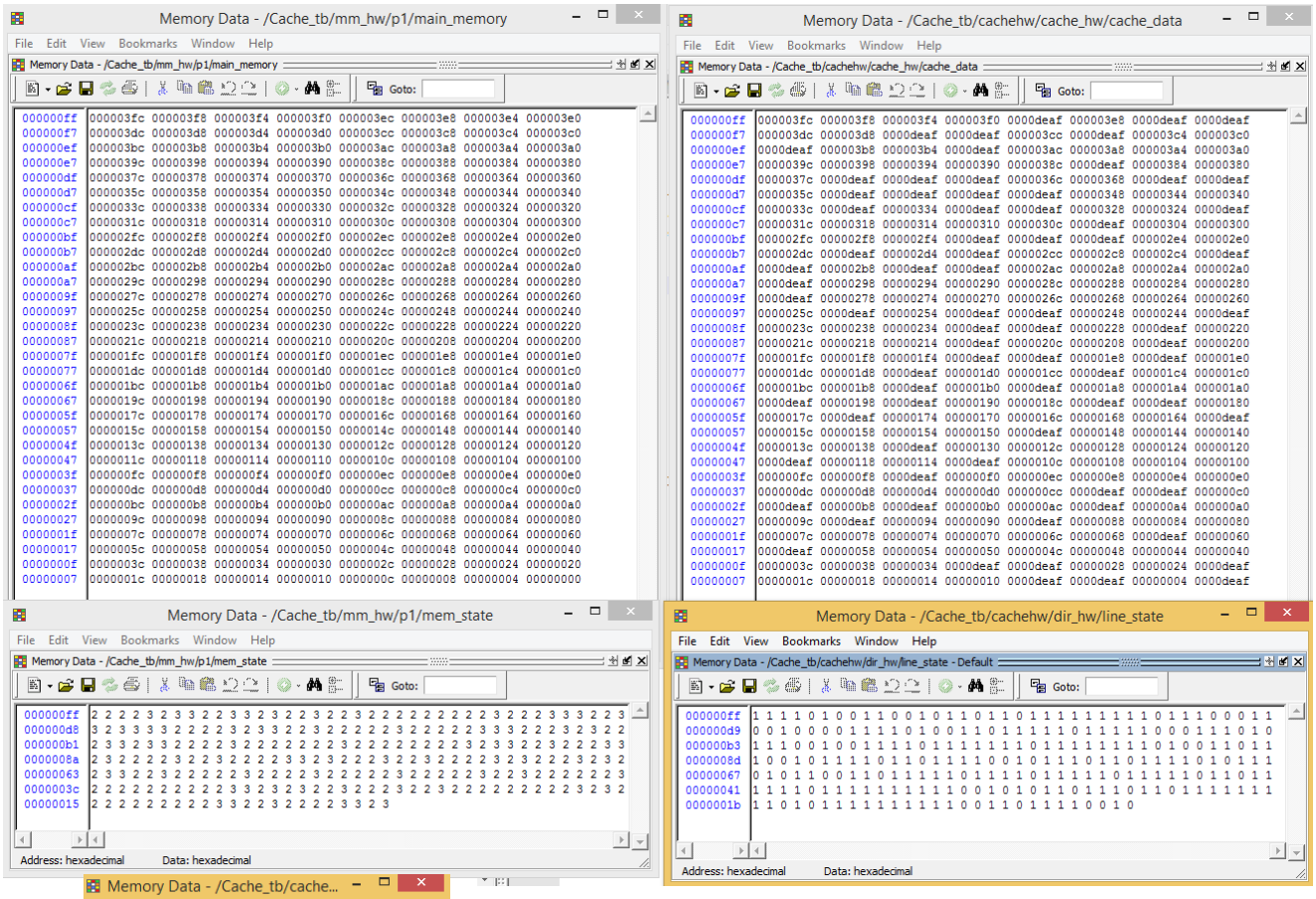
```
fil = open("page1ids.dat","w")
write_pagid_file(0,255,1,fil)
fil.close()
fil = open("page2ids.dat","w")
write_pagid_file(256,511,1,fil)
fil.close()
fil = open("page3ids.dat","w")
write_pagid_file(512,767,1,fil)
fil.close()
fil = open("page4ids.dat","w")
write_pagid_file(768,1023,1,fil)
fil.close()
```

