ÍNDICE

Abstract 7

1. Introducción 9
1.1. Definición del proyecto 11
1.2. Organización de la memoria 12
1.3. Motivación 13
1.4. Objetivos del proyecto 15
1.5. Metodología 16
1.6. Acrónimos y conceptos 17
1.6.1. CycleSim 17
1.6.2. PitufoSim 20
1.6.3. Segmentation (pipelining) 21
1.6.4. Unidades funcionales 23
1.6.5. Procesadores superescalares 25
1.6.6. In-Order VS Out-Of-Order 27
1.6.6.1. In-Order 27
1.6.6.2. Out-Of-Order 27

2. Análisis previo 29
2.1. Punto de partida 31
2.1.1. El simulador 31
2.1.2. El módulo CPU 31
2.1.3. La traza 32
2.2. Planificación 33

3. Especificación y diseño 35
3.1. La CPU 37
3.2. La traza 42

4. Implementación 45
4.1. Introducción 47
4.2. Estructura de las clases desarrolladas 47
4.2.1. Clase “cpu” 47
4.2.2. Clase “segmentation” 49
4.3. Enumeraciones/definiciones 50
4.4. Implemetación 51
4.4.1. Librería “segmentation” 51
4.4.2. Librería “cpu” 54
4.4.3. Configuración y salida del simulador 91

5. Pintool y la creación de trazas 95
5.1. Introducción 97
5.2. Reconocimiento de instrucciones 98
5.3. Utilización de PinTool 100

6. Estudio económico 101

7. Conclusiones y líneas de futuro 105
7.1. Conclusiones 107
7.2. Líneas de futuro 109

8. Bibliografia 113

Anexo I: Planificación
Anexo II: Lista de instrucciones

ÍNDICE DE FIGURAS

Figuras del capítulo 1: Introducción

1.1. Módulo con 2 puertos de entrada (A y B) y 2 puertos de salida (C y D) 11
1.2. Datos históricos y ley de Moore 13
1.3. Problemas del desarrollo de los microprocesadores 14
1.4. Esquema del simulador coche 17
1.5. Ejemplo de conexión entre los módulos "mem" y "bus" del simulador 18
1.6. Fragmento del archivo de configuración "pitufosim.cfg" 20
1.7. Diseño de la segmentación de un procesador DLX 21
1.8. Ejemplo de la ejecución de tres instrucciones, que tardan 5 ciclos cada una, en un microprocesador sin segmentación 21
1.9. Ejemplo de la ejecución de tres instrucciones, que tardan 5 ciclos cada una, en un microprocesador segmentado 22
1.10. Superpipeline del Pentium 4 22
1.11. Ejemplo de CPU con una única unidad funcional de coma flotante 23
1.12. Ejemplo de CPU con dos unidades funcionales de coma flotante 24
1.13. La clasificación de Flynn 25
1.14. Ejemplo de procesador escalar 26
1.15. Ejemplo de procesador superescalar de grado 2 26
1.16. Detalle de la ejecución de 2 instrucciones en un procesador superescalar de grado 2 26

Figuras del capítulo 3: Especificación y diseño

3.1 Esquema de la comunicación entre estados del pipeline 37
3.2 Esquema del diseño de la CPU de este proyecto 38
3.3 Buffers de un estado cualquiera para este proyecto 38
3.4 Diagrama de clases de "cpu". 40
3.5 Diagrama de clases de "seg_state" y sus clases heredadas (segmentation.cpp) 41
3.6 Esquema de los archivos de traza utilizados en un ejemplo con 4 CPUs 42
3.7 Ejemplo de parte del contenido de los archivos de una traza 43
3.8 Instrucciones que unifican otras instrucciones debido a su comportamiento en la simulación 44

Figuras del capítulo 4: Implementación

4.1 Estructuras de datos de la clase "cpu" 47
4.2 Estructura de datos de los parámetros de la CPU 48
4.3 Estructura de datos de la clase base de los estados de la segmentación 49

4.4 Definición de las clases heredadas de “seg_state” 49
4.5 Esquema de la herencia de clases utilizada 49
4.6 Definiciones estados y unidades funcionales 50
4.7 Esquema de la segmentación. Comunicación entre estados mediante buffers de instrucciones 59
4.8 Esquema del trabajo secuencial de los estados de la CPU durante un ciclo 60
4.9 Definición de los ciclos iniciales de los estados del pipeline 61
4.10 Código para la creación y gestión de puestos de I/O 62
4.11 Definición del número de unidades funcionales de cada tipo 62
4.12 Código para la creación de las F.U. y definición de sus respectivas latencias 62
4.13 Código que crea nregisters registros 63
4.14 Llamada a la instrucción que carga el vector ins_to_unit[ ] 63
4.15 Ejemplo de error por fallo en el control de estados. La instrucción 4 se pierde 67
4.16 Comportamiento del sistema de control inverso en el anterior ejemplo 68
4.17 Código de control de Commit 69
4.18 Código de control de Execution 69
4.19 Código de control de Issue 70
4.20 Código de control de Decode 71
4.21 Código de control de Fetch 71
4.22 Código de control de los flags “branch” y “ending” 72
4.23 Efecto branch en el pipeline 73
4.24 Efecto branch en el pipeline simulado 73
4.25 Sección de código que controla la finalización de la CPU 74
4.26 Sección de código de la obtención de instrucciones 74
4.27 Sección de código del servicio de instrucción al Decode y verificación de Branch 75
4.28 Código de servició de instrucción a Issue 75
4.29 Sección de código de la resolución del estado Issue 76
4.30 Esquema estructural del estado 76
4.31 Esquema del caso genérico de ejecución de instrucciones 77
4.32 Código que trata la instrucción de tipo Load en el estado Execution 77
4.33 Código de tratamiento de las instrucciones que no son de acceso a memoria 78
4.34 Código que trata la segunda parte de una instrucción compleja 78
4.35 Código de la simulación del estado Commit 79
4.36 Código de la información extraida de la simulación de la CPU 80
4.37 Inicialización del vector ins_to_unit 81
4.38 Sección del código de la definición del vector “ins_to_unit[ ]” 81
4.39 Código que determina la unidad funcional que utiliza una instrucción 83
4.40 Código que consulta si hay disponible alguna unidad funcional de un tipo concreto 84
4.41 Estructura del método checkResources() 86
4.42 Código que verifica si hay una unidad funcional libre del tipo solicitado 86

4.43 Consulta de disponibilidad de los registros fuente 87
4.44 Consulta de disponibilidad de los registros destino 87
4.45 Código para la ocupación de los recursos necesarios para una instrucción ... 87
4.46 Código que consulta si puede finalizar la simulación de la CPU 89
4.47 Código para el tratamiento de las estadísticas de inactividad 90
4.48 Configuración del simulador (pitufosim.cfg) 91
4.49 Salida de la simulación de un binario HelloWorld 92

Figuras del capítulo 5: Pintool y la creación de trazas

5.1 Logotipo de PinTool 97
5.2 Ubicación de los archivos relacionados con el "Tracer" 97
5.3 Código que genera el listado de instrucciones que trabaja el procesador utilizado 98
5.4 Enlazado de la PinTool con Automake 100
5.5 Comando para generar una nueva traza 100

Figuras del capítulo 6: Estudio económico

6.1 Lista de horas utilizadas para el deasarrollo de este proyecto 103
6.2 Salario en función del puesto de trabajo 104
6.3 Coste humano del proyecto 104
6.4 Coste total del proyecto 104

Figuras del capítulo 7: Conclusiones y líneas de futuro

7.1 Estructura de los parámetros configurables de la CPU 111
7.2 Ejemplo del código de lectura de datos configurables 112
7.3 Estructura genérica del código de lectura de un nuevo parámetro configurable 112
7.4 Ejemplo de la definición de un nuevo parámetro configurable 112

Figuras del capítulo 8: Bibliografía

ABSTRACT

La evolución del campo de la informática ha sido impresionante desde su nacimiento. Si echamos un vistazo a la historia de la informática podremos apreciar los grandes cambios generados y el gran aumento de la capacidad computacional de los ordenadores hasta la actualidad. Este gran progreso de la oferta es fruto de una insaciable demanda poder computacional, hasta hoy en día. Actualmente los procesadores han encontrado un límite físico que les impide evolucionar según la Ley de Moore, obligando a los fabricantes de microprocesadores a investigar nuevas técnicas y tecnologías para paliar la demanda del mercado.

Debido a las anteriores premisas, surge un proyecto para desarrollar un simulador capaz de emular el trabajo de un microprocesador basado en la infraestructura CycleSim, con el objetivo de estudiar los cambios en el rendimiento de un procesador modificando la configuración de los diferentes elementos de este.

El proyecto al cual hace referencia esta memoria es una de las piezas del proyecto anteriormente mencionado. Concretamente se centra en desarrollar una CPU superescalar "in-order" para la infraestructura CycleSim, partiendo de una versión inicial ya existente. Además aumentará el número de instrucciones reconocidas por la CPU para poder generar trazas más completas a simular.

CAPÍTULO 1:

INTRODUCCIÓN

CAPÍTULO 1: INTRODUCCIÓN

1.1. Definición del proyecto

Este documento contiene la memoria del proyecto final de carrera "Desarrollo de un multiprocesador superescalar in-order en CycleSim". En este proyecto ampliamos una infraestructura de simulación de procesadores multicore llamada CycleSim. CycleSim es una infraestructura (librería C++) que permite la emulación/simulación de sistemas de proceso específico, entre otros la simulación de procesos multicore. Esta infraestructura desarrollada por el departamento de Arquitectura de Computadores de la UPC se encuentra en versiones iniciales. Por ello, es este proyecto ampliaremos los módulos (clases C++) de dicha infraestructura, concretamente aquellos módulos que procesan las instrucciones.

CycleSim como infraestructura de simulación o motor de simulación, construye simuladores a partir de la conexión de diferentes módulos y puertos. Por ejemplo, un módulo "Cola" recibiría información por sus entradas (puertos de entrada), procesaría la información y la enviaría por sus salidas (puertos de salida). En este proyecto nos centraremos en los módulos LectorTrazas y CPU. Como sus propios nombres indican, estos módulos se encargan de leer las instrucciones de varios ficheros ("la traza") e interpretar las instrucciones. al iniciar este proyecto, en la versión disponible de CycleSim, el módulo CPU sólo ejecutaba/simulaba instrucciones de tipo LOAD/STORE, es decir, instrucciones de lectura y escritura a memoria.

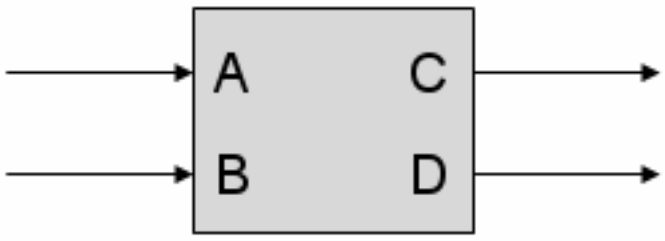


Fig.1.1: módulo con dos puertos de entrada (A y B) y dos puertos de salida (C y D) fuente: http://pcsostres.ac.upc.edu/cyclesim/doku.php/syntax:modules

El objetivo principal del proyecto es ampliar el módulo CPU para procesar el máximo número posible de instrucciones simulando un procesador multicore in-order (en orden). Para ello se tendrán en cuenta los conceptos teóricos propios de un procesador de este tipo con la finalidad de crear un simulador que se asemeje lo más posible a la realidad.

1.2. Organización de la memoria

La memoria esta estructurada en varios capítulos, que aportan datos independientes. La primera parte nos introduce al proyecto, explicando las motivaciones y los objetivos de este. Después se despliega un conjunto de contenidos didácticos sobre aspectos técnicos para su correcta comprensión y una explicación elaborada sobre el diseño que se ha decido implementar en el módulo de la CPU. Posteriormente se introduce en la implementación explicando el código desarrollado y los resultados de algunas pruebas. Para poder entender mejor el simulador se dedica un capítulo a explicar la obtención de las trazas con el programa Pin y su funcionamiento. También se adjuntará un manual, que tiene la intención de ayudar en futuras modificaciones y finalmente las conclusiones sobre el proyecto y la bibliografía utilizada durante su desarrollo.

Breve esquema de la estructura del proyecto:

1.- Introducción
2.- Análisis previo
3.- Especificación y diseño
4.- Implementación
5.- Pin y la creación trazas
6.- Estudio económico
7.- Conclusiones y líneas de futuro
8.- Bibliografía
Anexo I: Planificación
Anexo II: Lista de instrucciones

1.3. Motivación

La evolución de los microprocesadores ha sido veloz y escalada en respuesta a un gran mercado que estaba abierto a utilizar cada vez una mayor potencia de cálculo para un software cada vez más sofisticado.

Si fijamos la vista en lo que concierne a los microprocesadores, pudimos presenciar muchos años de lucha entre las principales empresas desarrolladoras de estos, las cuales sacaban al mercado modelos cada vez más potentes en cuestión de meses, utilizando varios equipos de trabajo y desarrollo de forma simultanea con objetivos diferentes.

De esta forma se cumplía la Ley de Moore, que, adaptada a los ordenadores actuales, pronosticó que aproximadamente se duplicaría la potencia de los procesadores cada 18 meses, hasta la actualidad.

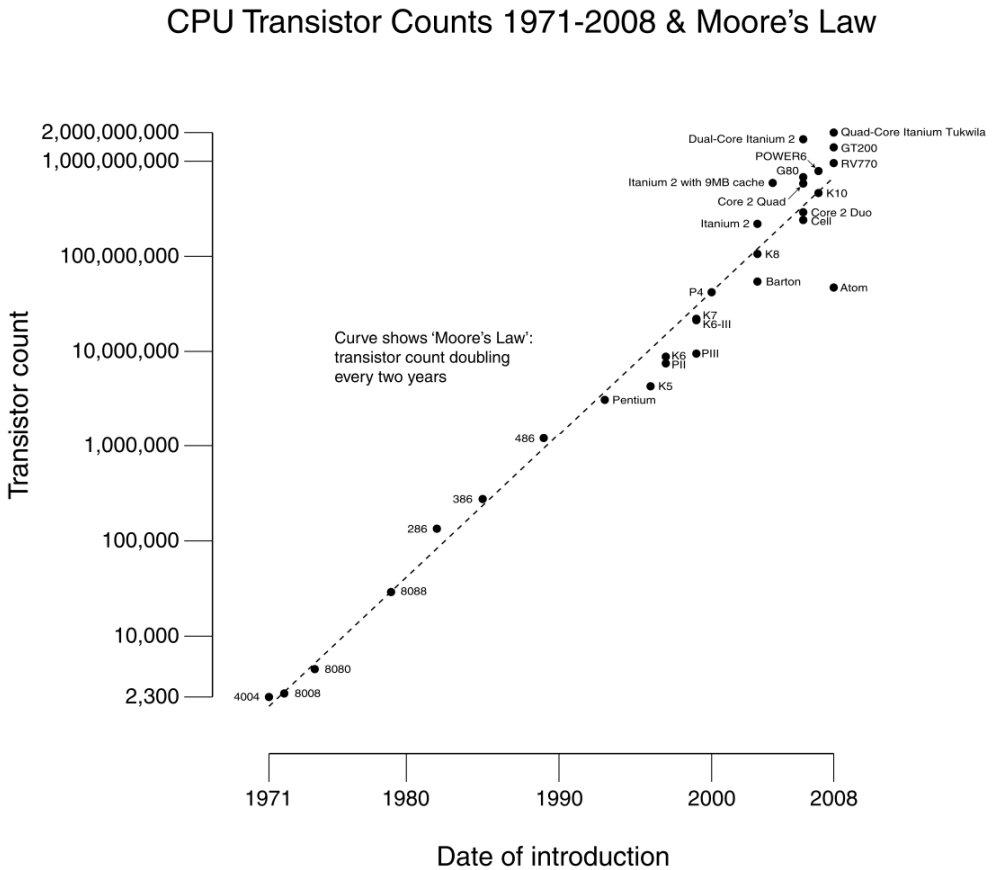


Fig 1.2: Datos históricos y ley de Moore.
Fuente: http://en.wikipedia.org/wiki/Moore's_law

El cofundador de Intel, Gordon E. Moore, pronosticó esta "ley" el 19 de abril de 1965 y el mismo afirmo que esta "predicción" seguiría teniendo validez hasta el año 2011 (fuente: artículo de *Rafael Fernandez-Font Pérez,* ver

bibliografía). De hecho, no es propiamente una ley, sino una predicción que se cumplía y que su sector la señaló claramente como válida, aunque ha sufrido mutaciones adaptándose a nuevas tecnologías (en realidad esta predicción se realizó antes del nacimiento de los microprocesadores).

Actualmente, los problemas producidos por el suministro de potencia eléctrica, los problemas en la refrigeración y el aumento del coste en la construcción de chips ya han puesto en jaque a la famosa predicción. Puesto que este camino ya no se puede seguir sin más, se esta profundizando en la búsqueda y desarrollo de técnicas para incrementar el rendimiento del microprocesador.

Fig 1.3: Problemas del desarrollo de los microprocesadores.

Se ha apostado por la utilización de múltiples CPUs de gran frecuencia teniendo en cuenta el gasto de energía, el calor que emiten y técnicas para aprovechar el paralelismo y la profundidad de las instrucciones.

Este proyecto trata de desarrollar una parte que pueda simular parcialmente una máquina (la parte que nos interesa), con la finalidad de obtener una configuración ideal para obtener el mejor rendimiento posible utilizando el menor hardware necesario, rebajando así su coste final.

1.4. Objetivos del proyecto

A continuación se listan los objetivos declarados par este proyecto:

1.- Programar un multiprocesador superescalar in-order para el entorno de simulación CycleSim.

2.- Mejora de las funcionalidades de la CPU.

3.- Mejorar la lectura de trazas.

4.- Simular todas las etapas del pipeline de un procesados superescalar.

5.- Controlar las dependencias de datos entre instrucciones.

6.- Controlar la disponibilidad de las diferentes unidades funcionales.

7.- Ampliación de las opciones configurables de la CPU.

8.- Reconocimiento del máximo número posible de instrucciones.

1.5. Metodología

Para desarrollar este proyecto ha sido necesario un amplio estudio inicial, con la finalidad de aprender todos los conceptos necesarios para poder crear un simulador con las características apropiadas.

Durante este periodo se obtuvo información a través de Internet en multitud de páginas Web, contrastando información para asegurar la veracidad de la misma. También se adquirió información de varios libros especializados en el la temática, que me ayudaron sobretodo a dar los primeros pasos en el mundo de los procesadores superescalares y el proceso de segmentación. Los títulos de los libros consultados pueden consultarse en la bibliografía.

El libro de "Arquitectura de Computadores: un enfoque cuantitativo", de Hennessy y Patterson, es especialmente didáctico en el comienzo del estudio, gracias a este, pude adquirir una base que me serviría para progresar. El resto de libros son utilizados para profundizar en los diferentes aspectos del temario.

Para desarrollar el proyecto utilicé un portátil con un procesador Intel Core 2 Duo T5500 1.66GHz, utilizando una distribución Ubuntu de 64bits, pues la de 32btis propiciaba problemas con el simulador, que estaba configurado para 64bits.

La infraestructura del simulador esta desarrollada en C++. Para programar el módulo de la CPU y las clases necesarias para su funcionamiento utilice GEdit y la herramienta Clewn, que combina el debugger GDB y el GVim. Clewn ha resultado ser una herramienta muy útil para debugar el programa, bastante intuitivo en ciertos aspectos, de manejo sencillo y con mucha información sobre la utilización de GDB en Internet. De esta manera pude aprovechar sus funcionalidades rápidamente.

La infraestructura CycleSim y el simulador PitufoSim me los facilito mi director de proyecto y ha sido el punto de partida de este. Una vez asumidos los conocimientos necesarios y realizando las consultas precisas con el director del proyecto para aclarar las dudas que surgían, el proyecto fue cogiendo forma.

Por último, se utilizó el programa Pin (www.pintool.org) para crear una traza que representase la ejecución de un programa, que posteriormente sería simulado. El programa Pin se trata de un instrumentador de código, con el cual se puede introducir código propio en la ejecución de un programa, de esta manera se pueden obtener las instrucciones utilizadas durante la ejecución de un programa y extraer unos patrones para emular su ejecución.

1.6. Acrónimos y conceptos

A lo largo de este capítulo explicaré los conocimientos necesarios para poder entender este proyecto y su finalidad. Dada la naturaleza del trabajo se ha de asumir que los conceptos a tener en cuenta no son pocos ni triviales, por lo que creo necesario explicarlos, aunque será necesario que el lector posea ciertos conocimientos básicos sobre el campo de la informática.

