Sistemas Informáticos

Curso 2009 - 2010

Simulador del procesador MIPS sobre el formalismo DEVS

Alumnos:
Francisco Alejandro Calvo Valdés
José Félix Roldán Ramírez
Alfonso San Miguel Sánchez

Director de proyecto:
José Luis Risco Martin

Facultad de Informática

Universidad Complutense de Madrid

Autorización

Autorizamos a la Universidad Complutense de Madrid a utilizar y/o difundir con fines académicos y no comerciales, siempre mencionando expresamente a sus autores, tanto la propia memoria como el código, la documentación y/o el prototipo desarrollado.

Francisco Alejandro Calvo Valdés

José Félix Roldán Ramírez

Alfonso San Miguel Sánchez

Agradecimientos

Queremos agradecer de manera especial a nuestro profesor José Luis Risco Martín por su apoyo y ayuda a lo largo del último año. Agradecer también a nuestros familiares y amigos por su comprensión y ayuda en todo momento.

Queremos también agradecer a los Doctores Yu Chen, de Hewlett Packard, Hessam S. Sarjoughian de la Universidad de Arizona y Gabriel Wainer de la Universidad de Carleton su ayuda desinteresada proporcionándonos sus artículos científicos relativos a implementaciones prácticas en DEVS.

Palabras clave

Modelización y simulación, sistemas de eventos discretos, repertorio de instrucciones, diseño del procesador, DEVS, MIPS.

Resumen

Ciertos conceptos relativos a los procesadores, tales como detalles de implementación, análisis de rendimiento, consumo de energía y fiabilidad son fundamentales en los cursos orientados a arquitectura de computadores. El procesador MIPS (del inglés Microprocessor without Interlocked Pipeline Stages) se utiliza actualmente en muchas Universidades para enseñar estas materias.

En este proyecto presentamos un simulador del procesador MIPS, que facilitará la enseñanza de este procesador en cursos de arquitectura de computadores. Desarrollamos varios modelos del procesador basado en un ciclo, en varios, o en un cauce segmentado. Los modelos se construyen de acuerdo a una especificación formal denominada DEVS (del inglés Discrete EVent Systems specification). Para ello definimos una colección elemental de modelos combinacionales y secuenciales, que se combinan para formar el procesador final. Gracias al uso de un compilador cruzado, se puede importar código escrito en c y traducirlo a lenguaje ensamblador. El simulador recibe como entrada este código, permitiendo analizar el comportamiento interno del procesador, el estado de los módulos y el valor de las señales de control en cada momento de la ejecución. Para facilitar esta tarea elaboramos una interfaz gráfica que nos permite visualizar los resultados de la ejecución, con el objetivo de comparar las diversas implementaciones del MIPS.

There are certain concepts about processors, like implementation details, performance analisys, energy consumption and reliability which are fundamental in all learning courses related to computer architecture. The MIPS processor (Microprocessor without Interlocked Pipeline Stages) is being used as a basis to teach these courses.

In this proyect we proudly present a full simulator of the MIPS processor architecture, which will serve as a strong support for all kind of computer architecture learning courses. We have developed three versions of the model: monocycle, multicycle and pipelined. These models have been built using DEVS (Discrete EVent Systems specification). With this purpose on mind we have defined a basic set of sequential and combinational models that combine themselves to set up the final processor. By using a cross compiler, we can import programs devoloped in c language and generate a binary file including the corresponding Assemby Language code. Our simulator receives these binaries as input to execute programs, allowing us to check and analyze the inner behaviour of the processor, the state of the sequential and combinational models, and values for all the Control Unit signals at anytime in the execution process. To simplify this task, we have made a GUI (Graphic User Interface) which logs the results of every instruction, so we can compare them to study the differences between each MIPS implementation.

Índice

1. Introducción y trabajo relacionado

1.1. Introducción

Según unos estudios recientes la ciencia de la simulación puede ser considerada como un paradigma complementario a la ciencia experimental [8] . En dicho estudio se identifica la simulación como parte necesaria para la conceptualización, especificación y desarrollo de sistemas complejos. Aparte de la investigación y el desarrollo, la simulación también juega un papel importante en la enseñanza. Por ejemplo en muchos departamentos en facultades de Informática o diversas ingenierías, los simuladores son usados para la enseñanza de los fundamentos de las arquitecturas de computadores. Si comparamos la simulación con experimentos reales usando hardware, la simulación posee muchas más ventajas (facilidad de uso, de configuración y de modificación) además de tener un coste mucho menor. Si consideramos también los métodos tradicionales de enseñanza y nos fijamos en los materiales que usan tales como esquemáticos, diagramas y descripciones de texto, la simulación puede mejorar considerablemente el aprendizaje de los estudiantes así como servir de ayuda a la labor docente del profesor debido a la capacidad del sistema para la visualización de resultados.

Cuando usamos la simulación como un medio para entender, analizar y estudiar el comportamiento de un sistema, es importante tener un entorno de simulación y modelado adecuado. Con este entorno la estructura y el comportamiento del sistema puede ser exactamente formulado como un conjunto de modelos de simulación, pudiendo ser posteriormente ejecutados y analizados sus resultados. Es de gran ayuda tener in entorno que implemente el marco de simulación y modelado y pueda soportar la especificación del modelo, la ejecución de la simulación y la interacción con el usuario. Es también conveniente para el entorno que soporte la descripción de modelos tanto de manera lógica como en tiempo real y *permita a los modelos ejecutarse en una o varias máquinas*. Con un entorno como este los modeladores pueden centrarse en crear las especificaciones de los modelos y realizar experimentos de simulación para verificar y validar el modelo sin gastar el tiempo implementado rutinas de simulación. En el contexto de la educación, es extremadamente útil para tal ambiente el proveer al entorno de un método de visualización facilitando el aprendizaje y entendimiento de las a veces complejas arquitecturas hardware.

El objetivo principal de este proyecto es desarrollar un simulador de un procesador MIPS (Microprocessor without Interlocked Pipeline Stages) que pueda ayudar a estudiantes y a profesores en su labor docente. Los simuladores de arquitecturas de computadores existentes están diseñados en diversos niveles de abstracción: componentes hardware, ISAs (instruction Set Architectures) o sistemas que incluyen procesador, memoria, E/S, etc. El ámbito de nuestro proyecto está dentro de los componentes hardware, específicamente procesadores MIPS32. Los simuladores de procesadores MIPS existentes no modelan a nivel de transferencia de segundos (RT level).

La Figura 1 presenta un esquema de los distintos niveles de abstracción a la hora de diseñar un sistema hardware. Nuestro proyecto se mueve dentro del nivel de transferencia de registros.

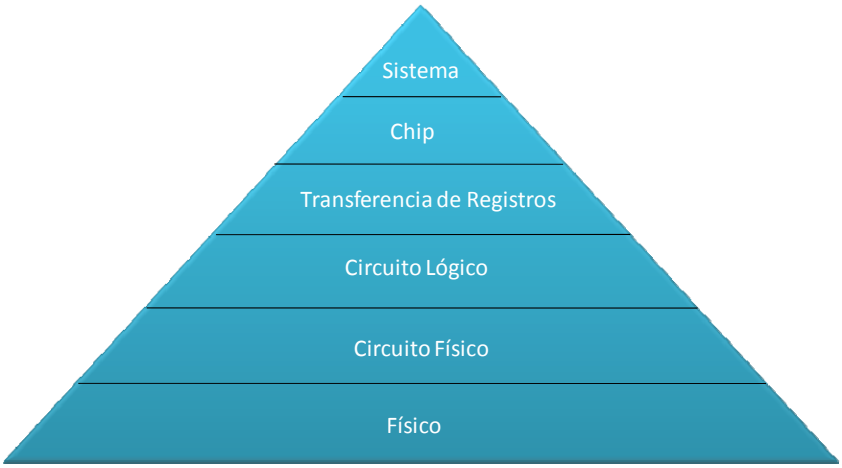


Figura 1. Niveles de abstracción hardware

Nuestro simulador proporciona modelos para el monociclo, multiciclo y el procesador segmentado descritos en profundidad en la Sección 3. Todos estos modelos han sido desarrollados usando DEVS (Discrete Events System Specification) [1]. Han sido varios los simuladores y trabajos anteriores realizados sobre arquitecturas de computadores. Algunos de ellos han sido implementados con lenguajes de programación convencionales como por ejemplo el simulador WinMIPS64 [9] y aunque también se han realizado simuladores de arquitecturas en DEVS, como por ejemplo el ALFA-1 del procesador ALFA [10] incluso simuladores MIPS en otros formalismos como VERILOG o VHDL. Sin embargo este proyecto ha sido íntegramente desarrollado siguiendo el formalismo DEVS. Se ha escogido como formalismo DEVS porque aparte de cumplir todos los requisitos relevantes en cuanto a técnicas de simulación y modelado, es muy apropiado para modelar sistemas de arquitecturas de computadores.

1.2. Trabajo relacionado

El repertorio de instrucciones MIPS ISA es ampliamente usado en la actualidad en sistemas integrados. Pertenece a la familia de los procesadores de arquitectura RISC (Reduced Instruction Set Computer) y existen múltiples versiones de él, desde el precursor MIPS I hasta el MIPS V, pasando por el MIPS32 y el MIPS64. MIPS I, MIPS II y MIPS32 soportan registros de 32 bits, mientras que los demás soportan registros de 64 bits.

Debido a su diseño simple ha sido usado para la enseñanza en cursos universitarios de arquitectura de computadores. En la enseñanza de cursos de arquitectura de computadores se ha observado que muchos estudiantes tienen muchas dificultades para entender las diferentes implementaciones de los procesadores y sus diferentes análisis. Aunque existen muchos simuladores sobre arquitecturas de computadores, ninguno de ellos está dirigido específicamente a la labor docente. Uno de los objetivos principales de este proyecto ha sido crear un simulador que sea capaz de ayudar a los estudiantes a entender, analizar y estudiar las diversas implementaciones del procesador MIPS. El simulador ha de cumplir los siguientes requisitos:

A. Debe modelar el procesador MIPS32 en sus variantes monociclo, multiciclo y segmentado.

B. Debe proveer datos estadísticos (CPI, contador de ciclos, etc.) que sirvan para estudiar el rendimiento.
C. Debe tener algún tipo de visualización gráfica para ser capaces de comprender como las señales del procesador cambian de valor (Cronograma).
D. Debe tener la capacidad de poder ser usado en cualquier plataforma independientemente del hardware sobre el que se ejecute.

Antes de comenzar con el diseño de nuestro simulador, primero examinamos las herramientas existentes con el fin de estudiarlas y analizarlas a fondo intentando ver cosas que pudieran servirnos de inspiración y localizar fallos que pudieran tener a fin de crear un proyecto más robusto y algo más completo que los simuladores existentes.

Destacamos a continuación en una pequeña comparativa, las diferencias más significativas entre nuestro simulador y otros de los simuladores MIPS más importantes de la actualidad.

Simulador	A	B	C	D	Implementación
WinMIPS64	N	S	N	N	C++
EduMIPS64	N	S	N	S	Java
SimpleMIPSPipeline	N	N	N	N	HASE
MiniMIPS	S	N	N	N	C
WebMIPS	N	N	N	S	ASP, html
ProcessorSim	N	N	S	S	Java, XML
Proyecto	S	S	S	S	Java, XDEVS

Figura 2. Tabla comparativa de los distintos simuladores.

Como puede observarse en la tabla de la Figura 2 hemos realizado una comparación de los simuladores MIPS de la actualidad. Hemos indicado si cumplen los requisitos especificados más arriba, indicando S (si) o N (no) si cumplen las especificaciones. Además resaltamos en cada una el lenguaje con el que se ha implementado.

WinMIPS64 [11], EduMIPS64 [12] y SimpleMIPSPipeline [13] son simuladores orientados a procesadores pipeline, están centrados en modelar los aspectos más formales de las etapas del pipeline, sin prestar atención a las demás versiones MIPS, monociclo y multiciclo. Nuestro simulador trata todas las versiones MIPS, desde la más básica monociclo, complicándola poco a poco haciéndola más compleja hasta llegar al modelo segmentado. EduMIPS64 es una reedición en Java de WinMIPS64. MiniMIPS [14] tiene un objetivo similar al de nuestro proyecto aunque el modelado de sus componentes (unidad de control, memoria, ALU, multiplexores, etc.) están por encima del nivel de atracción hardware de RT level, como puede verse en la Figura 1. Una de las características de nuestro simulador es que es capaz de abstraerse hasta ese nivel, cosa que ningún simulador hasta el momento había hecho. Además en el simulador MiniMIPS algunos de sus atributos tales como el retardo (necesario para analizar el rendimiento) no pueden ser modificados, no posee ninguna animación para ver resultados sólo un pequeño contador de ciclos, al estar implementado en C, requiere una máquina UNIX para funcionar. WebMIPS [15] sólo trata la ruta segmentada, modela todos los componentes pudiendo el usuario observar las salidas y entradas de cada módulo viendo su valor en un instante de tiempo, sin embargo no es posible observar como estas señales van cambiando durante la ejecución. En cuanto al estudio del rendimiento proporciona un contador de ciclos. Por último el simulador ProcessorSim [16] puede ser configurado para

cualquier configuración del MIPS, proporciona una animación que muestra como las instrucciones se ejecutan dentro del procesador y como cada componente de la ruta de datos va cambiando en cada ciclo de ejecución, esto es de gran ayuda para los estudiantes. Por el contrario este simulador no posee retardos, algo que sí hemos tratado en nuestro simulador.

Nuestro simulador ha tratado de aunar en uno sólo todas necesidades que pueden surgirle a los estudiantes o a los profesores en su labor docente.

2. Contribuciones del proyecto

Como se ha explicado en detalle en la Sección 1.2, nuestro proyecto consiste en un simulador que pretende servir de base para la enseñanza de asignaturas de arquitecturas de computadores.

El simulador pretende ayudar al estudiante a comprender mejor el uso y funcionamiento de las rutas de datos más importantes: monociclo, multiciclo y segmentada. Pretende también mediantes las herramientas visuales servir de apoyo al estudio y el entendimiento del procesador MIPS, uno de los más usados en el ámbito académico hoy en día. Muchos de los conceptos relativos a los procesadores, como su implementación, análisis de la eficiencia, fiabilidad o consumo son indispensables en cursos académicos dónde se enseñan asignaturas relativas a la arquitectura de los computadores.

A continuación vamos a presentar el entorno desarrollado y el funcionamiento del simulador así como los módulos que lo componen y cómo interactúan entre ellos.

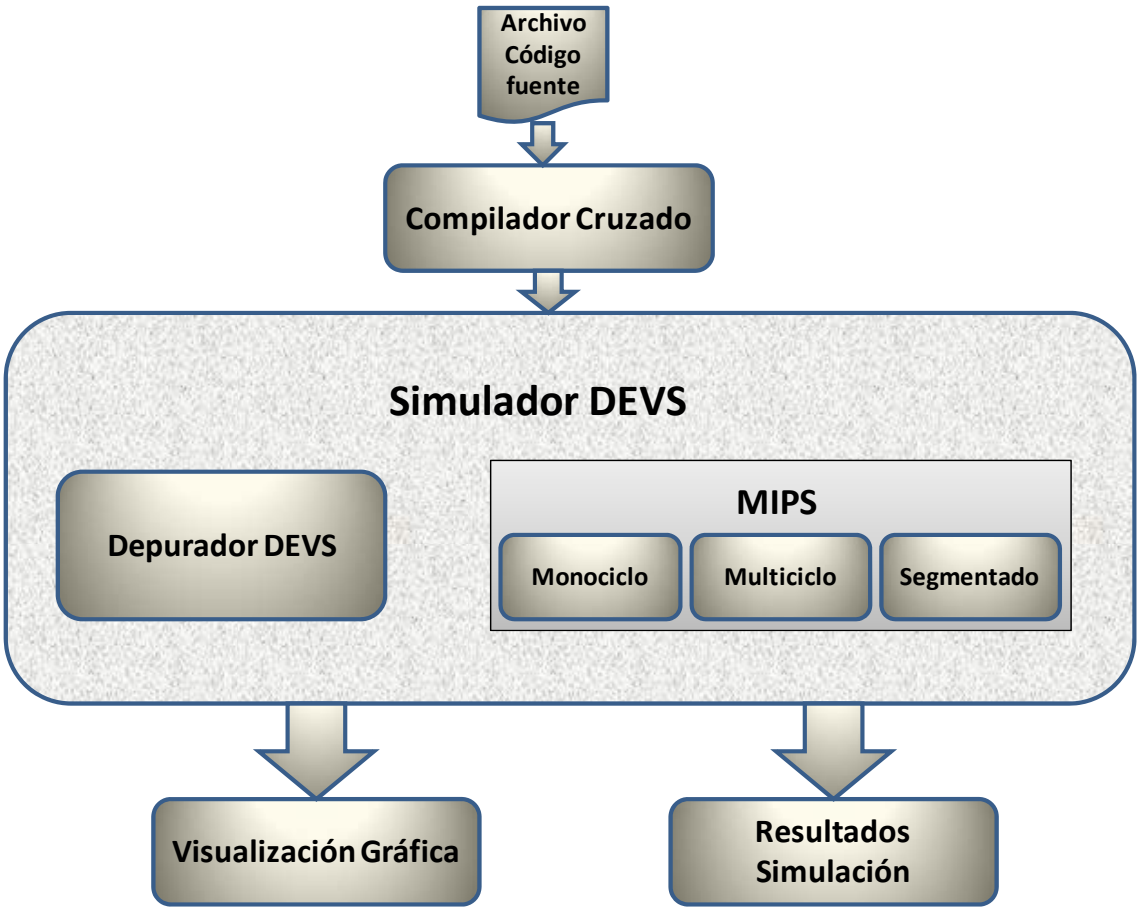


Figura 3. Entorno de simulación

Como se pude ver en la Figura 3 se presenta el entorno que ha sido desarrollado, así como las partes de las que consta y las interconexiones existentes entre los módulos, que se detallan a continuación.

Para probar el simulador ha de disponerse de un código fuente escrito en lenguaje C, cuyo código sea correcto y no presente errores léxicos ni sintácticos. A continuación se introduce este código en el compilador cruzado que hemos usado para la realización del proyecto, el compilador Cygwin, que puede verse con más detalle en el Apéndice I. El compilador cruzado genera archivos binarios a partir del código fuente en C y los transforma a lenguaje ensamblador que reconoce el MIPS32. En las pruebas realizadas se han generado archivos binarios de tres algoritmos ampliamente conocidos que han sido usados como benchmarks de nuestro proyecto: generación del término n-ésimo de la serie de Fibonacci, cálculo del máximo común de dos números y cálculo del producto escalar de dos vectores. Una vez transformado el código a formato ensamblador ya está preparado para entrar en el simulador. El archivo binario se ejecuta sobre la plataforma MIPS deseada (monociclo, multiciclo o segmentada) dentro del simulador DEVS. Como puede observarse en la Figura 3 dentro del simulador existe un módulo llamado depurador, este módulo ha sido introducido con el objetivo de poder depurar el modelo DEVS. Este modelo a pesar de no ser muy complejo es bastante complicado de depurar con métodos tradicionales de depuración, de esta dificultad surge la necesidad de crear un módulo que no existía previamente, a pesar de haber muchos simuladores DEVS ya desarrollados. Una vez que el archivo binario ha corrido dentro de la plataforma, se han desarrollado dos módulos para analizar y estudiar la situación de los resultados de la simulación. Por una parte existe un módulo que permite una visualización gráfica de los resultados obtenidos, siendo de gran utilidad para el estudiante en su labor de aprendizaje poder ver los resultados de manera interactiva. Existe además un módulo que analiza los resultados de la simulación proporcionando estadísticas de la ejecución del procesador tales como CPI, numero de ciclos, tipos de instrucciones ejecutadas, etc.

El conjunto de los módulos que forman el simulador constituye un gran avance, debido a que proporciona una visión académica y educativa desde un punto de vista no abordado hasta ahora. A pesar de que en la actualidad existen múltiples simuladores tanto de MIPS, como sobre DEVS, ninguno ha sido enfocado para la educación como el nuestro y aunque existen algunos simuladores de gran calidad, no consiguen acercarse a las necesidades del estudiante de asignaturas de arquitectura de computadores.

3. Definición del procesador MIPS32

3.1. Introducción

Como se ha descrito en secciones previas de esta memoria, el propósito del modelo desarrollado está fundamentalmente orientado a su uso en la Universidad, en cursos relacionados con arquitectura de computadores. Es por ello que no se ha implementado todo el repertorio de instrucciones del MIPS32, sino las más relevantes de la unidad entera. Existe abundante documentación en la literatura referente a este procesador. Cabe destacar el libro de Patterson-Hennessy [4], que es sin duda referencia obligada en todos los cursos de arquitectura de computadores basados en este procesador. Sin embargo, incluso en las diferentes ediciones de este libro (ver ediciones 2, 3 y 4 de la referencia) la documentación del procesador varía ligeramente, adaptando la ruta de datos y el controlador del procesador a los contenidos de cada edición en particular.

En este trabajo utilizamos como base la ruta de datos del MIPS32 utilizada en las diferentes asignaturas de la Facultad de Informática de la Universidad Complutense de Madrid, como por ejemplo Ampliación de Estructura de Computadores (tercer curso de ingeniería informática), y Estructura y Tecnología de Computadores (segundo curso de Ingeniería Técnica en Informática de Sistemas y el mismo curso en Ingeniería Técnica en Informática de Gestión).

El presente capítulo describe y amplía de forma fundamentalmente teórica la ruta de datos utilizada en estas asignaturas (y prácticamente idéntica a la documentada en [4]). Esta ruta de datos contiene básicamente cinco instrucciones: add, sub, beq, lw y sw. En este proyecto incorporamos las instrucciones and, or, slt, addiu, slti, j y jr, además del soporte a las denominadas pseudoinstrucciones. Con ello podremos simular prácticamente cualquier programa binario con aritmética entera.

3.2. Descripción general

Con el nombre **MIPS (Microprocessor *without Interlocked Pipeline Stages)*** se denomina a una familia de procesadores RISC desarrollada por MIPS computer systems.

Estos procesadores fueron ideados por **John L. Hennessy** y su equipo en la universidad de Stanford en 1981. La idea que utilizaron para llevar a cabo el diseño del procesador era mejorar drásticamente el rendimiento del procesador empleando segmentación, que por aquella época era una técnica muy conocida pero muy difícil de implementar. Otra de las cosas que trataron de introducir en el MIPS fueron los bloqueos del procesador, que posibilitaban que una instrucción que tardase varios ciclos en completar su ejecución dejase de cargar datos de los registros de segmentación. Pero los bloqueos podían ser largos y eso suponía una gran barrera para mejorar el rendimiento del procesador, así que decidieron que todas las etapas de una instrucción tenían que tardar un único ciclo en completarse. Esto hacia mucho más largas algunas operaciones como la multiplicación y la división, y desechaba muchas instrucciones útiles, pero en conjunto resultaba una mejora cuantiosa en el rendimiento.

Debido a estas ideas tan lejos de la concepción de los procesadores hasta el momento hubo bastante polémica alrededor de este procesador, y muchos especialistas tacharon este proyecto de demasiado ambicioso y pronosticaron que nunca conseguirían alcanzar los objetivos propuestos.

En la actualidad estos procesadores se emplean a menudo para la fabricación se sistemas empotrados, en dispositivos para Windows CE, en routers CISCO, en videoconsolas, etc. Ha pasado por muchas revisiones que han aumentado su rendimiento:(tamaño de memoria, velocidad de reloj…) desde el MIPS y el MIPS V hasta las actuales MIPS32 y MIPS64, que trabajan con 32 y 64 bits respectivamente. Incluso se ha logrado implementar una versión de MIPS **superescalar** (R800 en 1994) que es capaz de ejecutar 2 instrucciones de memoria y de la ALU en un solo ciclo.

Este procesador forma parte de la familia **RISC *(Reduced Instruction Set Computer)*** de procesadores. Es un tipo de microprocesador con las siguientes características fundamentales:

- Instrucciones de tamaños fijos y presentados en un reducido número de formatos.
- Sólo las instrucciones de carga y almacenamiento acceden a la memoria por datos.

Además estos procesadores suelen disponer de muchos registros de propósito general para ser empleados en la ejecución, minimizando de esta manera los accesos a memoria cada vez que se necesita un dato.

El objetivo de diseñar máquinas con esta arquitectura es posibilitar la ejecución y el paralelismo en la ejecución de instrucciones y reducir los accesos a memoria para aumentar el rendimiento y la velocidad de ejecución. Las máquinas RISC protagonizan la tendencia actual de construcción de microprocesadores. PowerPC, DEC y Alpha son ejemplos de algunos de ellos.

Además, debido al repertorio simple y claro de instrucciones, el MIPS se emplea a menudo como medio a través del cual enseñar Arquitectura de computadores en escuelas técnicas y universidades. De hecho, uno de los fines de este proyecto de Sistemas Informáticos es la creación de un programa que permita a los profesores que impartan asignaturas o cursos relacionados con este tema mostrar a sus alumnos el funcionamiento de este procesador, de una manera desglosada y simple, y que ellos mismos por su cuenta tengan la oportunidad de usar esta herramienta como complemento a sus estudios.

3.3. Funcionamiento

Para el desarrollo de la memoria, vamos a analizar a fondo el funcionamiento del procesador en sus 3 versiones más básicas de la ruta de datos, incorporando en el proceso las instrucciones implementadas en el simulador. Estas versiones son:

- **Monociclo**: Un ciclo por instrucción y tiempo de ciclo largo
- **Multiciclo**: Varios ciclos por instrucción y tiempo de ciclo corto.
- **Segmentado**: Varios ciclos por instrucción , tiempo de ciclo corto y ejecución de varias instrucciones de forma simultánea.

3.4. Formato de instrucción máquina

Todas las instrucciones del repertorio del MIPS tienen 32 bits de anchura y son de 3 tipos: (1) *instrucciones aritméticas*, que como su propio nombre indica, se encargan de realizar operaciones aritméticas, (2) *instrucciones con referencia a memoria*, que sirven para

almacenar datos en memoria o extraer datos de la misma, y (3) *instrucciones de salto*, que sirven para realizar bifurcaciones en el programa.

La Figura 4 representa los formatos de las 3 instrucciones anteriormente descritas, indicando cuántos bits corresponden a cada campo. Los campos de la instrucción son por tanto:

- **op:** identificador de instrucción
- **rs,rt,rd:** identificadores registros fuente/destino
- **desp:** cantidad a desplazar (en operaciones de desplazamiento)
- **funct:** selecciona la operación aritmética a realizar
- **dirección:** dirección destino del salto

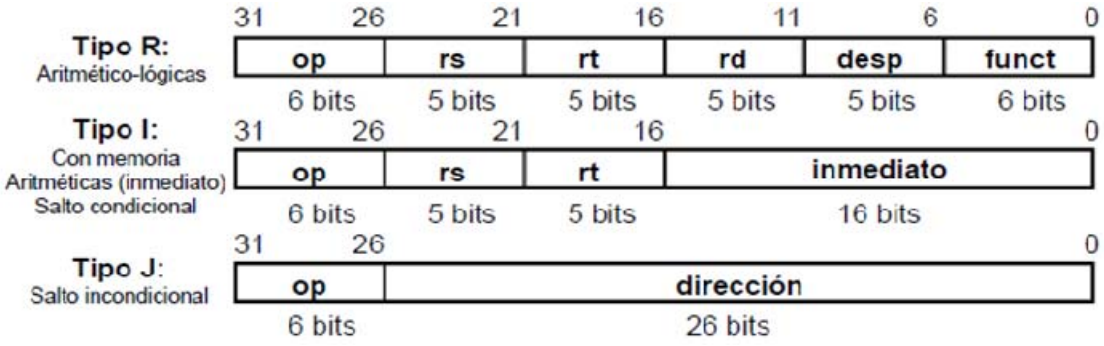


Figura 4. Formatos de instrucción.

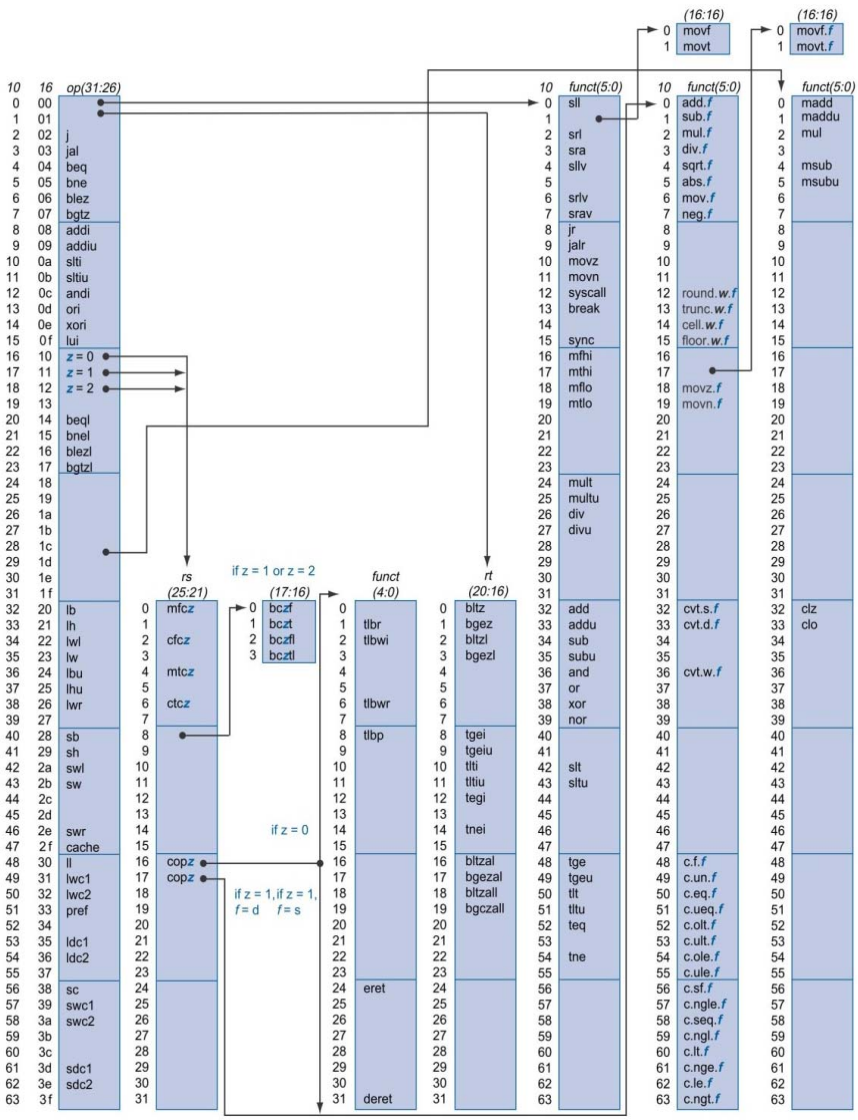


Figura 5. Repertorio de instrucciones MIPS32.

La Figura 5 ilustra el mapa de los códigos de operación de las instrucciones al completo del MIPS32, tomado del libro **Patterson-Hennesy**, del Apéndice A. En este apéndice se recogen todas y cada una de las instrucciones que el MIPS puede ejecutar.

Los valores de cada campo en la figura se muestran a su izquierda. La primera columna muestra los valores en base 10 y la segunda en base 16 para el campo op de la instrucción (bit 31 al 26). Este campo op especifica completamente la operación del MIPS excepto para 6 valores determinados de op : 0,1,16,17,18 y 19. Estas operaciones vienen determinadas por otros campos identificados por punteros.

El último campo (funct) usa “f” para representar “s” si rs=15 y op=17 o op = 17. El segundo campo (rs) utiliza “z” para representar 0,1,2, o 3 si op=16,17,18 y 19, respectivamente.

Si rs = 16, la operación se especifica en otro lugar. Para el caso de z=0, la operación a realizar se especifica en el cuarto campo (bits 4 a 0); Si z = 1, entonces las operaciones lo hacen en el campo con f=s.

Si rs = 17 y z = 1, las operaciones se especifican en el último campo con f=d.

Ejemplo de programa MIPS: fibonacci

Vamos a plantear un ejemplo simple de programa MIPS. Partiremos de un código en c que calcula la sucesión de fibonacci, hasta el término 32, sumando de forma iterativa.

Versión en C

```
int main() {
        int f0, f1, f2, i;
        f0 = 0;
        f1 = 1;
        f2 = 0;
        i = 0;
        for(i=0; i<32; i++) {
                f2 = f1 + f0;
                f0 = f1;
                f1 = f2;
        }
        return 0;
}
```

Tras compilar el código anterior, obtenemos a través del compilador cruzado gcc el archivo binario del procesador MIPS. En la versión MIPS, también aparecen las direcciones de memoria de cada instrucción en hexadecimal.