Finally the MESI state file was created randomly inside each page contains an 80% possibility of an Exclusive state and 20% of Invalid state. MESI states definition is an enumeration in order:

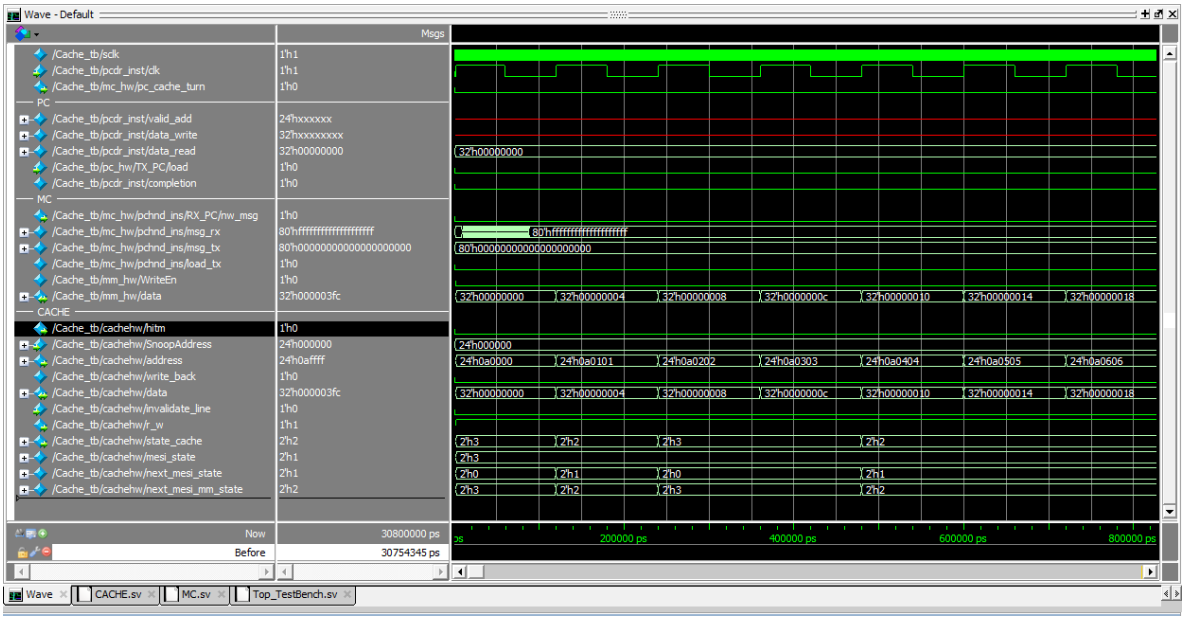
M	2'h0
E	2'h1
S	2'h2
I	2'h3

1. CPU Read with cache miss without PC transaction

CPU read with cache miss is intended to fill the entire cache with one random chosen page. When an invalid state inside the address is detected the cache is filled with 0x0000deaf data value and no write back is performed as it is not required. Check all possible states in cache are M or E states and inside memory are S or I states.



In the left we can see all the page identifier array from the cache has been filled up.