1.6.1. CycleSim

CycleSim es una infraestructura creada en C++, que se utiliza como motor para la creación de simuladores basados en trazas. Esta infraestructura se genera mediante diferentes módulos interconectados formando un simulador. Como ejemplo ilustrativo, podríamos indicar el de un coche que posee diferentes partes en su estructura, tiene un chasis, un sistema de transmisión, un sistema de refrigeración, otro eléctrico, etc., los cuales representarían los diferentes módulos del coche, y también tiene un motor, que hace funcionar el resto de piezas mecánicas, el cual representaría CycleSim. Uniendo estas piezas obtenemos el coche y nuestro simulador.

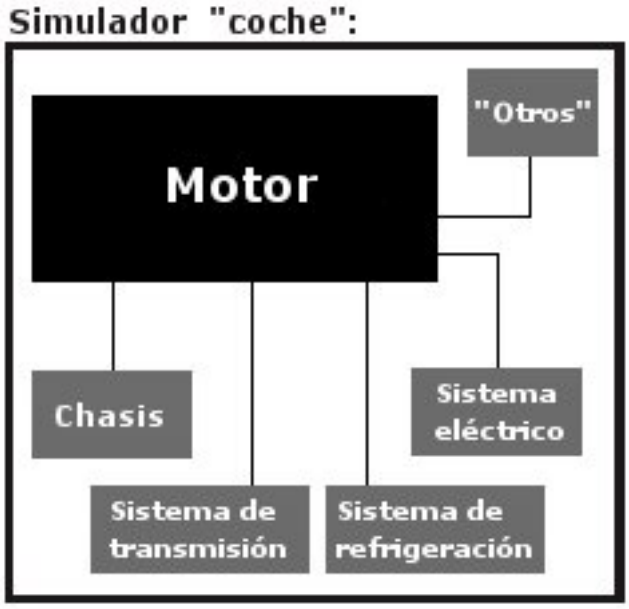


Fig. 1.4: Esquema del simulador coche, donde las cuadros grises representan los módulos y el negro CycleSim.

Esta infraestructura controla la ejecución de todos los módulos del simulador en cada uno de sus ciclos emulados. De esta forma, se puede decidir de forma condicional la ejecución de un módulo concreto en un ciclo determinado. Por ejemplo, si deseamos simular que el módulo CPU ha de

esperar unos recursos que llegarán en 6 ciclos, debemos definir el parámetro next_cycle con el valor del ciclo actual más 6 ciclos. De esta manera no se ejecutará inútilmente y se evitaran posibles errores.

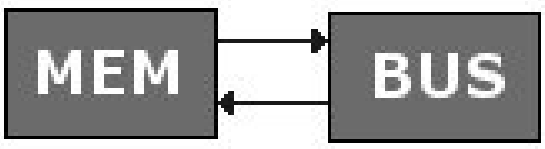
CycleSim trabaja con dos clases base a través de las cuales podemos programar e interconectar para crear un simulador, se trata de "Sims" y "Modules". La clase Sim es la base del simulador que queremos desarrollar. A diferencia de los módulos, este posee una función "main" que debe tener una estructura estándar compuesta por:

1.- Creación de los módulos del simulador.
2.- Conexión apropiada de los módulos.
3.- Inicio del motor de simulación.

Los Modules son las piezas con las cuales realizaremos nuestro "puzzle". Su concepto es bastante simple: la clase module posee una serie de parámetros básicos que utiliza el simulador para su control como:

- _name→ Identifica el módulo con una cadena de caracteres.
- _stats → Estado del módulo.
- next_cycle → El siguiente ciclo en que se ejecutará este módulo.
- address → Dirección base del módulo.

La conexión entre los diferentes módulos se realiza en el simulador. En la siguiente figura podemos observar un ejemplo gráfico y su código de cómo se realiza esta conexión.



Código para la conexión

```
mem[i]->outdata >> bus.in.data[i];
mem[i]->indata << bus.out.data[i];
mem[i]->reject << bus.out.reject[i];
mem[i]->busy >> bus.in.busy[i];
```

Fig. 1.5: Ejemplo de conexión entre los módulos "mem" y "bus" del simulador.

Además, se definen unos métodos requeridos para cada módulo que son necesarios para realizar su correcta ejecución y programación. Cada ciclo del simulador esta dividido en dos (start y end), de tal forma, que el simulador ejecuta todas las funciones start() de los módulos que tienen definida su ejecución en ciclo actual, y posteriormente se ejecutaran todas las funciones end() correspondientes a los mismos módulos.

La clase "Module" define como virtual los siguientes métodos, para que las clases heredadas de esta puedan definir sus propias funciones, las cuales serás las que realmente serán ejecutadas. Los métodos declarados virtuales con esta función son:

- reset( )
- start( )
- end( )
- print_hook( )

Para poder especificar un módulo que trabaje para un simulador basado en CycleSim es necesario conocer en que situaciones van a ser llamados estos métodos. Se expondrán a continuación.

Reset: define todos los parámetros y estructuras del módulo con los valores iniciales. Es utilizado para la inicialización del módulo en la etapa inicial del simulador y puede ser utilizado para realizar tareas de reinicialización.

Los módulos son ejecutados en dos fases durante un mismo ciclo (Start y End), produciendo que se realice la ejecución secuencial ordenada de los métodos Start de todos los módulos programados para ejecutarse en ese ciclo y posteriormente, los métodos End de estos mismo módulos y en el mismo orden.

Start: función que se ejecuta al inicio del ciclo si el parámetro next_cycle es igual al ciclo actual.

End: con la misma condición de ejecución que Start, se asegura que si ésta es ejecutada al inicio de un ciclo, End también será ejecutado al final del mismo ciclo.

print_hook: función que se ejecuta durante la finalización del simulador para extraer por pantalla los resultados estadísticos de cada uno de los módulos utilizados durante la simulación.

Con estos y otros parámetros, se puede controlar un módulo o desarrollar uno nuevo con un nuevo propósito, tan solo hemos de crear una nueva clase que sea heredada de la clase "Module" y añadir nuestros nuevos parámetros y funciones que realicen los resultados deseados. Finalmente tan solo se tendrá que definir la relación de los módulos en la función "main" del simulador para que estos se puedan comunicar.

1.6.2. PitufoSim

Este es el simulador objeto de desarrollado de este proyecto, concretamente del desarrollo del módulo CPU. El simulador pretende emular el comportamiento de un procesador sin el traspaso de datos, únicamente simulando el tráfico de memoria y trabajando con el comportamiento lógico de los componentes del sistema como son el procesador, la memoria, el bus, etc.

PitufoSim simula la ejecución de una aplicación basada en trazas, que son las instrucciones que serian ejecutadas por un programa. Esta simulación depende de muchos factores, por ejemplo, según la latencia de una unidad funcional, se podría producir un bloqueo en la ejecución por falta de recursos, obteniendo como resultado que la simulación se ha visto obligada a realizar más ciclos de simulación, debido a esta falta de recursos.

Estos parámetros y sus valores serán objeto de estudio, una vez finalizado el simulador y por lo tanto es conveniente poder modificarlos con facilidad. PitufoSim se puede configurar a través del archivo externo pitufosim.cfg, que permite cambiar la parámetros configurables del simulador sin necesidad de compilarlo nuevamente, pudiendo así modificar rápidamente parámetros como la latencia o el número de unidades funcionales de algún tipo, el número de registros o la traza que se va a simular por ejemplo.

```
sim.page_size = 4096
trace = /home/chus/scratch/trunk/src/build/src/simulator/PitufoSim/helloworld

bus.requests_limit = 1

processor.n = 1
processor.0.lm.address = 0x1000000000000000
processor.0.lm.latency = 10
processor.0.lm.size = 64
processor.0.cache.line_size = 128
processor.0.dma.queue_size = 16
processor.0.tlb.size = 128
processor.0.ins.fetch_width = 4
processor.0.ins.regs_latency = 1
processor.0.ins.num_io_ports = 1
processor.0.ins.func_units_whitout = 100
processor.0.ins.func_units_memory_access = 2
processor.0.ins.func_units_int = 2
processor.0.ins.func_units_float = 2
processor.0.ins.func_units_branch = 1
processor.0.ins.number_registers = 1024
processor.0.ins.end_type = 0
processor.0.ins.lat_unit_whitout = 1
processor.0.ins.lat_unit_memory_access = 6
processor.0.ins.lat_unit_int = 2
processor.0.ins.lat_unit_float = 4
processor.0.ins.lat_unit_branch = 3
```

Fig. 1.6: Fragmento del archivo de configuración "pitufosim.cfg"

1.6.3. Segmentación (*pipelining*)

Este es un método por el cual se puede aumentar el rendimiento del microprocesador explotando el paralelismo de flujo. La técnica consiste en dividir las instrucciones en diferentes estados de forma que cada uno de estos tarde aproximadamente el mismo tiempo en realizarse (~1 ciclo de reloj). De esta forma se consigue un solapamiento de instrucciones que procesan diferentes estados dentro de la segmentación.

Aunque no disminuye el tiempo de ejecución de una instrucción (en realidad es aumentado ligeramente debido a las latencias entre estados), obtiene como resultado la finalización de una instrucción por cada tiempo de ejecución de un estado, una vez se llenado el pipeline.

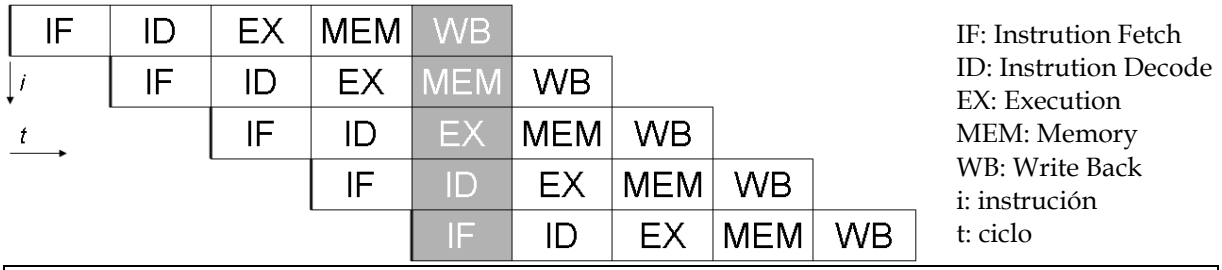


Fig. 1.7: Diseño de la segmentación de un procesador DLX. Este modelo se suele utilizar en la mayoría de documentos didácticos.
Fuente: http://es.wikipedia.org/wiki/Segmentaci%C3%B3n_de_cauce

Una de las ventajas de esta técnica es que es completamente transparente para el programador y por lo tanto no ha de tener ningún factor más en cuenta a la hora de programar si desea hacerlo de forma eficiente.

Veamos un ejemplo para ilustrar esta técnica:

- En el primer caso se muestra un ejemplo plano, en el que la CPU no utiliza segmentación:

Ciclos	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Instrucciones	Ins1					Ins2					Ins3				

Fig. 1.8: Ejemplo de la ejecución de tres instrucciones, que tardan 5 ciclos cada una, en un microprocesador sin segmentación. El resultado son 15 ciclos para esta ejecución.

- En contraposición, la siguiente tabla muestra una CPU que si utiliza una segmentación (se presupone que las instrucciones no contienen dependencias de datos ni de unidades funcionales entre ellas):

Ciclos	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Instrucciones	Ins1														
		Ins2													
			Ins3												

Fig. 1.9: Ejemplo de la ejecución de tres instrucciones, que tardan 5 ciclos cada una, en un microprocesador segmentado. El resultado son 7 ciclos para esta ejecución.

En las dos figuras anteriores podemos observar que en condiciones ideales (no necesariamente reales), el rendimiento producido por esta técnica mejora notablemente respecto a su ausencia. Claro esta que la implementación de esta técnica no es algo trivial y conlleva un coste adicional, pero los beneficios son irrefutables.

Unas empresas apostaron por aprovechar al máximo esta técnica, como podemos ver a continuación. Otras en cambio, utilizaron la segmentación con una baja profundidad, pero a cambio de una lógica simple que les permitiría explotar una frecuencia mayor.

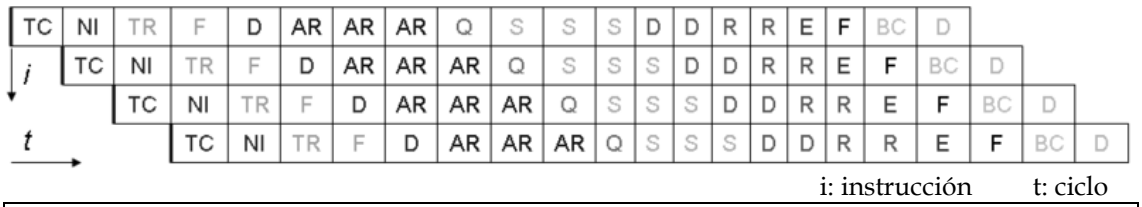


Fig. 1.10: Superpipeline del Pentium 4
Fuente: http://es.wikipedia.org/wiki/Segmentaci%C3%B3n_(inform%C3%A1tica)

1.6.4. Unidades funcionales

Las unidades funcionales son componentes específicos que forman parte de la CPU y realizan las operaciones y cálculos solicitados por las instrucciones ejecutadas. Suelen contener algunos registros, unidades internas para su propio uso (como ALU's o de punto flotante) y componentes específicos para cada unidad.

Podemos encontrarnos unidades funcionales de diferentes tipos en función de su funcionalidad, como por ejemplo:

- Unidad de enteros
- Unidad de punto flotante
- Unidad de salto
- Unidad de acceso a memoria
- Unidad de control
- Unidades especificas (según fabricante)

Cada fabricante evoluciona las unidades funcionales de sus procesadores para poder dar servicio al repertorio de instrucciones que utiliza y para la mejora de su rendimiento. En los procesadores superescalares se encuentran varias unidades funcionales de un mismo tipo trabajando en paralelo, con la finalidad de servir más de una instrucción de un mismo tipo simultáneamente, sin necesidad de esperar la liberación de la unidad funcional.

Como ejemplo ilustrador de este problema podemos poner la unidad de punto flotante, cuya ejecución es larga. Si la CPU recibe dos instrucciones de punto flotante seguidas y solo tuviese una unidad punto flotante el resultado (sin tener en cuenta otras medidas) seria el siguiente:

Ciclo	1	2	3	4	5	6	7	8	9	10	11	12
Instrucción 1	IF	ID	EX	EX	EX	MEM	WB					
Instrucción 2		IF	ID			EX	EX	EX	MEM	WB		

Fig. 1.11: Ejemplo de CPU con una única unidad funcional de coma flotante.

La instrucción 2 intenta obtener la unidad de punto flotante, pero como sólo hay una, se queda a la espera de la anterior instrucción finalice su uso. En el ciclo 5 se libera la unidad y en el 6 es utilizada por la segunda instrucción. Se puede apreciar en este ejemplo como se propaga la segunda instrucción a través de los ciclos entrando en inactividad.

En cambio, con dos unidades funcionales de punto flotante y teniendo en cuenta que ambas instrucciones no tienen dependencias de datos entre ellas:

Ciclo	1	2	3	4	5	6	7	8	9	10	11	12
Instrucción 1	IF	ID	EX	EX	EX	MEM	WB					
Instrucción 2		IF	ID	EX	EX	EX	MEM	WB				

Fig. 1.12: Ejemplo de CPU con dos unidades funcionales de coma flotante.

Se sirve la instrucción 1 y al ciclo siguiente se serviría la instrucción 2, pero ambas instrucciones tendrán recursos libres para poder realizar sus ejecuciones sin necesidad de realizar ninguna espera adicional.

La ventaja es obvia, ¿por qué no duplicar o cuadruplicar sin más todas las unidades funcionales? Hay más parámetros a tener en cuenta para la inserción de más unidades funcionales de un mismo tipo. El primero y como la mayoría de veces el más importante, es el coste del chip: más hardware implica más coste y mayores problemas de integración. Otros factores a tener en cuenta son la energía consumida, la latencia producida, el incremento de la lógica añadida para decidir que unidad esta libre y cual servirla, la tendencia del número de instrucciones que se ejecutan secuencialmente de ese mismo tipo de instrucción (es vital no añadir hardware inútil), el espacio ocupado (es finito, el procesador ha de ser fiel al socket para el que es diseñado), etc.

1.6.5. Procesadores superescalares

Se trata de un tipo de microarquitectura de procesador que aprovecha el paralelismo de instrucciones para ejecutar varias instrucciones por ciclo de reloj. En contraposición, a los procesadores que trabajan con una instrucción por ciclo simultáneamente, los llamamos escalares.

Además es compatible con el paralelismo de flujo, pudiendo así explotar ambos recursos para obtener un notable rendimiento.

Un procesador superescalar es categorizado como de tipo MIMD en la clasificación de Flynn.

SISD → Flujo de instrucciones único – Flujo de datos único SIMD → Flujo de instrucciones único – Flujo de datos múltiple MISD → Flujo de instrucciones múltiple – Flujo de datos único MIMD →Flujo de instrucciones múltiple – Flujo de datos múltiple
Fig. 1.13: La clasificación de Flynn divide a las computadoras en cuatro grupos principales

Denominamos "grado", en un procesador superescalar, al número máximo de instrucciones que es capaz de manejar, en una misma etapa concreta, en un ciclo. De esta forma, un procesador que es capaz de decodificar 4 instrucciones simultáneamente en un mismo ciclo, decimos que tiene grado 4 en su etapa "decode".

El número y tipo de unidades funciones definen el grado de cada una de las etapas del pipeline en un procesador superescalar. Una mala combinación de unidades funcionales resultaría en un impacto negativo en el rendimiento del procesador.

Es necesario estudiar empírica y estadísticamente los resultados obtenidos con diferentes configuraciones, con la finalidad de hallar la forma correcta con la que el hardware del procesador no sea desaprovechado ni insuficiente durante la ejecución de diferentes aplicaciones. Con este método se trata de obtener un buena relación entre rendimiento y coste, pues esto define en mayor parte el existo del procesador en el mercado.

Partiendo del ejemplo del concepto anteriormente explicado, la segmentación, este sería el comportamiento (siempre hablando en condiciones ideales) de un procesador escalar segmentado:

Ciclos	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Instrucciones	Ins1														
		Ins2													
			Ins3												

Fig. 1.14: Ejemplo de procesador escalar.

Para poder observar la diferencia de este concepto, comparemos ahora con un procesador superescalar de grado 2, también en condiciones ideales.

Ciclos	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Instrucciones	Ins1														
	Ins2														
		Ins3													
		Ins4													
			Ins5												
			Ins6												

Fig. 1.15: Ejemplo de procesador superescalar de grado 2.

Como podemos apreciar, el procesador superescalar de grado 2 ejecuta el doble de instrucciones que el escalar en el mismo número de ciclos. Hagamos un zoom en las instrucciones 1 y 2 del ejemplo:

Ciclo	**1**	**2**	**3**	**4**	**5**
Estado	**Fetch**	**Decode**	**Issue**	**Execution**	**Commit**
Instrucción	Ins1	Ins1	Ins1	Ins1	Ins1
	Ins2	Ins2	Ins2	Ins2	Ins2

Fig. 1.16: Detalle de la ejecución de 2 instrucciones en un procesador superescalar de grado 2

En el procesador escalar se tratan varias instrucciones en el pipeline, pero tan solo 1 por estado. Aquí podemos observar, en el caso superescalar de grado 2, que se trabaja con más de una instrucción por estado, de forma paralela.

1.6.6. In-Order VS Out-Of-Order

Estos paradigmas definen el modelo o patron que utiliza un procesador para trabajar con las instrucciones a ejecutar.

1.6.6.1. In-Order

Tal y como su nombre indica, "en orden", es un tipo de procesador que se obtiene como resultado de una ejecución lineal de las instrucciones leídas. La teoría de este es bastante simple, en la etapa inicial obtenemos una instrucción y la trasladamos al siguiente estado del pipeline hasta que finalice todos los estados. Esto implica que si una instrucción esta esperando obtener recursos para ser lanzada a ejecutar en el estado de lanzamiento, ninguna otra instrucción podrá adelantarse aunque su ejecución no modifique el curso normal del programa, lo que se traduce directamente en ciclos de inactividad por parte de la CPU, a la espera de que los recursos de la instrucción que esta bloqueando el pipeline lleguen, lo que disminuye enormemente el rendimiento del procesador. Por otra parte, tiene sus ventajas, el control de la ejecución es simplificado al máximo, lo que produce una rápida lógica y veloz ejecución, además de que resulta más económico en coste y energía.

1.6.6.2. Out-Of-Order

Los procesadores del tipo "Fuera de orden" permiten la ejecución de las instrucciones de forma "no ordenada", una vez entran en el pipeline. Esto permite que una instrucción bloqueada por falta de recursos deje paso a la siguiente instrucción que tenga los recursos necesarios, realizándose un desorden en la ejecución de las instrucciones. De esta forma, se aprovechan los ciclos de inactividad que producían las instrucciones bloqueadas, obteniendo un mayor rendimiento del procesador.

