# Índice de documentos

Este tomo contiene los siguientes documentos:

**Memoria Proyecto Final de Carrera**

**Ingeniería en Informática**

# Vatios: Simulador de Procesador con Estimación de Potencia


**Autor: Juan Antonio Victorio Ferrer**

**Director: Enrique F. Torres Moreno**

**Dpto. de Informática e Ingeniería de Sistemas**

**Centro Politécnico Superior**


**Curso Académico 2006-2007**

gaZ


*Agradecimientos:*

*A mi director de proyecto, Enrique Torres, por haberme ayudado de modo excepcional a recorrer este camino.*

*A todos los miembros del grupo de arquitecturas, por haberme apoyado en todo momento*

*A mis familiares y amigos, por soportarme y animarme en los momentos difíciles.*

*A mis padres, por apoyarme durante todo mi paso por la universidad y por darme todo su cariño durante toda mi vida.*

# Tabla de Contenidos

# Índice de tablas



# Índice de figuras

# 1. Introducción

Este documento es la memoria del proyecto de fin de carrera realizado por Juan Antonio Victorio Ferrer dentro de la titulación de Ingeniería en Informática. Ha sido dirigido por Enrique Torres Moreno y desarrollado dentro del grupo de Arquitectura de Computadores (gaZ) de la Universidad de Zaragoza. El proyecto comenzó en Marzo de 2006 y va a ser defendido en la convocatoria de Abril de 2007.

# 2. Situación y problemática

Este proyecto se enmarca dentro del grupo de Arquitectura de Computadores de Zaragoza (gaZ). El gaZ cuenta con una historia como grupo investigador de 13 años, y se dedica a investigar en computación de altas prestaciones.

La arquitectura de computadores es la ciencia (o arte) de diseñar la estructura de los microprocesadores, esto incluye diseñar, elegir e interconectar distintos componentes, evaluar distintos compromisos y tomar las decisiones oportunas hasta conseguir los objetivos en cuanto a funcionalidad, rendimiento, consumos y costes.

La arquitectura de computadores puede dividirse en varias subcategorías, por poner varios ejemplos, el diseño del “Instruction Set Architecture”, que es la imagen que ofrece el microprocesador a los programadores, el diseño y dimensionado de la jerarquía de memoria, el diseño de los buses de interconexión, etc.…

La arquitectura de computadores va ligada a la tecnología electrónica, que es quien aporta la base para implementar físicamente los diseños. También esta muy ligada al software, ya que como objetivo se busca optimizar la ejecución de las aplicaciones. Cuando hablamos de la implementación física de un diseño, nos referimos a como se sitúan cada uno de los componentes (transistores) y como se trazan los cables (buses) que interconectan estos componentes.

Dada esta situación, se puede decir que la mejora en rendimiento de los procesadores se produce debido a mejoras que vienen tanto del ámbito de la arquitectura de procesadores, como del ámbito de la tecnología electrónica.

# 3. Problemática del Consumo

Históricamente, el consumo de potencia no ha sido un factor clave a la hora de diseñar un procesador CMOS de altas prestaciones, sin embargo la reducción del tamaño de los transistores y la miniaturización de los circuitos ha hecho que la densidad de potencia (vatios disipados por superficie del procesador) se dispare, y la potencia como factor de diseño aumente en importancia, llegando a ser uno de los factores críticos.

Además de todo esto, la potencia disipada por la actividad en los circuitos es y ha sido un factor clave a la hora de diseñar sistemas empotrados y dispositivos móviles, como PDA's, ordenadores portátiles, teléfonos móviles…, ya que la energía consumida, junto con la capacidad de las baterías condiciona el tiempo de funcionamiento del sistema.

En los ordenadores que no dependan de baterías para su funcionamiento, la potencia sigue siendo un factor crítico, ya que debido al aumento de la densidad de potencia, se han tenido que diseñar sistemas de refrigeración con costes altos, esto incluye ventiladores, disipadores y aires acondicionados. De hecho, se podría decir que por cada vatio que consume un procesador realizando sus cálculos, tenemos que gastar otro vatio para disipar el calor que ha sido generado. Otro factor no menos importante por el que se debe cuidar la potencia disipada, es la ecología.