2. *CPU Read with cache hit clean or modified*

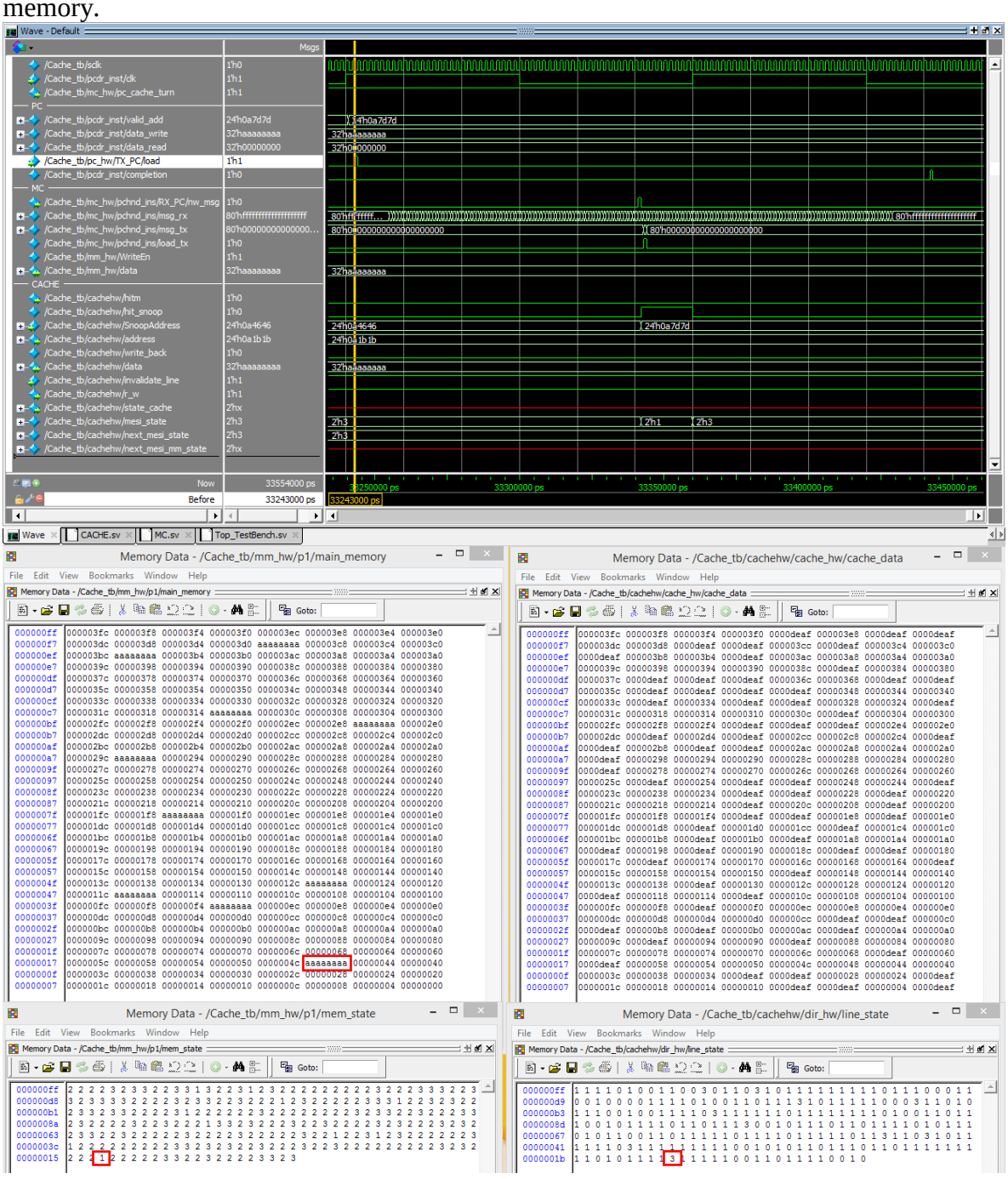
CPU Read works properly with the same page already filled up in the cache, the driver choses a random address for that page and reads it from memory, check the states are E and M states detected and all are cache hits reviewed by the checker.

```
# ---------------Cycle          10 ------------------------
# GenDriver.sv:240:31920 ns: SENDING Core Read Transaction
# GenDriver.sv:337:32040 ns: Value found in MM DATA= 0x0000006c ADDRESS= 0x0a1b1b PAGE = 0x00000000 status = 0x2
# SysDefines.sv:122:32040 ns: PASS! Hit cache! Correct Data read from cache in E and S state ADDR= 0x0a1b1b VAL = 0x0000006c

 ---------------Cycle           7 ------------------------
 GenDriver.sv:240:31560 ns: SENDING Core Read Transaction
 GenDriver.sv:337:31680 ns: Value found in MM DATA= 0x00000098 ADDRESS= 0x0a2626 PAGE = 0x00000000 status = 0x3
 SysDefines.sv:126:31680 ns: PASS!  Hit cache! Data line in MESI M state
```