El proceso es mucho más complejo que el de su predecesor. Siguiendo la teoría, para la ejecución en un procesador out-of-order se realiza el Fetch de una instrucción (que recibe en orden), la decodifica y la encola en una lista de instrucciones. Esta cola del estado Issue, comprueba si hay disponibles recursos para ejecutar su primera instrucción, en caso de que no haya recursos, comprueba si la siguiente instrucción tiene dependencias con la que esta en espera y si no los tiene (y esta nueva instrucción si tiene los recursos que necesita) lanza esa instrucción.

De esta situación surge la ejecución fuera de orden, pues lo más probable es que esa instrucción termine su ejecución antes que la que había saltado. Para que el programa no sufra cambios es su ejecución normal, la instrucción

adelanta esperará en el estado "write-back" del pipeline hasta que la anterior instrucción finalice y se realice el "commit" de la misma forma en la que entraron en la CPU, ordenada.

Para poder realizar esto es necesario tener varios factores en cuenta, como las interrupciones, problemas de acceso a mismo punto de la memoria, etc. Los conflictos de dependencias de datos son filtrados durante el lanzamiento de tal forma que no sea necesario dar marcha atrás, no llegan a producirse.

Por otra parte, ¿qué pasa si se adelantan por ejemplo a una instrucción de salto, y el resultado posterior de ese salto implica la no ejecución de esa instrucción? La instrucción adelantada estará esperando en el estado write-back del pipeline, sin hacer commit, por lo tanto, el resultado de su ejecución no es apreciado fuera de la CPU y es desechado. Aun así, para poder retornar al estado previo es necesario lanzar el protocolo de "marcha atras" de la CPU.

Estos son algunos de los peligros que implica la ejecución fuera de orden y como podemos imaginar, la lógica utiliza es mucho más compleja y conlleva problemas colaterales a tener en cuenta. Por supuesto, estos problemas los podemos añadir a la obtención de un procesador de mayor rendimiento y menor tasa de inactividad.

CAPÍTULO 2:

ANÁLISIS PREVIO

CAPÍTULO 2: ANÁLISIS PREVIO

2.1. Punto de partida

A continuación comentaré la situación desde la cual empecé a trabajar. Este proyecto desarrolla un módulo concreto del simulador PitufoSim, la CPU, aunque en vez de desarrollar el término correcto seria evolucionar, pues este proyecto parte de una versión existente con unas funciones básicas sobre instrucciones de acceso a memoria.

2.1.1. El simulador.

El primer contacto con el simulador se realizó con una versión bastante simple, aunque muy útil para realizar su función didáctica. Una vez asentada una base, se trabajo a partir de otra versión más elaborada, con una CPU que continuaba siendo muy simple.

2.1.2. El módulo CPU.

Esta CPU obtenía instrucciones y únicamente ejecutaba aquellas que eran accesos a memoria (LOAD/STORE). Su procesamiento era básico, se trabaja una única instrucción simultánea y no se obtenía la siguiente hasta la completa finalización de esta. Tampoco existía una implementación de un pipeline, para el proceso de varias instrucciones simultáneas dentro de una CPU, ni el concepto de unidad funcional, ni el concepto de Fetch_Width, a través del cual se podía definir el número máximo de instrucciones por ciclo que se podía traer a la CPU. La versión preliminar de la CPU contenía una versión inicial del control de dependencias, basándose en la ocupación de registros y puertos en el caso de los accesos a memoria. El control realizado era bastante pobre debido a que no definía ningún tipo de diferencia entre las latencias producidas por las diferentes instrucciones reconocidas, de tal forma que un movimiento de registros producía la misma latencia que una multiplicación de floats, hecho bastante irreal.

Aun así, esa versión inicial de control de dependencias se convertiría en un punto de partida factible para desarrollar mi propia versión para este proyecto.

2.1.3. La traza.

Llamamos traza a un conjunto de datos que constituyen una secuencia de acciones o instrucciones, las cuales representan la ejecución de un programa.

La traza es creada mediante una pintool programada en C++ para el intrumentador de código Pin (PinTool). Pin ejecuta el binario que deseamos simular e inyecta código en cada instrucción, permitiendo obtener la información necesaria para generar unos datos, los cuales representan la ejecución de este binario.

La primera traza con la que trabajé también me fue proporcionada juntamente con el simulador. Esta traza simulaba la ejecución de un programa "HelloWorld" y tan sólo identificaba las instrucciones de acceso a memoria y los saltos, el resto de instrucciones venia dado como instrucciones de tipo "Others".

Esto es debido a que la pintool que generar la traza correspondiente a un binario, tan solo era capaz de identificar los accesos a memoria (LOAD, LOAD2, STORE y LOADSTORE) y los saltos (sin hacer ninguna diferenciación entre las diferentes instrucciones de salto). Este punto es también uno de los objetivo de mejora para el proyecto.

2.2. Planificación

Nota: En el Anexo I se puede observar una planificación más detallada del proyecto mediante un diagrama de Gantt.

Dada la naturaleza del proyecto, ha sido necesario realizar un estudio inicial para adquirir los conocimientos previos para el desarrollo de este proyecto.

A este estudio previo se le ha de añadir además la asimilación y compresión de las herramientas utilizadas. La comprensión del entorno CycleSim fue uno de los puntos más destacados, ya que no había trabajado nunca con él, ni con una herramienta similar, lo que supuso una inversión en tiempo considerable.

Para la realización de estas tareas se realizó una planificación inicial de 1 mes para obtener los conocimientos necesarios y otro más para la asimilación de la primera versión del simulador, contando 4 horas trabajadas diariamente, 5 días a la semana.

En tiempo real trabajado ha sido un poco más, teniendo en cuenta que durante la implementación fue necesario realizar consultas sobre varios temas y la asimilación completa de la segunda versión del simulador, de mayor complejidad.

La implementación del simulador se planifica con un tiempo aproximado de dos meses para el desarrollo del simulador y su debug, combinándolo con la escritura de la memoria.

Posteriormente se realizarían cambios en la pintool con la finalidad de crear unas trazas mas completas, con el reconocimiento de un número mayor de instrucciones. El tiempo para esta tarea seria variable, pues sabia como abordar este punto, por lo que se planifica entre 2 y 4 semanas combinándolos con otras tareas.

A continuación, resulta necesario un tiempo aproximado de una o dos semanas para la realización de optimizaciones en el código y diversas comprobaciones en el simulador para asegurar el correcto funcionamiento.

Debido a que finalmente la defensa del proyecto ha sido aplazada, la preparación de la presentación retrasaría hasta tener la memoria completamente finalizada. De esta manera he obtenido algo más de tiempo para poder profundizar más otros aspectos del proyecto.

CAPÍTULO 3:

ESPECIFICACIÓN

Y DISEÑO

CAPITULO 3: ESPECIFICACIÓN Y DISEÑO

En este capítulo se expone la especificación y las decisiones de diseño que se han tomado para el desarrollo del proyecto.

3.1. La CPU.

El módulo utilizado para el desarrollo de la CPU recibe el mismo nombre, "cpu" y se ha diseñado basando su estructura en estados, para simular el trabajo de un pipeline. Los estados, que son definidos mediante la librería "segmentation" que hemos generado durante el desarrollo de este proyecto, provienen a la CPU de una estructura de trabajo que permite simular el comportamiento de una segmentación de forma sencilla.

El objetivo declarado inicialmente, crear un multiprocesador superescalar en orden, induce a proponer una CPU compuesta por un pipeline específico y estático.

La decisión inicial de no utilizar un número variable de estados se debe a que no tiene sentido generar de forma dinámica el número de estados, pues el trabajo de cada uno de estos variaría de forma drástica en función de la profundidad del pipeline y también simplificamos de esta manera la estructura de la CPU.

El pipeline para este proyecto se define con profundidad 5 y consta de los siguientes estados:

1. - Fetch
2. - Decode
3. - Issue
4. - Execution
5. - Commit

Este mismo orden será el utilizado para el tratamiento de las instrucciones, que transferirán la instrucción siempre a su siguiente estado, siguiendo el esquema que hay a continuación:

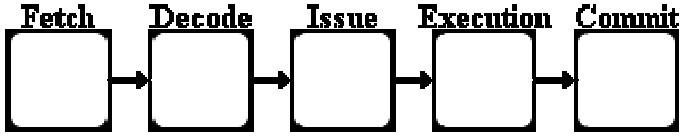


Fig. 3.1: Esquema de la comunicación entre estados del pipeline.

El siguiente esquema representa el diseño con el cual se simula el trabajo realizado por una CPU en orden de profundidad 5, que hemos utilizado para la realización de este proyecto.

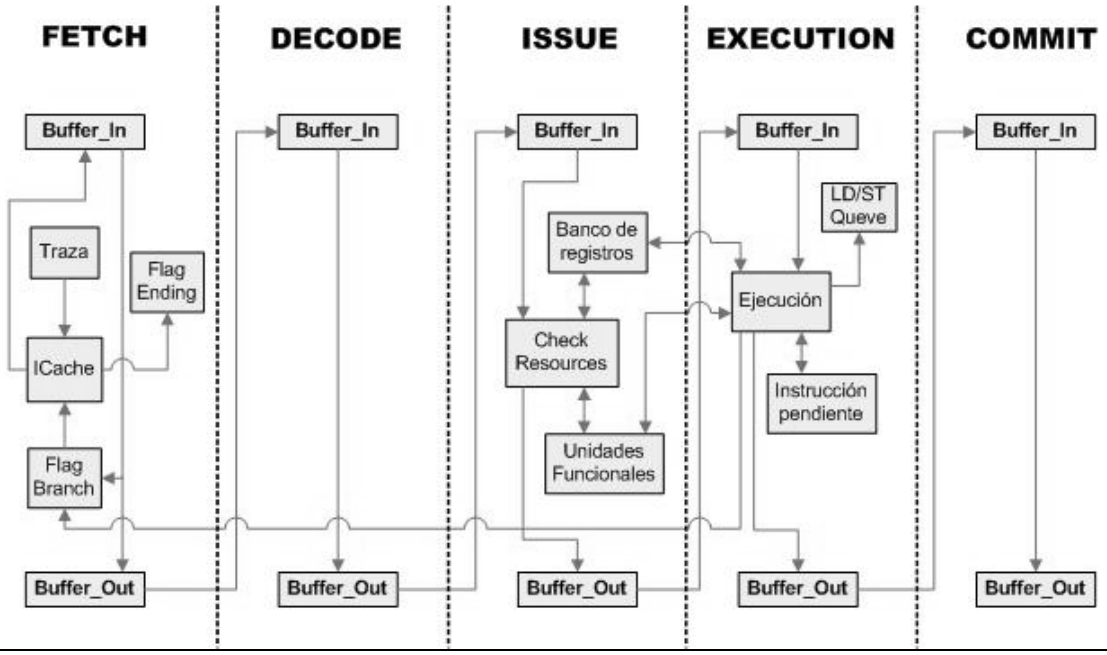


Fig. 3.2: Esquema del diseño de la CPU de este proyecto.

A simple vista podemos observar similitudes y diferencias claras en el diseño de los diferentes estados, en función del trabajo simulado por cada uno de estos. Para el diseño base de los estados se ha decido utilizar dos buffers, uno de entrada y otro de salida, para asegurar la integridad de todas las instrucciones, mediante los cuales se suministran las instrucciones de un estado al siguiente.

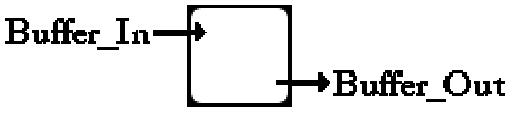


Fig. 3.3: Buffers de un estado cualquiera para este proyecto.

Supone un punto delicado la transmisión de instrucciones entre los buffers de salida hacia los buffers de entrada, ya que si no se realiza correctamente se podrían obtener resultados no deseados. Para evitar esto es necesario un sistema de control que indique a los diferentes estados si se les permite o no la transmisión de la instrucción. Recordemos para esto que la infraestructura de simulación ejecutaba la dos funciones de cada módulo por ciclo: Start() y End(). Para el diseño de esta CPU se utilizado la parte inicial del ciclo (Start) como una función de control que estudia la situación de todos los estados en el ciclo anterior e indica que estados pueden recibir nuevas

instrucciones y que otros estados quedaran bloqueados. Por otro lado, al final de la simulación del ciclo (End), se simulará la ejecución del diseño de la Figura 3.3.

El trabajo realizado para la simulación en cada estado de la segmentación será el expuesto de forma sencilla a continuación:

Fetch: En este estado la ICache comprueba si tiene instrucciones para servir, en caso negativo obtiene de la traza un número de instrucciones igual al indicado por el parámetro FetchWidth y comprueba que la traza no haya finalizado para activar flag *ending*. Por otra parte, si había instrucciones en la ICache se sirven al buffer de entrada del mismo estado, se comprueba si se trata de una instrucción de salto (para activar el flag branch) y se envía al buffer de salida.

Decode: Debido a que las instrucciones son obtenidas de forma legible, este estado no realiza un trabajo real, pero ha sido integrado para simular su efecto en la segmentación. Cuando recibe una instrucción en su buffer de entrada lo transmite a su buffer de salida.

Issue: Cuando este estado recibe una instrucción comprueba si hay disponible los recursos necesarios para su ejecución (registros y unidad funcional), comprobando las dependencias que puedan haber. En caso de no tener los recursos disponibles la instrucción será bloqueada a la espera de recursos y una vez estén disponibles, se ocuparan dichos recursos y se enviará la instrucción al buffer de salida.

Execution: Aquí se realizaran las medidas oportunas en función del tipo de instrucción recibida desde Issue y se simulará su efecto en la ejecución. Se trata de uno de los estados más complejos en su simulación que tratará de forma específica las instrucciones de acceso a memoria y las instrucciones de salto. Una vez simulada completamente la ejecución, se enviará la instrucción al buffer de salida del propio estado.

Commit: Al igual que Decode, no realiza un trabajo real en la simulación para el caso de la CPU en orden, así que se encargará de recibir las instrucciones de Execution y simular así el efecto producido en la segmentación.

A continuación se mostraran los diagramas de clase que hacen referencia a la clase "cpu" y "seg_state" (y las clases heredadas de seg_state), donde se pueden visualizar los parámetros de cada clase y sus funciones. Para más detalle ver la correspondiente sección del capitulo 4: Implementación.

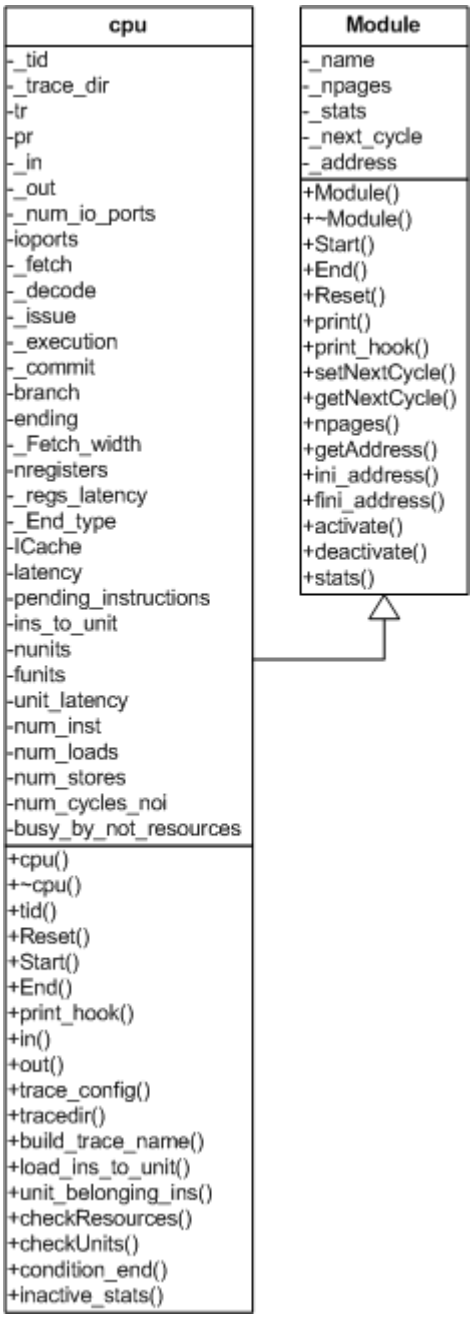


Fig. 3.4: Diagrama de clases de "cpu".

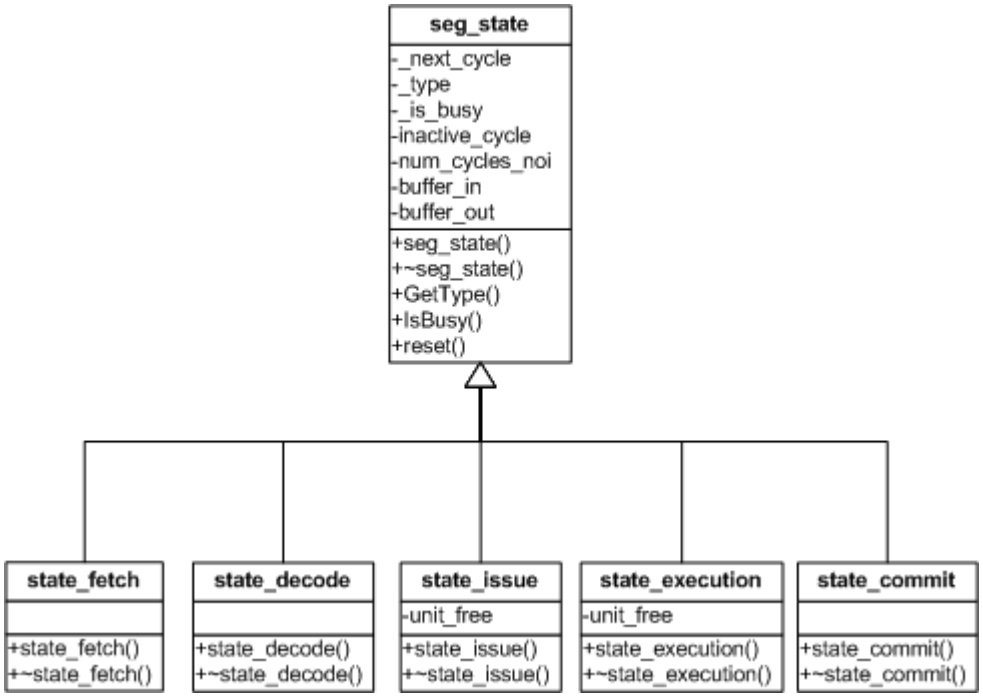


Fig. 3.5: Diagrama de clases de "seg_state" y sus clases heredadas (segmentation.cpp).

3.2. La traza.

El diseño de la traza se compone de tres tipos de archivos:

- p:prof.gz → contiene la definición ordenada de los bloques básicos (Basic Blocks) que se ejecutaran durante la simulación. Una forma de definir este archivo es como un diccionario, donde están determinados cada uno de los bloques básicos, compuestos por una serie de instrucciones y sus registros utilizados. Existe un único archivo prof al que hacen referencia todos los trace.

- t:0,p:trace.gz → el trace expone una lista ordenada de los bloques básicos (definidos en archivo prof) que se van simular en un módulo "cpu" concreto. Hay un archivo "trace" por cada thread que se va a utilizar en la configuración, lo que implica que cuando realizas una traza has de saber cuantos threads vas a configurar para la simulación.

- t:0,p:efecdir.gz → esta es la lista ordenada de direcciones efectivas, las cuales serán utilizadas por las instrucciones de acceso a memoria de un módulo "cpu". También existe un archivo por cada thread que conlleva a una relación entre los archivos "trace" y "efecdir" utilizados por el mismo thread, como podemos observar en la figura 3.5.

Se utilizan parejas de "trace" y "efecdir" para emular la ejecución de un thread. De forma ilustrativa, podríamos observar que en la ejecución simulada con 4 CPUs simultaneas se utilizará 4 archivos trace y 4 archivos efecdir, pero tan solo un único prof, pues todos los trace utilizan la misma definición de bloques básicos. En el siguiente esquema podemos visualizar este ejemplo.

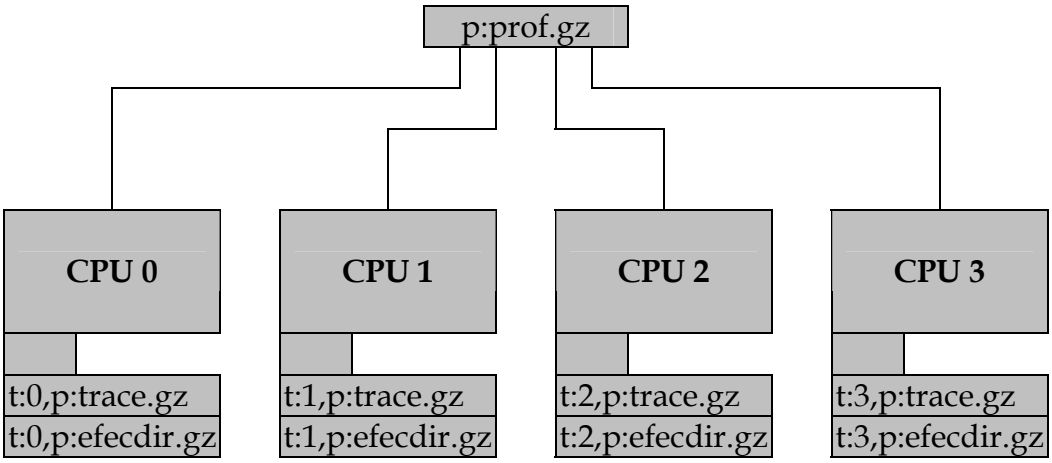


Fig. 3.6: Esquema de los archivos de traza utilizados en un ejemplo con 4 CPUs.