Archivo binario generado para la arquitectura MIPS32

```
00000000 <main>:

   0:           27bdffe8       addiu          sp,sp,-24
   4:           afbe0010       sw             s8,16(sp)
   8:           03a0f021       move           s8,sp
   c:           afc00000       sw             zero,0(s8)
  10:           24020001       li             v0,1
  14:           afc20004       sw             v0,4(s8)
  18:           afc00008       sw             zero,8(s8)
  1c:           afc0000c       sw             zero,12(s8)
  20:           afc0000c       sw             zero,12(s8)
  24:           8fc2000c       lw             v0,12(s8)
  28:           28420020       slti           v0,v0,32
  2c:           1040000d       beqz           v0,64 <main+0x64>
  30:           00000000       nop
  34:           8fc30004       lw             v1,4(s8)
  38:           8fc20000       lw             v0,0(s8)
  3c:           00621021       addu           v0,v1,v0
  40:           afc20008       sw             v0,8(s8)
  44:           8fc20004       lw             v0,4(s8)
  48:           afc20000       sw             v0,0(s8)
  4c:           8fc20008       lw             v0,8(s8)
  50:           afc20004       sw             v0,4(s8)
  54:           8fc2000c       lw             v0,12(s8)
  58:           24420001       addiu          v0,v0,1
  5c:           08000009       j              24 <main+0x24>
  60:           afc2000c       sw             v0,12(s8)
  64:           00001021       move           v0,zero
  68:           03c0e821       move           sp,s8
```

```
  6c:           8fbe0010        lw              s8,16(sp)
  70:           03e00008        jr              ra
  74:           27bd0018        addiu           sp,sp,24
```

Como podemos observar, en el código aparecen *pseudoinstrucciones* como li, move o beqz. Estas "macros" son traducidas por el procesador al repertorio de instrucciones sobre el que se ejecuta el programa. Sirven para mantener compatibilidad entre distintas familias de procesadores, y surgen debido al limitado conjunto de instrucciones que presenta un procesador tipo RISC.

Hay otras instrucciones especiales, como **nop** (del inglés no operation), traducida a sll $0,$0,0, **break** para romper la secuencia normal de ejecución, y **syscall**, para llamadas al sistema operativo.

3.5. Procesador monociclo

El procesador monociclo presenta tiempos de ciclo largos para la ejecución de las instrucciones. Para el correcto funcionamiento de la ruta de datos, el tiempo de ciclo debe ser igual o mayor al camino crítico que recorran las instrucciones a través del circuito combinacional, de lo contrario obtendríamos resultados inesperados debido a salidas y entradas incorrectas en los módulos.

Este tipo de ejecución es muy lento, e impide la implementación de operaciones con un tiempo de cálculo muy elevado como son la división y la multiplicación y otras instrucciones en punto flotante, además de operaciones con modos complejos de direccionamiento.

La ejecución monociclo obliga a adaptar el hardware, por ejemplo obliga a que la memoria se divida en dos módulos, memoria de datos y de instrucciones, de lo contrario sería imposible completar las lecturas y escrituras necesarias en un solo ciclo.

Temporización monociclo

La idea fundamental en que se sustenta la ruta de datos monociclo es la existencia de dos "barreras" secuenciales entre las cuales se ejecuta toda la carga de trabajo combinacional y que constituyen un ciclo(y por ende, una instrucción), como ilustra la Figura 6. Los únicos módulos secuenciales que aparecen en la ruta son el Registro que actúa de contador de programa(PC), el Banco de Registros y la Memoria de datos. Como podemos observar en la Figura 6, los elementos secuenciales de la ruta han de funcionar de forma coordinada (uno en cada flanco) para conseguir realizar todo el trabajo necesario en un solo ciclo. Los datos fluyen a través de los modulos combinacionales (sumadores, ALU…) calculando los valores correctos de las señales, siempre controlados por las señales que produce la Unidad de Control que regulan en qué momento se escribe en los registros y qué operaciones se realizan en cada componente combinacional.

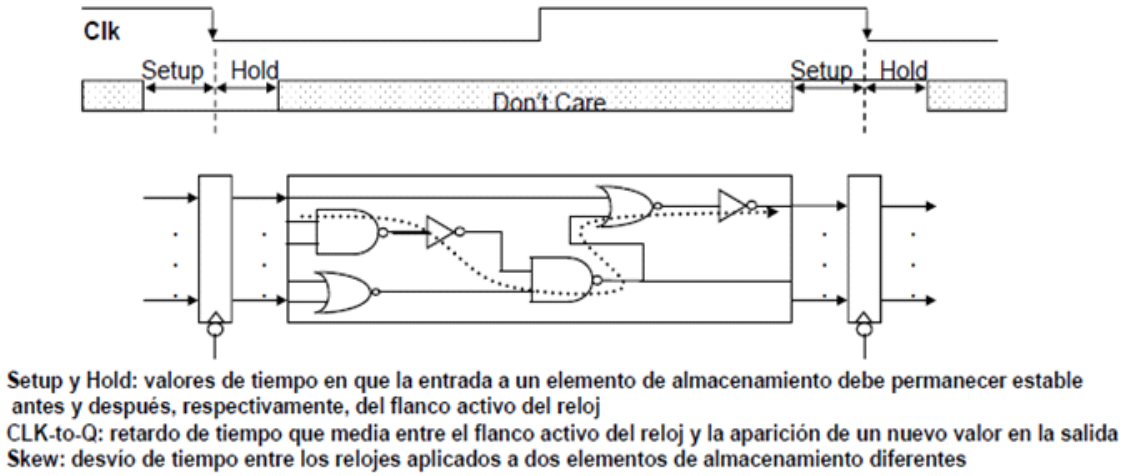


Figura 6. Esquema de temporización monociclo.

Al diseñar el procesador, hay que tener en cuenta parámetros de temporización muy específicos. Estos parámetros son a grandes rasgos los siguientes:

Tciclo : es el tiempo en el que se ejecuta una instrucción completa. Para permitir que se realicen todos los cálculos de forma correcta hay que analizar los retardos de la instrucción más lenta en su ejecución, en concreto del camino que tienen que recorrer las señales combinacionales de mayor retardo posible (camino con línea de putos en la Figura 6).

Setup: Tiempo que debe transcurrir tras antes del flanco de reloj en el cual la entrada a un registro debe permanecer estable para evitar introducir valores incorrectos.

Hold : Tiempo que debe transcurrir tras el flanco de reloj en el cual la entrada a un registro debe permanecer estable para evitar introducir valores incorrectos.

Clock Skew: Es el retardo asociado a las señales de reloj de los distintos biestables, necesario para que funcionen de forma correcta a pesar de recibir el flanco de reloj con cierto retraso.

Clk-to-Q : Es el tiempo que tarda en aparecer el valor correcto a la salida de un registro tras el flanco de reloj.

Todo lo anterior se resume en las siguientes dos ecuaciones:

- **Tciclo = CLK-to-Q + camino de máx. retardo + setup + clock skew**
- **(CLK-to-Q + camino de mín. Retardo – clock skew) > Hold**

En resumen, la ejecución típica de una instrucción en un procesador Monociclo consta de las siguientes fases:

- Los registros se cargan de forma simultánea y selectiva
- Los valores se propagan por las redes combinacionales hasta que quedan estables en la entrada de los registros
- Se repite el proceso indefinidamente

Finalmente, para el correcto funcionamiento de los elementos de almacenamiento todos deben estar sincronizados al mismo flanco de reloj.

Repertorio de instrucciones

A continuación describimos el repertorio de instrucciones MIPS que se han diseñado en este proyecto:

lw rt, inmed(rs) : BR[rt] ← MEM[BR[rs]+SignExt(inmed], PC ← PC + 4

La instrucción lw carga en registro una cantidad de 32 bits, obtenida de memoria.

0x23 : 6	rs : 5	rt : 5	inmed : 16

sw rt, inmed(rs) : MEM(BR[rs] + SignExt(inmed))← Br[rt]

PC←PC + 4

La instrucción sw carga el contenido de la memoria en la dirección especificada de 32 bits en un registro.

0x28 : 6	rs : 5	rt : 5	inmed : 16

add rd, rs, rt :

La instrucción add suma el contenido de dos registros y lo guarda en otro registro destino.

BR[rd] ← BR[rs] + BR[rt]

PC ← PC + 4

0 : 6	rs:5	rt : 5	rd : 5	0:5	0x20:6

sub rd, rs, rt

La instrucción sub resta el contenido de dos registros y lo guarda en otro registro destino.

BR[rd] ← BR[rs] – BR[rt]

PC ← PC + 4

0 : 6	rs:5	rt : 5	rd : 5	0:5	0x22:6

and rd, rs, rt

La instrucción and realiza el AND lógico del contenido de dos registros y lo guarda en otro registro destino.

BR[rd] ← BR[rs] and BR[rt]

PC ← PC + 4

0 : 6	rs:5	rt : 5	rd : 5	0:5	0x24: 6

or rd, rs, rt

La instrucción and realiza el AND lógico del contenido de dos registros y lo guarda en otro registro destino.

BR[rd] ← BR[rs] or BR[rt]

PC ← PC + 4

0 : 6	rs:5	rt : 5	rd : 5	0:5	0x25: 6

beq rs, rt, inmed

La instrucción beq realiza un salto a la dirección de memoria de 32 bits en caso de que el contenido de rs y rt sean iguales.

si (BR[rs] = BR[rt]) entonces (PC ← PC + 4 + 4·SignExt(inmed))

en otro caso PC ← PC + 4

4 : 6	rs:5	rt : 5	rd : 5	Inmed : 16

j inmed

La instrucción j introduce en el contador de programa una dirección especificada por el operando inmediato.

j PC ←4xINSTR[25-0]

2 : 6	Inmed:26

jr rs

La instrucción rs introduce en el contador de programa una dirección especificada por el contenido de un registro.

jr PC ← BR[rs]

0 : 6	rs:5	0 : 16	8 : 6

addiu rs,rt,inmed

La instrucción addiu realiza la suma entre el contenido de un registro y el inmediato y lo introduce en otro registro.

addiu BR[rt]←BR[rs] + SignExt(inmed)

9 : 6	rs:5	rt : 5	rd : 5	Inmed : 16

slti rs,rt,inmed

La instrucción slti comprueba si el valor de un registro es mayor que el operand inmediato y guarda el resultado de la comparación en otro registro

slti si BR[rs]<SignExt(inmed) BR[rt]=1, sino BR[rt] = 0

0: 6	rs:5	rt : 5	rd : 5	0:5	0x2a:6

slt rd, rs, rt

La instrucción slt compara el contenido de dos registros y introduce cual de ellos es mayor en un registro destino.

(si (BR[rs] < BR[rt]) entonces (BR[rd] ← 1)

en otro caso (BR[rd] ← 0))

PC ← PC+4

4 : 6	rs:5	rt : 5	rd : 5	Inmed : 16

TABLA CON LOS FORMATOS DE INSTRUCCIÓN

Lw	0x23 : 6	rs : 5	rt : 5	inmed : 16		
Sw	0x28 : 6	rs : 5	rt : 5	inmed : 16		
Add	0 : 6	rs:5	rt : 5	rd : 5	0:5	0x20:6
Addiu	9 : 6	rs:5	rt : 5	rd : 5	inmed : 16	
Sub	0 : 6	rs:5	rt : 5	rd : 5	0:5	0x22:6
And	0 : 6	rs:5	rt : 5	rd : 5	0:5	0x24:6
Or	0:6	rs : 5	rt : 5	rd : 5	0:5	0x25:6
beq	4:6	rs : 5	rt : 5	rd : 5	inmed : 16	
j	2:6	inmed:26				
jr	0:6	rs : 5	0:16	8:6		
slt	0:6	rs : 5	rt : 5	rd : 5	Inmed:16	

slti	4:6	rs : 5	rt : 5	rd : 5	0:5	0x2a:6

Los ciclos de las instrucciones comienzan buscando la instrucción en la memoria (fase fetch) y en función del tipo de instrucción se realiza una de las anteriores operaciones. Cuando la instrucción ha finalizado su ejecución, el ciclo vuelve a comenzar.

Componentes de la ruta de datos monociclo

Para implementar el subconjunto de instrucciones ejemplo del repertorio MIPS en una implementación monociclo se requieren de los componentes ilustrados en la Figura 7.

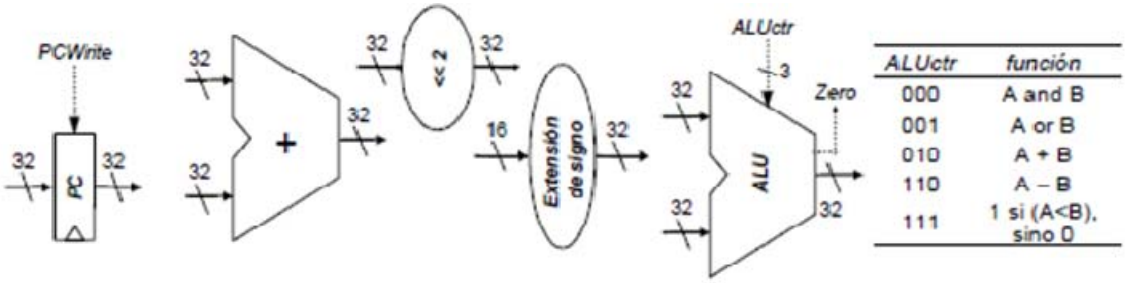


Figura 7. Elementos de la ruta de datos monociclo.

- Memoria de instrucciones
- Memoria de datos
- 32 registros de datos (visibles para el programador)
- PC (Contador de programa)
- 2 sumadores: para sumar 4 al PC, y para sumar al PC el valor inmediato de salto.
- ALU: capaz de realizar suma, resta, and, or, comparación de mayoría e indicación de que el resultado es cero (para realizar la comparación de igualdad mediante resta)
- Extensor de signo: para adaptar el operando inmediato de 16 bits al tamaño de palabra.
- Desplazador a la izquierda: para implementar la multiplicación por 4.

Los 32 registros de propósito general que vamos a usar se almacenan en un **Banco de registros** (Figura 8). Este banco de registros debe contar con las siguientes características:

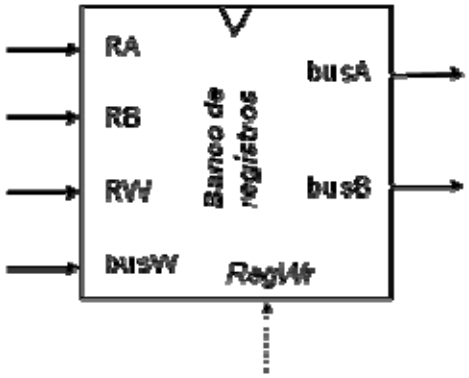


Figura 8. Banco de registros.

El banco de registros cuenta con la siguiente configuración de entradas y salidas:

- 2 salidas de datos de 32 bits
- 1 entrada de datos de 32 bits
- 3 entradas de 5 bits para la identificación de los registros
- 1 entrada de control para habilitar la escritura sobre uno de los registros
- 1 puerto de reloj (sólo determinante durante las operaciones de escritura, las de lectura son combinacionales)

La **memoria** (Figura 9) debe tener un comportamiento idealizado. Para ello debe estar integrada dentro de la CPU, y debe contar con las siguientes características: Direccionable por bytes y capaz de ofrecer y aceptar 4 bytes/acceso. Se supondrá que se comporta temporalmente como el banco de registros (síncronamente) y que tiene un tiempo de acceso menor que el tiempo de ciclo. Además, se supondrá dividida en dos para poder hacer dos accesos a memoria en el mismo ciclo:

- Memoria de instrucciones
- Memoria de datos

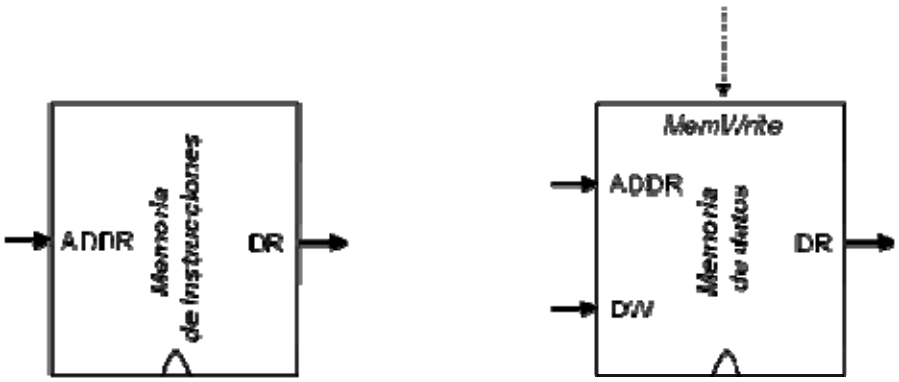


Figura 9. Memorias de la ruta de datos.

La configuración de entradas y salidas es la siguiente:

- 1 entrada de dirección
- 1 salida de datos de 32 bits
- 1 entrada de datos de 32 bits(solo en la memoria de datos)

Además, la memoria de datos deberá contar con una señal de control *MemWrite* que controle cuándo se escribirán los datos, y de esta manera evitar escribir valores incorrectos.

La ejecución monociclo obliga a no usar más de una vez por instrucción cada recurso, es decir, hay que duplicarlo si es necesario, como ya se ha mencionado debe contener una memoria de instrucciones y datos separadas y habrá que añadir multiplexores cuando un valor pueda provenir de varias fuentes.

Temporización monociclo: ejemplo

Como vemos en la Figura 10, para completar la ejecución de instrucciones en un ciclo es necesario coordinar y sincronizar las lecturas y las escrituras de los registros del PC y del Banco de registros, y , como hemos explicado anteriormente, que el tiempo de ejecución de la parte combinacional de la ruta de datos no exceda en ningún caso el tiempo de ciclo.

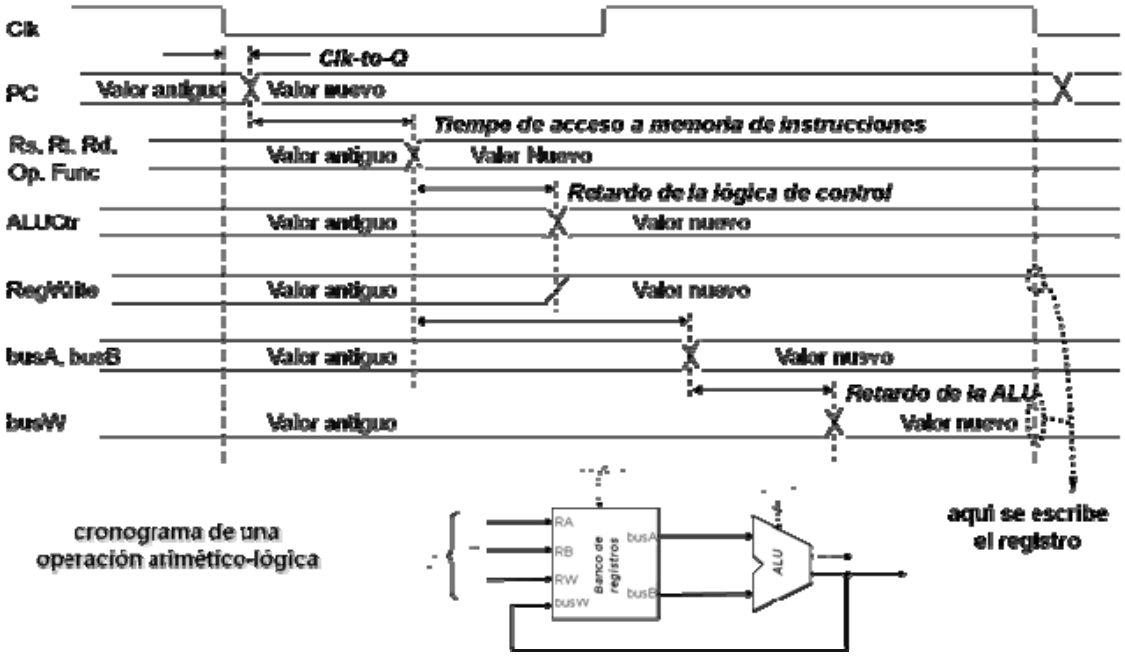


Figura 10. Cronograma para instrucción aritmético-lógica.

La Figura 9 representa la temporización de una instrucción tipo r. En ella podemos observar los distintos retardos que produce cada componente, que a grandes rasgos son los siguientes

- El Clk-to-Q, como ya hemos comentado anteriormente, representa el tiempo que tarda en conseguir el valor correcto a la salida de un biestable tras el flanco de reloj. Una vez ha pasado este tiempo, ya tenemos el valor correcto para continuar ejecutando la instrucción.

- Las memorias en general presentan un retardo importante. Como refleja la Figura 10, la memoria de instrucciones no es una excepción, ya que tiene que buscar la dirección especificada por el registro PC entre una gran cantidad de direcciones, lo que produce un retardo bastante elevado.

- La lógica de control que regula las señales que reciben los módulos, debido a su carácter combinacional, también presenta un cierto retardo producido por las puertas que representan los valores especificados en al tabla de verdad.

- La ALU, a su vez, también presenta cierto retardo para calcular los valores finales a las operaciones que realiza, puesto que se compone de una serie de puertas lógicas que realizan cada tipo de operación y controlan cúal de ellas se ejecuta en cada momento.

Estos retardos por sí solos no representan un problema, pero, como los datos se propagan a través de la ruta de datos, que es un circuito de cierta complejidad, han de ser tenidos en cuenta para asegurar el correcto funcionamiento de ésta y basarse en ellos para diseñar las posibles implementaciones .

Ruta de Datos

En la Figura 11 tenemos la ruta de datos implementada en este trabajo al completo, incluyendo las señales de la unidad de control que controlan el funcionamiento de los módulos.

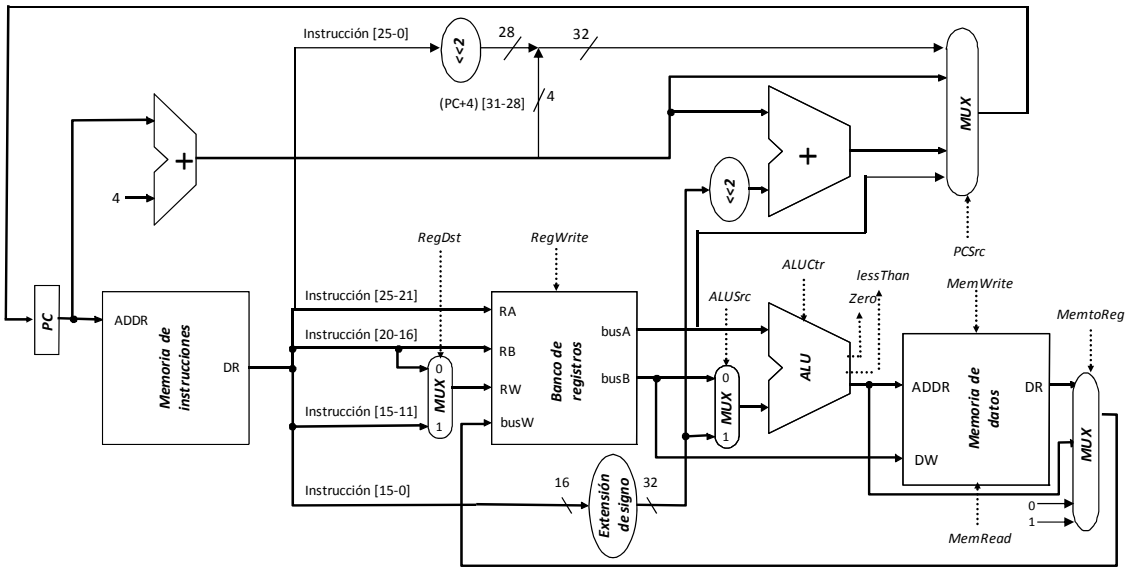


Figura 11. Ruta de datos monociclo.

Para controlar las señales combinacionales que gobiernan esta ruta de datos necesitaremos una **unidad de control** que, según el tipo de instrucción codifique sus señales de control correspondientes y las propague a los elementos combinacionales.

Así pues, las principales tareas de nuestro controlador serán: Por un lado, seleccionar las operaciones a realizar por los módulos multifunción (ALU, read/write, ...). Por otro lado, controlar el flujo de datos, activando la entrada de selección de los multiplexores y la señal de carga de los registros

Los valores de las señales de control en cada instrucción son los siguientes:

La tabla inferior representa las transferencias que se realizan entre los módulos de la ruta de datos. La parte de la instrucción en negrita representa las transferencias propiamente dichas y el resto los valores que toman las señales del controlador.

Instrucción de carga (lw)
BR[rt] ← MEM[BR[rs] + SignExt(inmed)], PC ← PC+4 RegDest←0, RegWrite←1, ALUCtr←010, PCSrc←0, MemWrite←0, MemRead←1, MemtoReg←1
Instrucción de almacenaje(sw)
Memoria(rs + SignExt(inmed)) ← rs, PC ← PC + 4 RegDest ←X, RegWrite ← 0, ALUsrc ← 1, ALUctr ← 010, PCSrc ← 0, MemWrite ← 1, MemRead ← 0, MemtoReg ← X
Instrucción add
rd ← rs + rt, PC ← PC + 4 RegDest ← 1, RegWrite ← 1, ALUsrc ← 0, ALUctr ← 010, PCSrc ← 0, MemWrite ← 0, MemRead ← 0, MemtoReg ← 0
Instrucción sub
rd ← rs - rt, PC ← PC + 4 RegDest ← 1, RegWrite ← 1, ALUsrc ← 0, ALUctr ← 110, PCSrc ← 0, MemWrite ← 0, MemRead ← 0, MemtoReg ← 0
Instrucción *and*
rd ← rs and rt, PC ← PC + 4 RegDest ← 1, RegWrite ← 1, ALUsrc ← 0, ALUctr ← 000, PCSrc ← 0, MemWrite ← 0, MemRead ← 0, MemtoReg ← 0
Instrucción *or*
rd ← rs or rt, PC ← PC + 4 RegDest ← 1, RegWrite ← 1, ALUsrc ← 0, ALUctr ← 001, PCSrc ← 0, MemWrite ← 0, MemRead ← 0, MemtoReg ← 0
Instrucción de salto condicional (beq)
si (rs = rt) entonces (PC ← PC + 4 + 4·SignExp(inmed)) en otro caso PC ← PC + 4 RegDest ←X, RegWrite ← 0, ALUsrc ← 0, ALUctr ← 110, PCSrc ← Zero, MemWrite ← 0, MemRead ← 0, MemtoReg ← X
Instrucción addiu
BR[rt]←BR[rs] + SignExt(inmed) RegDest ← 1, RegWrite ← 1, ALUsrc ← 1, ALUctr ← 010, PCSrc ← 0, MemWrite ← 0, MemRead ← 0, MemtoReg ← 0
Instrucción j
PC ←4xINSTR[25-0] RegDest ← x, RegWrite ←x, ALUsrc ←x, ALUctr ← x, PCSrc ← 2, MemWrite ← 0, MemRead ← 0, MemtoReg ← 0
Instrucción slti
si BR[rs]<SignExt(inmed) BR[rt]=1, sino BR[rt] = 0 RegDest ← 1, RegWrite ← 1, ALUsrc ← 1, ALUctr ← 110, PCSrc ← 0, MemWrite ← 0, MemRead ← 0, MemtoReg ← LessThan
Instrucción slt
Si BR[rs] < [BR]rt (BR[rd] ← 1) , sino (BR[rd] ← 0) , PC ← PC+4 RegDest ← 1, RegWrite ← 1, ALUsrc ← 0, ALUctr ← 001, PCSrc ← 0, MemWrite ← 0, MemRead ← 0, MemtoReg ← 0

En esta versión del procesador, como podemos ver, hay pocas señales de control y la realización de la unidad de control resulta relativamente simple. Por el contrario, en las versiones multiciclo y segmentada la realización del controlador resultará mucho más complicada debido a la presencia de multitud de registros con sus correspondientes señales de control en la ruta de datos.

Con el uso conjunto de esta ruta de datos al completo y la unidad de control quedaría completamente especificado nuestro pequeño repertorio de instrucciones. Vamos a ver cómo resulta la ejecución de una **instrucción lw** en nuestro procesador a lo largo del ciclo de reloj en la Figura 12.

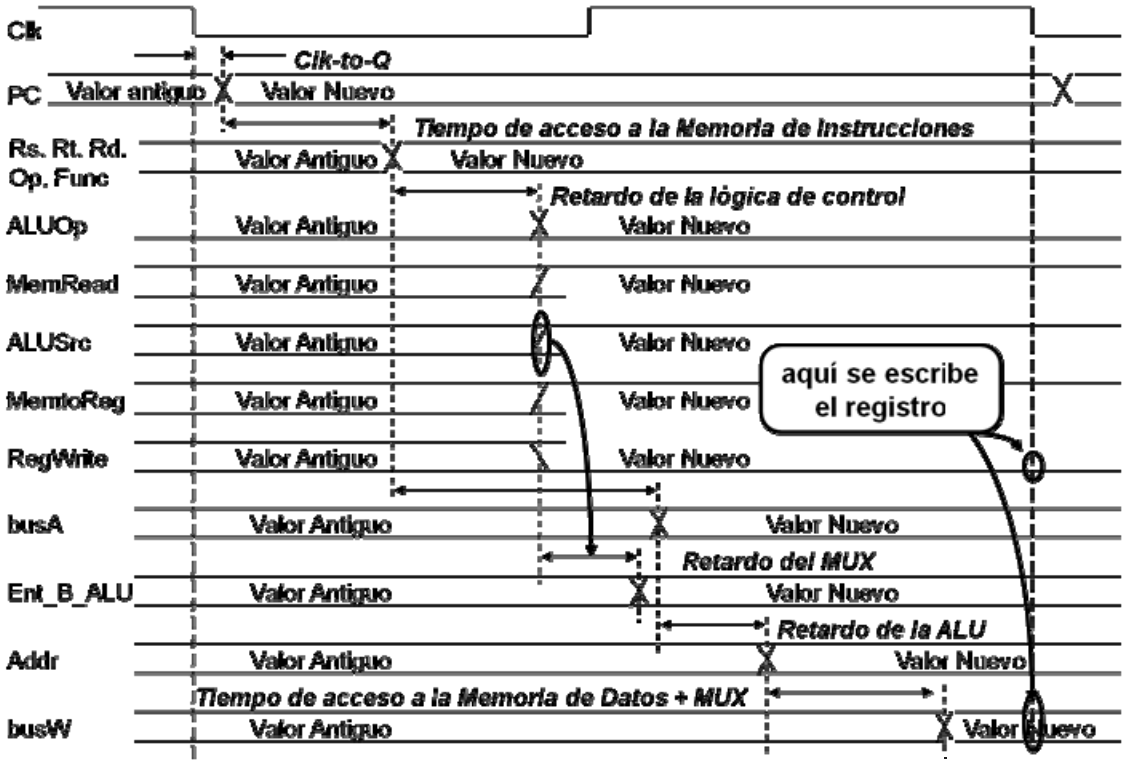


Figura 12. Cronograma de la instrucción lw.

La Figura 12 representa, de forma más desgranada que la Figura 10, ya comentada, un diagrama con la temporización y los retardos que aparecen en la instrucción load, así como las señales de la ruta de datos monociclo.

Esta Figura incluye, además de los retardos ya comentados, la propagación de los datos a través de los cables y el momento en el que éstos llegan. Vamos a pasar a describirlas brevemente:

Las señales Rs, Rt, Rd representan los registros del Banco de Registros, y indican en que momento éstos reciben los valores correctos.

Las señales PC, Op y Func representan al Contador de Programa, y la operación y Función que se realizarán posteriormente en la ALU respectivamente que se obtienen del registro IR.

La señal ALUOp representa la operación que realiza la ALU, indicada por la Unidad de Control, y MemRead controla el momento en el cual leemos de memoria.

Las señales ALUSrc y MemToReg controlan los multiplexores a la entrada de la ALU y a la salida de la memoria de datos.

La señal busA representa la salida A del banco de registros, que se suma a la Ent_B de la ALU para obtener la dirección final, representada por Addr. Por último, en la señal busW se pone el valor que buscábamos de la memoria.

Con esto quedaría completo nuestro procesador monociclo. Para añadir nuevas instrucciones, únicamente deberíamos adaptar la ruta para añadir las nuevas funcionalidades que necesitemos, y modificar la unidad de control convenientemente.

Sin embargo, la ruta de datos monociclo presenta algunas deficiencias. La más relevante consiste en que el periodo de reloj se debe adaptar a la instrucción más lenta. La Figura 13 ilustra el problema. Muchas instrucciones se podrían ejecutar en un tiempo menor al de la instrucción más lenta. Siguiendo el ejemplo de la Figura 13, una instrucción sw tarda menos que una instrucción lw. Las instrucciones tipo beq y aritmético-lógicas son aún más rápidas.