3. *PC Write Transaction to MM with hit clean*

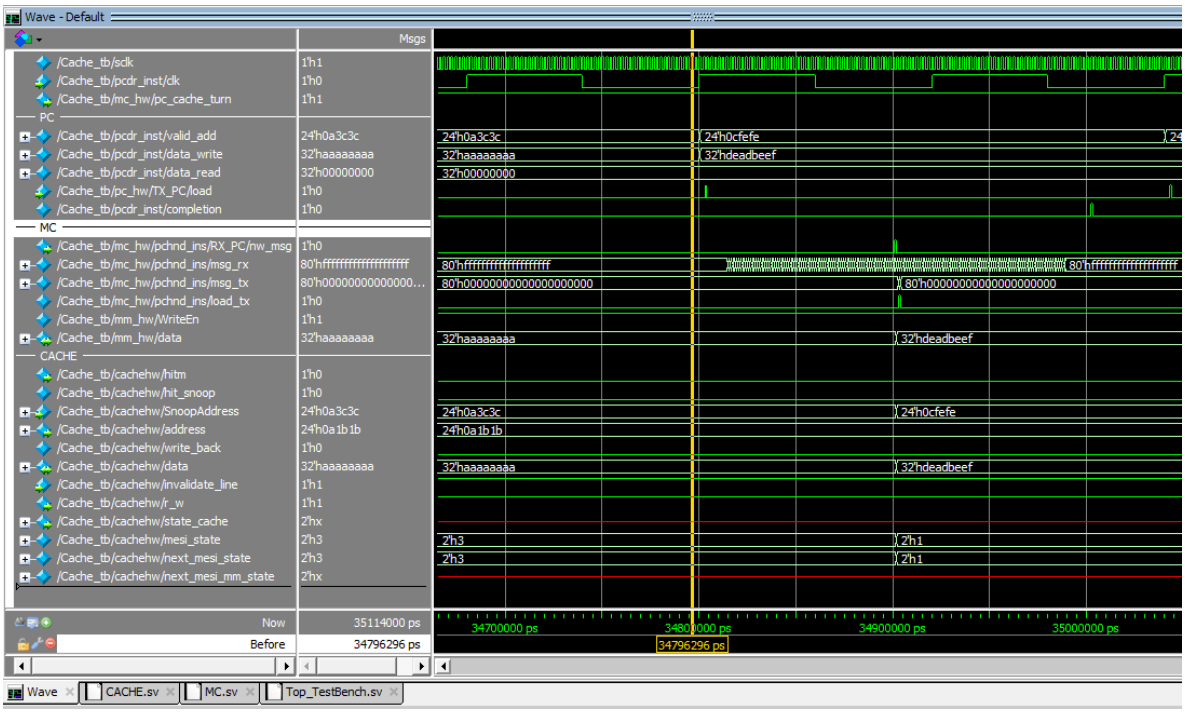
PC Write with MM hit clean mainly exercises the snoop mechanism to send the line in the cache to invalid state (3) and the main memory line states to exclusive state (1). Test consists in 10 random address writes of 0xAAAAAAAA value into the page inside cache memory.



