Universidad Nacional del Sur

Tesis de Magíster en Ingeniería

Microprocesador RISC basado en MIPS32 con conectividad AMBA AHB-Lite

Ariel Oroz De Gaetano

BAHÍA BLANCA ARGENTINA

10 de febrero de 2019

Prefacio

Esta tesis se presenta como parte de los requisitos para optar al grado Académico de Magíster en Ingeniería, de la Universidad Nacional del Sur y no ha sido presentada previamente para la obtención de otro título en esta universidad u otra. La misma contiene los resultados obtenidos en investigaciones llevadas a cabo en el ámbito del Departamento de Ingeniería Eléctrica y de Computadoras durante el período comprendido entre el 1 de julio de 2013 y el 1 de julio de 2015, bajo la dirección del Dr. Martín Di Federico y el Dr. Pedro Julián.

Resumen

En este trabajo se presenta el estudio de arquitecturas de bus y la implementación de AMBA AHB-Lite en el marco de un sistema que incluye un microprocesador basado en la arquitectura MIPS32, cuya interfaz es adaptada para ser compatible con las especificaciones del protocolo mencionado. Se realiza una comparación del sistema implementado con respecto a otras arquitecturas estándar que emplean procesadores como Cortex-M0, OpenRISC1200, ZPU y LEON3, contrastando los requisitos que cada uno insume en sus implementaciones en FPGA y el desempeño alcanzado por cada uno.

Abstract

This work presents the study of bus architectures and the implementation of AMBA AHB-Lite in the context of a system that includes a microprocessor based on the MIPS32 architecture, whose interface is modified to comply with the protocol's specifications. The implemented system is compared with other standard architectures that employ microprocessors such as Cortex-M0, OpenRISC1200, ZPU y LEON3, examining the requirements needed for each case in their FPGA implementation and the performance yielded by each of them.

Índice general

Índice de figuras

Índice de tablas

Abreviaturas y terminología

AHB-Lite	**A**dvanced **H**igh-Performance **B**us **Lite**
ALU	**A**rithmetic-**L**ogic **U**nit
AMBA	**A**dvanced **M**icrocontroller **B**us **A**rchitecture
CISC	**C**omplex **I**nstruction **S**et **C**omputer
CPU	**C**entral **P**rocessing **U**nit
DMA	**D**irect **M**emory **A**ccess
DSM	**D**eep **S**ub**m**icron
DSP	**D**igital **S**ignal **P**roccesor
EX	**E**xecution
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FPR	**F**loating **P**oint **R**egister
FPU	**F**loating **P**oint **U**nit
GPR	**G**eneral **P**urpose **R**egister
ID	**I**nstruction **D**ecode
IF	**I**nstruction **F**etch
IP	**I**ntellectual **P**roperty
IR	**I**nstruction **R**egister
LIFO	**L**ast **I**n **F**irst **O**ut
MIPS	**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages
MPSoC	**M**ulti-**P**rocessor **S**ystem-**o**n-**C**hip
PC	**P**rogram **C**ounter
RAW	**R**ead **A**fter **W**rite
RISC	**R**educed **I**nstruction **S**et **C**omputer
RR	**R**ound **R**obin
SoC	**S**ystem-**o**n-**C**hip

SPARC	**S**calable **P**rocessor **ARC**hitecture
TDMA	**T**ime **D**ivision **M**ultiple **A**ccess
VLSI	**V**ery **L**arge **S**cale **I**ntegration
WAR	**W**rite **A**fter **R**ead
WAW	**W**rite **A**fter **W**rite
WB	**W**rite **B**ack
Acknowledge	Señal de confirmación a una petición o pauta
Arbiter	Dispositivo que administra el uso del bus
Bit	Unidad básica en el sistema de representación binario
Buffer	Medio regulador de la tasa de transferencia de datos
Bus	Conjunto de cables que provee un medio de transmisión de señales
Byte	Conjunto de 8 bits
Datapath	Conjunto de unidades funcionales que proveen medios de almacenamiento y transmisión de datos
Delay slot	Espacio que ocupa una instrucción, inmediatamente posterior en ejecución a un salto, en el *pipeline*
Double word	Conjunto de 64 bits
Fetch	Acción de buscar una nueva instrucción
Flag	Bit de indicación de determinado acontecimiento del sistema
Half word	Conjunto de 16 bits
Master	Dispositivo que inicia transacciones en un sistema de comunicación
Offset	Concepto de desplazamiento respecto a una referencia
Pipeline	Concepto de paralelización de tareas análogo a la de una línea de montaje
Request	Señal de petición o pauta
Round Robin	Referencia a secuencia en orden circular
Slave	Dispositivo que responde a transacciones en un sistema de comunicación
Word	Conjunto de 32 bits

Capítulo 1

Introducción

1.1. Visión general

Los avances tecnológicos desarrollados en silicio permiten hoy en día desarrollar chips que poseen transistores en el orden de los miles de millones, y posibilitan duplicar esa cantidad cada dos años (Moore, 1965). Dichos chips que incorporan circuitos integrados se denominan SoC (por la sigla del inglés *System-On-Chip*). Los avances mencionados, en conjunto con la creciente demanda de cómputo requerida en los dispositivos comerciales actuales, convergen en el desarrollo de chips que cuentan con más de un procesador. En la tendencia de diseño actual de SoC surgen los MPSoC (del inglés *Multiprocessor System-On-Chip*) que constan de múltiples procesadores y cientos de componentes donde se incluyen memorias, DMAs, periféricos e interfaces externas integrados todos mediante una arquitectura de bus. Toda esta diversidad de elementos debe funcionar en conjunto de la manera más eficiente posible, amoldándose a los requerimientos de la aplicación para la que se diseña el chip, y es por esta razón que es fundamental la selección de una arquitectura de comunicación que se ajuste a dichos requerimientos.

Cuando la complejidad del sistema aumenta, las decisiones de diseño y variantes de optimización del mismo también lo hacen. Como consecuencia, la exploración del espacio de diseño resulta una tarea que insume un tiempo significativo en el proceso de diseño de sistemas y una posible manera de reducir los plazos consiste en utilizar bloques funcionales ya implementados, conocidos también como núcleos IP (sigla del inglés *Intellectual*

Property) como detalla Mathaikutty y Shukla, 2009. Estos bloques prediseñados son empleados para implementar funciones predefinidas, focalizándose en el aspecto reutilizable de los mismos. La reutilización también acelera la implementación de los sistemas impactando en distintas etapas del flujo de fabricación: diseño, testeo y verificación Werner, 2005.

El incremento en cantidad y variedad de componentes integrados en los diseños MPSoC y la creciente complejidad de las aplicaciones, ha llevado a que la comunicación entre los componentes integrados *on-chip* cumpla un rol crítico en la satisfacción de los requerimientos de desempeño. Según indican las estimaciones mostradas en el *International Technology Roadmap for Semiconductors* (Semiconductor Industry Association, 2015), la brecha entre el retardo de propagación de las señales en las líneas de interconexión y el de las compuertas muestra un incremento significativo en cada paso de miniaturización de la tecnología. Problemáticas de esta índole son corrientes si se considera el significativo número de transferencias de datos que acontecen simultáneamente en los diseños actuales. Para sobrellevar los requerimientos cada vez más exigentes de estos sistemas, las arquitecturas de comunicación *on-chip* también han experimentado cambios en dos aspectos fundamentales. En primer lugar, las arquitecturas más avanzadas emplean un mayor número de componentes lógicos y cables produciendo como consecuencia un incremento de consumo y requerimiento de área. En segundo lugar, las combinaciones entre topologías y protocolos para un sistema de comunicación crean un espacio diversificado de configuraciones que demanda mayor tiempo de evaluación durante el diseño (Pasricha y Dutt, 2008).

La propuesta de este trabajo se gesta en el contexto del proyecto de diseño de un chip para decodificación de televisión digital, cuyo estudio de factibilidad ya manifiesta cómo desde comienzos de la década del 2010 el diseño de estos sistemas incorporaba una cantidad creciente de bloques IP de distintos fabricantes y proveedores, surgiendo la necesidad de proveer una alternativa de comunicación entre los mismos mediante una arquitectura estándar.

El trabajo de la tesis propone el estudio de distintas arquitecturas de bus actualmente utilizadas en la industria, de acceso libre y popularmente divulgadas como CoreConnect de IBM, WISHBONE de Silicore Corporation y AMBA de ARM, y la implementación de la arquitectura AMBA AHB-Lite en un sistema que incluye un procesador basado en

arquitectura MIPS32. Para ello, fue necesario modificar la interfaz del procesador para que resulte compatible con las especificaciones del protocolo de bus.

La implementación del diseño del sistema se realiza mediante el lenguaje de descripción de hardware Verilog. El funcionamiento del sistema se corrobora mediante simulaciones. También se desarrolla un programa ensamblador para el lenguaje MIPS Assembly, que permite programar el procesador implementado. Además se realiza una comparación del sistema implementado con respecto a otras arquitecturas estándar: Cortex M0, LEON3, OpenRISC 2000 y ZPU. El fin es obtener una noción de requisitos y desempeño del sistema propuesto, sintetizándolo en la FPGA Spartan-6 XC6SLX45, y comparando los recursos insumidos por éste respecto a los requeridos al sintetizar cada arquitectura de los otros procesadores. Se compara también el desempeño ejecutando un mismo programa de ordenamiento en cada caso, obteniendo datos de frecuencia de operación, ciclos de reloj necesarios para completar la tarea y tiempo total de ejecución.

1.2. Objetivos del trabajo

El objetivo general de la tesis es investigar y desarrollar una interfaz de comunicación ampliamente utilizada como AMBA AHB-Lite, acoplada en el marco del diseño de un procesador. Para ello se proponen los siguientes objetivos puntuales:

- Estudiar los protocolos de comunicación de bus CoreConnect, WISHBONE, AMBA AHB-Lite.
- Estudiar la arquitectura del microprocesador MIPS32 y sus posibles variantes de implementación.
- Realizar la implementación del microprocesador con un conjunto de instrucciones reducido, adecuado para su empleo como bloque IP en aplicaciones FPGA. La implementación debe realizarse utilizando el lenguaje de descripción de hardware Verilog.
- Modificar la arquitectura del microprocesador para que sus instrucciones cumplan las especificaciones AMBA AHB-Lite. Incorporar el microprocesador en un sistema con los elementos básicos especificados por el protocolo para su posterior prueba.

- Establecer una comparación de requerimientos y desempeño de la implementación realizada con respecto a otros microprocesadores.

1.3. Marco de trabajo

Este trabajo se enmarcó dentro de los siguientes proyectos de investigación:

- FS TICS 001 TEAC 2010 : Plataforma Tecnológica para Sistemas de Tecnología Electrónica de Alta Complejidad.
- PICT 2010 2657 : 3D Gigascale Integrated Circuits for Nonlinear Computation, Filter and Fusion with Applications in Industrial Field Robotics.
- PAE 37079 : Proyecto Integrado en el Área de Microelectrónica para el Diseño de Circuitos Integrados.

El mismo se realizó mediante la siguiente beca:

- Proyecto FS TICS N° 001/10 - Agencia Nacional de Promoción Científica y Tecnológica (ANPCyT) a través de FONARSEC. Beca de Maestría tipo inicial. Periodo: Agosto 2013 – Agosto 2015.

La herramienta utilizada para la realización del diseño, Xilinx ISE WebPACK Design Suite, es provista por Xilinx Inc. como versión de licencia libre. La misma se empleó en la descripción y simulación del hardware, así también como en la síntesis.

1.4. Estructura de la tesis

Este documento se encuentra estructurado de la siguiente manera:

- En el Capítulo 2 se presentan todos los conceptos que constituyen el marco teórico del trabajo realizado. En primer lugar se realiza una introducción al concepto arquitectura de bus y sus variantes topológicas básicas. A continuación, se introducen los protocolos de bus, sus definiciones y posibilidades que brindan.

- En el Capítulo 3 se desarrolla un sistema basado en la arquitectura MIPS32 que utiliza el protocolo de comunicación AMBA AHB-Lite. Para ello se adapta el conjunto de instrucciones y se incorpora la lógica de control y *datapath* necesaria.
- El Capítulo 4 explica las tareas realizadas para la verificación del funcionamiento del sistema. Además se muestran los resultados de su síntesis en FPGA, así como la comparación de su desempeño frente a sistemas con otros procesadores también sintetizados en esa plataforma.
- Finalmente se expone la conclusión del trabajo.
- El documento cuenta con tres apéndices, en el A se incluye un breve repaso del concepto de arquitectura de computadoras, conjunto de instrucciones y el método de *pipeline*. En el B se detallan las características de la arquitectura MIPS32, su conjunto de instrucciones y el detalle de las instrucciones implementadas de la arquitectura para el sistema propuesto. Finalmente en el C se incluyen los códigos en lenguaje ensamblador que se utilizaron para la verificación y simulación.

1.5. Contribución

El trabajo realizado en esta tesis se encuentra plasmado en la publicación *Microprocesador RISC basado en MIPS32 con conectividad AMBA AHB-Lite*, del Congreso Bienal Argentino ARGENCON en su edición del año 2016. El microprocesador implementado se incorpora a la biblioteca de bloques IP del Centro de Micro y Nanoelctrónica del Instituto Nacional de Tecnología Industrial, cuya sede de Bahía Blanca fue el lugar físico donde se realizó el trabajo descripto en este documento.

Capítulo 2

Conceptos preliminares

2.1. Arquitecturas de bus

2.1.1. Introducción

Entre los tipos más sencillos y elementales de arquitecturas de comunicación se encuentra la de bus compartido esquematizado en la Fig. 2.1, que consiste en una serie de cables paralelos en donde se conectan todos elementos del chip. Para efectuar una transferencia de datos un componente debe tomar el control del bus en el momento de llevarla a cabo, y dado que el bus es compartido por todos, ninguno de los elementos restantes puede transferir información mientras el bus se encuentre en uso. Como consecuencia inmediata de esto, la arquitectura de bus compartido gana en simplicidad lo que pierde en paralelismo y desempeño del sistema, volviéndose impracticable en los sistemas en auge con múltiples procesadores.

Una variante que mantiene cierto grado de simplicidad aumentando el paralelismo es la arquitectura de bus compartido por jerarquías de la Fig. 2.2. Todos los componentes que operan a mayor frecuencia se conectan en los buses de alta jerarquía mientras que los de bajo desempeño y operación más lenta se conectan en las más bajas. La comunicación entre elementos de distintas jerarquías se lleva a cabo mediante puentes.

Alternativamente, existen arquitecturas como la de anillo (Fig. 2.3), donde un conjunto de buses unidireccionales y concéntricos conectan los elementos brindando la posibilidad de transferir datos con alto ancho de banda y frecuencia de operación.

Finalmente, existe una combinación entre el bus compartido y conexiones punto a punto

en la arquitectura conocida como cruzada y mostrada en la Fig. 2.4. Los procesadores y sus componentes locales se conectan a través de la arquitectura de bus compartido y dichos buses se conectan punto a punto acorde a la necesidad de la aplicación. Pueden hallarse en la práctica variaciones y combinaciones de las arquitecturas mencionadas, pero esencialmente todas están constituidas por dos características que son la topología y los parámetros de protocolo. La primera hace referencia a cómo los buses se interconectan entre sí en la arquitectura, mientras que la segunda refiere a las características del bus como su ancho de banda, frecuencias de reloj, tamaño de *buffers*, esquemas de arbitraje y otros factores determinados por el protocolo de la arquitectura.

Un protocolo de bus se utiliza para definir las formalidades involucradas en una transacción de datos entre componentes del chip; en particular, define el procedimiento a través del cual un dispositivo requiere una transacción, toma control del bus, transmite y espera señales de confirmación que concluyen la comunicación. En arquitecturas de comunicación, el término utilizado para los dispositivos que requieren el control del bus es "maestro" (derivado del inglés *master*) y su caso más común es el procesador, aunque no es el único que puede iniciar una transacción de datos. Los componentes que responden a estas transacciones son los denominados esclavos (derivado de *slaves* en inglés) y el ejemplo más corriente es una memoria. Ambos poseen respectivamente un conjunto de señales que conforman los puertos maestro y esclavo y se conectan al bus de datos. El encargado de administrar la posesión del bus por parte de los distintos maestros se conoce como árbitro (*arbiter* en inglés) que en conjunto con los puentes y decodificadores forma parte de los componentes elementales de una arquitectura de comunicación. Los puentes son los encargados de comunicar dos buses de jerarquías diferentes y poseen

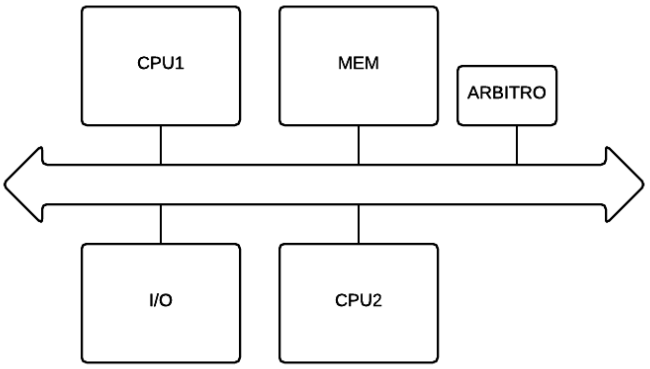


FIGURA 2.1: Diagrama de sistema de bus compartido.

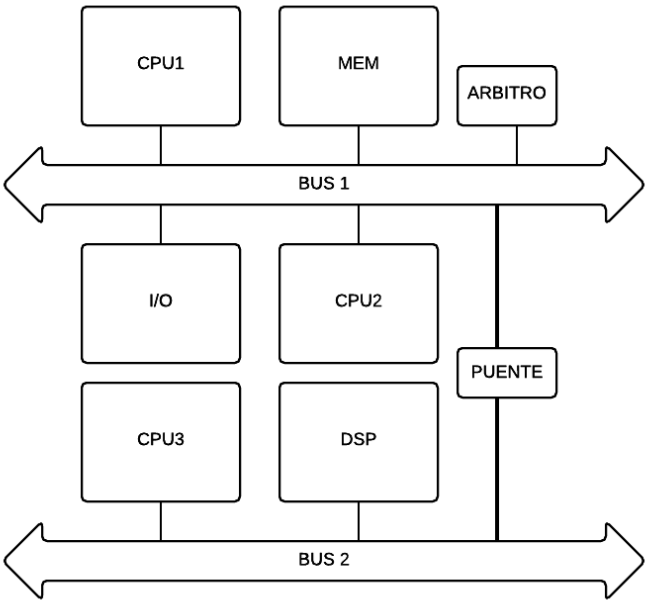


FIGURA 2.2: Diagrama de sistema de bus compartido jerárquico.

ambas interfaces (maestro y esclavo) como se aprecia en la Fig. 2.5 donde el puente recibe datos del bus 1 para transferirlos al bus 2.

Los principales tipos de señales de bus son tres: de dirección, de datos y de control. El conjunto de señales de dirección se denomina bus de direcciones, y su propósito es transmitir la dirección de destino para una transferencia de datos. El número de señales usualmente empleado es potencia de dos. Existe la posibilidad de implementar dos buses de direcciones, uno para las operaciones de escritura y otro para las de lectura, brindando mayor concurrencia al sistema a costa de una penalización de área. El conjunto de señales de datos se conoce como bus de datos y es por sus cables que se transmiten

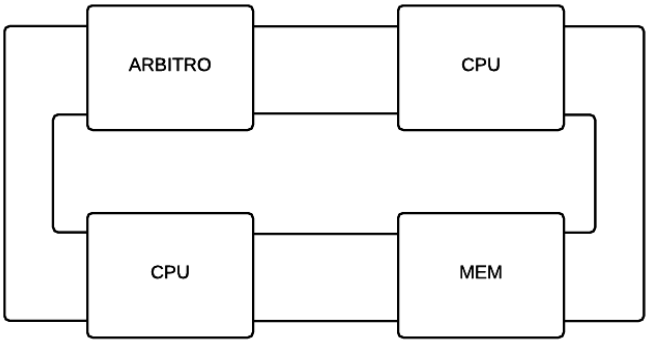


FIGURA 2.3: Diagrama de sistema de bus en anillo.

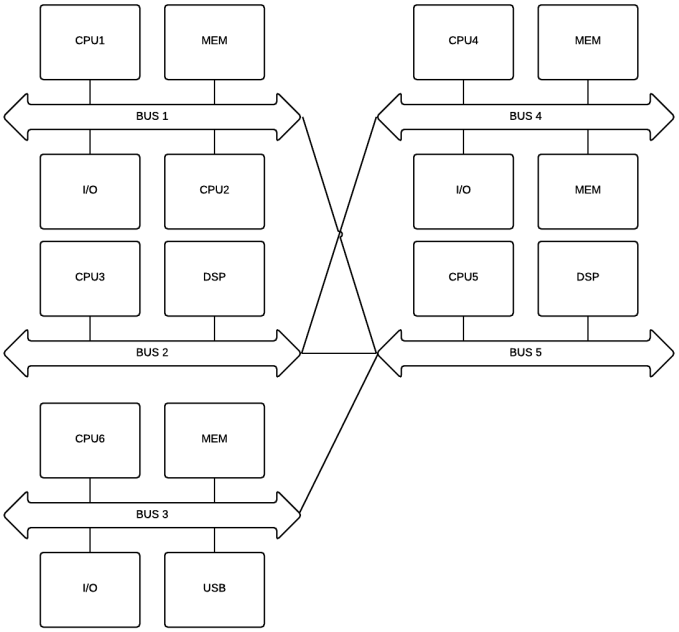


FIGURA 2.4: Diagrama de bus cruzado.

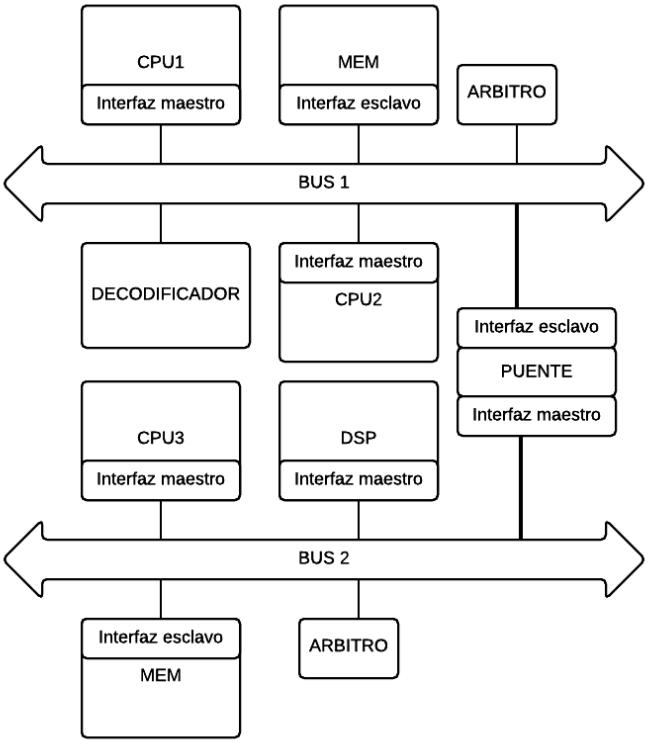


FIGURA 2.5: Diagrama de bus jerárquico con interfaces maestro-esclavo.

los datos de una transacción. De igual manera que en el bus de direcciones se acostumbra a utilizar una cantidad de señales que es potencia de dos, definiendo su ancho de

bus, y puede implementarse un bus de datos para escritura y otro para lectura. Otra alternativa es utilizar las mismas señales para datos y direcciones, multiplexando el uso de las mismas, que es adecuado para aplicaciones que no requieren alto rendimiento. Finalmente las señales de control del bus son todas las asociadas con el protocolo de la arquitectura y que aseguran la transmisión confiable de los datos. Entre ellas se encuentran esencialmente las señales utilizadas para requerir la posesión del bus (conocida con en inglés como *request*), la respuesta que garantiza dicha petición (*acknowledge*, término inglés que denota confirmación), señales de habilitación suministradas por el árbitro, señales de error de transacción provistas por los esclavos hacia los maestros y otras señales que brindan información sobre las características del paquete de datos que se transmite. Si entre las señales de control la arquitectura cuenta con un reloj se la considera de bus sincrónico. Éste funciona para el sistema como base temporal sobre la cual cada paso de una transacción de datos toma lugar. Los buses sincrónicos permiten transferencias rápidas pero requieren el uso de conversores de frecuencia a lo largo de la arquitectura debido a que en los chips actuales la mayoría de los dispositivos operan a frecuencias distintas a las del bus. También es necesaria la inserción de *buffers* a lo largo de señales de reloj que se propagan por el chip para cumplir con restricciones de temporizado. Contrariamente, un bus asincrónico funciona sin la señal de reloj y para garantizar la segura transacción de datos se requieren de otras señales suministradas por los componentes emisores y receptores de la comunicación. Los buses asincrónicos son de operación más lenta en comparación con su contraparte sincrónica debido a que requieren más señales de control para asegurar una transferencia. Su ventaja reside en que no necesita de conversores de frecuencia, y por lo tanto requieren menos área.

Cuando la arquitectura de comunicación es de bus compartido se debe contar con un componente cuya labor sea identificar el dispositivo receptor de la transacción. En los chips se diseña un mapa de direcciones en donde cada componente se encuentra localizado, y es el decodificador de la arquitectura quien se encarga de habilitar adecuadamente los dispositivos receptores de datos de cada transferencia según la dirección indicada por el maestro. Existen dos paradigmas principales de decodificación, centralizada y distribuida. Un esquema del primer tipo es como el mostrado en la Fig. 2.5, donde el decodificador es un componente más conectado al bus y opera coordinadamente con el árbitro. La ventaja de esta organización reside en que el sistema puede extenderse con facilidad siempre y cuando se cuente con capacidad de direccionamiento para los nuevos

componentes. La segunda organización de decodificación consiste en que cada esclavo posea su propio decodificador en su interfaz, cuando una transacción se inicia el maestro emite la dirección de destino que es atendida por todos los esclavos pero solo confirmada por el esclavo cuya dirección es objetivo. Esta implementación requiere menor cantidad de señales de control pero no brinda la simplicidad de adición de nuevos componentes y requiere de mayor utilización de lógica, ocupando más área.

En párrafos anteriores se mencionó la función y la importancia de un árbitro en la arquitectura de bus compartido, a continuación se reseña brevemente los diferentes esquemas de arbitraje mayormente utilizados, que son los que determinan el criterio con el cual se garantiza a los múltiples maestros el control del bus para una transacción. Cualquiera sea el esquema debe garantizarse que todos los esclavos accedan al bus de manera equitativa, asegurando que las transacciones se completen de forma correcta y demorando el tiempo mínimo necesario.

El esquema más simple es el de prioridad estática, en donde cada maestro tiene una prioridad fija asignada. Cuando más de un maestro debe ocupar el bus, es el de mayor prioridad quien siempre gana acceso. Esto puede llevarse a cabo de manera preferencial, implicando que una transferencia puede ser interrumpida si un maestro de mayor prioridad al que controla el bus necesita acceso; o de forma no preferencial, en cuyo caso no puede interrumpirse el control del bus. Si bien este esquema resulta atractivo por su simpleza, es vulnerable a impedir un acceso prolongado al bus a maestros con prioridad o rendimiento más bajo, pudiendo provocar retrasos en el funcionamiento de la arquitectura.

Un esquema alternativo que se asegura la equidad de acceso de los maestros es conocido como *Round Robin* (término en inglés referente a una ronda circular), en el cual cada maestro tiene su turno para acceder al bus. La desventaja evidente de este esquema es que transacciones importantes pueden verse demoradas significativamente hasta que llegue su turno de ser atendida, produciendo retrasos en el funcionamiento y desempeño de la arquitectura. Aumentando en complejidad se encuentra el esquema de acceso múltiple con división de tiempo (TDMA, del inglés *Time Division Multiple Access*), en el cual se asigna una ventana temporal de distinta duración para cada maestro acorde a sus respectivas necesidades. Con esta práctica se evita el retraso de operaciones importantes por parte de otras de menor relevancia mientras se mantiene la equidad de acceso para los maestros de menor prioridad.

Una propuesta distinta, de complejidad mayor y buen desempeño, es la de prioridad dinámica cuyo principio es idéntico al de prioridad estática con la diferencia que las prioridades pueden ser modificadas en tiempo de ejecución según sea requerido. Esta práctica precisa lógica adicional para el análisis de tráfico de datos, por ende no solo supone una mayor complejidad de implementación sino además un mayor recurso de área y consumo.

Se presentan a continuación arquitecturas de bus, conocidas y utilizadas ampliamente en la industria, con sus respectivas interfaces y protocolos.

2.1.2. CoreConnect

Es una arquitectura de bus on-chip desarrollada por IBM, ideada para facilitar la interconexión de bloques de procesamiento y periféricos de distintas fuentes en un diseño VLSI, y enfocada en la reutilización de los mismos. Se organiza en tres jerarquías: PLB (del inglés *Processor Local Bus*, OPB (del inglés *On-Chip Peripheral Bus* y DCR (del inglés *Device Control Register* que se detallan en las siguientes sub-secciones. La Fig. 2.7 muestra un esquema de conexión CoreConnect con las tres jerarquías mencionadas.

La arquitectura soporta anchos de bus de 32, 64 y 128 bits con la posibilidad de extender a 256. Utiliza vías separadas para la escritura y lectura de datos y soporta múltiples maestros. Provee características para diseños de alto desempeño como: *pipelinening*, transacciones partidas y *burst*. Las especificaciones de la arquitectura CoreConnect se encuentran abiertamente disponibles y se requiere de licencias para su utilización, con la adquisición de las mismas se tiene acceso a bloques IP para los módulos de arbitraje de cada jerarquía y OPB-PLB *bridge*.

2.1.2.1. PLB

Es la jerarquía más alta de la especificación CoreConnect y se utiliza para la comunicación de los bloque de mayor desempeño. Entre tus principales características se destaca su capacidad de soportar hasta 16 maestros y múltiples esclavos dependiendo del compromiso de frecuencia del reloj del bus que se desea alcanzar y la implementación del IP de arbitraje PLB que se utilice. Posee temporizado sincrónico mediante una única fuente de reloj compartida por todo el sistema. Brinda variedad de esquemas de arbitraje con

cuatro niveles de prioridades para solicitudes de maestros. Soporte para transferencias *burst* de distintos anchos, byte, half-word, word y double-word y posee habilitación de bytes del bus para transferencias de datos no alineadas.

Suele interconectar dispositivos de alto ancho de banda como procesadores, interfaces de memoria externa y controladores DMA. Ofrece soluciones para diseños de alto desempeño y baja latencia proveyendo buses separados de datos (escritura y lectura), direcciones y transacciones partidas para cada maestro del sistema. Transferencias de lectura y escritura concurrentes permitiendo un máximo de dos transferencias por ciclo de reloj. Reducción de latencia de bus mediante *pipelinening* de direcciones, permitiendo el solapamiento de solicitudes de escritura durante escritura de datos y hasta tres solicitudes de lecturas durante una lectura.

La Fig. 2.6 ilustra la conexión de múltiples maestros y esclavos mediante el IP de arbitraje PLB. Cada maestro se conecta al IP mediante buses de dirección y datos (lectura/escritura) y señales de control de transferencia. Los esclavos del bus PLB se conectan al IP a través del bus compartido.

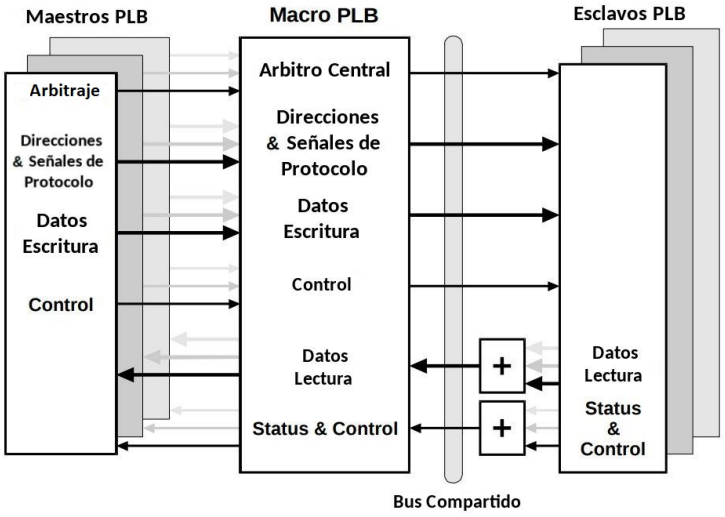


FIGURA 2.6: Esquema de conexión entre múltiples maestros y esclavos en la jerarquía PLB.

2.1.2.2. OPB

Es una arquitectura de bus secundaria ideada para aliviar los cuellos de botella en el desempeño del sistema, reduciendo la carga capacitiva del bus PLB. Los periféricos que

usualmente se conectan al OPB son puertos serie, paralelo, UARTs, GPIO, temporizadores y otros dispositivos de menor ancho de banda.

Provee un protocolo separado del PLB, sincrónico y con 32 bits de direcciones y datos. Dimensionamiento dinámico de bus brindando transferencias de byte, half-word y word. Protocolo secuencial para transferencias *burst* y soporte para varios maestros. Los maestros conectados al PLB pueden obtener acceso a los periféricos del bus OPB mediante un *bridge* IP. Éste actúa como un dispositivo esclavo en el bus PLB y maestro en OPB. Brinda soporte para transferencias de lectura y escritura de 32, 16 y 8 bits en el bus de datos del OPB, así como también transferencias *burst*. El *bridge* también realiza dimensionamiento dinámico del bus, permitiendo que dispositivos con diferente ancho de bits de datos se comuniquen eficientemente. Esto incluye la división de operaciones en sucesivas transferencias de menor ancho cuando el *bridge* debe transmitir datos a un esclavo de menor ancho de datos.

2.1.2.3. DCR

Es la arquitectura de menor jerarquía y se emplea para la lectura y escritura de registros de estado y configuración del sistema y, por ende, operaciones de menor desempeño. El protocolo brinda una transferencia ya sea de lectura o escritura cada dos ciclos de reloj, de manera sincrónica y se basa en una topología de anillo implementada comúnmente como un multiplexor distribuido. Permite un único maestro y su bus de direcciones consta de 10 bits, mientras que el de datos es de 32 bits.

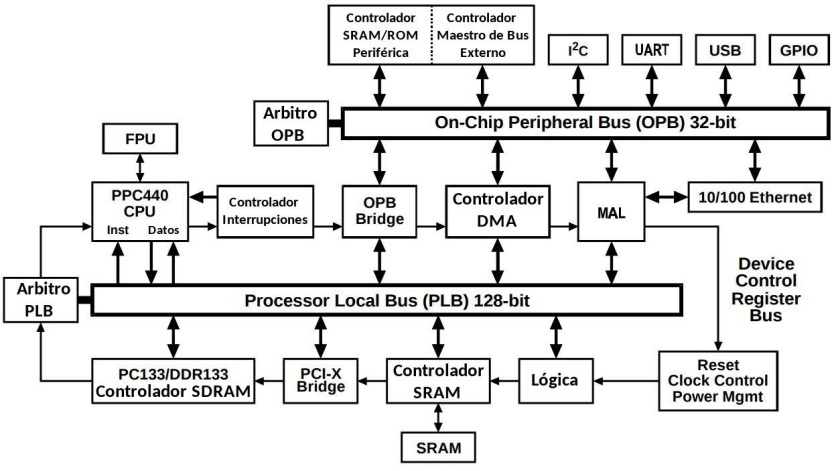


FIGURA 2.7: Esquema de conexión CoreConnect con las tres jerarquías: PLB, OPB y DCR.

2.1.3. WISHBONE

La arquitectura de bus para interconexión de SoC WISHBONE, creada por Silicore Corporation, ofrece una metodología flexible de diseño para utilizar con bloques IP. Se basa en la creación de una interfaz común para los bloques del sistema, mejorando la portabilidad y confiabilidad del mismo, sin comprometer las funciones específicas para las que los IP son empleados. Entre las principales características de la arquitectura se destaca que: no presenta jerarquías de bus, se organiza en el esquema de bloques maestroesclavo con posibilidad de múltiples maestros, soporta anchos de bus de 8, 16, 32 y 64 bits, permite cuatro variantes de interconexión entre bloques, brinda protocolo de *handshaking* para comunicación entre bloques que operan a distintas velocidades de transferencia, soporta transferencias de un ciclo de reloj, decodificación parcial de direcciones para bloques esclavos, metodología de arbitraje definible por el diseñador e identificación de transferencias mediante etiquetas configurables (*tags*) para datos, control de paridad y errores. WISHBONE soporta ambos tipos de organización de los datos de un operando, big y little endian. En el primero el byte más significativo del operando es transmitido en la parte alta de la dirección mientras que en el segundo caso se transmite por la parte baja.

Al tratarse de una arquitectura maestro esclavo, los bloques con interfaz maestro son los que inicializan las transferencias de datos con los bloques esclavos participantes de las mismas. En la Fig. 2.8 se muestra un esquema con el cual el maestro se comunica con el esclavo mediante una interfaz de conexión definida como INTERCON. Las Tablas 2.1, 2.2, 2.3 y 2.4 muestran y describen las señales de la interfaz para cada bloque del esquema ejemplificado. Todas las señales de los maestros y esclavos son entradas o salidas a los bloques pero no existen interfaces bi-direccionales o de tres estados. El protocolo de *handshaking*, además de permitir la comunicación entre bloques de distintas velocidades, brinda a los esclavos la posibilidad de aceptar, rechazar o solicitar reintentos de transferencias de datos al maestro mediante las señales ACK_O, ERR_O o RTY_O. Éstas últimas dos señales de la interfaz son opcionales.

Señal	Bloque	Descripción
CLK_O	SYSCON	Salida de reloj del sistema, coordina todas las actividades de la lógica interna de la interconexión WISHBONE.
RST_O	SYSCON	Fuerza el reinicio de todas las interfaces de la interconexión WISHBONE.

Tabla 2.1: Señales del bloque SYSCON de la interfaz WISHBONE.

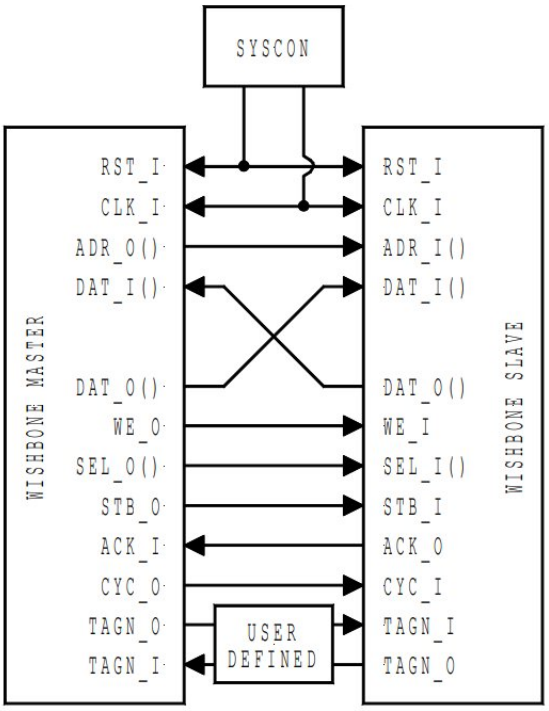


FIGURA 2.8: Esquema básico de conexión punto a punto con la interfaz WISHBONE.

La arquitectura WISHBONE permite cuatro tipos de interconexión entre maestros y esclavos. La conexión punto a punto conecta exclusivamente interfaces entre un solo maestro y un solo esclavo. La conexión como flujo de datos dota a todos los bloques de una interfaz maestro y esclavo, conectándolos entre si intercaladamente entre dichas interfaces. Los datos fluyen de un bloque a otro explotando ventajas de paralelismo en el procesamiento de los mismos, mejorando la velocidad de ejecución de tareas del sistema. También soporta conexión de bus compartido y bus cruzado (*crossbar-switch*), el bloque de arbitraje que determine cuándo un maestro en particular puede tomar control del bus debe ser definido por el diseñador del sistema y no está provisto como bloque IP por la arquitectura de bus WISHBONE.

Los tipos de transacciones básicas soportadas por la arquitectura son:

Escritura/lectura individual

Es la más elemental de las transacciones WISHBONE, que como su nombre sugiere, implica la transferencia de un dato individual entre un maestro y un esclavo del sistema.

Señal	Bloque	Descripción
CLK_I	Maestro Esclavo	Reloj del sistema provisto por el módulo SYSCON. Todas las señales WISHBONE son registradas con el flanco ascendente de esta señal.
DAT_I	Maestro Esclavo	Arreglo de entrada de datos binarios, el ancho está determinado por el diseño del sistema con un máximo posible de 64 bits.
DAT_O	Maestro Esclavo	Arreglo de salida de datos binarios, el ancho está determinado por el diseño del sistema con un máximo posible de 64 bits.
RST_I	Maestro Esclavo	Fuerza el reinicio de todas las interfaces de la interconexión WISHBONE. No es requisito que esta señal se utilice para reiniciar bloques o funciones internas de un IP del sistema.
TGD_I	Maestro Esclavo	Tipo de etiqueta para datos asociados al arreglo de entrada DAT_I. Contiene información de los datos transmitidos y es confirmada por la entrada STB_I.
TGD_O	Maestro Esclavo	Tipo de etiqueta para datos asociados al arreglo de entrada DAT_O. Contiene información de los datos transmitidos y es confirmada por la entrada STB_O.

Tabla 2.2: Señales de la interfaz WISHBONE compartidas por bloques maestros y esclavos.

Escritura/lectura en bloques

Puede ser pensada como una sucesión encadenada de transacciones individuales de escritura/lectura. El bus es ocupado por el maestro durante la totalidad de las transacciones simples hasta que se finaliza la transferencia del bloque.

Lectura-Modificación-Escritura

Es un tipo de transferencia usualmente utilizada en sistemas multi-tasking con múltiples procesadores. Permite la utilización de recursos compartidos del sistema mediante la utilización de semáforos. En este tipo de transferencia se leen y escriben datos en espacios de memoria en un solo ciclo de reloj, impidiendo el uso de un recurso común por parte de uno o más procesos. Se la conoce también como transferencia indivisible o dedicada, y se compone de dos fases, que pueden interpretarse como dos transferencias en bloques siendo la primera de lectura y la segunda de escritura.

Señal	Bloque	Descripción
ACK_I	Maestro	Entrada que indica la normal finalización de un ciclo de bus.
ADR_O	Maestro	Arreglo de salida de direcciones binarias de una transferencia.
CYC_O	Maestro	Salida que indica que un ciclo de bus válido se encuentra en curso. La señal se debe mantener activa por la duración total del ciclo.
STALL_I	Maestro	Entrada utilizada en modo pipeline, indica que el esclavo del ciclo actual no se encuentra apto para agregar otra transferencia a la cola de transacciones del pipeline.
ERR_I	Maestro	Entrada que indica la finalización anormal del ciclo de bus.
LOCK_O	Maestro	Salida que indica que el ciclo actual de bus no puede ser interrumpido. No se debe conceder el acceso al bus a ningún otro maestro hasta no finalizarse el ciclo actual completo.
RTY_I	Maestro	Entrada que indica que la interfaz no está disponible para realizar una transferencia, el ciclo de bus debe mantenerse y reintentar el acceso al mismo.
SEL_O	Maestro	El arreglo de salida de selección señala en qué bits del arreglo de datos de entrada o salida DAT_I/DAT_O se encuentran los datos válidos de la transferencia del ciclo de lectura o escritura respectivamente.
STB_O	Maestro	Salida que indica un ciclo de transferencia de datos válido. También confirma otras señales asociadas al ciclo actual como SEL_O, TGD_O, TGA_O.
TGA_O	Maestro	Tipo de etiqueta que contiene información asociada a la dirección indicada por ADR_O.
TGC_O	Maestro	Tipo de etiqueta que contiene información asociada a los ciclos de bus, es confirmada por la señal CYC_O. Puede emplearse para discriminar entre los tipos de ciclos de bus WISHBONE.
WE_O	Maestro	Salida de habilitación de escritura, indica si el ciclo actual de bus es de lectura o escritura.

Tabla 2.3: Señales de la interfaz WISHBONE del bloque maestro.