Figura 13. Ejecución de las instrucciones en un procesador monociclo.

Para aprovechar esta característica y usando el MIPS monociclo como base, en los siguientes apartados desarrollaremos la versión multiciclo y segmentada del procesador.

3.6. Procesador multiciclo

Para la implementación multiciclo, vamos a emplear el mismo conjunto de instrucciones que en el monociclo. Esta implementación pretende mejorar algunos de los aspectos característicos de la anterior ruta de datos y así aumentar el rendimiento. En primer lugar el reloj debe tener igual periodo que la instrucción más lenta. Por un lado, dado que dicho periodo es fijo, en las instrucciones rápidas se desperdicia tiempo. Por el otro, en repertorios reales, existen instrucciones muy largas: aritmética en punto flotante, modos de direccionamiento complejos, etc. En segundo lugar no se puede reusar hardware. Si en una instrucción se necesitara hacer 4 sumas (resolver los 3 modos de direccionamiento de los operandos y sumarlos) se necesitarían 4 sumadores.

La solución que se propone a estos problemas es la siguiente: dividir la ejecución de la instrucción en varios ciclos más pequeños. De esta manera cada instrucción usará el número de ciclos que necesite, en este caso variable por tipo de instrucción. Igualmente, un mismo elemento hardware puede ser utilizado varias veces en la ejecución de una instrucción si se hace en ciclos diferentes. Por el contrario, se requieren elementos adicionales para almacenar valores desde el ciclo en que se calculan hasta el ciclo en que se usan.

Recordamos el repertorio de instrucciones implementado anteriormente:

- lw rt, inmed(rs)
- sw rt, inmed(rs)
- add rd, rs, rt
- sub rd, rs, rt
- and rd, rs, rt
- or rd, rs, rt
- slt rd, rs, rt
- beq rs, rt, inmed
- j label
- jr rs
- slti rt, rs, inmed
- addiu rt, rs, inmed

Temporización Multiciclo

En el procesador multiciclo las instrucciones se dividen en cinco fases: búsqueda, decodificación, cálculo de operandos/búsqueda y ejecución/almacenaje. En cada ciclo se lee/escribe en el registro/memoria correspondiente.

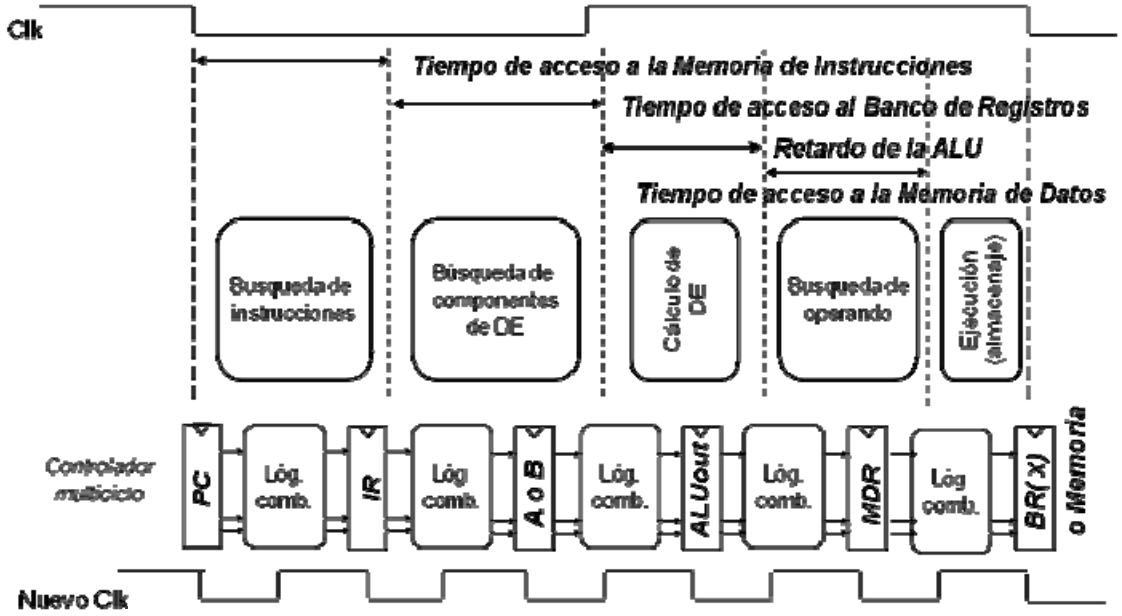


Figura 14. Temporización multiciclo.

La Figura 14 representa de forma esquemática cómo se subdivide en etapas la ejecución Multiciclo de instrucciones. Las etapas resultantes vienen dadas por los caminos que pueden recorrer los datos entre registros(en un solo ciclo) o memoria.

La búsqueda de instrucciones consiste en extraer de la memoria la instrucción, con su tiempo de acceso asociado, siguiente a ejecutar, para posteriormente almacenarse en el registro IR.

Tras pasar por el registro IR, pasamos a la fase de Búsqueda de Componentes de DE, los datos pasan por el Banco de Registros en caso de ser necesario y se procesan hasta llegar a la fase de Cálculo de DE, en la cual se calculan mediante la ALU los valores que emplearemos como dirección de memoria o operandos.

Tras esta fase viene la búsqueda de operandos, los datos pasan por la memoria de datos(con su correspondiente retardo en caso de escritura/lectura) y vuelven al banco de registros en instrucciones de carga o almacenaje, para por último acabar con la fase de ejecución que termina el proceso.

Ruta de datos multiciclo

A continuación la Figura 15 ilustra la ruta de datos multiciclo al completo. Esta ruta de datos implementa el mismo repertorio de instrucciones que la ruta monociclo analizada anteriormente. Como podemos ver en la imagen, a simple vista se pueden percibir una serie de cambios importantes :

- Ahora todas las operaciones de suma que se realizan las calcula la propia ALU, lo que permite ahorrar componentes.
- Necesitamos registros intermedios (A, B , MDR , ALUo..) para guardar los valores intermedio tras cada ciclo.
- Algunos multiplexores han aumentado su número de entradas a 4 para aportar mas flexibilidad a la ruta y ahorrar hardware.
- La Unidad de Control será algo más complicada de realizar, puesto que la inclusión de los nuevos registros implica la existencia de muchas mas señales de control, lo que aumenta significativamente la complejidad del circuito combinacional que actúa como controlador.

La realización de la ruta de datos Multiciclo implica emplear mas hardware, pero la mejora tan significativa que se produce en los tiempos de ejecución merece la pena en la mayoría de los casos y mejora ampliamente el rendimiento frente a la ejecución monociclo.

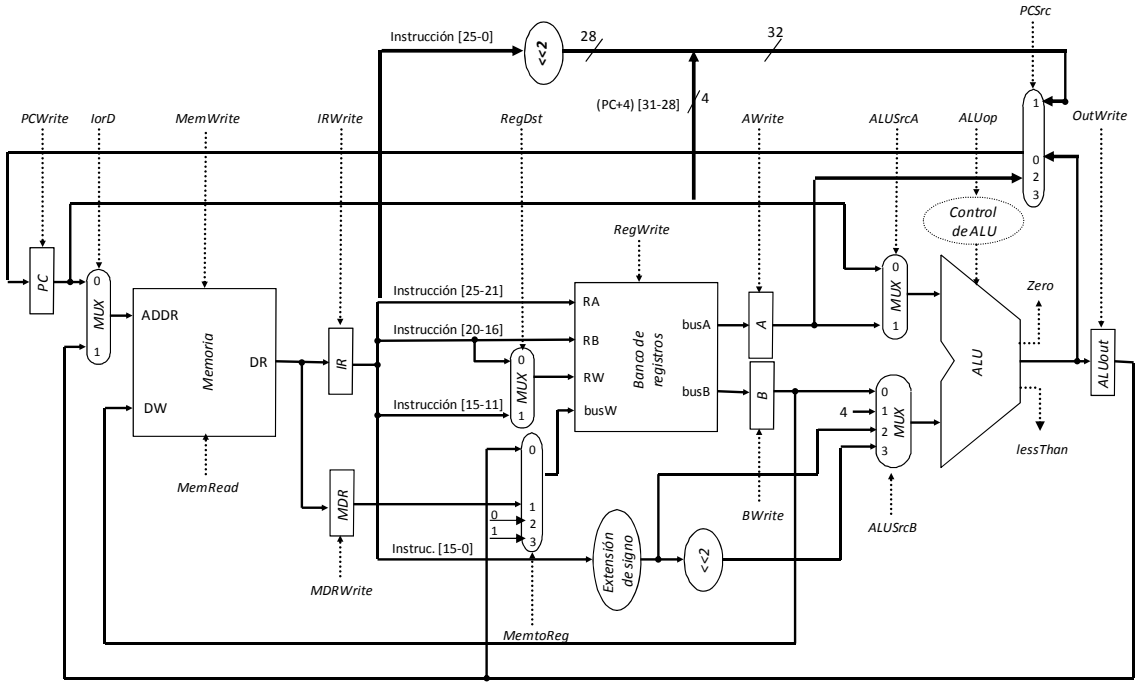


Figura 15. Ruta de datos multiciclo.

Unidad de control multiciclo

Ahora la unidad de control del procesador será una máquina de estados, cuyas salidas relativas a cada estado serán las señales de control que rijan el comportamiento de los elementos combinacionales y los registros.

A continuación describiremos las transferencias a nivel de registro que tendrán lugar en cada instrucción:

Instrucción lw

Transferencias entre registros "lógicas"

BR(rt) ← Memoria(BR(rs) + SignExt(inmed)), PC ← PC + 4

Transferencias entre registros "físicas"

1. IR ← Memoria(PC), PC ← PC + 4
2. A ← BR(rs)
3. ALUout ← A + SignExt(inmed)
4. MDR ← Memoria(ALUout)
5. BR(rt) ← MDR

Instrucción sw

Transferencias entre registros "lógicas"

Memoria(BR(rs) + SignExt(inmed)) ← BR(rt), PC ← PC + 4

Transferencias entre registros "físicas"

1. IR ← Memoria(PC), PC ← PC + 4
2. A ← BR(rs), B ← BR(rt)
3. ALUout ← A + SignExt(inmed)
4. Memoria(ALUout) ← B

Instrucción aritmética (tipo r)

Transferencias entre registros "lógicas"

BR(rd) ← BR(rs) funct BR(rt), PC ← PC + 4

Transferencias entre registros "físicas"

1. IR ← Memoria(PC), PC ← PC + 4
2. A ← BR(rs), B ← BR(rt)
3. ALUout ← A funct B
4. BR(rd) ← ALUout

Instrucción beq

Transferencias entre registros "lógicas"

si (BR(rs) = BR(rt))
entonces PC ← PC + 4 + 4·SignExt(inmed)
sino PC ← PC + 4

Transferencias entre registros "físicas"

1. IR ← Memoria(PC), PC ← PC + 4
2. A ← BR(rs), B ← BR(rt),
3. A - B
4. si Zero entonces PC ← PC + 4·SignExt(inmed)

Instrucción j

Transferencias entre registros lógicas:

1. PC ← 4 x SignExt(inmed)

Transferencias entre registros "físicas":

1. IR←MEM[PC]
2. PC ← 4 x IR[25-0]

Instrucción jr

Transferencias entre registros lógicas:

1. PC ← BR[rs]

Transferencias entre registros "físicas":

1. IR←MEM[PC]
2. A ← BR[rs]
3. PC ← A

Instrucción slti

Transferencias entre registros lógicas:

1. si BR[rs]<SignExt(inmed) BR[rt]=1, sino BR[rt] = 0

Transferencias entre registros "físicas":

1. IR ← Memoria(PC), PC ← PC + 4
2. A ← BR(rs), B ← BR(rt)
3. ALUout ← A - SignExt(inmed)
4. si ALUout < 0 entonces BR[rt]=1, sino BR[rt] = 0

Instrucción slt

Transferencias entre registros lógicas:

1. si BR[rs]<SignExt(inmed) BR[rt]=1, sino BR[rt] = 0

Transferencias entre registros "físicas":

1. IR ← Memoria(PC), PC ← PC + 4
2. A ← BR(rs), B ← BR(rt)
3. ALUout ← A - B
4. si ALUout < 0 entonces BR[rt]=1, sino BR[rt] = 0

Instrucción addiu

Transferencias entre registros "lógicas"

1.BR(rd) ← BR(rs) funct SignExt(inmed), PC ← PC + 4

Transferencias entre registros "físicas"

1. IR ← Memoria(PC), PC ← PC + 4
2. A ← BR(rs), B ← BR(rt)
3. ALUout ← A funct SignExt(inmed)
4. BR(rd) ← ALUout

Como podemos apreciar, los estados 0 y 1 son similares (o iguales) en casi todas las instrucciones. Para simplificar la máquina de estados, hacemos coincidir estos estados en todas las instrucciones. De esta forma, la secuencia de control será más sencilla. Con estas aproximaciones, el diagrama de estados quedaría de la siguiente manera (Figura 16).

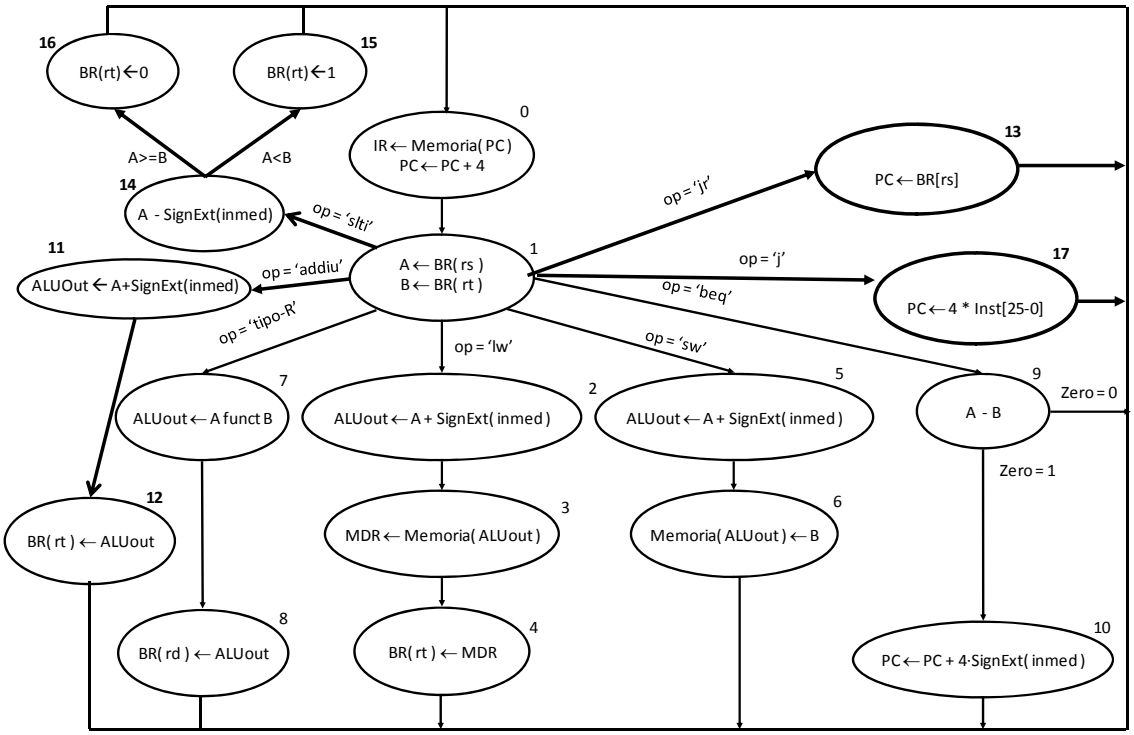


Figura 16. Diagrama de estados multiciclo.

La Figura 16 representa el diagrama completo de estados de nuestro procesador. Nótese que hemos incluido todas las nuevas instrucciones(como por ejemplo slti) que hemos implementado. Hemos numerado cada estado con un número que emplearemos para realizar la codificación necesaria para la elaboración de la tabla de verdad del controlador.

Sobre las líneas que unen unos estados con otros aparecen anotaciones como el tipo de instrucción determinado por sus campos o los resultados de comparaciones, que determinan el camino que toma la ejecución en la máquina de estados.

Dentro de cada estado aparecen las transferencias que en éste se realizan, cada una en un ciclo.

Como podemos observar, los estados 0 y 1 son comunes a todos los tipos de instrucción, y a partir de ahí según el tipo de operación que se vaya a realizar toma un camino u otro.

El **CPI** de una instrucción determinada se define como el número de ciclos de reloj que transcurren para la ejecución de una instrucción. Es una medida que se emplea, además del tiempo de ciclo, para evaluar el rendimiento de un procesador en cuanto a tiempo de ejecución. Para calcular el tiempo de ejecución de una instrucción :

$$\mathbf{T_{ejecución} = T_{ciclo} \times CPI}$$

El CPI se emplea para hallar otro paramétro de gran importancia, el **CPI en promedio**. Para calcularlo:

$$\mathbf{CPI_{medio}} = \frac{\sum \boldsymbol{CPIinstrucción} * \boldsymbol{frecuencia\ de\ aparición(instrucción)}}{\boldsymbol{nº\ de\ tipos\ de\ instrucción}}$$

A continuación presentamos una tabla con los distintos CPIs de cada instrucción para nuestra implementación Multiciclo:

CPI de las instrucciones:	
Tipo-R	4
Instrucción lw	5
Instrucción sw	4
Instrucción beq(salta)	3
Instrucción beq(no salta)	4
Instrucción slt	4
Instrucción slti	4
Instrucción j	3
Instrucción jr	3

Estado actual	op	Zero	Estado siguiente	IRWrite	PCWrite	AWrite	BWrite	ALUSrcA	ALUScrB	ALUOp	OutWrite	MemWrite	MemRead	IorD	MDRWrite	MemtoReg	RegDest	RegWrite
0000	XXXXXX	X	0001	1	1			0	01	00 (add)		0	1	0				0
0001	100011 (lw)	X	0010	0	0	1	1					0	0					0
0001	101011 (sw)	X	0101															
0001	000000 (tipo-R)	X	0111															
0001	000100 (beq)	X	1001															
0010	XXXXXX	X	0011	0	0			1	10	00 (add)	1	0	0					0
0011	XXXXXX	X	0100	0	0							0	1	1	1			0
0100	XXXXXX	X	0000	0	0							0	0			1	0	1
0101	XXXXXX	X	0110	0	0		0	1	10	00 (add)	1	0	0					0
0110	XXXXXX	X	0000	0	0							1	0	1				0
0111	XXXXXX	X	1000	0	0			1	00	10 (funct)	1	0	0					0
1000	XXXXXX	X	0000	0	0							0	0			0	1	1
1001	XXXXXX	0	0000	0	0			1	00	01 (sub)		0	0					0
1001	XXXXXX	1	1010															
1010	XXXXXX	X	0000	0	1			0	11	00 (add)		0	0					0

Figura 17. Tabla de transición de estados del procesador multiciclo.

La Figura 17 representa la tabla de verdad completa empleada para la realización del controlador empleando una máquina de Moore. Como podemos observar, para cada estado tenemos como salidas las señales que gobiernan los módulos combinacionales o las escrituras/lecturas de registros y memoria.

Cabe destacar la existencia de la señal especial Zero, la cual se emplea para comparaciones y saltos, que comprueba si el resultado de una operación en la ALU es igual a 0. También para

cada estado se especifica el estado siguiente en la máquina de estados. Hemos incorporado otra señal adicional LessThan que se activa cuando el operando A es menor que el operando B.

Ejemplo 1: Ejecución de una instrucción Load ciclo a ciclo

Vamos a ilustrar a continuación un diagrama ciclo a ciclo de la ejecución de una instrucción de carga (sobre una ruta de datos simplificada) y observar así el recorrido de los datos a través del circuito, marcado en rojo.

Como podemos observar en la Figura 18, en el primer ciclo de reloj se carga guarda en el PC el valor de la siguiente dirección de memoria (PC + 4) y se carga en el registro IR la dirección de memoria que entra por el puerto ADDR de la Memoria. Para realizar estas acciones, es necesario activar las señales de escritura PCWrite y IRWrite, seleccionar la entrada 0 del multiplexor IorD y las entradas 0 y 1 de los multiplexores ALUSrcA y ALUSrcB, respectivamente. Además, es necesario seleccionar la suma como la operación a realizar en la ALU

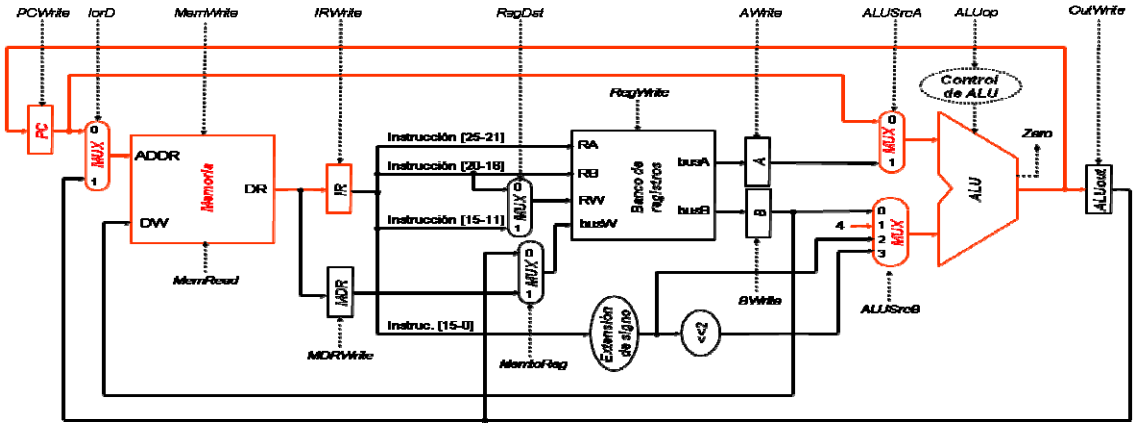


Figura 18. Pimer ciclo de la instrucción lw, la ejecución comienza calculando los valores del PC y actualizando el contenido del registro IR con la instrucción actual.

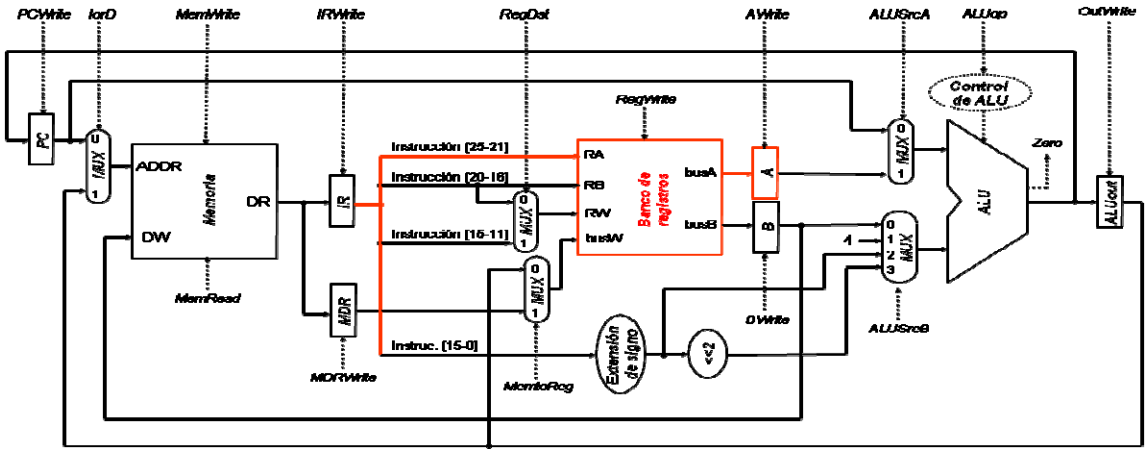


Figura 19. Segundo ciclo, extraemos del banco de registros la dirección base sobre la que se aplicará el offset de la instrucción de carga.

En el segundo ciclo de reloj, el contenido de IR determina que registro interno del Banco de Registros se carga en el Registro A. Para cargar el contenido en el registro A es necesario también activar su señal de control AWrite (Figura 19).

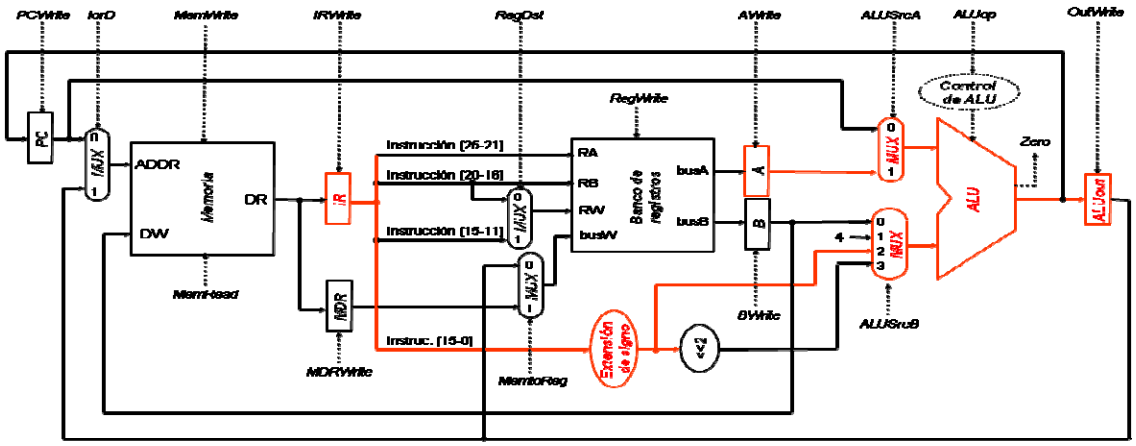


Figura 20. Tercer ciclo, se realiza la suma del inmediato con el contenido del registro A para calcular la dirección efectiva a cargar de la memoria, para posteriormente guardarla en el registro ALUout.

En el ciclo siguiente se extrae del registro IR el operando inmediato y se efectúa una suma en la unidad aritmético lógica para calcular la dirección final efectiva de memoria. El resultado se guarda en el registro ALUo. Para ello es necesario activar las señales ALUoWrite, seleccionar las entradas 1 y 2 de los multiplexores ALUSrcA y ALUSrcB respectivamente y seleccionar la suma como operación a realizar en la ALU (Figura 20).

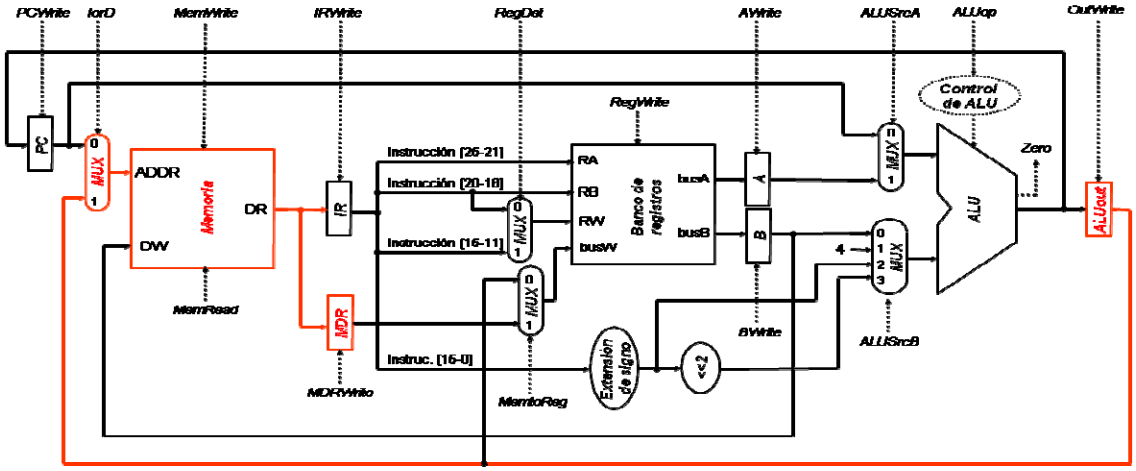


Figura 21. Cuarto ciclo, el contenido del registro ALUout se emplea para extraer la posición deseada de la memoria, que guardamos en el registro MDR.

En el cuarto ciclo de ejecución se lee el contenido del registro ALUo y se introduce por el puerto ADDR de la memoria para introducir en el registro MDR el contenido de la dirección. Para ello necesitamos activar la señal MDRWrite y poner la señal IorD a 1, además de activar MemRead para leer de memoria (Figura 21).

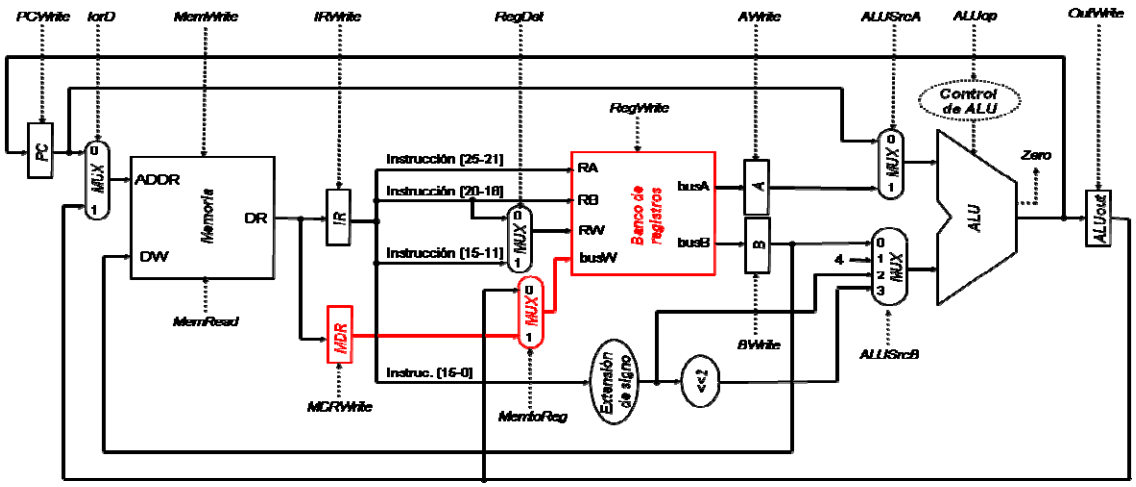


Figura 22. Quinto ciclo, El contenido del registro MDR se guarda en el banco de registros, en el registro destino.

Por último, como refleja la Figura 22, en el quinto ciclo de reloj extraemos el contenido de MDR y guardamos en el Banco de registros, en el registro destino, el valor extraído de la memoria. Para efectuar esta acción necesitamos seleccionar la entrada 1 del multiplexor MemToReg y activar la señal RegWrite.

Con esto termina la ejecución de la instrucción load, y el valor pedido de la memoria queda guardado en el registro destino.

Ejemplo: Comparación Multiciclo/Monociclo con un programa sencillo

En este ejemplo vamos a comparar la ejecución de un programa MIPS sencillo, con instrucciones variadas, para comparar la cantidad de ciclos empleados y el tiempo total de ejecución en sus versiones monociclo y multiciclo.

Empleando un estudio de la frecuencia de las instrucciones de cada tipo obtenemos un CPI promedio que usaremos para calcular el tiempo de ejecución de un número grande de instrucciones.

Supongamos por tanto que se ejecuta el siguiente conjunto de instrucciones:

```
lw $t2, 0($t3)
lw $t3, 4($t3)
beq $t2, $t3, Label #asumir que no se salta
add $t5, $t2, $t3
sw $t5, 8($t3)
```

Suponemos además que el tiempo de ciclo del procesador multiciclo es cinco veces menor que el procesador monociclo. La Figura 23 muestra el cronograma de ejecución del programa en ambos procesadores (primero el multiciclo y debajo el monociclo).

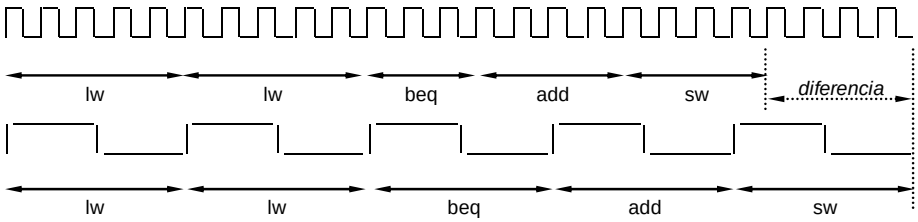


Figura 23. Procesador monociclo vs. multiciclo.

Como observamos en el ejemplo, la ejecución del programa en su versión monociclo es mucho más rápida. Esto demuestra con creces que el empleo de varios ciclos más cortos mejora el rendimiento sustancialmente frente a la ejecución de instrucciones en un tiempo de ciclo más largo.