En la siguiente figura podemos observar parte del contenido de estos archivos:

p:prof.gz	t:0,p:trace.gz	t:0,p:efecdir.gz
1 0x7fb60bf83a50 2	1	0x7fff1419f4f8
325 0x7fb60bf83a50 1 15 1 12 N	2	0x7fff1419f4f0
1001 0x7fb60bf83a53 2 35 15 2 35 15 N	3	0x7fff1419f4e8
2 0x7fb60bf84060 21	4	0x7fff1419f4e0
1001 0x7fb60bf84060 2 14 15 1 15 N	7	0x7fff1419f4d8
325 0x7fb60bf84061 1 15 1 14 N	8	0x7fb60c1a1be0
1001 0x7fb60bf84064 2 25 15 1 15 N	7	0x7fb60c1a1fb8
1001 0x7fb60bf84066 2 24 15 1 15 N	8	0x7fb60c1a1da8
325 0x7fb60bf84068 1 12 1 24 N	9	0x7fb60c1a2528
1001 0x7fb60bf8406b 2 16 15 1 15 N	10	0x7fb60c1a1e48
680 0x7fb60bf8406c 1 15 2 15 34 N	12	0x7fb60c1a2538
617 0x7fb60bf84070 0 2 65 63 N	14	0x7fb60c1a25d8
656 0x7fb60bf84072 1 17 2 17 34 N	15	0x7fb60c1a1e58
325 0x7fb60bf84076 1 65 1 65 N	16	0x7fb60c1a2588
385 0x7fb60bf84078 2 17 19 2 17 34 N	17	0x7fb60c1a1e68
292 0x7fb60bf8407b 1 35 1 19 N	7	0x7fb60c1a27c0
1000 0x7fb60bf84082 2 19 35 2 19 34 N	8	0x7fb60c1a1e78
325 0x7fb60bf84089 1 19 1 18 N	7	0x7fb60c1a2590
1000 0x7fb60bf8408c 2 18 35 2 18 34 N	8	0x7fb60c1a1e88
1001 0x7fb60bf84093 2 35 17 0 N	7	0x7fb60c1a2598
1001 0x7fb60bf8409a 2 35 19 0 N	8	0x7fb60c1a1e98
1000 0x7fb60bf840a1 1 18 1 17 N	7	0x7fb60c1a25b8
1001 0x7fb60bf840a4 2 35 18 0 N	8	0x7fb60c1a1ea8
690 0x7fb60bf840ab 2 17 17 1 34 N	7	0x7fb60c1a25c0
1004 0x7fb60bf840ae 2 35 34 1 35 N	8	0x7fb60c1a1eb8

Fig. 3.7: Ejemplo de parte del contenido de los archivos de una traza.

En el contenido del archivo "p:prof.gz" podemos ver los dos primero bloques básicos de la traza (sus cabeceras estan en negrita). De sus cabeceras podremos ver el número identificador del bloque básico, seguido de una dirección de memoria y del número de instrucciones que componen ese bloque básico. De esta forma se diferencia que las dos líneas siguientes son dos instrucciones cuyos parámetros indican: tipo de instrucción, dirección de memoria, nº de registros fuente, los identificadores de los registros fuente, nº de registros destino, los identificadores de los registros destino y el parámetro predicated.

Las instrucciones simuladas en la traza corresponden a las instrucciones de la ISA correspondiente a la arquitectura Intel® 64 y IA-32. Como objeto de prueba se ha utilizado un ordenador con un procesador Intel Core 2 Duo T5500 1.66GHz.

Pin reconoce o agrupa en 727 instrucciones para este procesador, pero no todas las instrucciones son simuladas de forma directa en la traza. Para el objetivo del proyecto se simplifica unificando algunas instrucciones del procesador como si fueran la misma.

Este es el caso de las instrucciones que producen durante la creación de la traza un Load o un Store, o una combinación de dos Loads o un Load y un Store. De esta forma, un instrucción como un Add, en el caso concreto que necesite realizar un Load para conseguir uno de sus operandos y escribir el resultado en memoria, puede traducirse durante la traza en una instrucción de tipo LoadStore. Por otro lado, un Add cuyos operandos no necesiten realizar un acceso a memoria, será contemplado en la traza como una instrucción Add común.

También se unifican todas las instrucciones de salto como instrucciones de tipo Branch, ya que el efecto durante la simulación es el mismo. De esta forma, dos instrucciones de salto diferentes como pueden ser JB y JNL dadas por el procesador, se traducen en la traza como la misma instrucción, Branch.

El resto de instrucciones, aquellas que no realizan un acceso a memoria ni un salto, son definidas en la traza y se les relacionadas cada una de ellas con la unidad funcional que utilizan. Por ejemplo, la instrucción FSUB, Float Subtract, se relaciona a la unidad funcional de coma flotante, unidad que ocupará durante su simulación.

LOAD: instrucciones que cargan 1 operando de memoria **STORE:** instrucciones que escriben 1 operando en memoria **LOAD2:** instrucciones que cargan 2 operandos de memoria **LOADSTORE:** instrucciones que cargan 1 operando en memoria y escrito otro. **BRANCH:** cualquier instrucción capaz de producir un salto en la simulación consecutiva de instrucciones, ya sea de tipo JNZ (y similares), un JUMP, etc, que durante la generación de la traza haya sido resuelto como "salto efectivo" (es decir, solo si el salto es producido).
Fig.3.8: Instrucciones que unifican otras instrucciones debido a su comportamiento en la simulación.

CAPÍTULO 4:

IMPLEMENTACIÓN

CAPITULO 4: IMPLEMENTACIÓN

4.1. Introducción

Este capítulo profundizará sobre las técnicas de implementación que se han utilizado para el desarrollo del proyecto, obteniendo finalmente la simulación expuesta en el esquema de la figura 3.3. Inicialmente se expondrán las estructuras de datos y enumeraciones utilizadas por las clases "cpu" y la "segmentation", y posteriormente se explicaran las diferentes funciones que utilizan cada clase. De forma ilustrativa, se ha añadido al final de este capítulo un ejemplo del archivo de configuración del simulador y la salida producto de la simulación de un binario "HelloWorld".

4.2. Estructura de las clases desarrolladas

Se expondrán las estructuras de las clases "cpu" y "segmentation" para su visualización, serán explicadas posteriormente.

4.2.1. Clase "cpu"

La CPU es un módulo del simulador, por lo tanto, "cpu" es una clase heredada de la clase "Module" de la infraestructura CycleSim. Las estructuras de datos de la CPU son los siguientes:

```
class Cpu : public Module {
        int _tid;

        const std::string _trace_dir;
        const engine::config::File& _trace_config;

        std::vector< engine::port_pair< MemoryAccess * > > _in;
        std::vector< engine::port_pair< MemoryAccess * > > _out;
        std::vector< bool > ioports;

        state_fetch       _fetch;
        state_decode      _decode;
        state_issue       _issue;
        state_execution   _execution;
        state_commit      _commit;

        bool branch;
        bool ending;
        unsigned int      _Fetch_width;
        cycles_t          _regs_latency;
        unsigned int      _num_io_ports;
        unsigned int      _End_type;
        unsigned int nregisters;
```

```
        std::vector< cycles_t > registers;
        std::vector< cycles_t > latency;

        std::auto_ptr< ins::ProfReader >        pr;
        std::auto_ptr< ins::TraceReader >       tr;

        std::deque< ins::instruction_t * > ICache;

        bool _ins_pending;
        std::deque<MemoryAccess *> pending_instructions;

        int ins_to_unit [NUM_INS];
        unsigned long nunits[NUM_FUNC_UNITS_DIF];
        std::vector< std::vector< cycles_t > > funits;

        cycles_t        unit_latency[NUM_FUNC_UNITS_DIF];

        unsigned long num_inst;
        unsigned long num_loads;
        unsigned long return_loads;
        unsigned long num_stores;
        unsigned long num_cycles_noi;
        unsigned long busy_by_not_resources;
```

Fig. 4.1: Estructuras de datos de la clase "cpu".

Dentro de la librería "cpu" también podemos encontrar definida la estructura "CpuParams". Esta estructura se utiliza para la obtención de datos de configuración de la CPU. Durante la carga del simulador se genera una estructura de este tipo que se transmite posteriormente al constructor de la clase "cpu".

```
struct InsCpuParams {
        unsigned int fetch_width;
        cycles_t regs_latency;
        unsigned int num_io_ports;
        unsigned int number_units[NUM_FUNC_UNITS_DIF];
        unsigned int number_registers;
        std::deque< cycles_t > unit_latency;
};
struct CpuParams {
        ins::InsCpuParams ins_ps;
};
```

Fig. 4.2: Estructura de datos de los parámetros de la CPU.

4.2.2. Clase "segmentation"

Esta clase representa los estados de la segmentación desarrollados para este proyecto. Los estados tienen ligeras diferencias en lo que respecta a la estructura de datos, por lo que se decidió utilizar una clase base "seg_state" y otras cinco clases que representarán cada uno de los estados. La decisión de crear estas cinco clases diferentes tiene su finalidad en futuros desarrollos, ya que las diferencias entre ellas, para la implementación de la versión "en orden" son mínimas.

```
class seg_state
        cycles_t            _next_cycle;
        type_state          _type;
        bool                is_busy;
        bool                inactive_cycle;
        unsigned long       num_cycles_noi;
        ins::instruction_t * buffer_in;
        ins::instruction_t * buffer_out;
```

Fig. 4.3: Estructura de datos de la clase base de los estados de la segmentación.

De esta clase base surgen las siguientes clases heredadas:

```
class state_fetch : public cyclesim::modules::cpus::seg_state
class state_decode : public cyclesim::modules::cpus::seg_state
class state_issue : public cyclesim::modules::cpus::seg_state
class state_execution : public cyclesim::modules::cpus::seg_state
class state_commit : public cyclesim::modules::cpus::seg_state
```

Fig.4.4: Definición de las clases heredadas de "seg_state".

De hecho, las clases state_fetch, state_decode y state_commit no introducen ningún dato añadido a los datos de la clase base. Por otro lado, las clases state_issue y state_execution introducen un único nuevo dato a la estructura (las dos clases añade el mismo dato): "int unit_free".

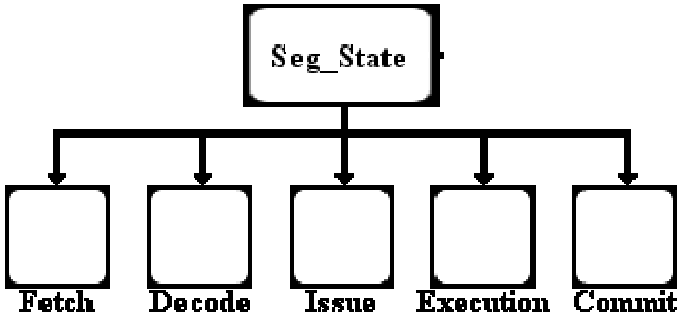


Fig. 4.5: Esquema de la herencia de clases utilizada.

4.3. Enumeraciones/definiciones

Para el desarrollo de la CPU se utilizan una serie de definiciones utilizadas en el código que hay que tener en cuenta.

```
#define NUM_FUNC_UNITS_DIF      5
#define UNIT_WHITOUT            0
#define UNIT_MEMORY_ACCESS      1
#define UNIT_INT                2
#define UNIT_FLOAT              3
#define UNIT_BRANCH             4

#define NUM_STATES_SEGMENTATION     5
#define STATE_FETCH                 0
#define STATE_DECODE                1
#define STATE_ISSUE                 2
#define STATE_EXECUTION             3
#define STATE_COMMIT                4
```

Fig. 4.6: Definiciones estados y unidades funcionales.

Existen multitud de definiciones y enumeraciones en el simulador, que procuran un correcto funcionamiento, pero al no resultar parte del desarrollo de este proyecto no resulta indispensable mencionar todos y cada uno de ellos. Por otra parte, las instrucciones definidas para este proyecto se pueden encontrar en un anexo al final de esta memoria.

4.4. Implementación

4.4.1. Librería "segmentation"

A continuación se especifican los métodos y parámetros de las clases mencionadas anteriormente, correspondientes a la librería "segmentation".

Class "seg_state"

Clase base utilizada por los cinco estados.

Métodos:

seg_state(type_state type): Constructor simple.

virtual ~seg_state(void): Destructor virtual.

type_state GetType(): Retorna parámetro _type.

bool IsBusy(): Retorna el parámetro is_busy.

void reset(): Reset de la clase.

Parámetros:

cycles_t _next_cycle: Indica el siguiente ciclo en el que deberá ejecutarse este estado. Puede que no sea necesario para todos los estados de esta implementación.

type_state _type: Identifica que estado es.

bool is_busy: Indica si el estado esta ocupado.

bool inactive_cycle: Indica si el estado ha tenido actividad durante el ciclo o si se ha mantenido ocioso.

unsigned long num_cycles_noi: Total de ciclos inactivos del estado. Este parámetro es utilizado para la obtención de estadísticas de la CPU.

ins::instruction_t * buffer_in: Instrucción de entrada al estado.

ins::instruction_t * buffer_out: Instrucción de salida del estado.

Class "state_fetch"

Clase heredada de seg_state.

Métodos:

state_fetch( ): Constructor de la clase.

~**state_fetch(void):** Destructor de la clase.

Parámetros:

Sin parámetros añadidos.

Class "state_decode"

Clase heredada de seg_state.

Métodos:

state_decode(type_state type): Constructor de la clase.

~**state_decode(void):** Destructor de la clase.

Parámetros:

Sin parámetros añadidos.

Class "state_issue"

Clase heredada de seg_state.

Métodos:

state_issue( ): Constructor de la clase.

~**state_issue(void):** Destructor de la clase.

Parámetros:

int unit_free: Parámetro de salida que informa del número de unidad funcional que estará libre para la ejecución de la instrucción lanzada a través del parámetro heredado "buffer_out".

Class "state_execution"

Clase heredada de seg_state.

Métodos:

state_execution( ): Constructor de la clase.

~**state_execution(void):** Destructor de la clase.

Parámetros:

int unit_free: Parámetro de entrada que informa del número de unidad funcional que ha de utilizar la instrucción que se encuentra en el parámetro heredado "buffer_in".

Class "state_commit"

Clase heredada de seg_state.

Métodos:

state_commit( ): Constructor de la clase.

~**state_commit(void):** Destructor de la clase.

Parámetros:

Sin parámetros añadidos.

4.4.2. Librería "cpu"

Antes de especificar la implementación de la CPU cabe recordar ciertos factores a tener en cuenta que regirán el funcionamiento de esta clase. CPU es un módulo del simulador PitufoSim y como tal, su ejecución esta controlada por éste a través de las cuatro funciones virtuales detalladas en el primer capítulo (concepto: CycleSim). Como el resto de los módulos del simulador, "cpu" posee los siguientes cuatro métodos que pueden ser llamados por el simulador y que serán los que realmente se van a ejecutar:

- void **Reset** (void)
- void **Start** (void)
- void **End** (void)
- void **print_hook** (void) const

Class "cpu"

Esta es una clase heredada de "module", para la implementación de la CPU, es necesario saber que el parámetro next_cycle de la clase "module" define el siguiente ciclo en el cual, el simulador ejecutará los métodos Start y End.

Para esta implementación se ha decidido que la CPU trabaje en todos los ciclos incrementando el parámetro next_cycle de la CPU en un ciclo, pues puede haber estados que estén bloqueados a la espera de datos y otros que puedan trabajar de forma independiente. En el caso de que todos los estados esten bloqueados, el sistema de control de la CPU impedirá que ninguno de estos estados trabaje, causando así una disminución mínima del rendimiento y simplificando considerablemente el control de la ejecución del módulo CPU. .

En el diseño de la CPU se utilizará un único objeto de cada una de las clases que representan los 5 estados de la segmentación.

A continuación se especifican los métodos y parámetros utilizados por la clase cpu.

Métodos:

Cpu(address_t, uint64_t, unsigned int, int, const CpuParams&, const std::string&, const engine::config::File&): Constructor de la clase mediante el cual se obtiene los datos de configuración de la cpu e introduce en los parámetros correspondiente de la clase.

~Cpu(void): Destructor de la clase.

int tid(void) const: Retorna el valor del parámetro "_tid".

void Reset(void): Establece los valores iniciales a todos los parámetros y estructuras de la CPU que no son constantes.

void Start(void): Realiza los controles necesarios en cada estado de la segmentación para indicar si pueden ser ejecutados durante la función End. En caso afirmativo, recogen la instrucción del buffer de salida del estado anterior y lo guardan en buffer de entrada del estado actual.

void End(void): Este método ejecuta cada uno de los estados de la segmentación que la función Start ha filtrado anteriormente. En cada uno de los estados se realizan las tareas correspondientes a ese estado para su simulación y si finalizan su ejecución, se introducirá la instrucción en el buffer de salida del propio estado.

void print_hook(void) const: extrae por pantalla los datos estadísticos.

const engine::config::File& trace_config(void) const: Retorna el valor del parámetro "_trace_config".

const std::string& trace_dir(void) const: Retorna el valor del parámetro "_trace_dir".

int unit_belonging_ins(int type): Identifica a que unidad funcional pertenece el tipo de instrucción pasado por parámetro.

bool checkResources(ins::instruction_t *): Retorna un valor booleano informando si hay recursos libres para poder lanzar la instrucción. Si tiene todos recursos libres, los ocupará.

int checkUnits(int type,cycles_t &next_c): Consulta la disponibilidad de las unidades funcionales y retorna el identificador que esta libre si lo hay, sino devolverá un número negativo.

bool condition_end(void): método que consulta si el trabajo de todos los estados ha finalizado. Se utiliza únicamente durante la fase de finalización de la CPU.

void inactive_stats(void): actualiza los valores de inactividad de cada estado.

Parámetros:

int _tid: Valor identificador de la CPU.

const std::string _trace_dir: Define el directorio de la traza que simulará la cpu.

const engine::config::File& _trace_config: Archivo de configuración de la traza señalada por _trace_dir.

std::vector< engine::port_pair< MemoryAccess * > > _in: puerto de entrada de la cpu.

std::vector< engine::port_pair< MemoryAccess * > > _out: puerto de salida de la cpu .

state_fetch _fetch: objeto que representa el estado fetch.

state_decode _decode: objeto que representa el estado decode.

state_issue _issue: objeto que representa el estado issue.

state_execution _execution: objeto que representa el estado execution.

state_commit _commit: objeto que representa el estado commit.

bool branch: identifica un estado de salto en la cpu.

bool ending: parámetro que define si la CPU a entrado en el proceso de finalización.

unsigned int _Fetch_width: define cuantas instrucciones por ciclo puede leer la cpu.

cycles_t _regs_latency: define los ciclos de la latencia por acceso a resgistros.

unsigned int _num_io_ports: número de puertos de entrada y salida de la cpu.

std::auto_ptr< ins::ProfReader > pr: objeto lector del archivo prof.

std::auto_ptr< ins::TraceReader > tr: objeto lector de la traza que ejecuta la cpu. Incluye el ProfReader como parametro.

std::deque <ins::instruction_t*> ICache: Lista de instrucciones leídas del TraceReader y listas para servir en el fetch.

unsigned int nregisters: define el número de registros de la CPU.

std::vector< cycles_t > registers: en este vector de cycles_t, se especifica a partir de que ciclo esta libre cada uno de los registros. El vector es cargado con "nregisters" registros: registers[nregisters]. El índice del vector identifica cada uno de los registros.

std::vector< cycles_t > latency: de manera similar a los registros, el vector latency[NUM_FUNC_UNITS_DIF] proporciona los ciclos de latencia que produce cada unidad funcional. El índice del vector identifica el tipo de unidad funcional cuya latencia guarda en su posición. Los tipos de unidades funcionales son definidos constantes, al igual que el parámetro NUM_FUNC_UNITS_DIF, que define el número de tipos diferentes de unidades funcionales que hay.

bool _ins_pending: informa de si existe alguna instrucción pendiente. Se utiliza para ejecutar instrucciones complejas de acceso a memoria (ejecutando 2 veces la etapa fetch), pero esta preparado para servir instrucciones complejas de otros tipo que se pudieran desarrollar.

std::deque<MemoryAccess *> pending_instructions: guarda el acceso a memoria que tendrá que hacer la segunda parte de la ejecución de instrucciones complejas de acceso a memoria (LOAD2 y LOADSTORE).

std::vector< bool > ioports: accesibilidad de los puertos de I/O.

unsigned long nunits[NUM_FUNC_UNITS_DIF]: guarda el número de unidades de cada tipo que ha sido configurado para esta cpu.

std::vector< std::vector< cycles_t > > funits: Vector de dos dimensiones que informa, de forma análoga a como trabajan los registros, de en que ciclo estará disponible una unidad funcional concreta. Para identificar una unidad funcional se define por el "tipo de unidad" y por el número de unidad de ese tipo especificado:

funits[Type][NUM_FUNC_UNITS_DIF]

Por ejemplo para saber cuando estará libre la unidad 0 de tipo Integer, accederíamos a funits[UNIT_INT][0]. Todas las unidades funcionales están ya definidas.