Señal	Bloque	Descripción
ACK_O	Esclavo	Salida que indica la normal finalización de un ciclo de bus.
ADR_I	Esclavo	Arreglo de entrada de direcciones binarias de una transferencia.
CYC_I	Esclavo	Entrada que indica que un ciclo de bus válido se encuentra en curso. La señal se debe mantener activa por la duración total del ciclo.
STALL_O	Esclavo	Salida utilizada en modo pipeline, que indica que el esclavo del ciclo actual no se encuentra apto para agregar otra transferencia a la cola de transacciones del pipeline.
ERR_O	Esclavo	Salida que indica la finalización anormal del ciclo de bus.
LOCK_I	Esclavo	Entrada que indica que el ciclo actual de bus no puede ser interrumpido. Un esclavo que recibe esta señal en estado activo no debe ser accedido por ningún otro maestro hasta que LOCK_I o CYC_I inviertan su estado.
RTY_O	Esclavo	Salida que indica que la interfaz no está disponible para realizar una transferencia, el ciclo de bus debe mantenerse y reintentar el acceso al mismo.
SEL_I	Esclavo	El arreglo de entrada de selección señala en qué bits del arreglo de datos de entrada o salida DAT_I/DAT_O se encuentran los datos válidos de la transferencia del ciclo de lectura o escritura respectivamente.
STB_I	Esclavo	Entrada que indica que el esclavo está seleccionado en el ciclo de bus actual. El mismo puede responder a otras señales WISHBONE solo si STB_I está activa (excepto por RST_I). Ante cada transición de STB_I a estado activo, el esclavo debe responder con las señales ACK_O, ERR_O o RTY_O.
TGA_I	Esclavo	Tipo de etiqueta que contiene información asociada a la dirección indicada por ADR_O.
TGC_I	Esclavo	Tip de etiqueta que contiene información asociada a los ciclos de bus, es confirmada por la señal CYC_O. Puede emplearse para discriminar entre los tipos de ciclos de bus WISHBONE.
WE_I	Esclavo	Entrada de habilitación de escritura, indica si el ciclo actual de bus es de lectura o escritura.

Tabla 2.4: Señales de la interfaz WISHBONE del bloque esclavo.

2.1.4. AMBA

La especificación de arquitectura de bus AMBA (sigla del inglés *Advanced Microcontroller Bus Architecture*), diseñada por ARM, es un estándar abierto que define un esquema de interconexión para los bloques funcionales que componen un *System-On-Chip* en circuitos integrados.

El objetivo de la arquitectura es especificar un protocolo de bus de comunicación de alto desempeño y flexibilidad, independiente de la tecnología y que pudiere optimizar la utilización de recursos en su implementación, promoviendo la reutilización de IPs. En este aspecto, brindar un estándar común de comunicación es un requisito esencial para la integración de bloques con diferentes características es en SoCs.

En la actualidad AMBA es uno de los estándares de comunicación más empleados en la industria de los circuitos integrados, otorgando conectividad a IPs como CPUs, GPUs y controladores de memoria, cumpliendo su meta de fomentar el diseño modular y compatibilizando componentes de distintos diseños y fabricantes. Entre las especificaciones de la arquitectura se definen varias interfaces que brindan soporte para sistemas de diversa complejidad, requerimientos de velocidad y potencia. Éstas interfaces han sido ampliadas y actualizadas desde la primera especificación donde se definieron el ASB (del inglés *Advanced System Bus*) y el APB (del inglés *Advanced Peripheral Bus*).

El protocolo ASB es la primera generación de AMBA que brinda conexión eficiente entre procesadores y memorias on-chip, con la posibilidad de integrar uno o más maestros, DSPs y DMAs. Por otro lado, el protocolo APB brinda soluciones para la conexión de periféricos de bajo desempeño, promoviendo la reducción de complejidad en las interfaces y permitiendo funcionalidades periféricas, suele emplearse en conjunto con ASB mediante la implementación de un *bridge* entre ambos que funciona como un esclavo del sistema ASB. La siguiente generación de AMBA introduce el protocolo AHB (del inglés *Advanced High-performance Bus*) con el propósito de brindar soluciones a diseños de mayor desempeño que los soportados por ASB, implementando características que favorecen la operación a mayor frecuencia y con ancho de datos mayores. Entre ellas se destacan transferencias *burst*, división de transacciones, operación en *pipeline* y buses de ancho hasta 128 bits. En la tercera generación se introducen dos nuevas interfaces, AXI3 (del inglés *Advanced eXtensible Interface*) y ATB (del inglés *Advanced Trace Bus*), y se amplían las especificaciones para AHB y APB. La interfaz AXI3 soporta operaciones a

alta velocidad, consideraciones en el consumo y transacciones de lectura y de escritura simultáneas, permitiendo tráfico de datos con alto desempeño. Comúnmente utilizado en diseños con múltiples procesadores, controladores gráficos y periféricos sofisticados donde el protocolo AHB no cumple la demanda de los requerimientos. El protocolo de bus ATB provee un sistema de seguimiento para propósitos de depuración. La ampliación AHB3 incluye interconexiones para periféricos donde no se requiere la interfaz AXI y se introduce el protocolo AHB-Lite para sistemas de alto desempeño y ancho de bus con un único maestro. APB3 incorpora transferencias de bajo ancho de banda para el acceso de registros de control. La cuarta generación de AMBA introduce cuatro nuevas interfaces, ACE (del inglés *AXI Coherency Extensions*), ACE-Lite, AXI4-Lite, AXI4-Stream y también se extiende AXI. La interfaz ACE incorpora interconexiones de alta velocidad que se emplean típicamente en aplicaciones móviles, ACE-Lite es un subconjunto de señales ACE adaptado para requerimientos menos demandantes. AXI 4 incorpora mejoras para el desempeño de sistemas con múltiples maestros y la versión adaptada para menores requerimientos AXI4-Lite. El propósito de AXI4-Stream es brindar transferencias de datos unidireccionales desde el maestro hacia el esclavo con la reducción de señales, permitiendo en un mismo conjunto de conexiones establecer transferencias de datos simples y múltiples de distintos anchos. La quinta y última generación introduce la interfaz CHI (del inglés Coherent Hub Interface) que permite comunicaciones a mayores velocidades y con nuevas características de diseño para reducir congestión. Su arquitectura se ha desarrollado con el propósito de mantener el desempeño del bus pese al incremento de componentes del sistema y el tráfico de datos producto de los mismos. Su aplicación es preferente en sistemas que requieren coherencia como dispositivos móviles, *networking* y *data centers*.

La Fig. 2.9 muestra un diagrama de bloques ejemplificando un sistema que utiliza AHB o ASB como arquitectura de bus principal. Los bloques del procesador y DMA actúan como maestros del sistema en dicho bus con la memoria on chip, la interfaz de memoria externa y el *bridge* como bloques esclavos. Este último establece un medio de conexión a un esquema APB para la interconexión de periféricos y otros bloques de menor desempeño, aislándolos de la arquitectura de bus principal. Las Tablas 2.6 y 2.5 resumen las interfaces de los maestros y esclavos de los protocolos AHB y APB respectivamente.

Señal	Bloque	Descripción
PCLK	Maestro Esclavo	Reloj del sistema para sincronizar las transferencias APB con flancos positivos.
PRESETn	Maestro Esclavo	Fuerza el reinicio de los bloques del sistema, activa en nivel bajo.
PADDR	Maestro	Indica que un esclavo está seleccionado y que hay una transferencia. Hay una señal PSEL para cada esclavo.
PSELx	Maestro	Indica que un esclavo está seleccionado y que hay una transferencia. Hay una señal PSEL para cada esclavo.
PENABLE	Maestro	Señal que indica el segundo ciclo y ciclos subsecuentes de una transferencia.
PWRITE	Maestro	Indica una transferencia de escritura cuando está en nivel alto y una de lectura cuando es bajo.
PWDATA	Maestro	Dato de escritura del bus cuando PWRITE está en nivel alto. Puede ser de hasta 32 bits.
PREADY	Esclavo	El esclavo utiliza esta señal para indicar que necesita extender una transferencia APB.
PRDATA	Esclavo	Dato de lectura del bus cuando PWRITE está en nivel bajo. Puede ser de hasta 32 bits.

Tabla 2.5: Señales de la interfaz del protocolo AMBA APB.

Señal	Bloque	Descripción
HCLK	Maestro Esclavo	Relojs del sistema para transferencias del bus por flancos positivos de la señal.
HRESETn	Maestro Esclavo	Fuerza el reinicio de los bloques en nivel bajo.
HADDR	Maestro	Bus de direcciones de hasta 1024 bits.
HWDATA	Maestro	Bus de datos del maestro al esclavo.
HWRITE	Maestro	Indica la operación de la transferencia. En estado alto escritura y en bajo lectura.
HRDATA	Esclavo	En operaciones de lectura, es el dato es leído desde el esclavo seleccionado.
HREADYOUT	Esclavo	En estado alto indica el final de la transferencia. El esclavo puede mantener baja esta señal para extender una transferencia.
HRESP	Esclavo	Indica si se producen errores en la transferencia.
HTRANS	Maestro	Indica tipo de transferencia a realizar.
HBURST	Maestro	Indica si es transferencia simple o burst.
HSEL	Maestro	Señal de selección de esclavos.
HMASTLOCK	Maestro	Indica que la transferencia debe procesarse sin interrupción ni producirse otra transacción.
HPROT	Maestro	Brinda información adicional del tipo de transferencia.
HSIZE	Maestro	Indica el ancho de datos de la transferencia.
HREADY	Multiplexor	Señal que indica a maestros y esclavos del sistema que las transferencias finalizaron.

Tabla 2.6: Señales de la interfaz del protocolo AMBA AHB.

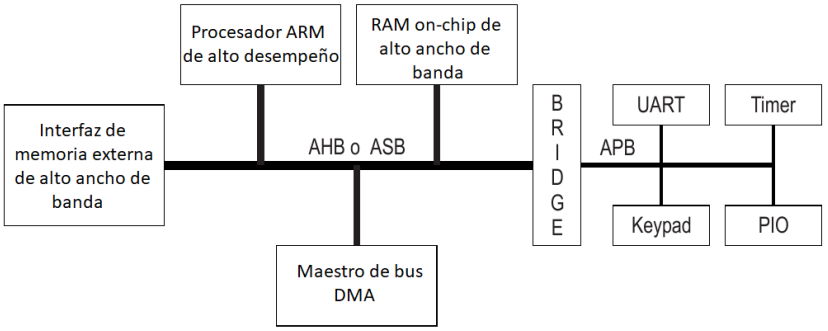


FIGURA 2.9: Esquema de arquitectura de comunicación AMBA AHB y APB conectadas mediante un bridge.

2.1.5. AMBA AHB-Lite

El protocolo AMBA AHB-Lite es una interfaz de bus y provee soluciones para diseños de alto desempeño. El sistema básico AHB-Lite mostrado en la Fig. 2.10 se compone de un dispositivo maestro, tres esclavos y su interconexión es llevada a cabo mediante de un decodificador y un multiplexor maestro-esclavo. Los dispositivos esclavos en AHB-Lite suelen ser memorias, interfaces de memoria externa y periféricos. El dispositivo maestro es el que se encarga de suministrar direcciones y señales de control con las que se inician las operaciones de lectura y escritura del sistema. Los dispositivos esclavos responden a estas transferencias y es trabajo del decodificador seleccionar al esclavo adecuado para la tarea exigida por el maestro. Los esclavos cuentan con señales para el maestro que le permiten indicar una falla en una transferencia, su correcta ejecución, o el requerimiento de más ciclos de reloj para completarla. Además de la tarea mencionada, el decodificador provee señales de control para el multiplexor que es el encargado de multiplexar los buses de lectura y las señales de respuesta de los esclavos al maestro.

Con los bloques mencionados el sistema AHB-Lite es funcional y opera transferencias en dos fases como se explica a continuación. El maestro inicialmente coloca en el bus de escritura la dirección y configura señales de control que brindan información de la naturaleza de la transferencia como: lectura/escritura, ancho de palabra, transferencia simple o en ráfagas. Durante esta primera fase, denominada de dirección, todos los esclavos deben ser capaces de tomar los datos suministrados. La segunda fase es la de datos y en el transcurso de su duración los esclavos deben, según el tipo de operación, suministrar el dato requerido o almacenarlo. En caso de no poder cumplirlo en el plazo de un ciclo de reloj, pueden generar demoras mediante sus señales de respuesta al maestro.

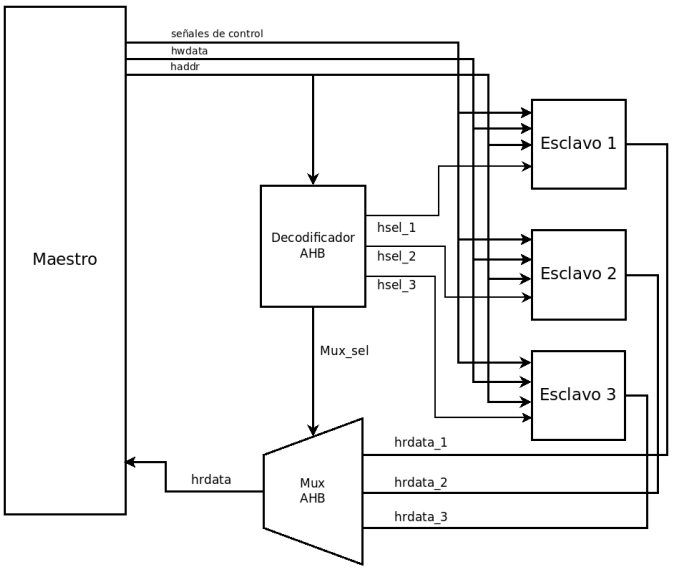


FIGURA 2.10: Sistema de bus AHB-Lite con un maestro y tres esclavos.

Se reseña a continuación una breve descripción de todas las señales involucradas en el protocolo AHB-Lite, agrupadas según el dispositivo que las controla. Las señales denominadas globales son aquellas que todos los dispositivos tienen como entrada y por ende toman parte de acciones del sistema completo. Entre ellas se encuentra la señal **hclk** que es la señal de reloj que temporiza todas las transferencias, y se sincronizan con su flanco ascendente. La segunda es la señal $\mathbf{hreset}_n$ que es la de reinicio del sistema, y está activa cuando su estado lógico es bajo.

Entre las señales provistas por el maestro se encuentra **haddr[31:0]** que se destina al decodificador y los esclavos como el bus de direcciones. La señal **hburst[2:0]** indica a los esclavos si se trata de una transferencia simple o en ráfagas, soportando ráfagas de ancho de palabra de 4, 8 y 16 bits. Para sistemas de maestros múltiples se utiliza la señal **hmastlock** indicando a los esclavos que la transferencia actual es dedicada y no puede interrumpirse para continuarse luego. La señal **hprot[3:0]** brinda información a los esclavos acerca de la protección de la transferencia, indicando si el acceso al bus es en modo privilegiado o modo usuario. Además se emplea por los maestros que utilizan memoria cache para indicar si la información es cacheable o no. La señal **hsize[2:0]** indica a los esclavos el tamaño de la transferencia, es decir, byte, palabra o media palabra. Si una transferencia determinada es del tipo ráfaga, es necesario que el maestro informe a los esclavos de la continuidad de las mismas o su finalización, para ello se emplea la señal **htrans[1:0]** cuyos cuatro estados (*idle*, *busy*, *nonsequential*, *secuential*) proveen la información necesaria. Como indicador de que una operación es de escritura el maestro

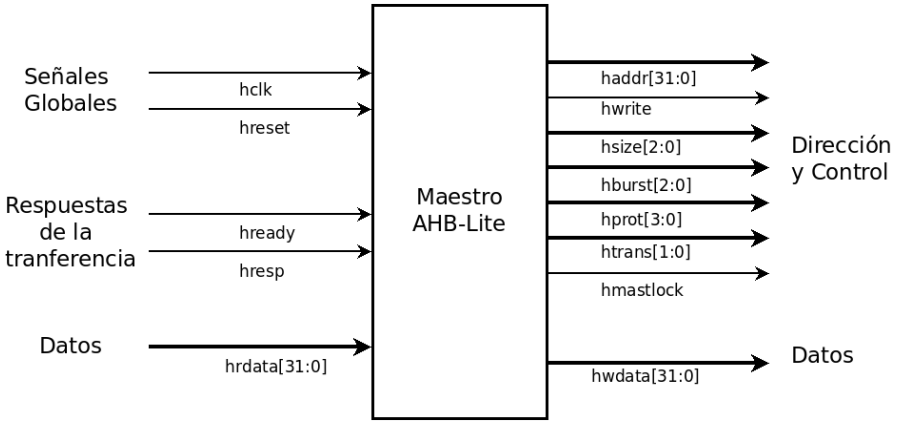


FIGURA 2.11: Esquema en bloque del maestro AHB-Lite.

activa en estado lógico alto la señal **hwrite**. En las operaciones de escritura el maestro envía a través del bus de escritura **hwdata[31:0]** los datos al esclavo. En la Fig. 2.11 se esquematiza el bloque del maestro con todas sus señales.

Las tres señales del esclavo son **hrdata[31:0]**, **hreadyout** y **hresp**, todas destinadas al multiplexor de esclavos. La primera es el bus de transmisión de datos de lectura, que el multiplexor comunica al maestro a través del esclavo requerido en la transferencia. La segunda señal, activa en estado lógico alto, indica al maestro que la transferencia se ha finalizado. Si es mantenida en estado bajo, prolonga la transferencia. La tercera señal es utilizada por el esclavo para indicar si se produjo alguna falla durante la transferencia. El esquema del esclavo se muestra en la Fig. 2.12.

Las salidas del decodificador corresponden a la decodificación combinatoria de las direcciones que el maestro le envía, resultando así una señal $\mathbf{hsel}_x$ para cada esclavo que se activa en estado lógico alto para indicar qué esclavo debe atender la transferencia actual. El esclavo seleccionado debe además verificar el estado de la señal **hready** antes de responder por la transferencia en curso.

Finalmente son tres las señales del multiplexor **hrdata[31:0]**, **hready** y **hresp**, todas ellas destinadas al maestro y la segunda también destinada a los esclavos. La primera señal es el bus de datos de lectura, donde se envía al maestro los datos del esclavo habilitado por el decodificador y seleccionado por el multiplexor. La segunda señal indica a todos los dispositivos maestros y esclavos si la última transferencia fue finalizada. La última señal indica si la operación, aunque finalizada, produjo alguna falla.

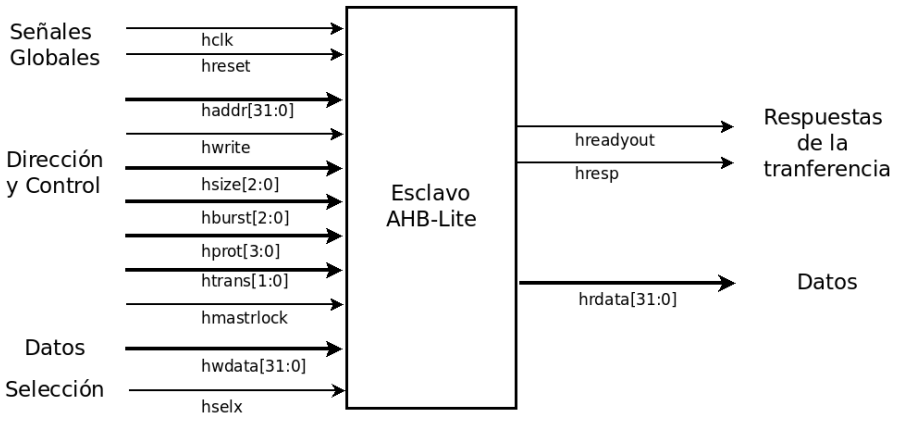


FIGURA 2.12: Esquema en bloque del esclavo AHB-Lite.

2.1.5.1. Descripción de Transferencias

Como se menciona en párrafos anteriores el protocolo soporta transferencias de datos desde 8 a 1024 bits según se configure la señal **hsize[2:0]** y acorde con la Tabla 2.7, siempre limitado en la práctica por el ancho real del bus de datos de los dispositivos involucrados en la comunicación. Para una aplicación de 32 bits se tienen transferencias de un byte $(000)_2$, media palabra $(001)_2$ y una palabra $(010)_2$. La señal **hsize** tiene el mismo temporizado que el bus de direcciones pero sus líneas deben permanecer invariables durante el transcurso de una transferencia en ráfagas.

El caso óptimo de una transferencia utilizando el protocolo es aquella en la que no se producen tiempos de espera, consistiendo la transferencia en un ciclo de dirección y un ciclo de datos. En primer lugar el maestro coloca la dirección en el bus de direcciones y configura las señales de control luego de un flanco ascendente de **hclk**. En el siguiente flanco ascendente, el esclavo registra toda la información y puede comenzar a responder con los datos pertinentes y la respuesta apropiada con **hready** para que el maestro registre los mismos en el siguiente flanco ascendente de **hclk**. En las figuras 2.13 y 2.14

hsize[2]	**hsize[1]**	**hsize[0]**	Ancho en bits
0	0	0	8
0	0	1	16
0	1	0	32
0	1	1	64
1	0	0	128
1	0	1	256
1	1	0	512
1	1	1	1024

TABLA 2.7: Valores de la señal de control **hsize** para cada tamaño de transferencia.

se aprecian ejemplos del funcionamiento descrito y se evidencia que cualquier fase de direcciones ocurre durante la fase de datos de la transferencia anterior. Este solapamiento de operaciones es el que permite alto desempeño de bus respetando el tiempo de respuesta requerido por cada uno de los esclavos del sistema. Durante una operación de escritura el maestro mantiene los datos estables a lo largo de los ciclos del tiempo de espera. Para operaciones de lectura no es necesario que los esclavos entreguen datos válidos sino hasta el momento que la transferencia esté por finalizar. Además, un esclavo puede insertar tiempos de espera en cualquier transferencia en la que requiera de tiempo adicional en presentar los datos. Debido al solapamiento de operaciones mencionado, generar esperas tiene el efecto de extender la duración de la fase de direcciones de la siguiente transferencia (caso de la dirección B en las figuras 2.15 y 2.16).

Las trasferencias tipo ráfagas consisten en transferencias continuas que pueden ser de 4 , 8 o 16 fases de datos, también pueden ser de longitud indefinida o durar sólo una fase de datos (ráfagas simples). Esta información la brinda el maestro en la señal **hburst[2:0]**.

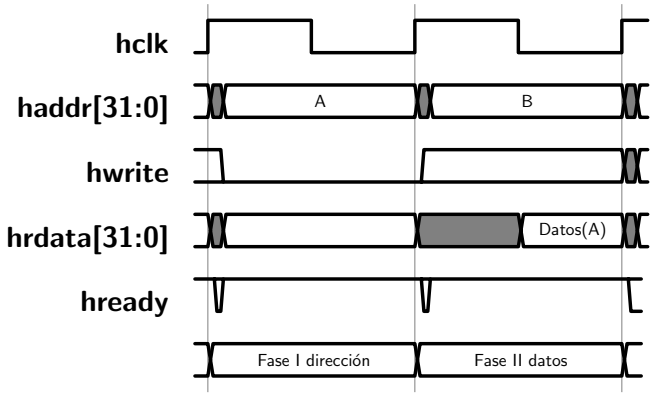


FIGURA 2.13: Diagrama temporal de lectura sin tiempos de espera en protocolo AMBA AHB-Lite.

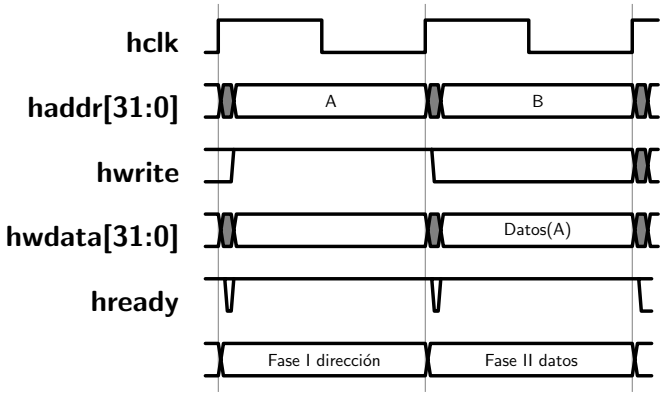


FIGURA 2.14: Diagrama temporal de escritura sin tiempos de espera en protocolo AMBA AHB-Lite.

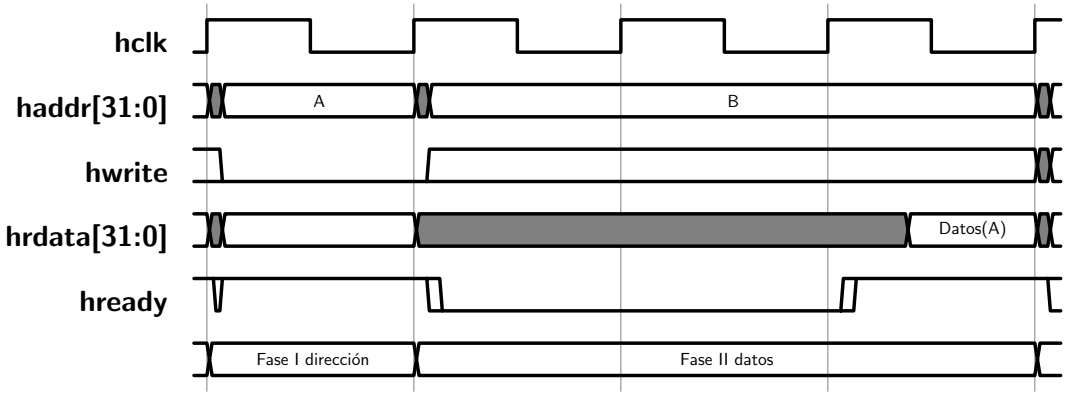


FIGURA 2.15: Diagrama temporal de lectura con tiempos de espera en protocolo AMBA AHB-Lite.

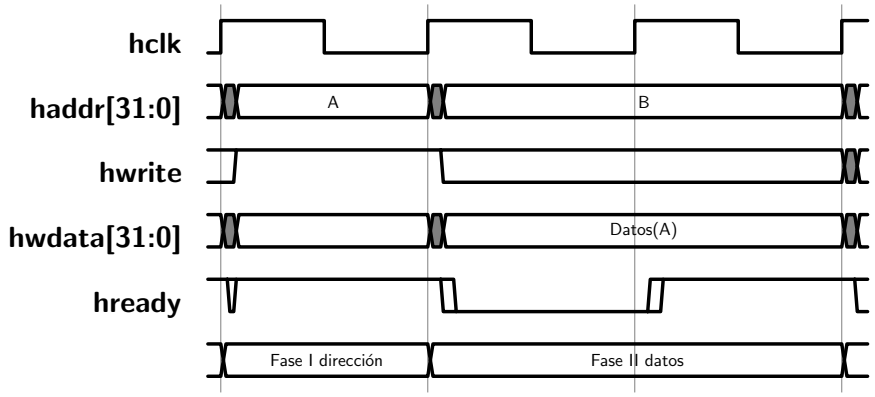


FIGURA 2.16: Diagrama temporal de escritura con tiempos de espera en protocolo AMBA AHB-Lite.

Existen dos tipos, las incrementales donde el acceso de cada ráfaga se realiza a direcciones de memoria continuas de forma incremental y las envueltas donde se fija una dirección de memoria inicial y final mediante las señales **hburst[2:0]** y **hsize[2:0]**. Cada ráfaga se realiza a direcciones de memoria continuas de forma incremental hasta alcanzada una dirección límite para seguir por la dirección inicial. En la Fig. 2.17 se puede observar una ráfaga incremental de 4 fases de datos donde el esclavo solicita un ciclo de espera mediante la señal **hready**.

Existen cuatro tipos de transferencias soportadas por el protocolo y se indican con la señal de control **htrans[1:0]** emitida por el maestro. Para indicar que no se requiere ninguna transferencia el maestro usa el estado *idle* $(00)_2$, generalmente usado al finalizar una transferencia dedicada (señalizada a través de **hmastlock**). Cuando la transferencia llevada a cabo es en ráfagas, el maestro puede necesitar insertar tiempos de espera entre una ráfaga y la siguiente. Para ello existe la transferencia *busy* $(01)_2$, y la dirección y las señales de control deben reflejar la siguiente ráfaga. Para comenzar una transferencia

en ráfagas o realizar una transferencia simple se utiliza el tipo *nonseq* $(10)_2$, ya que la transferencia simple es un caso especial de ráfaga. Para las ráfagas subsecuentes a la primera se utiliza el tipo de transferencia *seq* $(11)_2$ y dado que la información de control es la misma que para la primera ráfaga, se calcula la dirección de las siguientes sumando a la dirección inicial el tamaño de la transferencia (**hsize[2:0]**). La utilización de algunos de los estados descritos se ejemplifican en la Fig. 2.17.

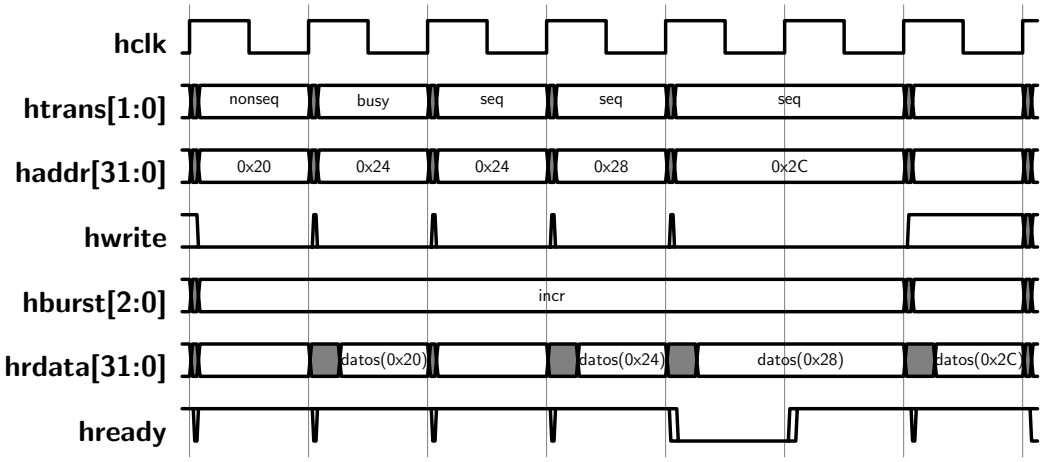


FIGURA 2.17: Diagrama temporal de lectura en ráfagas incrementales.

2.2. Arquitectura MIPS

2.2.1. Contexto histórico

El nombre MIPS es un acrónimo resultante de *Microprocessor without Interlocking Pipeline Stages*, y su diseño fue parte de un programa de investigación de VLSI en la Universidad de Stanford a comienzos de los años 80. El desarrollo del procesador se inició en una clase de estudiantes de grado de John Hennessy, quien luego impulsaría su diseño como uno de los primeros procesadores RISC junto con los que IBM y la Universidad de Berkeley realizaban durante el mismo período. Basado en dicho desarrollo se fundó en 1984 MIPS Computer Systems Inc., de la cual resultó el primer chip MIPS. Posteriormente, en 1992, la compañía fue comprada por Silicon Graphics para más tarde separarse bajo el nombre de MIPS Technologies en 1998. En la actualidad la arquitectura pertenece a la empresa Imagination Technologies. La arquitectura MIPS de 32 bits es un estándar de la industria de alta eficiencia que se encuentra actualmente en variedad de productos electrónicos, desde microcontroladores (en consolas como Playstation) hasta equipamiento de redes (Cisco). La arquitectura está provista de un robusto conjunto de instrucciones, escalabilidad de 32 a 64 bits y está optimizada para dar soporte a ejecución de lenguajes de alto nivel. El conjunto de instrucciones emplea el modelo de transferencia de datos *load/store* y sus instrucciones son de ancho fijo. Las operaciones aritmético-lógicas utilizan un formato de tres operandos permitiendo a los compiladores optimizar la formulación de operaciones complejas. Dispone de 32 registros de propósito general, lo cual facilita a los compiladores generar código de mejor desempeño brindando acceso rápido a datos que se utilizan con frecuencia. Desde el surgimiento de la arquitectura original MIPS I, la misma ha atravesado muchos cambios hasta MIPS V y las actuales MIPS32, MIPS64 y microMIPS. El diagrama de la Fig. 2.18 describe cronológicamente los lanzamientos de cada reforma de la arquitectura, señalando el quiebre en que se extiende la misma para la manipulación de datos en 64 bits con MIPS III. Cada escalafón evolutivo fue diseñado de manera que sea compatible hacia atrás con el resto de de los diseños. En el lanzamiento de MIPS IV y MIPS V se enfatizó el perfeccionamiento de las operaciones de punto flotante, aumentar la eficiencia de código y la transferencia de datos. Debido a la compatibilidad hacia atrás establecida, los cambios introducidos en las dos últimas mejoras no se reflejan en la versión de 32 bits de la arquitectura (MIPS I y MIPS II).

Entre los cambios sustanciales desde MIPS I a MIPS V se destaca la eliminación de la restricción en las instrucciones de carga y almacenamiento que requerían que tanto el registro base como la dirección de memoria indicados estuvieran alineados. A partir de MIPS32 la restricción se redujo a que solo la dirección, y no el registro base, lo fuera. En la arquitectura temprana se contaba con dos instrucciones para la transferencia de datos a los registros *hi* y *lo* para la futura operación de multiplicación o división. Ya en MIPS V se había eliminado dicha implementación y para MIPS32 los registros *hi* y *lo* estaban integrados en hardware para todos los usos y variantes de multiplicación y división. Los lanzamientos posteriores MIPS32 y MIPS64 surgieron como necesidad de proveer mayor desempeño a costos de complejidad en términos del conjunto de instrucciones. MIPS32 deriva de MIPS II agregando instrucciones seleccionadas de MIPS III, MIPS IV y MIPS

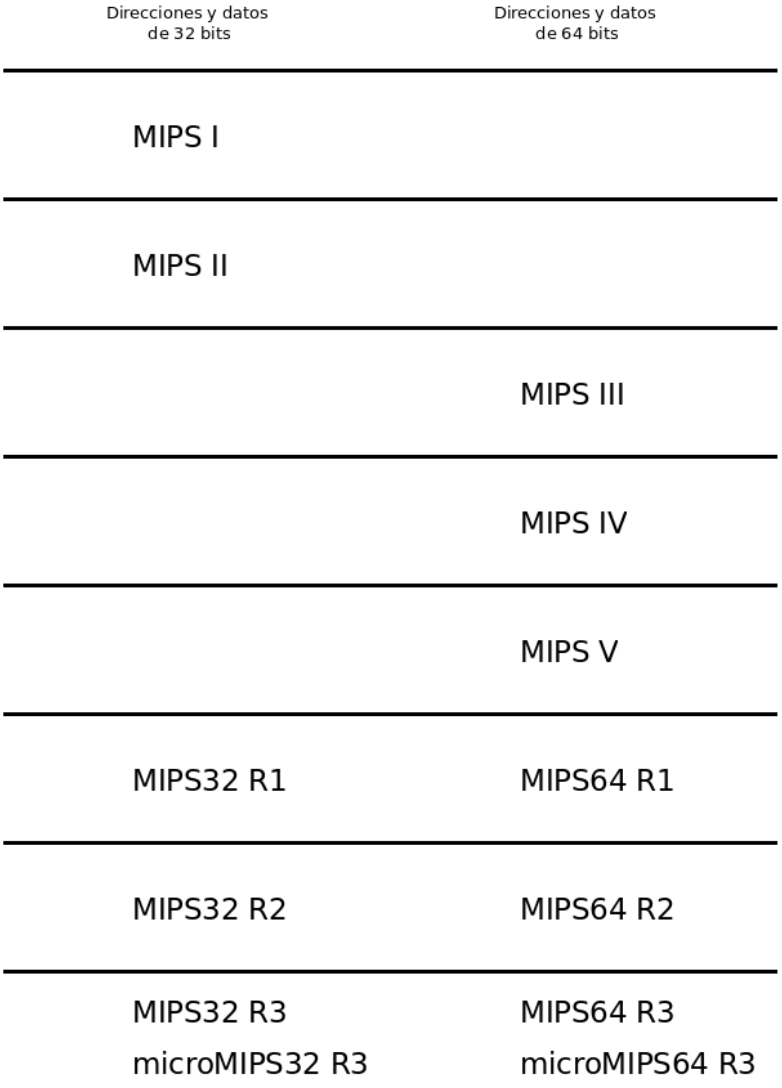


Figura 2.18: Evolución de la arquitectura MIPS.

V en pos de incorporar las mejoras en eficiencia de código y manejo de datos lograda en mencionados lanzamientos. MIPS64 deriva de MIPS V y es compatible hacia atrás con la funcionalidades de MIPS32, ambas incorporan el entorno privilegiado de instrucciones a la arquitectura con el objetivo de cubrir funciones necesarias del sistema operativo. Entre otras mejorías, en el segundo lanzamiento de la arquitectura se incorporaron al conjunto de instrucciones que permiten operar bits específicos de campos, desplazamientos y rotaciones en registros. También se agregan instrucciones para el manejo explícito de riesgos de flujo, se modifican los registros del Coprocesador 0 para identificación de niveles de cache y utilización de bits para vigilancia de estados de registros. Además, se proveen medios de soporte para coprocesadores de 64 bits compatibles con CPU de 32 bits. Las arquitecturas microMIPS32 y microMIPS64 aportan las mismas funcionalidades que MIPS32 y MIPS64 con el beneficio de aportar magnitudes de código menores. Son consideradas superconjuntos de MIPS32/64 ya que poseen el mismo conjunto de instrucciones de 32/64 bits y agregan instrucciones de 16 bits, apuntadas a aplicaciones donde la penalidad dominante es el espacio en memoria. La Fig. 2.19 muestra una representación conceptual de las arquitecturas en donde se observan definidas dos ramas, MIPS32/64 y microMIPS32/64 con cierto nivel de compatibilidad. Es posible implementar combinaciones de ambas ramas de la familia de arquitecturas y en tales casos deben definirse métodos para alternar entre ambos conjuntos de instrucciones. Desde la perspectiva de código, todos los mnemónicos de instrucciones y directivas de MIPS32/64 son completamente compatibles e interpretables por microMIPS32/64.

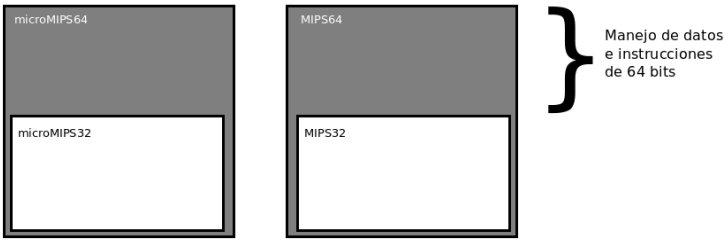


FIGURA 2.19: Diagrama de compatibilidad entre los lanzamientos de la arquitectura MIPS.

2.2.2. Diseño de la arquitectura a implementar

Se evalúan tres posibles enfoques para el diseño de la arquitectura del procesador considerando sus ventajas y desventajas. El primero de ellos es conocido en inglés como *Single-Cycle* (ciclo individual) y dicha denominación deriva de que la ejecución de las

instrucciones es llevada a cabo en un solo ciclo de reloj. Este enfoque ofrece un diseño cuya implementación es simple pero que resulta ineficiente dado que el período del reloj deberá ser como mínimo el requerido para realizar la instrucción más lenta. En el conjunto de instrucciones MIPS expuesto, dichas instrucciones son las de almacenamiento y lectura de memoria. Como consecuencia cualquier otra instrucción será ejecutada en un tiempo mayor al necesario, impactando en el desempeño de esta implementación. Otra desventaja radica en el hecho que las unidades funcionales del procesador pueden ser utilizadas una vez por cada ciclo, lo que requiere en algunos casos duplicar dichas unidades aumentando el requerimiento de hardware. El segundo enfoque evaluado mitiga las desventajas del *Single-Cycle* dividiendo la ejecución de las instrucciones en etapas que demandan un ciclo de reloj para completarse. La división es realizada de manera que cada etapa tenga como máximo una operación aritmético-lógica, un acceso a registro o un acceso a memoria. Como resultado se tiene un conjunto de instrucciones que requieren distinto número de ciclos de reloj para su ejecución y por ello se denomina a este enfoque *Multi-Cycle* (múltiples ciclos). Los beneficios de la división en etapas se ven reflejados en la posibilidad de reutilizar unidades funcionales durante la ejecución de una misma instrucción, siempre y cuando las mismas no sean requeridas para la misma etapa. Esto reduce el requerimiento de hardware de la implementación respecto del *Single-Cycle*. Por otro lado el desempeño no se ve penalizado cuando se ejecutan instrucciones cortas (de menor cantidad de etapas) permitiendo mayor frecuencia de operación. La complejidad de implementación del diseño *Multi-Cycle* aumenta debido a que requiere más lógica de control y caminos de datos. Además, como consecuencia de la división de instrucciones en distinto número de etapas, la incorporación del *pipeline* para este diseño resulta impráctica. Finalmente el tercer enfoque es el diseño que implementa la técnica del *pipeline* permitiendo finalizar la ejecución de una instrucción cada ciclo de reloj tal como se explica en el Apéndice A. Si bien posibilita aumentar el desempeño del procesador, la implementación del *pipeline* demanda la adición de hardware ya que es necesario replicar unidades funcionales por cada etapa de *pipeline* y agregar lógica de control para todo el proceso. Con las características expuestas de cada enfoque se opta por implementar el diseño *Multi-Cycle* para obtener un procesador cuyo diseño sea un compromiso entre alto desempeño y que requiera la menor cantidad de recursos de hardware posible.

Capítulo 3

Diseño de un sistema AMBA AHB-Lite

3.1. Diseño

En el trabajo presentado se desarrolla un sistema que utiliza el protocolo de comunicación AMBA AHB-Lite. Se utiliza como maestro del mismo un microprocesador basado en la arquitectura MIPS32, lo que implica que su conjunto de instrucciones se basa en las que se muestran en dicha arquitectura en la Sección 2.2 y el Apéndice B, implementando solo un subconjunto de las mismas e incorporando la lógica de control y *datapath* necesaria para cumplir las especificaciones del protocolo de bus. Para llevar a cabo la implementación del resto del sistema AHB-Lite se divide la memoria en dos bloques de 512 bytes cada uno (conformando un total de memoria de 1 GByte) volviendo necesaria la incorporación de un decodificador y un multiplexor al sistema para poder direccionar los datos transmitidos desde los distintos esclavos. El esquema conceptual del sistema descrito es como muestra la Fig. 3.1.

Todos los elementos del sistema son diseñados mediante el lenguaje de descripción de hardware Verilog, comenzando por la descripción del comportamiento de cada bloque que conforma el *datapath* y continuando con la conexión de los mismos para integrarlo. Por otro lado se realiza la descripción comportamental del control del procesador, el cual constituye una máquina de estados que manipula las señales de control acorde a la

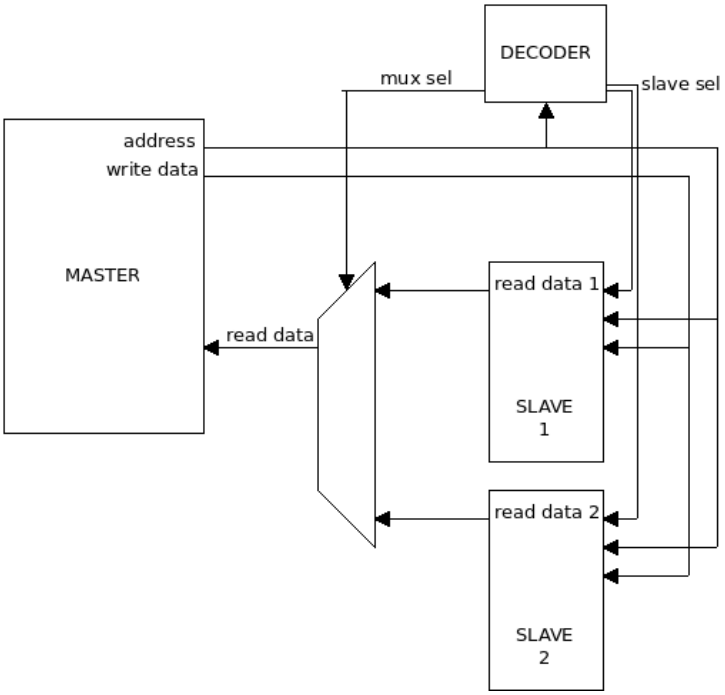


FIGURA 3.1: Esquema en bloques generalizado del sistema propuesto.

instrucción actual del flujo y el control de la ALU. La Fig. 3.2 describe el esquema en bloques de la conformación del procesador (maestro).