Para avalar todo esto, podemos comentar que diseños de procesadores recientes, como el "Core Duo" de Intel han dado un paso atrás en rendimiento con respecto a diseños anteriores, poniendo énfasis en el consumo como se muestra en la Tabla 1. Queda probado, por lo tanto, que la potencia es un factor que nos condiciona a validar o rechazar posibles mejoras en el procesador, y que los diseñadores de microprocesadores deberían poseer herramientas que les ayudaran a tomar estas decisiones.

| Pentium 4 | Clock Speed | Power |
|---|---|---|
| P4 2.4 C | 2.4GHz | 67.6W |
| P4 3.06 HT | 3.06GHz | 81.8W |
| 560J | 3.6GHz | 115W |

| Intel Core Duo | Clock Speed | Power |
|---|---|---|
| T2600 | 2.16GHz | 31W |
| L2400 | 1.66GHz | 15W |
| U2500 | 1.20GHz | 9W |

**Tabla 1 Consumos de procesadores típicos de la gama Intel Pentium 4 e Intel Core Duo**

# 4. Estado del Arte - Simuladores

Es indudable que los simuladores son fundamentales a la hora de diseñar un procesador, proporcionan al diseñador información con la que poder tomar decisiones, sin tener que llegar a implementar físicamente ese diseño creando prototipos.

Tradicionalmente los simuladores se han centrado en hacer una simulación temporal del paso de las instrucciones por el segmentado del procesador, calculando medidas del rendimiento como puede ser el IPC (instrucciones por ciclo), número de accesos a memoria, saltos mal predichos, etc… Sin embargo, en la actualidad se busca un compromiso entre el rendimiento y el consumo, buscando soluciones que mejoren significativamente el rendimiento sin penalizar excesivamente el consumo.

Sin embargo, hasta hace poco no existían herramientas que además permitieran calcular cuánta energía se consumía al ejecutar una instrucción, un programa o al acceder a un banco de memoria determinado. Ha sido en los últimos años, cuando han empezado a desarrollarse herramientas que tienen en cuenta la potencia. A continuación se va a describir un subconjunto representativo de estas herramientas y sus principales características.

## *4.1. SPICE*

SPICE es un simulador para circuitos, de carácter general, que permite modelar todo tipo de componentes eléctricos y electrónicos, incluyendo los tipos más comunes de dispositivos semiconductores. SPICE tiene su origen en el “Electrical Engineering and Computer Sciences Department” en la Universidad de California, en Berkeley. Los resultados generados por SPICE son muy detallados y precisos, pero SPICE presenta inconvenientes serios para realizar simulaciones en el área de la arquitectura de computadores.

El principal inconveniente es que nos obliga a tener una descripción muy detallada (a nivel de transistor) del circuito que vamos a simular. Esta descripción no se encuentra disponible en las etapas previas del diseño de un procesador, ya que todavía no se sabe en concreto que unidades van a componer el procesador y que características tendrán.

Aún suponiendo que tuviéramos esta descripción, dado que la precisión de las simulaciones con SPICE es muy alta, el tiempo de CPU que cuesta realizar una simulación es muy alto, por ello, con la capacidad de cálculo actual, no es viable obtener resultados en cuanto a potencia

de la simulación de una ejecución de un Benchmark representativo (por ejemplo SPEC) sobre un procesador.

Por ambos motivos, es mucho menos viable realizar un barrido de simulación a lo largo de todo un abanico de opciones de diseño, con el objetivo de decidirnos por la opción que más reduzca el consumo.

## 4.2. CACTI

CACTI es un simulador, desarrollado en su primera versión dentro del DEC Western Research Laboratory en Palo Alto, California en 1994 [3] y que actualmente se encuentra en la versión 4.2 [6] y es desarrollado por David Tarjan, Shyamkumar Thoziyoor y Norman P. Jouppi dentro de la compañía HP, en HP Laboratories Palo Alto, California. La versión 5.0 se puede encontrar como versión Beta.

CACTI es una herramienta para modelar memorias cache, que ha sido ampliamente adoptada por los arquitectos. En su primera versión solamente modelaba los parámetros temporales de las memorias cache (retardos), pero en las últimas versiones se han incorporado un modelo de área (área de silicio ocupada por las memorias) y de potencia. En las distintas versiones de CACTI [4][5] se han ido adaptando y corrigiendo los modelos y parámetros tecnológicos con el fin de optimizar los resultados para los tamaños de cache más comunes. CACTI esta escrito en C, puede utilizarse libremente, bien descargándolo de la página Web: http://www.hpl.hp.com/personal/Norman_Jouppi/cacti4.html o bien utilizando un servicio Web, mostrado en la Figura 1, también accesible desde esa misma página.