Parámetros estadísticos:

unsigned long num_inst: número de instrucciones ejecutadas.

unsigned long num_loads: número de "loads" realizados a memoria.

unsigned long return_loads: número de "loads" que han vuelto de la memoria.

unsigned long num_stores: número de "stores" realizados a memoria.

unsigned long num_cycles_noi: número de ciclos de inactividad de la CPU.

Funcionamiento

El trabajo de la CPU se divide en cuatro fases, de las cuales, la segunda y la tercera se repiten cíclicamente durante el transcurso de la simulación:

- Iniciación: Se encarga de la obtención previa de los datos y parámetros necesarios para la simulación e inicializa las estructuras de datos necesarias.

- Control: Decide en cada ciclo que estados de la segmentación van a ser ejecutados. Realizado por el método Start().

- Ejecución: Simula la ejecución de los diferentes estados que ha designado la fase de Control. Realizado por el método End().

- Finalización: una vez extraída la última instrucción del tracereader, se pone en marcha esta fase que procura la finalización correcta de las últimas instrucciones en cada uno de los diferentes estados.

Funcionamiento estructural de la segmentación:

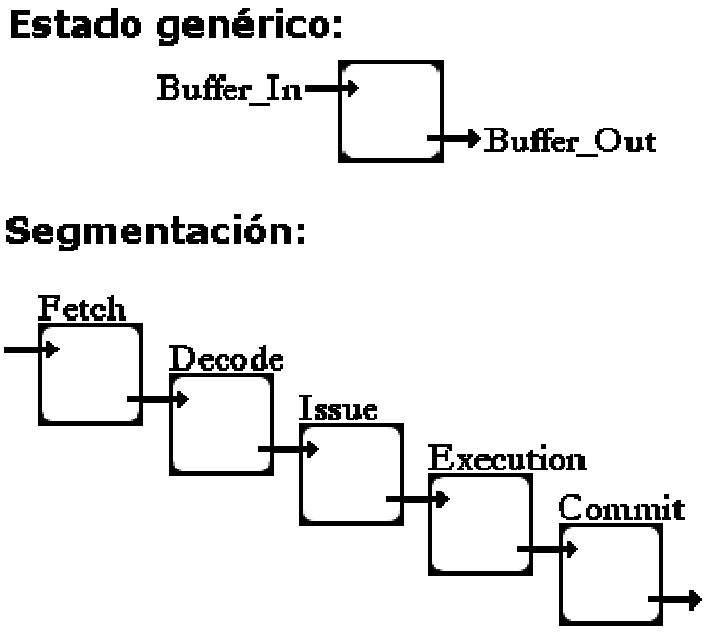


Fig. 4.7: Esquema de la segmentación. Comunicación entre estados mediante buffers de instrucciones.

Tal y como podemos observar en el esquema anterior, cada estado se comunica con el siguiente mediante el "buffer_out" del estado que ha acabado con su trabajo en la instrucción y se le da al siguiente estado, que la recibirá por su "buffer_in" (posteriormente veremos este proceso).

Como es lógico el estado Fetch no recibe instrucciones de otro estado, sino que obtiene directamente la instrucción de la ICache. De forma parecida, Commit no pasa la instrucción a ningún estado, sino que una vez simulada su ejecución, simplemente deshecha la instrucción.

Para poder asegurar la existencia única de cada una de las instrucciones, cada vez que se transmite una instrucción al siguiente estado, el estado emisor elimina de su puntero la instrucción. De esta forma, se produciría tal efecto al transmitir una instrucción ya simulada en el estado Fetch, hacia el Decode:

```
_decode.buffer_in = _fetch.buffer_out;
_fetch.buffer_out = NULL;
```

Evitando la posibilidad de que exista más de un puntero señalando una misma instrucción, se evita también de caer fácilmente en errores fatales para la simulación y asegurar la integridad del programa.

De la misma manera que el módulo CPU se ejecuta en dos fases durante un mismo ciclo (Start y End), la simulación de la segmentación queda dividida en dos fases: una de inicio de ciclo (en el Start) y otra al final del ciclo (en el End).

	Ejecución secuencial de estados →				
Inicio de ciclo (método Start)	Función: Control				
	Fetch	Decode	Issue	Execution	Commit
Fin de ciclo (método End)	Función: Simulación de ejecución				
	Fetch	Decode	Issue	Execution	Commit

Fig. 4.8: Esquema del trabajo secuencial de los estados de la CPU durante un ciclo.

Siguiendo lo explicado en el comienzo del capítulo, durante el inicio del ciclo se realiza el control de la CPU. En este caso, una CPU segmentada, por lo tanto, el control de la CPU se transforma en el control de la segmentación.

De la misma forma, la simulación de ejecución de la CPU se realiza a través de la ejecución de cada uno de los estados de la segmentación que componen la CPU.

A continuación se explica meticulosamente la implementación de los métodos de la clase "cpu". Aquellas funciones que sean triviales no serán explicadas, ya que definición anterior es apta para su comprensión.

Función cpu( )

Este método es el constructor de la clase e inicializa los datos que permanecerán estáticos durante todo el programa. También generará las estructuras necesarias para la simulación, que posteriormente se inicializaran durante el reset

El constructor utiliza la asignación por defecto de algunos parámetros de la cpu que son introducidos en su sentencia de creación. Con estos parámetros se inicializa la clase "Module" heredada, el identificador de CPU, el directorio de la traza y su archivo de configuración, se define el _FetchWidth, la latencia producida por los registros, el numero de puertos de I/O, el número de registros y la lista de latencias de las unidades funcionales.

A continuación se expondran las diferentes tareas que se realizan en el cuerpo del constructor:

Definición de los ciclos iniciales de los estados de la segmentación:

Esta inicialización esta en realidad parcialmente utilizada, ya que no todos los estados utilizan durante el control de estos el parámetro _next_cycle. Aun así, he visto apropiado la inicialización de todos de cara a futuras ampliaciones de este módulo.

```
cycles_t ini_cycle = Sim::cycle;
_fetch._next_cycle = ini_cycle;
_decode._next_cycle = ini_cycle + cycles_t(1);
_issue._next_cycle = ini_cycle + cycles_t(2);
_execution._next_cycle = ini_cycle + cycles_t(3);
_commit._next_cycle = ini_cycle + cycles_t(4);
```

Fig. 4.9: Definición de los ciclos iniciales de los estados del pipeline.

Creación de puertos de entrada/salida:

Mediante el siguiente código se crea el número de puertos configurado en el parámetro "nports" y el vector que informa de si esta ocupado o no.

```
for (unsigned int i = 0; i < nports; i++) {
        _in.push_back(engine::port_pair< MemoryAccess * >());
        _out.push_back(engine::port_pair< MemoryAccess * >());
}

for (size_t i = 0; i < _num_io_ports; i++)
        ioports.push_back(0);
```

Fig. 4.10: Código para la creación y gestión de puestos de I/O.

Cargando el número de unidades funcionales de cada tipo:

En este vector se carga el número de unidades funcionales de cada tipo, pero aun así, nótese que la unidad de acceso a memoria es posteriormente forzada a tener el mismo número de puertos de I/O, fruto de una decisión para esta implementación de las unidades funcionales.

```
for(int i=0;i<NUM_FUNC_UNITS_DIF;i++)
                nunits[i]=ps.ins_ps.number_units[i];
nunits[UNIT_MEMORY_ACCESS]=_num_io_ports;
```

Fig. 4.11: Definición del número de unidades funcionales de cada tipo.

Creación de las unidades funcionales y definición de sus latencias:

Las unidades funcionales, es un vector de 2 dimensiones, por lo que será necesario generar una dimensión correspondiente al tipo de F.U. (unidad funcional) y después la otra con el número unidades de cada tipo:

funits[type][num_units_of_type]

Se aprovecha el mismo bucle también generar e inicializar un vector con las latencias correspondientes a cada unidad funcional.

```
std::deque< cycles_t > _unit_latency = ps.ins_ps.unit_latency;
for (size_t i = 0; i < NUM_FUNC_UNITS_DIF; i++) {
        latency.push_back(_unit_latency.front());
        _unit_latency.pop_front();
        std::vector< cycles_t > tmp;
        for (size_t j = 0; j < nunits[i]; j++)
                tmp.push_back(0);
        funits.push_back(tmp);
}
```

Fig. 4.12: Código para la creación de las F.U. y definición de sus respectivas latencias.

Creación de registros

Se crea un número de registros definido por "nregisters" en um vector de elemntos de tipo "cycles_t" (ciclos).

```
for (size_t i = 0; i < nregisters; i++)
        registers.push_back(0);
```

Fig. 4.13: Código que crea nregisters registros.

Inicialización del vector ins_to_unit

El vector "ins_to_unit[instruction]" define para cada una de las instrucciones utilizadas, la unidad funcional a la que corresponde. De este modo, "ins_to_unit[ADD]" contendrá el valor "UNIT_INT". La carga se realiza mediante la función load_ins_to_unit(), que se explicará posteriormente.

```
load_ins_to_unit();
```

Fig. 4.14: Llamada a la instrucción que carga el vector ins_to_unit[ ]

Función Reset()

Inicializa/Reinicializa los datos y estructuras siguientes:

Datos estadisticos:

```
num_inst = 0;
num_loads = 0;
return_loads=0;
num_stores = 0;
num_cycles_noi = 0;
busy_by_not_resource = 0;
```

Condición de salto:

```
branch=false;
```

Fase de finalización:

```
ending=false;
```

Reset de los estados del pipeline, mediante su propia función:

```
_fetch.reset();
_decode.reset();
_issue.reset();
_execution.reset();
_commit.reset();
```

Borrado las instrucciones pendientes de servir.

```
ICache.clear();
```

Registros:

```
for(int i=0;i<nregisters;i++)
registers[i]=0;
```

Puesta a cero de todas las unidades funcionales de los diferentes tipos:

```
for(int i=0;i<NUM_FUNC_UNITS_DIF;i++)
for(unsigned int j=0;j<nunits[i];j++)
        funits[i][j]=0;
```

Reset puertos:

```
for(int i=0;i<_num_io_ports;i++) {
this->out(i).data().sendNothing();
this->out(i).busy() = false;
ioports[i]=false;
}
```

Generación del TraceReader:

```
const std::string& region ="";
pr.reset(new impl::ins::ProfReader(impl::Cpu::build_trace_name("", -1,
"prof")));
tr.reset(new impl::ins::TraceReader(*pr,
impl::Cpu::build_trace_name(region, this->tid(), "trace"),
impl::Cpu::build_trace_name(region, this->tid(), "efecdir")));
```

Borrado de instrucciones pendientes:

```
_ins_pending = false;
pending_instructions.clear();
```

Se tiene cuenta que no es necesario reinicializar aquellos parámetros que se han inicializado mediante el constructor y que no van a ser modificados de ninguna forma, por lo que no afectaría en la ejecución del programa, como por ejemplo el "_trace_dir".

Función Start()

Para realizar el control de pipeline y permitir así la simulación de cada estado en la función End(), tendremos los siguientes datos en cuenta:

1.- Si el estado actual tiene ocupado su buffer de salida. Esto indica que la siguiente instrucción no ha leído la instrucción anterior, por lo tanto, no se puede ejecutar el estado actual, ya que sobrescribiríamos la instrucción que se encuentra en espera (Para todos los estados).

2-. Si el estado actual tiene una instrucción en el buffer de entrada que todavía no ha simulado el propio estado por falta de recursos (Para todos los estados).

3.- Si existe una instrucción en el buffer de salida del anterior estado que pueda recibir el estado actual (Para todos los estados menos Fetch).

4.- Si se ha detectado una instrucción de salto ("branch") y se está simulando su efecto en el pipeline actualmente. (En el estado Fetch)

5- Si existe una instrucción pendiente para la ejecución de instrucciones complejas (Para el estado Issue).

6.- Si algún estado tiene definida su próxima ejecución para un ciclo posterior (por espera de recursos).

Como vemos hay bastantes cosas a tener en cuenta durante el control de la CPU y de forma ordenada para no inducir errores. Pero no todas estas variantes se han de tener en cuenta en todos los estados, pues algunas, debido a la implementación o significado en la simulación, no tienen sentido ser controladas.

Cada uno de los estados posee un control individualizado, pero todos han de cumplir la premisa de que su buffer de salida no contenga ninguna instrucción (buffer_out = NULL), ya que si permite la ejecución sin haber servido la instrucción anterior, esta se perdería y la simulación no seria correcta.

Tras realizar diferentes versiones del proceso de control, se ha desarrollado una versión más compacta y eficiente realizando un control de estados en orden inverso. De esta forma el proceso de control secuencial de estados se realiza en este orden: Commit >> Execution >> Issue >> Decode >> Fetch. De esta forma se conoce si el siguiente estado estará bloqueado y por lo tanto no podrá recibir la siguiente instrucción

Con esta técnica se puede controlar de forma más eficaz el efecto embudo que produce un estado del pipeline, cuando no puede realizar su trabajo por falta de recursos. Este efecto embudo es el que se pretende evitarse o aligerarse con las versiones Out-Of-Order de los procesadores.

A continuación se explica el control realizado en cada uno de los estados.

Control de los estados en la simulación:

Para poder simular el trabajo simultáneo de los diferentes estados del pipeline, era necesario encontrar la forma en que todos los estados leyesen durante el inicio del ciclo la instrucción de entrada, que provienen de los buffer_out del ciclo anterior. De esta forma, durante el final del ciclo, se podrían realizar las simulaciones necesarias dejando la nueva instrucción en el buffer_out para el siguiente ciclo.

Para poder realizar la tarea de inicio de ciclo correctamente, es necesario tener en cuenta que no podemos obtener una nueva instrucción para un estado, si el estado posterior no ha podido resolver la instrucción anterior. El siguiente esquema muestra esta situación, teniendo en cuenta que las instrucciones de la fase de inicio esta en los buffer_in y las de fin en los buffer_out que corresponden a cada estado:

Ciclo	Fase	Fetch	Decode	Issue	Execution	Commit
n	Inicio	Ins 5	Ins 4	Ins 3	Ins 2	Ins 1
	Fin	Ins 5	Ins 4	- Null -	Ins 2	Ins 1
n+1	Inicio	Ins 6	Ins 5	Ins 3	- Null -	Ins 2
	Fin	Ins 6	Ins 5	- Null -	- Null -	Ins 2

Fig. 4.15: Ejemplo de error por fallo en el control de estados. La Instrucción 4 se pierde.

En el primer ciclo vemos que estan cargados todos los buffers_in sin problemas en la fase de "inicio", pero en la fase de "fin" de este mismo ciclo representa que la etapa Issue no ha podido resolver por falta de recursos, dejando sin instrucción el buffer_out de Issue (Null). Si no tuviésemos este hecho en cuenta, durante el ciclo siguiente, se cometería el error en que los estados anteriores Fetch y Decode, continuasen simulando instrucciones y perdiendo tantas como ciclos espere la Instrucción 3 por sus recursos.

De este ejemplo obtenemos la idea de que necesitamos saber en que situación se encuentra el estado posterior y si este ha resuelto o no la instrucción anterior. En el caso de que el estado posterior no resolviera la instrucción del ciclo anterior, se procede al bloqueo del estado antes de obtener una nueva instrucción, evitando de esta manera la perdida de instrucciones.

Para realizar este concepto, se ha creado un control de estados en sentido inverso. Mediante este método el estado que se esta comprobando en el momento actual conoce como se ha comportado el estado siguiente, ya que se ha comprobado antes. En la siguiente figura podemos comprobar como se comportaría este método de control frente al error mostrado anteriormente.

En la figura 4.16, a diferencia de la figra anterior, no muestra la posición de las instrucciones en cada estado, sinó la situación en la que se encuentran los diferentes estados (Control: Free/Block) y si se ha completado el trabajo de ese estado en la simulación de la instrucción en cada uno de los estados (Resolución: YES/NO).

Ciclo	Fase	Commit	Execution	Issue	Decode	Fetch
n	Inicio	free	free	Free	free	free
	Fin	YES	YES	NO	YES	YES
n+1	Inicio	free	free	BLOCK	BLOCK	BLOCK
	Fin	YES	NO	NO	NO	NO

Fig. 4.16: Comportamiento del sistema de control inverso en el anterior ejemplo.

Podemos apreciar que el estado Issue no resuelve en el primer ciclo ya que no posee recursos para lanzar la instrucción, pero el resto de estados del ciclo si ha resuelto satisfactoriamente sus tareas. Esto provocará un bloqueo en el estado Issue que se propagará en el siguiente ciclo y además no podrá ofrecer la instrucción al estado Execution, que quedará inactivo en el siguiente ciclo. En el siguiente ciclo se comprueba la fase de Control de la siguiente manera:

Commit: no tiene problemas, ejecuta con normalidad.

Execution: ya que Commit leyó el buffer_out de Execution en su ciclo anteior, no se bloquea (aunque no recibe instrucción nueva por parte de Issue).

Issue: no resolvió en el ciclo anterior, por lo que tiene ocupado su buffer_in, por lo tanto se bloquea en espera de recursos y no resuelve.

Decode: Issue no ha leído el buffer_out de decode, por lo tanto no puede realizar ningún trabajo y se bloquea.

Fetch: por el efecto embudo queda bloqueado ya que Decode no pudo leer la instrucción del buffer_out de Fetch, por lo tanto, no trabaja y se bloquea.

A continuación explicaré como realiza el control cada uno de los estados de forma independiente. Por el trabajo que desempeñan cada uno de los estados del pipeline, las razones por las cuales pueden llegar a bloquearse un estado varian.

Ya que la secuencia de control es realizada de forma invertida, explicaré cada una de las fases de control en el orden el cual son ejecutadas dentro del método Start(), con el consecuente orden: Commit, Execution, Issue, Decode y Fetch.

1.- Commit

El control para la etapa Commit es trivial, ya que jamás produce un estado de ocupación para la versión In-Order del simulador. Esto afecta directamente sobre el siguiente estado ya que jamás se verá afectado por el efecto embudo. De esta forma el único trabajo que realiza este control es la lectura del buffer de salida de Execution y la anulación de éste.

```
_commit.buffer_in=_execution.buffer_out;
_execution.buffer_out = NULL;
```

Fig. 4.17: Código de control de Commit.

Asignando un puntero nulo al buffer_out del estado Execution, informamos al estado Execution de que la instrucción que sirvió en el ciclo anterior ha sido leída y por lo tanto esta libre de poder simular la ejecución de una nueva instrucción en este ciclo.

2.- Execution

Lo primero que debería consultar este estado antes de hacer cualquier otra comprobación es averiguar si Commit ha leído la instrucción del ciclo anterior, ubicada en buffer_out. Este resulta un caso especial, pues se puede apreciar que el código que realiza esta tarea esta "comentado" y por lo tanto no se ejecutará. Esto se debe a que el estado Commit nunca entrara en un estado de bloqueo como comentamos, por lo tanto no tiene sentido comprobar un parámetro que no va a variar.

```
//if (_execution.buffer_out == NULL){
        if (!_ins_pending){
                if (_execution.buffer_in==NULL  && _issue.buffer_out!=NULL){
                        _execution.buffer_in=_issue.buffer_out;
                        _issue.buffer_out=NULL;
                        _execution.unit_free=_issue.unit_free;
                        _execution.is_busy=false;
                }
        }else{
                if (_execution._next_cycle >= Sim::cycle) _execution.is_busy=false;
        }
//}else{
        //_execution.is_busy=true;
//}
```

Fig. 4.18: Código de control de Execution.

Las características específicas de control de este estado se rigen en un primer orden por la simulación del ciclo anterior, en función de si la instrucción simulada era o no una instrucción compleja de acceso a memoria (LOAD2 o LOADSTORE), que produciría una instrucción pendiente de acceso a memoria (_ins_pending=true).

Si no hay instrucción pendiente y el estado Issue tiene una instrucción para servir, se procede a la lectura de los datos necesarios para la simulación del estado Execution.

En el caso de que exista instrucción pendiente, se consultará el parámetro _next_cycle, el cual señala el ciclo en el cual la instrucción pendiente tendrá los recursos necesario para terminar de simular su ejecución.