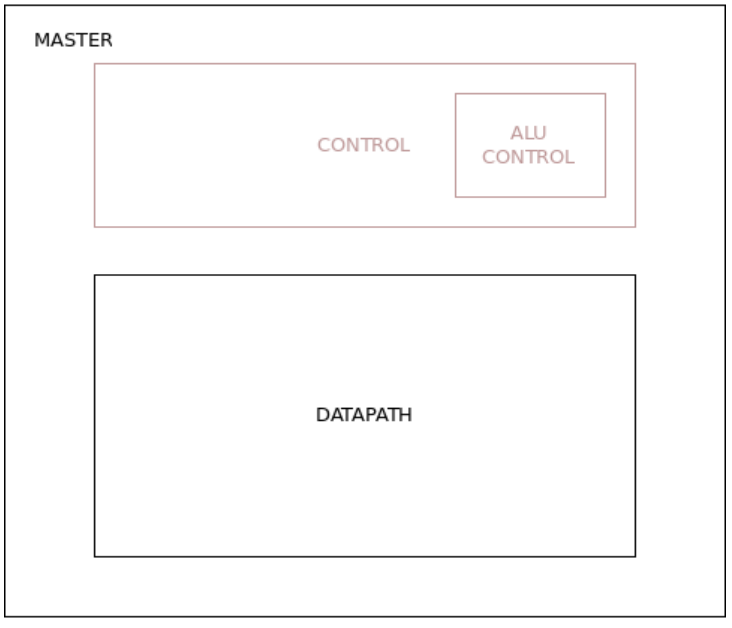


FIGURA 3.2: Esquema interno de la composición en bloques del maestro.

3.1.1. Datapath

En la primer subsección de este capítulo se abarca el diseño y funcionalidad de todos los elementos que componen el sistema. Éstos proveen medios de almacenamiento y transmisión de datos, es decir, conforman las vías por donde los datos transitan acorde con la necesidad de cada instrucción para ser manipulados por los elementos del procesador y llevar a cabo las tareas requeridas por el programa. Al conjunto de buses de comunicación, registros, multiplexores y unidades de cálculo se lo denomina *datapath*, y se explica a continuación la composición y conformación del correspondiente al procesador presentado junto con sus respectivas simulaciones.

3.1.1.1. Registros contador de programa y de instrucción

Los bloques fundamentales para integrar el diseño del *datapath* son los registros del contador del programa para señalar la dirección de la próxima instrucción (PC), el registro de instrucción que mantiene la instrucción actual durante toda su ejecución (IR), el banco de registros (*Register File*), la memoria y la ALU. Posteriormente se conectan entre estos otros bloques como multiplexores, extensores de signo, desplazamientos de bits y otros registros intermedios que almacenan datos de las etapas de la instrucción. Para el PC se diseña un bloque con una entrada y una salida de datos de 32 bits, una entrada de reloj, una de habilitación y una de reinicio (Fig. 3.3). La operación del bloque consiste en mostrar a la salida el dato de entrada, siempre y cuando la señal de habilitación esté en valor lógico alto (1) y la de reinicio en bajo (0), ante flancos ascendentes del reloj. En caso de señal de reinicio en alto, independientemente de la habilitación del registro, el valor de la salida cambia a cero señalando la dirección inicial de la memoria donde comienza el bloque de instrucciones para este procesador. En el caso del IR las entradas y salidas son iguales que en el PC, con la diferencia que éste no cuenta con señal de entrada de reinicio (Fig. 3.3). Su funcionamiento es equivalente excepto por la capacidad de ser reiniciado. La Fig. 3.4 muestra la simulación del comportamiento de los bloques PC e IR respectivamente.

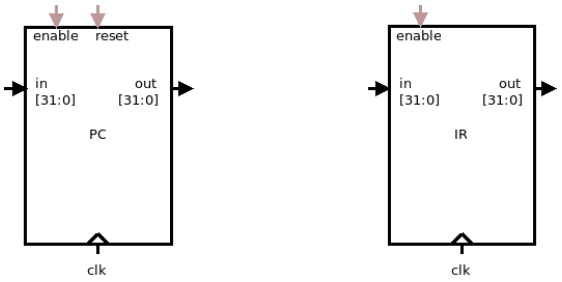


FIGURA 3.3: Esquema en bloques de los registros del contador de programa (PC) y registro de instrucción (IR).

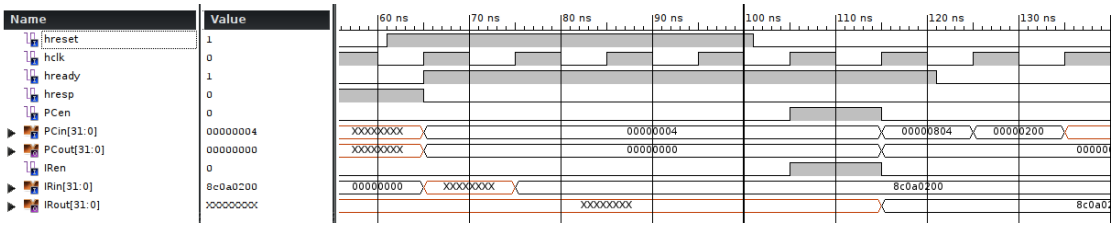


FIGURA 3.4: Simulación de funcionamiento del contador de programa y el registro de instrucción.

3.1.1.2. Banco de registros

El banco de registros está conformado por 32 registros de 32 bits, posee una señal de reloj, una de señalización de lectura/escritura, dos entradas de 5 bits (**addrReadReg1** y **addrReadReg2**) para seleccionar registros a leer y cuyos contenidos son mostrados por las salidas de 32 bits **readData1** y **readData2** respectivamente. También tiene una entrada de datos de 32 bits **dataWrite** que en caso de escritura provee los datos a almacenar en el registro indicado por la entrada de 5 bits **addrWriteReg**. Finalmente cuenta con una salida de 32 bits denominada **watchdog**, conectada al registro reservado para el *watchdog* del procesador (26). El modelo en bloque del banco de registros se muestra en la Fig. 3.5. El bit más significativo (31) del registro del *watchdog* indica si el mismo está habilitado (con un 1) y los 31 bits restantes son utilizados para almacenar el valor del temporizado del *watchdog* permitiendo al usuario configurar la cantidad de ciclos de espera tolerables durante una transacción en el bus. Si el *watchdog* está habilitado una vez cumplido dicho lapso éste interrumpirá el funcionamiento del procesador y saltará al vector de interrupción del *watchdog* en la última dirección de memoria (255) reservada para que el usuario especifique la tarea a realizar en tal caso. Otro registro reservado del banco es el cero, cuyo valor es siempre nulo para propósitos de la arquitectura y no puede ser alterado por el usuario. La lectura de registros del banco es asincrónica, de manera que no depende de flancos del reloj sino que que las salidas **readData1** y **readData2**

reflejan los datos contenidos en los registros seleccionados por las entradas **addrReadReg1** y **addrReadReg2**. Por el contrario la escritura de registros es sincrónica y ante flancos ascendentes del reloj con la señal de escritura **regWrite** habilitada (1) se escribe el dato de la entrada **dataWrite** en el registro señalado por **addrWriteReg**. La Fig. 3.6 muestra los resultados de simulación del bloque *Register File*.

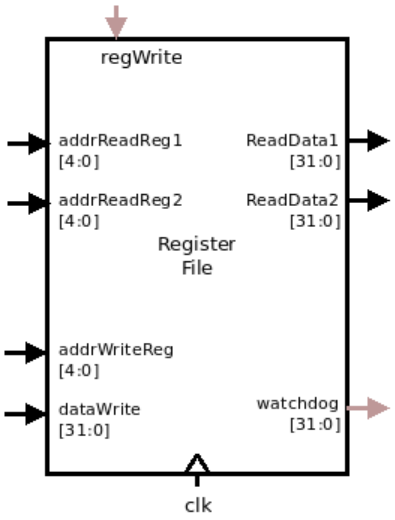


FIGURA 3.5: Esquema en bloque del banco de registros (Register File).

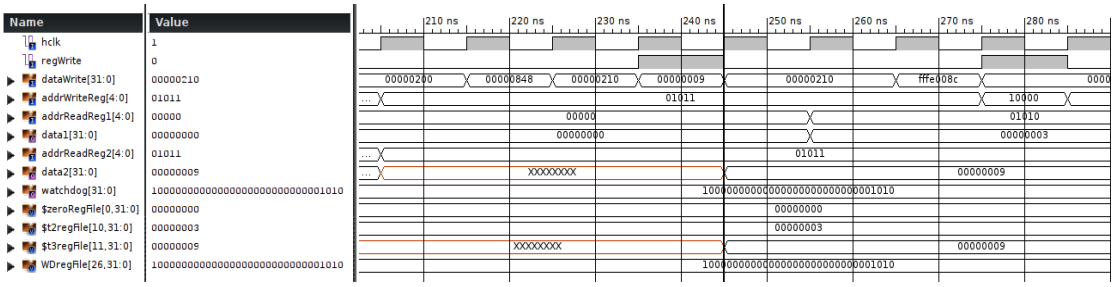


FIGURA 3.6: Simulación de funcionamiento del banco de registros.
Se escribe en el registro $\$t3$ el valor 9 en $t = 245\ ns$.

3.1.1.3. Unidad aritmético-lógica

La ALU es un bloque enteramente combinatorio donde se llevan a cabo todas las operaciones aritméticas con signo y lógicas entre registros del banco. Consta de tres entradas, dos de ellas de 32 bits para los respectivos operandos (**aluInA**, **aluInB**) y la tercera de 4 bits (**aluControl**) es la señal del control de la ALU que determina la operación a realizarse. Las salidas también son tres, una de 32 bits para el resultado (**aluResult**) y dos de un bit para flags de cero (**aluZero**) y desbordamiento (**aluOverflow**). Dado que está integrada por completo de lógica combinatoria, los resultados de las operaciones

son asincrónicos al reloj y los mismos se actualizan ante cambios en los operandos o la operación del mismo modo que lo hacen los flags. Ante resultados nulos el flag de cero cambia a estado activo (1) e inactivo en caso contrario (0). Para aquellas operaciones aritméticas cuyo resultado exceda la capacidad de representación en 32 bits, se produce desbordamiento señalizado con **aluOverflow** de manera activa (1) e inactiva en caso adverso (0). En las figuras 3.7 y 3.8 se ve respectivamente el modelo en bloque de la ALU y su simulación.

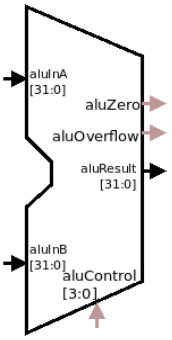


FIGURA 3.7: Esquema en bloque de la unidad aritmético-lógica.

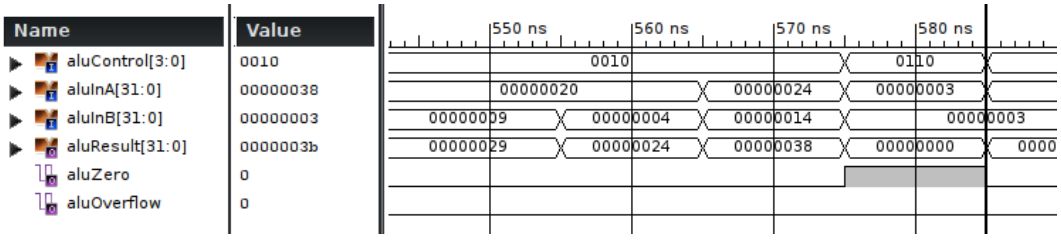


FIGURA 3.8: Simulación del funcionamiento de la ALU.
Hasta $t = 575\ ns$ **aluResult** refleja la suma de los operandos A y B. Luego la resta con resultado nulo activa el *flag* **aluZero**.

3.1.1.4. Memoria

Para el diseño de la memoria se crea un bloque de 128 palabras (512 bytes) de capacidad de almacenamiento, con entradas y salidas de datos y señales de control para cumplir con las especificaciones del protocolo AMBA. Para los datos se cuenta con dos entradas y una salida, todas de 32 bits. Una de las entradas es de dirección (**haddr**) y otra de datos para escritura (**hwdata**), para la lectura de datos se tiene la salida **hrdata**. Las entradas de control incluyen la señal de selección (**hsel**), la señal de bus libre (**hready**), el reloj (**hclk**), la habilitación de escritura (**hwrite**), y la señal de transacción bloqueada (**hmastlock**) todas ellas de un bit. Otras señales de entrada de control son la de tamaño de transacción (**hsize**) de tres bits, la señal de ráfagas (**hburst**) del mismo ancho, la señal

de tipo de transferencia (**htrans**) de 2 bits y finalmente la señal de transacción protegida (**hprot**) de 3 bits de ancho. Las señales de salida de control son dos, ambas de un bit, e indican liberación del bus por parte de la memoria (**hreadyout**) y la señalización de error de la transferencia (**hresp**). La Fig. 3.9 muestra el modelo del bloque.

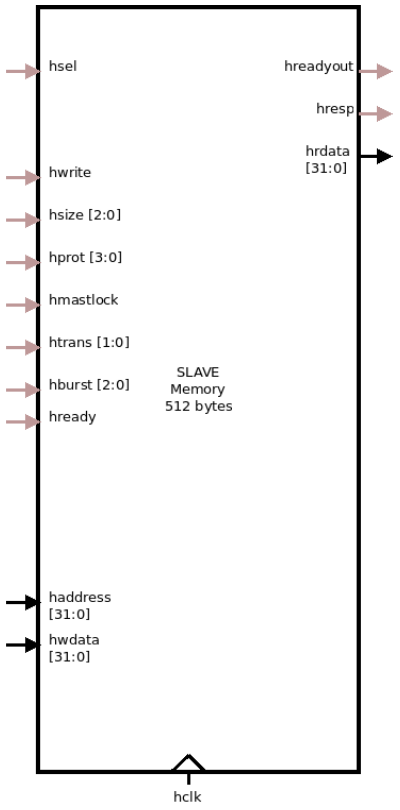


FIGURA 3.9: Esquema en bloque de la memoria.

El propósito de la memoria es almacenar datos que manipula el procesador y el programa. Para cumplir con las especificaciones AHB-Lite la memoria posee un controlador interno que permite escribir y leer datos en tres diferentes formatos (palabra, media palabra y byte) mediante las señales de control que identifican el tipo y formato de la transacción. Considerando que el bus se encuentra disponible (**hready** = 1 y **hresp** = 0), al producirse un flanco ascendente de reloj el controlador almacena en registros internos los parámetros de transacción como la dirección, el tamaño de la transacción, si se trata escritura y si el esclavo respectivo al controlador se encuentra seleccionado. Se lleva a cabo así la primera fase de una transferencia de datos acorde al protocolo. En el siguiente flanco ascendente sucede la segunda fase y si la transacción es de escritura, se almacena el dato acorde a los parámetros registrados por el controlador al mismo tiempo que se almacenan los nuevos para la siguiente transacción. Para las transferencias de datos menores a 32 bits el protocolo indica que es responsabilidad del maestro

enviar los datos de interés por las líneas activas en las que se pretende que el esclavo las almacene. Es decir que si se quisiera almacenar 8 bits en el byte más significativo de una determinada palabra en memoria, los datos enviados por el maestro deben colocarse en los 8 bits más significativos del bus de datos (31 al 26) ya que el controlador de memoria interpreta que los datos de interés se encuentran en esos 8 bits y no actualiza el resto de la palabra. El funcionamiento descrito evidencia que la escritura de datos se realiza de forma sincrónica. Contrariamente, el controlador lee datos de la memoria de manera asincrónica actualizando los datos del bus de salida **hrdata** según se indique en el bus de dirección **haddr**. Independientemente del tamaño de datos indicado en la lectura, el controlador actualiza el bus con la palabra completa indicada por la dirección. El protocolo AHB-Lite especifica que es el maestro el encargado de tomar del bus el subconjunto de bits apropiado acorde al tamaño de la transacción indicado en **hsize**. La Fig. 3.10 muestra una simulación del funcionamiento del bloque.

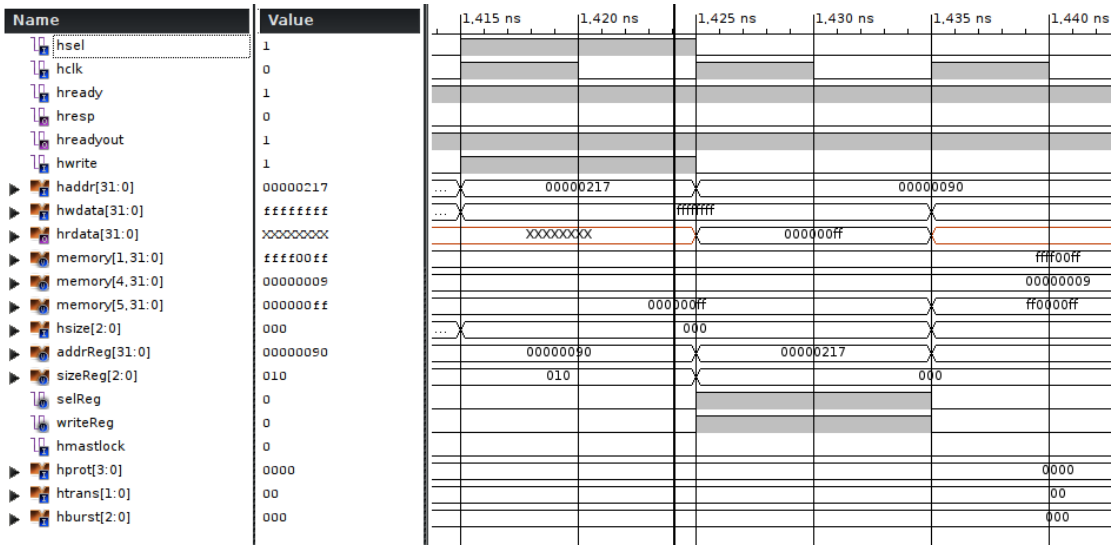


FIGURA 3.10: Simulación de funcionamiento de la memoria.
En $t = 1{,}425\ ns$ se indica en la fase de dirección la escritura del tercer byte del contenido de **hwdata** en **memory[5]**, en la fase de datos ($t = 1{,}435\ ns$) se produce la escritura.

3.1.1.5. Decodificador y multiplexor AMBA

Como se menciona en párrafos anteriores existe más de un esclavo en el sistema propuesto y por ende debe contarse con un decodificador y un multiplexor para determinar el esclavo destino o fuente de datos en una transacción con el maestro. El multiplexor (Fig. 3.12) es un bloque con dos entradas de 32 bits donde arriban los datos de los

buses de lectura de los esclavos. Posee también seis entradas de un bit: **hreadyout0** y **hreadyout1** para las señales de bus liberado de sus respectivos esclavos y de manera análoga **hresp0** y **hresp1** para el error de transferencia, por último **hreadyout_nomap** y **hresp_nomap** son entradas respectivamente conectadas a valores lógicos 0 y 1 para deshabilitar la transferencia de datos en caso de requerirse una transacción de una dirección de memoria no mapeada. Cuenta además con una señal de reinicio y otra de reloj para el registro interno que almacena la selección del esclavo de la transferencia actual y finalmente la entrada de selección de multiplexor (**muxSel**) que es de dos bits. Entre sus salidas se encuentra la de datos de lectura (**hrdata**) de 32 bits, y la de liberación de bus (**hready**) y error de transferencia (**hresp**). La función del multiplexor de esclavos es precisamente mostrar en su salida, según indique el registro interno de selección de multiplexación, el bus de lectura de datos y las señales de control del esclavo involucrado de la transacción. La información de selección es provista por el decodificador. El bloque decodificador se diseña con una entrada de 32 bits para la dirección de la transacción (**haddr**), dos salidas de un bit para los selectores de habilitación de cada esclavo (**hsel**) y una salida de dos bits para manipular la selección del multiplexor (**muxSel**) como muestra la Fig. 3.11. Su funcionamiento consiste en decodificar la dirección provista por el maestro y habilitar el esclavo indicado para la transacción, mediante las señales de selección, e indicar al multiplexor que debe proveer la información de dicho esclavo. Se realiza una simulación de funcionamiento para cada bloque, plasmadas en las figuras 3.13 y 3.14.

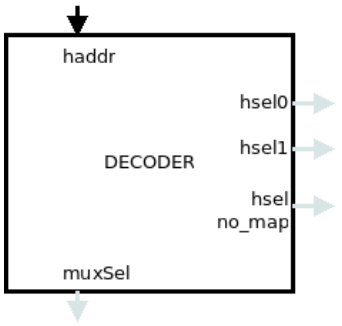


FIGURA 3.11: Esquema en bloque del decodificador del sistema.

3.1.1.6. Composición del *datapath*

Una vez disponibles los bloques elementales, se procede a interconectarlos para confeccionar el *datapath* como se detalla en los siguientes párrafos. Comenzando por el registro

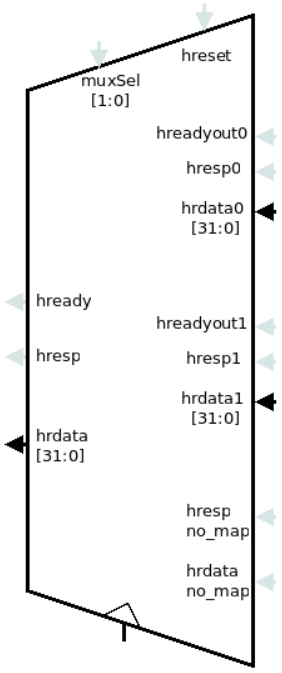


FIGURA 3.12: Esquema en bloque del multiplexor de los esclavos.

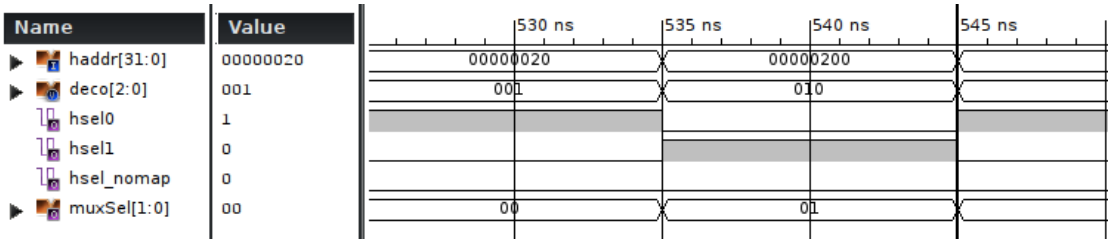


FIGURA 3.13: Simulación de funcionamiento del multiplexor del sistema.
Cuando en $t = 535\ ns$ cambia la dirección en **haddr** a una mapeada en el otro esclavo, cambian las señales de selección **hsel** y **muxSel** correspondientemente.

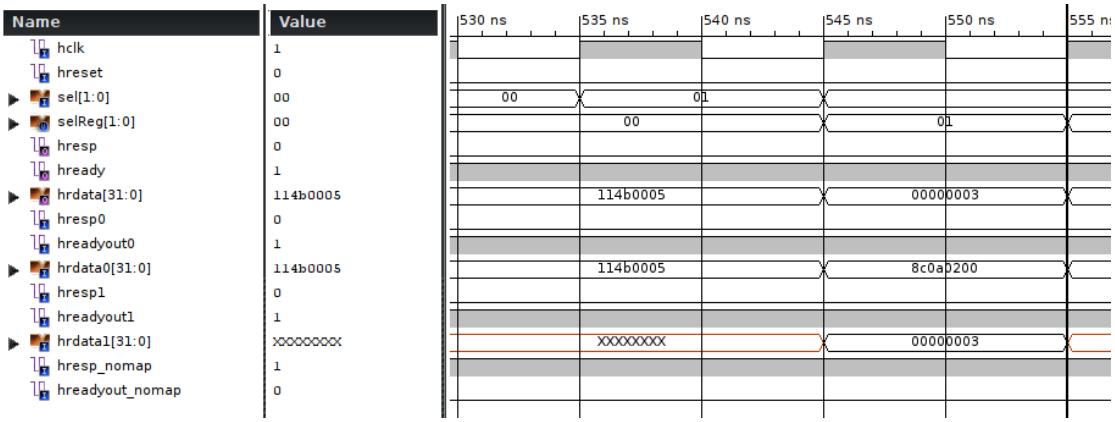


FIGURA 3.14: Simulación de funcionamiento del decodificador del sistema.
La salida **hrdata** refleja la salida **hrdata0** hasta que en $t = 545\ ns$ **selReg** cambia y se refleja la salida **hrdata1** del otro esclavo.

de instrucciones, se conectan los bits que conforman la salida a diferentes destinos acorde con el campo de la instrucción que representan. De este modo los bits de salida [25:21], [20:16] y [15:11] que corresponden a campos registros en las instrucciones tipo R deben ser conectados a entradas del banco de registros como muestra la Fig. 3.15. El primer campo de registro es siempre un registro para lectura y se conecta en la entrada **addrReadReg1**. Los dos campos de registro restantes pueden indicar un registro a ser escrito en el banco y por ende deben multiplexarse sus respectivos bits a la entrada **addrWriteReg** del banco de registros, para ello se coloca un multiplexor de dos entradas de 5 bits como se ve en la Fig. 3.15. El segundo campo de registro también puede ser un registro a ser leído en una instrucción, y sus respectivos bits se conectan a la entrada **addrReadReg2** del banco.

En los 16 bits menos significativos del IR se encuentra el campo inmediato utilizado en las instrucciones de tipo I que, como muestra la Fig. 3.16, son conectados a un bus con tres posibles destinos: un extensor de signo, un extensor de ceros y un bloque de desplazamiento de 16 bits. El extensor de signo es un bloque combinatorio cuya entrada de 16 bits toma el dato inmediato y conserva el signo, replicando el bit más significativo 16 veces, para expresar a la salida dicho dato en 32 bits. Se requiere la funcionalidad de este bloque para las operaciones aritméticas entre registros e inmediatos. En la Fig. 3.16 se aprecia también que la salida de signo extendida se conecta a otro bloque de desplazamiento de dos bits. Éste desplaza los dos bits más significativos de su entrada de 32 y coloca dos ceros en los bits menos significativos. Su funcionalidad se precisa

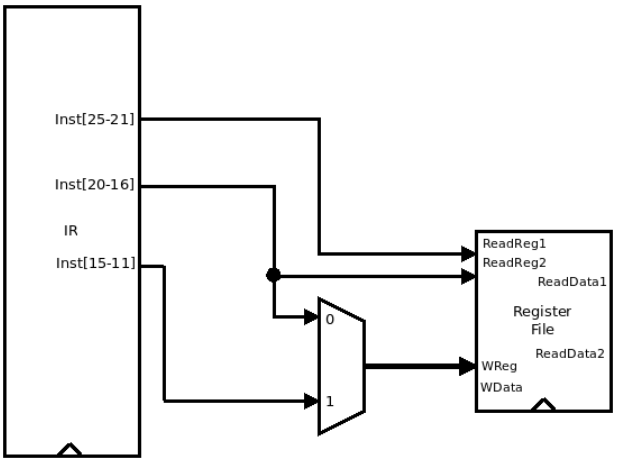


FIGURA 3.15: Primer paso de la confección del *datapath*.
Conexión entre los campos de registro del registro de instrucción y el banco de registros.

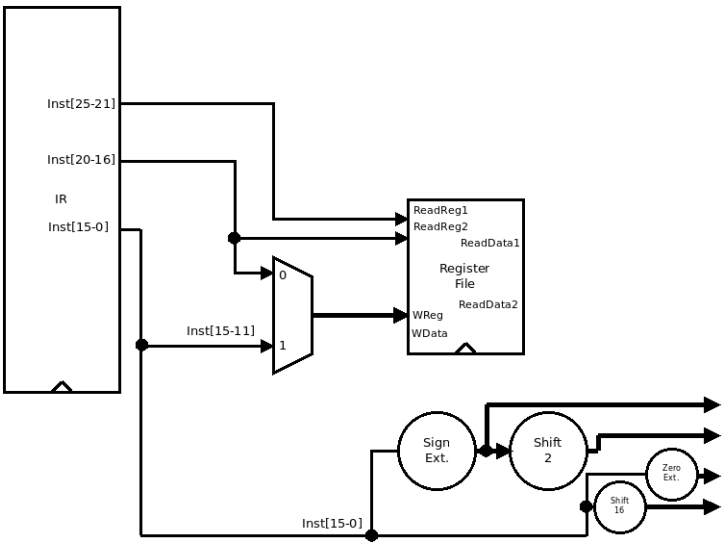


FIGURA 3.16: Segundo paso de la confección del *datapath*.
Se agregan los extensores de signo conectados al campo inmediato del registro de instrucción.

para el cálculo de direcciones de memoria en las instrucciones *branch*. Similarmente a la extensión de signo, el bloque extensor de ceros tiene una entrada de 16 bits y una salida de 32. En vez de replicar el bit 16 de la entrada, coloca 16 ceros para extender a 32 bits el dato y ser utilizado en las operaciones lógicas entre registros e inmediatos. Por último el bloque de desplazamiento de 16 bits conforma un dato de salida de 32 bits colocando los 16 bits del inmediato en la parte alta de la palabra y ceros en la parte baja. Su funcionalidad es utilizada en las instrucciones que cargan un valor inmediato en registros. La tabla 3.1 ejemplifica el funcionamiento de cada uno de los extensores.

Los datos de registros leídos del banco (IR) deben ser almacenados por registros intermedios que retienen sus datos a lo largo de la ejecución de la instrucción. Para eso se colocan los registros de 32 bits, A y B como muestra la Fig. 3.17, del mismo tipo que

Bloque	Entrada	Salida
Extensor de signo	0010111011100001	00000000000000000010111011100001
	1101110010000001	11111111111111111101110010000001
Extensor de cero	0010111011100001	00000000000000000010111011100001
	1101110010000001	00000000000000001101110010000001
Desplazamiento en 2 bits (32 a 32 bits)	00010010001011000010111011100101	01001000101100001011101110010100
Desplazamiento en 16 bits	1101110010000001	11011100100000010000000000000000

TABLA 3.1: Ejemplos de funcionamiento entrada-salida de los bloques extensores.

se utiliza para el IR. A sus entradas se conectan las salidas de datos del banco de registros **addrReadData1** y **addrReadData2** respectivamente. Finalmente la entrada **dataWrite** recibe los datos a ser almacenados en los registros del banco que pueden provenir de dos fuentes: un resultado calculado por la ALU en una operación entre registros o un dato cargado de memoria. Se conecta un multiplexor de dos entradas de 32 bits según indica la Fig. 3.17.

A continuación se incorpora la unidad aritmético-lógica al *datapath*. Recordando que este bloque realiza cálculos para múltiples propósitos, se repasan los mismos con el objetivo de identificar las distintas fuentes que deben ser provistas de conexión a las entradas de los operandos. La actualización del PC para señalar la siguiente instrucción a ejecutar consiste en sumar 4 al PC actual, requiriendo que el valor del PC y dicha constante sean provistos a los operandos. Por otra parte los accesos a memoria se realizan calculando la dirección a partir de un registro leído del banco y un *offset* indicado en el campo inmediato extendido en signo, para ello se dispone de una conexión entre la salida del registro A y del extensor de signo a la entrada de operandos. En esta instancia se hace evidente la necesidad de multiplexores para seleccionar las fuentes que suministran los operandos según la instrucción. Continuando con las posibles operaciones, se tiene las instrucciones tipo R que operan dos registros del banco. Como ya fue planteado el camino para el registro A, resta proveer uno para el registro B. Las instrucciones que operan registros con inmediatos precisan distinto tratamiento según sean aritméticas o lógicas,

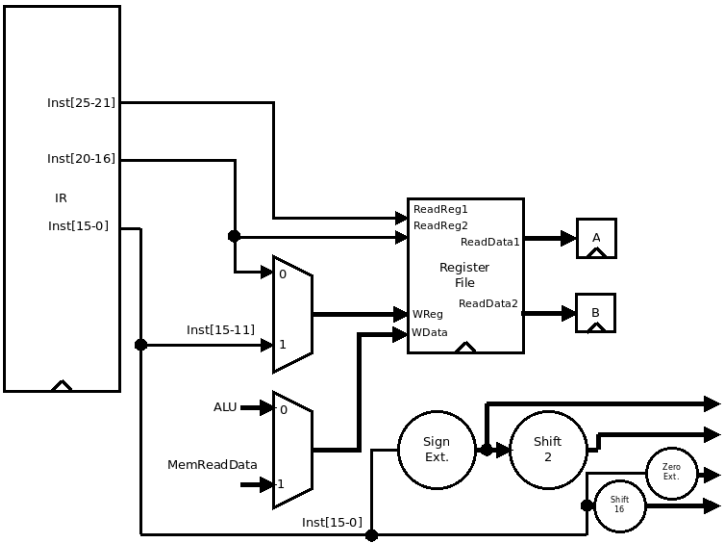


FIGURA 3.17: Tercer paso de la confección del *datapath*.
Registros auxiliares A y B, multiplexor para datos a escribir en el banco de registros.

en el primer caso se utilizan las mismas conexiones dispuestas para la lectura o escritura en memoria mientras que para el segundo caso debe extenderse el inmediato con ceros. Se provee entonces una conexión del extensor de ceros a una de las entradas de operando. Similarmente se necesita una conexión entre el bloque desplazamiento de 16 bits para las operaciones de carga de constantes a registro. Por último, en el cálculo de dirección de *branch* se adquiere el valor del salto desde el campo inmediato con signo y desplazado en dos bits, precisando un camino desde el bloque desplazamiento de dos hacia una entrada de operando. Resumiendo todas las conexiones repasadas y como indica la Fig. 3.18, se conecta un multiplexor de dos entradas a una de las entradas de operando de la ALU y en éste se coloca la salida del registro A y una salida proveniente del PC. En la entrada restante del operando de la ALU se conecta la salida de un multiplexor de cinco entradas, en las cuales se colocan el registro B, la constante 4, la salida del extensor de signo, la salida del bloque desplazamiento en 2 bits, la salida del extensor de cero y la del bloque desplazamiento de 16 bits. La salida de resultado de la ALU es suministrado a un registro (ALUOut) para almacenamiento intermedio de la operación.

En este punto solo restan ubicar y conectar el PC y los esclavos con el resto de los bloques que arbitran en protocolo. Para la entrada del contador de programa se conecta un multiplexor de cuatro entradas como indica la Fig. 3.19, dado que son cuatro las

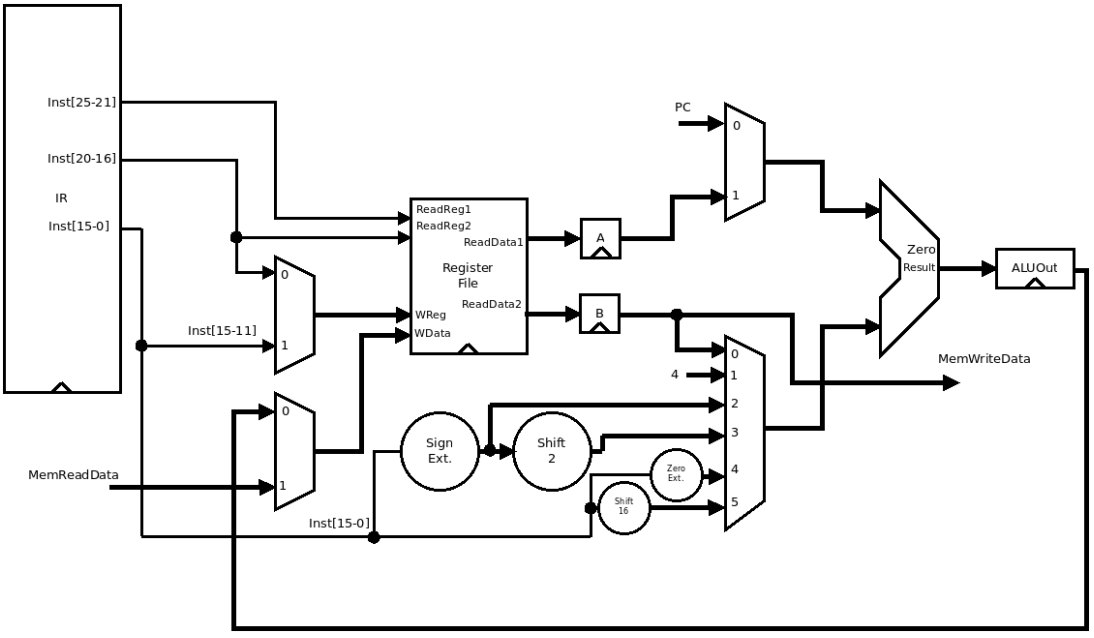


FIGURA 3.18: Cuarto paso de la confección del *datapath*.
Adición de la ALU, multiplexores para suministrar los operandos desde distintas fuentes y el registro auxiliar de la ALU.

posibles fuentes que actualizan su contenido. En la ejecución secuencial de un programa el PC se actualiza en 4, resultado de un cálculo efectuado por la ALU y por ello se conecta la salida **aluResult** a una de las entradas del multiplexor. También las instrucciones *branch* pueden potencialmente modificar el PC, en ellas la dirección a saltar se calcula por la ALU y un ciclo más tarde ésta evalúa la condición de *branch*. La salida del registro ALUOut se conecta a otra entrada del multiplexor y es la que proveerá la dirección al PC (un ciclo posterior a su cálculo) en caso de cumplirse la condición. Acorde con el funcionamiento del *watchdog* explicado en párrafos anteriores, éste puede interrumpir la secuencialidad de la ejecución de instrucciones actualizando el PC para que éste señale la última dirección en memoria donde se encuentra el vector de interrupción del *watchdog*. Se conecta entonces la tercer entrada del multiplexor del PC al valor constante 1023. Finalmente los saltos incondicionales son otro tipo de instrucción que altera el estado del PC. La dirección del salto se compone de una dirección indicada en los 26 bits menos significativos del registro de instrucciones, que se desplaza dos bits agregando ceros, y los cuatro bits más significativos del PC de la próxima instrucción. Para ello se conecta el registro desplazamiento de la figura 3.19 cuya entrada es de 26 bits y su salida de 28 se concatena con los bits del PC. Con esta disposición el salto incondicional permite saltar a cualquier dirección alineada en una región de 256MB del PC. Por último, el bus de dirección del *datapath* que se conecta al decodificador y los esclavos es provisto de un multiplexor de dos entradas, ya que la dirección puede provenir del PC cuando se busca una nueva instrucción o por la ALU cuando se calcula una dirección en memoria para cargar o almacenar datos.

Es necesario analizar las especificaciones del protocolo AHB-Lite y de la arquitectura del procesador a la hora de conectar los esclavos y la lógica de artbitraje al resto del *datapath*. En las instrucciones de escritura en memoria se guarda el estado de alguno de los registros del banco según lo especifica la instrucción en el campo de registro destino. Dicho campo es leído en la entrada **addrReadReg2** del banco y se almacena en el registro B de 32 bits. Para escrituras de palabras en memoria bastaría con conectar la salida del registro al bus de datos **hwdata**, sin embargo se pueden efectuar escrituras de menor ancho y es necesario un medio que adecue el formato de los datos según el funcionamiento de la arquitectura y lo especificado por el protocolo. Acorde con la arquitectura, en el almacenamiento de un registro de 8 o 16 bits en memoria los bits activos (que contienen los datos) deben encontrarse siempre en la parte baja del registro.

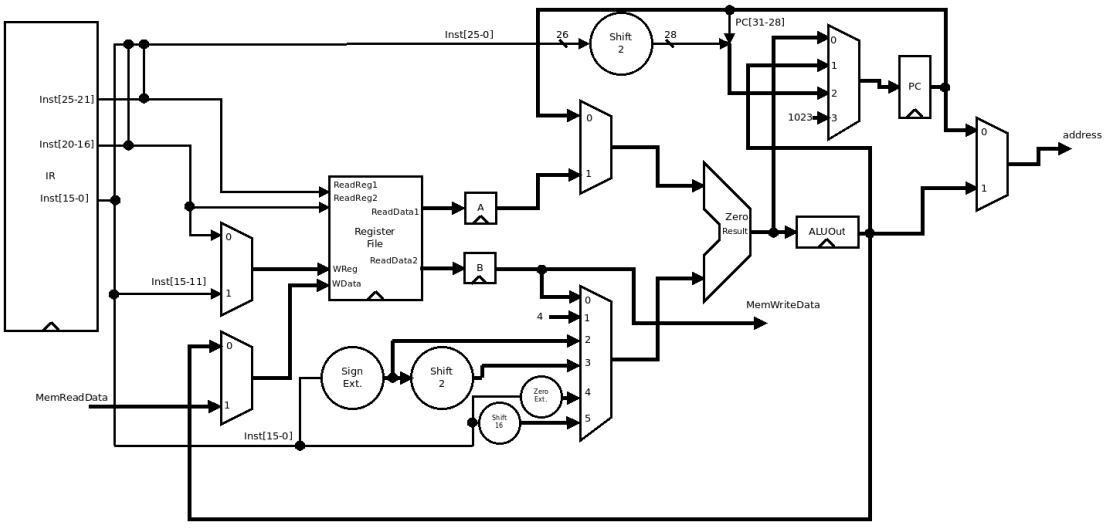


FIGURA 3.19: Quinto paso de la confección del *datapath*.
Se agrega el registro del PC, multiplexor para proveer sus distintas fuentes de escritura, registro de desplazamiento para la dirección de salto y multiplexor para los datos del bus de direcciones.

Esto quiere decir que los datos deben estar en los bits 7 a 0 de un registro de 32 (para 8 bits) o de 15 a 0 (para 16 bits). Por otra parte el protocolo especifica que en el bus de datos deben colocarse lo datos de interés en las líneas activas en las que se pretende que se almacenen, implicando que si se desea almacenar un dato de un byte en los bits 24 a 16 de una palabra en memoria los datos deben ser enviados por las líneas 24 a 16 del bus. Para lograr adaptar las líneas por donde los datos salen del procesador a donde deben ser transmitidas por el bus, se coloca un multiplexor de tres entradas de 32 bits y cuya salida se conecta al bus de escritura de datos **hwdata** como la figura 3.20 indica. Durante escrituras de un byte se replica cuatro veces los bits 7 a 0 del registro B y se concatenan para ubicarlo en todas las líneas de byte del bus **hwdata** por medio de una de las entradas del multiplexor. De forma similar se resuelve la escritura de medias palabras, solo que se replica dos veces en el bus los 16 bits menos significativos del registro B. En la entrada restante del multiplexor se conecta la salida de B de forma directa para la escritura de palabras. De esta manera el esclavo puede recibir los datos en la línea activa correspondiente a la transacción.

El ejemplo simulado que se muestra en la Fig. 3.21 muestra que el registro B cambia su valor a $(FFFF00FF)_{16}$ mientras que la señal **hsize** indica transferencia de palabra $(10)_2$, la salida del multiplexor refleja el mismo valor que el registro. Cuando **hsize**

cambia a transferencia de byte $(00)_2$, la salida cambia mostrando el contenido del byte menos significativo de B extendido en todo el bus $(FFFFFFFF)_{16}$.

De modo similar se requiere de lógica para dar el correcto formato a los datos transferidos en las instrucciones de lectura. En éstas se almacena un dato proveniente de la memoria en algún registro del banco indicado por el campo de registro destino de la instrucción. Indistintamente del ancho de bits de la transferencia el esclavo escribe en el bus de datos **hrdata** la palabra completa (32 bits) que contiene la información requerida. Como indica el protocolo AHB-Lite, es el maestro quien selecciona los datos pertinentes según las señales de control de la transferencia. Por otra parte la arquitectura MIPS32 especifica que en el almacenamiento de datos de 8 o 16 bits en un registro del banco se efectúa en los bits menos significativos del mismo y con la correspondiente extensión de signo. Para la adaptación se crea un bloque denominado Formato de Datos de Memoria mostrado en la Fig. 3.22 y en cuya entrada de 32 bits se coloca el bus de lectura **hrdata**. Cuenta con un bit de habilitación, una entrada de dirección de dos bits, la señal de tamaño de transferencia AMBA (**hsize**) de tres y la salida de datos de 32 bits. Internamente posee un registro donde guarda los dos bits menos significativos de la dirección de memoria que calcula la ALU durante la instrucción. El bloque identifica el tamaño de la transferencia y la posición del dato en el bus **hrdata** mediante las señales de control **hsize** y los dos bits menos significativos de la dirección, luego el dato se extiende en signo y es transmitido a la entrada de escritura del banco de registros (**dataWrite**) a través del mutiplexor.