Bajo los mismos supuestos de temporización, supongamos que se ejecuta ahora un programa con el siguiente perfil de porcentaje instrucciones aritmético lógicas, lw, sw, y beq, del cual podemos calcular su CPI:

Operación	Frecuencia	Ciclos	CPI
Tipo-R	50 %	4	2.0
Lw	20 %	5	1.0
Sw	10 %	4	0.4
beq (salta)	2.5 %	4	0.1
beq (no salta)	17.5 %	3	0.53
			4.03

Bajo este supuesto, 1 millón de instrucciones tardarán en ejecutarse:

$$t_{multi} = 10^6 \cdot CPI_{multi} \cdot t_{multi} = 10^6 \cdot 4.03 \cdot t_{multi}$$

$$t_{mono} = 10^6 \cdot CPI_{mono} \cdot t_{mono} = 10^6 \cdot 1 \cdot 5 \cdot t_{multi}$$

de donde

$$\frac{t_{multi}}{t_{mono}} = \frac{4.03}{5} = 0.8$$

o lo que es lo mismo, los programas tardan en ejecutarse un 20% menos en el procesador multiciclo.

Optimización del diagrama de estados del procesador Multiciclo

El diagrama de estados multiciclo expuesto anteriormente presenta algunos registros que no son necesarios. Además, se puede optimizar la ejecución de la instrucción de salto condicional adelantando el cálculo del destino del salto al estado 1. A continuación ilustramos cómo se puede optimizar la ruta de datos multiciclo sobre un conjunto de instrucciones reducido, en concreto el tratado en [4].

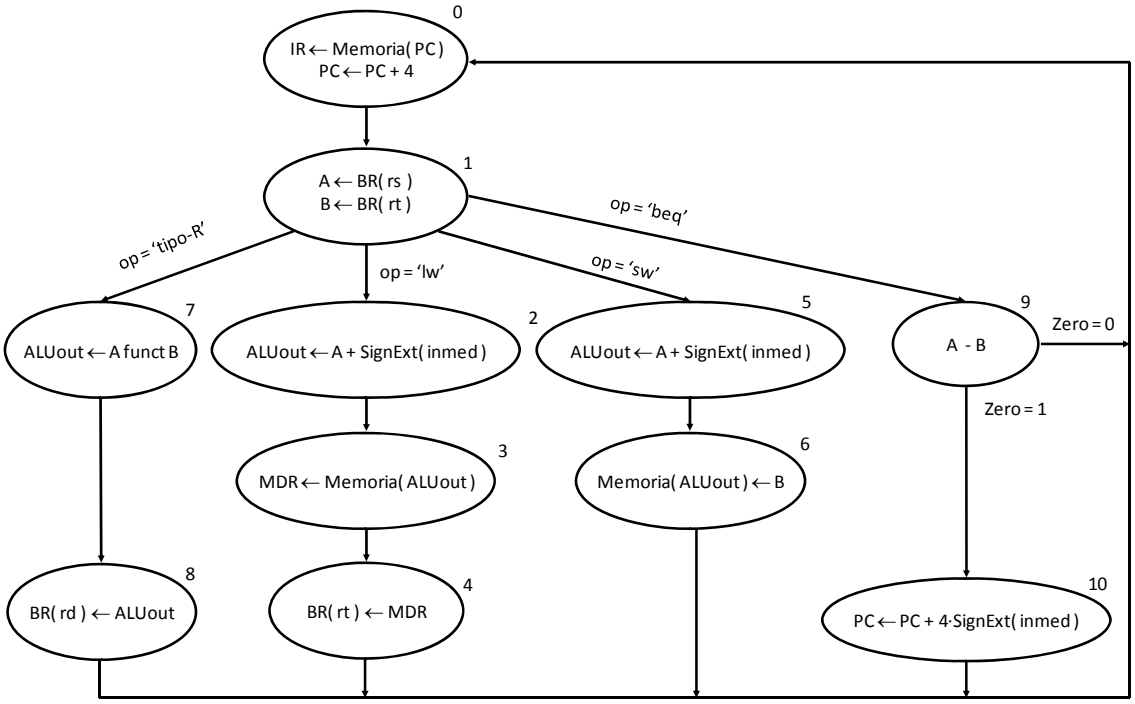


Figura 24. Diagrama de estados con el repertorio de instrucciones simplificado.

Estas optimizaciones se pueden obtener fácilmente examinando el diagrama de estados simplificado de la Figura 24. Como podemos observar, los registros A y B, una vez cargados son utilizados en el estado inmediatamente siguiente. O bien, como es el caso del estado 6, las entradas a estos registros aún no han sido modificadas por la ruta de datos, lo que significa que podemos prescindir de estos registros. Lo mismo ocurre con los registros MDR y ALUOut.

Además viendo el estado 10 de la Figura 24, podemos adelantar el cálculo del destino del salto al estado 1. En este caso sí que tendremos que utilizar un registro para almacenar dicho destino. De esta forma, la instrucción beq tardará 3 ciclos, independientemente de si se salta o no.

La ruta de datos modificada quedará tal como ilustra la Figura 25, donde se han eliminado los registros MDR, A y B, se ha renombrado ALUOut a Target y se ha añadido el multiplexor PCSrc.

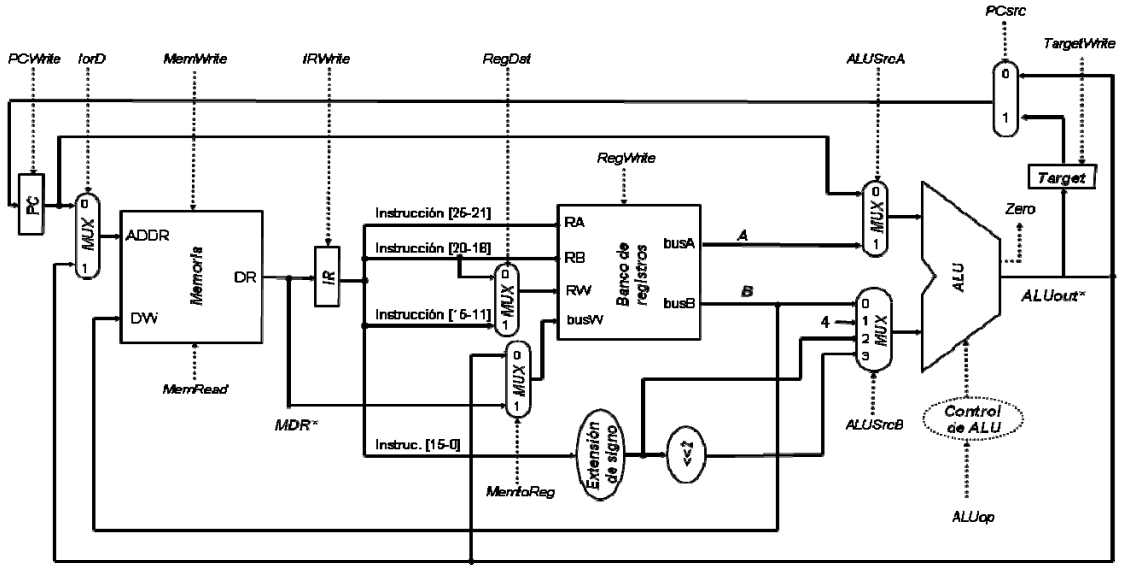


Figura 25. Ruta de datos optimizada.

Con estas modificaciones, el diagrama de estados de la Figura 24 queda como refleja la Figura 26. Los registros eliminados A, B, MDR y ALUOut se marcan en la Figura 26 como A*, B*, MDR*, y ALUOut*, respectivamente. Nótese que aunque algunos estados podrían eliminarse (como por ejemplo el 6), no se eliminan porque habría que aumentar considerablemente el tiempo de ciclo.

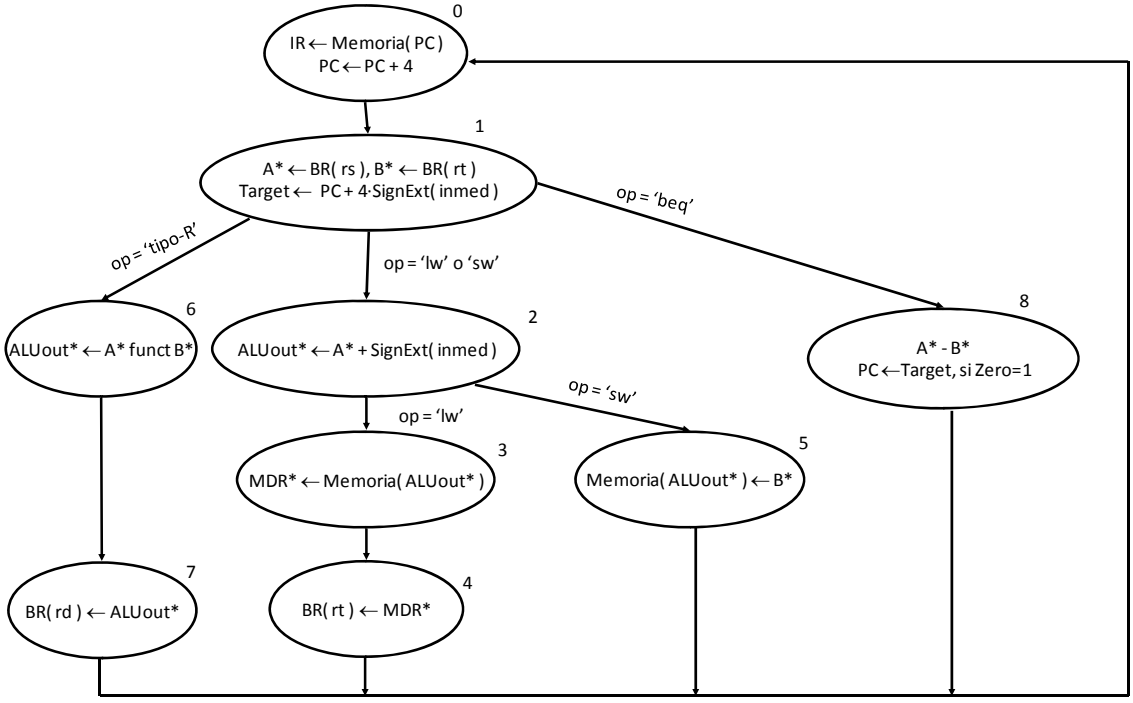


Figura 26. Diagrama de estados optimizado.

La siguiente tabla compara los valores del CPI (ciclos por instrucción) antes y después de la modificación en el diagrama de estados. Como podemos observar, conseguimos ahorrar un ciclo en la instrucción beq, en caso de que no se realice el salto.

Tipo de instrucción	CPI antes	CPI después
Tipo R	4	4
Lw	5	5
Sw	4	4
beq (salta)	3	3
beq (no salta)	4	3

En definitiva, este diagrama alternativo de la ruta de datos optimiza sustancialmente el tiempo de ejecución, puesto que reduce los ciclos empleados para la ejecución en las instrucciones beq, además de ahorrar hardware en la ruta de datos (3 registros).

3.7. Procesador segmentado

Introducción a la segmentación

La idea del procesador segmentado va más allá de la ejecución de instrucciones en varios ciclos, y propone como idea principal la utilización de los módulos que dejan de emplearse en el transcurso de la ejecución de una instrucción para las que la siguen, solapando su ejecución. Este sistema hace un uso óptimo de la ruta de datos, pero plantea algunas complicaciones que veremos en detalle.

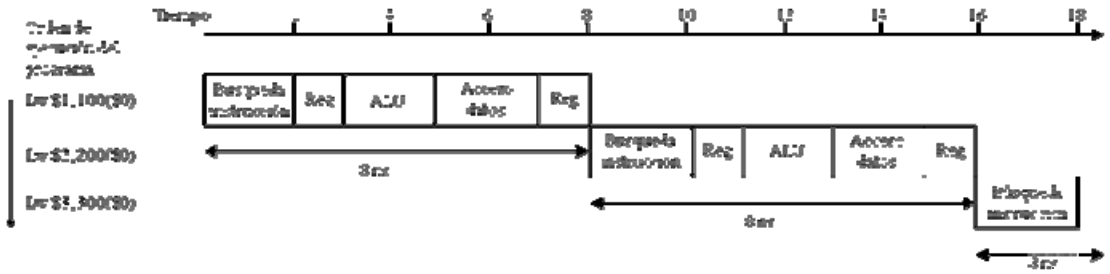


Figura 27. Ejecución de tres instrucciones en el procesador monociclo.

En la Figura 27 hemos planteado un ejemplo de la ejecución de 3 instrucciones Load en un procesador implementado con la ruta de datos monociclo. En el diagrama podemos observar el tiempo de ciclo completo (8 ns) y las distintas subetapas en las que se divide el flujo de datos por la red combinacional. En cada momento, sólo se ejecuta una instrucción y por lo tanto, quedan una gran cantidad de módulos inactivos mientras que se realizan las operaciones. Estas situaciones se agravan cuando es necesario realizar operaciones mas complicadas como divisiones o multiplicaciones que suponen una secuencia de sumas o restas.

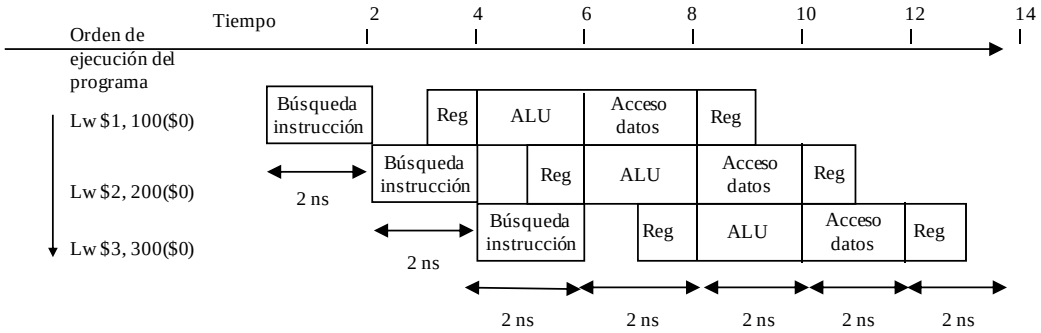


Figura 28. Ejecución de tres instrucciones en el procesador segmentado.

Sin embargo, la Figura 28 representa la ejecución segmentada de las tres mismas instrucciones. Como vemos en la Figura, ahora las instrucciones se realizan en su CPI correspondiente (para el caso del load es 5), con la salvedad que cuando los datos liberan una etapa de la ruta de datos, los módulos que antes quedaban inactivos ahora se utilizan para comenzar la ejecución de las instrucciones siguientes.

Comparando en términos de tiempo de ejecución los diagramas de la Figura 27 y la Figura 28, se aprecia una diferencia más que palpable en cuanto a la velocidad de ejecución de instrucciones. En el caso del diagrama Monociclo el procesador tarda 16 ns en terminar la ejecución, mientras que la versión segmentada tarda 6ns menos. Estas diferencias se hacen mucho mas grandes cuando ejecutamos conjuntos más grandes de instrucciones.

Para facilitar la segmentación, el repertorio de instrucciones debe satisfacer algunos aspectos importantes:

- Todas las instrucciones deben tener igual anchura.
- El repertorio de instrucciones debe contener pocos formatos de instrucción, lo que dota de mayor simplicidad a la ruta de datos.
- La búsqueda de operandos en memoria debe estar presente sólo en operaciones de carga y almacenamiento.

Pero a su vez también añade problemas adicionales, en su mayoría provoca conflictos estructurales, que pueden ser de los tipos siguientes: conflictos de datos, conflictos de control, gestión de interrupciones, y ejecución fuera de orden.

Teniendo estas ideas en mente, vamos a pasar a la construcción del procesador, empezando por la ruta de datos. Por simplicidad, se construye el procesador con el conjunto reducido de instrucciones. Posteriormente este diseño se expande en el simulador para soportar todas las instrucciones mencionadas en este proyecto.

Ruta de datos del MIPS segmentado

Para la elaboración de la ruta de datos segmentada partiremos inicialmente de la ruta en su versión monociclo y de la idea ya comentada de la segmentación, intentando arreglar los problemas que se van planteando y ajustando el circuito hasta que todo quede completamente especificado.

La ejecución se dividirá en 5 fases, cada una de las cuales debe realizarse en un ciclo. Partiremos de las fases ya comentadas , generales a todas las instrucciones :

En la fase de **búsqueda de la instrucción** obtendremos de la memoria la siguiente instrucción a ejecutar, indicada por el Contador de Programa, y la guardaremos en un registro para procesarla posteriormente.

En la fase de **decodificación** sacaremos del registro IR todos los datos necesarios (registros de los que leer/escribir , operando inmediato…) para comenzar los cálculos que realizara la ALU posteriormente.

En la fase de **ejecución** calcularemos las direcciones o los resultados de las operaciones que demande la ejecución de la instrucción en curso.

En la fase de **memoria** leeremos o escribiremos en la memoria los datos que precisemos para instrucciones de carga o almacenaje.

Por último, en la fase de **escritura** finalizaremos la ejecución guardando los datos y los resultados de las operaciones realizadas en el banco de registros

En resumen, las cinco etapas en que se divide la ruta de datos son:

- IF: Búsqueda de la instrucción e incremento del PC
- ID: Decodificación de la instrucción y búsqueda de operandos en los registros
- EX: Ejecución
- MEM: Acceso a la memoria de datos
- WB: Escritura en el banco de registros.

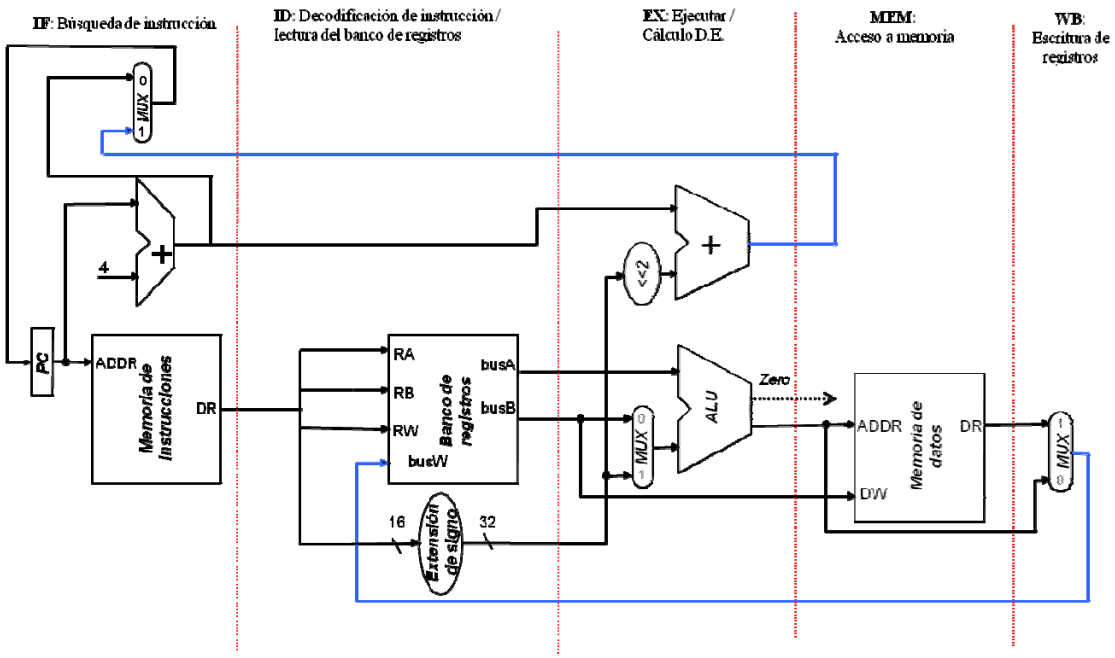


Figura 29. Ruta de datos segmentada (1)

Partiendo de esta ruta de datos inicial (Figura 29), necesitamos conservar los valores intermedios que toman los datos, para que no se pierdan en cuanto comience la siguiente instrucción su ejecución. Para ello emplearemos registros verticales entre cada bloque de la ruta de datos. La ruta de datos resultante será la de la Figura 30.

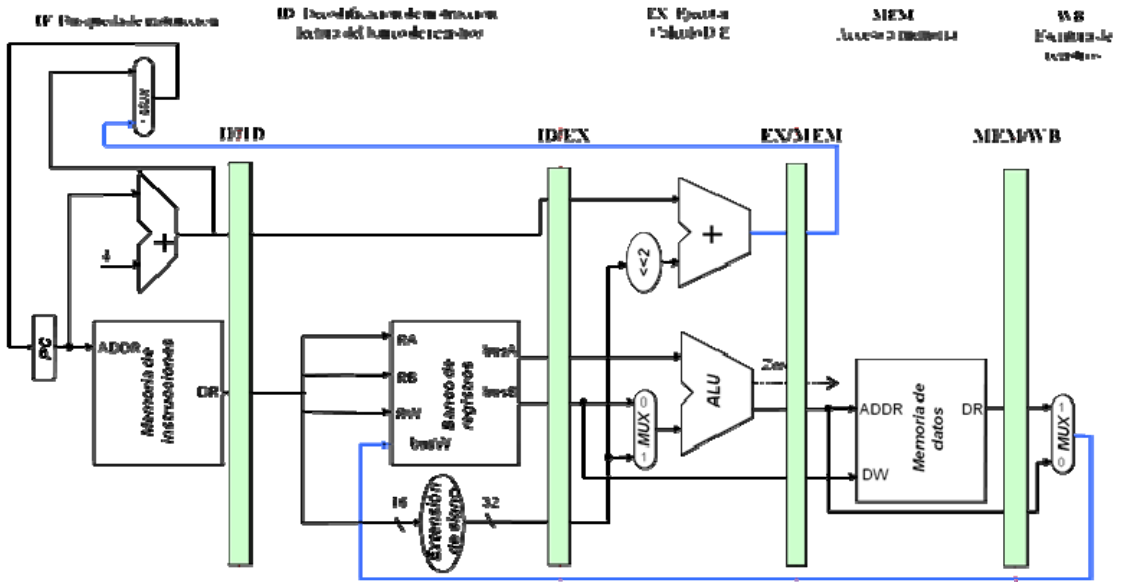


Figura 30. Ruta de datos segmentada (2)

Esta ruta de datos todavía no es suficiente para todo el conjunto de instrucciones implementado anteriormente en la ruta multiciclo. Como vemos en una instrucción Load el dato que entra al banco de registros por el puerto RW en la fase WB de la ejecución seria el obtenido de las instrucciones posteriores, no de la ejecutada inicialmente. Para solucionar esto necesitamos propagar también el valor de entrada RW al banco de registros a través de los registros colocados entre etapa y etapa. El resultado quedaría tal y como se refleja en la Figura 31.

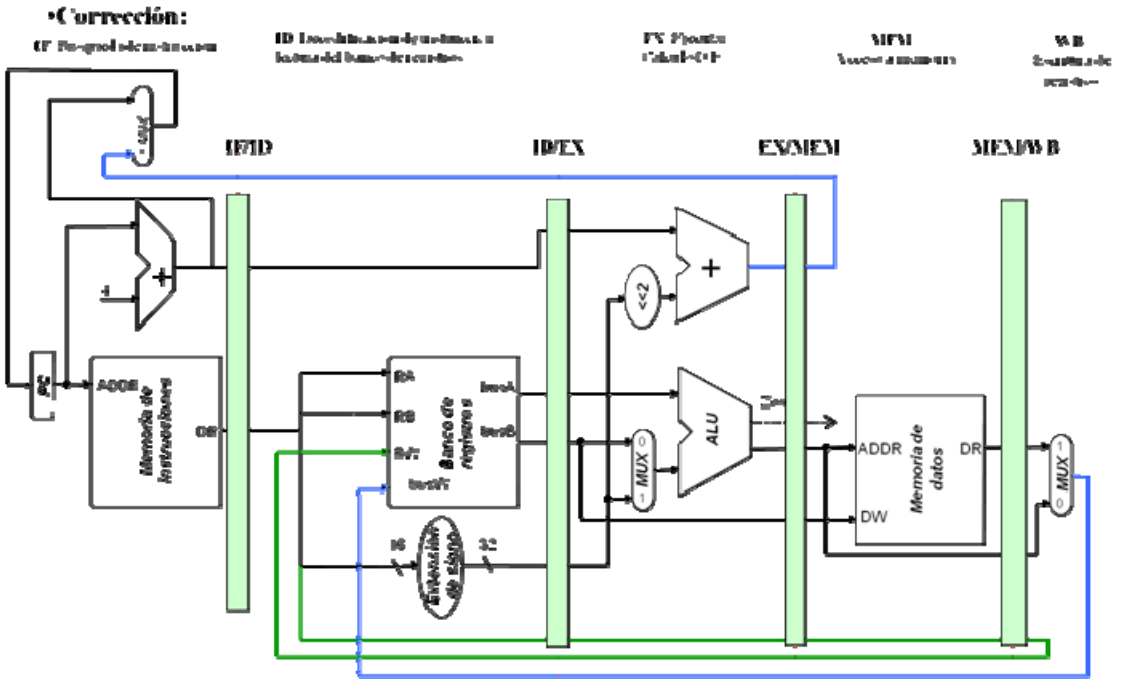


Figura 31. Ruta de datos segmentada (3)

Con esta corrección sí que llegaría de forma correcta la entrada al puerto RW en la última fase de ejecución.

Diseño del control segmentado

Para el diseño del controlador necesitamos tener en cuenta las siguientes señales de control de los registros/módulos combinacionales (Figura 32).

Por un lado , necesitaremos una señal para el multiplexor relativo al PC, a la que nombraremos PCSrc, y otra para la Unidad Aritmético Lógica, que denominaremos ALUSrc, que determine si seleccionamos el valor de la salida B del BR o si por el contrario emplearemos el operando inmediato. También necesitaremos otra señal de control que regule el multiplexor MemToReg a la salida de la Memoria y otra para el multiplexor de la fase de ejecución RegDst que propaga los datos a los registros intermedios.

La memoria y el Banco de Registros necesitaran sus señales correspondientes para lectura/escritura: MemRead/MemWrite para la Memoria y RegWrite para el Banco de registros. Por último, necesitamos una señal de control que regule la operación que se realice en la ALU y una señal Branch que nos indique si el salto en ejecución se realiza o no.

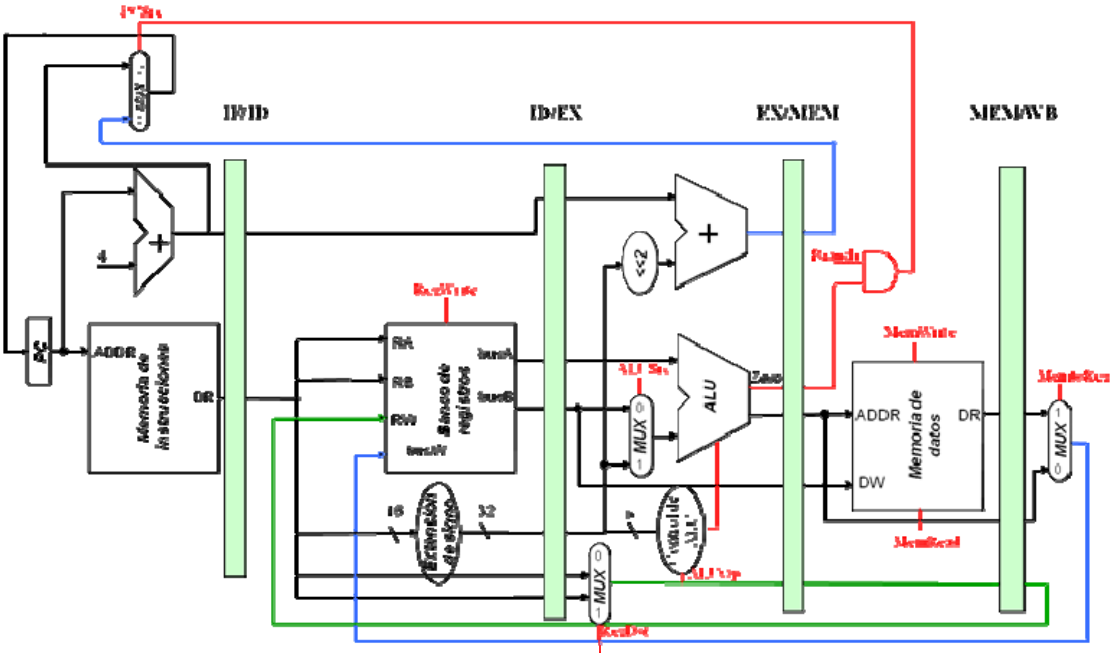


Figura 32. Ruta de datos segmentada (4)

Los valores de las líneas de control se pueden calcular fácilmente examinando la ruta de datos de la Figura 32, tal como ilustra la siguiente tabla.

Instrucción	Líneas de control del estado de Ejecución/Cálculo de DE				Líneas de control del estado de acceso a memoria			Líneas de control del estado de WB	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Mem to Reg
Formato-R	1	1	0	0	0	0	0	1	0
Lw	0	0	0	1	0	1	0	1	1
Sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Las señales de control se generan en la Unidad de Control y se propagan como un dato más a través de los registros (ver Figura 33).

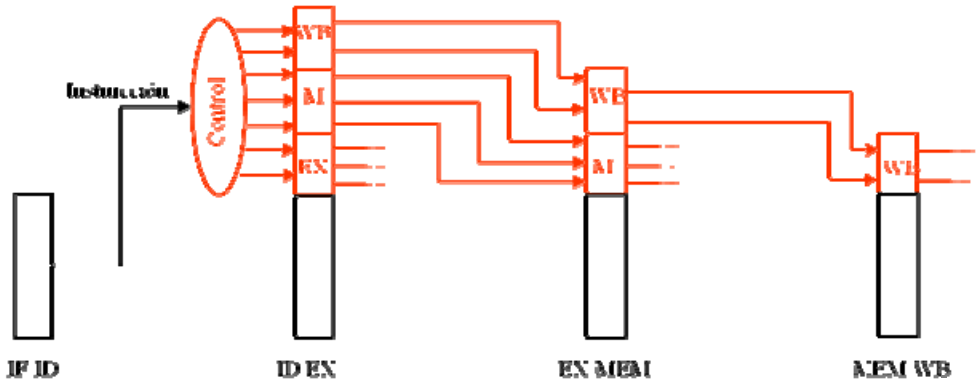


Figura 33. Propagación de las señales de control.

Empleando esta idea, vamos a finalizar la ruta de datos. Cada etapa de la ejecución tendrá asignados los valores de control de los módulos contenidos en ella (ver Figura 34).

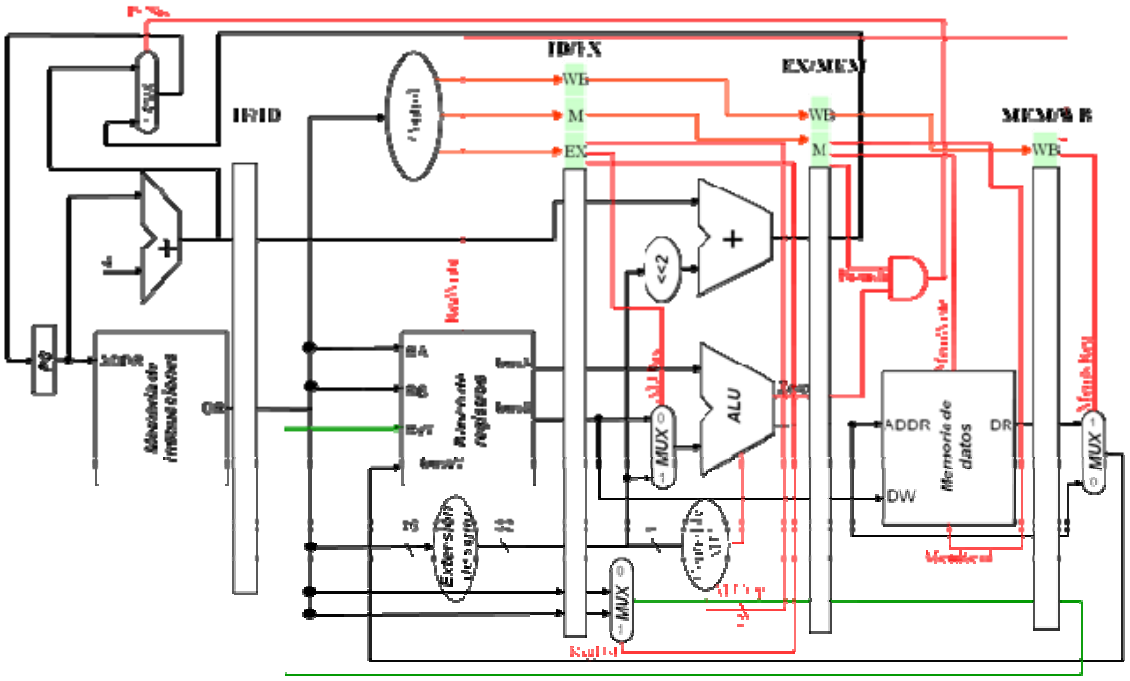


Figura 34. Ruta de datos segmentada.