3.- Issue

Para realizar la correcta simulación de este estado, uno de los más problemáticos de la simulación, procedemos a comprobar si el estado Execution ha leído la anterior instrucción lanzada.

```
if (_issue.buffer_out==NULL){
        if(Sim::cycle >= _issue._next_cycle){
                if (_issue.buffer_in==NULL && _decode.buffer_out != NULL){
                        _issue.buffer_in=_decode.buffer_out;
                        _decode.buffer_out=NULL;
                }
                _issue.is_busy=false;
        }
}else{
        _issue.is_busy=true;
}
```

Fig. 4.19: Código de control de Issue.

Si esta libre el buffer de salida, se procede a comprobar si el estado tiene una previsión de lanzamiento posterior por falta de recursos, mediante el parámetro _next_cycle del estado. Esto evita recalcular innecesariamente la existencia de recursos durante la etapa de ejecución de Issue, ya que se conoce el ciclo en el cual habrá recursos disponibles.

Si esto se cumple y además no existe una instrucción parada en el buffer de entrada por lanza, se comprueba si existe una instrucción en el buffer de salida del estado Decode por leer. Si hay se procede a su lectura y eliminación del buffer_out de decode.

4.- Decode

El código de control de este estado sigue las reglas básicas de la estructura de control que se ha planteado. Comprueba si tiene libre el buffer de salida y si es así verifica que no tener una instrucción por simular en su buffer_in y si existe instrucción en el buffer de salida del estado Fetch para leer.

Si se cumplen los requisitos lee la instrucción de Fetch y borra su buffer_out, evitando duplicación de instrucciones.

```
if (_decode.buffer_out==NULL){
        if (_decode.buffer_in == NULL && _fetch.buffer_out!=NULL){
                _decode.buffer_in=_fetch.buffer_out;
                _fetch.buffer_out=NULL;
                _decode.is_busy=false;
        } else _decode.is_busy=true;
}else{
        _decode.is_busy=true;
}
```

Fig. 4.20: Código de control de Decode.

5.- Fetch

Debido al trabajo realizado por este estado, su control es reducido únicamente a la comprobación de su buffer de salida. Si el estado Decode ha leído la instrucción servida en el ciclo anterior, se procede a la resolución del estado Fetch.

```
if (_fetch.buffer_out==NULL){
        _fetch.is_busy=false;
}else{
        _fetch.is_busy=true;
}
```

Fig. 4.21: Código de control de Fetch.

Función End()

Esta función ejerce la mayor parte del trabajo de la simulación de la CPU. Para la explicación de su implementación dividiré la sección con los diferentes estados de forma análoga a la explicación del Start(), en su orden de ejecución.

Como base para la simulación de cada uno de los estados de la segmentación, se realiza una comprobación de su ocupación. Si el estado se encuentra ocupado, no ejecutará el código perteneciente al trabajo del ese estado.

Este parámetro (is_busy) pude modificarse tanto en la función Start() como en End(), pero únicamente podrá poner el valor "false" la función encarga del control de los diferentes estados (Start( )).

1.- Fetch

El trabajo de la simulación realizada en este estado se divide en:

- Procurar alimentar de instrucciones la ICache cuando este vacía.
- Controlar la finalización del trabajo de la CPU si la traza ya no contiene más instrucciones.
- Servir la siguiente instrucción de la ICache a través del buffer_out.
- Comprobar si la función que se va a servir es de tipo "branch", para comenzar el proceso del efecto salto en la segmentación.

Además de la consulta sobre si está ocupado o no el Fetch, este estado realiza de forma adicional dos comprobaciones más para su ejecución, debido a sus funcionalidades extra de implementación: la finalización y el efecto salto (o efecto "branch").

```
if((!_fetch.IsBusy() && !branch) || (!_fetch.IsBusy() && ending))
```

Fig. 4.22: Código de control de los flags "branch" y "ending".

El efecto "branch" surge cuando una instrucción de salto se resuelve como efectivo. Esto provoca (en la realidad, no en esta simulación, explicaré el porqué) que las instrucciones de los estados anteriores sean invalidados, ya que fueron servidos de forma especulativa antes de la decisión del salto.

En caso de que la instrucción de salto de la instrucción hubiera sido negativa, las instrucciones de los estados anteriores serian correctos y no habría provocado ninguna irregularidad en el transcurso del pipeline. Existen técnicas

de predicción de saltos, pero esta versión de la CPU no incluye ninguno, así que se implementa el resultado por defecto.

Veamos un ejemplo sobre el resultado producido:

Ciclos	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Ins 1 (Instr. Branch)	F	D	I	E	C										
Ins 2		F	D	I	E	C									
Ins 3			F	D	I	E	C								
Ins 4				F	D	I	E	C							
Ins 5					F	D	I	E	C						

Fig. 4.23: Efecto branch en el pipeline.

En la figura anterior podemos observar que la instrucción 1, es una instrucción de salto y durante su etapa Execution (en el ciclo 4), se resuelve de forma afirmativa. Pero cuando es resuelto ya tiene instrucciones en el pipeline, que serian las que tendrían que ejecutarse de haber sido fallida la condición de salto (especulación).

Por lo tanto, la CPU debería ignorar los resultados de las tres instrucciones empezadas prematuramente y continuar con la ejecución resuelta por el salto con la instrucción 5 (ciclo 5). Se perdería entonces las instrucciones realizadas en los estados anteriores a Execution.

En la simulación, debemos tomar como afirmativos todos los saltos que nos da la traza (los saltos no realizados no se dan en la traza) y simular el efecto que se produce en el pipeline: un retardo de 3 ciclos en servir la siguiente instrucción.

Concretamente, lo que esta implementación realiza para simular este efecto es consultar en el FETCH si la instrucción es de salto, en caso afirmativo activamos el flag "branch" que impide la ejecución del Fetch y por lo tanto impide también el servicio de instrucciones. Cuando la instrucción de salto llega a la etapa Execution, como resultado de su ejecución baja el flag "branch" y se reanuda el servicio de instrucciones produciendo la simulación deseada:

Ciclos	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Instr. Branch)	F	D	I	E	C										
Next Instr.					F	D	I	E	C						

Fig. 4.24: Efecto branch en el pipeline simulado.

Vemos que de esta forma, tras encontrar en el Fetch una instrucción de salto, se retrasa el siguiente Fetch hasta después del estado Execution de la instrucción de salto.

Tras este inciso sobre el "efecto branch" retomamos las condiciones de ejecución del estado fetch con su otra funcionalidad: finalización.

Cuando la ICache se vacía y realizamos el proceso de "recarga" de Fetch_Width instrucciones en la ICache, finalmente le llegará una instrucción NULL del TraceReader. Esto significa que la traza ha finalizado y no hay más instrucciones por servir mas las queden en la ICache.

Cuando esto ocurre se levanta el flag "ending", que en será consultado en todos los ciclos una vez se haya servido la última instrucción de la ICache.

```
if (ending){
        if( condition_end()){
                next_cycle=this->NEVER;
                return;
        }
}
```

Fig. 4.25: Sección de código que controla la finalización de la CPU.

El método "condition_end()" será explicado posteriormente, pero como su nombre indica, consulta si ha finalizado toda la simulación de la CPU. Para finalizar este módulo, se pondrá el valor especial NEVER como siguiente ciclo y se retornará al simulador.

La función principal del Fetch es la de obtener las instrucciones y servirlas al siguiente estado. Para ello utilizaremos una ICache (ficticia, ya que no se comporta como una cache) donde se colocaran las instrucciones obtenidas a través del TraceReader. Para alimentar la ICache de instrucciones se realiza el siguiente código:

```
if (ICache.size()<=0 && !ending){
        for(int i=0;i<_Fetch_width;i++) {
                _fetch.buffer_in = tr->getNextInstruction();
                if (_fetch.buffer_in != NULL) {
                        ICache.push_back(_fetch.buffer_in);
                        num_inst++;
                }else {
                        ending = true;
                        break;
                }
        }
}
```

Fig. 4.26: Sección de código de la obtención de instrucciones.

En negrita la captura de la instrucción del TraceReader.

Finalmente para enviar al siguiente estado la instrucción correspondiente en la simulación, se realizará el siguiente código:

```
if (ICache.size()>0) {
        _fetch.buffer_in = ICache.front();
        ICache.pop_front();
        if ((unit_belonging_ins(_fetch.buffer_in->type)) == UNIT_BRANCH) {
                branch = true;
        }
        _fetch.buffer_out=_fetch.buffer_in;
}
```

Fig. 4.27: Sección de código del servicio de instrucción al Decode y verificación de Branch.

Se puede apreciar en estas líneas que se aprovecha el servicio de la instrucción para verificar si ésta es o no una instrucción de salto, mediante el uso de la función "unit_belonging_ins". En caso afirmativo, se activa el flag de branch, comenzando el proceso que simulará este evento.

2.- Decode

Decode no realiza ningún trabajo extra en la simulación, no obstante, el mantener una instrucción en su buffer ya causa el efecto deseado sobre la simulación:

```
if (_decode.buffer_in!=NULL){
        _decode.buffer_out=_decode.buffer_in;
        _decode.buffer_in=NULL;
}
```

Fig. 4.28: Código de servició de instrucción a Issue.

Cuando su ejecución es permitida por el control del Start(), pone la instrucción en su buffer de salida para que lo recoja el estado Issue.

3.- Issue

Este estado se encarga de lanzar la instrucción recibida cuando existen los recursos necesarios para su ejecución:

```
if(!checkResources(_issue.buffer_in)) {
        _issue.buffer_out=NULL
        _issue.is_busy=true;
}else{
        _issue.buffer_out=_issue.buffer_in;
        _issue.buffer_in=NULL;
}
```

Fig. 4.29:Sección de código de la resolución del estado Issue.

El trabajo se realiza realmente en la función "checkResources" (se entrará posteriomente en detalle con esta función), donde se comprueba el estado de los recursos necesarios para la ejecución. En caso de no tenerlos, Issue se muestra como ocupada hasta el ciclo "_issue._next_cycle", que es definido durante la ejecución de "checkResources".

Posteriormente explicaré la función "checkResources", mostrando el método utilizado para la consulta y solicitud de recursos, que simulará la ejecución de una instrucción.

4.- Execution

Aquí se realiza la simulación de ejecución de instrucciones, pero cabe recordar que no se simularan más que los accesos a memorias e instrucciones que por su naturaleza modifican el comportamiento del pipeline.

El estado se divide en dos partes, en función de si hay o no una instrucción pendiente (instrucción compleja). Si hay instrucción pendiente será ejecutada, en caso contrario se realiza la ejecución normal de la instrucción que proviene del buffer de entrada del estado.

```
if (_ins_pending){
        //Ejecución de la instrucción pendiente
}else if (_execution.buffer_in != NULL){
        //Ejecución de la instrucción de entrada
}
```

Fig. 4.30: Esquema estructural del estado.

En el caso general, se atenderá la instrucción de entrada. Para ello, se utiliza un "Switch-Case" que diferencia el tipo de instrucción:

```
switch(_execution.buffer_in->type) {
        case LOAD: {  //Código que trata el LOAD
                }
        case LOAD2: { //Código que trata el LOAD2
                }
        case LOADSTORE: { //Código que trata el LOADSTORE
                }
        case STORE: { //Código que trata el STORE
                }
        default:{ //Código que trata el caso "default"
                }
}
```

Fig. 4.31: Esquema del caso genérico de ejecución de instrucciones.

Observando este esqueleto del "Switch-Case" utilizado podemos observar que distinguimos las instrucciones de accesos a memoria del resto. El tratamiento de estos es muy similar: obtenemos una dirección del archivo "efectdir", obtenemos un Memory Access en función de si se trata de una lectura o una escritura y lo enviamos por un puerto de salida de la CPU.

Las instrucciones LOAD2 y LOADSTORE generan 2 accesos consecutivos y será necesario realizarlo en dos ejecuciones, por lo se realiza el primer acceso, se obtiene otra dirección y se activa el proceso de instrucción pendiente. Veamos el caso LOAD a modo de ejemplo:

```
case LOAD: {
        address_t next_address = tr->getNextAddress();
        if (next_address == 0) {
                printf ("LD Next address = 0\n");
        }else{
                ma = memory_access_pool.getLoad(next_address, 8,
                        this->getAddress()+address_t (
                        _execution.buffer_in->targets[0]));
                num_loads++;
                ioports[0]=true;
                this->out(_execution.unit_free).data() = ma;
        }
        break;
}
```

Fig. 4.32: Código que trata la instrucción de tipo Load en el estado Execution.

El resto de accesos son similares, teniendo en cuenta que el memory access de las escrituras se realiza con getStore en vez de con getLoad.

¿Qué ocurre con el resto de instrucciones? El caso "default" es analizado para buscar instrucciones de salto cuando el flag "branch" está activado por el estado Fetch. Si lo está y ha llegado la instrucción de salto al estado Execution se baja el flag:

```
default:        {
        if (branch && ((unit_belonging_ins(
                _execution.buffer_in->type)) == UNIT_BRANCH)){
                branch = false;
        }
        break;
}
```

Fig. 4.33: Código de tratamiento de las instrucciones que no son de acceso a memoria.

El resto de instrucciones son tratadas de forma genérica ya que no influyen de forma anormal en el transcurso del pipeline. Estas instrucciones ocuparan los registros que utilicen y la unidad funcional asociada a cada instrucción para su simulación.

Saliendo del caso regular, si hay instrucción pendiente por resolver será necesario volver a solicitar recursos para su ejecución. En este caso, los registros ya han sido "bloqueados", por lo que necesitará únicamente una unidad libre:

```
if (_ins_pending){
        ma = pending_instructions.front();
        cycles_t c(0);
        int unit =checkUnits(UNIT_MEMORY_ACCESS,c);
        if ( (unit != -1) && !ioports[0]) {
            if (!this->in(unit).busy()) {
                pending_instructions.pop_front();
                if (typeid(*ma) == typeid(MemoryAccessLoad)) num_loads++;
                else num_stores++;
                ioports[0]=true;
                this->out(unit).data() = ma;
                _ins_pending=false;
                for(int j=0;j<_execution.buffer_in->target_number;j++)
                        registers[_execution.buffer_in->targets[j]]= Sim::cycle +
                                latency[UNIT_MEMORY_ACCESS] + _regs_latency;
                if (_issue._next_cycle == this->NEVER){
                        _issue._next_cycle = Sim::cycle +
                                latency[UNIT_MEMORY_ACCESS] + _regs_latency;
```

```
                }
            }
        }else{
                _execution.is_busy=true;
                _execution._next_cycle = c;
        }
}
```

Fig. 4.34: Código que trata la segunda parte de una instrucción compleja.

Mediante el método "checkUnits" se solicita una unidad libre del tipo acceso a memoria, si no está libre, se definirá el ciclo en el que estará libre esa unidad con el ciclo de la variable "c". Definido este ciclo "c" como el siguiente en ser ejecutado el estado Execution, se pondrá el estado en modo ocupado a la espera de que llegue dicho ciclo.

En caso contrario, si hay una unidad libre se procederá igual que un acceso normal y se desbloquearan los registros utilizados y el estado _issue.

Finalmente, si no hay instrucciones pendientes (para asegurar que acaba su simulación), se pone la instrucción en el buffer de salida.

5.- Commit

Al igual que Decode, no realiza ningún trabajo de simulación, por lo que si recibe una instrucción, su función es ponerla a NULL.

```
if (_commit.buffer_in!=NULL) {
        _commit.buffer_out = _commit.buffer_in;
        _commit.buffer_in=NULL;
}
```

Fig. 4.35: Código de la simulación del estado Commit.

Esto, aunque pueda parecer un trabajo inútil en lo que a la simulación se refiere, tiene impacto sobre la condición de finalización, produciendo que hasta que buffer no sea NULL, aunque exista una instrucción en el pipeline y no se finalice la CPU, que es el comportamiento deseado.

La asignación del buffer de entrada al buffer de salida tampoco realiza un trabajo real, pero resulta muy útil para observar el comportamiento de los estados en modo debug y evidentemente el compilador descartará esta asignación cuando se compila sin información de debug.

Función print_hook()

Esta función extrae por pantalla los resultados de la CPU obtenidos por la simulación. Adicionalmente es utilizada para el desarrollo aumentando el número de estadísticas extraídas para el análisis del correcto funcionamiento.

La función es llamada una única vez por el simulador tras la finalización completa de la simulación.

Los datos que nos interesan para la simulación son:

```
printf("    Instructions: %lu\n", num_inst);
printf("    Loads(go-back): %lu--%lu\n", num_loads,return_loads);
printf("    Stores: %lu\n", num_stores);
printf("    Cycles no issue-> CPU: %lu\n", num_cycles_noi);
printf("    Locks due to resource: %lu\n", busy_by_not_resource);
printf("    >> Info States of pipeline <<\n");
printf("    Cycles no issue-> Fetch: %lu\n", _fetch.num_cycles_noi);
printf("    Cycles no issue-> Decode: %lu\n", _decode.num_cycles_noi);
printf("    Cycles no issue-> Issue: %lu\n", _issue.num_cycles_noi);
printf("    Cycles no issue-> Execution: %lu\n", _execution.num_cycles_noi);
printf("    Cycles no issue-> Commit: %lu\n", _commit.num_cycles_noi);
```

Fig. 4.36: Código de la información extraida de la simulación de la CPU.

Función load_ins_to_unit()

Este método se encarga de definir que unidad funcional será asignada a cada una de las instrucciones. De esta manera, cuando una instrucción vaya a ser lanzada durante la etapa Issue del pipeline, consultará que tipo de unidad funcional utilizara la instrucción, ocupando dicho recurso.

Actualmente, la traza facilita un total 727 instrucciones diferentes, más los 5 tipos filtrados (4 de acceso a memoria y 1 de salto) que afectan de forma específica en la simulación. Estos últimos tipos de instrucción no son valores incluidos en la definición de este vector, ya que son previamente filtrados y no tiene sentido añadirlos. Se define este vector de la siguiente manera, siendo NUM_INS el número de instrucciones que contempla este vector (727):

int ins_to_unit[NUM_INS];

En una primera instancia se inicializa el vector con el valor que corresponde a la unidad funcional UNIT_WHITOUT (sin unidad específica), que se toma por defecto. Esta unidad funcional representa una unidad genérica que engloba a las instrucciones simples que no necesitan utilizar una unidad funcional específica y su ejecución se resuelve en un ciclo, como puede ser un movimiento de registros.

```
for (int i=0; i < NUM_INS; i++)
        ins_to_unit[i] = UNIT_WHITOUT;
```

Fig. 4.37: Inicialización del vector ins_to_unit.

A continuación es necesario definir las instrucciones que corresponden a las unidades funciones de enteros y de cómo flotante. Esto realiza de forma manual, mediante la consulta individual de cada una de las instrucciones en el manual "Intel® 64 and IA-32 Architectures. Software Developer's Manual" en su sección "Instruction Set Reference".

```
…
ins_to_unit[INST_INC]=UNIT_INT;
ins_to_unit[INST_MUL]=UNIT_INT;
ins_to_unit[INST_NEG]=UNIT_INT;
ins_to_unit[INST_OR]=UNIT_INT;
for (int i=INST_PABSB; i<= INST_PADDW; i++) ins_to_unit[i] = UNIT_INT;
for (int i=INST_PAND; i<= INST_PANDN; i++) ins_to_unit[i] = UNIT_INT;
for (int i=INST_PAVGB; i<= INST_PAVGW; i++) ins_to_unit[i] = UNIT_INT;
…
```

Fig. 4.38: Sección del código de la definición del vector "ins_to_unit[ ]".

En el código anterior podemos observar una parte de las definiciones que especifican la Unidad de Enteros (UNIT_INT).

Los identificadores de las instrucciones del procesador que nos facilita Pin vienen ordenados de forma alfabética por la descripción corta de la instrucción, por ejemplo "MUL". Esto resulta favorable ya que muchos identificadores consecutivos hacen referencia a la misma unidad funcional, por ejemplo la mayoría de instrucciones que comienzan por Fxxx utilizan la unidad de coma flotante. Aprovecharemos este formato para realizar bucles y agilizar la definición de las instrucciones como podemos ver en código anterior, evitando de de esta forma el tener que realizar estas definiciones una por una (puede consultarse la lista de instrucciones enteras de forma ordenada en el anexo II).

Función unit_belonging_ins()

Este método se encarga de identificar a que unidad funcional pertenece una instrucción específica, con la finalidad de averiguar que tipo unidad funcional va a necesitar ocupar dicha instrucción.

Se trata de una sencilla función con un "Switch-Case" determinado por el tipo de instrucción que se esta examinado. En primer lugar se filtra si la instrucción en de acceso a memoria o de salto, en caso contrario, retorna el valor almacenado en el vector "ins_to_unit[type]", donde "type" corresponde al tipo de instrucción que deseamos consultar.

```
switch(type) {
        case LOAD: return UNIT_MEMORY_ACCESS;
        case STORE: return UNIT_MEMORY_ACCESS;
        case LOAD2: return UNIT_MEMORY_ACCESS;
        case LOADSTORE: return UNIT_MEMORY_ACCESS;
        case BRANCH: return UNIT_BRANCH;
        default: return ins_to_unit[type];
}
```

Fig. 4.39: Código que determina la unidad funcional que utiliza una instrucción.