La Fig. 3.23 muestra la simulación del bloque, cuando la señal de habilitación se halla activa (1) la señal **hsize** indica una transferencia de byte $(000)_2$ y la dirección registrada en **addr10Reg** $(10)_2$ especifica que se trata del segundo byte del bus **hrdata** cuyo valor es $(FF)_{16}$. Se aprecia que la salida representa dicho valor extendido en signo.

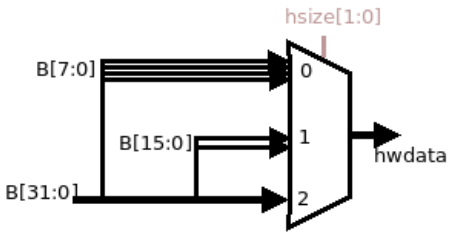


FIGURA 3.20: Multiplexor de adaptación.
Adapta los datos del banco de registros al bus de datos acorde con el tamaño de la transferencia.

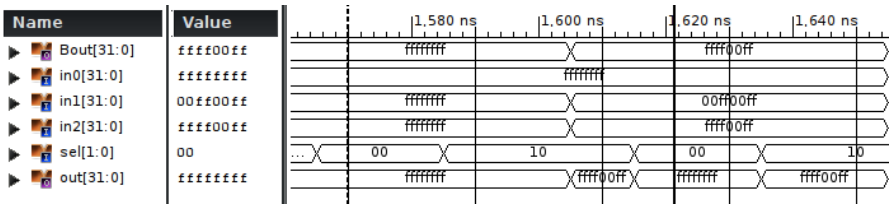


FIGURA 3.21: Simulación de funcionamiento de adaptación de datos del multiplexor.

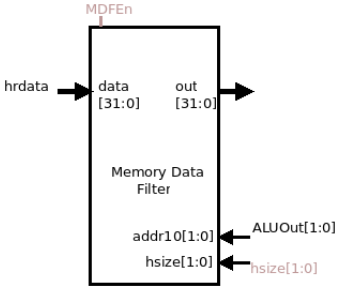


FIGURA 3.22: Esquema en bloque del Formato de Datos de Memoria.

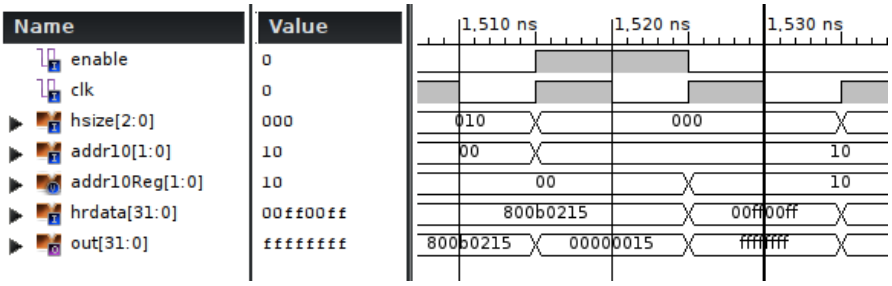


FIGURA 3.23: Simulación de funcionamiento del Formato de Datos de Memoria.

Con todos los bloques explicados y las conexiones detalladas pueden finalmente ubicarse las dos memorias y la lógica de arbitraje para completar el *datapath*. En la entrada de dirección de cada memoria y del decodificador se coloca el bus **haddr** y en la de datos de entrada de las memorias el bus **hwdata**. Cada salida de datos de los esclavos se conecta a su respectiva entrada **hrdata0** y **hrdata1** del multiplexor, y la salida de éste al bus **hrdata**. El diagrama completo del *datapath* es el mostrado en la Fig. 3.24.

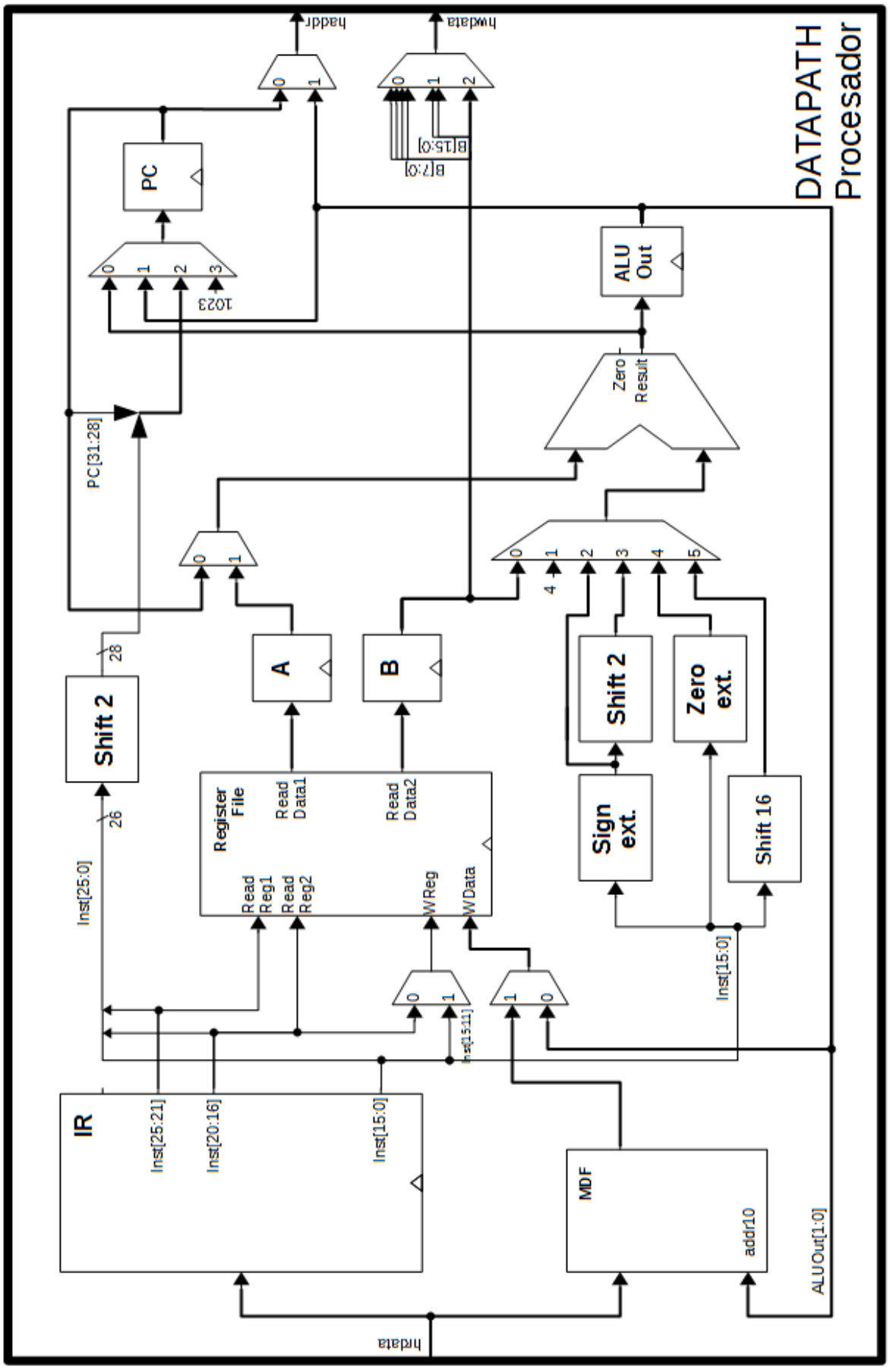


FIGURA 3.24: Diagrama del *datapath* del maestro.

3.1.2. Control

En 3.1.1 se proveen conexiones y medios que permiten la transmisión de datos entre elementos necesarios para realizar la tarea de una instrucción determinada. Es fundamental que además del medio se cuente con el correcto arbitraje y temporizado para que los datos pertinentes a la tarea sean los que se transmiten por los medios provistos. Dicho conjunto de señales son las denominadas de control y se agrupan formando entradas y salidas de un bloque llamado de tal forma. Éste se diseña como una máquina de estados que dependiendo de las señales de control y del estado actual, evoluciona a otros estados que actualizan los valores de las señales. Con cada ciclo de reloj la máquina evoluciona de estado en estado, impulsando el flujo de datos a través de los elementos que los operan, para producir el resultado deseado de las instrucciones que se ejecutan en el programa del procesador.

En esta subsección del capítulo se detalla la importancia de cada señal de control implicada en el funcionamiento del procesador y en la ejecución de cada instrucción implementada. Se explica el diseño del bloque de control y el tratamiento de las señales del protocolo AMBA AHB-Lite que no son utilizadas por el procesador. Se explica la máquina de estados del procesador construyéndola a partir de las instrucciones implementadas de la arquitectura. Se desarrolla estado por estado a lo largo de cada instrucción y el manejo de las señales de control que se requieren para realizarlas.

Las entradas del bloque principal de control mostrado en la Fig. 3.25 son **watchdog[31:0]**, **op[5:0]**, **hresp**, **hready**, **hreset** y la del reloj **hclk**. La primera de ellas provee la información del *watchdog* al control, que posteriormente se detalla como es utilizada por éste. El *opcode* de la instrucción es provisto al bloque por medio de **op[5:0]** y es clave para la determinación del flujo de datos y la ejecución de la instrucción. Las tres señales restantes corresponden al protocolo AHB-Lite y son respectivamente la señalización de liberación del bus, el estado de error de una transferencia y la señal de reinicio del sistema completo. El conjunto de salidas del bloque puede separarse en tres grupos: habilitaciones, señales de selección y señales del protocolo de comunicación. Entre las habilitaciones se encuentra la del registro de instrucción **IRWrite**, la del banco de registros **RegWrite**, las habilitaciones **PCWrite** (incondicional) y **PCWriteCond** (condicional) del PC y finalmente las de los registros auxiliares **MDFEn**, **AEn**, **BEn** y **ALUEn**. La lógica que manipula la habilitación del PC consiste en la habilitación

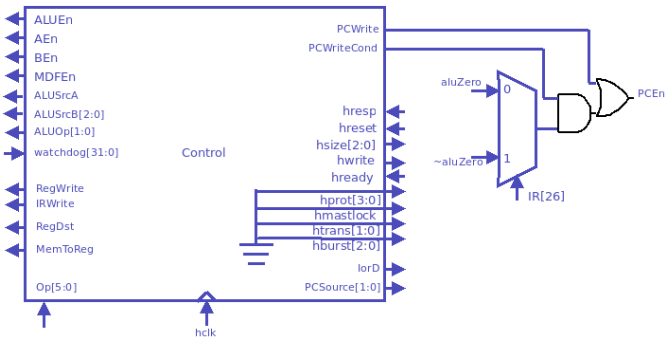


FIGURA 3.25: Esquema en bloque de la unidad de control del procesador.

incondicional o el conjunto de la habilitación condicional y la condición de igualdad o desigualdad dependiendo de qué instrucción de *branch* se trata. Las señales de selección son aquellas que son conectadas a los multiplexores y determinan qué entrada de éstos se refleja en la salida. La señal **RegDst** selecciona el campo de registro del IR que se provee como registro a escribir en el banco, mientras que **MemToReg** determina si el dato a escribir en ese registro proviene de memoria o del registro ALUOut. Las señales **ALUSrcA** y **ALUSrcB** se utilizan para proveer los operandos a la ALU según requiera la instrucción, **PCSource** para seleccionar la fuente de datos que actualiza al PC y **IorD** determina si el bus de direcciones es ocupado por una dirección contenedora de datos o instrucciones. Las señales del protocolo de comunicación y su funcionamiento fueron explicados en el Capítulo 1, cabe mencionar que **hsize** también cumple un propósito de selección en el multiplexor que provee los datos al bus de escritura **hwdata**. Además, debido a que la arquitectura MIPS32 no posee instrucciones de transferencia en ráfagas ni requiere de transferencias dedicadas, las señales **hprot**, **hmastlock**, **htrans** y **hburst** se encuentran conectadas a tierra como se esquematiza en la Fig. 3.25. Por último, la salida **aluOp** es una señal interna que se provee al sub-bloque ALUControl mostrado en la Fig. 3.26. Ésta determina si la instrucción requiere operar para un acceso a memoria, *branch*, operación entre registros u operación entre inmediatos. En los últimos dos casos, el bloque ALUControl determina el tipo de operación con de los 6 bits menos significativos del IR (campo *inst*) para las instrucciones tipo R y del *opcode* para las tipo I. Luego mediante su salida **aluControl** instruye la operación a la ALU (tabla 3.2).

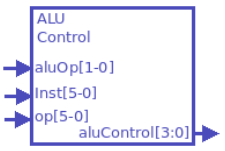


FIGURA 3.26: Esquema en bloque de la unidad de control de la ALU.

aluOp[1:0]	*inst*[5:0]	*opcode*[5:0]	**aluControl**[3:0]	Instrucción	Operación
00	xxxxxx	xxxxxx	0010	LB LH LW SB SH SW	Suma
01	xxxxxx	xxxxxx	0110	BEQ BNE	Resta
10	100000	000000	0010	ADD	Suma (R)
	100010		0110	SUB	Resta (R)
	100100		0000	AND	And (R)
	100101		0001	OR	Or (R)
	101010		0111	SLT	Resta (R)
	100111		0011	NOR	Nor (R)
	100110		0100	XOR	Xor (R)
11	xxxxxx	001000	0010	ADDI	Suma (I)
		001100	0000	ANDI	And (I)
		001101	0001	ORI	Or (I)
		001010	0111	SLTI	Resta (I)
		001110	0100	XORI	Xor (I)
		001111	0001	LUI	Or (I)

Tabla 3.2: Tabla de entrada-salida del sub-bloque ALUControl.
Salida **ALUControl** según entrada **aluOp** y los campos *inst* y *opcode*.

3.1.2.1. Estados compartidos y estados de instrucciones tipo R

Existen tres estados de la máquina que se caracterizan por ser empleados por todas las instrucciones. Uno de ellos es el estado de reinicio del procesador ($ST31 - RST$), que es accedido independientemente del estado actual de la máquina y de las salidas si la señal **hreset** está en estado lógico alto al producirse un flanco ascendente de **hclk**. El valor de las señales de control durante este estado se refleja en la Fig. 3.27, donde se aprecia que la mayoría de ellas son llevadas a nivel lógico bajo excepto por **hsize** y **MemtoReg**. La primera de estas señales se configura para transferencia de 32 bits de ancho y, dado que durante el reinicio la salida del PC cambia a cero, con **MemtoReg** = 1 se deja listo al procesador para comenzar a buscar la instrucción en la dirección cero de memoria una vez finalizado el estado de reinicio. Como se menciona anteriormente el primer estado de cualquier instrucción es el de búsqueda de instrucción ($ST0 - IF$), conocido en inglés como *fetch*. Durante éste se accede a la dirección en memoria indicada por el PC para leer los datos de la instrucción a ejecutar y que se almacenará en el registro de instrucción. La Fig. 3.27 muestra que la señal de control de escritura **hwrite** se coloca en estado bajo, el ancho de datos a transmitir se configura en 32 bits con **hsize** = $(010)_2$ y, dado que se lee una instrucción, la señal **IorD** se coloca en estado bajo para que el bus **haddr** transmita la salida del PC. La escritura del registro de instrucciones se habilita mediante **IRWrite** = 1. Simultáneamente se debe actualizar al PC para que cargue la dirección de la próxima instrucción, lo cual se lleva a cabo proveyendo a la ALU el valor actual del PC con **ALUSrcA** = 0, la constante 4 con **ALUSrcB** = $(001)_2$ y efectuando la suma (**ALUOp** = $(00)_2$). El resultado se almacena en el PC provisto por **PCSource** = $(00)_2$ y con la habilitación **PCWrite** = 1. En lo que respecta al protocolo de bus la máquina puede detenerse en este estado durante los ciclos de reloj que requiera la memoria para proveer los datos. Únicamente cuando las señales **hready** y **hresp** estén respectivamente en estado alto y bajo se evoluciona al siguiente estado también común a todas las instrucciones y conocido como decodificación ($ST1 - ID$). En éste se refleja en la salida del registro de instrucciones la instrucción buscada en el estado anterior, proveyendo los campos de datos a los bloques involucrados en la ejecución. Además, se supone predeterminadamente que durante este estado la instrucción se trata de un *branch* y se calcula la dirección del mismo. En caso de no tratarse de dicho tipo de instrucción, la dirección calculada se descarta en los estados de ejecución posteriores. Para la determinación de la dirección se proveen a la ALU la dirección del PC que ahora

apunta a la siguiente instrucción mediante **ALUSrcA** = 0 y el valor del salto obtenido del campo inmediato y extendido en signo con desplazamiento indicado con **ALUSrcB** = $(011)_2$. Se efectúa la suma mediante **ALUOp** = $(00)_2$ y se habilita para escritura los registros A, B y ALUOut como grafica la Fig. 3.27. Finalmente, a través del campo de *opcode* de la instrucción se determina durante la decodificación de qué instrucción se trata y se evoluciona al estado correspondiente. Para el caso ejemplificado en la figura 3.27 se trata de una instrucción del tipo R, con *opcode* = $(000000)_2$, y se procede al estado $(ST6-EX)$ en el cual se destinan a la ALU los operandos desde los registros A y B con **ALUSrcA** = 1 y **ALUSrcB** = $(000)_2$. Dichos registros fueron almacenados durante la transición de estados con los valores de los registros señalados en los campos del IR. La señal **ALUOp** = $(10)_2$ indica al control de la ALU que se trata de una instrucción tipo R y la operación a realizar es interpretada de los seis bits menos significativos del registro de instrucción (**IR**[**5:0**]), se habilita el registro de la ALU para almacenar el resultado de la operación con **ALUEn**. Al producirse el siguiente flanco de **hclk**, se evoluciona al estado $(ST7-RI)$ en el que se almacena el resultado de la operación en un registro del banco indicado en el campo correspondiente de la instrucción almacenada en IR. Dicho registro se provee al banco con el selector del multiplexor **RegDest** = 1 y se habilita la escritura del éste mediante **RegWrite**. Finalmente el dato del resultado se provee desde el registro de la ALU al banco de registros con el selector del multiplexor **MemToReg** = 0. Se observa de la Fig. 3.27 que durante este último estado de ejecución se requiere nuevamente que las señales del bus **hready** y **hresp** indiquen que éste se encuentra libre para otra transferencia. Esto es debido a que el próximo estado es el de búsqueda de la siguiente instrucción $(ST0-IF)$ y como se explica en **??** se requieren dos fases para las comunicaciones del protocolo. Durante $(ST7-IR)$ se coloca en el bus **haddr** la dirección de la próxima instrucción contenida en el PC, y es preciso que los esclavos hayan finalizado cualquier otra transferencia y el bus se encuentre libre para que en la transición al estado de búsqueda se transmita correctamente la instrucción. Este es un requisito que todos los estados previos a $(ST0-IF)$ deben cumplir, como se verá conforme se complete la estructura de la máquina de estados. La Fig. 3.28 muestra los resultados de simular la ejecución de dos instrucciones tipo R, una suma y una resta respectivamente, operando los registros $\$t2$ y $\$t3$ y guardando el resultado en $\$s0$. Puede apreciarse la evolución de todas las señales involucradas en los estados explicados, así como también la de los estados mismos en **currentState** y **nextState**.

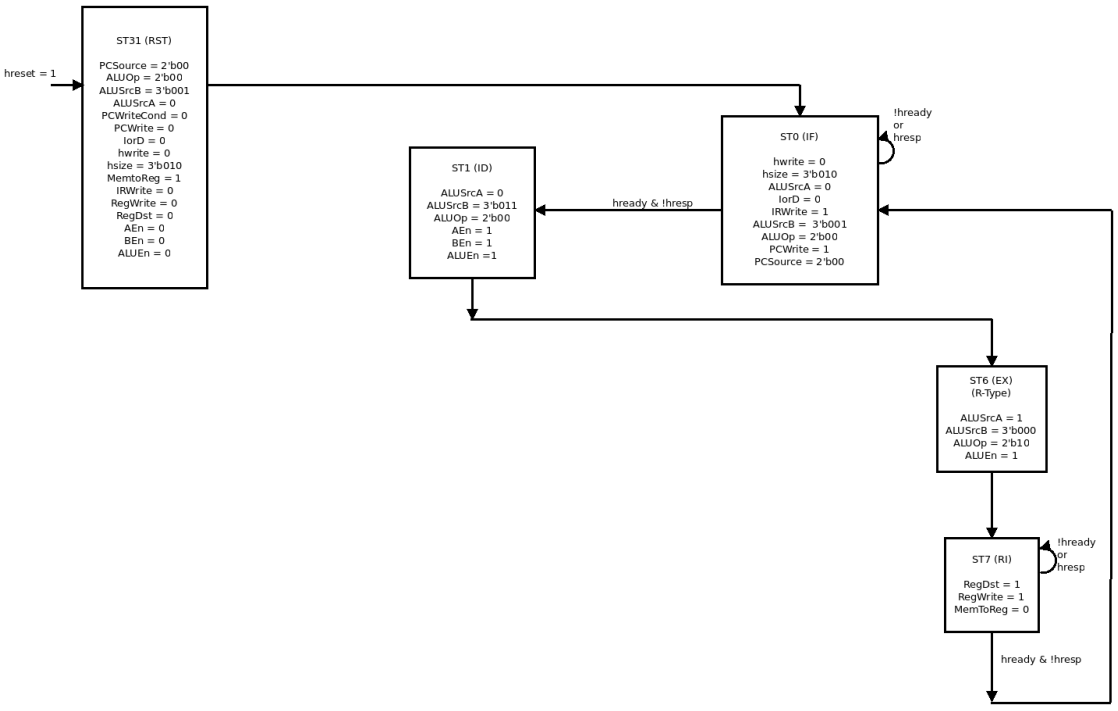


FIGURA 3.27: Estados comunes a todas las instrucciones, reinicio del procesador y estados de instrucciones tipo R.

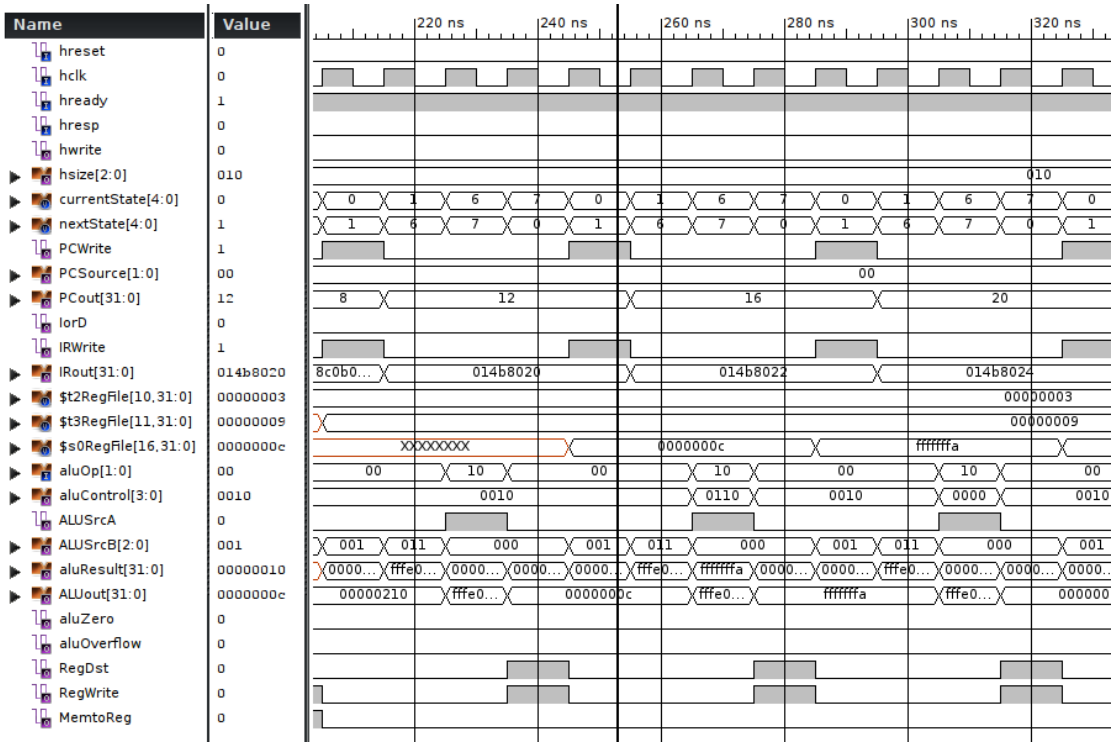


FIGURA 3.28: Simulación de instrucciones tipo R, suma y resta.
El contenido de los registros $\$t2$ y $\$t3$ almacenando el resultado en $\$s0$.

3.1.2.2. Estados de instrucción *Load Word* (LW)

A continuación se describen los estados implicados en la instrucción de cargar una palabra (32 bits) de memoria a un registro del banco y se muestra la constitución de la máquina de estados con la incorporación de éstos en la Fig. 3.29. Suponiendo concluido el estado de búsqueda (IF) y que la máquina se encuentra actualmente en el estado de decodificación, se almacena en el siguiente flanco de reloj el registro que se utiliza como base para calcular la dirección en memoria y el registro destino donde se almacenan los datos en los registros A y B respectivamente. El *opcode* = $(100011)_2$ determina en el estado ID que debe evolucionarse al estado de cálculo de la dirección en memoria ($ST2 - MemAddr$), en el cual se provee a la ALU los operandos del registro base con **ALUSrcA** = 1 y el valor inmediato extendido en signo del *offset* con **ALUSrcB** = $(010)_2$ para efectuar su suma acorde con **ALUOp** = $(00)_2$. Se habilita el registro de la ALU (**ALUEn**) para almacenar el resultado. En el estado siguiente de la máquina ($ST3 - LW$) se manipulan las señales del protocolo para indicar que se trata de una lectura de 32 bits con **hwrite** = 0 y **hsize** = $(010)_2$. El bus de direcciones **haddr** es provisto de la dirección de memoria calculada y almacenada ahora en ALUOut mediante la señal de selección **IorD** = 1. Finalmente, se habilita el bloque de formato de datos de memoria **MDFEn** para que internamente registre los dos bits menos significativos de la dirección y así organizar la información recibida del esclavo en el próximo ciclo de reloj. Dada la comunicación requerida con el esclavo, el protocolo exige que las señales del estado del bus **hready** y **hresp** se encuentren en estado lógico alto y bajo respectivamente para evolucionar al próximo estado. El último ciclo de reloj requerido en esta instrucción involucra al estado ($ST4 - MemComp$), donde finalmente los datos provenientes del esclavo a través del bus **hrdata** se almacenan en el registro destino indicado en la instrucción y que es provisto al banco de registros mediante la selección **RegDst** = 0. Se habilita la escritura del banco según **IRWrite** = 1 y se selecciona la procedencia del dato desde memoria con **MemToReg** = 1. Se plasma el resultado obtenido de simular la ejecución de la instrucción LW en la Fig. 3.30, donde se carga el contenido de la palabra 128 de memoria en el registro $t2 del banco. Se incluyen todas las señales pertinentes a los estados en la simulación y se aprecian los tiempos de espera introducidos durante los estados 3 y 4 mientras la señal de estado del bus **hready** se encuentra en estado lógico bajo.

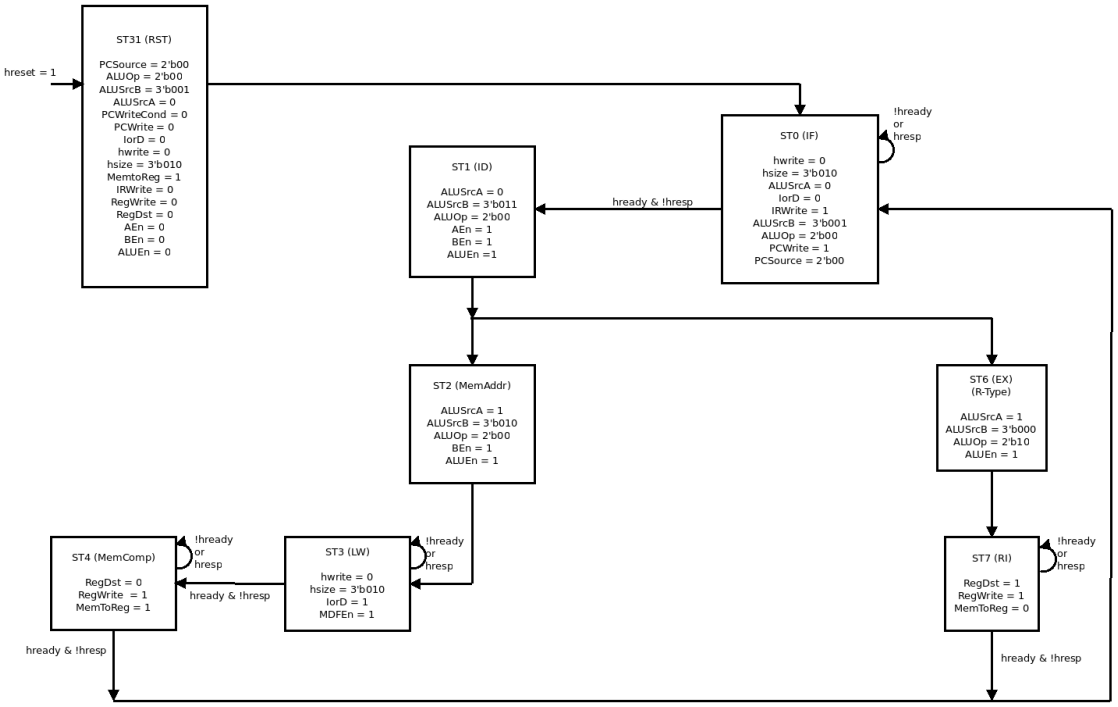


FIGURA 3.29: Máquina de estados con la adición de los estados involucrados en la instrucción LW.

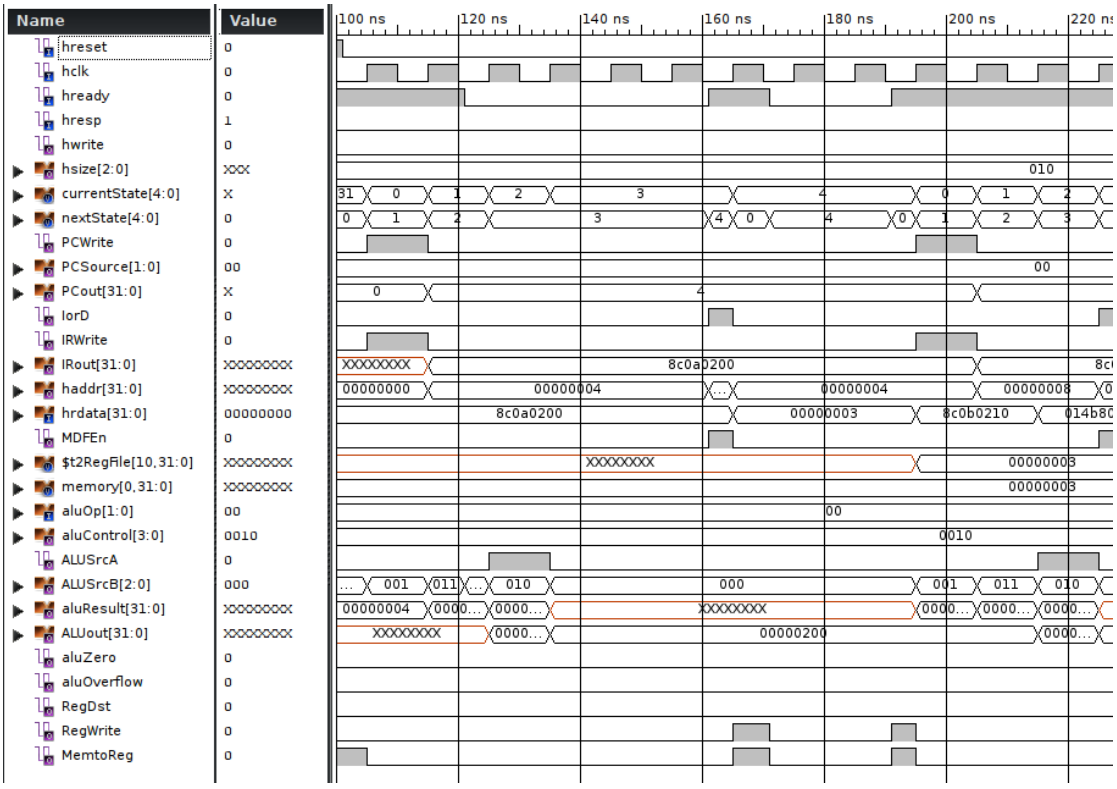


FIGURA 3.30: Simulación de instrucción LW.
Los datos de la palabra 128 en memoria se almacenan en el registro $\$t2$.

3.1.2.3. Estados de instrucción *Store Word* (SW)

Complementariamente a cargar una palabra de memoria, se implementa la instrucción para almacenar en ella una palabra de algún registro del procesador. Debido a que las instrucciones SW y LW son del mismo tipo, se ejecutan utilizando los mismos estados hasta ($ST2 - MemAddr$) (figura 3.31) donde se calcula la dirección de memoria destino para el caso de la instrucción ahora tratada. En el siguiente estado de la ejecución ($ST5 - SW$) se manipulan las señales del protocolo de modo análogo a como se realiza en LW, excepto que para este caso se indica escritura de datos de 32 bits con **hwrite** = 1 y **hsize** = $(010)_2$. La dirección de memoria destino es provista a los esclavos por el bus **haddr** a través de la señal de selección **IorD** = 1. Como el sentido del flujo de datos en una instrucción de almacenamiento es desde el procesador hacia la memoria, no se requiere utilizar el bloque de filtrado de datos. El contenido del registro que se desea almacenar en memoria se alberga en el registro B durante la transición de estados $ID/MemAddr$. Desde éste se provee el dato a guardar en memoria mediante el bus **hwdata** previo a ser adaptado por el multiplexor tratado en la Fig. 3.20 según el ancho de la transferencia. Se evoluciona al siguiente estado cuando las señales **hready** y **hresp** indiquen bus liberado según el protocolo. Finalmente el último estado de la ejecución de la instrucción ($ST10 - Comp$) que es requerido para volver a colocar en el bus **haddr** la dirección de la próxima instrucción apuntada por el PC. Para ello la señal de selección **IorD** se cambia estado lógico bajo al igual que la señal de escritura **hwrite**. Los resultados obtenidos al simular la ejecución de la instrucción se plasman en la Fig. 3.32, donde los datos albergados en el registro $\$t3$ se almacenan en la palabra 133 de memoria. De igual modo que para la instrucción LW se aprecian tiempos de espera introducidos por el esclavo y la evolución de todas las señales de los estados.

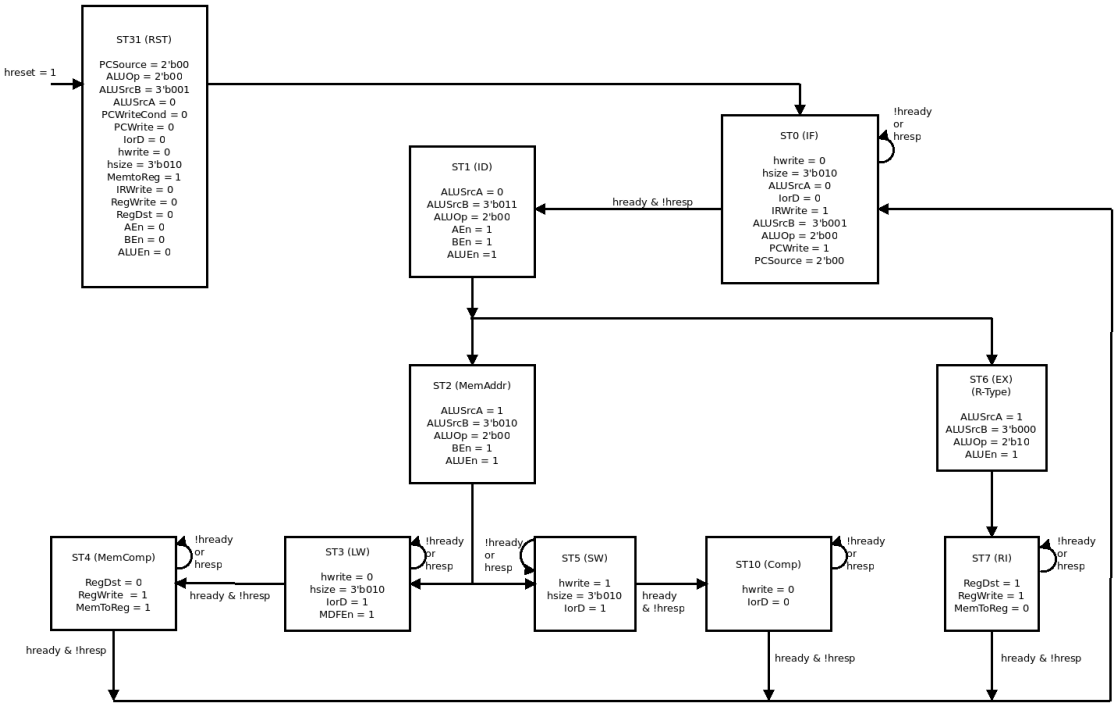


FIGURA 3.31: Máquina de estados con la adición de los estados involucrados en la instrucción SW.

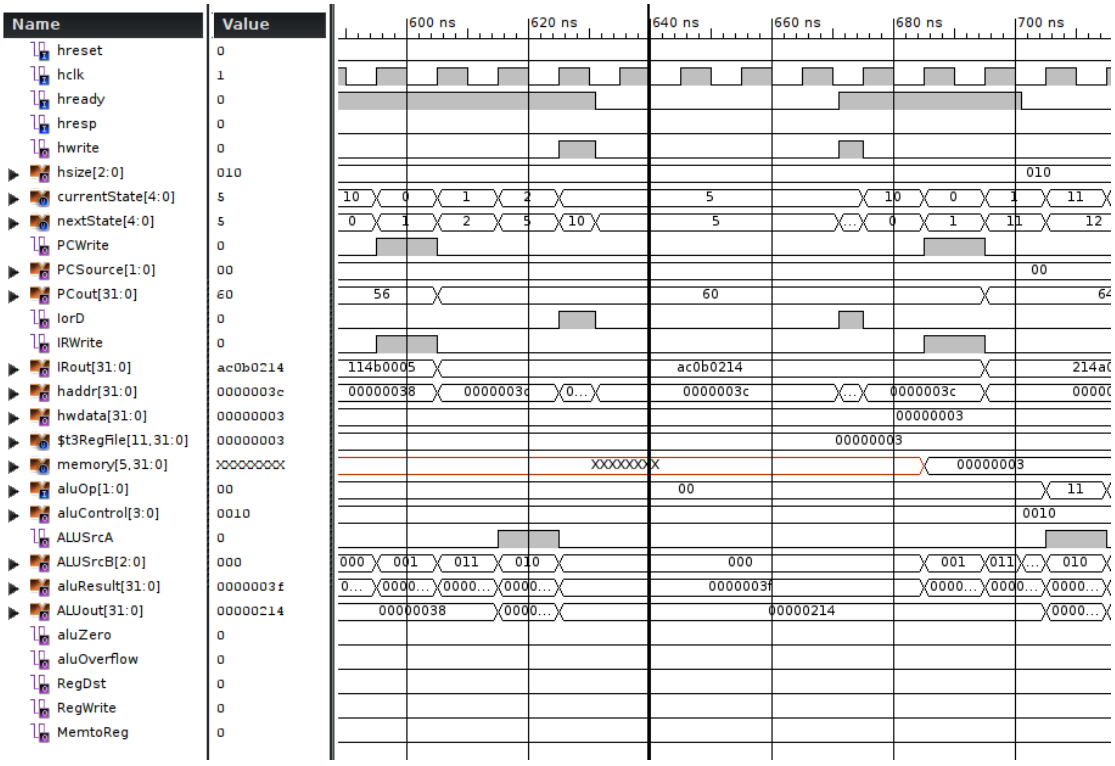


FIGURA 3.32: Simulación de instrucción SW.

Los datos albergados en el registro $\$t3$ se almacenan en la palabra 133 de memoria.

3.1.2.4. Estados de instrucciones de salto y *branch*

Del conjunto de instrucciones de salto y *branch* se implementan tres: BEQ, BNE y J. Las dos primeras son condicionales, produciéndose el salto si la comparación entre dos registros es igual o desigual respectivamente. La tercera es el salto incondicional tratado en secciones anteriores. Partiendo desde el estado de decodificación, el *opcode* = $(000010)_2$ determina si se trata de la instrucción J y la máquina procede al estado $(ST9-J)$ como indica la Fig. 3.33. En éste se habilita la escritura del PC con **PCWrite** = 1, sin controlar ninguna otra condición. Para proveer la dirección del salto que se forma del modo explicado en 3.1.1.6, se coloca la señal de selección **PCSource** = $(10)_2$. El último estado para concluir la ejecución de un salto incondicional es nuevamente $(ST10-Comp)$ donde se provee al bus **haddr** la dirección del PC recientemente actualizada. En la Fig. 3.34 se muestran los resultados de simular una instrucción de salto que altera el flujo del programa para retornar al inicio en la dirección cero.

Para los saltos condicionales se recuerda al lector que la dirección del salto se calcula durante el estado de decodificación. Tanto la instrucción BEQ (*opcode*=$(000100)_2$) como BNE (*opcode*=$(000101)_2$) conducen al estado $(ST8 - Branch)$ y es en éste donde se utiliza la dirección calculada y albergada en ALUOut. Como se indica en la Fig. 3.33 en *Branch* se emplea la ALU para evaluar la condición (igualdad o desigualdad según la instrucción) proveyendo los operandos desde los registros A y B con **ALUSrcA** = 1 y **ALUSrcB** = $(000)_2$. Se señaliza efectuar la resta de los operandos con **ALUOp** = $(01)_2$ y se provee en la entrada de datos del PC la dirección almacenada en ALUOut con **PCSource** = $(01)_2$. Dada la necesidad de cumplirse la condición para modificar el PC, se activa la señal condicional de habilitación **PCWriteCond**. Esta habilitación parcial en conjunto con el la señal de *flag* **aluZero** o **!aluZero** determinan si se produce el salto en caso de tratarse de la instrucción BEQ o BNE respectivamente. Con o sin actualización del PC, el siguiente estado en la ejecución es nuevamente $(ST10-Comp)$. La Fig. 3.35 muestra la simulación de la instrucción BEQ en la cual se actualiza el valor del registro $\$t3$ de modo que sea igual a $\$t2$ y luego se evalúa su igualdad para realizar un salto de 5 lugares en memoria.

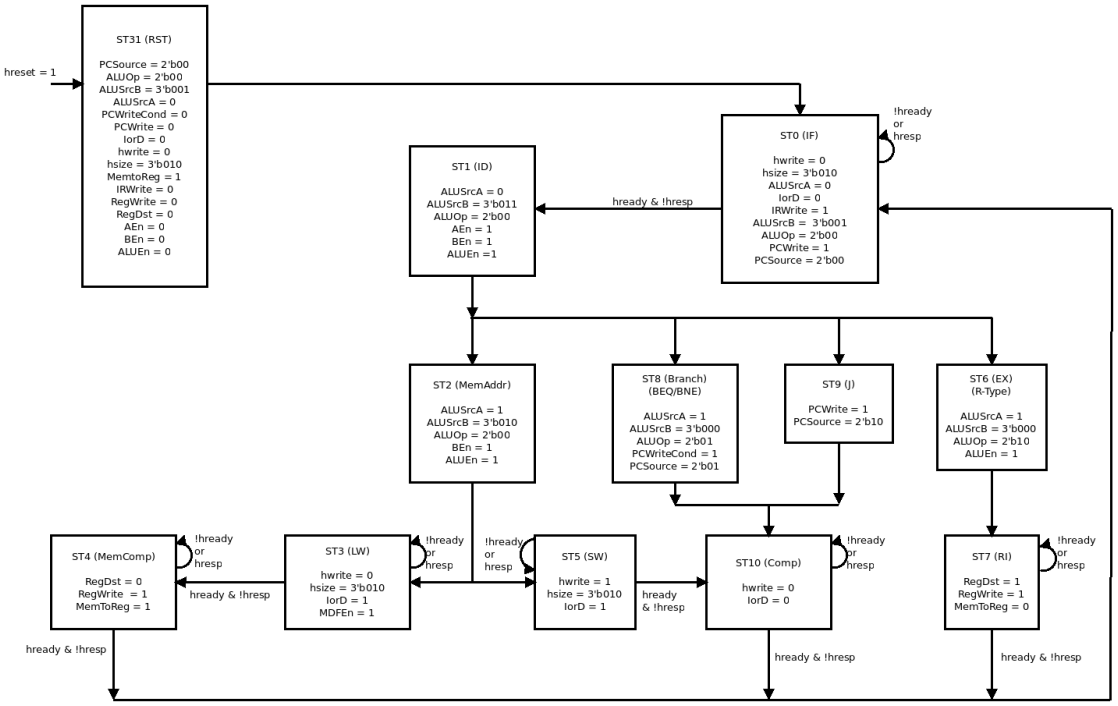


FIGURA 3.33: Máquina de estados con la adición de estados para instrucciones *branch* y saltos.