La Figura 34 representa la ruta de datos segmentada completa. Como podemos observar, en ella ya aparecen las señales de control, ya cableadas para que se propaguen a la vez que los datos a través de los registros situados entre las etapas.

Como vemos la ruta de datos segmentada requiere de bastante hardware para que funcione correctamente debido a la necesidad de guardar una cantidad importante de datos entre ciclo y ciclo.

A pesar de que el concepto de la segmentación aporta muchas mejoras respecto a las versiones anteriores, no siempre mejora el rendimiento, como se refleja en las pruebas realizadas posteriormente en la sección de resultados, y no esta exenta de problemas de diseño.

Una de las cuestiones más importantes es la necesidad de anticipar operandos en la ejecución de instrucciones para conseguir un rendimiento adecuado. En esta versión sin anticipación, cada vez que una instrucción necesita un dato tiene que esperar a que la instrucción que utiliza ese dato finalice, y evitar tomar un valor incorrecto. Esta situación se vuelve más grave si cabe si ejecutamos instrucciones de punto flotante, que requieren muchos ciclos para su cálculo, en este tipo de instrucciones cada una de las posteriores debe esperar a que finalice el cálculo en punto flotante, malgastando de esta forma una gran cantidad de tiempo de ejecución. Así pues, es vital contar con la existencia de este tipo de situaciones, y emplear módulos que las detecten y sean capaces de resolverlas.

Este tema lo trataremos en la siguiente sección.

MIPS segmentado con anticipación de operandos

El MIPS segmentado presenta tres tipos de conflictos: conflictos de datos, conflictos de control y conflictos estructurales. Comenzamos con los conflictos de datos. En términos generales, existen tres tipos posibles, escritura después de lectura, escritura después de escritura y lectura después de escritura. A continuación mostramos un ejemplo indicando cuáles requieren de modificaciones en el procesador.

Conflictos por escritura después de lectura (no existen en el MIPS):

- sub $2, **$1**, $3
- add **$1**, $4, $5

Un conflicto de escritura después de lectura se produce cuando un dato que se está empleando para una instrucción ha de ser modificado en la siguiente.

Conflictos de escritura después de escritura (no existen en el MIPS):

- sub **$2**, $1, $3
- add **$2**, $4, $5

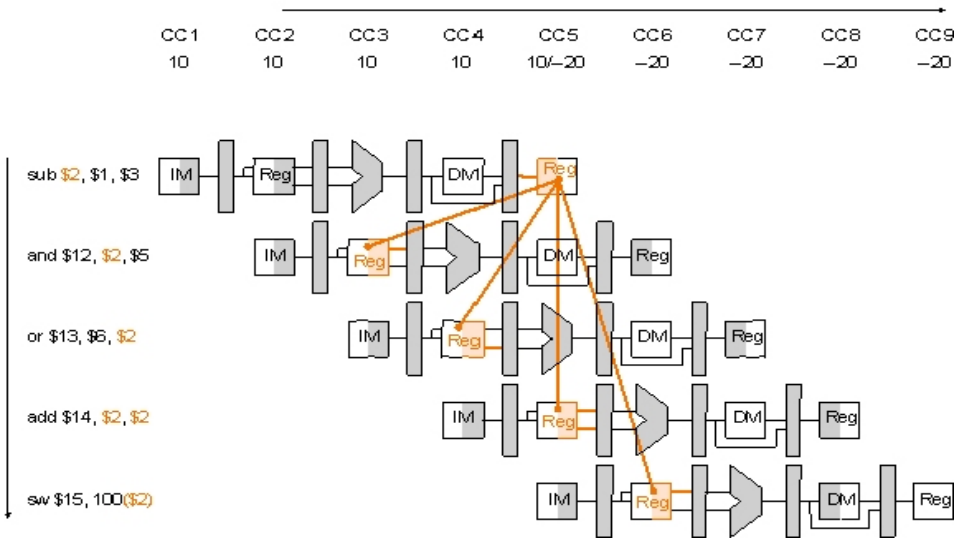
Un conflicto de escritura después de escritura se produce cuando tratamos de escribir un dato tras una escritura del mismo dato.

Conflictos de lectura después de escritura (LDE) (hay que modificar el contol y ruta de datos para subsanarlos):

- sub **$2**, $1, $3
- add $4, **$2**, $5

Este tipo de conflictos se producen cuando necesitamos consultar un dato tras una escritura de ese dato. Si, según la idea de segmentación, la escritura no ha finalizado por completo podemos usar para el cálculo de la instrucción add un valor incorrecto, y alterar el resultado del flujo de instrucciones del programa, incurriendo en gran cantidad de errores.

En la ruta de datos segmentada de la Figura 34 no hemos tenido en cuenta aún este aspecto, asi que tendremos que modificar la ruta de datos y el control segmentado para subsanar el problema.



La Figura ilustra esta situación. Como las instrucciones posteriores a la resta(sub) requieren el contenido del registro **$2** para efectuarse, en caso de no existir anticipación deberían esperar a que la ALU calculase el valor y, mediante un cable conseguir el valor antes de que la instrucción finalice. Este tipo de situaciones se subsanarán con una Unidad de Anticipación (Forwarding Unit) que trataremos más adelante.

Conflictos de control: caso del Load

En cuanto a los conflictos de control, existe un problema particular ante un conflicto LDE provocado por una instrucción lw . Para una instrucción tipo lw no es suficiente con anticipar los datos puesto que la escritura en el Banco de registros se produce en la fase de Escritura, asi que cualquier instrucción posterior que emplee el registro destino estará tomando valores incorrectos de éste.

Así pues, es necesario detectar esta situación y rellenar con instrucciones NOP(instrucciones vacías) hasta conseguir el valor del registro destino esperado. Vamos a ilustrar este tipo de situaciones con un ejemplo:

Ejemplo de conflicto con lw

`lw r1, 0(r0)`
`sub r4, r1, r5`
`and r6, r1, r7`
`or r8, r1, r9`
`xor r10, r1, r11`

Como vemos en el código del ejemplo, el Load escribe en el registro r1, y el sub posterior emplea ese resultado para una resta. La anticipación no nos garantizaría el valor correcto, asi que es necesario que la instrucción llegue a la fase de escritura para poder anticipar el operando y continuar la ejecución de la siguiente instrucción (Figura 35).

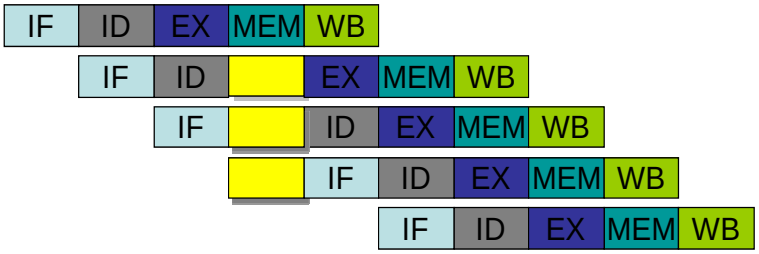


Figura 35. Ejecución de lw con dependencias. Como es imposible anticipar operandos, la ruta de datos tiene que insertar una instrucción nop para solventar el problema.

Conflictos de control: saltos

Finalmente, se produce un conflicto estructural principalmente ante instrucciones de salto. Si un salto se produce, la ruta de datos debe eliminar las instrucciones previamente cargadas en la ruta de datos.

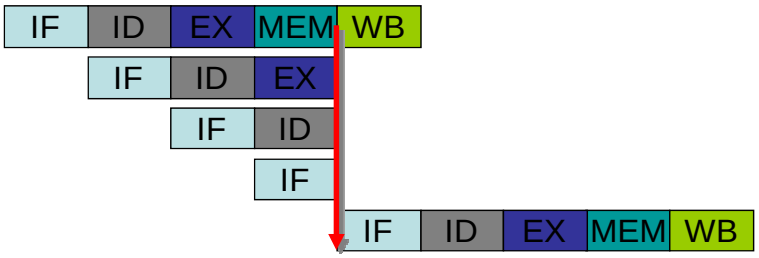


Figura 36. Ejemplo de instrucción beq. beq es la primera instrucción. Si se salta, esta deberá limpiar las tres siguientes instrucciones, ya que no se puede permitir que estas finalizen.

La Figura 36 muestra un ejemplo de ejecución de la instrucción beq. La ruta de datos decide si se salta o no en la etapa MEM (ver Figura 36). Cuando se decide saltar, se deben eliminar las tres instrucciones posteriores a beq, ya que se encuentran en ejecución en la ruta de datos.

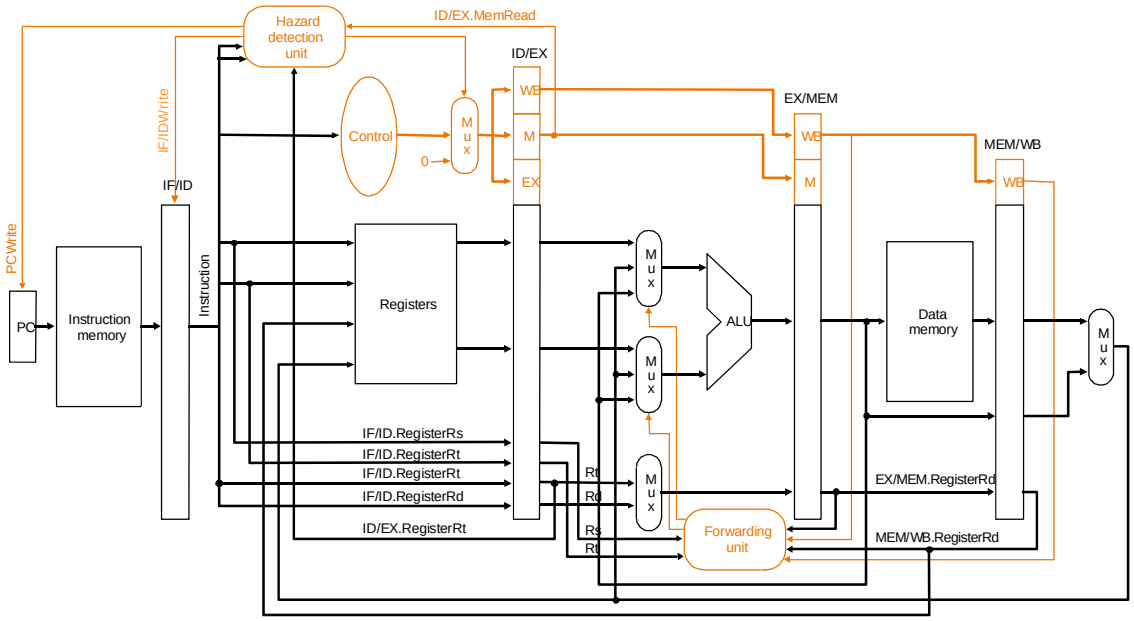


Figura 37. MIPS segmentado con anticipación y bloqueo.

La Figura 37 muestra el diseño final (simplificado y a título ilustrativo) de la ruta de datos con anticipación de operandos y bloqueo del cauce. Dispone de dos unidades para resolver estos conflictos.

La unidad **"Forwarding Unit"** se encargará de anticipar los datos directamente desde la salida de los registros intermedios y seleccionar las entradas de los multiplexores a la entrada de la ALU que garanticen un resultado correcto de las instrucciones posteriores.

Por otro lado, la **"Hazard Detection Unit"** se encargará de detectar los riesgos estructurales provocados por conflictos de control e insertar instrucciones NOP(bloquear el pipeline) cuando sea necesario(caso del Load).

La siguiente tabla resume las acciones que hay que tomar en función de los distintos conflictos.

Conflictos de datos

`Riesgo EX:`
`if (XM.RegWrite && (XM.Rd != 0) && (XM.Rd == DX.Rs))`
`  anticiparA = 10`
 	
`if (XM.RegWrite && (XM.Rd != 0) && (XM.Rd == DX.Rt))`
`  anticiparB = 10`
 	
`Riesgo MEM:`
`if (MW.RegWrite && (MW.Rd != 0) && (XM.Rd != DX.Rs) && (MW.Rd == DX.Rs))`
`  anticiparA = 01`
 	
`if (MW.RegWrite && (MW.Rd != 0) && (XM.Rd != DX.Rt) && (MW.Rd == DX.Rt))`
`  anticiparB = 01`

Conflicto de control con lw
`if (DX.MemRead && ((DX.Rt == FD.Rs) || (DX.Rt == FD.Rt)))` `  bloquear el pipeline;`
Conflicto estructural con beq
`Si se salta, la instrucción debe borrar las tres etapas anteriores.`

A continuación, ilustramos con un ejemplo los problemas que pueden surgir debido a los conflicto de datos. Estos problemas se resuelven en la ruta de datos de la Figura 37.

Ejemplo: Ejecución de instrucciones en un procesador segmentado

El siguiente fragmento de código se ejecuta en el MIPS con segmentación de la Figura 34:

Programa de ejemplo
`sub $1,$2,$3`
`add $4,$5,$6`
`sub $5,$4,$8`
`add $7,$2,$3`
`add $9,$7,$3`
`lw $1,10($6)`
`add $3,$1,$4`
`sub $6,$7,$8`

Con este supuesto vamos a calcular: (a) El número de ciclos necesarios para ejecutar este código si no existe la posibilidad de anticipar operandos o reordenar el código, (b) el número de ciclos si existe anticipación de operandos, pero no reordenación de código. Finalemente, (c) tratar de reordenar el código para conseguir que el número de ciclos sea mínimo, si no hay anticipación de operandos. ¿Cuántos ciclos son necesarios en este caso?

Las dependencias son las que aparecen en negrita:

Programa de ejemplo
`sub $1,$2,$3`
`add` **`$4`**`,$5,$6`
`sub $5,`**`$4`**`,$8`
`add` **`$7`**`,$2,$3`
`add $9,`**`$7`**`,$3`
`lw` **`$1`**`,10($6)`
`add $3,`**`$1`**`,$4`
`sub $6,$7,$8`

Si no hubiese dependencias las 8 instrucciones tardarían en ejecutarse 12 ciclos, pero al haber tres dependencias se necesitan 9 ciclos más, en total 21.

Al utilizar la anticipación de operandos desaparece la penalización por conflictos LDE entre instrucciones aritmético-lógicas, pero no desaparece totalmente cuando el conflicto es entre una instrucción LW y la siguiente instrucción utiliza el dato buscado en memoria. En este caso existe un ciclo de penalización, es decir Nº de ciclos total = 13.

Finalmente, el objetivo de la reordenación de código es separar lo más posible las instrucciones con dependencias LDE, sin modificar la funcionalidad del código, y sin producir nuevas dependencias de datos.

En este caso es bastante sencillo:

Programa de ejemplo
`add $4,$5,$6`
`sub $1,$2,$3`
`add $7,$2,$3`
`sub $5,$4,$8`
`lw $1,10($6)`

```
add $9,$7,$3
sub $6,$7,$8
add $3,$1,$4
```

Con esto, las dependencias han desaparecido y la ejecución durará 12 ciclos.

4. Implementación del procesador MIPS32 usando DEVS

4.1. Introducción

DEVS es una abreviación de **D**iscrete **Ev**ent **S**ystem specification. Fue introducido por Bernard Zeigler a mediados de la década de los 70 [1]. DEVS permite representar todos aquellos sistemas cuyo comportamiento pueda describirse mediante una secuencia de eventos. DEVS proporciona un formalismo teórico para el modelado de sistemas de eventos discretos como composición de subsistemas. Cada subsistema puede ser un modelo simple (atómico) o compuesto (acoplado). Cada modelo puede integrarse en un nivel de jerarquía permitiendo testear la corrección de los modelos.

Un modelo DEVS procesa una serie de eventos de entrada y según estos y el estado actual, provoca una secuencia de eventos de salida.

Figura 38. DEVS - esquema general.

4.2. Descripción

Modelo DEVS simple

Un modelo DEVS atómico viene dado por la 7-tupla:

$$M = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta >$$

Donde

X: Conjunto de eventos de entrada

S: Conjunto de estados

Y: Conjunto de eventos de salida

$\delta_{int}: S \rightarrow S$, función de transición interna

$\delta_{ext}: Q \times X \rightarrow S$, función de transición externa, con

$$\mathrm{Q} = \{ (\mathrm{s}, \mathrm{e}) \ / \ \mathrm{s} \in \mathrm{S}, \mathrm{e} \in [0, \mathrm{ta}(\mathrm{s})] \}$$

$\lambda: S \rightarrow Y$, función de salida

$ta: S \rightarrow R_0^+$, función de duración (avance temporal)

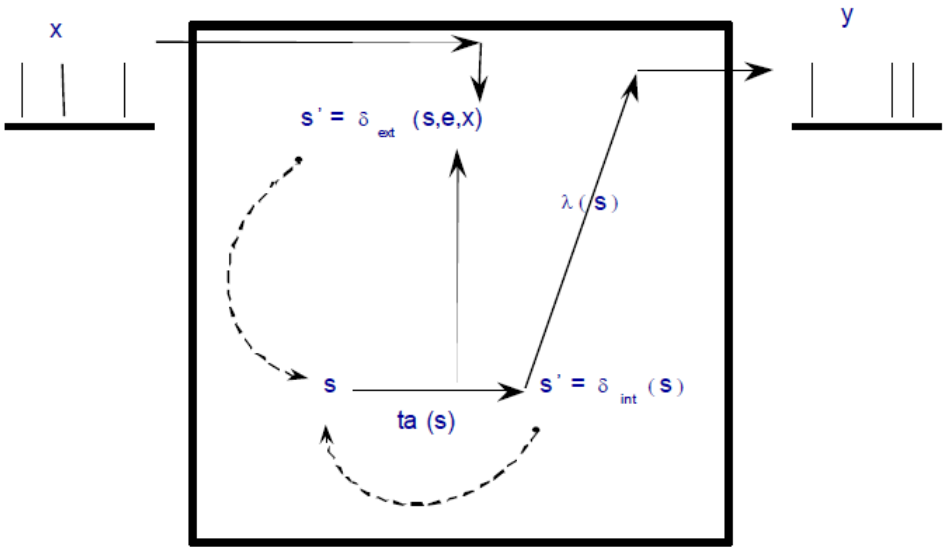


Figura 39. Secuencia de ejecución de un modelo DEVS.

La Figura 39 ilustra la secuencia de ejecución de un modelo DEVS. Los modelos se comunican entre sí a través de puertos, de manera que nos permiten mandar eventos a otros modelos y recibir eventos de otros modelos. Cada estado del modelo tiene un tiempo de vida definido por la función de duración. Cuando el tiempo de vida asociado a un estado acaba, la transición interna se activa y se produce un cambio interno de estado. Después de este cambio, el estado actual puede reflejarse en los puertos de salida. Los valores de salida son enviados por la función de salida, que debe ejecutarse después de la transición interna. En cualquier instante el modelo puede recibir eventos externos de entrada, cuando ocurre la función de transición externa se activa y con la información del estado actual, los valores de entrada y la función de duración se realiza la transición a un nuevo estado, cuyo tiempo de vida se inicia.

Para ilustrar el funcionamiento de un modelo DEVS atómico, vamos a desarrollar el siguiente ejemplo:

"Un modulo de almacenamiento que responde a una entrada la almacena de manera permanente o hasta que llegue una nueva entrada. Para indicar que no llega ningún elemento dejaremos la señal de entrada con el valor 0. Cuando se recibe como señal de entrada un 0, el modelo responde con el último valor almacenado."

A continuación se muestra el pseudocódigo de dicho modelo:

Código de ejemplo

```
Clase Almacenamiento{

  Elemento elemento;
  int retardo;

public Almacenamiento(int ret){
  elemento = 0;
  retardo = ret;
}

  inicializa(){
  fase = "passive";
  sigma = INFINITY;
  Elemento = 0;
```

```
  retardo = 10;
  inicializar();
}

deltaExterna(double e,double input){
  si (phaseIs("passive"))entonces{
  si (input != 0) entonces
  elemento = input;
  sino
  holdIn("respond", retardo);
}
}

deltaInterna( ){
  passivate();
}

lambda( ){
  si (phaseIs("respond")) entonces
    devolver elemento;
  else
    devolver 0;
}
}
```

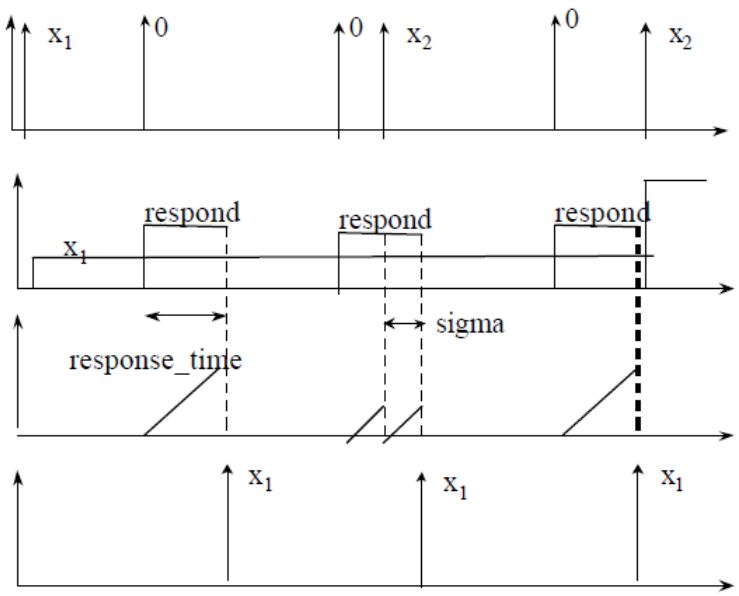


Figura 40. Diagrama temporal de ejecución del ejemplo.

La Figura 40 muestra el diagrama temporal de ejecución del modelo. Existen tres variables de estado: la fase con valores {“pasivo”,”respuesta”}, sigma con valores reales positivos y elemento con valores reales distintos de cero. El valor de fase “respuesta” es necesario para indicar cuando una respuesta está en camino. Sigma guarda el valor de la función de avance (tiempo restante en el estado actual). Cuando una señal a cero llega, sigma pasa a valer el valor del retardo y fase toma el valor “respuesta”. Sin embargo si el sistema está en fase “respuesta” y una señal externa llega, la entrada es ignorada, como puede verse en la segunda parte de la Figura 40. Cuando el tiempo de retardo expira, la función de salida produce el valor almacenado y la transición interna transita al estado “pasivo”. Quedaría por decidir qué sucede cuando una señal de entrada llega justo en el momento que se acaba el tiempo de estado (tercera parte de la Figura 40).

Modelo DEVS acoplado

Podemos usar modelos atómicos DEVS para construir modelos acoplados, definidos por:

$$CM = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, select >$$

Donde

X: Conjunto de eventos de entrada

Y: Conjunto de eventos de salida

D: Índice de componentes (∀i∈D)

M_i: Modelo DEVS básico

I_i: Influencias del modelo i (∀j ∈ Ii)

Z_{ij}: $Y_i \rightarrow X_j$, función de translación de i a j

$select$: Criterio de selección en caso de "empate"

Cada modelo acoplado consiste en una serie de modelos (atómicos o acoplados) conectados a través de puertos entrada/salida. Cada componente se identifica como un número índice. La función de translación usa un índice de influencias creado para cada modelo (I_i). La función de fine que salidas del modelo Mi están conectadas a entradas del modelo M_j. Cuando dos submodelos tienen eventos simultáneos, la función de selección decide cual debe ser actividad primero.

Para ilustrar el funcionamiento de un modelo DEVS acoplado, vamos a desarrollar el siguiente ejemplo:

"El sistema mostrado a continuación se denomina EF-P (experimental frame processor). El modulo Generator genera trabajos en intervalos de tiempo y los envía por el puerto out. El modulo Transducer acepta los trabajos que le llegan del Generator por su puerto arrived y guarda su tiempo de llegada. También acepta trabajos por el puerto solved. Cuando un trabajo llega al puerto solved, el Transducer empareja este trabajo con el trabajo previo que había llegado anteriormente por el puerto arrived y calcula la diferencia de tiempos. Los modulos Generator y Transducer forman un modelo acoplado llamado Experimental frame. El Experimental frame manda los trabajos del puerto out del Generator al modulo Processor que los reenvía tras un periodo de tiempo al puerto solved del modulo Transducer. Si el Processor está ocupado cuando le llega un nuevo trabajo, lo descarta. Finalmente el Transducer para la generación de trabajos enviando un evento cualquiera por su puerto out al puerto stop del Generator"

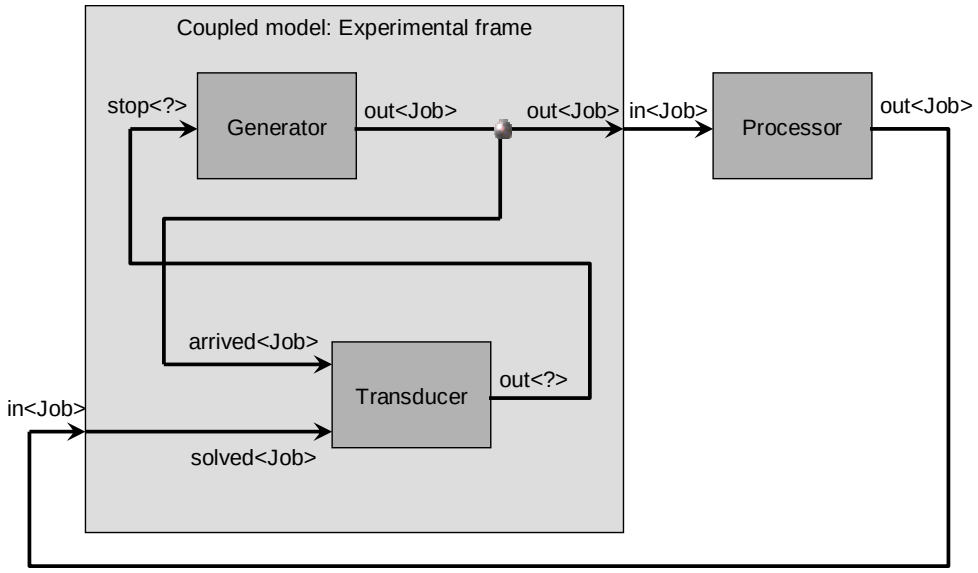


Figura 41. Model DEVS acoplado ef-p.

En la Figura 41 podemos observar el modelo acoplado formado por dos submodelos EF y P. El submodelo EF está a su vez formado por dos submodelos G y T El acoplamiento del conjunto es posible mediante el acoplamiento de los puertos entre modelos:

- Del puerto out de G, al puerto out de EF
- Del puerto out de G, al puerto $arrived$de T
- Del puerto out de T, al puerto $stop$ de G
- Del puerto in de EF, al puerto $solved$de T
- Del puerto out de EF, al puerto in de P
- Del puerto out de P, al puerto in de EF

Una vez acoplado todos los submodelos, el modelo $EF - P$ puede usarse como un nuevo modelo atómico.

Simulación de sistemas DEVS

Los modelos DEVS pueden ser simulados de manera muy sencilla y eficiente.

El siguiente algoritmo puede usarse para simular modelos DEVS:

1. Identificamos el modelo atómico que según la función de avance de tiempo (ta) y el tiempo transcurrido debe ser el siguiente en realizar la transición interna (δ_{int}). Llamaremos d^*a dicho sistema y denominamos tn al tiempo de dicha transición.
2. Avanzamos el tiempo total de la simulación t hasta $t = tn$ y ejecutamos la función de transición interna (δ_{int}) del modelo d^*.
3. Propagamos el evento de salida producido por d^* a todos los modelos atómicos conectados al puerto de salida de dicho evento y ejecutamos las funciones de transición externas correspondientes (δ_{int}).
4. Volvemos al paso 1.

Para implementar este algoritmo en un programa debe usarse una estructura jerárquica del modelo.

Cada modelo atómico tiene asociado un simulador DEVS, mientras que cada modelo acoplado tiene asociado un coordinador DEVS. La Figura 42 ilustra esta correspondencia.

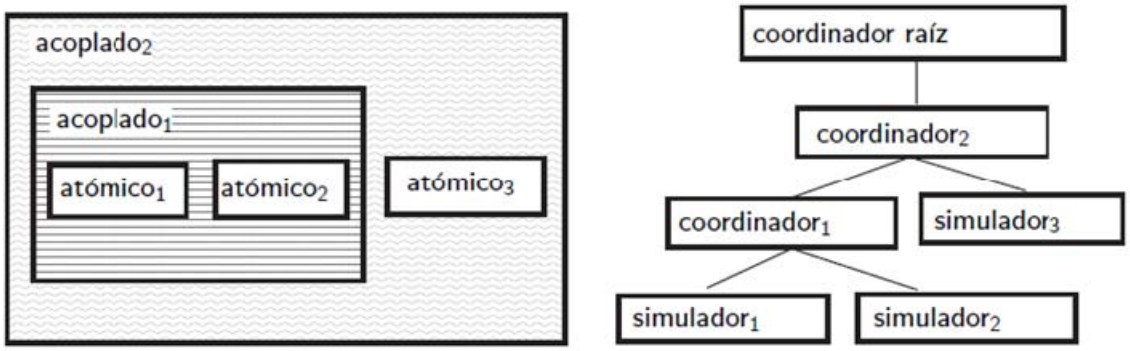


Figura 42. Correspondencia entre modelos y simuladores.

Existen gran variedad de paquetes y herramientas que nos permiten desarrollar modelos DEVS y simular directamente sistemas DEVS, algunas de ellas son:

- DEVS/C++
- DEVSJAVA
- PowerDevs
- XDEVS

En el proyecto que se desarrolla a continuación hemos escogido XDEVS por estar desarrollado en el Departamento de Arquitectura de Computadores y Automática de la Universidad Complutense de Madrid. Además, xDEVS ha sido validado con la simulación de varios modelos publicados en congresos y revistas de ámbito internacional [2], [3].

XDEVS

xDEVS es una plataforma de modelado y simulación DEVS implementada en JAVA. Está disponible como código abierto en sourceforge (http://www.sourceforge.net), concretamente en la página web: https://sourceforge.net/projects/xdevs/.

xDEVS permite el modelado de componentes atómicos y acoplados. También cuenta con una extensa librería de componentes predefinida, principalmente orientadas a control de vehículos aéreos y otros elementos de ámbito militar. También cuenta con algunos modelos diseñados para la optimización usando programación evolutiva. La simulación de modelos se realiza con Coordinadores, implementados para ejecutar simulaciones en tiempo virtual o en tiempo real.