Las simulaciones realizadas con CACTI son relativamente ligeras en cuanto a tiempo de simulación y los resultados generados han sido verificados con SPICE, y se encuentran dentro de un margen de error menor al 10%.

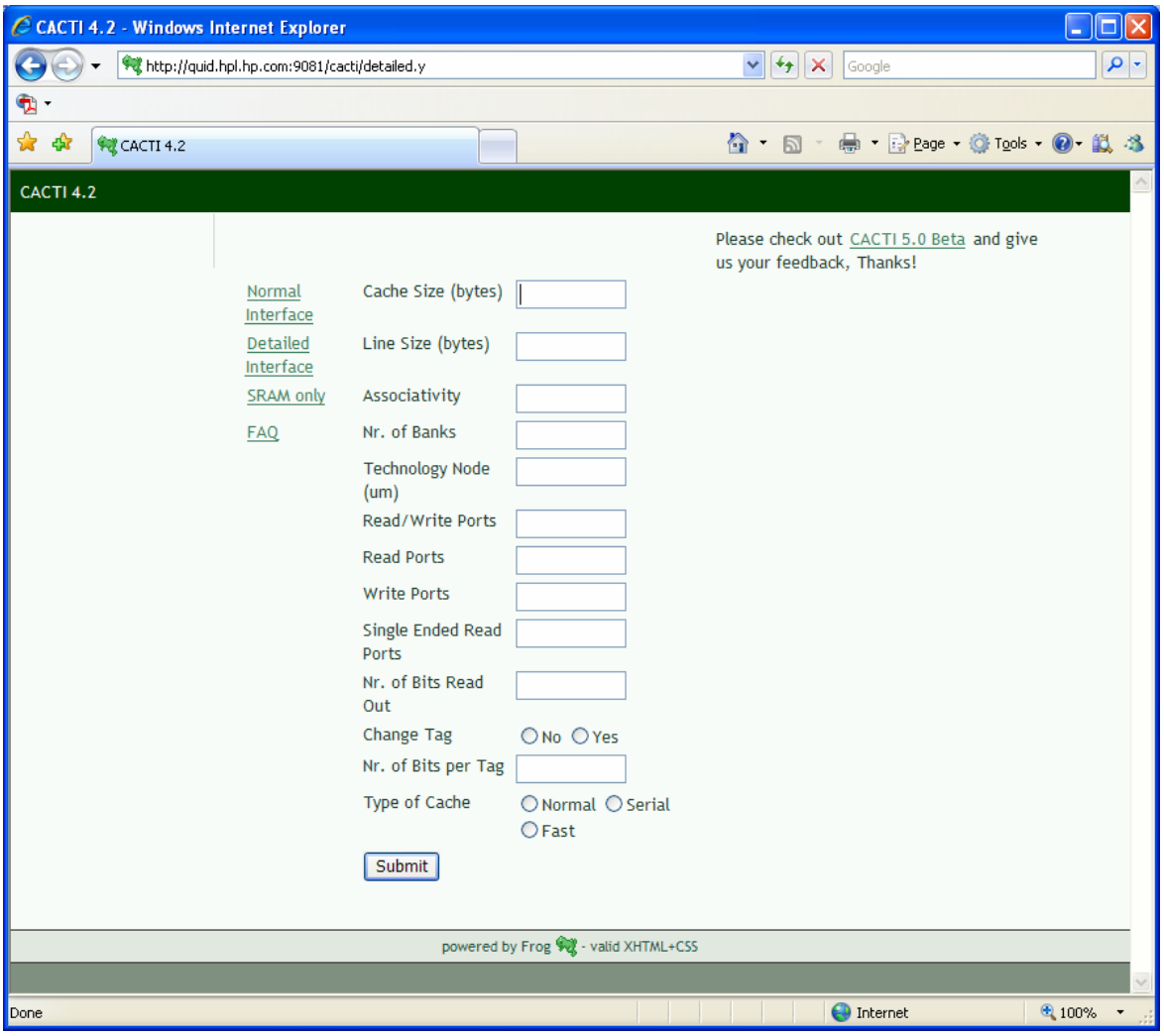



**Figura 1 Vista de la versión Web de la aplicación CACTI 4.2**

### 4.3. Wattch

Wattch [1] es un simulador basado en SimpleScalar presentado en el año 2000 en el International Symposium on Computer Architecture (ISCA) y desarrollado por David Brooks de la universidad de Princeton. Este simulador puede descargarse desde: http://www.eecs.harvard.edu/~dbrooks/wattch-form.html