El vector "ins_to_unit[type]", como se ha comentado previamente, tiene almacenados los tipos de unidad funcional, utilizando los diferentes tipos de instrucción como índice. Las instrucciones filtradas previamente son casos especiales no recogidos por este vector para facilitar la generación de la traza.

Función checkUnits()

Dado el tipo de unidad funcional utilizada, retorna la primera unidad libre de ese tipo. En caso de no encontrar unidad libre de ese tipo, se retorna un "-1" como unidad y se define el parámetro "next_c" (que se para por referencia) el primer ciclo en el que habrá una unidad libre de ese tipo.

Se diferencia entre el caso de acceso a memoria y el resto en esta implementación, debido a que el acceso memoria se comprueba por los puertos libres y el resto por las unidades funcionales.

```
int _unit=-1;
if(type == UNIT_MEMORY_ACCESS) {
        next_c=this->NEVER;
        for(unsigned int i=0;i<nunits[type];i++) {
            if(!this->in(i).busy()) {
                if(funits[type][i]<=Sim::cycle) {
                        _unit=i;
                        break;
                }else{
                        next_c=(next_c<funits[type][i])?next_c:funits[type][i];
                }
            }
        }
}else{
        next_c=funits[type][0];
        for(unsigned int i=0;i<nunits[type];i++) {
                if(funits[type][i]<=Sim::cycle) {
                        _unit=i;
                        break;
                }else{
                        next_c=(next_c<funits[type][i])?next_c:funits[type][i];
                }
        }
}
return _unit;
```

Fig. 4.40: Código que consulta si hay disponible alguna unidad funcional de un tipo concreto.

Como se puede ver el funcionamiento es bastante simple y se retorna siempre el ciclo de la unidad funcional que estará libre en primer lugar (en caso de que no haya ninguna libre).

Función checkResources()

Esta función se encarga de la consulta de recursos para decidir si una instrucción puede ser lanzada desde la etapa Issue, controlando las dependecias entre instrucciones.

Existen varios tipos de dependencias que hay que tener en cuenta para que las instrucciones sean ejecutadas/simuladas correctamente:

1.- Dependencias estructurales: este tipo de dependencia hace referencia a las unidades funcionales y se produce cuando dos o más instrucciones necesitan la misma unidad funcional y no hay un número suficiente de esta para ejecutar todas las instrucciones.

2.- Dependencias de datos: se produce cuando dos o más instrucciones necesitan acceder a un mismo registro o posición de memoria. Existen tres tipos diferentes de dependencias de memoria:

- Read after write (RAW): se produce cuando una instrucción necesita un dato cuyo cálculo esta en proceso y por lo tanto no esta disponible. Si se ejecutase la instrucción sin que el cálculo de la isntrucción anterior estuviese finalizado se estaría utilizando un dato incorrecto y el programa no produciría el resultado deseado.

r1 ← r2 + r3
r4 ← r1 + 16

- Write after read (WAR): se produce cuando una instrucción realiza una operación con un dato de entrada que va a ser modificado posteriormente. Si antes de utilizar este dato se finalizase la instrucción posterior, este dato habría sido modificado cambiando el comportamiento del programa.

r1 ← r2 + r3
r2 ← r4 + r5

- Write after write (WAW): se produce cuando dos o más instrucciones necesitan escribir en la misma memoria/registro, necesitando posponer la escritura de la última instrucción hasta que finalicen las anteriores.

r1 ← r2 + r3
r1 ← r4 + 8

Las dependencias WAR y WAW también son conocidas como falsas dependencias y pueden ser evitadas mediantes un renombramiento de registros estático o dinámico.

Los recursos que debe comprobar la función "checkResources" son:

1.- Unidad funcional libre
2.- Registros fuente
3.- Registros destino

La estructura de la función sigue el esqueleto siguiente:

```
bool ready=true;

//Consulta de Unidad Funcional

//Consulta de Registros Fuente

//Consulta de Registros Destino

//Ocupar recursos si están todos libres

return ready;
```

Fig. 4.41: Estructura del método checkResources().

En caso de que ninguna de las consultas de recursos falle, el valor de ready permanecerá como "true". A continuación se explica en detalle.

En primer lugar se comprueba si esta libre la unidad funcional que utiliza la instrucción. Para realizar esto utiliza dos métodos previamente explicados: "unit_belonging_ins" y "checkUnits".

```
int type = unit_belonging_ins(ins->type);
cycles_t next_c(0);
int _unit = checkUnits(type,next_c);
if (_unit==-1) {
        if(next_c>Sim::cycle)_issue._next_cycle = next_c;
        ready = false;
}
```

Fig. 4.42: Código que verifica si hay una unidad funcional libre del tipo solicitado.

Seguidamente será necesario comprobar si los registros fuente están listos, pues podrían estar "bloqueados" (usándolos) por una instrucción anterior.

```
for(int i=0;i<ins->source_number;i++) {
        if(registers[ins->sources[i]]>Sim::cycle) {
                if((_issue._next_cycle <= registers[ins->sources[i]]) &&
                        (registers[ins->targets[i]]!=this->NEVER)) {
                                _issue._next_cycle=registers[ins->sources[i]];
                }
                ready = false;
        }
}
```

Fig. 4.43: Consulta de disponibilidad de los registros fuente.

Y como última consulta de recursos se comprueba que los registros destino no estén en uso, evitando así un error fatal.

```
for(int i=0;i<ins->target_number;i++){
        if (registers[ins->targets[i]]>Sim::cycle){
                ready = false;
                if((_issue._next_cycle <= registers[ins->targets[i]]) &&
                        (registers[ins->targets[i]]!=this->NEVER)) {
                                _issue._next_cycle=registers[ins->targets[i]];
                }
        }
}
```

Fig. 4.44: Consulta de disponibilidad de los registros destino.

Finalmente, si todos los recursos solicitados están disponibles, se procede a ocupar los registros y la unidad para lanzar la instrucción al estado Execution.

```
if (ready==true){

        for(int i=0;i<ins->source_number;i++)
                registers[ins->sources[i]]= Sim::cycle + _regs_latency;

        if ((ins->type == LOAD2) ||(ins->type == LOADSTORE)){
                for(int j=0;j<ins->target_number;j++){
                        registers[ins->targets[j]]= this->NEVER;
                }
        }else{
```

```
                for(int j=0;j<ins->target_number;j++)
                        registers[ins->targets[j]]= Sim::cycle +
                                latency[type] + _regs_latency;
        }

        funits[type][_unit]= Sim::cycle + latency[type];
        _issue.unit_free=_unit;
}
```

Fig. 4.45: Código para la ocupación de los recursos necesarios para una instrucción.

Cabe todavía realizar un comentario, sobre el texto marcado en negrita en el código. Para poder evitar problemas con los recursos de las instrucciones complejas, ocupamos estos registros con el ciclo especial NEVER y posteriormente, cuando se ejecute el segundo acceso a memoria de estas instrucciones, se desbloquearan estos registros con la latencia pertinente.

Función condition_end()

El método "condition_end()" se utiliza durante el proceso de finalización de la CPU. Para esta implementación y una vez comenzado el proceso de finalización, la CPU habrá acabado con la simulación de todas las instrucciones cuando "todos" los buffers de instrucciones de los estados de la CPU estén vacíos.

Si nos fijamos en el código siguiente veremos que hay dos excepciones, correspondientes a los extremos de la segmentación:

- "_fetch.buffer_in"
- "_commit.buffer_out"

Esto se debe a que la ICache no sirve nunca una instrucción nula en el "_fetch.buffer_in" y a que "_commit.buffer_out" jamás llega a albergar ninguna instrucción para esta implementación, pues carece de sentido para la simulación.

```
return (_decode.buffer_in==NULL) && (_issue.buffer_in==NULL) &&
        (_execution.buffer_in==NULL) && (_commit.buffer_in==NULL) &&
        (_fetch.buffer_out==NULL)  && (_decode.buffer_out==NULL) &&
        (_issue.buffer_out==NULL) && (_execution.buffer_out==NULL);
```

Fig. .4.46: Código que consulta si puede finalizar la simulación de la CPU.

La utilidad como función es principalmente hacer el código más legible.

Función inactive_stats()

Como su nombre indica, se encarga de comprobar la inactividad de los diferentes estados para que cuando finalice la simulación se pueden mostrar en las estadísticas.

Este método se lanza al finalizar el ciclo en la CPU, es decir, al final de la función End(), una vez los estados ya han realizado el trabajo pertinente.

```
int i=0;

if(_fetch.inactive_cycle) {
        _fetch.num_cycles_noi++;
        i++;
}
if(_decode.inactive_cycle) {
        _decode.num_cycles_noi++;
        i++;
}
if(_issue.inactive_cycle) {
        _issue.num_cycles_noi++;
        i++;
}
if(_execution.inactive_cycle) {
        _execution.num_cycles_noi++;
        i++;
}
if(_commit.inactive_cycle) {
        _commit.num_cycles_noi++;
        i++;
}

if (i>=NUM_STATES_SEGMENTATION) num_cycles_noi++;
```

Fig 4.47: Código para el tratamiento de las estadísticas de inactividad.

En primer lugar se comprueba cada uno de los estados para ver si tienen activado el flag de inactividad y en caso afirmativo se suma a los ciclos inactivos de este estado.

Los flags de inactividad se controlan durante el método End() y se levantan si el estado esta ocupado o si no ha recibido una instrucción de .

Finalmente, se recuentan los estados inactivos del ciclo actual y si todos los estados se declaran inactivos se incrementa en 1 la inactividad de la CPU.

4.4.3. Configuración y salida del simulador

La información de salida del simulador nos muestra los módulos utilizados por la simulación y extrae los resultados estadísticos que buscamos para este proyecto.

Como ejemplo utilizaré la siguiente configuración para realizar una simulación del binario correspondiente a un programa HelloWorld.

```
Content-Type: application/X-cyclesim-config; version="1"


sim.page_size = 4096
trace = /home/chus/trunk/src/build/src/simulator/PitufoSim/helloworld

bus.requests_limit = 1

processor.n = 1
processor.0.lm.address = 0x1000000000000000
processor.0.lm.latency = 10
processor.0.lm.size = 64
processor.0.cache.line_size = 128
processor.0.dma.queue_size = 16
processor.0.tlb.size = 128
processor.0.ins.fetch_width = 4
processor.0.ins.regs_latency = 1
processor.0.ins.num_io_ports = 1
processor.0.ins.func_units_whitout = 100
processor.0.ins.func_units_memory_access = 2
processor.0.ins.func_units_int = 2
processor.0.ins.func_units_float = 2
processor.0.ins.func_units_branch = 1
processor.0.ins.number_registers = 1024
processor.0.ins.end_type = 0
processor.0.ins.lat_unit_whitout = 1
processor.0.ins.lat_unit_memory_access = 2
processor.0.ins.lat_unit_int = 2
processor.0.ins.lat_unit_float = 6
processor.0.ins.lat_unit_branch = 4

memory.n = 1
memory.0.latency = 150
memory.0.dram.data_rate = 800
memory.0.dram.burst = 8
memory.0.dram.cas = 5
memory.0.dram.dqss = 1
memory.0.dram.cwl = 5
memory.0.dram.ras = 15
memory.0.dram.rc = 20
memory.0.dram.rcd = 5
memory.0.dram.rp = 5
memory.0.dram.wr = 6
memory.0.dram.wtr = 3
memory.0.dram.vdd = 1.975
memory.0.dram.idd0 = 65
memory.0.dram.idd2n = 50
memory.0.dram.idd3n = 50
memory.0.dram.idd4w = 130
memory.0.dram.idd4r = 130
memory.0.dram.idd5 = 200
memory.0.dram.rfcmin = 110
memory.0.dram.refi = 7800
memory.0.dram.pdq_rd = 3.2
memory.0.dram.pdq_wr = 5.6
memory.0.dram.pdq_rd_oth = 24.9
memory.0.dram.pdq_wr_oth = 20.8
memory.0.mic.unified_queue = true
```

```
memory.0.mic.autoprecharge = 2
memory.0.mic.n_drams = 4
memory.0.mic.n_banks = 8
memory.0.mic.page_size = 8
memory.0.mic.interleave = 0
```

Fig. 4.48: Configuración del simulador (pitufosim.cfg):

Utilizando la anterior configuración el resultado obtenido tras la ejecución del programa fue la siguiente. Hay que tener en cuenta que este resultado se obtuvo utilizando la versión Release del simulador, que realiza múltiples optimizaciones en su compilación.

```
CycleSim Simulator Infrastructure 1.99 - pitufosim frontend
Copyright (c) 2007-2009 Barcelona Supercomputing Center
Copyright (c) 2007-2009 Universitat Politecnica de Catalunya

Reseting modules...
Start of Simulation: #CPU:1-#Mem:1
Memory map:
                  OS[0]: 0x0000000000000000-0xffffffffffffffff, pages 0
         Syncronizer[0]: 0x0000000000000000-0x0000000000000fff, pages 1
                 Mic[0]: 0x0000000000000000-0xbfffffffffffffff, pages 3377699720527872
                 Nic[0]: 0xc000000000000000-0xc000000000000fff, pages 1
                 CPU[0]: 0xc000000000001000-0xc000000000001fff, pages 1
    MemoryController[0]: 0xc000000000002000-0xc000000000002fff, pages 1
                 Nic[1]: 0xc000000000003000-0xc000000000003fff, pages 1
          PerfectMic[0]: 0xc000000000004000-0xc000000000043fff, pages 64
                 Bus[0]: 0xc000000000044000-0xc000000000044fff, pages 1
Running simulation...
Sim::Run end
End of Simulation.
Printing statistics...
sim_cycle = 4210009
sim_warp = 1120.00266033
sim_time = 5s
sim_time = 5751824114ns
sim_speed = 841979 cycles/s
sim_warp_speed = 842001 cycles/s
*** OS[0]
OS Started
*** Syncronizer[0]
*** Mic[0]
    Data sent: 2452392 bytes
    Loads: 306549 Stores 109454
    Busy cycles: 0
    PageTable Accesses: 213
    Pending Accesses: 0
    Power breakdown by modules:
    *** DRAM 0
      Number of loads: 306549
      Number of stores: 109454
      Background power: 98.75 mW
      Activate power: 58.55 mW
      Write power: 32.86 mW
      Read power: 92.04 mW
      IOTerminationPower: 3.03 mW
      Refresh power: 4.18 mW
      Total power: 289.40 mW
    Total power:
      Number of loads: 306549
      Number of stores: 109454
      Background power: 98.75 mW
      Activate power: 58.55 mW
      Write power: 32.86 mW
      Read power: 92.04 mW
      IOTerminationPower: 3.03 mW
      Refresh power: 4.18 mW
      Total power: 289.40 mW
```

```
*** Nic[0]
    busy_cycles: 0
    data_sent: 2452392
    num_rejects: 2563
    num_requests: 306549
*** CPU[0]
    Instructions: 1277473
    Loads(go-back): 306336--0
    Stores: 109454
    Cycles no issue-> CPU: 1087031
    Locks due to resource: 730627
    >> Info States of pipeline <<
    Cycles no issue-> Fetch: 2932386
    Cycles no issue-> Decode: 2932390
    Cycles no issue-> Issue: 2201763
    Cycles no issue-> Execution: 2895284
    Cycles no issue-> Commit: 2932390
*** MemoryController[0]
    Busy cycles: 0
    Loads sent: 306549
    Stores sent: 109454
    Loads Received: 306549
*** TLB:
    TLBmisses 213
    TLBhalfmisses 1000
    TLBld 213 TLBldback 213
    MCldrequests 306336 MCstrequests 109454
    Pending TLB_MSHR 0
    Pending MC Output 0
*** Nic[1]
    busy_cycles: 0
    data_sent: 3328024
    num_rejects: 15246
    num_requests: 416003
*** PerfectMic[0]
    Data sent: 0 bytes
    Loads: 0 Stores 0
    Busy cycles: 0
    PageTable Accesses: 0
*** Bus[0]
    port_rejects: [2563,15246]
    port_requests: [309112,431249]
```

Fig. 4.49: Salida de la simulación de un binario HelloWorld con la configuración anterior.

CAPÍTULO 5:

PINTOOL Y LA CREACIÓN DE TRAZAS

CAPÍTULO 5: PINTOOL Y LA CREACIÓN DE TRAZAS

5.1. Introducción

La herramienta Pin (www.pintool.com), como ya había introducción previamente, se trata de un instrumentador dinámico que nos permite insertar código, previamente a la ejecución de un binario.

Fig. 5.1: Logotipo de PinTool.

El objetivo de usar esta herramienta en este proyecto es la de generar unos ficheros de traza que representarán la ejecución del binario utilizado. Mediante esta traza, se podrá simular la ejecución de dicho binario en el simulador PitufoSim que utiliza este proyecto.

La pintool utilizada para generar esta traza utiliza otras librerías suplementarias, como la librería de instrucciones. Esta librería se encarga de realizar un filtro durante la ejecución de cada instrucción, con la finalidad de averiguar la instrucción que se está ejecutando y poder representarla en la traza. Esta es la base de PlainTracer.

Podemos encontrar el código referente a estas librerías en la localización de archivos que indica seguidamente, refiriéndose a los archivos adjuntados a este proyecto:

PlainTracer: /trunk/src/tracers/plain/plain-tracer.cpp Instrucciones: /trunk/src/tracers/common/instruction.cpp
Fig. 5.2: Ubicación de los archivos relacionados con la "pintool".

5.2. Reconocimiento de instrucciones

Uno de los objetivos de este proyecto era el reconocimiento del número máximo de instrucciones posibles en el simulador. Para enfocar este objetivo era necesario saber de que premisa se partía, es decir, cuantas instrucciones había representadas en la versión inicial y su papel en la simulación.

Al inicio de este proyecto se representaban 5 instrucciones, 4 de acceso a memoria y una que representaba los saltos realizados. El siguiente paso era averiguar que repertorio de instrucciones me ofrecía el procesador que estaba utilizando.

Para ello, busqué en el manual online de Pin las funciones que me permitía utilizar esta herramienta. Mientras indagaba resultaba fácil darse cuenta de que Pin era una herramienta muy potente y encontré los métodos que me ayudarían a resolver el problema.

INS_Opcode(ins): Dada una instrucción (ins), retorna un entero con número que representa el opcode de la instrucción ins.

OPCODE_StringShort(opcode): dado un entero (opcode) que representa el opcode de una instrucción, retorna un std::string con una descripción corta, como por ejemplo "ADD" o "XOR".

A continuación era necesario obtener una lista de instrucciones y sus opcodes para poder utilizarlo en el simulador, para esto generé un código en la función "main" del tracer-reader.cpp, con el cual obtendría la lista de instrucciones que buscaba. Además aprovecharía este código para generar el formato de salida idóneo para introducir en el código del simulador.

```
std::string name_inst;
std::string no_inst = OPCODE_StringShort(3000); //Opcode innexistente
for (int i=0; i<5000; i++){
      name_inst = OPCODE_StringShort(i);
      if (name_inst == no_inst) break;  //No hay más Opcodes
      else {
            std::cout << "#define INST_" << name_inst;
            std::cout << "   " << i <<"\n";
      }
}
```

Fig. 5.3: Código que genera el listado de instrucciones que trabaja el procesador utilizado.

Para entender como trabaja este código es necesario saber que Pin nos va a ofrecer un número de instrucciones menor de 1000, para un procesador como el utilizado para este proyecto o similar. Por esta razón, el código inicialmente guarda en la variable "no_inst" el valor de la descripción de una instrucción que no existe (3000), para posteriormente realizar un bucle en el cual se obtienen las instrucciones empezando por la primera, hasta que encuentre una instrucción inexistente (nunca llegara a 5000).

El código de la figura 5.3 obtiene de forma ordena una lista de todas las instrucciones del procesador y los extrae por la salida estándar, por lo tanto, para obtener una nueva lista de instrucciones tan solo es necesario utilizar el siguiente comando:

"./pin -t {localización del plain-tracer} -- /bin/ls >> ins_list.hpp"

Con este comando generaremos la traza que representa la ejecución de un "ls" y redirigimos los datos extraídos por la salida estándar al archivo especificado. De esta forma obtendremos la lista de instrucciones deseada en el archivo "ins_list.hpp", solo tendremos que purgar del final del archivo el resultado del "ls" que también es añadido.

Ahora únicamente tendremos que sustituir este archivo en la carpeta "/trunk/src/modules/cpus/ins/" y compilar el simulador para que este reconozca el nuevo juego de instrucciones, si es que utilizamos un procesador con una ISA diferente.

Nota: El código anterior encargado de extraer esta lista de instrucciones permanece comentado en la función "main" para no causar molestia durante la generación de nuevas trazas.

5.3. Utilización del PinTool

Para poder crear trazas nuevas, la herramienta Pin tan solo necesita un "pintool", en el caso que corresponde a este proyecto el "plain-tracer" y un binario que será objeto de simulación.