4. PC Write Transaction to MM with miss

PC Write transaction writes 0xdeadbeef with miss and verifies the value is in main memory.

```
---------------Cycle           1 ----------------------
GenDriver.sv:301:34800 ns: SENDING PC Write Transaction
GenDriver.sv:337:35040 ns: Value found in MM DATA= 0xdeadbeef ADDRESS= 0x0cfefe PAGE = 0x00000002 status = 0x1
SysDefines.sv:97:35040 ns: PASS Correct Data checked in correct state from MM ADDR= 0x0cfefe VAL = 0xdeadbeef
---------------Cycle           2 ----------------------
GenDriver.sv:301:35040 ns: SENDING PC Write Transaction
GenDriver.sv:337:35280 ns: Value found in MM DATA= 0xdeadbeef ADDRESS= 0x0b1818 PAGE = 0x00000001 status = 0x1
SysDefines.sv:97:35280 ns: PASS Correct Data checked in correct state from MM ADDR= 0x0b1818 VAL = 0xdeadbeef
---------------Cycle           3 ----------------------
GenDriver.sv:301:35280 ns: SENDING PC Write Transaction
GenDriver.sv:337:35520 ns: Value found in MM DATA= 0xdeadbeef ADDRESS= 0x0d2424 PAGE = 0x00000003 status = 0x1
SysDefines.sv:97:35520 ns: PASS Correct Data checked in correct state from MM ADDR= 0x0d2424 VAL = 0xdeadbeef
```



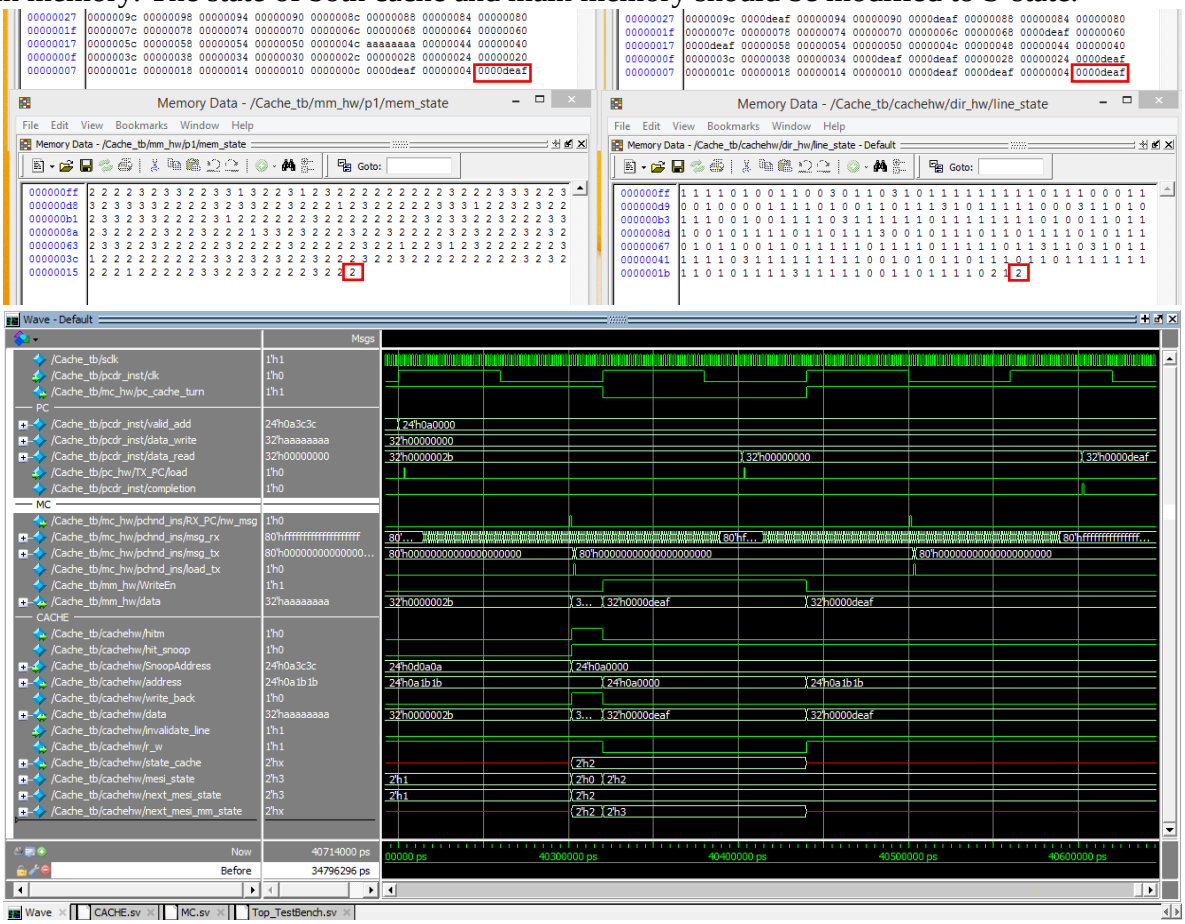
5. PC Read Transaction to MM with hit clean or miss