SimpleScalar [2] es un conjunto de simuladores y herramientas, cuyo máximo exponente es `sim-outorder`, un simulador de procesador superescalar fuera de orden. Modela las principales unidades de cualquier procesador de este tipo y dado que su distribución es libre y viene acompañado con una documentación suficiente, sirve como base para la creación de simuladores por parte de numerosos grupos de investigación en universidades y empresas. SimpleScalar no es una herramienta que modele potencia, pero se presenta como una buena base de la cual partir a la hora de crear un simulador. La página Web oficial desde la cual puede ser descargado es: http://www.simplescalar.com/

Wattch se presenta como un framework para analizar y optimizar la potencia disipada a nivel de arquitectura. Wattch se perfila como una alternativa muy interesante a las herramientas disponibles, ya que es una herramienta que modela el consumo de un procesador superescalar, es varios órdenes de magnitud más rápido que SPICE y de alto nivel, es decir, no requiere un modelo físico del procesador, sino que requiere simplemente los parámetros que lo definen, como son el ancho del procesador, tamaños y asociatividades de las caches, número de puertos de memoria, número de unidades funcionales, etc. Estas características la sitúan como herramienta idónea a la hora de tomar decisiones tempranas del diseño.

Wattch ha sido ampliamente utilizado en la investigación en los últimos años a la hora de proponer nuevas estructuras, alternativas de diseño, etc. y sus resultados han sido utilizados para validar numerosos artículos de investigación en los últimos años.

# 5. Objetivos del proyecto

Dado el interés que tiene una herramienta como Wattch, y después de un breve análisis de su funcionamiento se decidió fijar los objetivos de este proyecto.

- En primer lugar, y dado que la documentación que hay sobre Wattch se reduce al artículo presentado en el ISCA, se estableció como primer objetivo de este proyecto realizar un análisis en profundidad de Wattch y ampliar la documentación existente sobre este simulador.

- Como segundo objetivo de este proyecto se decidió construir un simulador que mejorara a Wattch, identificando y corrigiendo deficiencias que presenta Wattch y mejorando el modelo de potencia. También se decidió añadir mayor flexibilidad al simulador en varios puntos. Consiguiendo un simulador libre, con documentación en inglés y disponible a disposición pública en la página Web: http://webdiis.unizar.es/gaz/vatios/ . Este simulador se denomina `sim-vatios`.

- Como tercer y último objetivo de este proyecto se propuso crear un simulador específico para el gaZ (grupo de arquitectura de la Universidad de Zaragoza). Este simulador modela un procesador más actual y específico que el modelado por Wattch (que en definitiva es el mismo que el modelado por SimpleScalar). Además se propuso crear nuevos modelos de potencia que permitan obtener estimaciones de consumo de este nuevo modelo de procesador. Este simulador se denomina `sim-gaz`.

# 6. Metodología – Línea de trabajo

En primer lugar debería decir que para garantizar la buena marcha del proyecto, desde un primer momento se establecieron reuniones con frecuencia de al menos una reunión semanal con el director, donde se comentaban los últimos resultados obtenidos y se planificaba el trabajo de los próximos días. Las fases por las que ha transcurrido este proyecto se pueden ver de modo gráfico en la Figura 2.

Las fases de documentación han sido un punto clave a la hora de afrontar este proyecto. Antes de realizar cualquier análisis o de programar, ha sido necesario documentarse ampliamente sobre aspectos generales de arquitectura de computadores [7][17], tecnología electrónica [15][21] y de consumo energético [8][9][23].

Previo a la elaboración de la propuesta del proyecto, se hizo un estudio de los simuladores disponibles, sus características, y su popularidad a la hora de ser utilizados como herramientas para validar trabajos de investigación.

La propuesta de este proyecto se concretó en una presentación del proyecto al grupo de arquitecturas, donde se presentaron los objetivos y se obtuvo realimentación. Además, durante toda la duración del proyecto se han ido manteniendo reuniones periódicas con varios miembros del grupo de arquitecturas.