Para ilustrar el uso de xDEVS, mostramos ahora el ejemplo de implementación de la Figura 41. En los siguientes listados se muestran los diferentes elementos presentes en el modelo:

Modelo Generator

```
public class Generator extends Atomic {
    public static final String inPortStop = "stop";
    public static final String outPortOut = "out";
       // Ports
       protected Port<Job> stop;
       protected Port<Job> out;
```

```
      // State
      protected double period;
      protected int count;

      public Generator(String name, double period) {
            super(name);
        stop = new Port<Job>(Generator.inPortStop);
        out = new Port<Job>(Generator.outPortOut);
        super.addInport(stop);
        super.addOutport(out);
            this.period = period;
            this.holdIn("active", period);
            count = 0;
      }

    public Generator(String name) {
            this(name, 1);
      }


      public void deltint() {
            count++;
            this.holdIn("active", period);
      }

      public void deltext(double e) {
        super.resume(e);
            super.passivate();
      }

      public void lambda() {
            Job job = new Job("" + count + "");
        out.setValue(job);
      }
}
```

Modelo Processor

```
public class Processor extends Atomic {
    public static final String inPortIn = "in";
    public static final String outPortOut = "out";
      // Ports
      protected Port<Job> in;
      protected Port<Job> out;
      // State
      protected Job currentJob;
      protected double processingTime;

      public Processor(String name, double processingTime){
            super(name);
        in = new Port<Job>(Processor.inPortIn);
        out = new Port<Job>(Processor.outPortOut);
        super.addInport(in);
        super.addOutport(out);
            this.processingTime = processingTime;
            currentJob = null;
      }

    public Processor(String name) {
        this(name, 2.5);
      }


      public void deltint() {
        super.passivate();
            currentJob = null;
```

```
      }

      public void  deltext(double e) {
        super.resume(e);
             if (super.phaseIs("passive")) {
                   Job job = in.getValue();
             currentJob = job;
            super.holdIn("active", processingTime);
             }
      }

      public void lambda() {
        if(super.phaseIs("active"))
            out.setValue(currentJob);
    }

}
```

Modelo Transducer

```
public class Transducer extends Atomic {
    private           static           Logger            logger           =
Logger.getLogger(Transducer.class.getName());

    public static final String inPortArrived = "arrived";
    public static final String inPortSolved = "solved";
    public static final String outPortOut = "out";
      // Ports
      protected Port<Job> arrived;
      protected Port<Job> solved;
      protected Port<Job> out;

      // State
      protected ArrayList<Job> jobsArrived;
      protected ArrayList<Job> jobsSolved;
      protected double observationTime;
      protected double total_ta;
      protected double clock;

      public Transducer(String name, double observationTime) {
             super(name);
             arrived = new Port<Job>(Transducer.inPortArrived);
             solved = new Port<Job>(Transducer.inPortSolved);
             out = new Port<Job>(Transducer.outPortOut);
        super.addInport(arrived);
        super.addInport(solved);
        super.addOutport(out);
             jobsArrived = new ArrayList<Job>();
             jobsSolved = new ArrayList<Job>();
             this.observationTime = observationTime;
        super.holdIn("active", observationTime);
             total_ta = 0;
             clock = 0;
      }

      public Transducer(String name) {
        this(name, 60);
      }

      public void deltint() {
             clock = clock + getSigma();
             double throughput;
             double avg_ta_time;
             if(!jobsSolved.isEmpty()) {
                   avg_ta_time = total_ta / jobsSolved.size();
                   if (clock > 0.0) throughput = jobsSolved.size() / clock;
                   else throughput = 0.0;
```

```
              }
              else {
                     avg_ta_time = 0.0;
                     throughput = 0.0;
              }
        if(logger.isLoggable(Level.INFO)) {
            logger.info("End time: " + clock);
            logger.info("jobs arrived : " + jobsArrived.size ());
            logger.info("jobs solved : " + jobsSolved.size());
            logger.info("AVERAGE TA = " + avg_ta_time);
            logger.info("THROUGHPUT = " + throughput);
        }
        super.passivate();
      }

      public void deltext(double e) {
        super.resume(e);
             clock = clock + e;

        Job job = null;

        if(!arrived.isEmpty()) {
            job = arrived.getValue();
            if(logger.isLoggable(Level.INFO))
                logger.info("Start job " + job.name + " @ t = " + clock);
                    job.time = clock;
                    jobsArrived.add(job);
             }

             if(!solved.isEmpty()) {
            job = solved.getValue();
                    total_ta += (clock - job.time);
            if(logger.isLoggable(Level.INFO))
                logger.info("Finish job " + job.name + " @ t = " + clock);
                    job.time = clock;
                    jobsSolved.add(job);
             }
      }


      public void lambda() {
             Job job = new Job("null");
        out.setValue(job);
      }

}
```

Modelo acoplado EF

```
public class Ef extends Coupled {
    public static final String inPortIn = "in";
    public static final String outPortOut = "out";
       // Ports
       protected Port<Job> in;
       protected Port<Job> out;

       public Ef(String name, double period, double observationTime) {
              super(name);
        in = new Port<Job>(Ef.inPortIn);
        out = new Port<Job>(Ef.outPortOut);
        super.addInport(in);
        super.addOutport(out);

              Generator generator = new Generator("Generator", period);
              Transducer    transducer    =    new    Transducer("Transducer",
observationTime);
              addComponent(generator);
```

```
            addComponent(transducer);

            addCoupling(this, in, transducer, transducer.solved);
            addCoupling(generator, generator.out, this, out);
            addCoupling(generator,         generator.out,         transducer,
transducer.arrived);
            addCoupling(transducer,         transducer.out,         generator,
generator.stop);
        }
}
```

Modelo acoplado raíz (ef-p) y simulación en xDEVS

```
public class Efp extends Coupled {

    public Efp(String name) {
        super(name);
        Processor processor = new Processor("Processor", 2.5);
        Ef ef = new Ef("ExpFrame", 1, 60);
        addComponent(ef);
        addComponent(processor);
        addCoupling(ef, ef.out, processor, processor.in);
        addCoupling(processor, processor.out, ef, ef.in);
    }

    public static void main(String args[]) {
        Efp efp = new Efp("Coordinator");
        Coordinator coordinator = new Coordinator(efp);
        coordinator.simulate(Long.MAX_VALUE);
    }

}
```

Habiendo explicado DEVS y cómo se implementa un ejemplo usando la biblioteca de componentes xDEVS, pasamos a definir el procesador MIPS32 usando el formalismo DEVS.

4.3. Implementando la arquitectura MIPS como modelo DEVS

La arquitectura del MIPS presentada en la Sección 3 de esta memoria ha sido implementada en su totalidad usando XDEVS.

En primer lugar, para implementar la ruta de datos monociclo, multiciclo y segmentada, se requieren de ciertos elementos comunes a todas ellas: reloj, unidades de memoria (de datos, de instrucciones, o ambas), registros (contador de programa, registro de instrucción, etc.), banco de registros, unidad aritmético-lógica, sumadores, y finalmente, módulos combinacionales y/o secuenciales encargados fundamentalmente del control.

Para ello se ha analizado el comportamiento de cada componente, especificando las entradas, salidas y tiempos, para luego definir cada uno de ellos siguiendo la especificación de DEVS. Luego cada elemento ha sido implementado en XDEVS, para finalmente construir el modelo acoplado general del MIPS conectando todo los submodelos previamente definidos.

Diagrama UML

En primer lugar mostramos el diagrama UML de todos los componentes desarrollados para este proyecto, para después, describir la implementación DEVS de algunos de ellos.

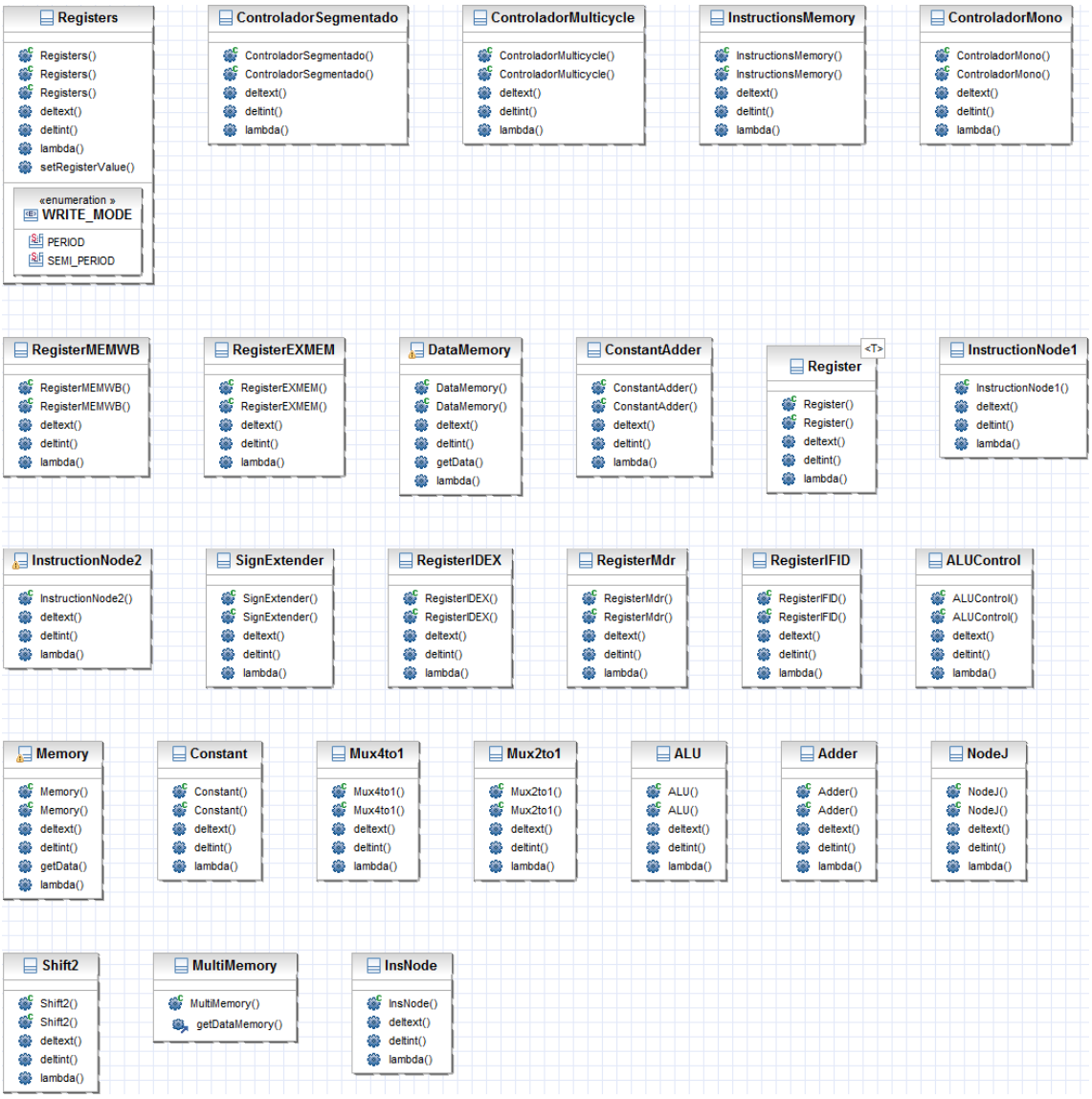


Figura 43. Diagrama de componentes desarrollados en este proyecto.

La Figura 43 ilustra los componentes desarrollados en este proyecto, todos implementados en xDEVS. Estos componentes son los que se utilizan para construir las tres rutas de datos MIPS (monociclo, multiciclo y segmentado), así como los respectivos controladores. Por citar algunos, podemos encontrar multiplextores 2 a 1 y 4 a 1(Mux2to1 y Mux4to1 en la Figura 43), el banco de registros (Registers), la memoria de instrucciones (InstructionsMemory), la memoria de datos (DataMemory), desplazadores (Shift2), la unidad aritmético-lógica (ALU), registros de propósito general (Register), etc. Todos y cada uno de estos componentes derivan de la clase Atomic o Coupled de xDEVS, dependiendo de si se trata un modelo atómico o acoplado, respectivamente.

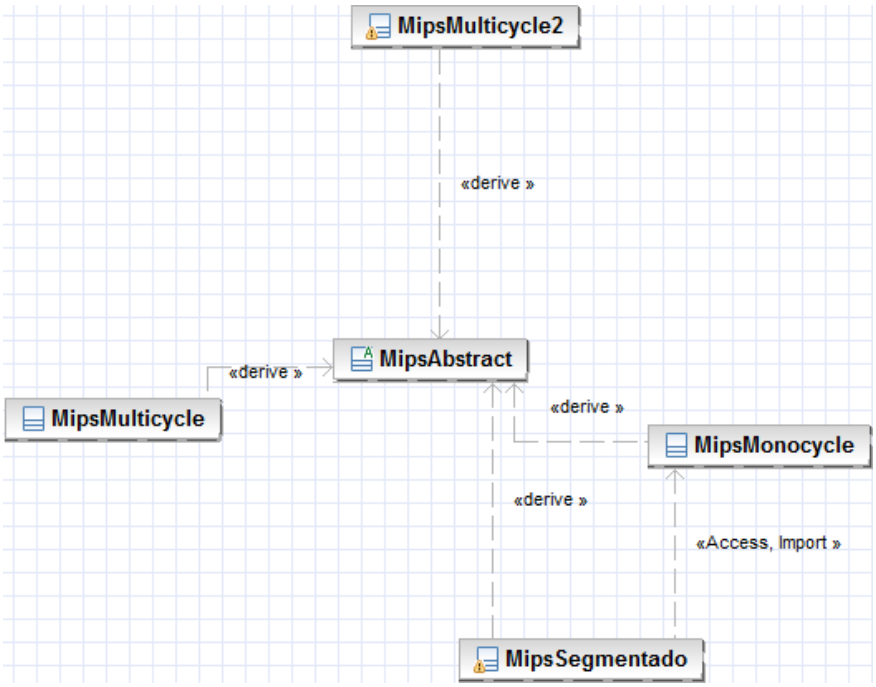


Figura 44. Diagrama de arquitecturas desarrolladas.

La Figura 44 ilustra los cuatro tipos de procesadores MIPS implementados: el procesador monociclo (MipsMonocycle), el procesador multiciclo (MipsMulticycle), el procesador multiciclo optimizado (MipsMulticycle2) y el procesador segmentado (MipsSegmentado). Todos ellos son modelos acoplados que derivan de la clase abstracta MipsAbstract, que a su vez deriva de la clase Coupled de xDEVS.

Estos modelos acoplados únicamente se encargan de crear los componentes de la respectiva ruta de datos y ensamblarlos mediante acoplamientos, como veremos más adelante.

A continuación, definimos utilizando DEVS algunos de los elementos más importantes de los procesadores implementados.

Elementos de la ruta de datos

Mostramos ahora los modelos de cada componente que forma nuestra ruta de datos. Para no indicar constantemente la función de avance de tiempo, suponemos que esta siempre devuelve *sigma*.

Reloj

Representa el reloj de la CPU, donde el tiempo de periodo puede ser configurado.

$$\text{Reloj} = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta >$$

Puertos de entrada: $\emptyset$

Puertos de salida = $\{out\}$

$X \in \emptyset$

$S = \{\sigma \in \mathbb{R}_0^+, y \in \mathbb{R}_0^+ , T \in \mathbb{R}_0^+ \; es\; el\; periodo\; de\; reloj\}$

$S_0 = \{0, 0, T\}$ es el estado en el instante inicial.

$Y \in \{0, 1\}$

$\delta_{int}()$

$\sigma = {}^{T}/_{2}, y = 1 - y$

$\delta_{ext}()$ No hace nada

$\lambda()$

$$out = y$$

Sumador

Representa el modulo sumador, que se encarga de sumar los valores de sus dos entradas y mostrar el resultado por su salida.

$$\text{Sumador} = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta >$$

Puertos de entrada: $\{opA, opB\}$

Puertos de salida = $\{out\}$

$X \in \mathbb{R}$

$S = \{\sigma \in \mathbb{R}_0^+, y \in \mathbb{R}_0^+, t \in \mathbb{R}_0^+ \; es\; tiempo\; de\; retado\}$

$S_0 = \{0, 0, t\}$ es el estado en el instante inicial.

$Y \in \mathbb{R}$

$\delta_{int}()$

$$passivate()$$

$\delta_{ext}()$

$x1 = opA; x2 = opB$

$\lambda()$

$$out = x1 + x2$$

Multiplexor 2 a 1

Representa el modulo multiplexor, que dados dos valores de entrada y una señal de control, se encarga de escoger un valor de entre sus dos entradas y mostrar el resultado por su salida.

$$\text{Multiplexor} = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta >$$

Puertos de entrada: $\{in0, in1, control\}$

Puertos de salida = $\{out\}$

$X \in \mathbb{R}$

$S = \{\sigma \in \mathbb{R}_0^+, y \in \mathbb{R}_0^+ , t \in \mathbb{R}_0^+ \; es\; tiempo\; de\; retado\}$

$S_0 = \{0, 0, t\}$ es el estado en el instante inicial.

$Y \in X$

$\delta_{int}()$

$passivate()$

$\delta_{ext}()$

$x1 = in0; x2 = in1; ct = control$

$\lambda()$

$si\ (ct == 1) entonces$

$out = x1$

$sino$

$out = x2$

Registro

Representa el modulo registro, que dado un valor de entrada y una señal de control, si la señal de control vale 1, almacena el valor de entrada en el registro y si vale 0, saca por el puerto se salido el valor almacenado.

Registro $= < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta >$

Puertos de entrada: $\{in, regWrite, reloj\}$

Puertos de salida = $\{out\}$

$X \in \mathbb{R}$

$S = \{\sigma \in \mathbb{R}_0^+, y \in \mathbb{R}_0^+ , te \in \mathbb{R}_0^+ \; es\; tiempo\; de\; retado\; en\; escritura,$
$tl \in \mathbb{R}_0^+ \; es\; tiempo\; de\; retardo\; en\; lectura\}$

$S_0 = \{0, 0, te, tl\}$ es el estado en el instante inicial.

$Y \in X$

$\delta_{int}()$

$passivate()$

$\delta_{ext}()$

$x = in; ct = regWrite; clk = reloj$

$si\ \big(ct = 1\ \wedge flancoSubida(clk)\big) entonces$

$temp = x$

$\lambda()$

$out = temp$

Memoria de datos

Definimos ahora la memoria de datos, que se encarga de almacenar los datos del programa. La memoria de instrucciones es equivalente, pero no requiere escritura. Finalmente, la memoria combinada de datos e instrucciones del procesador multiciclo se construye simplemente acoplando las dos memorias mencionadas.

$$\text{Memoria} = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta >$$

$$\text{Puertos de entrada: } \{CLK, ADDR, DW, MemRead, MemWrite\}$$

$$\text{Puertos de salida = } \{DR\}$$

$X \in \big\{\{0,1\}, \mathbb{Z}, \mathbb{Z}, \{0,1\}, \{0,1\}\big\}$

$S = \{\sigma \in \mathbb{R}_0^+, data \in \mathbb{Z}^n, tw \in \mathbb{R}_0^+ \text{ es tiempo de retado en escritura},$
$tr \in \mathbb{R}_0^+ \text{ es tiempo de retardo en lectura}\}$

$S_0 = \{\infty, 0^n, tw, tr\}$ es el estado en el instante inicial.

$Y \in \mathbb{Z}$

$\delta_{int}()$

$passivate()$

δ_{ext}(X',e)

$X = X'$

$si(MemRead = 1) entonces$

$\sigma = tr$

$si\ (MemWrite = 1, CLK = \downarrow) entonces$

$data[ADDR] = \text{DW}, \sigma = tw$

$\lambda()$

$Y = data[ADDR]$

Banco de Registros

La unidad entera del procesador diseñado cuenta con un banco de 32 registros. Es una memoria pequeña y rápida utilizada para las operaciones aritmético-lógicas. Al igual que la memoria de datos, es un dispositivo de lectura/escritura.

$$\text{Banco de registros} = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta >$$

$$\text{Puertos de entrada: } \{CLK, RegWrite, RA, RB, RW, busW\}$$

$$\text{Puertos de salida = } \{busA, busB\}$$

$X \in \{\{0,1\}, \{0,1\}, \mathbb{Z}^+, \mathbb{Z}^+, \mathbb{Z}^+, \mathbb{Z}\}$

$S = \{\sigma \in \mathbb{R}_0^+, data \in \mathbb{Z}^n, tw \in \mathbb{R}_0^+ \text{ es tiempo de retado en escritura},$
$tr \in \mathbb{R}_0^+ \text{ es tiempo de retardo en lectura}\}$

$S_0 = \{\infty, 0^n, tw, tr\}$ es el estado en el instante inicial.

$Y \in \{\mathbb{Z}, \mathbb{Z}\}$

$\delta_{int}()$

$passivate()$

$\delta_{ext}(X', e)$

$X = X'$

$si(RA \text{ or } RB) entonces$

$\sigma = tr$

$si\ (RegWrite = 1, CLK = \downarrow) entonces$

$data[RW] = \text{busW}, \sigma = tw$

$\lambda()$

$Y = \{\text{si (RA) } data[RA], si\ (RB)\ data[RB]\}$

Para comprender mejor la relación entre las especificaciones DEVS y su implementación en xDEVS, listamos a continuación el código referente a la memoria de datos.

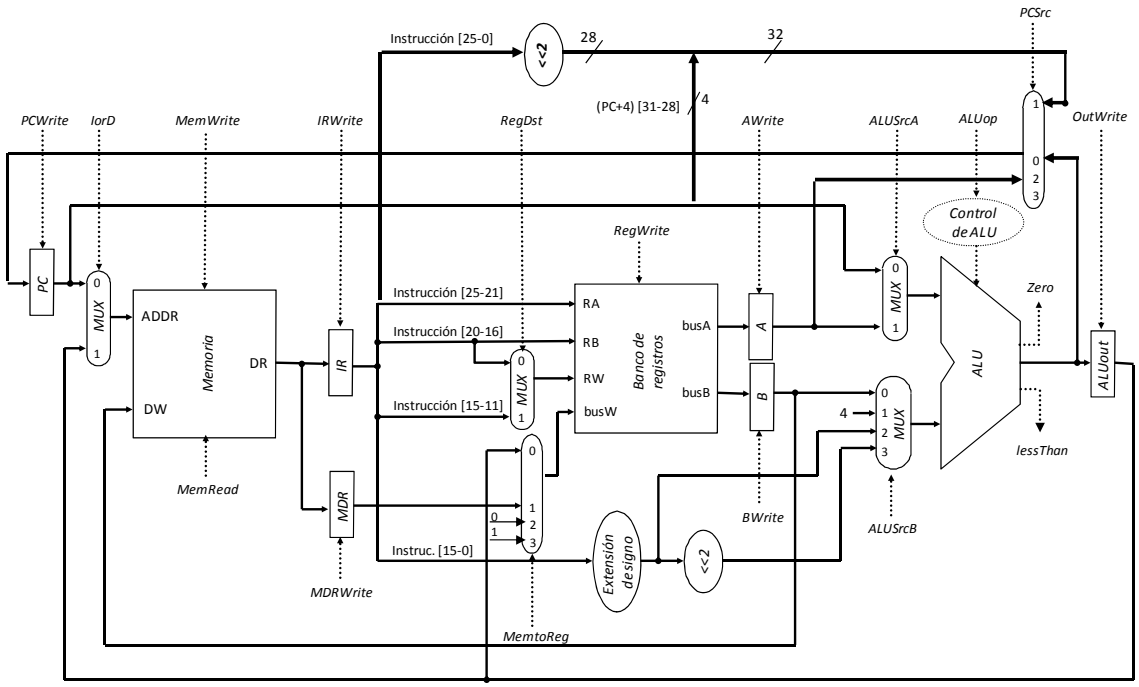
DataMemory

```
public DataMemory(String name, Double delayRead, Double delayWrite) {
  super(name);
  super.addInport(CLK);
  super.addInport(ADDR);
  super.addInport(DW);
  super.addInport(MemWrite);
  super.addInport(MemRead);
  super.addOutport(DR);

  valueAtCLK = null;
  valueAtADDR = null;
  valueAtDW = null;
```

```
  valueAtMemRead = null;
  valueAtMemWrite = null;
  valueToDR = null;

  this.delayRead = delayRead;
  this.delayWrite = delayWrite;

  super.passivate();
}

public DataMemory(String name) {
  this(name, 0.0, 0.0);
}

public void deltint() {
  super.passivate();
}

public void deltext(double e) {
  super.resume(e);
  // Primero procesamos las seÃ±ales de lectura asÃ-ncrona
  if (ADDR.getValue()!=null) {
    valueAtADDR = ADDR.getValue();
  }

  if (!DW.isEmpty()) {
    valueAtDW = DW.getValue();
  }

  if (!MemRead.isEmpty()) {
    valueAtMemRead = MemRead.getValue();
    if(valueAtMemRead==1 && valueAtADDR!=null)
      super.holdIn("Read", delayRead);
  }

  // Ahora las seÃ±ales sÃ-ncronas (escritura)

  if (!MemWrite.isEmpty()) {
    valueAtMemWrite = MemWrite.getValue();
  }

  Integer tempValueAtCLK = CLK.getValue();
  if (tempValueAtCLK != null) {
    if (valueAtMemWrite != null && valueAtMemWrite == 1 && valueAtCLK != null
&& valueAtCLK == 1 && tempValueAtCLK == 0) {
      if (valueAtADDR != null) {
        data.put(valueAtADDR, valueAtDW);
        super.holdIn("Write", delayWrite);
      }
    }
    valueAtCLK = tempValueAtCLK;
  }
}

public void lambda() {
  if(valueAtMemRead!=null && valueAtMemRead==1 && valueAtADDR!=null) {
    valueToDR = data.get(valueAtADDR);
    DR.setValue(valueToDR);
  }
}
```

4.4. Ensamblaje de los elementos para construir las distintas versiones

Para entender mejor el funcionamiento del sistema y cómo los elementos se relacionan entre sí, mostramos a continuación el ensamblaje de los elementos que forman la ruta multiciclo de la Figura 15.



Para ello, usamos la sintaxis definida por XDEVS y que hemos usado para el desarrollo del proyecto. Para una mejor comprensión hemos dividido los distintos acoplamientos agruparlos por categorías según el origen del ensamblado, de tal manera podemos encontrarnos con acoplamientos de la salida de la memoria de instrucciones, de la salida de multiplexores, etc.

Estructura xDEVS del procesador MIPS multiciclo

```
public class MipsMulticycle extends MipsAbstract {
  // Código omitido por brevedad
  // …
  // Components:
  clock = new Clock("Clock", 1200.0);
  //clock.setLoggerOutputActive(true);
  super.addComponent(clock); // multicycle's PC

  pc = new Register<Integer>("PC", 0);
  pc.setLoggerOutputActive(true);
  super.addComponent(pc);

  iorD = new Mux2to1("IorD");
  super.addComponent(iorD);

  memory = new Memory("Memory", instructions, 200.0, 200.0);
  //memory.setLoggerOutputActive(true);
  //memory.setLoggerActive(true);
  super.addComponent(memory);

  ir = new Register<String>("IR", null);
  //ir.setLoggerActive(true);
  super.addComponent(ir);
```

```
  mdr = new RegisterMdr("MDR");
  super.addComponent(mdr);

  gnd = new GND("Gnd");
  super.addComponent(gnd);

  vcc = new VCC("Vcc");
  super.addComponent(vcc);


  insNode = new InsNode("Node");
  //insNode.setLoggerActive(true);
  super.addComponent(insNode);

  shift2j = new Shift2("Shift2j");
  super.addComponent(shift2j);

  regDst = new Mux2to1("RegDst"); // mux of the Registers Bank
  super.addComponent(regDst);

  memToReg = new Mux4to1("MemToReg");
  super.addComponent(memToReg);

  registers = new Registers("Registers");// Register Bank creation
  super.addComponent(registers);
  // AÃ±adimos la salida del programa artificialmente, que cargue en el
registro ra(31)
  // el nÃºmero de instrucciones. De esta forma, al llegar a jr, sÃ³lo
ejecutarÃ¡ la instrucciÃ³n siguiente
  // al jr y acabarÃ¡ el programa.
  Integer destinyAsInt = 4*instructions.size();
  registers.setRegisterValue(31, destinyAsInt);

  signExt = new SignExtender("SignExt");
  super.addComponent(signExt);

  regA = new Register<Integer>("A",0);
  super.addComponent(regA);

  regB = new Register<Integer>("B",0);
  super.addComponent(regB);

  shifter = new Shift2("Shifter2");
  super.addComponent(shifter);

  constant = new Constant("4", 4);
  super.addComponent(constant);

  aluSrcA = new Mux2to1("MuxALUA");
  super.addComponent(aluSrcA);

  aluSrcB = new Mux4to1("MuxALUB");
  super.addComponent(aluSrcB);

  aluCtrl = new ALUControl("ALUCtrl");
  super.addComponent(aluCtrl);

  pcSrc = new Mux4to1("PCSrc");
  super.addComponent(pcSrc);

  alu = new ALU("ALU"); // creation of the ALU
  super.addComponent(alu);

  aluOut = new Register<Integer>("ALUOut",0);
  super.addComponent(aluOut);

  ctrl = new ControladorMulticycle("Ctrl");
  super.addComponent(ctrl);
```

```
  // COUPLINGS
  // Salidas del PC
  super.addCoupling(pc, Register.outOutName, iorD, Mux2to1.inIn0Name);
  super.addCoupling(pc, Register.outOutName, aluSrcA, Mux2to1.inIn0Name);
  // Salidas de IorD
  super.addCoupling(iorD, Mux2to1.outOutName, memory, Memory.inAddrName);
  // Salidas de la memoria
  super.addCoupling(memory, Memory.outDrName, ir, Register.inInName);
  super.addCoupling(memory, Memory.outDrName, mdr, Register.inInName);
  super.addCoupling(memory, Memory.outStopName, clock, Clock.inName);
  // Salidas del IR
  super.addCoupling(ir, Register.outOutName, insNode, InsNode.inInName);
  // Salidas del MDR
  super.addCoupling(mdr, Register.outOutName, memToReg, Mux4to1.inIn1Name);
  // Salidas del GND
  super.addCoupling(gnd, GND.outName, memToReg, Mux4to1.inIn2Name);
  // Salidas del VCC
  super.addCoupling(vcc, VCC.outName, memToReg, Mux4to1.inIn3Name);
  // Salidas del InsNode
  super.addCoupling(insNode, InsNode.outOut2500Name, shift2j,
Shift2.inPortInName);
  super.addCoupling(insNode, InsNode.outOut2521Name, registers,
Registers.inRAName);
  super.addCoupling(insNode, InsNode.outOut2016Name, registers,
Registers.inRBName);
  super.addCoupling(insNode, InsNode.outOut2016Name, regDst,
Mux2to1.inIn0Name);
  super.addCoupling(insNode, InsNode.outOut1511Name, regDst,
Mux2to1.inIn1Name);
  super.addCoupling(insNode, InsNode.outOut1500Name, signExt,
SignExtender.inPortInName);
  super.addCoupling(insNode, InsNode.outOut3126Name, ctrl,
ControladorMulticycle.inOpName);
  super.addCoupling(insNode, InsNode.outOut0500Name, ctrl,
ControladorMulticycle.inFunctName);
  super.addCoupling(insNode, InsNode.outOut0500Name, aluCtrl,
ALUControl.FunctName);
  // Salidas del registro de desplazamiento para j
  super.addCoupling(shift2j, Shift2.outPortOutName, pcSrc, Mux4to1.inIn2Name);
  // Salidas de RegDst
  super.addCoupling(regDst, Mux2to1.outOutName, registers,
Registers.inRWName);
  // Salidas de MemToReg
  super.addCoupling(memToReg, Mux4to1.outOutName, registers,
Registers.inBusWName);
  // Salidas del banco de registros
  super.addCoupling(registers, Registers.outBusAName, regA,
Register.inInName);
  super.addCoupling(registers, Registers.outBusBName, regB,
Register.inInName);
  // Salidas del extensor de signo
  super.addCoupling(signExt, SignExtender.outPortOutName, aluSrcB,
Mux4to1.inIn2Name);
  super.addCoupling(signExt, SignExtender.outPortOutName, shifter,
Shift2.inPortInName);
  // Salidas del registro A
  super.addCoupling(regA, Register.outOutName, aluSrcA, Mux2to1.inIn1Name);
  super.addCoupling(regA, Register.outOutName, pcSrc, Mux4to1.inIn1Name);
  // Salidas del registro B
  super.addCoupling(regB, Register.outOutName, memory, Memory.inDwName);
  super.addCoupling(regB, Register.outOutName, aluSrcB, Mux4to1.inIn0Name);
  // Salidas de la constante 4:
  super.addCoupling(constant, Constant.outOutName, aluSrcB,
Mux4to1.inIn1Name);
  // Salidas del registro de desplazamiento 1
  super.addCoupling(shifter, Shift2.outPortOutName, aluSrcB,
Mux4to1.inIn3Name);
```

```
  // Salidas de ALUSrcA
  super.addCoupling(aluSrcA, Mux2to1.outOutName, alu, ALU.inOpAName);
  // Salidas de ALUSrcB
  super.addCoupling(aluSrcB, Mux4to1.outOutName, alu, ALU.inOpBName);
  // Salidas del control de la ALU
  super.addCoupling(aluCtrl, ALUControl.ALUCtrlName, alu, ALU.inCtrlName);
  // Salidas del PCSrc
  super.addCoupling(pcSrc, Mux4to1.outOutName, pc, Register.inInName);
  // Salidas de la ALU
  super.addCoupling(alu, ALU.outOutName, pcSrc, Mux4to1.inIn0Name);
  super.addCoupling(alu, ALU.outOutName, aluOut, Register.inInName);
  super.addCoupling(alu, ALU.outZeroName, ctrl,
ControladorMulticycle.inZeroName);
  super.addCoupling(alu, ALU.outLessThanName, ctrl,
ControladorMulticycle.inLessThanName);
  // Salidas de ALUOut
  super.addCoupling(aluOut, Register.outOutName, iorD, Mux2to1.inIn1Name);
  super.addCoupling(aluOut, Register.outOutName, memToReg, Mux4to1.inIn0Name);
  // Salidas del reloj
  super.addCoupling(clock, Clock.outName, pc, Register.inClkName);
  super.addCoupling(clock, Clock.outName, memory, Memory.inClkName);
  super.addCoupling(clock, Clock.outName, ir, Register.inClkName);
  super.addCoupling(clock, Clock.outName, mdr, Register.inClkName);
  super.addCoupling(clock, Clock.outName, registers, Registers.inCLKName);
  super.addCoupling(clock, Clock.outName, regA, Register.inClkName);
  super.addCoupling(clock, Clock.outName, regB, Register.inClkName);
  super.addCoupling(clock, Clock.outName, aluOut, Register.inClkName);
  super.addCoupling(clock, Clock.outName, ctrl,
ControladorMulticycle.inClkName);
  // Salidas del controlador:
  super.addCoupling(ctrl, ControladorMulticycle.outPCWriteName, pc,
Register.inRegWriteName);
  super.addCoupling(ctrl, ControladorMulticycle.outIorDName, iorD,
Mux2to1.inCtrlName);
  super.addCoupling(ctrl, ControladorMulticycle.outMemWriteName, memory,
Memory.inMemWriteName);
  super.addCoupling(ctrl, ControladorMulticycle.outMemReadName, memory,
Memory.inMemReadName);
  super.addCoupling(ctrl, ControladorMulticycle.outIRWriteName, ir,
Register.inRegWriteName);
  super.addCoupling(ctrl, ControladorMulticycle.outMDRWriteName, mdr,
Register.inRegWriteName);
  super.addCoupling(ctrl, ControladorMulticycle.outRegDstName, regDst,
Mux2to1.inCtrlName);
  super.addCoupling(ctrl, ControladorMulticycle.outMemtoRegName, memToReg,
Mux4to1.inCtrlName);
  super.addCoupling(ctrl, ControladorMulticycle.outRegWriteName, registers,
Registers.inRegWriteName);
  super.addCoupling(ctrl, ControladorMulticycle.outAWriteName, regA,
Register.inRegWriteName);
  super.addCoupling(ctrl, ControladorMulticycle.outBWriteName, regB,
Register.inRegWriteName);
  super.addCoupling(ctrl, ControladorMulticycle.outALUSrcAName, aluSrcA,
Mux2to1.inCtrlName);
  super.addCoupling(ctrl, ControladorMulticycle.outALUSrcBName, aluSrcB,
Mux2to1.inCtrlName);
  super.addCoupling(ctrl, ControladorMulticycle.outPcSrcName, pcSrc,
Mux4to1.inCtrlName);
  super.addCoupling(ctrl, ControladorMulticycle.outALUOpName, aluCtrl,
ALUControl.ALUopName);
  super.addCoupling(ctrl, ControladorMulticycle.outOutWriteName, aluOut,
Register.inRegWriteName);
```

5. Interfaz gráfica para la visualización de señales.

5.1. Motivación

Al analizar nuestro proyecto, hemos tomado conciencia de que era necesario un medio de visualización de resultados para el MIPS, tanto monociclo, multiciclo como segmentado, diferente al cual disponíamos. Por este motivo hemos construido una interfaz gráfica, a través de la cual, podíamos visualizar de una manera más representativa, los resultados de ejecución de un programa sobre el MIPS.