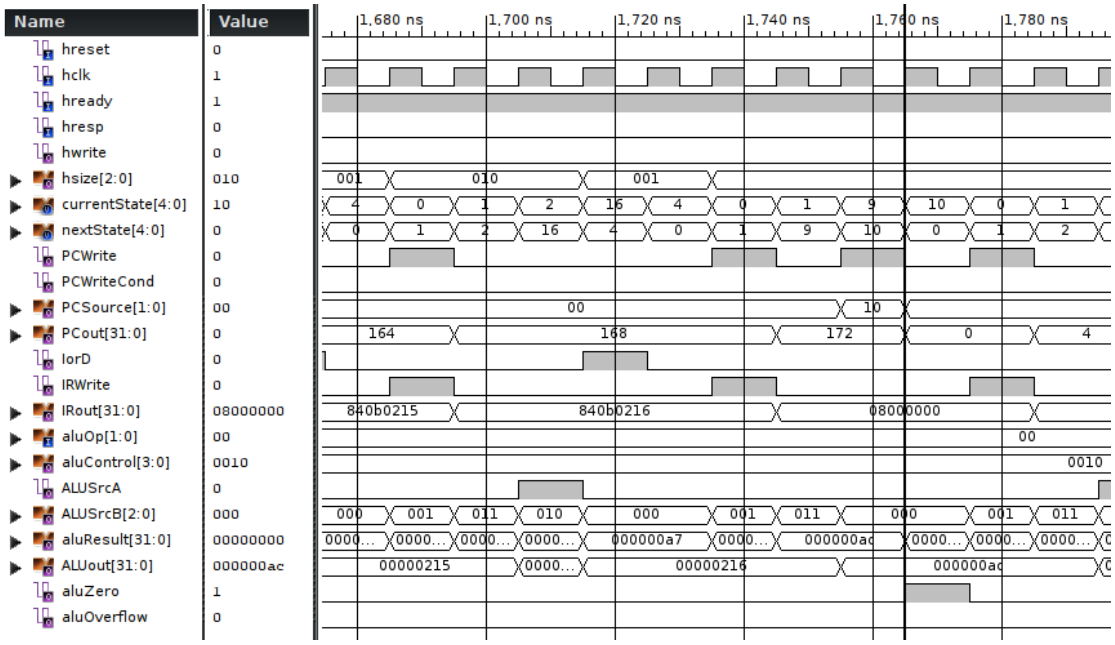


FIGURA 3.34: Simulación de instrucción J.
El flujo de la ejecución de instrucción se altera y se retorna a la dirección inicial en memoria.

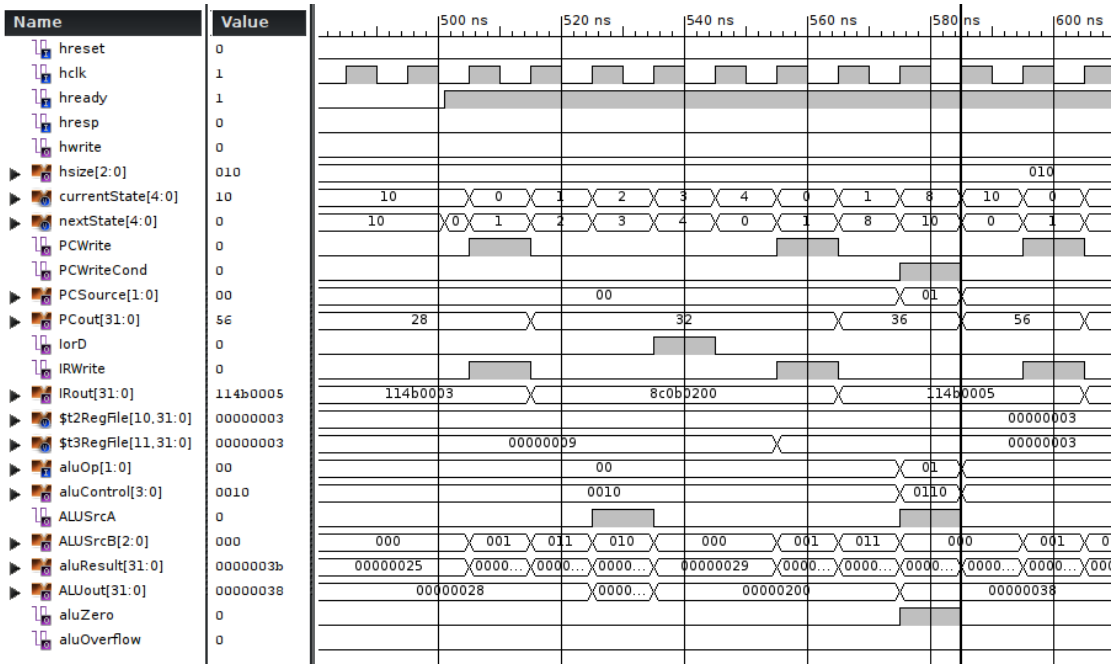


FIGURA 3.35: Simulación de instrucción BEQ.
Se produce un salto de cinco direcciones al comprobar que $\$t2$ y $\$t3$ son iguales.

3.1.2.5. Estados de instrucción *Load Upper Immediate* (LUI)

Con el fin de poseer un medio de almacenar constantes en los registros del procesador se implementa la instrucción LUI. Ésta carga los 16 bits más significativos de un registro con un valor inmediato incluido en la instrucción y, combinada con instrucciones lógicas de inmediatos que se explican posteriormente, brinda la posibilidad de cargar constantes de 32 bits en registros de propósito general. Partiendo nuevamente desde la decodificación, el *opcode* $= (001111)_2$ determina que el siguiente estado se trata de $(ST13 - LUI)$ según la figura 3.36. Se provee como uno de los operandos de la ALU el contenido del registro A con **ALUSrcA** $= 1$, que alberga el valor nulo del registro $\$zero$ acorde con el formato de la instrucción LUI. El otro operando es el campo inmediato de la instrucción desplazado 16 bits, provisto con **ALUSrcB** $= (101)_2$, y se indica a la ALU la suma con inmediato con **ALUOp** $= (11)_2$. Finalmente se habilita el registro de la ALU con **ALUEn** y ante el siguiente flanco del reloj la máquina evoluciona al estado $(ST12 - ImmComp)$. En éste se almacena el valor inmediato desplazado con la señal **RegDst** $= 0$ para indicar el registro destino del banco, habilitando la escritura de éste con **RegWrite** y tomando el dato albergado ahora en ALUOut mediante **MemToReg** $= 0$. La simulación de la ejecución de la instrucción LUI realizada se muestra en la Fig. 3.37 donde se carga en el registro $\$t2$ la constante $(FFFF)_2$.

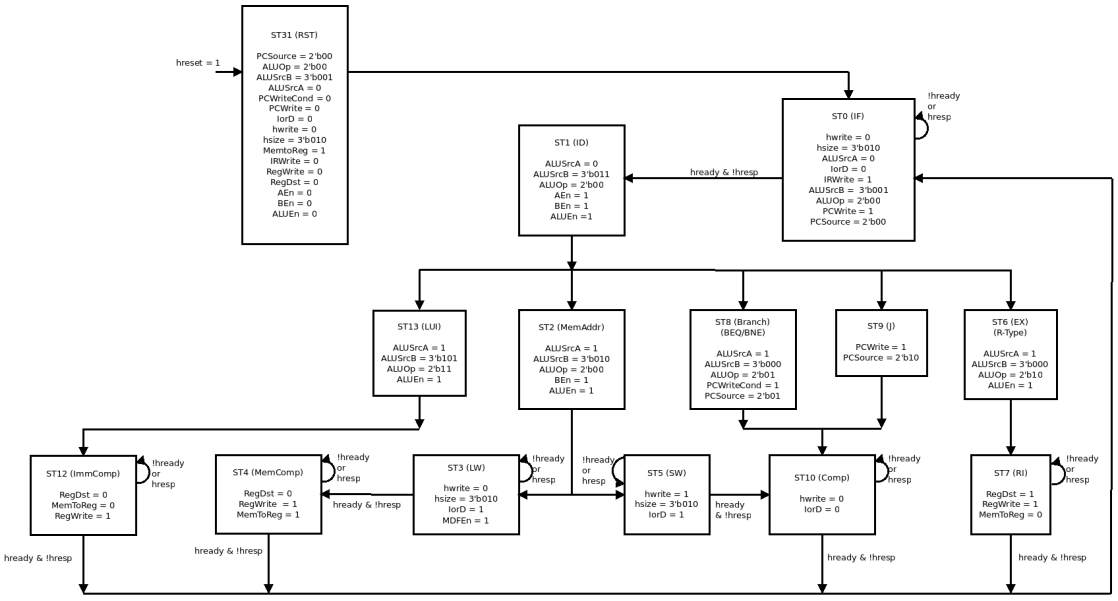


FIGURA 3.36: Máquina de estados con la adición de estados para instrucción LUI.

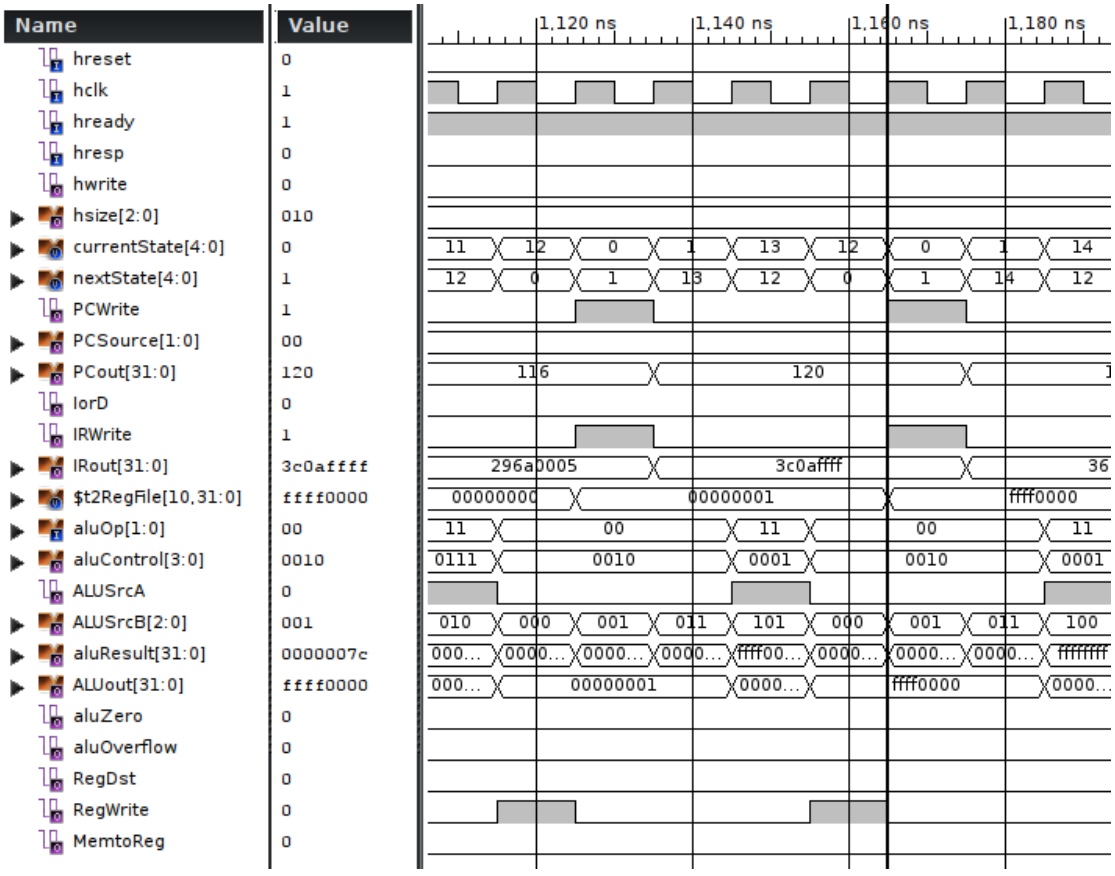


FIGURA 3.37: Simulación de instrucción LUI.
El registro $\$t2$ se carga con la constante $(FFFF)_2$ desplazada 16 bits.

3.1.2.6. Estados de instrucciones aritmético-lógicas con inmediatos

Se implementan también instrucciones para operar aritmética y lógica entre registros de propósito general y valores inmediatos. Si bien las operaciones son análogas a las que se realizan en las instrucciones tipo R, el *opcode* de cada operación con inmediatos es distinto y los respectivos detalles pueden consultarse en las hojas de instrucciones del Apéndice A.

Partiendo del estado de decodificación y en caso de identificarse una instrucción aritmética con inmediato, se procede al estado ($ST11 - ImmArith$) acorde con la Fig. 3.38. Se provee el contenido del registro A (**ALUSrcA** = 1) como operando de la ALU con los datos del registro a operar de la instrucción. El inmediato se provee a la ALU con la señal de selección **ALUSrcB** = $(010)_2$, donde el mismo se extiende en signo para el posterior cálculo. Se indica a la ALU la operación con inmediato según **ALUOp** = $(11)_2$ y el tipo de operación es dictado por el control de la ALU a partir del *opcode*. Se almacena el resultado en el registro de la ALU para lo cual se lo habilita con **ALUEn**. El último estado de la ejecución corresponde a ($ST12 - ImmComp$) nuevamente, y es donde se almacena el resultado en el registro destino especificado en la instrucción.

Si en la decodificación el *opcode* correspondiese a una instrucción lógica con inmediato, se evoluciona al estado ($ST14 - ImmLogic$). De igual modo se provee a la ALU el valor del registro a operar desde el registro A, con la señal **ALUSrcA** = 1. Dado que el dato inmediato es operado lógicamente, no debe extenderse en signo sino que se completa con 16 ceros delante para completar los 32 bits. Se provee dicho dato como segundo operando a la ALU a través de **ALUSrcB** = $(100)_2$ y nuevamente **ALUOp** = $(11)_2$ para indicar operación con inmediato y se habilita ALUOut (**ALUEn**). Se almacena el resultado en el siguiente estado ($ST12 - ImmComp$). La Fig. 3.39 muestra los resultados de simular la operación lógica con inmediato ANDI entre el registro $\$s0 = (FFFFFFFE)_{16}$ y el valor inmediato $2 = (00000002)_{16}$, el resultado se almacena en $\$t2$. Mediante la ejecución consecutiva de las instrucciones LUI y ORI aplicadas sobre el mismo registro destino, puede cargarse en dos pasos una constante de 32 bits en dicho registro desde el código ensamblador. En la primera se proveen los 16 bits más significativos y luego se opera ORI con el inmediato de los 16 bits menos significativos.

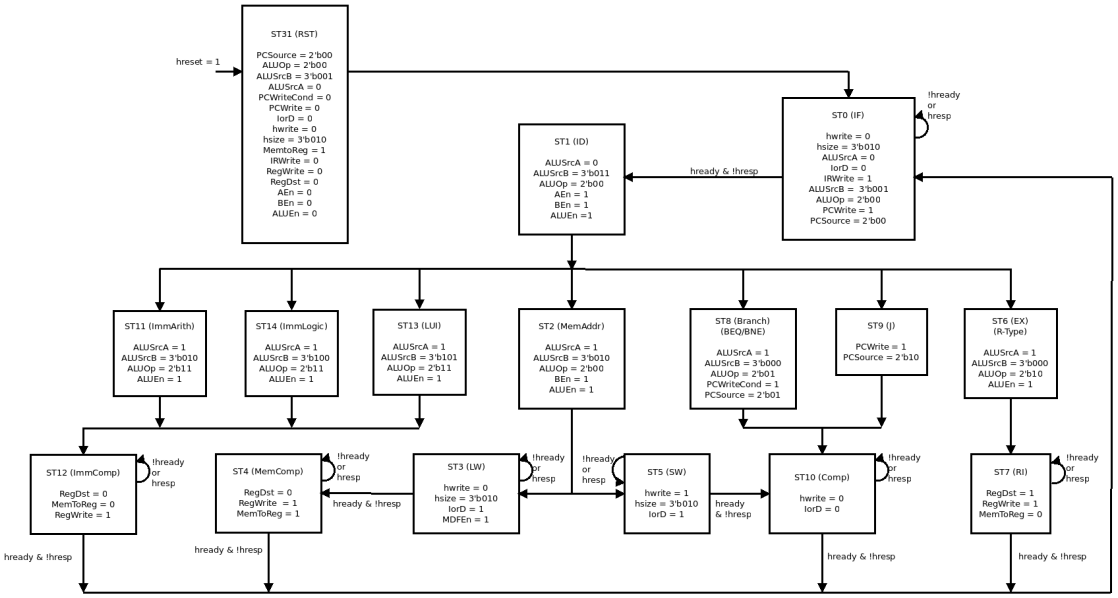


FIGURA 3.38: Máquina de estados con la adición de estados para instrucciones aritmético-lógicas con inmediatos.

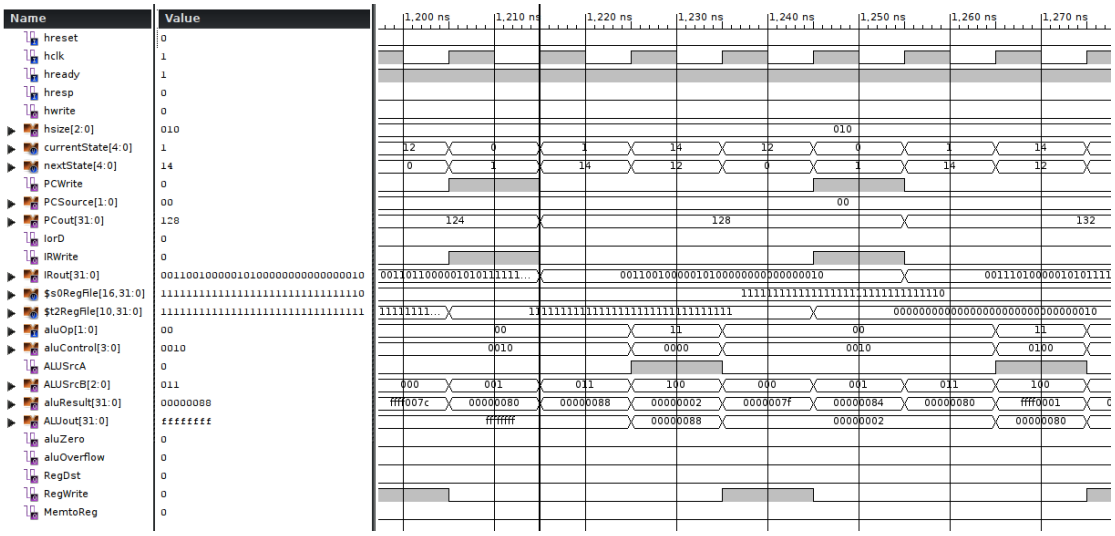


FIGURA 3.39: Simulación de instrucción ANDI.
En el registro $\$t2$ se almacena el resultado de la operación *and* entre el registro $\$s0$ y la constante 2.

3.1.2.7. Estados de instrucciones de carga y almacenamiento de ancho menor a 32 bits

Entre las instrucciones que transfieren datos de ancho menor a 32 bits, desde y hacia la memoria, se implementan: SB (*store byte*) para almacenar 8 bits, SH (*store half word*) para almacenar 16 bits y sus contrapartes LB y LH para cargar desde memoria 8 y 16 bits respectivamente. El esquema de la máquina de estados actualizado en la Fig. 3.40 muestra los nuevos estados involucrados en la ejecución de estas cuatro instrucciones. Dada la similitud entre LB, SH y la ya implementada instrucción LW, los estados implicados y la progresión es idéntica al caso de ésta última. Partiendo desde el cálculo de la dirección en memoria ($ST2 - MemAddr$) se evoluciona a los estados ($ST16 - LH$) o ($ST15 - LB$) según sus respectivos *opcodes*. Como ambos casos son de lectura de la memoria la señal de **hwrite** = 0 y el bus **haddr** se ocupa con el dato de la dirección provisto con la señal de selección **IorD** = 1. Se habilita en cualquiera de los dos estados el Formato de Datos de Memoria con **MDFEn**. La única señal del protocolo que difiere entre ambos es **hsize**, que indica transferencia de 16 bits en LH con **hsize** = $(001)_2$ y de 8 bits en LB con **hsize** = $(000)_2$. Ambas instrucciones completan su ejecución con el estado ($ST4 - MemComp$) en el cual se almacena en el registro del procesador indicado en la instrucción el dato buscado en memoria. De modo análogo la similitud entre SB, SH y la instrucción SW es evidente y se reutilizan estados existentes. La señal de escritura para estas instrucciones es **hwrite** = 1 y nuevamente la dirección calculada se transmite por **haddr** mediante **IorD** = 1. El ancho se determina con la señal **hsize** = $(001)_2$ para SH y el caso de SB con **hsize** = $(000)_2$, ambas instrucciones concluyen la ejecución con el estado ($ST10 - Comp$). Todos estos nuevos estados deben cumplir con el protocolo controlando las señales del estado de bus **hready** y **hresp** debido a que acceden o requieren información de los esclavos. La Fig. 3.41 muestra los resultados de simular las instrucciones LB y SB. Primero se almacena en el registro $\$t3$ el byte 1 de la palabra 133 de memoria (**memory[5]**). La siguiente instrucción almacena en el byte 0 de la misma palabra el contenido del registro $\$t2$.

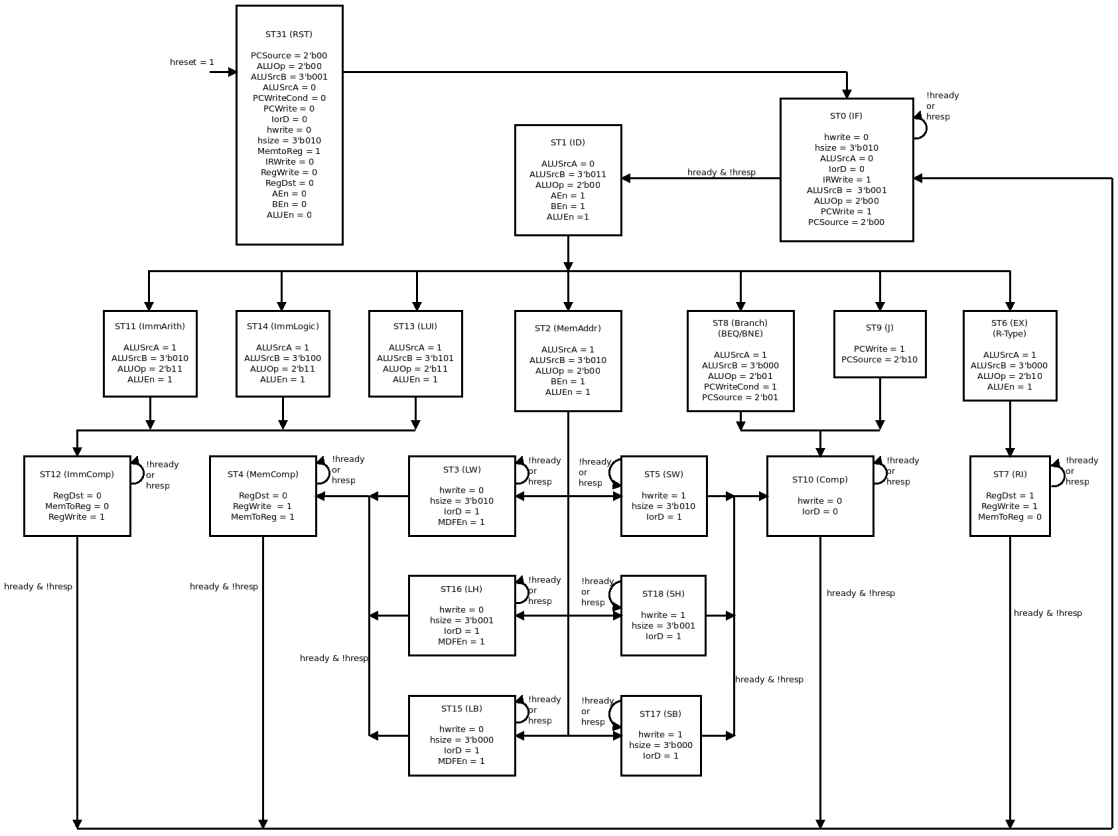


FIGURA 3.40: Máquina de estados con la adición de estados para instrucciones de transferencia de datos de ancho menor a 32 bits.

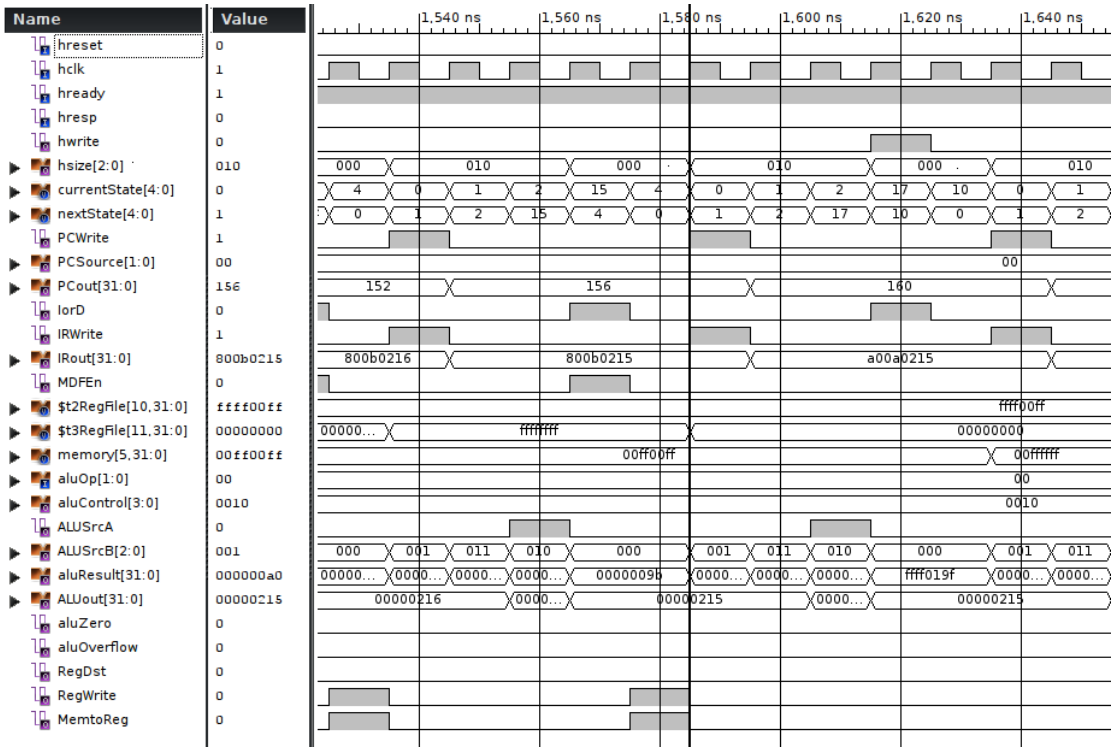


FIGURA 3.41: Simulación de las instrucciones LB y SB.

3.1.2.8. Estados de instrucciones del *watchdog*

Por último, se incorporan estados a la máquina para implementar el *watchdog* como una solución al potencial caso que un esclavo requiera indefinidos tiempos de espera para una transacción y suspenda el programa ejecutado por el procesador. Acorde con lo mostrado en la figura 3.42, se agrega una señal de habilitación **cuentaEnable** a cada estado que verifica las señales de bus del protocolo. Si el esclavo requiere introducir tiempos de espera durante alguno de estos estados (**hready** = 0 o **hresp** = 1), la habilitación se coloca en estado lógico alto y se incrementa la cuenta de ciclos de espera con cada flanco ascendente del reloj. Cuando el conteo supera el valor especificado como límite en el los 31 bits del registro del *watchdog*, se accede al estado ($ST30 - Watchdog$). En éste se habilita la escritura del PC (**PRWrite** = 1) y se le provee la última dirección en memoria (**PCSource** = $(11)_2$). En dicha dirección el programador puede configurar la instrucción a llevar a cabo para salvar la suspensión del procesador acorde a la tarea que éste realiza. Finalmente, en el estado $Watchdog$ se reinicia la cuenta del mismo. Si por el contrario, durante el conteo de tiempos de espera el esclavo finaliza la tarea y la ejecución de la instrucción continua, debe reiniciarse la cuenta una vez accedido el estado siguiente al que introdujo la pausa. Para ello se habilita la señal de reinicio de cuenta en los estados IF, ID, $Comp$ y $MemComp$ como muestra la Fig. 3.42. Se simula la funcionalidad del *watchdog* cargando el registro con $(80000002)_{16}$, habilitando el funcionamiento con el bit 32 en estado alto y en los restantes 31 bits se configura para un máximo de dos tiempos de espera. Como se ve en la Fig. 3.43 se retrasa la ejecución durante el estado LW de una instrucción y al detectarse el tercer tiempo de espera se accede al estado $Watchdog$ actualizando el PC a la dirección 1023 en memoria.

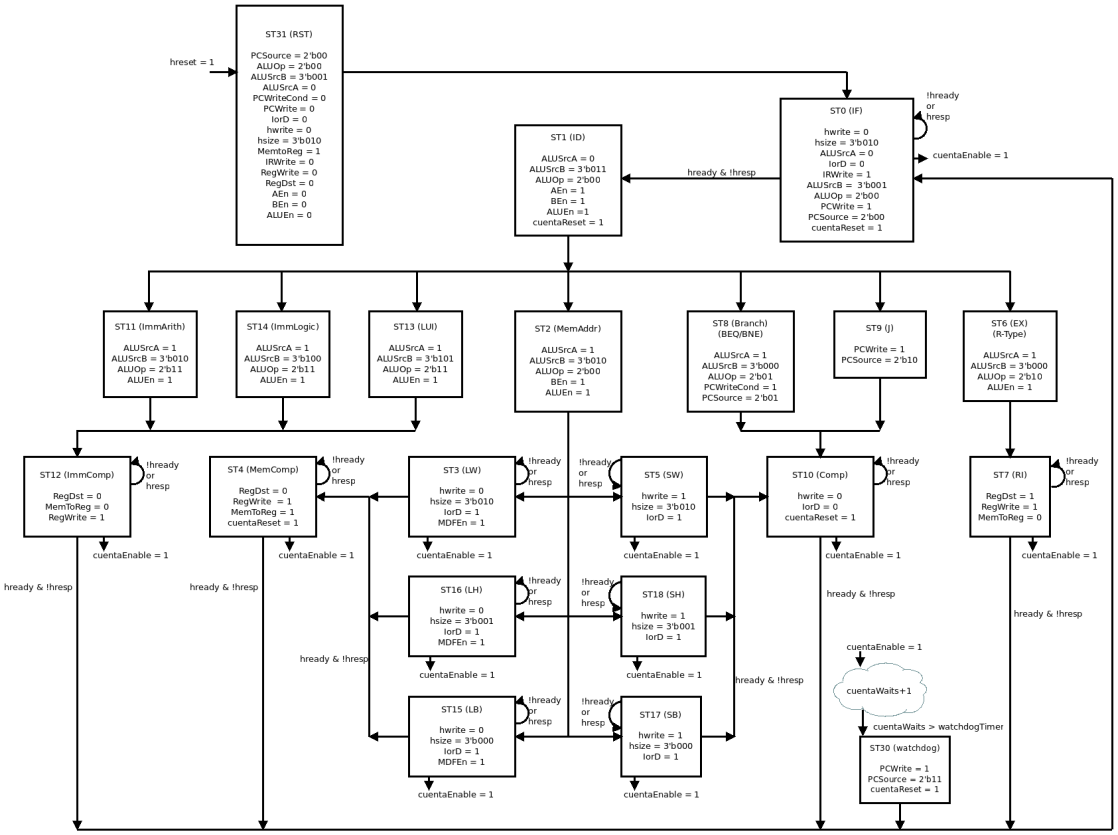


FIGURA 3.42: Máquina de estados con la adición de estados para el funcionamiento del *watchdog*.

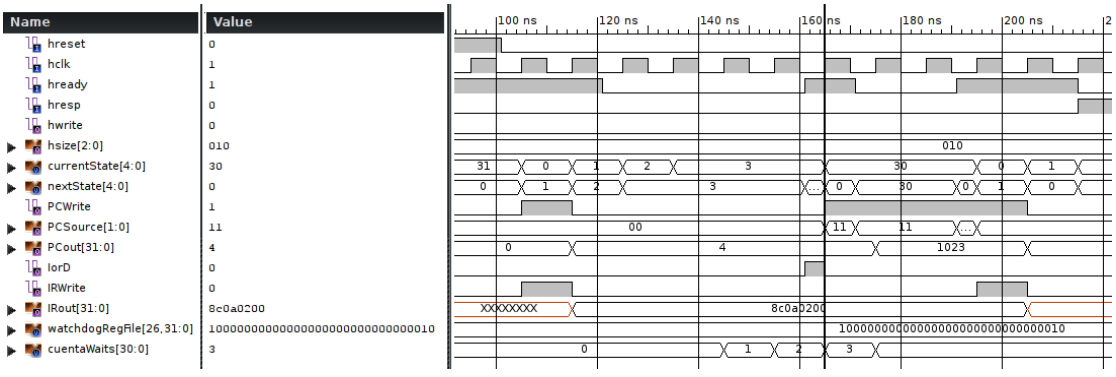


FIGURA 3.43: Simulación de la operación del *watchdog*.
Se modifica el contenido del PC al detectarse el tercer ciclo de espera introducido por un esclavo.

3.1.3. Integración del sistema completo

Se lleva a cabo la integración de todos los bloques diseñados conformando el maestro con sus elementos del *datapath* diferenciados en color negro respecto de los elementos del control de color azul, tal como se ve en la Fig. 3.45. Los restantes bloques del sistema se incluyen en la Fig. 3.44.

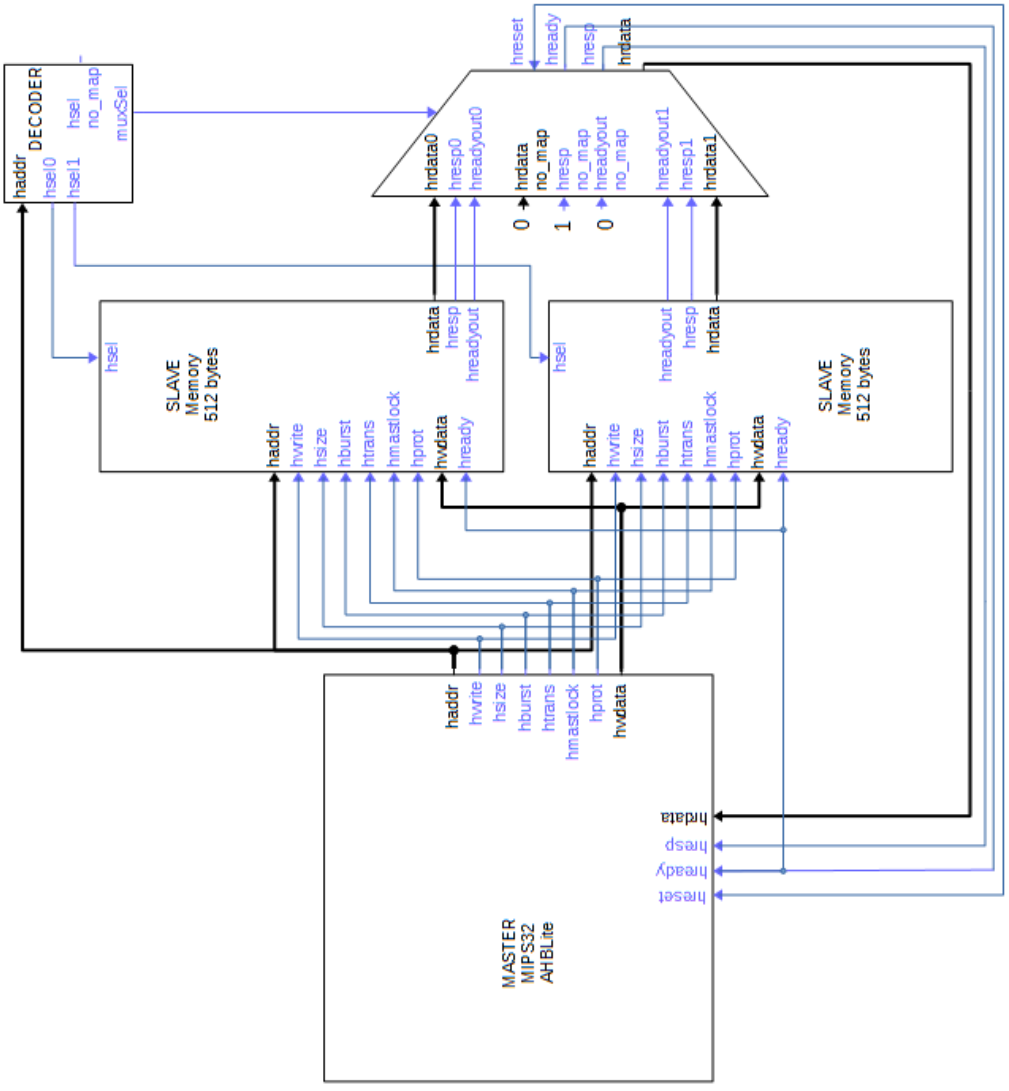


FIGURA 3.44: Esquema en bloques del sistema completo.

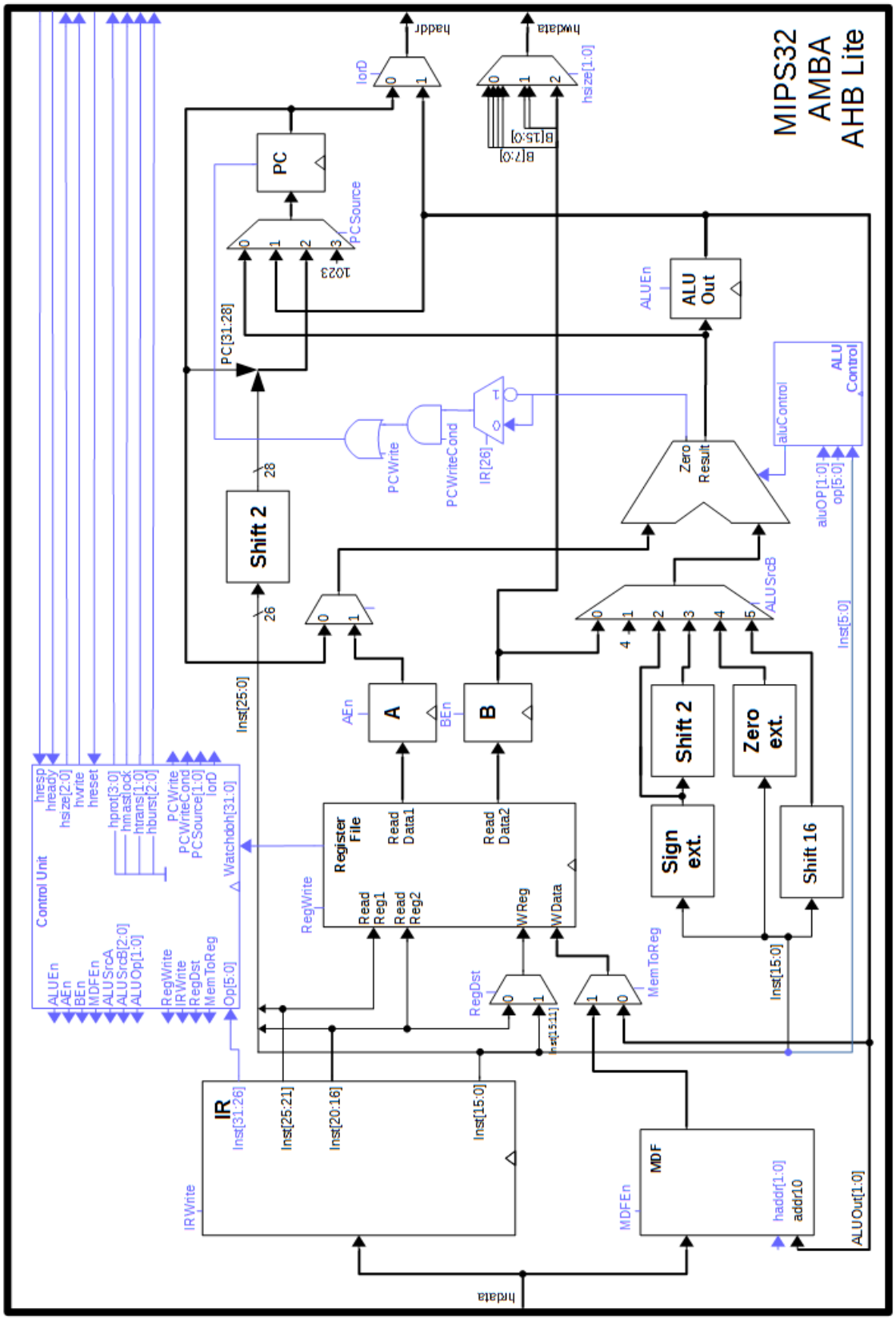


FIGURA 3.45: Esquema en bloques del maestro completo.

Capítulo 4

Simulación, verificación y síntesis

4.1. Verificación y prueba de desempeño

Con el diseño del sistema completo, conectado y simulado el funcionamiento de sus componentes de forma individual, se procede a verificar el comportamiento de todo el conjunto. El objetivo es corroborar que un código de programa cargado en la memoria sea ejecutado con éxito, probando que cada una de las instrucciones implementadas lleve a cabo su función entregando los resultados esperados y asegurando que el protocolo AHB-Lite sea respetado acorde a las especificaciones.

4.1.1. Ensamblador

En términos generales un ensamblador es un programa que toma como entrada instrucciones de computadora básicas y las convierte a su respectivo patrón de bits que el procesador de la computadora puede interpretar para llevar a cabo las operaciones de la instrucción. El programa utilizado como entrada del ensamblador se denomina código fuente, y está escrito en un lenguaje de bajo nivel con una fuerte correspondencia entre el lenguaje y las instrucciones de la arquitectura del procesador para el que se programa. Cada lenguaje ensamblador es exclusivo de su arquitectura. El código de salida del ensamblador es llamado código máquina y es una secuencia de unos y ceros dificilmente interpretable por un programador. En la actualidad la programación en código ensamblador se encuentra mayormente relegada a casos donde se requiere un control eficiente

sobre las operaciones del procesador utilizado. Además, se requiere conocer el conjunto de instrucciones particular del procesador.

Existen dos tipos de ensambladores clasificados según la cantidad de pasadas que éste necesite realizar al código fuente para producir el código máquina. Los ensambladores de una sola pasada poseen la ventaja de ser rápidos, y escriben el código máquina directamente como se almacenará en memoria conforme leen lás líneas del programa. Su principal inconveniente surge cuando encuentran una etiqueta en el código u otro tipo de referencia futura que no se encuentra aún definida. Para evitarlo, debe indicarse previamente a los ensambladores de una sola pasada todas las áreas del código antes de que sean referenciadas. Los ensambladores de pasadas múltiples mitigan este problema generando una tabla con todos los símbolos y sus respectivos valores en una primer pasada al código fuente. Luego utilizan los datos de la misma en las pasadas sucesivas para ensamblar cada instrucción y generar el código máquina.