PC Read transaction read the value from memory with miss and verifies the value is the same as in main memory.

```
# ---------------Cycle          10 ----------------------
# GenDriver.sv:278:39720 ns: SENDING PC Read Transaction
# GenDriver.sv:337:39922 ns: Value found in MM DATA= 0x0000002b ADDRESS= 0x0d0a0a PAGE = 0x00000003 status = 0x1
# SysDefines.sv:97:39922 ns: PASS Correct Data checked in correct state from MM ADDR= 0x0d0a0a VAL = 0x0000002b
```

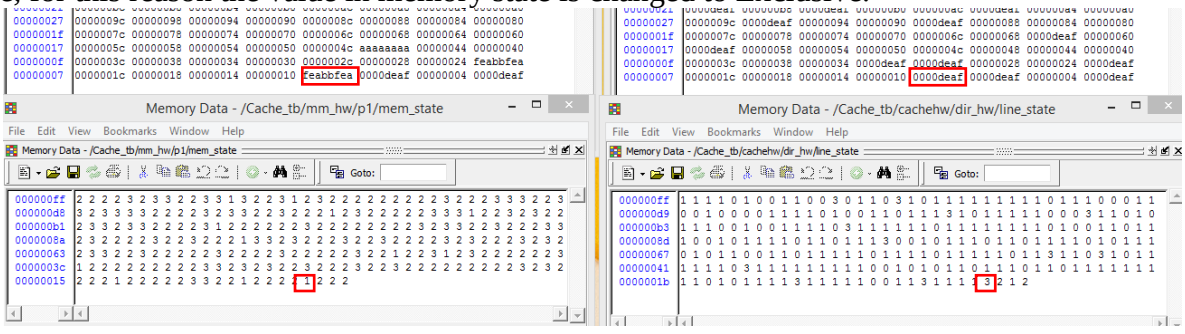
6. *PC Read Transaction to MM with hit modified*

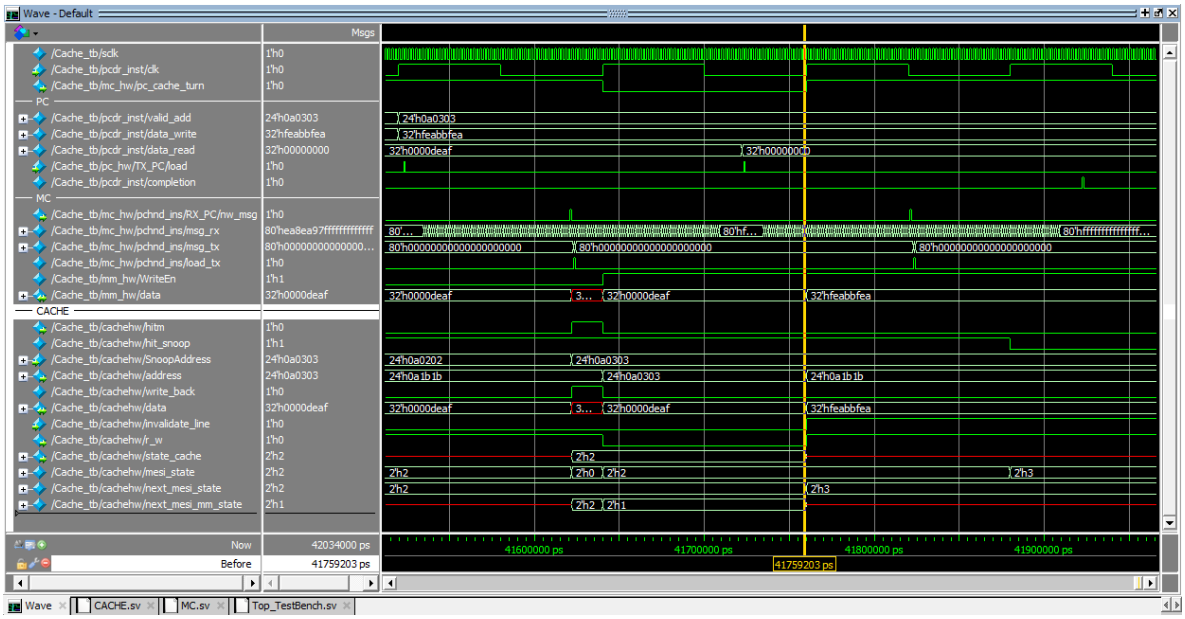
In this example only the modified data from cache is tried to be accessed by the PC. For this scenario a write back mechanism must be triggered in order to send the data from cache to main memory. The state of both cache and main memory should be modified to S-state.