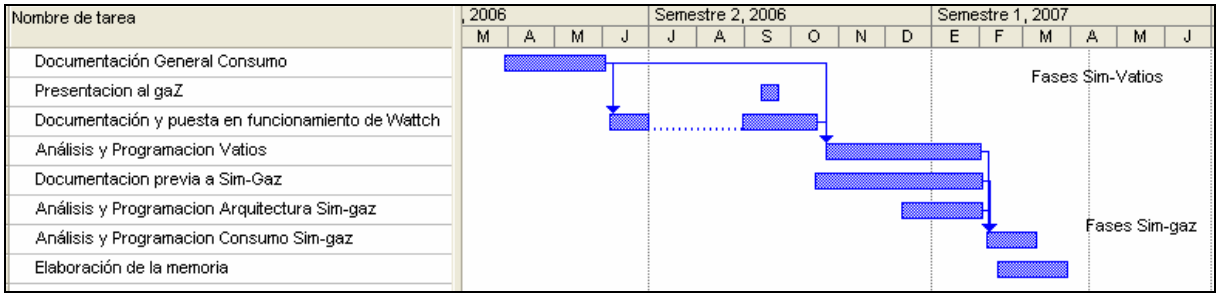



**Figura 2 Diagrama Gantt del proyecto**

Una vez presentado y aprobado por el grupo, documentado, y habiendo estudiado el funcionamiento de Wattch se pasó al análisis e implementación de `sim-vatios`. Este simulador ha sido liberado al público, y además constituyó un primer paso para la implementación de `sim-gaz`.

Con respecto a la programación de `sim-vatios`, cabe decir que se programó manteniendo la compatibilidad de resultados con Wattch. Esta metodología nos ayudó a verificar el

desarrollo de la herramienta hasta el momento de sustituir la librería de CACTI 1.0 por la de CACTI 4.2, que fue el último paso de la programación de `sim-vatios`. Se ha mantenido una versión llamada `sim-testvatios` para verificar que los cambios realizados siguieran manteniendo la compatibilidad cuando así fuera necesario.

Paralelamente a la programación de `sim-vatios` hubo una fase de documentación muy extensa, necesaria para implementar `sim-gaz`, ya que este simulador iba a simular técnicas aplicadas en procesadores modernos. Este tipo de procesadores cuentan con técnicas no estudiadas en las asignaturas de arquitectura de computadores a lo largo de la carrera.

Esta fase de documentación incluye el estudio de artículos de investigación presentados en los principales congresos de arquitectura de computadores, la lectura de varias patentes y el estudio de manuales de procesadores como el Itanium II.

Una vez programado el simulador `sim-gaz`, habiendo hecho un estudio de cómo se iba a modelar la potencia y el consumo, se integró el módulo de consumo de `sim-vatios` en `sim-gaz`. Quedando terminada la aplicación. En la Figura 3 se puede ver como la elaboración de `sim-vatios` fue un paso previo a al elaboración de `sim-gaz.`

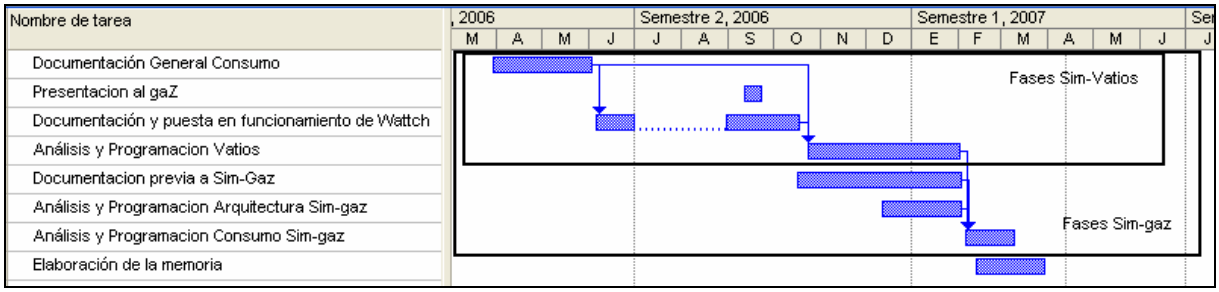



**Figura 3 Diagrama de Gannt diferenciando objetivos**

# 7. Estudio y documentación de Wattch

Como primer objetivo en este proyecto se propuso hacer un estudio detallado del funcionamiento de Wattch, generando documentación que ampliara la ya existente, que básicamente se limita al artículo presentado al congreso ISCA. Wattch es usado en gran número de artículos de investigación como prueba para verificar o validar modificaciones o alternativas propuestas. Se considera necesario comprender el funcionamiento de la herramienta más allá de la información que proporciona la documentación disponible.