Para realizar los programas que evalúan el funcionamiento y desempeño del procesador propuesto, se diseña un ensamblador de código MIPS de dos pasadas. El *script* se realiza en Octave y toma como entrada un archivo de texto con el programa codificado en código ensamblador MIPS y devuelve dos archivos de texto con las instrucciones traducidas en binario y hexadecimal respectivamente como ejemplifica la figura 4.1. Durante la primer pasada el ensamblador recorre todas las líneas identificando etiquetas (*labels*) o símbolos con referencias futuras, en caso de encontrarlas guarda el número de línea de la misma junto con su nombre. Si el formato de la etiqueta no corresponde a la sintaxis se interrumpe el ensamblado notificando al programador la línea que contiene el error. En la segunda pasada el ensamblador inicia la interpretación línea a línea, salteando aquellas que se encuentren vacías. Para ello se invoca en cada línea la función de interpretación de instrucción luego de excluir los posibles comentarios del código o etiquetas que ya se identificaron en la primer pasada. La función toma como entrada el contenido de la línea, su número y etiqueta en caso de tenerla. Procede entonces a identificar el mnemónico de la instrucción, del cual determina los bits del *opcode* y la cantidad de parámetros que la misma requiere para extraerlos del resto de la instrucción. En todas las operaciones de transferencia de datos en memoria se separan los parámetros de *offset*, registros base y destino. Para todas las instrucciones de operaciones aritmético-lógicas se utiliza el mismo *opcode* pero se determinan los últimos 11 bits de la instrucción según la operación y se separan los parámetros de los registros a operar. Las operaciones con inmediatos tienen

```
1    LW   $t2, 512($zero) #$t2 <= palabra[128].
2    LW   $t3, 528($zero) #$t3 <= palabra[132].
3    ADD  $s0, $t2, $t3   #$s0 <= $t2+$t3.
```

(a) Instrucciones en ensamblador MIPS.

```
1    10001100000010100000001000000000
2    10001100000010110000001000010000
3    00000001010010111000000000100000
```

(b) Instrucciones traducidas en lenguaje máquina.

FIGURA 4.1: Ejemplo de funcionamiento del ensamblador.

un solo registro como parámetro y el valor inmediato, la función determina los bits el *opcode* de cada una según la operación. En las instrucciones de salto incondicional se determinan los bits del *opcode* y se toma el parámetro de la dirección a saltar, mientras que en los saltos condicionales se toman los de los registros a comparar y el *offset* para la dirección del salto. En cualquiera de los casos mencionados si el mnemónico no es reconocido se produce el correspondiente error de ensamblado. En caso contrario el ensamblador pasa los parámetros tomados a la función que los identifica y convierte a binario. Esta última función controla que los registros a operar sean válidos y no contengan errores de sintáxis. Si algún parámetro es una etiqueta, dirección o valor inmediato, corrobora la existencia de la misma o la validez de los datos. Finalmente se concatenan todos los valores binarios traducidos, *opcode*, registros, *offset* o dirección y se conforma la línea actual en código máquina.

4.1.2. Prueba de funcionamiento

Para los bloques que conforman el procesador y el sistema completo se realizan simulaciones individuales de cada uno mediante entornos de verificación conocidos por su término en inglés *testbenches*, que son corridos en el software Xilinx ISE y su simulador ISim. Estos entornos permiten generar, a modo de estímulos externos que varían en el tiempo, valores asignables a las entradas de los bloques para luego poder observar el funcionamiento interno y las salidas.

Verificar los bloques aislados es relativamente simple, sin embargo no es igual para el caso del sistema completo donde no existen entradas directas que puedan estimularse para probar las operaciones que éste puede desarrollar. Para esto se escribe un programa

en código ensamblador MIPS, mostrado en el Apéndice B, que prueba todas las instrucciones implementadas del procesador y verifica su correcto funcionamiento. En primer lugar se cargan en registros dos valores de 32 bits almacenados en memoria y luego se realizan todas las operaciones aritméticas y lógicas con los mismos. Luego se prueban las instrucciones de *branch* ante igualdad y desigualdad, en casos donde la ejecución del programa debe saltar y también casos en donde la condición de salto no se cumple. Seguidamente se prueban operaciones aritméticas con valores inmediatos, copiar el contenido entre registros, comparar valores mayores o menores entre registros, cargar valores de 32 bits inmediatos y finalmente todas las operaciones de carga y almacenamiento en memoria con todos los anchos de palabra disponibles. El programa se traduce a lenguaje máquina con el ensamblador detallado en la sección anterior y es cargado en la memoria del sistema completo en el *testbench*.

4.1.3. Síntesis

Entre los objetivos del trabajo de esta tesis se propuso que el microprocesador diseñado conforme parte de una biblioteca de bloques IP para implementación en FPGAs. Con el fin de determinar los recursos de hardware que el microprocesador propuesto insume, se procede a realizar la síntesis lógica en una FPGA Spartan-6 XC6SLX45 de Xilinx.

Las FPGAs (sigla del inglés *Field Programmable Gate Array*) son dispositivos semiconductores que incluyen bloques de lógica programable, interfaces de entrada-salida y elementos de ruteo que pueden ser programados. De este modo un determinado diseño de hardware puede implementado en FPGA con la fundamental ventaja de poder actualizarse el diseño reprogramando el dispositivo. La lógica programable utilizada en FPGAs se agrupa en bloques configurables (CLB por su sigla en inglés *Configurable Logic Block*) y contienen elementos como flip-flops y las denominadas LUTs (de su sigla en inglés *Look-Up Table*) que son tablas de múltiples entradas y una salida, con las que pueden implementarse funciones Booleanas. Cada CLB se encuentra rodeado por canales de ruteo que pueden conectarse entre sí mediante *switches* programables. De esta forma se conectan entradas y salidas entre CLBs, para conformar el diseño del hardware a implementar. En el caso de la Spartan-6 XC6SLX45, los CLBs están divididos en dos porciones (llamadas *slices*) que a su vez contienen 4 LUTs (de 6 entradas) cada una y 8 flip-flops. Las LUTs de algunas *slices* pueden ser configuradas como bloques de memoria

RAM distribuida de 64 bits o registros desplazamiento. La FPGA posee en total 6.822 *slices*, 54.576 flip-flops y 401 Kb de memoria distribuida. También posee capacidades de procesamiento digital de señales y bloques de memoria RAM (2088 Kb en total). Dado el diseño RTL descripto en Verilog, su síntesis lógica posee dos principales etapas: la optimización lógica y el mapeo. La primera transforma los circuitos en nivel compuerta a representaciones acordes a la tecnología disponible en la FPGA y puede optimizar el diseño en pos de reducir la cantidad de recursos a utilizar o a incrementar la frecuencia de operación. El mapeo transforma el diseño en nivel de compuertas lógicas en un circuito implementado en la lógica programable de la FPGA.

Con fines de establecer una comparación de requerimiento de recursos de implementación entre el procesador propuesto y otros de arquitectura RISC, se incluyen los datos de síntesis obtenidos de un trabajo evaluativo realizado en el Centro de Micro y Nanoelectrónica del Bicentenario del INTI con sede Bahía Blanca. Los resultados de síntesis se muestran en la tabla 4.1, todos los procesadores fueron sintetizados dos veces en la Spartan-6 XC6SLX45, con optimización lógica apuntada a reducción de recursos (AM) y optimización apuntada a máxima frecuencia de operación (AV). El valor porcentual que acompaña cada dato de síntesis refleja el porcentaje de recursos utilizado respecto del total. Se incluye también la estimación de la máxima frecuencia operativa alcanzable, provista por la herramienta de síntesis. Las principales características de las arquitecturas de los procesadores utilizados para la comparación se detallan a continuación.

ARM Cortex-M0

Es un procesador RISC de 32 bits implementado como arquitectura Von Neumann de ARMv6-M, siendo el más limitado de la familia de procesadores Cortex M0. Posee un *pipeline* de tres etapas y utiliza el protocolo de bus AMBA 3 AHB-Lite.

OpenRISC1200

Este procesador es una implementación de la familia OpenRISC1000. Su arquitectura es Harvard, RISC de 32 bits. Posee cinco etapas de *pipeline*, soporte para memoria virtual y capacidades de procesamiento digital de señales. Sus memorias de cache de datos e instrucciones son de 8KB, mapeo directo y palabra de 16 bytes. Provee bloques

adicionales para *debugging*, control de interrupciones programable, temporizador de alta resolución y soporte para administración de consumo energético. Utiliza el protocolo de bus Wishbone.

ZPU Medium

Este procesador de 32 bits esta diseñado como una máquina de pila (en inglés *stack machine*) apuntado a implementarse en sistemas con FPGA, y se destaca en su bajo requerimiento de recursos. Como máquina de pila, almacena todos los cálculos que realiza en una pila en lugar de utilizar registros. Posee una interfaz de memoria de un solo puerto, de manera que todos los medios (datos, instrucciones, IO) acceden por dicha interfaz. Provee soporte para el protocolo de bus Wishbone.

LEON3

Este procesador fue desarrollado inicialmente por la ESTEC (*European Space Research and Technology Centre*) y luego por Gaisler Research. Su arquitectura RISC está basada en SPARC-V8, de 32 bits, Harvard. Posee *pipeline* de siete etapas. Utiliza protocolo de bus AMBA-2.0 AHB. Su modelo VHDL está apuntado a diseños system-on-chip y es altamente configurable mediante parametrizaciones.

4.1.4. Prueba de desempeño

Como prueba de desempeño del procesador propuesto y con el fin de establecer una comparación con el resto de los procesadores sintetizados, se propone realizar el ordenamiento ascendente de un arreglo de cien elementos mediante la técnica conocida en inglés como *bubblesort* (ordenamiento por burbujeo). El algoritmo empleado revisa cada

	Procesador diseñado		ARM Cortex-M0		OpenRISC 1200		ZPU Medium		LEON3	
	AV	AM	AV	AM	AV	AM	AV	AM	AV	AM
Registros	1235 (2,26 %)	1225 (2,24 %)	850 (1,55 %)	841 (1,54 %)	1883 (3,45 %)	1851 (3,39 %)	360 (0,65 %)	206 (0,37 %)	1121 (2,05 %)	1037 (1,9 %)
LUTs	1630 (5,97 %)	1586 (5.81 %)	3475 (12,73 %)	3072 (11,25 %)	4650 (17,04 %)	4617 (16,91 %)	1091 (3,99 %)	1069 (3,91 %)	3529 (12,93 %)	2908 (10,65 %)
Frecuencia máx. [MHz]	79,88	68,34	53,4	35,5	114,34	113,83	162,52	162,5	97,63	61,96

Tabla 4.1: Tabla comparativa de los resultados de la síntesis de los procesadores optimizada según alta velocidad (AV) o área mínima (AM).

elemento del arreglo a ordenar comparándolo con el siguiente, si el orden entre ellos no es el adecuado los intercambia de posición. Se procede con el siguiente par de elementos en el arreglo y se revisa el mismo tantas veces como sea necesario hasta recorrer el arreglo completo sin producir ningún intercambio. Como resultado, el arreglo se encuentra finalmente ordenado de manera ascendente.

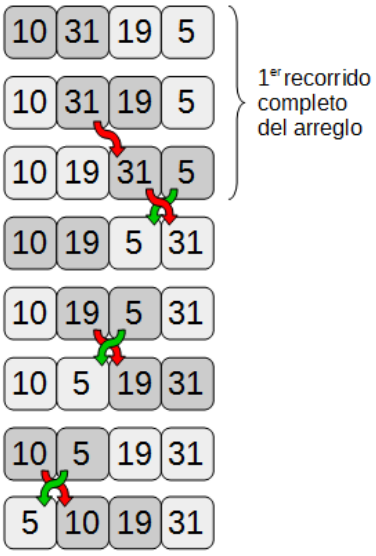


FIGURA 4.2: Ejemplo ordenamiento por burbujeo.

La posición de los elementos en el ordenamiento juegan un papel muy importante en la determinación del rendimiento. Como ejemplifica la figura 4.2, los elementos mayores al principio del arreglo son rápidamente trasladados hacia el final, mientras los elementos menores en el fondo de la lista se mueven a la parte superior muy lentamente. Considerando el peor caso como un arreglo de n elementos que inicialmente se encuentra en orden descendente, se necesita realizar $n-1$ intercambios del primer elemento para ubicarlo al final del arreglo. Seguidamente $n-2$ intercambios para ubicar el segundo elemento en el ante último lugar, $n-3$ para el tercero, etc. De este modo el número de intercambios resulta en la función de la ecuación 4.1, determinando que el orden de complejidad del algoritmo para el peor caso es cuadrático respecto a la longitud del arreglo a ordenar, $\Theta(i(n)) = n^2$.

$$i(n) = (n-1) + (n-2) + (n-3) + ... + 1 = \frac{n^2 - n}{2} \tag{4.1}$$

Por otra parte, se considera el mejor caso cuando el arreglo ya se encuentra ordenado. Aquí no se producen intercambios pero es necesario que el arreglo sea recorrido al menos

una vez para determinarlo, resultando en un orden de complejidad $\Theta(i(n)) = n$. Se considera apropiado este tipo de ordenamiento para la prueba de desempeño debido a su simple implementación y a que permite evaluar la mayoría de las instrucciones implementadas del procesador propuesto.

El código del algoritmo de ordenamiento por burbujeo en lenguaje ensamblador MIPS puede verse en el Apéndice B, junto con los respectivos comentarios en cada línea que clarifican su comprensión. El mismo arreglo desordenado es empleado para la prueba de desempeño de cada procesador mencionado en la Subsección 4.1.3. Los resultados obtenidos luego de correr el algoritmo en cada uno se muestran en la tabla 4.2.

	Procesador diseñado		ARM Cortex-M0		OpenRISC 1200		ZPU Medium		LEON3	
	AV	AM	AV	AM	AV	AM	AV	AM	AV	AM
Ciclos de reloj	206.965		153.080		199.520		2.161.318		98.691	
Tiempo de ejecución [ms]	2,59	3,03	2,87	4.31	1,74	1,75	13,3	13,3	1,01	1,59

Tabla 4.2: Comparación de desempeño en ordenamiento con *bubblesort* de los procesadores según ciclos de reloj y tiempo de ejecución.

4.1.5. Resultados

Se observa de las tablas 4.1 y 4.2 que la síntesis apuntada a optimizar en velocidad de operación arroja mejores resultados. Los recursos requeridos por cada procesador no son significativamente mayores a los indicados por la síntesis apuntada a optimización de recursos y, excepto en Open-RISC1200 y ZPU, la máxima frecuencia alcanzable por los procesadores es mayor. El microprocesador propuesto se encuentra en segundo lugar en requerir la menor cantidad de recursos de hardware de la FPGA, luego de ZPU. La máxima frecuencia de operación estimada lo coloca en cuarto lugar, con una frecuencia cercana a la alcanzada por LEON3.

En cuanto a los resultados de desempeño se destaca que el procesador ZPU, a pesar de resultar el de mayor frecuencia de operación, requiere una cantidad de ciclos y tiempo de ejecución sustancialmente mayor a cualquier otro procesador como consecuencia de realizar todas las operaciones con datos almacenados en una pila. La cantidad de ciclos necesarios para el ordenamiento por el procesador propuesto es comparable con los requeridos por Cortex-M0 y Open-RISC1200, pero ampliamente superado por LEON3. Este último también arroja el menor tiempo de ejecución para la tarea, seguido por Open-RISC1200 que requiere aproximadamente el doble y en tercer lugar el procesador propuesto.

Capítulo 5

Conclusión

En esta tesis se presentó el diseño de un microprocesador RISC, basado en la arquitectura MIPS32 Multiciclo que implementa un subconjunto de las instrucciones de la arquitectura, y que se adaptó para cumplir con las especificaciones del protocolo de comunicación AMBA AHB-Lite. El microprocesador constituye el maestro del sistema, que emplea como esclavos dos módulos de memoria para el almacenamiento de datos y programa del procesador, un decodificador de direcciones de datos y un multiplexor. Se realizó la verificación de funcionamiento de todos los elementos del sistema y se obtuvieron resultados de sus requerimientos y desempeño de procesamiento para su implementación en FPGA Spartan-6 XC6SLX45.

La elección de la arquitectura MIPS32 se basó en que es uno de los primeros procesadores RISC en ser exitosamente empleado en diversidad de sistemas e incluso vigente en la actualidad. La arquitectura fue diseñada específicamente con fines educacionales, resultando en un diseño RISC en su concepto más puro y que sienta las bases para aprendizaje de arquitecturas más complejas. También dispone de abundante bibliografía e implementaciones que facilitaron su estudio. Se seleccionó el protocolo de comunicación AMBA AHB-Lite por tratase de un estándar abierto de especificaciones para la interconexión y manejo de bloques funcionales en diseño de sistemas. Es uno de los protocolos más ampliamente adoptados por la industria de semiconductores y como consecuencia existe un vasto conjunto de productos IP y herramientas que son compatibles con el mismo.

El flujo del diseño consistió en definir inicialmente las funcionalidades que el microprocesador debe realizar y que consecuentemente derivó en la elección de las instrucciones

del conjunto de MIPS32 que se implementaron. Se planteó como metodología dividir cada bloque funcional en subloques más simples, uniformando la complejidad de cada etapa de diseño y permitiendo un progreso organizado. A partir de las instrucciones seleccionadas y los tipos de datos involucrados, se confeccionó el *datapath* de modo que los datos cuenten con medios físicos para su almacenamiento y transmisión, desde y hacia, el maestro y los esclavos del sistema. Luego, a partir de los elementos del *datapath* y el orden de ejecución de las instrucciones tal como especifica la arquitectura MIPS32 Multiciclo y el protocolo AMBA AHB-Lite, se diseñó la unidad de control que administra el manejo de habilitaciones de bloques, selecciones de caminos para el flujo de datos y sincronización de transacciones. Se incorporó la implementación de un *watchdog* configurable, de manera el programador pueda especificar el conjunto de tareas que el microprocesador debe realizar en caso que el bus sea tomado por un tiempo mayor al admisible. Posteriormente a la descripción de cada bloque del diseño se realizó su verificación funcional mediante *testbenches* que simularon su operación. Este procedimiento garantizó que durante la integración de los bloques diseñados, la complejidad de verificación no se viera sustancialmente incrementada ya que cada subloque estaba individualmente verificado. También se simplificó el proceso de depuración de errores en las jerarquías superiores, debido a que se identificó rápidamente en qué subloques se produjeron.

Acorde con los objetivos puntuales propuestos en esta tesis, se diseñó un microprocesador RISC MIPS32 Multiciclo, con funcionalidades esenciales de procesamiento y compatible con el protocolo de comunicación AMBA AHB-Lite. Su diseño descripto en Verilog le otorga reusabilidad, la capacidad de ampliar sus funcionalidades y lo convierte en un bloque de procesamiento viable para incorporar en un sistema a implementar en FPGA. Los resultados comparativos comprueban que se alcanzó una relación de compromiso satisfactoria entre un diseño de bajo insumo de recursos y capacidad de procesamiento. Consecuentemente se evidencia que si se tiene el conocimiento suficiente de la aplicación específica para la que se diseña, crear un diseño propio y adecuado a dichas especificaciones arroja mejores resultados y brinda beneficios en términos de costos frente a adquirir un diseño de propósito general. Además, con la disponibilidad de dispositivos como FPGAs para la implementación de circuitos y sistemas, se puede tener un control más profundo del diseño gracias a la posibilidad de mejorarlo y optimizarlo, dentro de las capacidades del dispositivo, sin la necesidad de refabricación.

Apéndice A

Arquitectura de computadoras

Este apéndice contiene un breve repaso del concepto de arquitectura de computadoras, conjunto de instrucciones y el método de pipeline.

A.1. Arquitecturas de computadoras

A.1.1. Introducción

La arquitectura de una computadora se relaciona con el punto de vista o perspectiva del programador. Está definida por su conjunto de instrucciones y un conjunto de datos almacenados. En la actualidad se cuenta con variedades de arquitecturas como son x86, MIPS, SPARC, entre otras. Las palabras manipuladas por un lenguaje de computadora son las instrucciones, y todos los programas que corren en ella utilizan el mismo conjunto de instrucciones. Cualquiera sea la aplicación de software, cada tarea y operación se compila en una serie de instrucciones elementales como suma, resta, saltos y operaciones lógicas. Cada instrucción define la operación que debe realizarse así como también los operandos involucrados, que pueden estar almacenados en memorias, registros o estar incluidos directamente en la instrucción.
El hardware de la computadora conforma un sistema digital que solo opera con valores lógicos 1 y 0, es decir que la información de instrucciones deben codificarse en formato binario que comúnmente es referido como lenguaje máquina. Debido a lo tedioso que puede ser para el ser humano la representación de las instrucciones en dicho lenguaje, se utiliza un lenguaje simbólico denominado ensamblador. A diferencia de la primera impresión que se tiene del concepto de arquitectura, ésta no define el hardware subyacente. Existe, de hecho, distintas implementaciones para una sola arquitectura como es el caso de las diseñadas por Intel y AMD ambas del tipo x86. El diseño en hardware responde enteramente a las necesidades de la aplicación, dando lugar a microprocesadores optimizados para alto rendimiento y otros para bajo consumo en el caso de las computadoras portátiles. La forma en que se dispone de las memorias, registros, procesadores y todos los componentes del sistema es lo que define la microarquitectura.

A.1.2. Von Neumann

La arquitectura de Von Neumann, llamada así en honor a su propulsor, describe un diseño de arquitectura de computadoras digitales cuya característica principal es el uso compartido de la memoria para almacenamiento de instrucciones y datos como esquematiza la Fig. A.1. Para el año en que fue propuesta, su concepto era revolucionario ya que los programas de computadora eran en si mismos parte de la máquina, separada de

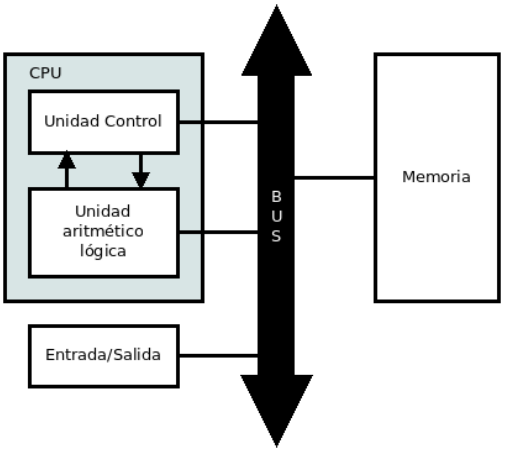


FIGURA A.1: Diagrama en bloques de arquitectura Von Neumann.

los datos que ésta procesaba. El enfoque de Von Neumann propone un almacenamiento del programa, permitiendo ser modificado dentro de las posibilidades de la computadora y convirtiéndola en una máquina de múltiples propósitos. Existen otras características sugeridas en la arquitectura relacionadas con la inclusión de una unidad aritmético-lógica, una unidad de control, memoria, medios de entrada y salida de datos y un bus de comunicación para interconectar todos los elementos mencionados. La máquina funciona operando los siguientes pasos en secuencia:

1. Búsqueda en memoria de la siguiente instrucción a ejecutar desde un contador de programa.
2. Decodificación de la instrucción por parte de la unidad de control.
3. Ejecución de la instrucción, comandada por la unidad de control, efectuada por la unidad aritmético-lógica. Actualización del contador del programa para continuar operación.
4. Vuelta al primer paso.

Debido al uso compartido del bus de comunicación entre CPU y memoria para datos e instrucciones, no puede ocurrir al mismo tiempo un acceso a ésta por datos y por una instrucción. Esto puede impactar en el desempeño de la máquina ya que las instrucciones solo pueden llevarse a cabo en forma secuencial, produciendo accesos repetidos a memoria. Sin embargo, se pueden mitigar dichos efectos utilizando técnicas de paralelismo que se cubren más adelante en este texto.

A.1.3. Harvard

Otro arquetipo popular es la arquitectura Harvard, cuyo nombre toma origen de la computadora Hardvard Mark I que tenía sus instrucciones y sus datos almacenados en medios de naturaleza diferente y por ende separados. Dado que en esta arquitectura no se comparte el bus para datos e instrucciones, las características de las memorias empleadas para cada tipo no necesariamente son iguales. Pueden diferir en el ancho de palabra, la tecnología de implementación, la estructura de direcciones y en los tiempos de acceso. Además, el CPU puede simultáneamente leer una instrucción de memoria mientras lleva a cabo un acceso a memoria por datos, permitiendo que esta arquitectura sea más veloz para una determinada complejidad circuital. El esquema básico de la arquitectura Harvard se ve en la Fig. A.2. En los CPU actuales de alto desempeño conviven aspectos de ambas arquitecturas, en donde la memoria cache es la que se divide en datos e instrucciones ofreciendo las ventajas de paralelismo, pero en caso de no hallarse en cache el procesador debe extender su búsqueda a la memoria donde residen datos e instrucciones juntas. La principal ventaja de acceso simultáneo ofrecido por la arquitectura Harvard se ve reducida frente a la implementación mixta mencionada y esquematizada en la Fig. A.3.

Se observan en la práctica arquitecturas puramente Harvard en aplicaciones específicas como las que realizan los procesadores digitales de señales (DSP de su sigla en inglés). Éstos ejecutan algoritmos para procesamiento de audio y/o video que suelen ser pequeños y optimizados, utilizan múltiples buses de datos (uno de instrucciones, y para datos pueden incluso tener más de uno para lecturas y escrituras). En microcontroladores también se aprovecha la ventaja de acceso simultáneo ya que las memorias empleadas para datos e instrucciones son pequeñas, y no utilizan cache.

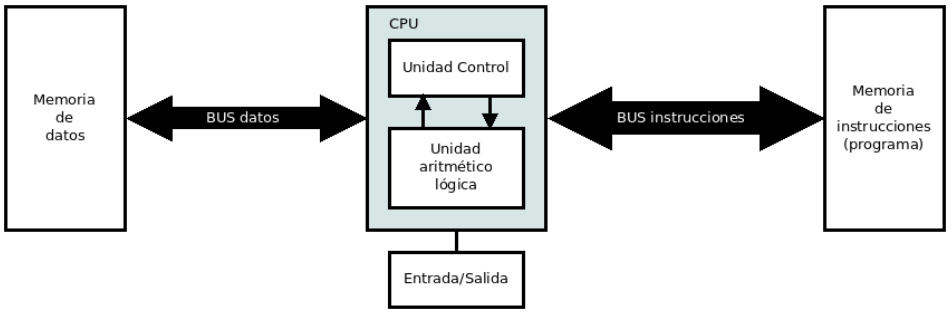


FIGURA A.2: Diagrama en bloques de arquitectura Harvard.

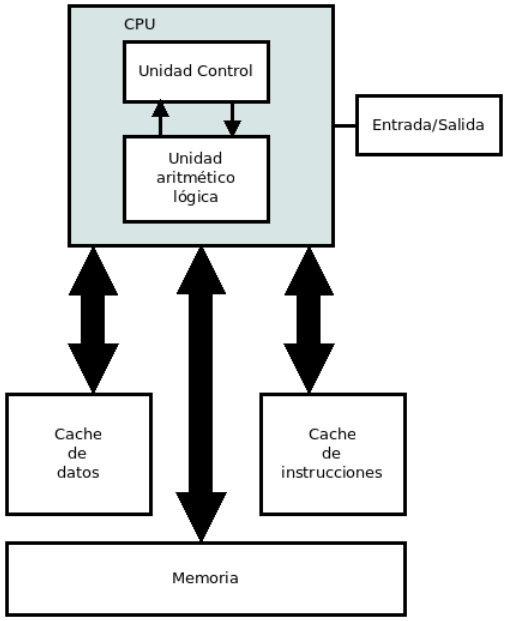


FIGURA A.3: Diagrama de arquitectura híbrida Von Neumann y Harvard.

A.1.4. Arquitecturas RISC y CISC

RISC, sigla del inglés *Reduced Instruction Set Computer* (computadora con conjunto de instrucciones reducido) es un tipo de arquitectura de microprocesadores caracterizado por utilizar una colección optimizada y pequeña de instrucciones. Surge como contraparte de las arquitecturas CISC (*Complex Instruction Set Computer*) cuyas instrucciones son mas especializadas llevando a cabo varias subtareas para concretar una instrucción. Para fines de los años 70 los diseñadores de procesadores se enfocaron en simplificar el conjunto de instrucciones de las arquitecturas, lo que redujo su cantidad de instrucciones y también otorgó nuevas características a la arquitectura:

- Instrucciones simples. Ancho de instrucción fijo y pocos formatos de instrucción. Las operaciones se realizan solo entre registros, se utilizan instrucciones separadas de carga y almacenamiento para leer o escribir en la memoria respectivamente.
- Modos de direccionamiento simples. En conjunto con instrucciones de ancho fijo y tipos de datos facilitan el paralelismo de ejecución. Los direccionamientos complejos se llevan a cabo combinando operaciones con direccionamiento simple.
- Reducción de los tipos de datos. Las estructuras complejas de datos ofrecen flexibilidad a costa de tiempo de ejecución. Las arquitecturas RISC soportan los tipos de datos simples (enteros, caracteres, etc.)

- Extenso banco de registros de propósito general. Para prevenir numerosos accesos a memoria.

El término CISC fue acuñado con el surgimiento de RISC, para denominar a toda arquitectura que no fuera de conjunto reducido de instrucciones. Puede contar con una gran colección de instrucciones que además suelen ser de mayor complejidad, ejecutándose mediante varias operaciones de menor nivel que corren de forma transparente accediendo a memoria, efectuando cálculos, etc. Los procesadores que utilizan esta arquitectura fueron pensados para facilitar la escritura de compiladores, mejorar la eficiencia de ejecución y brindar mejor soporte para lenguajes de programación de alto nivel. Otras de sus características son:

- Pocos registros de propósito múltiple. Muchas instrucciones operan de memoria a memoria.
- Variedad de registros de propósito especial. Para el manejo de interrupciones, pila, y funciones específicas.
- Decodificación de instrucciones complejas.

Dado que el término CISC se consolida como consecuencia de la propuesta RISC, y se aplica como generalidad a toda arquitectura que no utilice instrucciones específicas de carga-almacenamiento, existen en la actualidad procesadores CISC con menor cantidad de instrucciones que un RISC e incluso menor complejidad de implementación. Si bien esto resulta contradictorio, la diferencia radica en el enfoque de cada uno para realizar accesos a memoria. El siguiente ejemplo pretende clarificar cada uno de ellos analizando la instrucción de multiplicación. El objetivo de una arquitectura CISC es llevar a cabo la tarea con la menor cantidad de líneas de código posible, y para ello el procesador debe contar con un hardware capaz de ejecutar una serie de operaciones. En el caso de la multiplicación debe cargar por separado dos valores de la memoria a dos registros, calcular el producto mediante la unidad de ejecución y luego almacenar dicho resultado. Todas estas tareas se llevan a cabo con una línea de código ensamblador como:

$MUL \;\; dirMem[A], dirMem[B].$

El procesador opera directamente en la memoria sin requerir de instrucciones explícitas de carga o almacenamiento, asemejándose a un lenguaje de alto nivel. Algunas ventajas

son que el compilador tiene menor trabajo para traducir un lenguaje de alto nivel a dicho código ensamblador y el largo del programa es menor, ocupando menos memoria. En contraparte, la multiplicación en una arquitectura RISC se lleva a cabo en pequeñas tareas sucesivas como:

$LOAD \quad R1, dirMem[A]$
$LOAD \quad R2, dirMem[B]$
$PROD \quad R1, R2$
$STORE \quad dirMem, R1$

El enfoque RISC requiere menor cantidad de hardware comparado con el que es necesario para llevar a cabo instrucciones complejas. El uso explícito de las instrucciones de carga y almacenamiento en memoria reduce la cantidad de trabajo realizada por el procesador, ya que en CISC todos los registros implícitamente utilizados son borrados y existen muchas situaciones en las que el valor borrado debe usarse inmediatamente después, requiriendo su recarga.

A.2. La técnica del Pipeline

A.2.1. Descripción

Partiendo desde el modelo de Von Neumann se consideran los siguientes pasos necesarios para la ejecución de una instrucción: en primer lugar la búsqueda de la siguiente instrucción (*fetch*) indicado por el contador del programa de la máquina, segundo la decodificación de la misma, tercero su ejecución y finalmente en cuarto lugar el almacenamiento del resultado. En los primeros diseños de procesadores se observó que esta estructura descripta carecía de eficiencia al ejecutarse secuencialmente instrucción por instrucción. Surge entonces como una de las primeras formas de paralelismo la superposición de los pasos de ejecución de instrucciones contiguas, conocido hoy como *pipelining*. Conceptualmente se suele asociar el *pipeline* con una línea de montaje. Suponiendo tres distintas labores llamadas X, Y, Z, se las divide en n sublabores tal que resultan $X_1, X_2, ...X_n, Y_1, Y_2, ...Y_n, Z_1, Z_2, ...Z_n$, todas con un tiempo de procesamiento similar. Si la línea de montaje posee distintas estaciones, cada sublabor es procesado por distintas etapas del proceso a su debido tiempo. Pueden entonces superponerse en ejecución las

sublabores de las tres labores de manera que cuando la sublabor X_1 termina la primer etapa, comienza a procesarse la sublabor Y_1 por esta misma mientras que en simultáneo la sublabor X_2 es procesada por la segunda etapa. Es así como puede incrementarse la cantidad de labores producidas por unidad de tiempo, o en el caso del procesador, más instrucciones.

Se procede a inspeccionar el funcionamiento básico de un *pipeline* de cinco etapas basado en las instrucciones ejecutadas por un procesador RISC, que posee una arquitectura tal que todas las instrucciones se llevan a cabo empleando registros o con operandos inmediatos excepto por las que acceden a memoria. Todas las instrucciones poseen igual cantidad de bits, el procesador cuenta con un banco de registros de múltiples propósitos, un contador de programa, un cache de instrucciones y uno de datos respaldados por una jerarquía de memoria. Para mayor simplicidad en la descripción se supone idealidad del funcionamiento de las cache. El procesador puede llevar a cabo tres tipos de instrucciones: aritmético-lógicas, de control como saltos condicionales y no condicionales, o de carga-almacenamiento (donde la instrucción de carga es la que más etapas requiere). Durante la búsqueda de instrucción (IF del inglés *instruction fetch*) se toma de memoria la instrucción señalada por el puntero del programa que es posteriormente incrementado, luego en la etapa de decodificación (ID, *instruction decode*) se reconoce el tipo de instrucción y se configuran señales de control pertinentes. En la etapa de ejecución se computa la dirección y en la siguiente, de acceso a memoria (Mem), se toma el contenido del lugar de memoria indicado por la instrucción. Finalmente la etapa de escritura (WB por *writeback*) se escribe en el registro destino los datos adquiridos. Acorde con esta breve descripción se puede diagramar un esquema esencial del *pipeline* como muestra la Fig. A.4, en la que se aprecia un registro de *pipeline* entre cada etapa con el nombre de la etapa anterior y siguiente.

La representación gráfica del paralelismo logrado con *pipelining* se describe en la Fig. A.5, suponiendo una ejecución secuencial de las instrucciones. La consecuencia inmediatamente apreciable de esta implementación es que los recursos disponibles del sistema no pueden ser compartidos por las etapas durante la ejecución. Si no se contara con una memoria para instrucciones separada de la memoria de datos, se produciría interferencia durante una instrucción de carga cuando una instrucción subsecuente comience su etapa de *fetch*. Otra consecuencia es que dependiendo la instrucción ejecutada, se producen

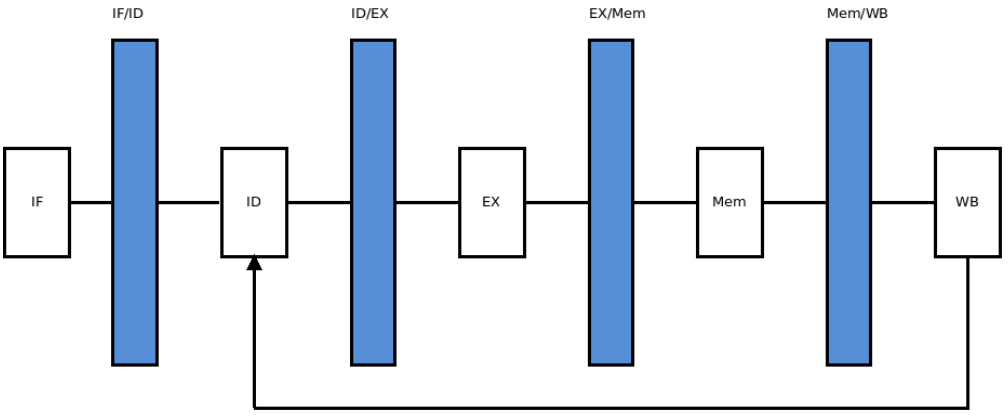


FIGURA A.4: División de la ejecución en cinco etapas para implementación de *pipeline*.

etapas en las que no se realiza ninguna tarea en particular (estado *idle*) como por ejemplo una aritmético-lógica en cuya etapa Mem solo se estaría pasando el resultado a la siguiente etapa ya que no se emplea la memoria.

Realizando un análisis de las etapas se procede a identificar los recursos necesarios para las mismas. Tanto la etapa de búsqueda (IF) como la de decodificación (ID) son comunes a toda las instrucciones realizadas por el procesador RISC. En la primera se busca la instrucción indicada por el puntero (PC) y se incrementa el mismo, tanto la instrucción como el valor del puntero son almacenados en el registro IF/ID. Se requiere entonces al menos un sumador para el incremento del puntero y la memoria donde se almacenan las instrucciones. Para la etapa de decodificación donde se interpreta la instrucción se requiere de una unidad de control cuya lógica configura las señales pertinentes a la ejecución, agregada al *pipeline* en la Fig. A.6. El resto de la información contenida en la instrucción como registros fuente y destino, tipo de operación aritmética, desplazamientos y extensiones de bits de valores inmediatos son almacenados en el registro ID/EX.

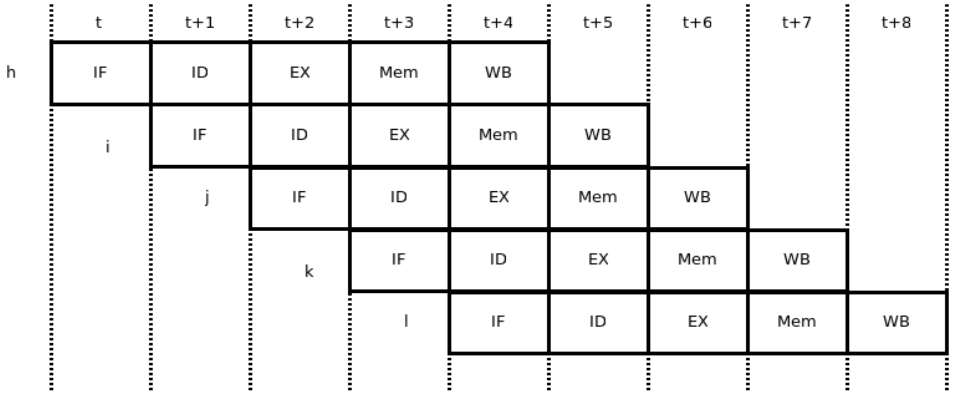


FIGURA A.5: Diagrama conceptual de operación con *pipeline*.

En la etapa de ejecución (EX) se computa una operación aritmético-lógica o una dirección de memoria, requiriéndose una ALU para tal fin. Los resultados se almacenan en el registro EX/Mem que también recibe datos del ID/EX. En la etapa de memoria (Mem), se realizan tareas de lectura o escritura según el caso de ser una instrucción de carga o almacenamiento. En dicho caso se almacena en el registro Mem/WB el contenido de la memoria de datos (si es una carga) o los resultados de una operación aritmético-lógica pasados directamente desde el registro EX/Mem. Finalmente en la etapa de escritura (WB) se almacena el resultado de la instrucción en el registro destino del banco de registros.

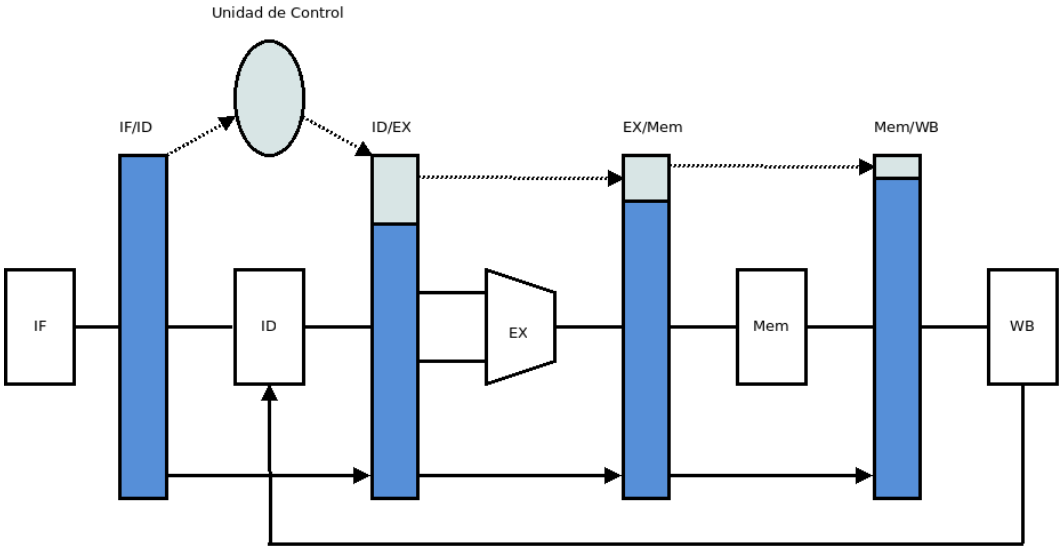


FIGURA A.6: Esquema de *pipeline* básico de cinco etapas.

A.2.2. Riesgos

Considerando los factores que afectan la operación ideal del *pipeline*, existen tres tipos de riesgos que comprometen su funcionamiento. Cuando distintas etapas requieren de los mismos recursos del sistema para llevar a cabo su tarea, se trata de un riesgo estructural. Cuando, por otro lado, una instrucción en el *pipeline* depende del resultado aún no concretado de una instrucción previa, se lo denomina riesgo de datos. Si en cambio el flujo de control no es secuencial, se lo llama riesgo de control.

Se procede a analizar el caso de riesgo de datos con más detalle, definiendo que una instrucción j es dependiente de la instrucción i si la salida de ésta última es un operando de la primera. A este tipo de dependencia se la llama lectura pos escritura (RAW del inglés *Read After Write*). Otro tipo de dependencia es la de escritura pos lectura (WAR, por *Write After Read*) y sucede cuando un registro usado como operando de una instrucción es escrito en una instrucción posterior concurrente en el *pipeline*. También existen las dependencias escritura pos escritura (WAW, *Write After Write*) cuando dos instrucciones escriben en el mismo registro.

En el ejemplo de la Fig. A.7 se producen tres dependencias RAW: j depende de i ya que uno de los operandos de la suma es el resultado que se almacena en $R3$ en i, k depende de i por la misma razón pero además su primer operando se almacena en $R2$ en la instrucción j. También se produce una dependencia WAR entre k e i, ya que la primera escribe en $R7$ que contiene uno de los operandos de la instrucción i. Para el caso del *pipeline* simple presentado en esta sección, los dos últimos tipos de dependencia (WAR y WAW) no generan inconvenientes ya que se resuelven fácilmente empleando distintos registros para la escritura en caso de disponerse un amplio banco de registros. Por otra parte, las dependencias RAW del ejemplo son dictadas por el orden del programa (conjunto de instrucciones) y deben ser resueltas. Se considera una serie de instrucciones que efectúan operaciones aritméticas como en el ejemplo, si la instrucción i sucede en el tiempo t, el resultado se escribe en $R3$ en $t+4$. Además, la instrucción j requiere dicho dato en $t+2$ de la etapa ID/EX que, como puede observarse en la Fig. A.8, ya se encuentra calculado

$$
\begin{aligned}
i &: R3 <= R4 + R7 \\
j &: R2 <= R3 + R8 \\
k &: R7 <= R2 + R3
\end{aligned}
$$

FIGURA A.7: Ejemplo de riesgo de datos tipo RAW aritmético en el *pipeline*.

por la ALU para ese instante. Es decir, con un adecuado direccionamiento (cables y multiplexores) se puede adelantar el resultado al registro ID/EX para ser usado como operando del próximo cálculo efectuado por la ALU (instrucción j). De igual manera, en $t + 3$ el resultado almacenado en $R3$ se encuentra en el registro EX/Mem y puede adelantarse al registro ID/EX para la instrucción k. Para el control de este tipo de riesgos se adiciona al sistema de *pipeline* la Unidad de Comunicación de Datos, tal como se muestra en la Fig. A.9.