Siguiendo el orden lógico en primer lugar necesitamos generar el "pintool" que será inyectado en la ejecución del binario, sino esta creado ya. Para hacerlo debemos crear enlace entre la PinTool y el automake, para que al hacer "make" se compile el "plain-tracer".

Para esto es necesario que desde la carpeta "build" del proyecto ejecutemos el siguiente comando expuesto a continuación.

Carpeta Build: "/trunk/src/build/" Comando: "../../configure --enable-debug=yes PIN_KIT={ubicación del Pin}" Tracer compilado: /trunk/src/build/src/tracer/plain/.libs/libplain-tracer.so
Fig. 5.4: Enlazado de la PinTool con Automake.

Ahora cuando realicemos un "make" desde la carpeta "build" se generará el plain-tracer con la extensión ".so". El "tracer" quedará ubicado en la carpeta previamente señalada como "Tracer compilado".

Una vez tenemos el "plain-tracer" compilado tan solo se ha de ejecutar el siguiente comando para generar una traza nueva desde la ubicación del PinTool.

Comando: "./pin -t {ubicación del plain-tracer} -- {ubicación del binario}"
Fig. 5.5: Comando para generar una nueva traza.

En la ubicación actual se creará un directorio llamado "trace/" que contendrá la traza. Si el directorio ya existe la PinTool extraerá un mensaje advirtiendo que no puede crear el directorio y detendrá la ejecución, por lo que es necesario renombrar eliminar el directorio "trace/" si existe previamente.

Como se puede apreciar, la creación de nuevas trazas, una vez ya se dispone del tracer compilado es una tarea sencilla, óptima para realizar las pruebas que nos sean necesarias.

CAPÍTULO 6:

ESTUDIO ECONÓMICO

CAPÍTULO 6: ESTUDIO ECONÓMICO

Para llevar a cabo este estudio, se anotaron las horas transcurridas al final de cada sesión de trabajo (horas aproximadas) y el trabajo concreto que se realizó en esa sesión, desde el comienzo del proyecto. De esta forma se han obtenido los siguientes datos:

Tarea	Porcentaje	Horas dedicadas
Estudio previo	16.30 %	70.5 horas
Instalación	3.82 %	16.5 horas
Estudio del código inicial	9.94 %	43.0 horas
Diseño y especificación	12.02 %	52.0 horas
Implementación	21.85 %	94.5 horas
Depuración	17.23 %	74.5 horas
Memoria	12.02 %	52.0 horas
Presentación	6.82 %	29.5 horas
Total de horas dedicadas al proyecto		432.5 horas

Fig. 6.1: Lista de horas utilizadas para el desarrollo de este proyecto.

Se ha realizado una diferencia distición de tareas para agrupar el trabajo realizado desde el comienzo. El "estudio previo" contempla todas las tareas de busqueda de información y el aprendizaje del material didáctico obtenido. La "instalación" hace referencia al tiempo empleado para la puesta a punto de todo el software necesitado para este proyecto y todas sus versiones. El "estudio del código inicial" engloba las horas dedicadas a estudiar el código del simulador (las dos versiones iniciales) y la pintool, para la correcta comprensión del código. La tarea de "diseño y especificación" hace referencia a las horas dedicadas a diseñar las nuevas funciones incorporadas al simulador y a la pintool. La "implementación" y la "depuración" engloban el trabajo realizado para crear el simulador diseñado y la comprovación de su correcto funcionamiento. Por último, la "memoria" y la "presentación" representan las horas de trabajo que se han utilizado para el desarrollo de esta memoria y la preparación de la presentación de este proyecto.

Teniendo en cuenta la figura 6.1, contemplamos que se dedica a este proyecto dos personas: un analista y un programador. Distribuimos las tareas anteriormente comentadas entre ellos según su función en el proyecyo y definimos un coste por hora de trabajado según los datos obtenidos en:

http://www.oficinaempleo.com/empresa/puestos/analista.htm
http://www.oficinaempleo.com/empresa/puestos/programador.htm

De esta información extraemos los siguientes datos económicos respecto a estas dos personas:

	Analista	Programador
Salario anual	36.000 €/año	30.000 €/año
Salario mensual	3.000 €/mes	2.500 €/mes
Salario por hora	18.75 €/hora	15,63 €/hora

Fig. 6.2: Salario en función del puesto de trabajo.

A continuación, utilizando los datos del salario por hora de los dos puestos de trabajo y las asignaciones de las tareas de la figura 6.1, obtendremos los costes de personal generados por el proyecto:

Tarea	Asignación	Coste
Estudio previo	Analista	1.321,88 €
Instalación	Analista	309,38 €
Estudio del código inicial	Programador	672,09 €
Diseño y especificación	Analista	975,00 €
Implementación	Programador	1.477,04 €
Depuración	Programador	1.164,44 €
Memoria	Programador	812,76 €
Presentación	Analista	553,13 €
Coste total de horas dedicadas al proyecto		**7.285,70 €**

Fig. 6.3: Coste humano del proyecto.

Finalmente añadiremos el hardware utilizado para el desarrollo del proyecto que se trata de un portatil SONY VGN-C1Z valorado en 1.600 €. Teniendo en cuenta que el software utilizado es libre, obtendremos los costes finales del proyecto:

	Coste
Trabajo humano	7.285,70 €
Hardware	1.600,00 €
Coste total del proyecto	**8.885.70 €**

Fig. 6.4: Coste total del proyecto.

CAPÍTULO 7:

CONCLUSIONES Y

LÍNIAS DE FUTURO

CAPITULO 7: CONCLUSIONES Y LÍNEAS DE FUTURO

7.1. Conclusiones

Durante el desarrollo de este proyecto se han obtenido todos los objetivos que se definieron desde su comienzo. A continuación se expondrán estos objetivos y se realizará un breve comentario al respecto.

1.- Programar un multiprocesador superescalar in-order para el entorno de simulación CycleSim. Se ha conseguido con éxito la programación de una CPU in-order que simula la ejecución de las instrucciones recibidas, mediante la programación de un pipeline de profundidad cinco. Con esta CPU, y el simulador PitufoSim, se obtiene un multiprocesador superescalar in-order.

2.- Mejora de las funcionalidades de la CPU. Se han añadido mejoras en la CPU como la inclusión de un pipeline simulado, lectura de varias instrucciones por ciclo (Fetch_Width), creación de unidades funcionales y el control recursos sobre estas, reconocimiento de todas las instrucciones, etc.

3.- Mejorar la lectura de trazas. Se han mejorado las trazas mediante el reconocimiento de todas las instrucciones disponibles por el procesador. El simulador puede reconocer cada una de las diferentes instrucciones gracias a la generación de una lista de instrucciones que incluye todas las instrucciones que proporciona Pin para el procesador utilizado en proyecto. También se incluyen en la memoria los datos necesarios para utilizar un procesador que utilice un conjunto de instrucciones diferente al utilizado para el desarrollo de este proyecto.

4.- Simular todas las etapas del pipeline de un procesados superescalar. Para cumplir este objetivo se ha programado un pipeline de profundidad cinco, simulando una CPU In-Order. Para esto se ha diseño una librería nueva (segmentation), con la cual se puede representar los diferentes estados. Para poder simular este punto se ha diseñado un sistema de control del pipeline con una gran eficiencia, con el fin de evitar un colapso durante la simulación.

5.- Controlar las dependencias de datos entre instrucciones. Objetivo imprescindible para la correcta simulación de la CPU. Para ello, se generan unidades funcionales y se comprueba durante la etapa Issue (lanzamiento), que la instrucción que se va lanzar tiene una unidad funcional libre del tipo que utiliza esa instrucción en concreto. En caso de que este libre se consulta si los registros que esta utilizando no están siendo utilizados por alguna instrucción lanzada previamente. En caso de que se dispongan de todos los recursos necesarios, se ocupan estos recursos para que las siguientes instrucciones no los utilicen y se pasa la instrucción al estado Execution.

Con este sistema se controla a la perfección las dependencias de datos, cumpliendo con este objetivo.

6.- Controlar la disponibilidad de las diferentes unidades funcionales. El número de unidades funcionales de cada tipo se puede configurar a través del archivo de configuración "pitufosim.cfg". De esta forma se puede consultar la disponibilidad de la unidad funcional que va a utilizar la instrucción que va a ser lanzada. Si alguna de estas unidades funcionales está libre se procede a su ocupación (si el resto de recursos de la instrucción están libres). La ocupación de una unidad funcional se realiza con otro parámetro configurable, la latencia de las diferentes unidades funcionales. Existe un parámetro para definir la latencia de cada una de las diferentes unidades funcionales, ya que esta latencia no será la misma para una unidad de enteros que para una de floats.

Mediante este sistema se ha conseguido programar esta disponibilidad de las unidades funcionales.

7.- Ampliación de las opciones configurables de la CPU. Se han ampliado las opciones configurables: el número de diferentes unidades funcionales de cada tipo y las definiciones de sus correspondientes latencias.

Aunque ya estaba incluida el parámetro "fetch_width" se ha procedido a la utilización de este parámetro, ya que en la versión inicial no se utilizaba.

8.- Reconocimiento del máximo número posibles de instrucciones. Para poder abarcar este objetivo se han desarrollado cambios tanto en el programa que genera la traza como en el simulador. Fue necesario realizar un estudio de la PinTool para poder entender las posibilidades que presentaba y opté por reconocer todas las instrucciones que me podía ofrecer el procesador. De esta forma, el simulador ha pasado de reconocer 5 instrucciones diferentes, a reconocer 727 instrucciones más las 5 que son tratadas de forma especial. Por lo tanto considero que este objetivo se ha cumplido satisfactoriamente.

7.2. Líneas de futuro

Esta es una pequeña guía para el desarrollo de determinados aspectos del módulo "cpu" para futuros cambios.

A continuación he generado una lista de posibles mejoras a implementar en esta CPU en futuras actualizaciones. Hay casos sencillos y otros más complejos que podrían requerir un estudio previo:

1.- Reconocimiento de un repertorio de instrucciones (ISA) distinto.

2.- Mejoras de rendimiento en el control de la segmentación. Este punto es crítico para el buen funcionamiento en largas simulaciones. Se han probado diferentes sistemas de control hasta llegar al sistema actual (control invertido de estados) que ya posee un rendimiento satisfactorio, aunque podría estudiarse otro método en búsqueda de una mayor eficiencia.

3.- Inserción de un modelo Out-Of-Order a partir de los estados creados, modificando los estados Issue (para introducir una cola de lanzamiento) y Commit (para controlar la finalización en orden).

4.- Aumento de los parámetros configurables del módulo de la cpu.

5.- Aumento de las estadísticas obtenidas.

6.- Mejoras de rendimiento de la etapa Execution eliminando el concepto de instrucciones pendientes y utilizando el _execution._next_cycle para la demora de ciclos.

8.- Modificación en el pipeline que permita configurar el número de estados con la trabaja la CPU, permitiendo escoger la profundidad de la segmentación en el pipeline. Utilizando etapas clave que realicen el trabajo (como por ejemplo las cinco etapas elaboradas en este proyecto) y utilizando etapas "relleno" genéricas (entre los estados anteriores) que únicamente traspasen las instrucciones a través del buffer. Se obtiene de esta manera la simulación deseada por la profundidad del pipeline sin la realización de trabajo específico por parte de los estados dinámicos de la simulación.

En las siguientes páginas se mostrará como realizar algunas mejoras de las que he mencionado anteriormente.

Reconocimiento de un repertorio de instrucciones diferente (ISA)

Actualmente el simulador reconoce todas las instrucciones del procesador utilizado para el desarrollo del simulador, el T5500 de Intel®. Este posee una ISA de 727 instrucciones (incluyendo la instrucción 0: INVALID).

Para poder generar trazas con un equipo que posea un procesador con una ISA diferente, seria necesario actualizar esta lista de instrucciones. Nótese que he especificado que tan solo seria necesario si se quisiera generar nuevas trazas, ya que el simulador puede utilizarse en cualquier equipo sin importar su procesador, pues si la traza ya estuviese creada, el simulador tendría que mantener la misma configuración de instrucciones para su correcto funcionamiento.

En el capítulo referente a PinTool se comenta como generar una nueva lista, pero faltaría advertir al desarrollador que debería modificar los valores de las instrucciones filtradas específicamente (de acceso a memoria y de salto), por otros valores más elevados, en el caso de que el repertorio de instrucciones llegase a la 1000 instrucciones, pues crearía un conflicto entre instrucciones. Esto modificación de valores se tendría que hacer en el "tracer-plain" y en el simulador.

Teniendo esto en cuenta, el "tracer-plain" generará automáticamente una traza correcta con la nueva ISA sin necesidad de realizar ningún cambio. Por otra parte, para que el simulador interprete correctamente las instrucciones, deberá actualizar la lista de instrucciones que posee actualmente.

Para generar la nueva lista, solo hay que quitar los comentarios ("/*", "*/") que ejecutan el código que genera la lista, compilar nuevamente el "tracer-plain" y simular cualquier binario (un "/bin/ls" por ejemplo). Con el archivos "ins_list.hpp" creado y purgado, se substituye por el que está utiliza el simulador.

A continuación se ha de modificar el valor del número máximo de instrucciones que reconocerá el simulador, en el archivo "cpu.hpp" y generar la relación de las nuevas instrucciones con la unidad funcional que van ha utilizar.

En caso de que alguna instrucción realice una acción que influya en el comportamiento común del pipeline deberá de programarse en consecuencia dentro de la función "End()" de la librería "cpu.cpp".

Introducción de nuevos parámetros configurables en la "cpu"

La creación de nuevos parámetros configurables es crítica para los objetivos del simulador, por lo que resulta interesante tener un referente para generar nuevos parámetros configurables. Dado que este proyecto se centra en la "cpu", explicaré como introducir parámetros para la CPU, aunque no existirá gran diferencia al resto de módulos.

Paso 1: Para realizar esto utilizaremos la estructura que recibe la CPU para su configuración, la cual había comentado previamente, y añadiremos una nueva variable o estructura que deseemos hacer configurable, según nuestros objetivos.

```
struct InsCpuParams {
        unsigned int fetch_width;
        cycles_t regs_latency;
        unsigned int num_io_ports;
        unsigned int number_units[NUM_FUNC_UNITS_DIF];
        unsigned int number_registers;
        std::deque< cycles_t > unit_latency;
        Type NewVariable;
};

struct CpuParams {
        ins::InsCpuParams ins_ps;
};
```

Fig. 7.1: Estructura de los parámetros configurables de la CPU.

Si lo que queremos es configurar un grupo de variables del mismo tipo, es aconsejable utilizar una estructura tipo <vector> o <deque>, como en el caso de "unit_latency", que configura las latencias de las unidades funcionales.

Paso 2: Una vez pongamos los datos en la nueva variable ya estará disponible para ser utilizada desde el módulo "cpu". Para cargar las variables utilizaremos una aplicación común de configuración llamada "config.cpp" que podemos encontrar en el directorio: "/trunk/src/simulator/common".

Aquí editaremos el método que se encarga de la carga de datos configurables de la CPU de instrucciones:

read_ins_cpu_params(const config::File& config, const int module)

Si nos fijamos veremos que lo primero que hace es crear una estructura del tipo "InsCpuParams", donde cargará los datos leídos.

Veamos dos ejemplos de lectura de datos que realiza actualmente para poder partir de aquí:

```
ps.fetch_width = config::get_required< unsigned int >
        (config, impl::param_name("processor", module, "ins.fetch_width"),
        config::NativeParser< unsigned int >(),
        config::NullValidator< unsigned int >());

ps.unit_latency.push_back(config::get_required< cyclesim::engine::cycles >
        (config, impl::param_name("processor", module, "ins.lat_unit_whitout"),
        config::NativeParser< cyclesim::engine::cycles >(),
        config::NullValidator< cyclesim::engine::cycles >()));
```

Fig. 7.2: Ejemplo del código de lectura de datos configurables.

Las dos líneas del "pitufosim.cfg "que cargan este código son:

```
processor.0.ins.fetch_width = 4
processor.0.ins.lat_unit_whitout = 1
```

Se puede ver en negrita en el código los datos a tener en cuenta para la lectura (en orden): donde se va a guardar el dato, el tipo de dato que se va a leer y la estructura a localiza en el archivo de configuración. De este último se estructura por nombre de módulo, número de módulo y texto que lo identifica.

De esta manera obtendremos:

```
ps.fetch_width = 4
ps.unit_latency.push_back = 1  (Aquí encola un ciclo de valor 1)
```

Siguiendo el paso 1, definiriamos la estructura:

```
ps.NewVariable = config::get_required< Type >
        (config, impl::param_name("processor", module, "NewVariable"),
        config::NativeParser< unsigned int >(),
        config::NullValidator< unsigned int >());
```

Fig. 7.3: Estructura genérica del código de lectura de un nuevo parámetro configurable.

Paso 3: Modificar el archivo de configuración "pitufosim.cfg" que se encuentra "/trunk/src/build/src/simulator/PitufoSim" (¡Observemos que esta en la carpeta compilada!). Añadimos la línea deseada:

```
processor.0.NewVariable = valor
```

Fig. 7.4: Ejemplo de la definición de un nuevo parámetro configurable.

CAPITULO 8:

BIBLIOGRAFÍA

CAPITULO 8: BIBLIOGRAFÍA

Libros

[PatHen93] David A. Patterson, John L. Hennessy
"Arquitectura de computadores: Un enfoque cuantitativo"
Ed. Mc Graw Hill, 1993

[Ana95] Pedro de Miguel Anasagasti
"arquitectura de computadores: fundamentos e introducción al paralelismo"
Ed. Paraninfo, 1995

[PatHen00] David A. Patterson, John L. Hennessy
"Estructura y diseño de computadores (Volumen 2)"
Ed. Reverté S.A., 2000

[HVZ03] Carl Hamacher, Zvonko Vranesic, Safwat Zaky
"Organización de computadores"
Ed. Mc Graw Hill, 2003

[SheLip06] John Paul Shen, Mikko H. Lipasti
"Arquitectura de computadores: Fundamentos de los procesadores superescalares"
Ed. Mc Graw Hill, 2006

Apuntes

"Processor Architecture"
Antonio Gonzalez

Arquitectura de computadores: Teoría 1 - 5
J.M. Llabería, E. Herrada, A. Olivé

Soporte digital

AC: Web CycleSim [Enero 2010]
< http://pcsostres.ac.upc.edu/cyclesim/doku.php >

Pin: PinTool [Enero 2010]
< http://www.pintool.org >

"Intel® 64 and IA-32 Architectures. Software Developer's Manual.
Volume 2A: Instruction Set Reference, A-M"
Intel Corporation, September 2009

"Intel® 64 and IA-32 Architectures. Software Developer's Manual.
Volume 2B: Instruction Set Reference, N-Z"
Intel Corporation, September 2009

Web EUITIO: "Moore y la ley de Moore" [Enero 2010]
Artículo escrito por *Rafael Fernandez-Font Pérez*
< http://petra.euitio.uniovi.es/~arrai/historia/trilobytes/5-Moore%20y%20la%20ley%20de%20Moore/Moore.htm >

Wikipedia: CPU [Enero 2010]
< http://es.wikipedia.org/wiki/CPU >

Wikipedia: Segmentación (Informática) [Enero 2010]
< http://es.wikipedia.org/wiki/Segmentaci%C3%B3n_(inform%C3%A1tica) >

Wikipedia: Segmentación de cauce [Enero 2010]
< http://es.wikipedia.org/wiki/Segmentaci%C3%B3n_de_cauce >

Wikipedia: Superescalar [Enero 2010]
< http://es.wikipedia.org/wiki/Superescalar >

Wikipedia: Unidad Funcional [Enero 2010]
< http://es.wikipedia.org/wiki/Unidad_funcional >

Wikipedia: Out-Of_Order [Enero 2010]
< http://es.wikipedia.org/wiki/Ejecuci%C3%B3n_fuera_de_orden >
< http://en.wikipedia.org/wiki/Out-of-order_execution >

Wikipedia: Dependencia de datos [Enero 2010]
<http://es.wikipedia.org/wiki/Dependencia_de_datos>

Datsi: Arquitectura de computadores [Enero 2010]
< http://www.datsi.fi.upm.es/docencia/Arquitectura/apres2.pdf >

Iteso: Organización y arquitectura de computadores: "Paralelismo a nivel de instrucción procesadores superescalares" [Enero 2010]
<http://iteso.mx/~jluis/acpdf/oto-04/11-Procesadores-superescalares.PDF >

FIB: Facultad de Informática de Barcelona [Enero 2010]
< http://www.fib.upc.edu/fib/>

UPC: Universidad Politécnica de Catalunya [Enero 2010]
< http://www.upc.edu/>

Oficina Empleo [Enero 2010]
<http://www.oficinaempleo.com/empresa/puestos/analista.htm>

Oficina Empleo [Enero 2010]
<http://www.oficinaempleo.com/empresa/puestos/programador.htm>