Inicialmente, al ejecutar un programa en MIPS, los resultados de ejecución se podían visualizar a través de un archivo log, cuyo formato está descrito más adelante. En este archivo, se visualizaban las señales de nuestro procesador, a intervalos de tiempo, conjuntamente con su valor. A través de este archivo, podíamos observar los resultados, pero de una manera menos representativa y rápida que con la interfaz, por ello se decidió crear esta interfaz, que representa los resultados obtenidos a través de un cronograma, por tanto, una forma más representativa y clara de contemplar los resultados obtenidos, y poder analizar más rápidamente, los valores de las señales y los intervalos en los que se producen.

Cabe destacar que esta contribución no es válida sólo para el procesador diseñado en este proyecto, sino para cualquier sistema implementado en xDEVS. Cualquier modelo atómico o acoplado en xDEVS imprime la secuencia de eventos en el fichero son sólo añadir una línea del tipo:

```
model.activeLog(true);
```

Con la línea anterior el modelo en cuestión imprime su secuencia de eventos en formato texto, lo que es difícil de interpretar cuando decenas de modelos registran su comportamiento. Es por ello que se hizo necesaria la creación del interfaz gráfico. La interfaz comienza a funcionar, cargando un archivo log, a través de la barra de menú Options, que contiene una pestaña, que nos permite realizar esta operación. Una vez cargado este fichero, la interfaz, analizará su contenido, y comenzará a dibujar los resultados, en función de lo que aparece en el fichero. El modo en que realiza esta acción será, tomado cada señal, y dibujando sus diferentes valores, en cada uno de los intervalos en los que los toma, y así sucesivamente, hasta que lea todo el archivo.

Finalmente, tendremos dibujado, un cronograma, y podremos observar perfectamente, cada una de las señales, y podremos observar de manera sencilla, los cambios en los valores de cada una que se producen, y los intervalos.

5.2. Formato

Como habíamos dicho previamente, la interfaz dispone de una barra de tareas. La única opción visible, es la pestaña Options, a través de la cual, se cargará el fichero de resultados log. Este fichero deberá tener un determinado formato, para que la interfaz funcione correctamente, y este formato, se describe en el siguiente punto.

Una vez cargado el archivo, la interfaz, dibujará todo lo que haya escrito en él, y el resultado final, será un cronograma típico, con valores señales, e intervalos de tiempo.

Para aclarar mejor este punto, observamos un ejemplo de nuestra interfaz funcionando, después de la carga de un fichero (ver Figura 45).

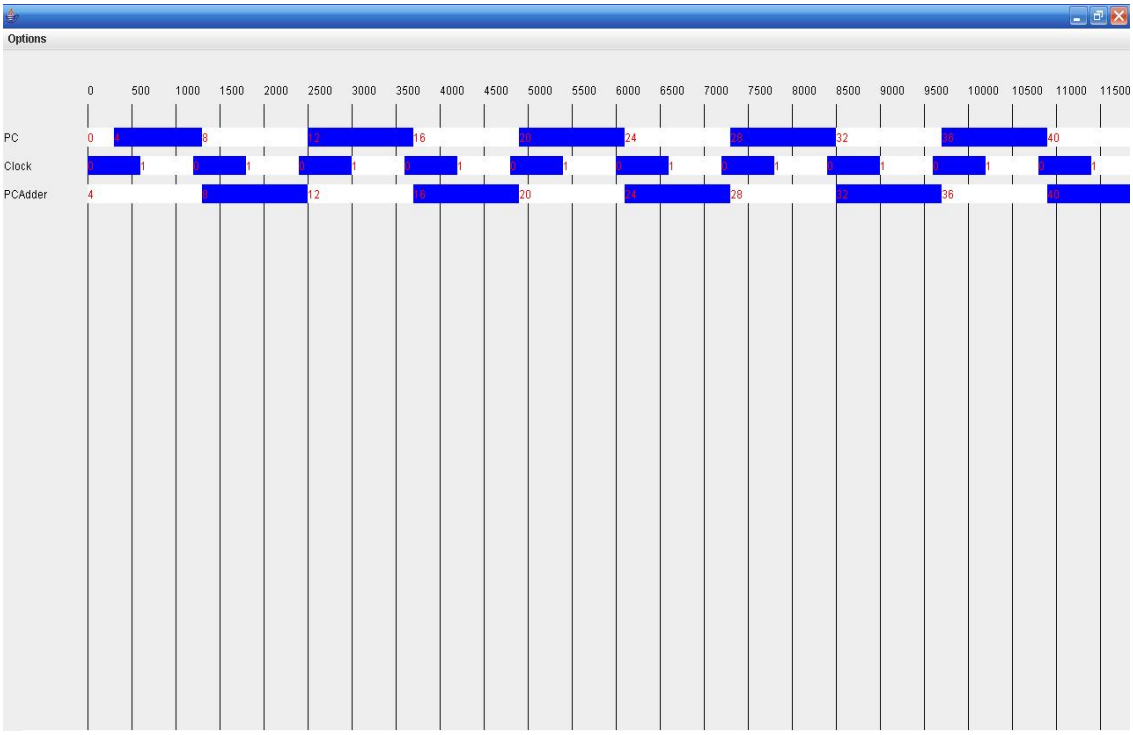


Figura 45. Ejemplo de ejecución.

Como podemos observar, el resultado, es típicamente un cronograma, que detallamos más en profundidad a continuación.

En este caso, el archivo log, sólo contenía los resultados de tres señales: PC, Clock y PCAdder.

Si observamos más en detalle la Figura 45, por ejemplo en la señal PC, su primer valor 0, está representado dentro de un rectángulo blanco, y con su valor en color rojo. Posteriormente, en el intervalo 300, se comprueba que se ha producido un cambio en el valor de la señal, y como resultado, la interfaz pinta este valor, 4, cambiando el color del rectángulo, para que clarifique más que la señal ha cambiado, y este valor se mantendrá hasta que la señal, experimenta un nuevo cambio, que es a partir del intervalo 1200.

Este proceso descrito es el mismo, para todas las señales, y como observamos en el ejemplo, tenemos una manera de observar resultados, bastante más clara y rápida de la que teníamos al principio.

5.3. Archivo log

En este apartado se describe, el formato que el archivo debe tener, para que nuestra interfaz sea capaz de representar los resultados de una ejecución. Es necesario que este archivo, tenga una extensión .log para poderlo cargar.

Para explicar su formato, nos vamos a servir del archivo que refleja el registro gráfico presentado en la Figura 45, y a partir de este, explicaremos en profundidad cada una de sus partes.

Archivo de registro

```
[INFO|00:00:00.016]: 0.0 -> PC:out::0
[INFO|00:00:00.047]: 0.0 -> Clock:out::0
[INFO|00:00:00.063]: 0.0 <- PC:clk::0
[INFO|00:00:00.063]: 0.0 <- PCAdder:opA::0
[INFO|00:00:00.063]: 100.0 -> PCAdder:out::4
[INFO|00:00:00.063]: 200.0 <- PC:RegWrite::1
[INFO|00:00:00.078]: 350.0 <- PC:in::4
[INFO|00:00:00.078]: 600.0 -> Clock:out::1
[INFO|00:00:00.078]: 600.0 <- PC:clk::1
[INFO|00:00:00.078]: 1200.0 -> Clock:out::0
[INFO|00:00:00.078]: 1200.0 <- PC:clk::0
[INFO|00:00:00.094]: 1200.0 -> PC:out::4
[INFO|00:00:00.094]: 1200.0 <- PCAdder:opA::4
[INFO|00:00:00.094]: 1300.0 -> PCAdder:out::8
[INFO|00:00:00.094]: 1300.0 <- PC:in::8
[INFO|00:00:00.094]: 1400.0 <- PC:RegWrite::1
[INFO|00:00:00.110]: 1400.0 <- PC:in::8
[INFO|00:00:00.110]: 1800.0 -> Clock:out::1
[INFO|00:00:00.110]: 1800.0 <- PC:clk::1
[INFO|00:00:00.110]: 2400.0 -> Clock:out::0
[INFO|00:00:00.110]: 2400.0 <- PC:clk::0
[INFO|00:00:00.110]: 2400.0 -> PC:out::8
[INFO|00:00:00.110]: 2400.0 <- PCAdder:opA::8
[INFO|00:00:00.110]: 2450.0 <- PC:in::8
[INFO|00:00:00.110]: 2500.0 -> PCAdder:out::12
[INFO|00:00:00.125]: 2500.0 <- PC:in::12
[INFO|00:00:00.125]: 2600.0 <- PC:RegWrite::1
[INFO|00:00:00.125]: 2600.0 <- PC:in::12
[INFO|00:00:00.141]: 3000.0 -> Clock:out::1
[INFO|00:00:00.141]: 3000.0 <- PC:clk::1
[INFO|00:00:00.141]: 3600.0 -> Clock:out::0
[INFO|00:00:00.141]: 3600.0 <- PC:clk::0
[INFO|00:00:00.156]: 3600.0 -> PC:out::12
[INFO|00:00:00.156]: 3600.0 <- PCAdder:opA::12
[INFO|00:00:00.156]: 3700.0 -> PCAdder:out::16
[INFO|00:00:00.156]: 3700.0 <- PC:in::16
[INFO|00:00:00.172]: 3800.0 <- PC:RegWrite::1
[INFO|00:00:00.172]: 3800.0 <- PC:in::16
[INFO|00:00:00.172]: 4200.0 -> Clock:out::1
[INFO|00:00:00.172]: 4200.0 <- PC:clk::1
[INFO|00:00:00.172]: 4800.0 -> Clock:out::0
[INFO|00:00:00.172]: 4800.0 <- PC:clk::0
[INFO|00:00:00.172]: 4800.0 -> PC:out::16
[INFO|00:00:00.172]: 4800.0 <- PCAdder:opA::16
[INFO|00:00:00.172]: 4900.0 -> PCAdder:out::20
[INFO|00:00:00.188]: 4900.0 <- PC:in::20
[INFO|00:00:00.188]: 5000.0 <- PC:RegWrite::1
[INFO|00:00:00.188]: 5000.0 <- PC:in::20
[INFO|00:00:00.188]: 5400.0 -> Clock:out::1
[INFO|00:00:00.188]: 5400.0 <- PC:clk::1
[INFO|00:00:00.188]: 6000.0 -> Clock:out::0
[INFO|00:00:00.188]: 6000.0 <- PC:clk::0
[INFO|00:00:00.188]: 6000.0 -> PC:out::20
[INFO|00:00:00.188]: 6000.0 <- PCAdder:opA::20
[INFO|00:00:00.188]: 6100.0 -> PCAdder:out::24
[INFO|00:00:00.188]: 6100.0 <- PC:in::24
[INFO|00:00:00.188]: 6200.0 <- PC:RegWrite::1
[INFO|00:00:00.188]: 6200.0 <- PC:in::24
[INFO|00:00:00.188]: 6250.0 <- PC:in::24
[INFO|00:00:00.188]: 6600.0 -> Clock:out::1
[INFO|00:00:00.188]: 6600.0 <- PC:clk::1
[INFO|00:00:00.203]: 7200.0 -> Clock:out::0
[INFO|00:00:00.203]: 7200.0 <- PC:clk::0
[INFO|00:00:00.203]: 7200.0 -> PC:out::24
[INFO|00:00:00.203]: 7200.0 <- PCAdder:opA::24
[INFO|00:00:00.203]: 7300.0 -> PCAdder:out::28
[INFO|00:00:00.203]: 7300.0 <- PC:in::28
```

```
[INFO|00:00:00.203]: 7400.0 <- PC:RegWrite::1
[INFO|00:00:00.203]: 7400.0 <- PC:in::28
[INFO|00:00:00.203]: 7450.0 <- PC:in::28
[INFO|00:00:00.203]: 7800.0 -> Clock:out::1
[INFO|00:00:00.203]: 7800.0 <- PC:clk::1
[INFO|00:00:00.203]: 8400.0 -> Clock:out::0
[INFO|00:00:00.203]: 8400.0 <- PC:clk::0
[INFO|00:00:00.203]: 8400.0 -> PC:out::28
[INFO|00:00:00.219]: 8400.0 <- PCAdder:opA::28
[INFO|00:00:00.219]: 8500.0 -> PCAdder:out::32
[INFO|00:00:00.219]: 8500.0 <- PC:in::32
[INFO|00:00:00.219]: 8600.0 <- PC:RegWrite::1
[INFO|00:00:00.219]: 8600.0 <- PC:in::32
[INFO|00:00:00.219]: 9000.0 -> Clock:out::1
[INFO|00:00:00.219]: 9000.0 <- PC:clk::1
[INFO|00:00:00.219]: 9600.0 -> Clock:out::0
[INFO|00:00:00.219]: 9600.0 <- PC:clk::0
[INFO|00:00:00.219]: 9600.0 -> PC:out::32
[INFO|00:00:00.219]: 9600.0 <- PCAdder:opA::32
[INFO|00:00:00.219]: 9700.0 -> PCAdder:out::36
[INFO|00:00:00.219]: 9700.0 <- PC:in::36
[INFO|00:00:00.219]: 9800.0 <- PC:RegWrite::1
[INFO|00:00:00.219]: 9800.0 <- PC:in::36
[INFO|00:00:00.219]: 10200.0 -> Clock:out::1
[INFO|00:00:00.219]: 10200.0 <- PC:clk::1
[INFO|00:00:00.219]: 10800.0 -> Clock:out::0
[INFO|00:00:00.219]: 10800.0 <- PC:clk::0
[INFO|00:00:00.219]: 10800.0 -> PC:out::36
[INFO|00:00:00.235]: 10800.0 <- PCAdder:opA::36
[INFO|00:00:00.235]: 10900.0 -> PCAdder:out::40
[INFO|00:00:00.235]: 10900.0 <- PC:in::40
[INFO|00:00:00.235]: 11000.0 <- PC:RegWrite::1
[INFO|00:00:00.235]: 11000.0 <- PC:in::40
[INFO|00:00:00.235]: 11400.0 -> Clock:out::1
[INFO|00:00:00.235]: 11400.0 <- PC:clk::1
[INFO|00:00:00.235]: 12000.0 -> Clock:out::0
[INFO|00:00:00.235]: 12000.0 <- PC:clk::0
[INFO|00:00:00.235]: 12000.0 -> PC:out::40
[INFO|00:00:00.235]: 12000.0 <- PCAdder:opA::40
[INFO|00:00:00.235]: 12100.0 -> PCAdder:out::44
[INFO|00:00:00.235]: 12100.0 <- PC:in::44
[INFO|00:00:00.235]: 12200.0 <- PC:RegWrite::1
[INFO|00:00:00.235]: 12200.0 <- PC:in::44
[INFO|00:00:00.235]: 12600.0 -> Clock:out::1
[INFO|00:00:00.235]: 12600.0 <- PC:clk::1
[INFO|00:00:00.235]: 13200.0 -> Clock:out::0
[INFO|00:00:00.235]: 13200.0 <- PC:clk::0
[INFO|00:00:00.235]: 13200.0 -> PC:out::44
[INFO|00:00:00.235]: 13200.0 <- PCAdder:opA::44
[INFO|00:00:00.235]: 13300.0 -> PCAdder:out::48
[INFO|00:00:00.235]: 13300.0 <- PC:in::48
[INFO|00:00:00.235]: 13400.0 <- PC:RegWrite::1
[INFO|00:00:00.235]: 13400.0 <- PC:in::48
[INFO|00:00:00.235]: 13450.0 <- PC:in::48
[INFO|00:00:00.250]: 13800.0 -> Clock:out::1
[INFO|00:00:00.250]: 13800.0 <- PC:clk::1
[INFO|00:00:00.250]: 14400.0 -> Clock:out::0
[INFO|00:00:00.250]: 14400.0 <- PC:clk::0
[INFO|00:00:00.250]: 14400.0 -> PC:out::48
[INFO|00:00:00.250]: 14400.0 <- PCAdder:opA::48
[INFO|00:00:00.250]: 14450.0 <- PC:in::48
[INFO|00:00:00.250]: 14500.0 -> PCAdder:out::52
[INFO|00:00:00.250]: 14500.0 <- PC:in::52
[INFO|00:00:00.250]: 14600.0 <- PC:RegWrite::1
[INFO|00:00:00.250]: 14600.0 <- PC:in::52
[INFO|00:00:00.250]: 15000.0 -> Clock:out::1
[INFO|00:00:00.250]: 15000.0 <- PC:clk::1
```

Como podemos observar cada línea del registro comienza con la parte “[INFO…..”. Como su propio nombre indica, esta es la información que nuestra interfaz tratará y representará. Al final de cada línea de información, vendrán escritas cada una de las señales a representar, con sus correspondientes intervalos de tiempo. Para explicar más correctamente la estructura que debe tener cada una de estas líneas, utilizamos una como ejemplo:

```
[INFO|00:00:00.219]: 10800.0 -> PC:out::36
```

La primera parte, “[INFO|00:00:00.219]:”, indica el tiempo transcurrido en la CPU real (la que está simulando el sistema) desde que comenzó la simulación, en este caso, 219 ms. La segunda parte “10800.0”, indica el tiempo virtual de simulación, en este caso 10800. Como se trata de tiempo virtual, estos pueden ser ps, ns o incluso ms. Dado que en al MIPS se le dota de un tiempo de ciclo de 1200 ns, se considera entonces 10800 ns. La tercera parte “->” indica que el evento registrado es de salida, y la cuarta parte, indica que por el puerto “out”, del elemento “PC” está saliendo un 36 (es decir, el contador de programa pasa a valer 36).

Por tanto, nuestra interfaz, tratará cada una de estas líneas, y representará, en la escala de tiempo virtual, el valor de la señal en un determinado intervalo de tiempo. Cuando la señal cambia su valor, nuestra interfaz, mostrará este cambio, cambiando el color de representación de la señal, justo en el intervalo en el que se produce este cambio.

6. Depurador DEVS

Esta parte del proyecto, trata un inconveniente que hemos observado en todos los simuladores DEVS. El problema observado, se debe, a que ningún simulador DEVS, disponía de un medio de visualización para depurar sistemas DEVS, a pesar de disponer de más de 20 simuladores en el planeta.

Con este medio, se permite depurar sistemas DEVS, de un modo más intuitivo, debido a que no tenemos que recorrer todo el código escrito, para depurar el sistema, sino que ahora podemos, depurarlo por módulos, hecho que reduce significativamente el tiempo empleado, a la hora de depurar.

Este depurador, se encarga de observar la implementación de un sistema DEVS, y representar sus módulos, junto con sus puertos de entrada y de salida, y las conexiones de dichos módulos. Para realizar esta representación, se usa la librería JGraphX, con sus componentes [5].

Para visualizar un modelo DEVS, basta con crear el modelo original y ubicar cada componente en unas coordenadas deseadas. La siguiente lista muestra un ejemplo de cómo se configura el procesador MIPS monociclo en este depurador:

Configuración del depurador para el procesador MIPS monociclo

```
public class MipsMonocycleView extends MipsMonocycle {

 public MipsMonocycleView(String name, String filePath) {
  super(name, filePath);
 }

 public static void main(String[] args) {
 MipsMonocycleView mips = null;
 CoupledView mipsView = null;
 try {
     mips   =   new   MipsMonocycleView("MIPS",   "test"   +   File.separator   +
"bench1_fibonacci.dis");
     mipsView = new CoupledView(mips);
     mipsView.setBounds("MIPS", 0, 0, 1800, 900);
     mipsView.setBounds(mips.clock.getName(), 0, 0, 20, 20);
     mipsView.setBounds(mips.ctrl.getName(), 0.05, 0.00, 110, 250);
     mipsView.setBounds(mips.aluCtrl.getName(), 0.35, 0.00, 80, 80);
     mipsView.setBounds(mips.and2.getName(), 0.45, 0.00, 80, 80);
     mipsView.setBounds(mips.muxjr.getName(), 0.02, 0.35, 120, 80);
     mipsView.setBounds(mips.pc.getName(), 0.02, 0.50, 100, 80);
     mipsView.setBounds(mips.muxPCSrc.getName(), 0.02, 0.65, 100, 80);
     mipsView.setBounds(mips.pcAdder.getName(), 0.15, 0.35, 80, 80);
     mipsView.setBounds(mips.shif2j.getName(), 0.17, 0.10, 80, 80);
     //mipsView.setBounds(mips.nodej.getName(), 0.25, 0.10, 90, 80);
     mipsView.setBounds(mips.branchAdder.getName(), 0.25, 0.20, 90, 80);
     mipsView.setBounds(mips.shif2.getName(), 0.25, 0.30, 90, 80);
     mipsView.setBounds(mips.signExt.getName(), 0.25, 0.40, 90, 80);
     mipsView.setBounds(mips.insMem.getName(), 0.12, 0.5, 320, 100);
     mipsView.setBounds(mips.insNode.getName(), 0.12, 0.70, 300, 120);
     mipsView.setBounds(mips.muxRegDst.getName(), 0.35, 0.60, 80, 80);
     mipsView.setBounds(mips.muxRegData.getName(), 0.35, 0.70, 80, 80);
     mipsView.setBounds(mips.gnd.getName(), 0.30, 0.85, 40, 40);
     mipsView.setBounds(mips.vcc.getName(), 0.30, 0.90, 40, 40);
     mipsView.setBounds(mips.muxSlt.getName(), 0.35, 0.80, 80, 80);
     mipsView.setBounds(mips.registers.getName(), 0.45, 0.45, 180, 300);
     mipsView.setBounds(mips.muxALUSrc.getName(), 0.60, 0.40, 80, 80);
     mipsView.setBounds(mips.alu.getName(), 0.65, 0.50, 180, 90);
     mipsView.setBounds(mips.dataMemory.getName(), 0.65, 0.65, 180, 150);
     mipsView.setBounds(mips.muxMemToReg.getName(), 0.80, 0.75, 100, 100);
```

```
 } catch (Exception ee) {
     ee.printStackTrace();
 }
 if (mipsView == null) {
     return;
 }
 CoordinatorView coordinator = new CoordinatorView(mipsView);
 coordinator.setVisible(true);
 //Coordinator coordinator = new Coordinator(mips);
 //coordinator.simulate(Long.MAX_VALUE);
}
}
```

Las coordenadas de los distintos componentes (x, y) se representan normalizadas. El ancho y el alto se representan en puntos.

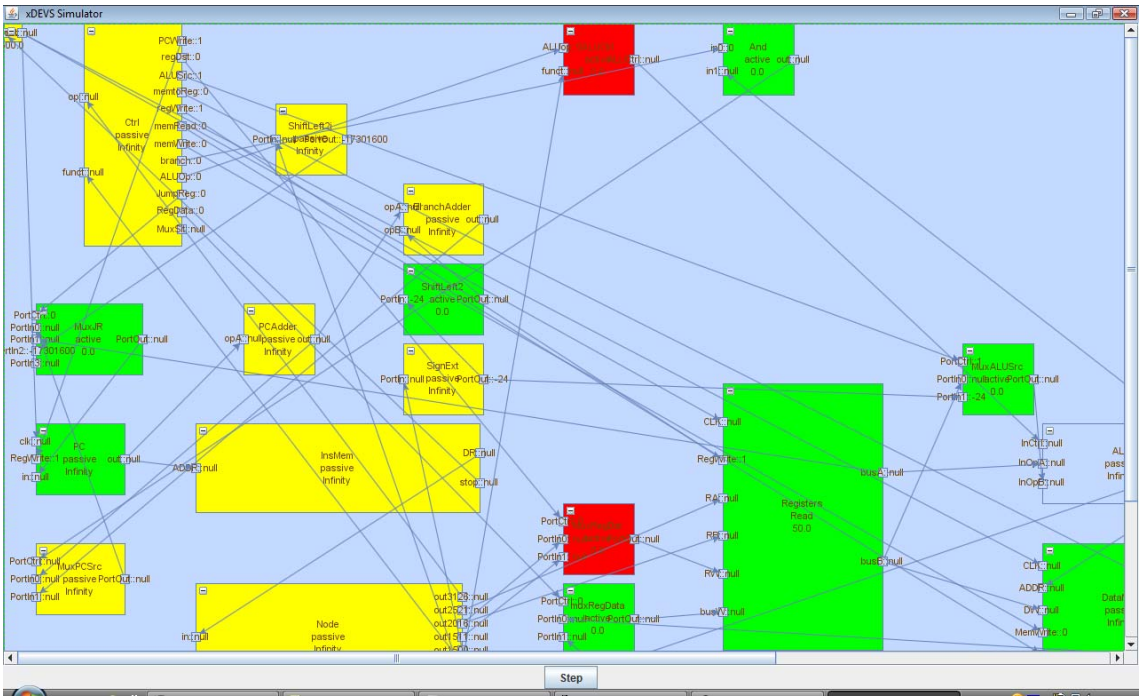


Figura 46. Depurador DEVS en ruta MIPS monociclo.

La Figura 46 muestra un ejemplo del depurador, que muestra la representación del MIPS monociclo. Cada componente representa en texto su nombre, su fase y su sigma. Cada línea representa un acoplamiento en el sistema. Un componente DEVS está en rojo si ha ejecutado ya su función de salida, en amarillo si ha ejecutado su transición interna, y en verde si ha ejecutado su transición externa. Cada pulsación en el botón “Step” (ver parte inferior de la Figura) avanza un ciclo en la simulación DEVS.

No obstante cabe destacar que la interpretación de la depuración debe de realizarla un experto en DEVS, ya que aunque gráficamente se pueda intuir que se corresponde con el procesador MIPS (viendo el banco de registros, el controlador, el contador de programa, …), la propagación de señales no es convencional como en un simulador de circuitos.

7. Resultados

En los capítulos anteriores hemos desarrollado una infraestructura para modelar sistemas en xDEVS y simularlos, permitiendo una depuración visual. Además, hemos desarrollado todos los componentes necesarios para la implementación de un procesador MIPS en tres versiones diferentes: monociclo, multiciclo y segmentado. Todo el software está disponible en la página oficial de xDEVS [17].

7.1. Visualización

Como se ha mencionado en capítulos anteriores, permitimos la visualización de la ejecución del procesador en distintos niveles de abstracción: depuración y cronograma. Aquí detallamos el uso del cronograma, ya que el depurador no proporciona información útil acerca de los datos y tiempos de procesamiento.

Un usuario puede observar las entradas y salidas del modelo, así como sus estados en un cronograma.

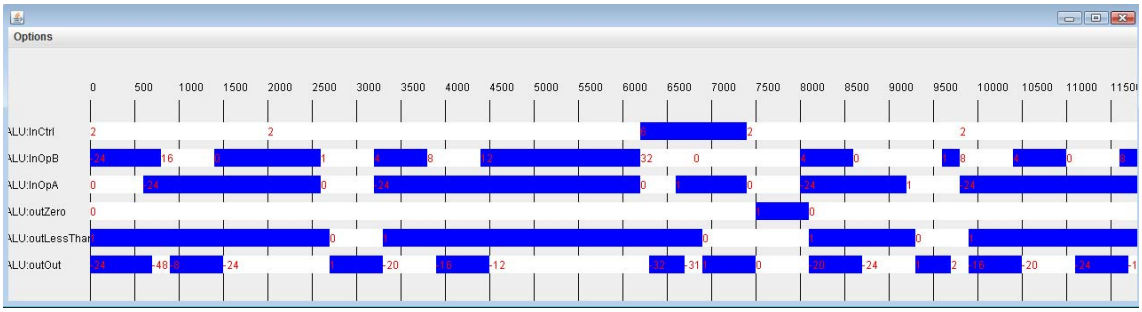


Figura 47. Entradas y salidas de la unidad aritmético lógica.

En la Figura 47 podemos observar el cronograma de las entradas y salidas de la unidad aritmético-lógica del procesador MIPS monociclo ejecutando el algoritmo de Fibonacci. La escala de tiempos es en picosegundos. Las entradas son: la señal de control (InCtrl), el operando A (inOpA) y el operando B (inOpB). Mientras que las salidas son: el resultado de la operación (outOut), una señal que se pone a 1 si el resultado es 0 (outZero) y una señal que se pone a uno si A es menor que B (outLessThan). La ALU presentada en la figura tiene un retardo de 100 ps. Podemos apreciar este retardo, por ejemplo, en el cálculo de -24 - 24 = -48. El último operando llega a los 650 ps. El resultado sale a los 750 ps.

7.2. Experimentos

En el siguiente conjunto de experimentos vamos a comparar el rendimiento de los procesadores monociclo, multiciclo y segmentado. Para ello vamos a realizar test sobre 3 algoritmos diferentes, que detallamos en el Apéndice II: cálculo del término 32 de la serie de Fibonacci (*FIB*), cálculo del producto escalar de dos vectores (*PROD*) y algoritmo de Euclides del máximo común divisor de los números 2048 y 4864 (*MCD*).

Para los tres procesadores establecemos los siguientes periodos de reloj:

- Monociclo: 600 ps
- Multiciclo: 200 ps
- Segmentado: 200 ps

Respecto al retardo de los distintos componentes, establecemos los siguientes valores para las tres arquitecturas: retardo de la memoria de 200 ps, retardo del banco de registros de 50 ps y retardo de la ALU de 100 ps. Estos son valores característicos que podemos encontrar en [4].

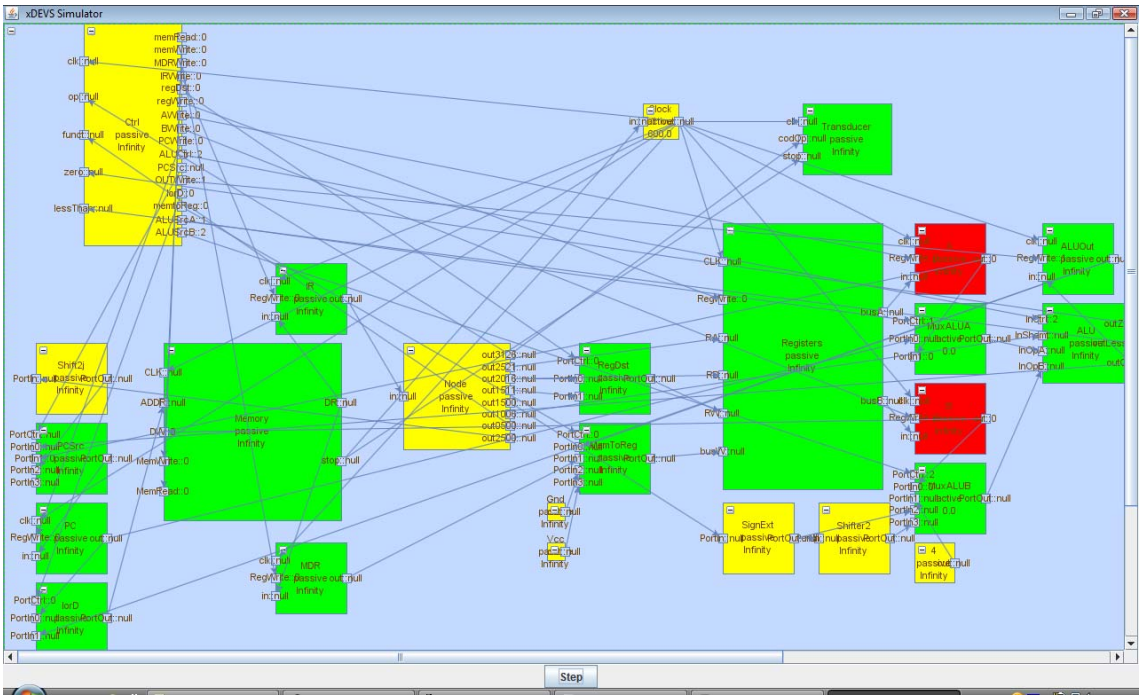


Figura 48. Depuración del algoritmo de Fibonacci.