Como resultado de este trabajo, se ha creado un documento que contiene una descripción detallada de Wattch, incluido en este tomo como Anexo A. En este anexo se describen generalidades del simulador, funcionamiento interno, clasificación de unidades modeladas, interacción con las librerías de CACTI y un repaso de todas las unidades que modela con una breve explicación para cada una de ellas.

Como puntos interesantes a resaltar del funcionamiento de Wattch, merecen especial mención en esta memoria los siguientes:

- Sin entrar en detalles se debe comentar que Wattch simula un modelo de procesador descrito en Complexity-Effective Superscalar Processors [10], cuyo contenido se amplia en el respectivo Tech Report [11], pero sin embargo este modelo se construye sobre el procesador de sim-outorder, que implementa una arquitectura completamente distinta.

- Wattch modela el consumo dinámico, es decir, el que se produce por la conmutación de los transistores al cambiar de estado. No modela el consumo estático, producido por las corrientes de fuga a pesar de que el circuito no conmute.

- Wattch incluye la definición de varios puntos tecnológicos (tecnologías de fabricación), pero el usuario tiene que escoger uno antes de compilar la aplicación. Cada vez que el usuario desea cambiar de punto tecnológico tiene que recompilar la aplicación.

- No todos los puntos tecnológicos son válidos. Wattch no es capaz de escalar, sino que para cada tecnología define una serie de parámetros, como factor de resistencia, factor de capacidad, factor de longitud, etc. todos ellos relativos a una tecnología concreta, por lo que no puede ser utilizado fuera de este rango de tecnologías.

- Wattch hace uso de CACTI a la hora de calcular la potencia de las memorias cache, sin embargo, y debido a que en la versión de CACTI 1.0, que es la que se incluye, no se calcula potencia, Wattch simplemente llama a CACTI para obtener los datos del subbanking de las memorias. Además las llamadas se hacen para la escala tecnológica para la que esta programado CACTI, que normalmente difiere de la que se quiere simular.

- Además Wattch utiliza dentro de sus propias fórmulas para el cálculo de potencia funciones primitivas definidas en CACTI. Los resultados de estas llamadas tampoco son apropiadamente escalados en función de la tecnología.

- Con respecto a la lógica del funcionamiento de Wattch se podría decir que primero calcula el pico de potencia de cada una de las unidades antes de empezar a simular, para luego calcular la energía dinámica realmente consumida escalando en función del uso-actividad de la unidad durante la ejecución.

- Wattch nos da los consumos en función de distintas técnicas de clock-gating, descritas en el artículo original. El clock-gating es una técnica que permite ahorrar energía en procesadores, inhibiendo la señal de reloj a circuitos que no necesitan conmutar.

# 8. Sim-vatios

Como segundo de los objetivos de este proyecto se propuso desarrollar un simulador que corrigiera ciertos fallos que presenta Wattch y ampliase su flexibilidad. Este simulador supone el primer desarrollo de este proyecto. Además supone un primer paso para el desarrollo de `sim-gaz`, que es el último objetivo de este proyecto.

## *8.1. Requisitos*

Una de los primeros requisitos para `sim-vatios` fue la arquitectura que modelaría. Dado que iba a ser una versión libre a disposición pública se decidió que la arquitectura fuera la misma que la de SimpleScalar y Wattch. Al implementar la misma arquitectura se consigue una versión orientada al público más general, ya que se facilita su adopción.

En primer lugar Wattch integra varios puntos tecnológicos, es decir, puede estimar la potencia consumida para un procesador implementado en distintas escalas de integración, el problema que se plantea es que si queremos simular el consumo en distintas escalas, tenemos que recompilar la herramienta, proceso que parece a primera vista innecesario. Se establece como requisito el poder simular para distintos puntos tecnológicos sin tener que recompilar, pasando como parámetro de ejecución al simulador el punto en el que queremos simular.

Wattch utiliza un único modelo de potencia para cada una de las unidades. Uno de los objetivos de este proyecto fue el de crear un simulador preparado para integrar varios modelos de potencia para cada una de las unidades. Adicionalmente se propuso como requisito que se pudiera excluir una unidad del cómputo de potencia, o también que el valor de potencia pico de una unidad pudiera ser introducido manualmente por el usuario al invocar el programa.