7. *PC Write Transaction to MM with hit modified*

For this test case the PC writes an address modified by the CPU in cache memory, it tries to write 0xFEABBFEA value, but first the cache writes back the value to then send the line to I-state, for this reason the value in memory state is changed to Exclusive.

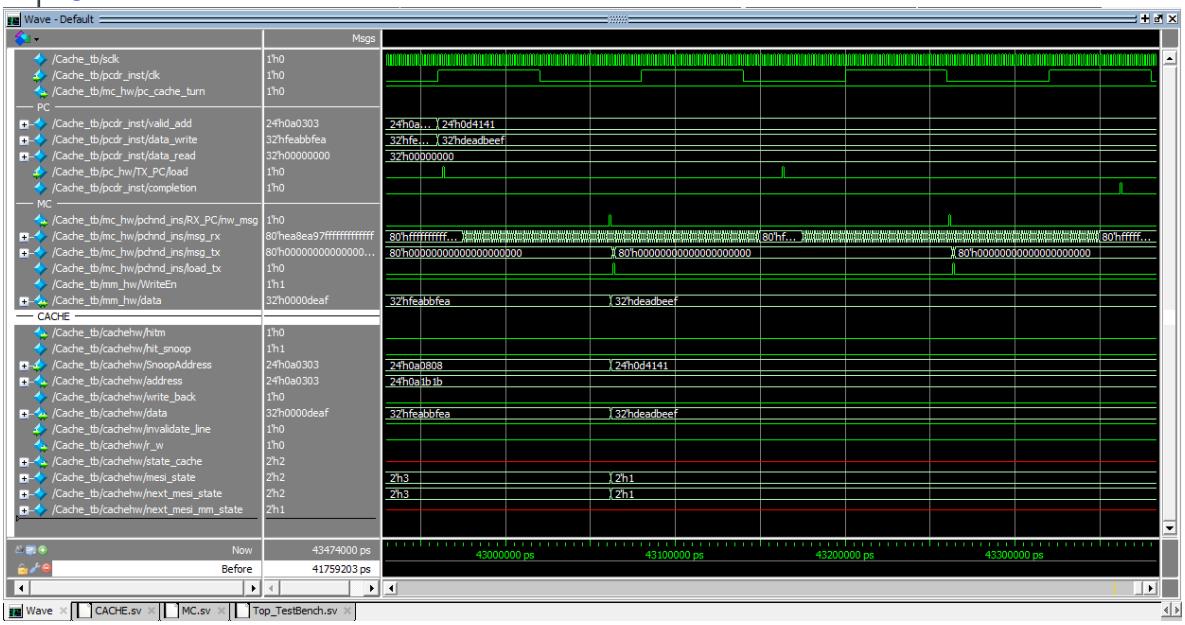




8. *PC Error Retry test*

Sending a message with the PC with an error injected and still finishes correctly.

```
# ---------------Cycle          10 ------------------------
# GenDriver.sv:278:39720 ns: SENDING PC Read Transaction
# GenDriver.sv:337:39922 ns: Value found in MM DATA= 0x0000002b ADDRESS= 0x0d0a0a PAGE = 0x00000003 status = 0x1
# SysDefines.sv:97:39922 ns: PASS Correct Data checked in correct state from MM ADDR= 0x0d0a0a VAL = 0x0000002b
```