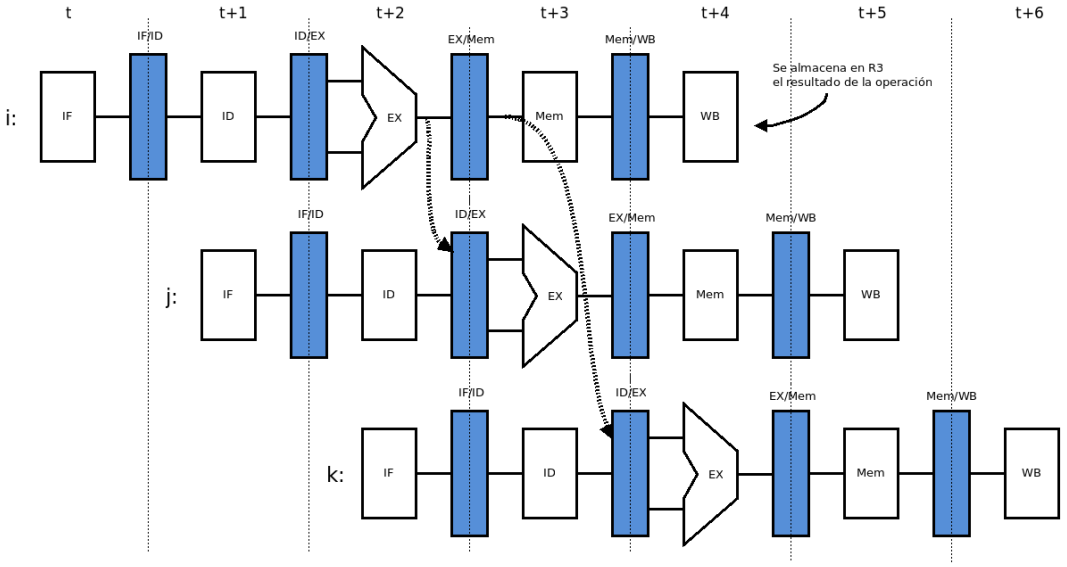


FIGURA A.8: Diagrama de riesgo de datos tipo RAW en el *pipeline*.

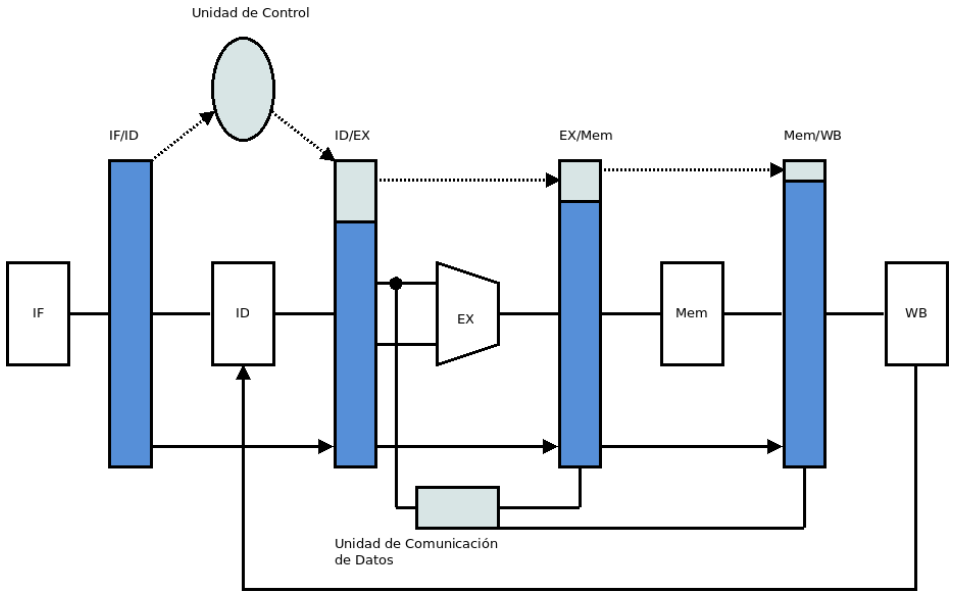


FIGURA A.9: Esquema de *pipeline* de 5 etapas con Unidad de Comunicación de Datos.

Muy diferente es el caso que involucra instrucciones del tipo carga-almacenamiento. Analizando el ejemplo de A.10, es evidente que no pueden adelantarse los datos debido a que la carga de datos de i no estará lista sino hasta el tiempo $t + 3$ durante la etapa Mem, mientras que es necesitado por la etapa EX de la instrucción j en $t+2$. La situación fuerza la necesidad de retrasar la instrucción j un ciclo utilizando una operación nula (No-Op) como muestra la Fig. A.11. El proceso de detección del riesgo e inserción del retraso se lleva a cabo mediante una unidad de retraso que se comunica con la unidad de control y ejerce cambios en la etapa IF, dado que es necesario retener la instrucción actual y no comenzar la búsqueda de la siguiente ante un retraso. La unidad de retraso debe también reconocer y solucionar casos particulares de las dependencias básicas enunciadas como por ejemplo cuando dos instrucciones escriben en el mismo registro que a la vez es utilizado como operando de la último instrucción, o cuando se mueven datos de un lugar de memoria a otro (carga y almacenamiento seguidos). La figura A.12 muestra el *pipeline* con la unidad de retraso agregada.

Continuando con los otros riesgos se analiza ahora el caso de los de control, producidos cuando se interrumpe la ejecución secuencial de instrucciones ante un salto. Se comienza abordando el caso de saltos condicionales, donde luego de la etapa IF e ID, se procede

$$i : R2 <= Mem[R1]$$
$$j : R4 <= R2 + R3$$

FIGURA A.10: Ejemplo de riesgo de datos tipo RAW con memoria en el *pipeline*.

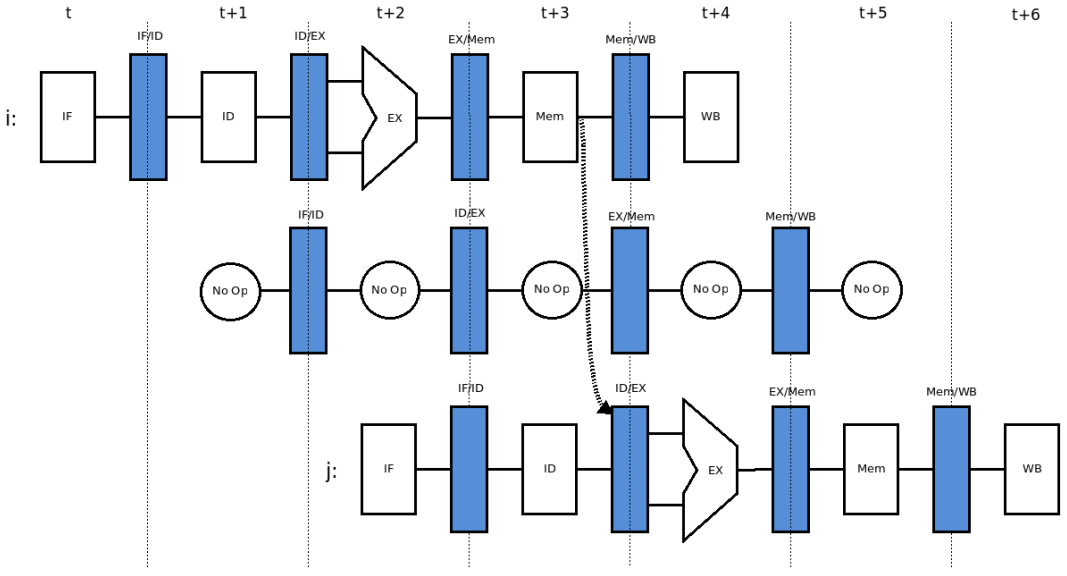


FIGURA A.11: Diagrama de riesgo de datos tipo RAW con inserción de retraso en el *pipeline*.

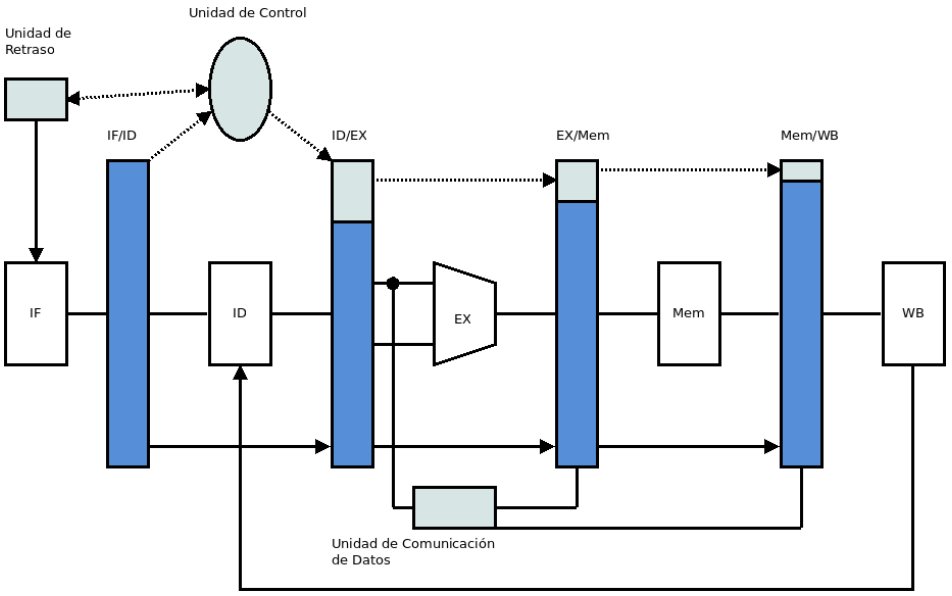


FIGURA A.12: Esquema de *pipeline* de cinco etapas con Unidad de Retraso.

a comparar el contenido de los dos operandos almacenados en registros, se configura el *flag* correspondiente al resultado y en caso de tomarse el salto se debe computar el nuevo PC y modificar el valor actual del mismo. Como se requieren dos cómputos en caso de tomarse el salto, debe utilizarse la ALU en dos ocasiones durante la ejecución de la instrucción. Es posible agregar otra ALU para dicho propósito o bien puede calcularse la dirección objetivo del salto durante la etapa ID para cada instrucción (aunque no sea un salto) y almacenarse dicha información en el registro ID/EX. En caso de que necesite ser empleada, se actualiza el PC con la nueva dirección durante la etapa EX ya que la solución a la operación lógica entre los registros se adquiere en dicha etapa. Si el salto no es tomado o la instrucción no se trata de un salto, la información computada se descarta. El riesgo acarreado por este tipo de instrucciones radica en que en caso de ser tomado un salto condicional, las dos siguientes instrucciones que yacen en el *pipeline* no corresponden al nuevo flujo dictado. La tercera aún no inicia su etapa IF para cuando se resuelve el salto y su accionar debe ser evitado. La manera mas simple de resolver la discontinuidad del flujo es colocar dos ciclos de retraso ante la presencia de toda instrucción de salto condicional ya sea tomado o no. De este modo se evita la búsqueda de nuevas instrucciones durante el retraso y se asegura la continuidad del flujo ya sea porque el PC fue actualizado por el salto tomado o haya permanecido invariable si no se tomó. La desventaja evidente de este método es que en cada salto tomado, se insertan retrasos que no producen ningún efecto y degradan la eficiencia. Una posible mejora es predecir que el salto nunca es tomado, suponiendo que las dos instrucciones siguientes

son correctas, y en caso de fallar la predicción anular las instrucciones con ciclos de no operación (No-op) ya que en el instante que se resuelve el salto dichas instrucciones no han modificado registros ni memoria.

Los dos riesgos de control restantes son las excepciones y las interrupciones cuya manipulación corresponde al administrador de excepciones incluido en el esquema del *pipeline* de la Fig. A.13. Las primeras son consecuencia de la ejecución de instrucciones como puede ser la detección de un *opcode* no existente en la etapa ID, divisiones por cero, *overflows* y otras situaciones que requieren atención inmediata y deben detener la secuencia del programa. Por el contrario, las interrupciones son consecuencia de estímulos externos a la ejecución de las instrucciones del programa como pueden ser fallas de energía o la comunicación con periféricos. Cualquiera sea el caso que acontezca, el estado de ejecución del programa al momento de la interrupción o excepción debe ser guardado y la interrupción atendida. Luego puede retomarse el curso del programa tal cual fue guardado excepto en casos que la interrupción fuerce abortar la ejecución. Ante una excepción producida por una instrucción (i) en el *pipeline*, el mecanismo elemental de tratamiento de ésta incluye: finalización normal de las instrucciones previas residentes en el *pipeline* (h, g, f, etc.) y guardado de resultados, anulación de la instrucción actual (i) y las posteriores residentes (j, k, etc.), guardado del valor del PC correspondiente a la instrucción actual (i). Un *flag* de excepción es insertado en el registro de *pipeline* donde ocurre la misma y todas sus lineas de control son anuladas, previniendo la posterior búsqueda de instrucciones. El *flag* de excepción es pasado a los registros de *pipeline*

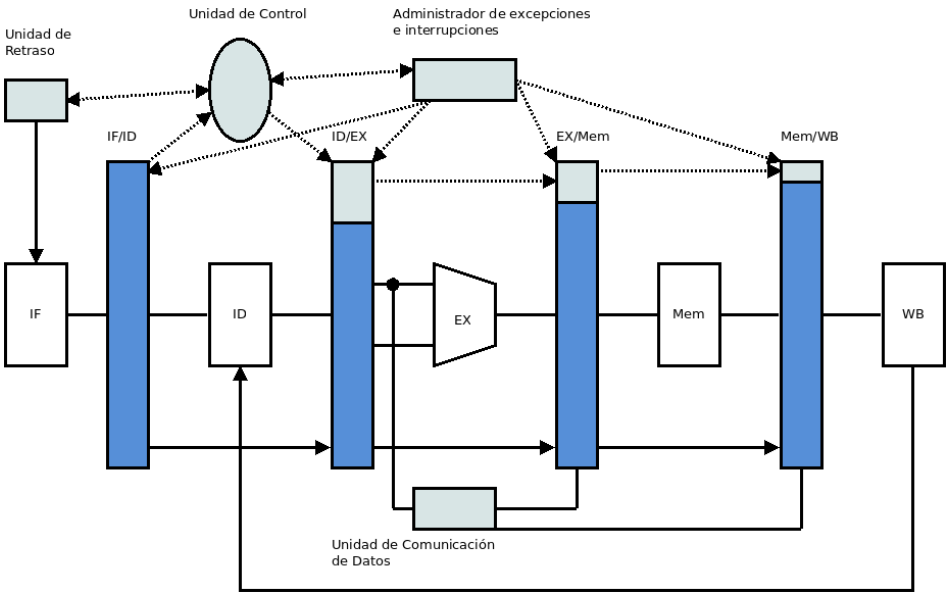


FIGURA A.13: Esquema de *pipeline* de cinco etapas con manejo de excepciones.

subsecuentes hasta que alcance Mem/WB y se guarda el PC allí almacenado que es el de la instrucción que produjo la excepción. A continuación se anulan todos los registros del *pipeline* y se otorga el control a un administrador de excepciones. El procedimiento descrito asegura que las excepciones sean atendidas según el orden del programa y no necesariamente en el orden cronológico en que éstas ocurren. El tratamiento de las interrupciones es análogo, controlándose en cada ciclo la presencia de las mismas. En caso de existir algún interrupción pendiente se detiene la etapa de búsqueda (IF), se limpia el *pipeline* completando las instrucciones ya residentes y se guarda el valor del PC de la primer instrucción que quedó sin buscarse en memoria. Luego el control de flujo es cedido al administrador de interrupciones.

Apéndice B

Características e instrucciones de la arquitectura MIPS32

Este apéndice resume las características principales de la arquitectura del microprocesador MIPS32 en las cuales se basa el diseño del bloque maestro del sistema implementado en el trabajo. Además se listan todas las instrucciones de la arquitectura MIPS32 y detalla en particular las especificaciones de formato de las instrucciones implementadas en el proyecto, tal como la arquitectura MIPS32 las define. Debido a que la implementación presentada en este documento no incluye manejo de excepciones y pipeline, es necesario adaptar la información plasmada en estas hojas a las necesidades del a implementación realizada.

B.1. Características de la arquitectura

B.1.1. Coprocesadores

La arquitectura MIPS define cuatro coprocesadores. El coprocesador 0 (CP0), también conocido como coprocesador de control de sistema, está incorporado al CPU y se le designa el sistema de memoria virtual y manejo de excepciones. Traduce las direcciones de memoria virtual a memoria física y maneja alternancias entre estados de usuario, supervisores y excepciones. Posee medios de control del sistema de cache y recuperación de errores del sistema. El coprocesador 1 (CP1) es también conocido como el FPU (sigla del inglés *Floating Point Processing Unit*) y es el encargado del procesamiento de punto flotante. El coprocesador 2 (CP2) es reservado para aplicaciones específicas configurables y por último el coprocesador 3 (CP3) que está reservado para el FPU en las implementaciones del primer lanzamiento 1 de MIPS64 y en todas las del segundo y subsiguientes lanzamientos.

B.1.2. Unidad de punto flotante FPU

En MIPS se definen los siguientes registros para la FPU. Treinta y dos registros de punto flotante (FPR) que son de 32 bits de ancho en una FPU de 32 bits y de 64 en una de dicho ancho. Cinco registros de control de FPU. Ocho códigos de condición de punto flotante pertenecientes al registro FCSR. Hasta el primer lanzamiento solo podía soportarse una FPU de 64 bits en la arquitectura MIPS64 y análogamente MIPS32 solo podía soportar una FPU de 32 bits. A partir del segundo lanzamiento es opcionalmente posible compatibilizar FPUs de 64 bits con ambas arquitecturas. Una FPU de 32 bits contiene 32 registros de 32 bits que pueden almacenar tipos de datos de 32 bits. Los tipos de datos de doble precisión se almacenan en parejas pares-impares de los FPRs mientras que los tipos de datos entero largo y par simple no son soportados. En una FPU de 64 bits se cuenta con 32 FPRS de 64 bits los cuales pueden almacenar cualquier tipo de dato.

B.1.3. Almacenamiento

Registros

Se dice que MIPS32 es una arquitectura de 32 bits ya que opera con datos de dicho ancho de palabra. Debido a que es una arquitectura RISC cuenta con un banco de registros, llamado usualmente en inglés *register file*, con 32 registros. Éstos se expresan en lenguaje ensamblador de MIPS con un signo $, una letra que identifica su uso general y un valor numérico. Los registros tienen múltiples usos (GPR sigla en inglés de *General Purpose Register*), siendo 18 de ellos utilizados generalmente para almacenar variables: $\$s0 - \$s7$, $\$t0 - \$t9$. En el caso del primer grupo la letra s proviene de *saved register* (del inglés, registro guardado) mientras que el segundo grupo la letra t representa *temporary register* (del inglés, registro temporal). Su diferencia radica en que ante algún interrupción de ejecución del programa el estado de los registros $\$s$ se guarda mientras que el de los $\$t$ se descarta. En la Tabla B.1 se listan todos los registros y una breve descripción de su utilización.

Nombre	Ubicación	Función
$\$zero$	0	valor constante 0
$\$at$	1	registro temporal del ensamblador
$\$v0 - \$v1$	2-3	registros para valor de retorno de funciones
$\$a0 - \$a3$	4-7	registros para argumentos de funciones
$\$t0 - \$t7$	8-15	registros para variables temporales
$\$s0 - \$s7$	16-23	registros para variables guardadas
$\$t8 - \$t9$	24-25	registros para variables temporales
$\$k0 - \$k1$	26-27	registros temporales destintados al sistema operativo
$\$gp$	28	registro de puntero global
$\$sp$	29	registro de puntero de pila
$\$fp$	30	registro de puntero de frame
$\$ra$	31	registro de dirección de retorno

Tabla B.1: Registros de propósito general.

Adicionalmente el microprocesador cuenta con 32 registros de punto flotante de 32 bits (FPR de la sigla en inglés *Floating-Point Register*).

Memoria

Otro medio disponible en la arquitectura para almacenar los datos es la memoria, que en comparación con el banco de registros posee mucha mayor capacidad pero a una velocidad de acceso menor. Utilizando la combinación entre memoria y registros la eficacia del procesador al correr un programa se ve notablemente incrementada. La memoria en la arquitectura MIPS32 puede interpretarse como un arreglo de palabras de datos, con 32 bits para direccionamiento. Es capaz de direccionar hasta bytes, que significa que cada byte en la memoria tiene una dirección única y se soporta la carga y almacenamiento de datos de tal ancho. A pesar de ello solo pueden realizarse transferencias de palabras alineadas, de manera que si cada palabra consta de 4 bytes con su propia dirección cada uno, las palabras poseen direcciones que son múltiplos de 4. Dado que las instrucciones de la arquitectura son de 32 bits de ancho, es preciso cuando se escribe el lenguaje ensamblador de MIPS indicar la dirección de memoria de una instrucción como múltiplo de 4. Es decir, las direcciones 1, 2 y 4 deben escribirse 4, 8 y 16 respectivamente. No son válidas direcciones no divisibles por 4 excepto en instrucciones que manipulen datos de ancho menor a 32 bits. Se ejemplifica el concepto de memoria en la Fig. B.1 con un arreglo de 512 bytes agrupados de a 4 (columnas) para formar 128 palabras (filas). En este ejemplo son suficientes 9 bits para direccionar los 512 bytes, y si se deseara seleccionar el byte 5 y 6 de la memoria debería indicarse la dirección $(000000101)_2$ y $(000000110)_2$ respectivamente. Podemos ver que el único cambio entre las direcciones de estos bytes se produce en los dos bits menos significativos y esto es consecuencia de

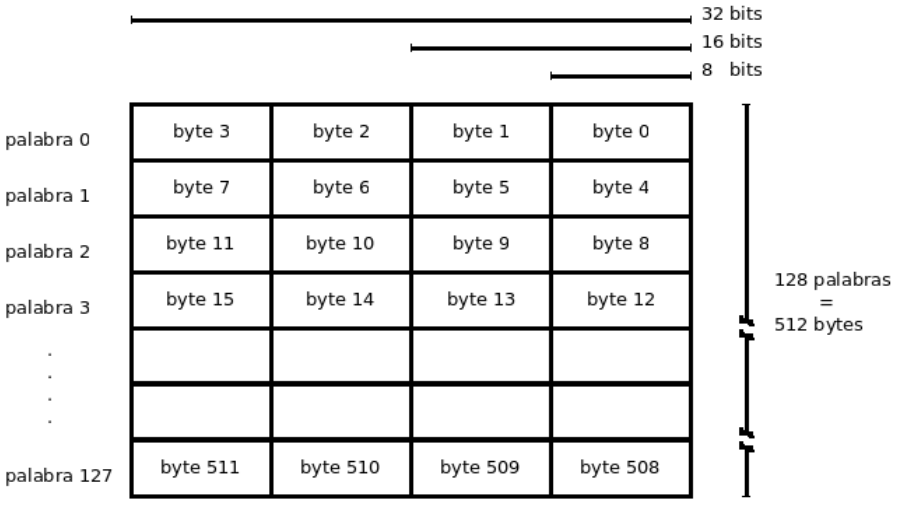


FIGURA B.1: Diagrama conceptual de memoria de 512 bytes como un arreglo de 128 palabras.

que ambos pertenecen a la misma palabra en memoria. Se concluye que si se realiza el direccionamiento de una palabra alineada (4 bytes de la fila) es suficiente con los 7 bits más significativos de los 9 totales. En el otro extremo, si se direcciona un byte en particular debe especificarse de cuál de los 4 bytes pertenecientes a la palabra direccionada se trata con los dos bits menos significativos restantes. Para direccionar medias palabras (16 bits) se puede prescindir del bit menos significativo de los 9, ya que en el bit 1 se indica si se trata de media palabra.

B.1.4. Tipos de datos

Los tipos de datos soportados en la arquitectura de 32 bits son: byte (8 bits), media palabra (*half word*, de 16 bits) y palabra completa (*word*, de 32 bits). Bytes y medias palabras se cargan en los registros con sus respectivos signos extendidos para completar los 32 bits del ancho de los registros.

B.1.5. Formato de instrucciones

MIPS utiliza instrucciones de ancho fijo de 32 bits. La arquitectura define tres tipos de instrucciones: tipo R, tipo I y tipo J. Se diferencian principalmente en los operandos empleados y en su naturaleza, integrando el primer tipo instrucciones que operan en tres registros. Las instrucciones del segundo tipo operan en dos registros y un valor inmediato de 16 bits incluido en la instrucción. Finalmente el tercer tipo opera directamente sobre un valor inmediato de 26 bits, también explícitamente incluido en la instrucción.

Instrucciones tipo R

Su nombre viene dado por "tipo registro", ya que se trata de instrucciones que operan datos almacenados en registros y almacenan el resultado también en uno. La instrucción distribuye sus 32 bits en seis campos, que son de cinco o seis bits como se describen a continuación y muestra en la Fig. B.2.

FIGURA B.2: Formato de campos de instrucción tipo R.

La operación a realizar se codifica en el primer y el último campo de seis bits llamado *opcode*, (*op* en su forma abreviada), y el campo *funct*. Los operandos se codifican en tres campos de cinco bits: *rs*, *rt* y *rd*. Los dos primeros indican los registros con los que se opera mientras que el tercero el registro destino para almacenar el resultado. El campo restante denominado *shamt* es también de cinco bits y se utiliza para las funciones de rotación de bits, en él se almacena el valor indicador del número de rotaciones. Para el resto de las instrucciones tipo R que no requieren rotación de bits el campo debe contener ceros. Los ejemplos más usuales de instrucciones de este tipo son las aritméticas y lógicas (Fig. B.3).

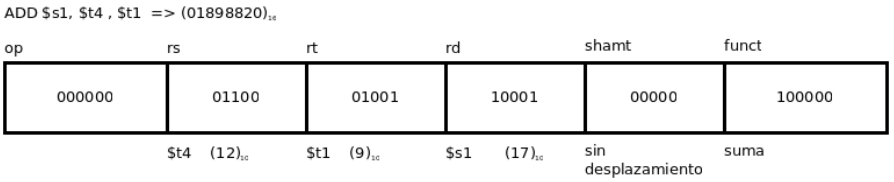


FIGURA B.3: Ejemplo de instrucción tipo R y sus campos.
Suma de los valores contenidos en los registros $\$t4$ y $\$t1$, almacenado en $\$s1$.

Instrucciones tipo I

Estas instrucciones toman su nombre de "tipo inmediato" ya que uno de sus campos alberga un valor utilizado en la operación y que no está almacenado en ningún medio del sistema. Los 32 bits se distribuyen esta vez en cuatro campos. Siempre en primer lugar se encuentra el *opcode* de seis bits, seguido ahora de tres operandos. Los dos primeros indican registros, *rs* y *rt*, siendo el segundo de destino para algunas operaciones. El tercer operando y cuarto campo es el valor inmediato, con 16 bits de ancho como muestra la Fig. B.4. Ejemplos de instrucciones tipo I son carga o almacenamiento de datos de o hacia la memoria y operaciones aritméticas donde el inmediato es uno de los operandos. Dado que en este último caso el operando puede ser negativo, el campo inmediato se interpreta en complemento a2 como en el ejemplo de la Fig. B.5. Además, es necesario considerar que las operaciones son de 32 bits de ancho y por ende se debe extender el valor inmediato agregando 16 ceros o unos dependiendo el signo.

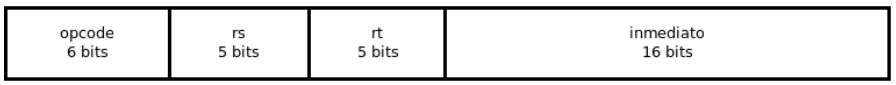


FIGURA B.4: Formato de campos de instrucción tipo I.

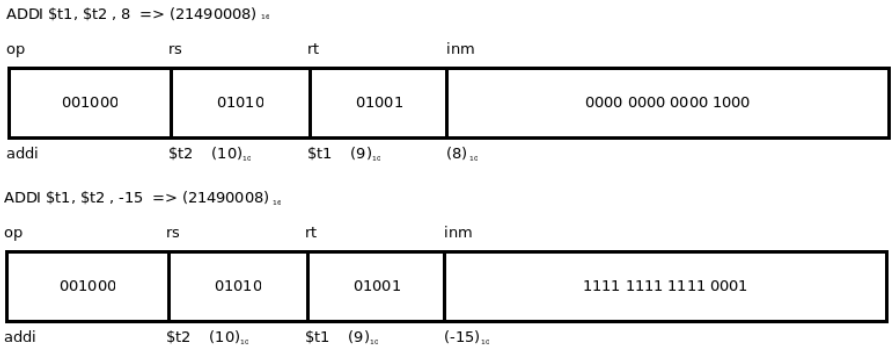


FIGURA B.5: Ejemplos de instrucciones tipo I.
Suma y resta entre el registro $\$t2$ y el valor inmediato 8 y 15 respectivamente.

Instrucciones tipo J

Obtienen su nombre por "tipo *jump*" (salto en inglés). Nuevamente lidera el campo *opcode* de seis bits seguido esta vez por el segundo y último campo de dirección con un ancho de 26 bits como muestra la Fig. B.6. Son utilizadas para saltos incondicionales entre líneas de un programa como ejemplifica la Fig. B.7.

FIGURA B.6: Formato de instrucción tipo J.

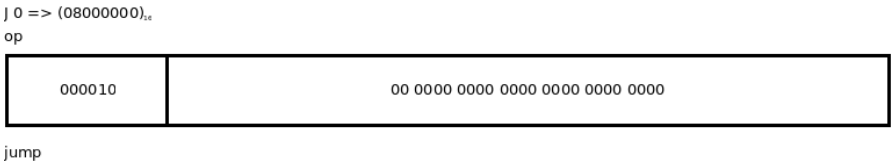


FIGURA B.7: Ejemplo de instrucción tipo J.
Salto a la dirección $(00000000)_{16}$.

B.1.6. Tipos de operaciones

En la arquitectura MIPS se define una variedad de instrucciones aritméticas y lógicas esenciales, las cuales además de su función específica son combinadas para generar otras instrucciones. Entre las aritméticas se cuenta con suma, resta, multiplicación y división con sus variantes que permiten operar entre registros (tipo R), con inmediatos (tipo I) y considerando signo o no. Para el caso de la multiplicación y división se emplean dos registros de uso especial llamados *hi* y *lo*, en representación de la mitad más significativa (*high* en inglés) y la menos significativa (*low*) del resultado que duplica en número de bits a sus operandos en la multiplicación. Para la división se almacena en *hi* el resto y en *lo* el cociente de los operandos. Entre las operaciones aritméticas que combinan las esenciales se encuentran las extensiones de signo, la multiplicación seguida de suma o resta del resultado con valores previamente almacenados en los registros *hi* y *lo* y la comparación de valores registro-registro o registro-inmediato con y sin signo. En la rama de las operaciones lógicas se hallan las básicas *and*, *or*, *xor* y *nor* todas ellas del tipo R, empleadas entre otros usos para enmascaramiento de bits no deseados (*and*) o para combinar bits de valores almacenados en distintos registros (*or*). También se soportan operaciones lógicas con valores inmediatos (*andi*, *ori*, *xori*) todas ellas del tipo I. En la Tabla B.2 se encuentran listadas todas las operaciones aritmético-lógicas. Otro subconjunto destacado de instrucciones es el que involucra a las variantes de carga, almacenamiento y control de memoria que caracterizan a la arquitectura RISC. Se hallan entre ellas la carga de palabras, medias palabras, bytes (con y sin signo) desde la memoria a los registros de propósito general. Análogamente están las instrucciones de almacenamiento en memoria para todos los formatos mencionados así como instrucciones de sincronización con memorias compartidas y cache, todas ellas listadas en la Tabla B.3. En el conjunto de instrucciones de desplazamiento se cuenta con opciones para desplazar hasta 31 bits de un registro en sentido izquierdo o derecho, agregando ceros (desplazamiento lógico) o el bit acorde al signo (desplazamiento aritmético) respectivamente. Las instrucciones tipo R de este conjunto especifican en un campo de la misma la cantidad de desplazamientos a realizar. La utilidad de estas operaciones radica en que N desplazamientos en sentido izquierdo equivale a multiplicar el valor en 2^N, y en caso de desplazar hacia la derecha equivale a dividir por tal valor. También se dispone de instrucciones de rotación de bits, los detalles de este subconjunto se listan en la Tabla B.4.

Dentro del conjunto de instrucciones de salto existen dos grupos diferenciados: los condicionales y los incondicionales. El primer conjunto contiene instrucciones que interrumpen el flujo secuencial del programa si se cumple una determinada condición. Pueden saltar previniendo la ejecución de instrucciones subsecuentes así como también regresar a una sección anterior del código para repetir un grupo de instrucciones ya ejecutadas. Entre las condiciones se puede comparar la igualdad entre dos valores almacenados en registros, su desigualdad, mayor o igual a cero y menor o igual a cero. Este tipo de saltos con condiciones se denominan *branch*. El segundo conjunto son los saltos que no verifican ninguna condición, análogamente pueden repetir código como prevenir ejecución de instrucciones subsecuentes. Este otro grupo se denomina *jump*. Se listan todas ellas en la Tabla B.5. Se cuenta también con un subconjunto de instrucciones para mover datos entre registros de propósito general de manera condicional, y cargar datos desde o hacia los registros *hi* y *lo* como se lista en la Tabla B.6. Finalmente se dispone de instrucciones de control para introducir estados de no operación, pausa de la ejecución, *breakpoints*, trampas para excepciones y otras funciones privilegiadas del sistema que se detallan en las Tablas B.7 a B.9. Respecto a la aritmética de punto flotante existe otro subconjunto de instrucciones especiales, en su mayoría análogas a las de aritmética convencional, para su conversión de formato, carga y almacenamiento de los mismos en memoria, operaciones aritmético-lógicas y otras operaciones listadas en las Tablas B.10 a B.14.

El primer registro del banco se denomina $\$zero$ y está reservado con el fin de simplificar la implementación de algunas instrucciones. Observando las listadas en las tablas se aprecia que no existen operaciones que explícitamente permitan mover un valor de un registro a otro incondicionalmente o la negación lógica de un valor almacenado. Para dichos casos se emplea el uso de instrucciones listadas en combinación del registro $\$zero$ para realizar la tarea. Mover un valor de un registro a otro se puede efectuar sumando el valor que se desea mover y el registro $zero, indicando como registro destino aquel a dónde se desea mover el valor.

"MOV $\$t0$, $\$t1$"= ADD $\$t0$, $\$t1$, $\$zero$

De manera similar, para negar todos los bits almacenados en un registro se puede efectuar la operación lógica *nor* (*O* negado) entre el registro $\$zero$ y dicho valor.

"NOT $\$t0$"= NOR $\$t0$, $\$t0$, $\$zero$

B.1.7. Modos de direccionamiento

MIPS32 emplea cinco modos de direccionamiento: registro, inmediato, base, relativo a PC y pseudo directo. Se describe brevemente cada uno de ellos a continuación.

- Direccionamiento por registro: se utilizan exclusivamente registros para toda fuente y destino de datos. Todas las instrucciones tipo R emplean este direccionamiento.
- Direccionamiento inmediato: se utilizan registros y un valor inmediato de 16 bits. Utilizado en algunas instrucciones tipo I.
- Direccionamiento base: la dirección efectiva de un dato en memoria se computa sumando la dirección contenida en un registro (rs) y la extensión de signo del valor inmediato de 16 bits. Se utiliza en las instrucciones de carga y almacenamiento de memoria.
- Direccionamiento relativo al PC: usado en las instrucciones de saltos incondicionales donde se actualiza el PC bajo cierta condición. Se toma el valor con signo del campo inmediato y se suma al valor actual del PC para computar el nuevo.
- Direccionamiento pseudo-directo: en este tipo de direccionamiento la instrucción incluye la dirección de interés completa. En esta arquitectura no puede implementarse de tal manera dado que 6 de los bits requeridos para especificar una dirección completa de 32 están destinados al opcode de la instrucción. Es por ello que casos como el *jump* especifican la dirección en un campo de 26 bits. Como consecuencia, los dos bits menos significativos deben ser siempre 0 ya que la dirección debe estar alineada (ser múltiplo de 4). Los cuatro bits más significativos se extraen de los cuatro más significativos del PC calculado para la siguiente instrucción. Debido a esto último, el rango de los saltos está limitado.

B.1.8. Mapa de memoria

Con direcciones de 32 bits de ancho la arquitectura MIPS32 posee un espacio de direcciones de 4 GB (2^{32} bytes) comenzando en $(00000000)_{16}$ y finalizando en $(FFFFFFFC)_{16}$. Dicho espacio se divide en cuatro bloques conceptuales mostrados en la Fig. B.8: el segmento de texto, el segmento de datos globales, el segmento de datos dinámicos y

el segmento reservado. En el primero es donde se almacena el programa en lenguaje máquina, con disponibilidad de albergar 256 MB de código. En el segmento global se almacenan las variables que sirven a todas las funciones del programa y son accesibles por estas. Las variables se definen durante el arranque del sistema, previo a la ejecución del programa, y se acceden combinando el uso del puntero global ($*gp*) y un *offset* de 16 bits. El registro $*gp* se inicializa en el valor $(100080000)_{16}$ y no se altera durante la ejecución del programa. El segmento de datos dinámico es el mayor de todos con una capacidad de 2 GB, y alberga todos los datos que se procesan y transfieren en el transcurso de la ejecución del programa. Uno de los métodos de almacenamiento es la pila, que crece desde el comienzo del segmento y en la cual se accede a cada dato en orden LIFO (sigla del inglés *Last In First Out*, el último en entrar es el primero en salir) y señala su tamaño actual a través del puntero $*sp*. En ella se guardan y restauran estados de los registros cuando son utilizadas por funciones consecuencia de la ejecución. Por último el segmento reservado no puede ser accedido por el programa, es parcialmente utilizado para interrupciones y para el mapa de memoria I/O.

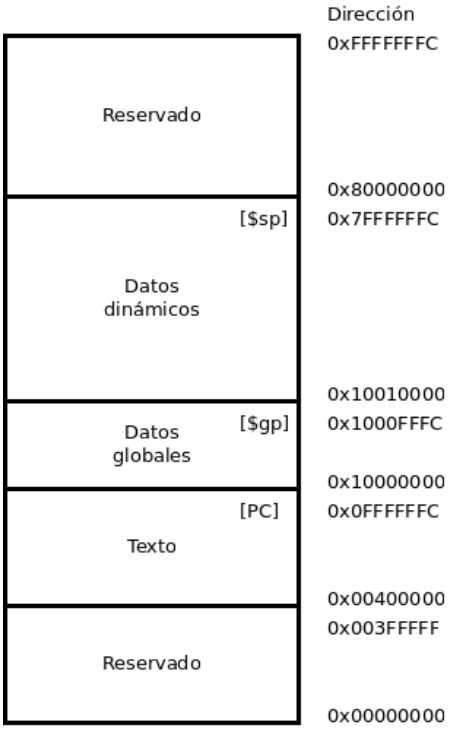


FIGURA B.8: Mapa de memoria de la arquitectura MIPS32.

B.2. Conjunto de instrucciones de la arquitectura MIPS32

Mnemónico	Función	Tipo
ADD	Sumar palabra	R
ADDI	Sumar palabra inmediata	I
ADDIU	Sumar palabra inmediata sin signo	I
ADDU	Sumar palabra sin signo	R
CLO	Contar cantidad de unos en una palabra	R
CLZ	Contar cantidad de ceros en un apalabra	R
DIV	Dividir palabra	R
DIVU	Dividir palabra sin signo	R
MADD	Multiplicar y sumar palabra a Hi, Lo	R
MADDU	Multiplicar y sumar palabra a Hi, Lo sin signo	R
MSUB	Multiplicar y restar palabra a Hi, Lo	R
MSUBU	Multiplicar y restar palabra a Hi, Lo sin signo	R
MUL	Multiplicar palabra a GPR	R
MULT	Multiplicar palabra	R
MULTU	Multiplicar palabra sin signo	R
SEB	Extender signo de Byte	R
SEH	Extender signo de media palabra	R
SLT	Poner a cero en caso de menor	R
SLTI	Poner a cero en caso de menor con inmediato	I
SLTIU	Poner a cero en caso de menor con inmediato sin signo	I
SLTU	Poner a cero en caso de menor sin signo	R
SUB	Sustraer palabra	R
SUBU	Sustraer palabra sin signo	R
AND	Y lógico	R
ANDI	Y lógico inmediato	I
LUI	Cargar inmediato en mitad más significativa de palabra	R
NOR	O lógico negado	R
OR	O lógico	R
ORI	O lógico inmediato	R
XOR	O exclusivo lógico	R
XORI	O exclusivo lógico inmediato	I

Tabla B.2: Instrucciones lógicas y aritméticas.

Mnemónico	Función	Tipo
LB	Cargar Byte	I
LBU	Cargar Byte sin signo	I
LH	Cargar media palabra	I
LHU	Cargar media palabra sin signo	I
LL	Cargar palabra enlazada	I
LW	Cargar palabra	I
LWL	Cargar palabra a izquierda	I
LWR	Cargar palabra a derecha	I
PREF	Pre-búsqueda	I
SB	Almacenar Byte	I
SC	Almacenar palabra condicional	I
SH	Almacenar media palabra	I
SW	Almacenar palabra	I
SWL	Almacenar palabra a izquierda	I
SWR	Almacenar palabra a derecha	I
SYNC	Sincronizar memoria compartida	-
SYNCI	Sincronizar escrituras en cache	I

Tabla B.3: Instrucciones de carga, almacenamiento y control de memoria.

Mnemónico	Función	Tipo
ROTR	Rotar palabra en sentido derecho	R
ROTRV	Rotar palabra en sentido derecho variable	R
SLL	Desplazar palabra en sentido izquierdo lógico	R
SLLV	Desplazar palabra en sentido izquierdo lógico variable	R
SRA	Desplazar palabra en sentido derecho aritmético	R
SRAV	Desplazar palabra en sentido derecho aritmético variable	R
SRL	Desplazar palabra en sentido derecho lógico	R
SRLV	Desplazar palabra en sentido derecho lógico variable	R

Tabla B.4: Instrucciones de desplazamiento y rotación.

Mnemónico	Función	Tipo
B	*Branch* incondicional	I
BAL	*Branch* y enlazar	I
BEQ	*Branch* ante igualdad	I
BGEZ	*Branch* ante mayor o igual a cero	I
BGEZAL	*Branch* ante mayor o igual a cero y enlazar	I
BGTZ	*Branch* ante mayor a cero	I
BLEZ	*Branch* ante menor o igual a cero	I
BLTZ	*Branch* ante menor a cero	I
BLTZAL	*Branch* ante menor a cero y enlazar	I
BNE	*Branch* ante desigualdad	I
J	Saltar incondicional	J
JAL	Saltar y enlazar	J
JALR	Saltar y enlazar registro	R
JALR.HB	Saltar y enlazar registro con barrera	R
JALX	Saltar con intercambio de enlace	J
JR	Saltar registro	R
JR.HB	Saltar registro con barrera de riesgos	R

Tabla B.5: Instrucciones de salto y *branch*.

Mnemónico	Función	Tipo
MFHI	Mover desde el registro Hi	R
MFLO	Mover desde el registro Lo	R
MOVF	Mover ante falso punto flotante	R
MOVN	Mover ante no nulo	R
MOVT	Mover ante verdadero punto flotante	R
MOVZ	Mover ante nulo	R
MTHI	Mover hacia el registro Hi	R
MTLO	Mover hacia el registro Lo	R
RDHWR	Leer registro de hardware	R

Tabla B.6: Instrucciones de movimiento de registros.

Mnemónico	Función	Tipo
BREAK	Breakpoint	-
SYSCALL	Llamada de sistema	-
TEQ	Atrapar ante igualdad	R
TEQI	Atrapar ante igual inmediato	I
TGE	Atrapar ante mayor o igual	R
TGEI	Atrapar ante mayor o igual inmediato	I
TGEIU	Atrapar ante mayor o igual inmediato sin signo	I
TGEU	Atrapar ante mayor o igual sin signo	R
TLT	Atrapar ante menor	R
TLTI	Atrapar ante menor inmediato	I
TLTIU	Atrapar ante menor inmediato sin signo	I
TLTU	Atrapar ante menor sin signo	R
TNE	Atrapar ante desigualdad	R
TNEI	Atrapar ante desigualdad inmediato	I

Tabla B.7: Instrucciones de atrapado de excepciones.