La Figura 48 muestra el proceso de depuración del algoritmo de Fibonacci, según se está ejecutando una instrucción *addiu*. Este proceso sirve para detectar posibles errores en la generación de valores en los componentes, así como el correcto acoplamiento entre todos los elementos.

En el siguiente paso procedemos a realizar un estudio cuantitativo de resultados. En primer lugar, ejecutamos los tres algoritmos en la ruta de datos monociclo para extraer un informe completo de número de instrucciones por tipo. Estos datos re presentan en la siguiente tabla:

CodOp	Instr.	FIB	PROD	MCD
0	Tipo-R	67	428	36
2	j	32	65	9
4	beq	33	77	19
9	addiu	35	77	4
10	slti	33	22	0
35	lw	194	448	63
43	sw	134	164	18
0	jr	1	1	1
		529	**1282**	**150**

De la tabla anterior se desprende que en los tres casos la instrucción predominante es lw. Por el contrario, la instrucción que se ejecuta menos veces es slti. Lo único que el componente Transductor incorporado al sistema no puede determinar es el número de instrucciones beq que saltan y cuáles de ellas no.

En un estudio más detallado, podemos comparar los tiempos de ejecución en los tres procesadores, para hacer una comparativa. Tal comparativa se refleja en la siguiente tabla:

	Monociclo			Multiciclo			Segmentado		
	FIB	**PROD**	**MCD**	**FIB**	**PROD**	**MCD**	**FIB**	**PROD**	**MCD**
Tiempo total (ns)	317.4	769.2	90	449	1089	127.6	225.4	551.3	64.5
Ciclo (ps)	600	600	600	200	200	200	200	200	200
Ciclos	529	1282	150	2245	5445	638	1127	2756	322
Instrucciones	529	1282	150	529	1282	150	529	1282	150
CPI	1	1	1	4.25	4.25	4.25	2.13	2.15	2.15

Como se puede observar en la tabla anterior, la ruta de datos multiciclo es la que peores resultados ofrece en términos de tiempo de ejecución. Ello se debe a la alta presencia de instrucciones lw, que al constar de 5 ciclos de ejecución, elevan el tiempo total de ejecución de la instrucción lw a 1000 ps, en contra de los 600 ps del ciclo de reloj en el procesador monociclo.

El procesador segmentado es el que presenta mejor rendimiento. Sin embargo, el CPI del mismo está lejos de ser el CPI ideal de un procesador segmentado: CPI = 1. Ello se debe a que la decisión de salto está ubicada en la etapa MEM del procesador (lo que son 3 ciclos adicionales de espera por cada salto) y que cada instrucción lw introduce una parada en el cauce.

Como trabajo futuro habría que adelantar la decisión de salto a la etapa ID y gestionar saltos retardados, de esta forma el comportamiento del nuestro MIPS segmentado se aproximaría de forma sustancial al ideal.

Bajo estos supuestos, la única manera de hacer al procesador multiciclo más rentable sería la optimización del módulo de memoria. Sin embargo, aún reduciendo el tiempo de la memoria a 100 ps, vemos que el procesador monociclo sigue siendo más rentable:

	Monociclo			Multiciclo		
	FIB	**PROD**	**MCD**	**FIB**	**PROD**	**MCD**
Tiempo total (ns)	212.1	514.1	60.2	226.7	549.9	64.4
Ciclo (ps)	401	401	401	101	101	101
Ciclos	529	1282	150	2245	5445	638
Instrucciones	529	1282	150	529	1282	150
CPI	1	1	1	4.25	4.25	4.25

Finalmente, cabe destacar como principal contribución de este proyecto que ninguno de los simuladores inspeccionados en el estado del arte contempla la posibilidad de variar parámetros como los tiempos de retardo de los distintos componentes. Es más, sólo algunos de ellos como WinMIPS64 y EduMIPS64 proveen datos sencillos como el número total de ciclos o el CPI. Por último, recordar que la versatilidad del proyecto desarrollado permitiría la incorporación de estudios relativos al consumo de potencia, entre otros, lo que se propone como trabajo futuro.

Conclusiones

El simulador que se ha desarrollado ha conseguido cumplir las expectativas con las que iniciamos este proyecto.

Desde el comienzo se planteo el simulador como una herramienta de ayuda al estudiante y al profesor en su labor docente. Con la ayuda del simulador, puede entenderse desde un punto de vista más ameno, la compleja teoría que rodea a los procesadores que se estudian en asignaturas de arquitectura de computadores. La ejecución del simulador permite estudiar a fondo su rendimiento, en las distintas implementaciones, además de conocer datos estadísticos, cómo número de instrucciones de cada tipo o CPI del programa. Es además una herramienta bastante flexible pues permite configurar distintos parámetros como el retardo de los módulos, según la conveniencia del usuario.

Al ejecutar los distintos benchmarks sobre el simulador, hemos obtenido algunos resultados sorprendentes. Al configurar los parámetros de la simulación tales como retardos con los valores característicos de [4] observamos que para los tres benchmarks los resultados en monociclo son mejores que en el multiciclo, hablando en términos temporales, aunque comparando ambas con el segmentado, éste mejora notablemente el rendimiento. Estos resultados son debidos al gran número de instrucciones de tipo load presentes en el código ensamblador de los tres benchmarks presentados, que van de un 30% en el benchmark de Fibonacci hasta un 50 % en el benchmark del producto escalar. Posteriormente se configuraron los retardos haciéndolos la mitad, pero con estos parámetros el monociclo seguía siendo ligeramente mejor que el multiciclo. Podemos observar también como el CPI del segmentado se sitúa en torno al 2, lejos del CPI = 1 ideal, esto se debe a la ubicación de la decisión del salto y la cantidad de paradas del pipeline debido a la multitud de instrucciones load.

Este simulador está disponible para su uso de manera libre en sourceforge [17].

Glosario

DEVS: Es un acrónimo del inglés para referirse a Discrete Event System Specification (Especificación de Sistemas de Eventos Discretos)

Es un término estándar en el campo de la Simulación para modular y analizar sistemas de diversos tipos, en particular, sistemas de eventos discretos y sistemas hibridos continuos y discretos.

MICROPROCESADOR: Se trata de un circuito integrado que contiene en su interior una Unidad Central de Proceso (CPU) y un conjunto de elementos logicos conectados con otros dispositivos como memorias y puertos de entrada/salida formando un sistema para cumplir con una aplicación especifica.

Para que el microprocesador pueda ejecutar su cometido debe ejecutar paso a paso un programa que consiste en una secuencia de instrucciones, almacenándolas en elementos de memoria que forman parte del mismo.

MIPS: Son las siglas de Microproccesor without Interlocked Pipeline Stages y es una familia de microprocesadores de arquitectura RISC (Reduced Instructions Set Computer) desarrollados por MIPS Technologies y usados en muchos sistemas integrados como por ejemplo en las videoconsolas Nintendo 64, Sony Playstation, Sony Playstation 2, etc.

CRONOGRAMA: Se trata de una herramienta, que sirve para la representación de señales, junto con sus valores, y que sirve para visualizar de un modo gráfico y más representativo, el resultado que produce un programa, al ser ejecutado en un procesador a lo largo de toda su ejecución.

SEGMENTACIÓN: Es un método, a través del cual, se consigue mejorar el rendimiento de microprocesadores descomponiendo cada instrucción ejecutada por los mismos, en varias etapas, para permitir procesar una instrucción diferente en cada una de las etapas, con lo que se consigue trabajar con varias instrucciones simultáneamente.

SIMULACIÓN: Es un término definido por R.E. Shannon [7] como el proceso de diseñar un modelo de un sistema real y llevar a término experiencias con él, con la finalidad de comprender el comportamiento del sistema o evaluar nuevas estrategias -dentro de los límites impuestos por un cierto criterio o un conjunto de ellos - para el funcionamiento del sistema.

SIMULADOR: Es una herramienta que permite la simulación de un sistema reproduciendo su comportamiento.

Referencias

[1] B. P. Zeigler, T. G. Kim, and H. Praehofer, Theory of Modeling and Simulation, New York: Academic Press, 2000.

[2] José L. Risco-Martín, Saurabh Mittal, J. M. Cruz & Bernard P. Zeigler: eUDEVS: Executable UML Using DEVS Theory of Modeling and Simulation. SIMULATION: Transactions of SCS 85(11-12): 750-777 (2009)

[3] José L. Risco-Martín, Saurabh Mittal, M. A. López-Peña & J. M. Cruz: A W3C XML schema for DEVS scenarios. In SpringSim '07: Proceedings of the 2007 spring simulation multiconference: 279-286 (San Diego, CA, USA, 2007)

[4] Patterson, D.A. and J.L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, 3ª edición, Morgan Kaufmann, San Fransisco, CA, 2007.

[5] JGraphX: http://www.jgraph.com/jgraphx.html

[6] CygWin: http://www.cygwin.com

[7] Robert Shannon and James D. Johannes: Systems simulation: the art and science. IEEE Transactions on Systems, Man and Cybernetics 6(10). pp. 723-724. 1976.

[8] NSF Blue Ribbon Panel. 2006. Simulation-Based Engineering Science—Revolutionizing Engineering Science Through Simulation, retrieved August 15, 2006, from http://www.ices.utexas.edu/events/SBES_Final_Report.pdf

[9] Scott, M. 2006. WinMips64, http://www.computing.dcu.ie/mike/winmips64.html.

[10] Wainer, G. 2002. CD++: a toolkit to define discrete-event models. Software, Practice and Experience, 32(3): 1261–1306.

[11] Scott, M. 2006. WinMips64, http://www.computing.dcu.ie/mike/winmips64.html.

[12] The EduMIPS64 Team. 2007. EduMIPS64, http://www.edumips.org/.

[13] Dolman, D. 2007. Computer architecture: Visualization of a simple MIPS pipeline, http://www.icsa.inf.ed.ac.uk/research/groups/hase/models/mips/.

[14] Bem, E.Z. and L. Petelczyc. 2003. MiniMIPS: a simulation project for the computer architecture laboratory. SIGCSE ’03, ACM Press, New York, pp. 64–68.

[15] Branovic, I., R. Giorgi and E. Martinelli. 2004. WebMIPS: a new web-based MIPS simulation environment for computer architecture education. Workshop on Computer Architecture Education, 31st International Symposium on Computer Architecture, Munich, Germany, June 2004, pp. 93–98. http://www.dii.unisi.it/giorgi/WEBMIPS/.

[16] Garton, J. 2005. ProcessorSim—a visual MIPS R2000 processor simulator, http://jamesgart.com/procsim/.

[17] xDEVS: http://sourceforge.net/projects/xdevs/

Apéndice I. Compilador cruzado

Introducción

Para el desarrollo del proyecto nos encontramos con la necesidad de transformar el código fuente a lenguaje MIPS, para poder ejecutarlo en el simulador. Ante esta necesidad tuvimos que elegir un compilador que satisfaciera nuestras demandas.

Un compilador (ver Figura 49) es un programa (o un conjunto de ellos) que traduce un código fuente escrito en un determinado lenguaje de programación (lenguaje fuente), generando un código equivalente en otro lenguaje (lenguaje objetivo).

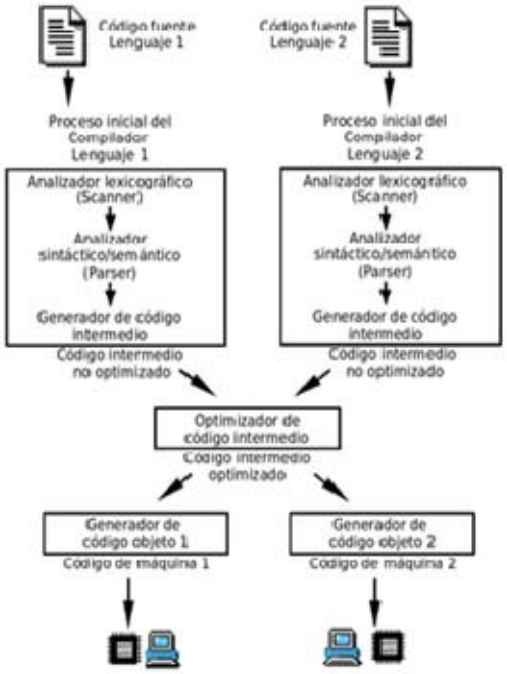


Figura 49. Compilador.

Dentro de los compiladores se enmarcan los compiladores cruzados. Estos son compiladores capaces de generar código ejecutable para una plataforma distinta de la que el compilador está corriendo.

El compilador cruzado que hemos escogido para el proyecto es el GCC, usado dentro de Cygwin, que nos permite generar binarios de código fuente escrito en C, en leguaje MIPS.

Cygwin

Cygwin es una colección de herramientas desarrollada por Cygnus Solutions para proporcionar un comportamiento similar a los sistemas Unix en Windows. Su objetivo es portar software que ejecuta en sistemas POSIX a Windows con una recompilación a partir de sus fuentes. Aunque los programas portados funcionan en todas las versiones de Windows, su comportamiento es mejor en Windows NT, Windows XP y Windows Server 2003.

En la actualidad, el paquete está mantenido principalmente por trabajadores de Red Hat. Se distribuye habitualmente bajo los términos de la GPL con la excepción de que permite ser enlazada con cualquier tipo de software libre cuya licencia esté de acuerdo con la definición de software libre. También es posible adquirir una cara licencia para distribuirla bajo otros tipos de licencia.

La instalación de Cygwin nos ha resultado algo compleja, por eso incluimos una pequeña guía - asistente de instalación:

Instalación de Cygwin

La instalación de Cygwin puede ocupar hasta 800 MB, pero escogiendo bien la descarga el tamaño es mucho menor. Desde la página principal de Cygwin (http://www.cygwin.com/) se descarga un stub installer (pequeño archivo instalador que descarga los módulos (paquetes en la terminología del instalador) deseados durante su ejecución) de unos 275 KBytes. Es aconsejable renombrar el nombre predeterminado (setup.exe) a otro más significativo (p.e.: cygwin_setup.exe).

Debe tenerse presente que, al usarse el mecanismo de instalación descrito en el párrafo anterior, deberás autorizar al instalador a efectuar conexiones a Internet, en el caso de que tengas un cortafuegos.

Al lanzar el instalador anterior, comenzará preguntando varios datos, de los que se proponen los siguientes valores recomendados:

- Origen de la instalación: Descargar desde Internet (Download from Internet). Si planeas hacer varias instalaciones, las descargas quedan guardadas para que no necesites descargar varias veces lo mismo (en instalaciones sucesivas, indicarías Install from Local Directory).
- Directorio raíz de instalación: C:\cygwin.
- Instalar para: Todos los usuarios (All Users).
- Tipo de archivo de texto predeterminado: Unix.
- Directorio local de paquetes: C:\cygwin\packages.
- Tipo de conexión a Internet: depende de cada caso, pero en entornos domésticos normalmente será Direct Connection.

En este momento el instalador conectará a Internet para recuperar la lista de *mirrors* (servidores desde los que es posible descargar los módulos de instalación).

- Seleccionar mirror: Cualquiera que funcione, pero es buena idea usar ftp://ftp.rediris.es.

Una vez seleccionado el mirror, el instalador recuperará la lista de paquetes disponibles y nos dejará seleccionarlos en una pantalla con un mecanismo muy similar al de los gestores de paquetes para Linux, organizando los paquetes en grupos.

- Columna Category: contiene los grupos de paquetes, que se despliegan y contraen al hacer clic sobre ellos.
- Columna Bin?: indica si el paquete está seleccionado para ser descargado.

- Columna Package: contiene los paquetes, que contendrán un solo archivo, varios archivos correspondientes a una única utilidad, o varias utilidades, dependiendo del paquete concreto.

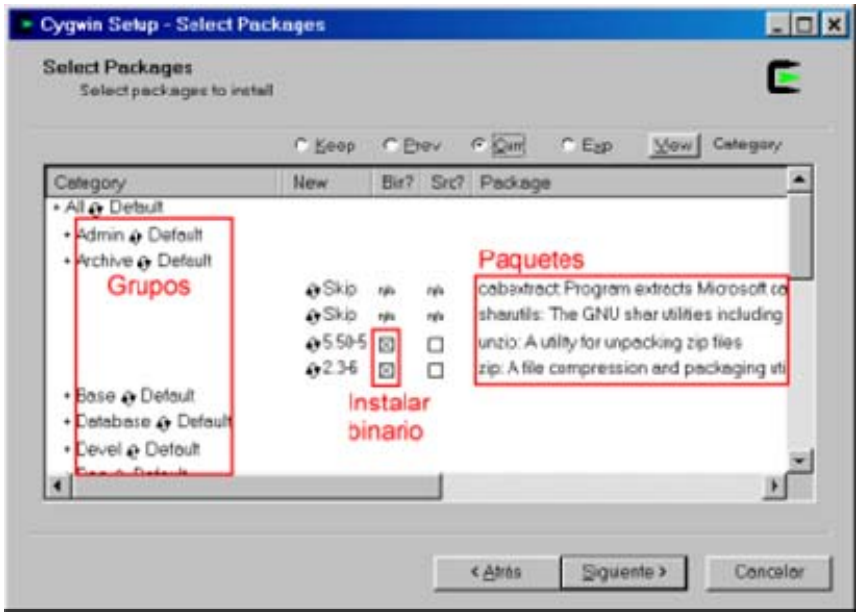


Figura 50. Selección de paquetes.

Pulsando en el botón "View" se pueden listar todos los paquetes en orden alfabético.

Una vez seleccionados los paquetes que se desean, podemos continuar con la instalación. Se recomienda seleccionar los paquetes: gcc-g++, make y libiconv (además de libiconv2, que instala en Cygwin por defecto).

Una vez seleccionados los paquetes, éstos serán descargados y, a continuación, instalados. Por último, aparecerá una ventana permitiendo crear accesos directos en el escritorio y finalizará la instalación.

Tras instalar Cygwin, nuestro directorio de usuario (/home/usuario) será (c:\cygwin\home\usuario).

Abrir Cygwin. Crear un directorio de descarga:

```
mkdir crosscomp

cd crosscomp
```

Ir a la página de GNU (http://www.gnu.org), buscar las fuentes de binutils y gcc y descargarlas en /home/usuario/crosscomp. Este manual fue probado con gcc-3.4.1.tar.bz2 y binutils-2.19.tar.bz2 (tener en cuenta que el gcc y binutils nuevo deben ser alguna versión inferior al instalado por cygwin, para ver la versión, ejecutar gcc -v en cygwin).

Descomprimir las fuentes:

```
tar xvf gcc-3.4.1.tar.bz2
```

```
tar xvf binutils-2.19.tar.bz2
```

Crear el directorio de compilación

```
mkdir mips

cd mips
```

Configurar el entorno de compilación para binutils:

```
../binutils-2.19/configure --prefix=/usr/local/gcc --enable-languages=c –program-prefix=mips- --disable-libssp --target=mips-linux-elf
```

Compilar binutils

```
make

make install
```

Limpiar el directorio de compilación:

```
rm -r *
```

(A lo mejor pregunta si borrar algunos archivos de sólo lectura, respondemos: y).

Actualizar el path:

```
export PATH=/usr/local/gcc/bin:$PATH
```

Configurar el entorno de compilación para gcc:

```
../gcc-3.4.1/configure --prefix=/usr/local/gcc --enable-languages=c --program-prefix=mips- --disable-libssp --target=mips-linux-elf
```

Compilar e instalar gcc

```
make

make install
```

Los ejecutables están en /usr/local/gcc/bin/mips-*

Uso de Cygwin

Una vez instalado, ya podemos generar código ensamblador y binario desde un archivo c.

```
export PATH=/usr/local/gcc/bin:$PATH

mips-gcc -march=mips32 -S archivo.c -o archivo.s

mips-as -march=mips32 archivo.s -o archivo.exe

mips-objdump -d archivo.exe > archivo.bin
```

Mostramos unos de los benchmarks, usados en el proyecto, el benchmark de Fibonnacci:

Código original en C	Código ensamblador para MIPS

`int main() {`
`  int f0, f1, f2, i;`
`  f0 = 0;`
`  f1 = 1;`
`  f2 = 0;`
`  i = 0;`
`  for(i=0; i<32; i++) {`
`  f2 = f1 + f0;`
`  f0 = f1;`
`  f1 = f2;`
`  }`
`  return 0;`
`}`
	`Disassembly of section .text:`
	
	`00000000 <main>:`
	`  0: 27bdffe8 addiu sp,sp,-24`
	`  4: afbe0010 sw s8,16(sp)`
	`  8: 03a0f021 move s8,sp`
	`  c: afc00000 sw zero,0(s8)`
	`  10: 24020001 li v0,1`
	`  14: afc20004 sw v0,4(s8)`
	`  18: afc00008 sw zero,8(s8)`
	`  1c: afc0000c sw zero,12(s8)`
	`  20: afc0000c sw zero,12(s8)`
	`  24: 8fc2000c lw v0,12(s8)`
	`  28: 28420020 slti v0,v0,32`
	`  2c: 1040000e beqz v0,68 <main+0x68>`
	`  30: 00000000 nop`
	`  34: 8fc30004 lw v1,4(s8)`
	`  38: 8fc20000 lw v0,0(s8)`
	`  3c: 00621021 addu v0,v1,v0`
	`  40: afc20008 sw v0,8(s8)`
	`  44: 8fc20004 lw v0,4(s8)`
	`  48: afc20000 sw v0,0(s8)`
	`  4c: 8fc20008 lw v0,8(s8)`
	`  50: afc20004 sw v0,4(s8)`
	`  54: 8fc2000c lw v0,12(s8)`
	`  58: 24420001 addiu v0,v0,1`
	`  5c: afc2000c sw v0,12(s8)`
	`  60: 08000009 j 24 <main+0x24>`
	`  64: 00000000 nop`
	`  68: 00001021 move v0,zero`
	`  6c: 03c0e821 move sp,s8`
	`  70: 8fbe0010 lw s8,16(sp)`
	`  74: 27bd0018 addiu sp,sp,24`
	`  78: 03e00008 jr ra`
	`  7c: 00000000 nop`

A la izquierda el códido original en C. A la derecha el binario generado por el compilador cruzado. Este archivo contiene la dirección de la instrucción (en hexadecimal), la codificación de la instrucción (también en hexadecimal), y el código ensamblador correspondiente.

Apéndice II – Benchmarks

A continuación proporcionamos el listado de los tres benchmarks con los que hemos trabajado.

Algoritmo de Fibonacci

El primer algoritmo calcula el término 31 de la serie de Fibonacci. En la tabla siguiente mostramos el código en C y su equivalente binario generado para el MIPS32:

Algoritmo de Fibonacci	Código ensamblador para MIPS

`int main() {`
`  int f0, f1, f2, i;`
`  f0 = 0;`
`  f1 = 1;`
`  f2 = 0;`
`  i = 0;`
`  for(i=0; i<32; i++) {`
`  f2 = f1 + f0;`
`  f0 = f1;`
`  f1 = f2;`
`  }`
`  return 0;`
`}`
	`Disassembly of section .text:`
	
	`00000000 <main>:`
	`  0: 27bdffe8 addiu sp,sp,-24`
	`  4: afbe0010 sw s8,16(sp)`
	`  8: 03a0f021 move s8,sp`
	`  c: afc00000 sw zero,0(s8)`
	`  10: 24020001 li v0,1`
	`  14: afc20004 sw v0,4(s8)`
	`  18: afc00008 sw zero,8(s8)`
	`  1c: afc0000c sw zero,12(s8)`
	`  20: afc0000c sw zero,12(s8)`
	`  24: 8fc2000c lw v0,12(s8)`
	`  28: 28420020 slti v0,v0,32`
	`  2c: 1040000e beqz v0,68 <main+0x68>`
	`  30: 00000000 nop`
	`  34: 8fc30004 lw v1,4(s8)`
	`  38: 8fc20000 lw v0,0(s8)`
	`  3c: 00621021 addu v0,v1,v0`
	`  40: afc20008 sw v0,8(s8)`
	`  44: 8fc20004 lw v0,4(s8)`
	`  48: afc20000 sw v0,0(s8)`
	`  4c: 8fc20008 lw v0,8(s8)`
	`  50: afc20004 sw v0,4(s8)`
	`  54: 8fc2000c lw v0,12(s8)`
	`  58: 24420001 addiu v0,v0,1`
	`  5c: afc2000c sw v0,12(s8)`
	`  60: 08000009 j 24 <main+0x24>`
	`  64: 00000000 nop`
	`  68: 00001021 move v0,zero`
	`  6c: 03c0e821 move sp,s8`
	`  70: 8fbe0010 lw s8,16(sp)`
	`  74: 27bd0018 addiu sp,sp,24`
	`  78: 03e00008 jr ra`
	`  7c: 00000000 nop`

Producto escalar de dos vectores

El segundo algoritmo es un producto escalar de dos vectores. Ambos vectores son de 10 elementos, el código en C y el binario correspondiente son los que siguen:

Producto escalar	Código ensamblador para MIPS

`int main() {`
`  int v1[10];`
`  int v2[10];`
`  int result;`
`  int temp;`
`  int i;`
`  int j;`
`  for(i=0; i<10; i++) {`
`  v1[i] = i;`
`  v2[i] = 9-i;`
`  }`
	`00000000 <main>:`
	`  0: 27bdff98 addiu sp,sp,-104`
	`  4: afbe0060 sw s8,96(sp)`
	`  8: 03a0f021 move s8,sp`
	`  c: afc00058 sw zero,88(s8)`
	`  10: 8fc20058 lw v0,88(s8)`
	`  14: 2842000a slti v0,v0,10`
	`  18: 10400012 beqz v0,64 <main+0x64>`
	`  1c: 00000000 nop`
	`  20: 8fc20058 lw v0,88(s8)`
	`  24: 00021080 sll v0,v0,0x2`

`  result = 0;`
`  for(i=0; i<10; i++) {`
`    temp = 0;`
`    for(j=0; j<v2[i]; j++) {`
`      temp = temp + v1[i];`
`    }`
`    result = result + temp;`
`  }`
 	
`  return 0;`
`}`
	`  28:  005e1821  addu  v1,v0,s8`
	`  2c:  8fc20058  lw  v0,88(s8)`
	`  30:  ac620000  sw  v0,0(v1)`
	`  34:  8fc20058  lw  v0,88(s8)`
	`  38:  00021080  sll  v0,v0,0x2`
	`  3c:  005e2021  addu  a0,v0,s8`
	`  40:  24030009  li  v1,9`
	`  44:  8fc20058  lw  v0,88(s8)`
	`  48:  00621023  subu  v0,v1,v0`
	`  4c:  ac820028  sw  v0,40(a0)`
	`  50:  8fc20058  lw  v0,88(s8)`
	`  54:  24420001  addiu  v0,v0,1`
	`  58:  afc20058  sw  v0,88(s8)`
	`  5c:  08000004  j  10 <main+0x10>`
	`  60:  00000000  nop`
	`  64:  afc00050  sw  zero,80(s8)`
	`  68:  afc00058  sw  zero,88(s8)`
	`  6c:  8fc20058  lw  v0,88(s8)`
	`  70:  2842000a  slti  v0,v0,10`
	`  74:  10400020  beqz  v0,f8 <main+0xf8>`
	`  78:  00000000  nop`
	`  7c:  afc00054  sw  zero,84(s8)`
	`  80:  afc0005c  sw  zero,92(s8)`
	`  84:  8fc20058  lw  v0,88(s8)`
	`  88:  00021080  sll  v0,v0,0x2`
	`  8c:  005e1021  addu  v0,v0,s8`
	`  90:  8c430028  lw  v1,40(v0)`
	`  94:  8fc2005c  lw  v0,92(s8)`
	`  98:  0043102a  slt  v0,v0,v1`
	`  9c:  1040000d  beqz  v0,d4 <main+0xd4>`
	`  a0:  00000000  nop`
	`  a4:  8fc20058  lw  v0,88(s8)`
	`  a8:  00021080  sll  v0,v0,0x2`
	`  ac:  005e1021  addu  v0,v0,s8`
	`  b0:  8fc30054  lw  v1,84(s8)`
	`  b4:  8c420000  lw  v0,0(v0)`
	`  b8:  00621021  addu  v0,v1,v0`
	`  bc:  afc20054  sw  v0,84(s8)`
	`  c0:  8fc2005c  lw  v0,92(s8)`
	`  c4:  24420001  addiu  v0,v0,1`
	`  c8:  afc2005c  sw  v0,92(s8)`
	`  cc:  08000021  j  84 <main+0x84>`
	`  d0:  00000000  nop`
	`  d4:  8fc30050  lw  v1,80(s8)`
	`  d8:  8fc20054  lw  v0,84(s8)`
	`  dc:  00621021  addu  v0,v1,v0`
	`  e0:  afc20050  sw  v0,80(s8)`
	`  e4:  8fc20058  lw  v0,88(s8)`
	`  e8:  24420001  addiu  v0,v0,1`
	`  ec:  afc20058  sw  v0,88(s8)`
	`  f0:  0800001b  j  6c <main+0x6c>`
	`  f4:  00000000  nop`
	`  f8:  00001021  move  v0,zero`
	`  fc:  03c0e821  move  sp,s8`
	`100:  8fbe0060  lw  s8,96(sp)`
	`104:  27bd0068  addiu  sp,sp,104`
	`108:  03e00008  jr  ra`
	`10c:  00000000  nop`

Algoritmo iterativo del máximo común divisor

Finalmente presentamos el listado del algoritmo de Euclides.

Algoritmo de Euclides	Código ensamblador para MIPS
`int main() {` `  int a = 2048;`	`00000000 <main>:` `  0:  27bdffe8  addiu  sp,sp,-24`

```
  int b = 4864;
  int t;
   while (a!=b) {
      if(a>b)
         a = a - b;
      else {
         t = a;
         a = b;
         b = b - t;
      }
   }

  return 0;
}
```

```
   4:  afbe0010      sw     s8,16(sp)
   8:  03a0f021      move   s8,sp
   c:  24020800      li     v0,2048
  10:  afc20000      sw     v0,0(s8)
  14:  24021300      li     v0,4864
  18:  afc20004      sw     v0,4(s8)
  1c:  8fc30000      lw     v1,0(s8)
  20:  8fc20004      lw     v0,4(s8)
  24:  10620016      beq    v1,v0,80
<main+0x80>
  28:  00000000      nop
  2c:  8fc20000      lw     v0,0(s8)
  30:  8fc30004      lw     v1,4(s8)
  34:  0062102a      slt    v0,v1,v0
  38:  10400007      beqz   v0,58 <main+0x58>
  3c:  00000000      nop
  40:  8fc20000      lw     v0,0(s8)
  44:  8fc30004      lw     v1,4(s8)
  48:  00431023      subu   v0,v0,v1
  4c:  afc20000      sw     v0,0(s8)
  50:  08000007      j      1c <main+0x1c>
  54:  00000000      nop
  58:  8fc20000      lw     v0,0(s8)
  5c:  afc20008      sw     v0,8(s8)
  60:  8fc20004      lw     v0,4(s8)
  64:  afc20000      sw     v0,0(s8)
  68:  8fc30004      lw     v1,4(s8)
  6c:  8fc20008      lw     v0,8(s8)
  70:  00621023      subu   v0,v1,v0
  74:  afc20004      sw     v0,4(s8)
  78:  08000007      j      1c <main+0x1c>
  7c:  00000000      nop
  80:  00001021      move   v0,zero
  84:  03c0e821      move   sp,s8
  88:  8fbe0010      lw     s8,16(sp)
  8c:  27bd0018      addiu  sp,sp,24
  90:  03e00008      jr     ra
  94:  00000000      nop
```