9. *PC to MM Data Coherency check*

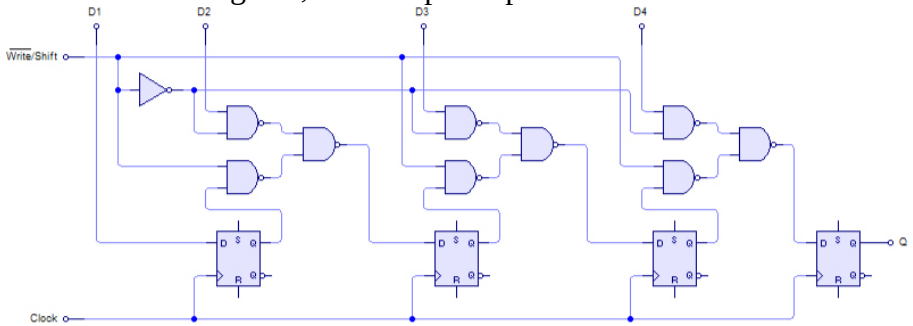
All the other write with hit and with miss also check the data integrity so this test is the same PC reads and writes.

Conclusion

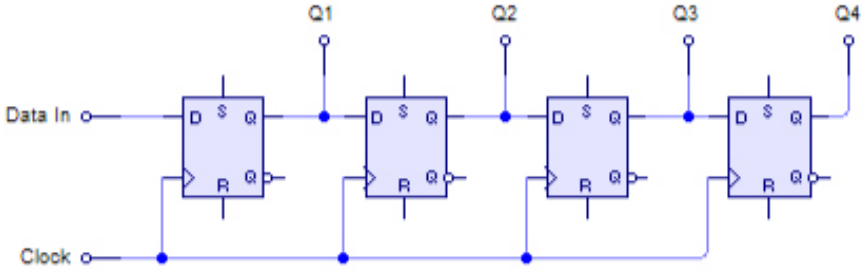
In this project, System CACHE with MESI implementation was a really complete example of how is the process to architecture, design, implement and validate a complete system. By the hang we are exercise the concepts of CACHE memory, main memory, serial peripheries, controllers, drivers and checkers.
We starting reviewing the requirements of this system and separate in the principal blocks and communications that we need it. At same time starting with the architecture and definition in how all block needed to work.
The next step that we work is in test plan that we will cover to valid if the system works properly, and this information give us the information to how we do the observability of system. And with the architecture and test plan we decide all the interfaces between the units.
When we started the implementation of system on System Verilog, we discover that in this step we need to implement more in cases that we expected, because the MESI protocol control was more complicate that we see in the first step.
And last we generate 8 tests to valid the functionality off all system, that cases are derigged to main functionality another's to full cover.

Glossary

Write back: CPU writes inside cache if the data is not necessary to update into main memory. Refresh MM only until a snoop is done or an S-sate data needs to be written.
Write through: CPU write always to the main memory, has the advantage of having the last data all the time but it more inefficient for cache usage.
MC: Memory Controller unit arbiters between main memory and all agents that try to access it. In this implementation cache and PC.
PC: Peripheral Component unit is considered any type of peripheral which requires data interaction with main memory.
CACHE: Also defined as cache, unit storing a copy of data of Main Memory. It is really fast to access for the CPU and it must follow MESI coherency protocol.
MM: Main memory of the system also considered the RAM. It has high latency to access for data.
Clean data: A data inside the cache that has not been modified by the CPU. It can be in E or S state.
Dirty data: A data inside the cache that was modified by the CPU. It must be in M-state.
PISO: Parallel In Serial Out register, an example implementation is shown below.



SIPO: Serial In Parallel Out register, an example implementation is shown below.



Bibliography

- *Bigelow, Narasiman, Suleman. "An evaluation of Snoopy Based Cache Coherence protocols" (PDF). ECE Department, University of Texas at Austin.*