Mnemónico	Función	Tipo
EHB	Barrera de riesgos de ejecución	-
NOP	Sin operación	-
PAUSE	Esperar por	-
SSNOP	Sin operación superescalar	-

Tabla B.8: Instrucciones de control de CPU.

Mnemónico	Función	Tipo
CACHE	Operación de cache	FP
DI	Deshabilitar interrupciones	FP
EI	Habilitar interrupciones	FP
ERET	Retorno de excepción	FP
MFC0	Mover desde coprocesador 0	FP
MTC0	Mover a coprocesador 0	FP
RDPGPR	Leer GPR de shadow set anterior	FP
TLBP	Verificar TLB por entrada apareada	FP
TLBR	Leer entrada a TLB indexada	FP
TLBWI	Escribir entrada a TLB indexada	FP
TLBWR	Escribir entrada TLB aleatoria	FP
WAIT	Pasar a modo de espera	FP
WRPGPR	Escribir en GPR de shadowset anterior	FP

Tabla B.9: Instrucciones de privilegio.

Mnemónico	Función	Tipo
ABS.fmt	Valor absoluto	FP
ADD.fmt	Sumar	FP
DIV.fmt	Dividir	FP
MADD.fmt	Multiplicar y sumar	FP
MSUB.fmt	Multiplicar y restar	FP
MUL.fmt	Multiplicar	FP
NEG.fmt	Negación	FP
NMADD.fmt	Multiplicar y sumar negativo	FP
NMSUB.fmt	Multiplicar y restar negativo	FP
RECIP.fmt	Aproximación recíproca	FP
RSQRT.fmt	Aproximación de raíz cuadrada recíproca	FP
SQRT.fmt	Raíz cuadrada	FP
SUB.fmt	Restar	FP

Tabla B.10: Instrucciones aritméticas con punto flotante.

Mnemónico	Función	Tipo
LDC1	Cargar palabra doble a punto flotante	FP
LWC1	Cargar palabra a punto flotante	FP
PREFX	Pre-búsqueda indexada	FP
SDC1	Almacenar doble palabra de punto flotante	FP
SWC1	Almacenar palabra de punto flotante	FP

Tabla B.11: Instrucciones de carga y almacenamiento para punto flotante.

Mnemónico	Función	Tipo
CEIL.W.fmt	Convertir de punto flotante a palabra con techo	FP
CVT.D.fmt	Convertir de punto flotante a doble palabra punto flotante	FP
CVT.S.fmt	Convertir de punto flotante a punto flotante simple	FP
CVT.W.fmt	Convertir de punto flotante a palabra	FP
FLOOR.W.fmt	Convertir de punto flotante a palabra con piso	FP
ROUND.W.fmt	Convertir de punto flotante a palabra con redondeo	FP
TRUNC.W.fmt	Convertir de punto flotante a palabra con redondeo	FP

Tabla B.12: Instrucciones de conversión con punto flotante.

Mnemónico	Función	Tipo
BC1F	*Branch* ante punto flotante falso	FP
BC1T	*Branch* ante punto flotante verdadero	FP
C.cond.fmt	Comparación con punto flotante	FP

Tabla B.13: Instrucciones de *branch* y comparación con punto flotante.

Mnemónico	Función	Tipo
CFC1	Mover palabra de control desde punto flotante	FP
CTC1	Mover palabra de control a punto flotante	FP
MFC1	Mover palabra desde punto flotante	FP
MFHC1	Mover palabra desde parte alta de registro de punto flotante	FP
MOV.fmt	Mover punto flotante	FP
MOVF.fmt	Mover punto flotante condicional ante punto flotante falso	FP
MOVN.fmt	Mover punto flotante condicional ante no nulo	FP
MOVT.fmt	Mover punto flotante condicional ante punto flotante verdadero	FP
MOVZ.fmt	Mover punto flotante condicional ante nulo	FP
MTC1	Mover palabra a punto flotante	FP
MTHC1	Mover palabra a parte alta de registro de punto flotante	FP

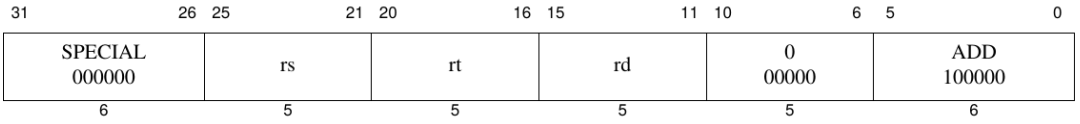
Tabla B.14: Instrucciones de transferencias entre registros.

Mnemónico	Función	Tipo
BC2F	*Branch* ante COP2 falso	FP
BC2T	*Branch* ante COP2 verdadero	FP
COP2	Operación de coprocesador 2	FP
LDC2	Cargar palabra doble a coprocesador 2	FP
LWC2	Cargar palabra a coprocesador 2	FP
SDC2	Almacenar palabra doble a coprocesador 2	FP
SWC2	Almacenar palabra a coprocesador 2	FP
CFC2	Mover palabra de control desde coprocesador 2	FP
CTC2	Mover palabra de control a coprocesador 2	FP
MFC2	Mover palabra desde comprocesador 2	FP
MFHC2	Mover palabra desde parte alta de registro de comprocesador 2	FP
MTC2	Mover palabra a comprocesador 2	FP
MTHC2	Mover palabra a parte alta de registro de comprocesador 2	FP

Tabla B.15: Instrucciones de coprocesador 2.

B.3. Descripción de instrucciones implementadas en el procesador propuesto

ADD



Formato: ADD rd, rs, rt

Propósito: Sumar palabra. Realiza la suma de dos enteros de 32 bits. En caso de *overflow*, debe atraparse excepción.

Descripción: $GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

La palabra de 32 bits contenida en GPR rt se suma a la contenida en GPR rs para producir un resultado de 32 bits.

- Si la suma resulta en un *overflow* en complemento a2 de 32 bits, el registro destino no se modifica y ocurre una excepción tipo *Integer Overflow*.
- Si la suma no produce *overflow*, el contenido de GRP rd se modifica con el resultado de la suma.

Restricciones: Ninguna.

Operación:

$temp \leftarrow (GPR[rs]_{31}||GPR[rs]_{31:0}) + (GPR[rt]_{31}||GPR[rt]_{31:0})$

if $temp_{32} \neq temp_{31}$ then

  SignalException(IntegerOverflow)

else

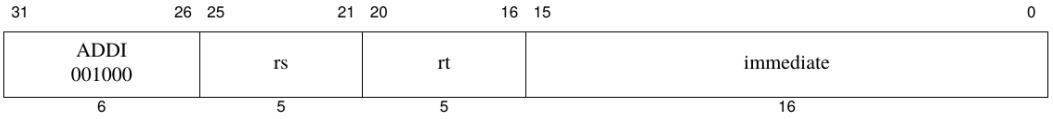
  $GPR[rd] \leftarrow temp$

endif

Excepciones: *Integer Overflow*.

Notas de programación: ADDU realiza la misma operación aritmética pero no desencadena excepciones.

ADDI



Formato: ADDI rt, rs, inmediato

Propósito: Sumar palabra inmediata. Realiza la suma de una constante y un entero de 32 bits. En caso de *overflow*, debe atraparse excepción.

Descripción: $GPR[rt] \leftarrow GPR[rs]+$ inmediato

El valor con signo inmediato de 16 bits se suma al contenido de GPR rs para producir un resultado de 32 bits.

- Si la suma resulta en un *overflow* en complemento a2 de 32 bits, el registro destino no se modifica y ocurre una excepción tipo *Integer Overflow*.
- Si la suma no produce *overflow*, el contenido de GRP rt se modifica con el resultado de la suma.

Restricciones: Ninguna.

Operación:

$temp \leftarrow (GPR[rs]_{31}||GPR[rs]_{31:0}) + sign\_extend(inmediato)$

if $temp_{32} \neq temp_{31}$ then

  SignalException(IntegerOverflow)

else

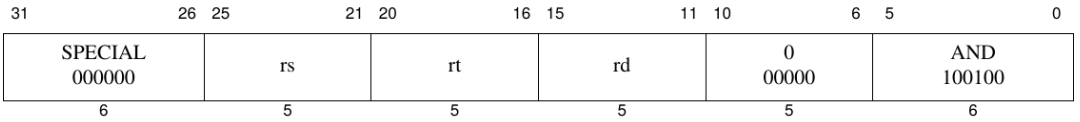
  $GPR[rt] \leftarrow temp$

endif

Excepciones: *Integer Overflow*.

Notas de programación: ADDIU realiza la misma operación aritmética pero no desencadena excepciones.

AND



Formato: AND rd, rs, rt

Propósito: Operación de intersección lógica bit a bit.

Descripción: $GPR[rd] \leftarrow GPR[rs] \;\; AND \;\; GPR[rt]$

El contenido de GPR[rs] se combina con el de GPR[rt] como la intersección lógica bit a bit. El resultado se almacena en GPR[rd].

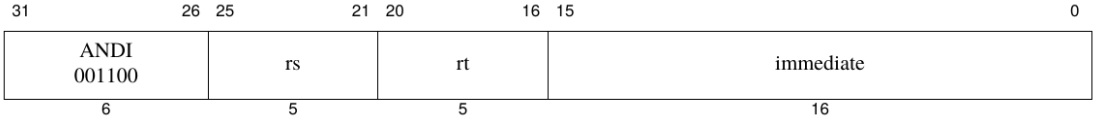
Restricciones: Ninguna.

Operación: $GPR[rd] \leftarrow GPR[rs] \;\; AND \;\; GPR[rt]$

Excepciones: Ninguna.

Notas de programación: Ninguna.

ANDI



Formato: ANDI rt, rs, inmediato

Propósito: Operación de intersección lógica bit a bit con una constante.

Descripción: $GPR[rt] \leftarrow GPR[rs]$AND inmediato

La constante de 16 bits se extiende con ceros a la izquierda y se combina con el contenido de GPR[rt] como la intersección lógica bit a bit. El resultado se almacena en GPR[rt].

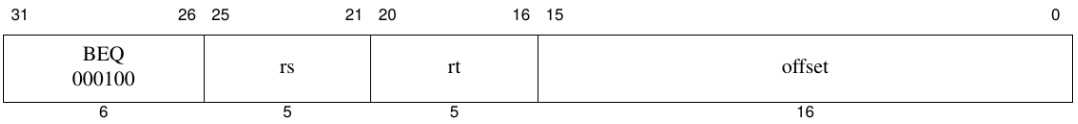
Restricciones: Ninguna.

Operación: $GPR[rd] \leftarrow GPR[rs] \;\; AND \;\; zero\_extend(inmediato)$

Excepciones: Ninguna.

Notas de programación: Ninguna.

BEQ



Formato: BEQ rs, rt, $offset$

Propósito: *Branch* ante igualdad. Compara el contenido de los GPRs y en caso de igualdad realiza un salto relativo al PC.

Descripción: if $GPR[rs] = GPR[rt]$ then branch

El campo *offset* de 16 bits, desplazado dos bits a la izquierda, se suma a la dirección de la siguiente instrucción a la de *branch* para formar la dirección de destinto relativa al PC. Si los contenidos de GPR rs y GPR rt son iguales, se salta la ejecución a la dirección calculada luego de ejecutar la instrucción del *delay slot*.

Restricciones: La operación del procesador es impredecible si una instrucción de salto, *branch*, ERET, DERET, o WAIT toma lugar en el *delay slot*.

Operación:

I: $target\_offset \leftarrow sign\_extend(offset||0^2)$

$condition \leftarrow (GPR[rs] = GPR[rt])$

I+1: if condition then

$PC \leftarrow PC + target\_offset$

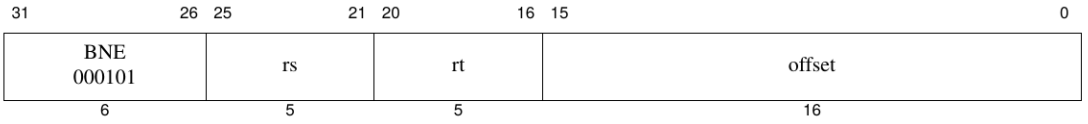
endif

Excepciones: Ninguna.

Notas de programación: El rango de *branch* con los del *offset* con signo de 18 bits es de ±128 KBytes. Se utiliza el salto (J) o salto con registro (JR) para saltar a direcciones fuera de dicho rango.

BEQ $zero, $zero, offset es la sentencia en lenguaje ensamblador para denotar un *branch* incondicional.

BNE



Formato: BNE rs, rt, $offset$

Propósito: *Branch* ante desigualdad. Compara el contenido de los GPRs y en caso de desigualdad realiza un salto relativo al PC.

Descripción: if $GPR[rs] \neq GPR[rt]$ then branch

El campo *offset* de 16 bits, desplazado dos bits a la izquierda, se suma a la dirección de la siguiente instrucción a la de *branch* para formar la dirección de destino relativa al PC. Si los contenidos de GPR rs y GPR rt no son iguales, se salta la ejecución a la dirección calculada luego de ejecutar la instrucción del *delay slot*.

Restricciones: La operación del procesador es impredecible si una instrucción de salto, *branch*, ERET, DERET, o WAIT toma lugar en el *delay slot*.

Operación:

I: $target\_offset \leftarrow sign\_extend(offset||0^2)$

$condition \leftarrow (GPR[rs] \neq GPR[rt])$

I+1: if condition then

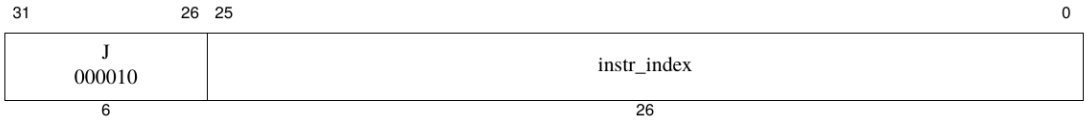
$PC \leftarrow PC + target\_offset$

endif

Excepciones: Ninguna.

Notas de programación: El rango de *branch* con los el *offset* con signo de 18 bits es de ±128 KBytes. Se utiliza el salto (J) o salto con registro (JR) para saltar a direcciones fuera de dicho rango.

J



Formato: J dirección

Propósito: Salto. Saltar dentro de la región de 256 MB alineada.

Descripción: Salto asociado a región del PC; la dirección efectiva se encuentra en la región actual de 256 MB del PC. Los 28 bits menos significativos de la dirección destino son los del campo *instr_ index* desplazados 2 bits. Los bits superiores restantes son los correspondientes a los de la dirección de la siguiente instrucción al salto.

Salta a la dirección efectiva destino. Se ejecuta la instrucción que sigue al salto, contenida en el *delay slot*.

Restricciones: La operación del procesador es impredecible si una instrucción de salto, *branch*, ERET, DERET, o WAIT toma lugar en el *delay slot*.

Operación:

I:

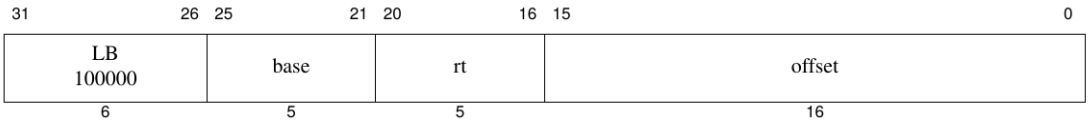
I+1: $PC \leftarrow PC_{GPRLEN-1\,:\,28} \,||\, instr\_index \,||\, 0^2$

Excepciones: Ninguna.

Notas de programación: El modo en que se forma la dirección destino, relativa a una región del PC, brinda grandes ventajas si todo el código del programa está contenido dentro de un rango de 256 MB. Permite que los *branchs* desde cualquier punto del programa arriben a cualquier sección del código.

Esta definición del salto tiene un caso particular: cuando la instrucción de salto está en la última palabra de una región de 256 MB, solo puede saltar a la siguiente región de 256 MB que contiene la instrucción del *delay slot*.

LB



Formato: LB rt, offset(base)

Propósito: Cargar byte. Carga un byte desde memoria, como un valor con signo.

Descripción: $GPR[rt] \leftarrow memory[GPR[base] + offset]$

El contenido de los 8 bits del lugar en memoria indicado por la dirección efectiva son buscados, extendidos en signo y almacenados en GPR rt. El contenido de 16 bits con signo del offset se suman con el de GPR *base* para formar la dirección efectiva.

Restricciones: Ninguna.

Operación:

$vAddr \leftarrow sign\_extend(offset) + GPR[base]$

$(pAddr, CCA) \leftarrow AddressTranslation(vAddr,\ DATA,\ LOAD)$

$pAddr \leftarrow pAddr_{PSIZE-1\ :\ 2}\ ||\ (pAddr_{1:0}\ xor\ ReverseEndian^2)$

$memword \leftarrow LoadMemory(CCA,\ BYTE,\ pAddr,\ vAddr,\ DATA)$

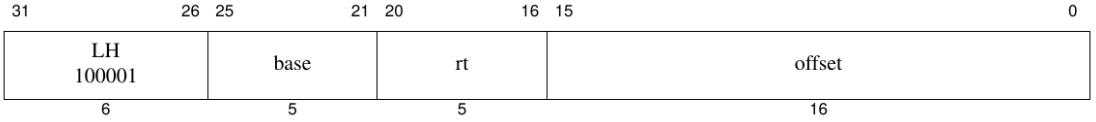
$byte \leftarrow vAddr_{1:0}\ xor\ BigEndianCPU^2$

$GPR[rt] \leftarrow sign\_extend(memword_{7+8*byte:8*byte})$

Excepciones: TLB Refill, TLB Invalid, Address Error, Watch.

Notas de programación: Ninguna.

LH



Formato: LH rt, offset(base)

Propósito: Cargar media palabra. Carga media palabra desde memoria, como un valor con signo.

Descripción: $GPR[rt] \leftarrow memory[GPR[base] + offset]$

El contenido de los 16 bits del lugar en memoria indicado por la dirección efectiva alineada son buscados, extendidos en signo y almacenados en GPR rt. El contenido de 16 bits con signo del offset se suman con el de GPR *base* para formar la dirección efectiva.

Restricciones: La dirección efectiva debe estar naturalmente alineada. Si el bit menos significativo de la dirección no es cero, se produce una excepción de error de dirección.

Operación:

$vAddr \leftarrow sign\_extend(offset) + GPR[base]$

if $vAddr_0 \neq 0$ then

$\quad SignalException(AddressError)$

endif

$(pAddr, CCA) \leftarrow AddressTranslation(vAddr,\ DATA,\ LOAD)$

$pAddr \leftarrow pAddr_{PSIZE-1\ :\ 2}\ ||\ (pAddr_{1:0}\ xor\ (ReverseEndian||0))$

$memword \leftarrow LoadMemory(CCA,\ HALFWORD,\ pAddr,\ vAddr,\ DATA)$

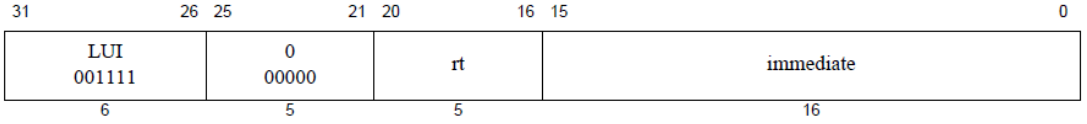
$byte \leftarrow vAddr_{1:0}\ xor\ BigEndianCPU||0$

$GPR[rt] \leftarrow sign\_extend(memword_{15+8*byte:8*byte})$

Excepciones: TLB Refill, TLB Invalid, Bus Error, Address Error, Watch.

Notas de programación: Ninguna.

LUI



Formato: LUI *rt*, inmediato

Propósito: Cargar inmediato en parte superior de registro. Carga una constante en la media palabra más significativa del GPR *rt*

Descripción: $GPR[rt] \leftarrow inmediato \mid\mid 0^{16}$

El valor inmediato de 16 bits es desplazado hacia la izquierda 16 bits y concatenado con 16 ceros en la media palabra menos significativa. El resultado se almacena en el registro de 32 bits GPR *rt*.

Restricciones: Ninguna.

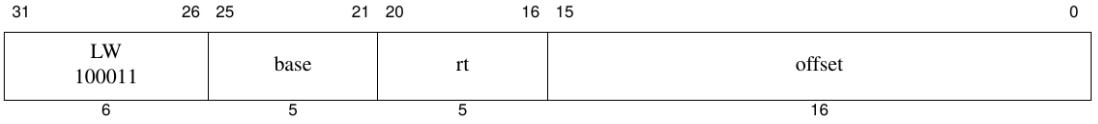
Operación:

$GPR[rt] \leftarrow inmediato \mid\mid 0^{16}$

Excepciones: Ninguna.

Notas de programación: Ninguna.

LW



Formato: LW *rt*, offset(base)

Propósito: Cargar palabra. Carga una palabra desde memoria, como un valor con signo.

Descripción: $GPR[rt] \leftarrow memory[GPR[base] + offset]$

El contenido de los 32 bits del lugar en memoria indicado por la dirección efectiva alineada son buscados, extendidos en signo y almacenados en GPR *rt*. El contenido de 16 bits con signo del offset se suman con el de GPR *base* para formar la dirección efectiva.

Restricciones: La dirección efectiva debe estar naturalmente alineada. Si los dos bits menos significativo de la dirección no son cero, se produce una excepción de error de dirección.

Operación:

$vAddr \leftarrow sign\_extend(offset) + GPR[base]$

if $vAddr_{1:0} \neq 0^2$ then

$SignalException(AddressError)$

endif

$(pAddr, CCA) \leftarrow AddressTranslation(vAddr,\ DATA,\ LOAD)$

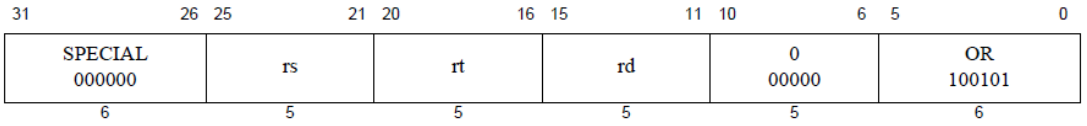
$memword \leftarrow LoadMemory(CCA,\ WORD,\ pAddr,\ vAddr,\ DATA)$

$GPR[rt] \leftarrow memword$

Excepciones: TLB Refill, TLB Invalid, Bus Error, Address Error, Watch.

Notas de programación: Ninguna.

OR



Formato: OR *rd*, *rs*, *rt*

Propósito: Unión. Realiza la unión lógica bit a bit.

Descripción: $GPR[rd] \leftarrow GPR[rs]\ OR\ GPR[rt]$

El contenido de GPR *rs* se combina con el de GPR *rt* como la unión lógica bit a bit. Se almacena el resultado en GPR *rd*.

Restricciones: Ninguna.

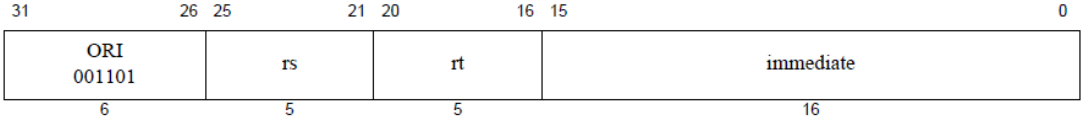
Operación:

$GPR[rd] \leftarrow GPR[rs]\ OR\ GPR[rt]$

Excepciones: Ninguna.

Notas de programación: Ninguna.

ORI



Formato: ORI *rt*, *rs*, inmediato

Propósito: Unión lógica con inmediato. Realiza la unión lógica con una constante.

Descripción: $GPR[rt] \leftarrow GPR[rs]\ OR\ inmediato$

Los 16 bits del valor inmediato son extendidos con ceros y combinados con el contenido de GPR *rs* como unión lógica. El resultado se almacena en los 32 bits de GPR *rt*.

Restricciones: Ninguna.

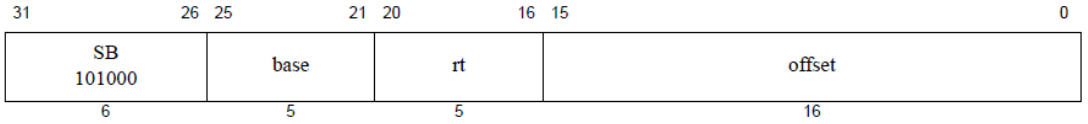
Operación:

$GPR[rt] \leftarrow GPR[rs]\ OR\ inmediato$

Excepciones: Ninguna.

Notas de programación: Ninguna.

SB



Formato: SB *rt*, offset(base)

Propósito: Almacenar byte. Almacena un byte en una palabra de memoria.

Descripción: $memory[GPR[base] + offset] \leftarrow GPR[rt]$

El byte menos significativo de GPR *rt* es almacenado en memoria en la dirección efectiva indicada. La dirección se forma con la suma de los 16 bits con signo del *offset* y el contenido del registro GPR base.

Restricciones: Ninguna.

Operación:

$vAddr \leftarrow sign\_extend(offset) + GPR[base]$

$(pAddr, CCA) \leftarrow AddressTranslation(vAddr, DATA, STORE)$

$pAddr \leftarrow pAddr_{PSIZE-1:2} \mid\mid (pAddr_{1:0}\ XOR\ ReverseEndian^2)$

$bytesel \leftarrow vAddr_{1:0}\ XOR\ BigEndianCPU^2$

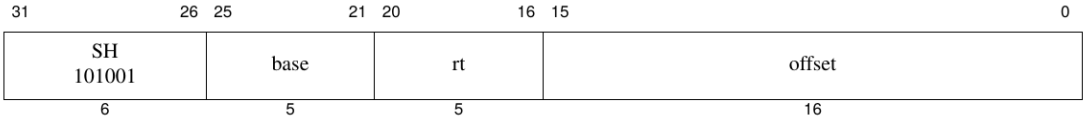
$dataword \leftarrow GPR[rt]_{31-8*bytesel:0} \mid\mid 0^{8*bytesel}$

$StoreMemory(CCA, BYTE, dataword, pAddr, vAddr, DATA)$

Excepciones: TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Watch.

Notas de programación: Ninguna.

SH



Formato: SH rt, offset(base)

Propósito: Almacenar media palabra. Almacena media palabra (16 bits) en memoria.

Descripción: $memory[GPR[base] + offset] \leftarrow GPR[rt]$

Los 16 bits menos significativos de GPR[rt] son almacenados en memoria según la dirección efectiva alineada. Los 16 bits de $offset$ se suman al contenido de GPR *base* para conformar la dirección efectiva.

Restricciones: La dirección efectiva debe estar alineada. Si el bit menos significativo de la dirección no es cero, ocurre un error de dirección.

Operación:

$vAddr \leftarrow sign\_extend(offset) + GPR[base]$

if $vAddr_0 \neq 0$ then

$SignalException(AddressError)$

endif

$(pAddr, CCA) \leftarrow AddressTranslation(vAddr, DATA, STORE)$

$pAddr \leftarrow pAddr_{PSIZE-1:2} \mid\mid (pAddr_{1:0} \; XOR \; (ReverseEndian \mid\mid 0))$

$bytesel \leftarrow vAddr_{1:0} \; XOR \; (BigEndianCPU \mid\mid 0)$

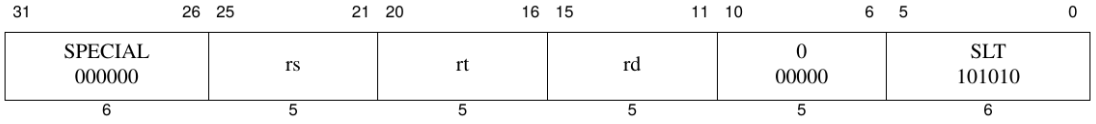
$dataword \leftarrow GPR[rt]_{31-8*bytesel1:0} \mid\mid 0^{8*bytesel}$

$StoreMemory(CCA, HALFWORD, dataword, pAddr, vAddr, DATA)$

Excepciones: TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch.

Notas de programación: Ninguna.

SLT



Formato: SLT rd, rs, rt

Propósito: Activar ante menor. Registrar el resultado de una comparación.

Descripción: $GPR[rd] \leftarrow (GPR[rs] < GPR[rt])$

Compara los contenidos de GPR rs y GPR rt como enteros con signo y registra el resultado de forma Booleana en GPR rd. Si GPR rs es menor que GPR rt, el resultado es 1 (verdadero) o de lo contrario 0 (falso).

La comparación aritmética no causa excepciones tipo *Integer Overflow*.

Restricciones: Ninguna.

Operación:

if $GPR[rs] < GPR[rt]$ then

$GPR[rd] \leftarrow 0^{GPRLEN-1} \,||\, 1$

else

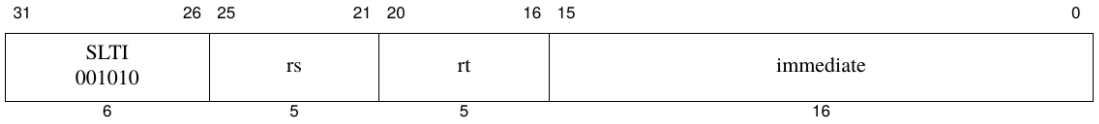
$GPR[rd] \leftarrow 0^{GPRLEN}$

endif

Excepciones: Ninguna.

Notas de programación: Ninguna.

SLTI



Formato: SLTI rt, rs, inmediato

Propósito: Activar ante menor con inmediato. Registrar el resultado de una comparación con una constante.

Descripción: $GPR[rt] \leftarrow (GPR[rs] < inmediato)$

Compara los contenidos de GPR rs y el inmediato con signo de 16 bits como enteros con signo y registra el resultado de forma Booleana en GPR rt. Si GPR rs es menor que el inmediato, el resultado es 1 (verdadero) o de lo contrario 0 (falso).

La comparación aritmética no causa excepciones tipo *Integer Overflow*.

Restricciones: Ninguna.

Operación:

if $GPR[rs] < sign\_extend(inmediato)$ then

$GPR[rt] \leftarrow 0^{GPRLEN-1} \,||\, 1$

else

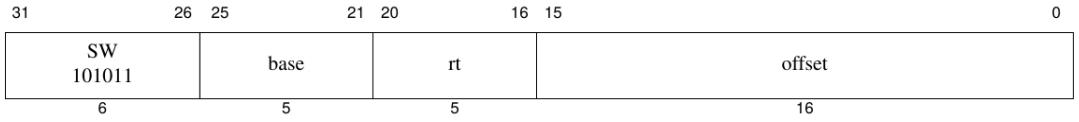
$GPR[rt] \leftarrow 0^{GPRLEN}$

endif

Excepciones: Ninguna.

Notas de programación: Ninguna.

SW



Formato: SW rt, offset(base)

Propósito: Almacenar palabra. Almacena media palabra (16 bits) en memoria.

Descripción: $memory[GPR[base] + offset] \leftarrow GPR[rt]$

Los 32 bits de GPR[rt] son almacenados en memoria según la dirección efectiva alineada. Los 16 bits de $offset$ se suman al contenido de GPR *base* para conformar la dirección efectiva.

Restricciones: La dirección efectiva debe estar alineada. Si alguno de los 2 bits menos significativos de la dirección no es cero, ocurre un error de dirección.

Operación:

$vAddr \leftarrow sign\_extend(offset) + GPR[base]$

if $vAddr_0 \neq 0^2$ then

$SignalException(AddressError)$

endif

$(pAddr, CCA) \leftarrow AddressTranslation(vAddr, DATA, STORE)$

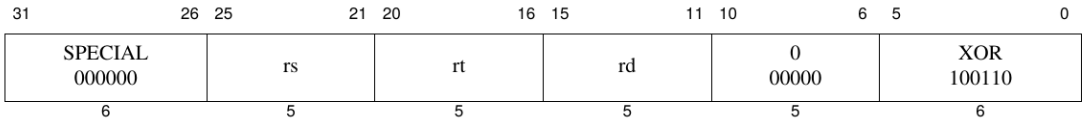
$dataword \leftarrow GPR[rt]$

$StoreMemory(CCA, WORD, dataword, pAddr, vAddr, DATA)$

Excepciones: TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch.

Notas de programación: Ninguna.

XOR



Formato: XOR *rd*, *rs*, *rt*

Propósito: Unión exclusiva. Realiza una unión exclusiva bit a bit.

Descripción: $GPR[rd] \leftarrow GPR[rs]\ XOR\ GPR[rt]$

Combina los contenidos de GPR *rs* y GPR *rt* como la unión lógica exclusiva bit a bit y almacena el resultado en GPR *rd*.

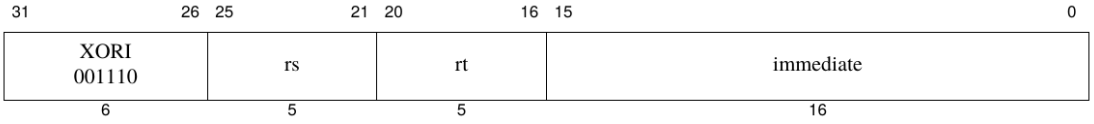
Restricciones: Ninguna.

Operación: $GPR[rd] \leftarrow GPR[rs]\ XOR\ GPR[rt]$.

Excepciones: Ninguna.

Notas de programación: Ninguna.

XORI



Formato: XORI *rt*, *rs*, inmediato

Propósito: Unión exclusiva con inmediato. Realiza una unión exclusiva bit a bit con una constante.

Descripción: $GPR[rt] \leftarrow GPR[rs]\ XOR\ inmediato$

Combina los contenidos de GPR *rs* y el inmediato de 16 bit con extensión de ceros como la unión lógica exclusiva bit a bit y almacena el resultado en GPR *rt*.

Restricciones: Ninguna.

Operación: $GPR[rd] \leftarrow GPR[rs]\ XOR\ zero\_extend(inmediato)$.

Excepciones: Ninguna.

Notas de programación: Ninguna.

Apéndice C

Códigos en lenguaje ensamblador MIPS

Este apéndice contiene los dos códigos en ensamblador MIPS utilizados para las pruebas de verificación de funcionamiento y comparación de desempeño.

C.1. Código ensamblador para verificación del sistema

```
1  LW   $t2, 512($zero) #Carga en el registro $t2 la palabra [128] de memoria.
2  LW   $t3, 528($zero) #Carga en el registro $t3 la palabra [132] de memoria.
3  ADD  $s0, $t2, $t3   #Guarda en $s0 el resultado de sumar $t2 y $t3.
4  SUB  $s0, $t2, $t3   #Guarda en $s0 el resultado de restar $t2 y $t3.
5  AND  $s0, $t2, $t3   #Guarda en $s0 el resultado de and entre $t2 y $t3.
6  OR   $s0, $t2, $t3   #Guarda en $s0 el resultado de or entre $t2 y $t3.
7  BEQ  $t2, $t3, 3     #Salta a dir 10 (PC=7+3) si $t2=$t3. No salta.
8  LW   $t3, 512($zero) #Carga en el registro $t3 la palabra [512] de memoria.
9  BEQ  $t2, $t3, 5     #Salta a dir 14 (PC=9+5) si $t2=$t3. Salta.
10 ADDI $t2, $t2, 1     #$t2 <= $t2+1. No deberia ser ejecutada por el BEQ anterior.
11 ADDI $t2, $t2, 1     #$t2 <= $t2+1. No deberia ser ejecutada por el BEQ anterior.
12 ADDI $t2, $t2, 1     #$t2 <= $t2+1. No deberia ser ejecutada por el BEQ anterior.
13 ADDI $t2, $t2, 1     #$t2 <= $t2+1. No deberia ser ejecutada por el BEQ anterior.
14 ADDI $t2, $t2, 1     #$t2 <= $t2+1. No deberia ser ejecutada por el BEQ anterior.
15 SW   $t3, 532($zero) #Escribe el dato de $t2 en la palabra [133]de memoria.
16 ADDI $t2, $t2, 1     #$t2 <= $t2+1.
17 ADDI $t3, $t3, -1    #$t3 <= $t3-1.
18 ADD  $t2, $t3, $zero #Como $zero = 0 entonces $t2 <= $t3.
19 BNE  $t2, $t3, 3     #Salta 3 direcciones si t2 != t3. No salta.
20 SLT  $t2, $zero, $t3 #Si $zero < $t3 entonces $t2 <= 1.
21 SLT  $t3, $t2, $zero #Si $t2 < $zero entonces $t3 <= 1. No es, y $t3 <= 0.
22 BNE  $t2, $t3, 3     #Salta 3 direcciones si t2 != t3. Salta.
23 ADDI $t2, $t2, 1     #$t2 <= $t2+1. No deberia ser ejecutada por el BNE anterior.
24 ADDI $t2, $t2, 1     #$t2 <= $t2+1. No deberia ser ejecutada por el BNE anterior.
25 ADDI $t2, $t2, 1     #$t2 <= $t2+1. No deberia ser ejecutada por el BNE anterior.
26 XOR  $s0, $t2, $t3   #Guarda en $s0 la operacion $t2 XOR $t3
27 NOR  $s0, $t2, $t3   #Guarda en $s0 la operacion $t2 NOR $t3
28 SLTI $t2, $t3, -5    #Si t3 < -5, t2 <= 1. Sino t2 < =0.
29 SLTI $t2, $t3, 5     #Si t3 < 5, t2 <= 1.
30 LUI  $t2, 65535      #$t2 <= FFFF00000.
31 ORI  $t2, $s0, 65535 #$t2 <= FFFFFFFF.
32 ANDI $t2, $s0, 2     #$t2 <= 000000002.
33 XORI $t2, $s0, 65535 #$t2 <= FFFFF0001.
34 LW   $t2, 516($zero) #$t2 <= FFFF00FF.
35 SB   $t2, 532($zero) #Almacena en el byte0 de la palabra [133], el byte0 de $t2.
36 SB   $t2, 535($zero) #Almacena en el byte3 de la palabra [133], el byte0 de $t2.
37 SH   $t2, 535($zero) #Almacena en la hword1 de la palabra [133], la hword0 de $t2.
38 LB   $t3, 534($zero) #Carga del byte2 de la palabra [133] en $t3.
39 LB   $t3, 533($zero) #Carga del byte1 de la palabra [133] en $t3.
40 SB   $t2, 533($zero) #Carga del byte0 de $t2 al byte1 de la palabra [133].
41 LH   $t3, 533($zero) #Carga de la hword0 de la palabra [133] en $t3.
42 LH   $t3, 534($zero) #Carga de la hword1 de la palabra [133] en $t3.
43 J    0               #Saltar y volver al  inicio de la memoria.
```

C.2. Código ensamblador para ordenamiento por burbujeo

```
1  LW    $t1, 528($zero)          # $t1 = N
2  ADDI  $t1, $t1, -1             # $t1 = N-1 (para outer loop)
3  LW    $t0, 532($zero)          # $t0 = dir donde comienza el arreglo
4  NEXTPASS: ADD $t2, $t1, $zero # $t2 <= $t1
5    ADD $t3, $t0, $zero          # $t3 <= $t0 = dir, para comenzar a recorrer el
6                                 # arreglo
7  NEXTCOMP: LW   $t4, 0($t3)    # $t4 <= valor i-esimo del arreglo
8    ADDI $t3, $t3, 4             # $t3 <= $t3 + 4 para apuntar el siguiente elemento
9    LW   $t5, 0($t3)             # $t5 <= valor i+1-esimo del arreglo
10   SLT  $t6, $t4, $t5           # Si $t4 < $t5 entonces $t6 <= 1
11   BNE  $t6, $zero, NOXCHG      # Si $t6 != 0 se salta a NOXCHG porque esta
12                                # ordenado
13   BEQ  $t4, $t5, NOXCHG        # Si $t4 = $t5 se salta a NOXCHG porque esta
14                                # ordenado
15   SW   $t4, 0($t3)             # Se ordena en memoria correctamente el orden
16                                # (menor a mayor) de los elementos
17   ADDI $t3, $t3, -4
18   SW   $t5, 0($t3)
19   ADDI $t3, $t3, 4
20 NOXCHG: ADDI $t2, $t2, -1      # Se decrementa el contador del ciclo interno $t2
21   BNE  $t2, $zero, NEXTCOMP    # Si aun $t2 != 0 se continua comparando a lo largo
22                                # del arreglo
23   ADDI $t1, $t1, -1            # Si se compararon todos los elementos se decrementa
24                                # el contador externo $t1
25   BNE  $t1, $zero, NEXTPASS    # Si aun $t1 != 0 se continua con una nueva peinada
26                                # del arreglo. Si $t1 = 0 se termina de ordenar.
```

Bibliografía

ARM Ltd. (2006). *AMBA 3 AHB-Lite Protocol v1.0 - Specification.*

Baer, J. (2010). *Microprocessor Architecture, From Simple Pipelines to Chip Multiprocessors.* New York, NY, USA.: Cambridge University Press.

Casillo, L. e I. Saraiva Silva (2012). "A methodology to adapt datapath architectures to a MIPS-1 model". En: *Brazilian Symposium on Computing System Engineering.*

Hennessy, J. L. y D. A. Patterson (2012). *Computer Architecture, A Quantitave Approach.* 5.ª ed. 255 Wyman Street, Waltham, MA, USA: Elsevier.

Mathaikutty, D. y S. Shukla (2009). *Metamodeling-Driven IP Reuse for SoC Integration and Microprocessor Design.* 685 Canton Street, Noorwood, MA, USA: Artech House.

MIPS Technologies Inc. (2011a). *MIPS32 Architecture Vol. I - Introduction to the MIPS32 Architecture.* Sunnyvale, CA: MIPS Technologies Inc.

MIPS Technologies Inc. (2011b). *MIPS32 Architecture Vol. II - The MIPS32 Instruction Set.* Sunnyvale, CA: MIPS Technologies Inc.

Money Harris, D. y S. Harris (2013). *Computer Architecture.* 2.ª ed. 255 Wyman Street, Waltham, MA, USA: Elsevier.

Moore, G. (1965). "Cramming More Components onto Integrated Circuits". En: *IEEE Electronics* 38.8, págs. 114-117.

Morris, K. (2006). "The Spirit of Standarization: IP Re-Use Takes Center Stage". En: *IC Design and Verification Journal.*

Neeraj, J. (2012). "VLSI Design and Optimized Implementation of a MIPS RISC Processor using XILINX Tool". En: *International Journal of Advanced Research in Computer Science and Electronics Engineer (IJARCSEE)* 1.10, págs. 52-56.

Pasricha, S. y N. Dutt (2008). *On-Chip Communication Architectures, System on Chip Interconnect.* 30 Corporative Drive, Suite 400, Burlington, MA, USA: Elsevier.

Patterson, D. A. y J. L. Hennessy (2012). *Computer Organization and Design, the hardware/software interface*. 4.ª ed. Waltham, MA, USA: Elsevier.

Qi, A., J. Guojie, S. Wen, C. Songsong y C. Shuai (2014). "Optimizing Memory Acces with Fast Address Computation on a MIPS Architecture". En: *9th IEEE International Conference on Networking, Architecture and Storage*.

Rama Krishna, V. y B. Venu Gopal (2012). "Design and analysis of 32-bit RISC Processor based on MIPS". En: *International Journal of Innovative Technology and Exploring Engineering (IJITEE))* 1.5, págs. 17-21.

Semiconductor Industry Association (2015). *More Moore Scaling: Opportunities and Inflection Points*. ITRS.

Strolenberg, C. (2005). *Hard IP Offers Some Hard-Core Values*. Available from: https://www.design-reuse.com/articles/2552/socs-design-tools-hard-ip-offers-some-hard-core-values.html [Accessed 09/15/2017]: [WWW] D&R Industry Articles.

Sutherland, S. (2007). *Verilog HDL Quick Reference Guide, based on the Verilog-2001 standard*. 22805 SW 92 PLace Tualatin, OR, USA: Sutherland HDL.

Sweetman, D. (2006). *See MIPS Run*. 255 Wyman Street, Waltham, MA, USA: Elsevier.

Werner, K. (2005). *IP Reuse Gets a Reality Check*. Available from: http://chipdesignmag.com/display.ph [Accessed 09/20/2017]: [WWW] Chip Design.

YunZhu, X. y D. YueHua (2008). "Instruction Decoder Module Design of 32-bit RISC CPU Based on MIPS". En: *Second International Conference on Genetic and Evolutionary Computing*.

Zulkifli, M., Y. Yudhanto, N. Soetharyo y T. Adiono (2009). "Reduced Stall MIPS Architecture using Pre-Fetching Accelerator". En: *International Conference on Electrical Engineering and Informatics*.