Un requisito muy significativo a la hora de desarrollar `sim-vatios` fue eliminar la restricción que obliga a re-simular la ejecución temporal (que puede ser muy costosa en tiempo) cada vez que el usuario quisiera cambiar de modelo de potencia o el punto tecnológico para el cual estamos calculando la potencia. Se podrá evitar tener que volver a realizar la simulación temporal siempre y cuando no cambie ningún parámetro arquitectónico que altere el comportamiento temporal de los componentes. Se entiende por simulación temporal el tomar un ejecutable o traza de ejecución e ir instrucción por instrucción simulando las etapas por las que pasa dentro del segmentado del procesador.

Esto lleva a dos casos de uso de la aplicación: El primero mostrado en la Figura 4, la primera simulación sobre una traza de ejecución y arquitectura concreta. Para esto se utilizará un ejecutable llamado `sim-vatios`. La ejecución generará un fichero con los resultados, y otro con parámetros de la arquitectura y uso/actividad observados durante la simulación de la traza.

El segundo caso de uso, mostrado en la Figura 5, representa las siguientes simulaciones sobre la misma traza de ejecución y arquitectura, variando parámetros tecnológicos y/o modelos de potencia. Para este segundo caso se utilizará un ejecutable llamado `power-vatios`. Se requerirá el fichero de uso/actividad/arquitectura correspondiente generado por `sim-vatios`, y se generará un fichero de resultados.

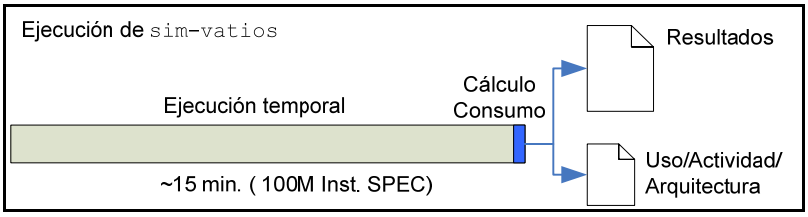



**Figura 4 Caso de uso: Primera simulación**

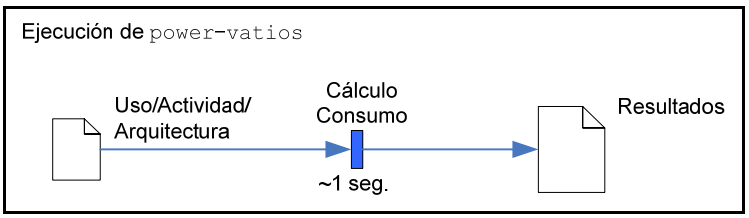



**Figura 5 Caso de uso: Siguientes simulaciones**

Dado que la versión de la herramienta CACTI que integra Wattch es la 1.0, y la versión actual es la 4.2, se decidió actualizar esta herramienta, ya que la nueva versión optimiza los resultados para los tamaños de cache más actuales.

Por último, se fijo como requisito crear un manual de usuario para que el simulador pudiera ser utilizado, y un manual del programador, para poder ser modificado si se considera necesario. Ambos en idioma inglés, ya que van a ser de dominio público (Anexos B y C).

## *8.2. Diseño e Implementación*

La parte más complicada del diseño fue encontrar un método para separar la ejecución temporal de la simulación temporal, del cálculo de la potencia. A continuación se describe, a rasgos generales, la solución que se propuso y adoptó.

El simulador realiza la simulación temporal, pero en vez de calcular ciclo a ciclo el consumo y acumularlo, `sim-vatios` almacena ciclo a ciclo estadísticas de uso/actividad de las unidades. Para guardar esta información solamente necesitamos estructuras de datos de tipo vector y ciclo a ciclo realizamos operaciones básicas de incremento. Por el contrario, Wattch realiza numerosas operaciones en punto flotante, que no son necesarias, y han sido eliminadas en sim-vatios. Con todo esto se mejora el rendimiento de nuestra aplicación frente a Wattch.

Con las estadísticas que se han almacenado ciclo a ciclo durante la simulación y la información de la arquitectura (que introduce el usuario al igual que en Wattch) conseguimos, al final de la simulación, calcular la potencia/energía consumida. A continuación se muestran por pantalla los resultados de consumo/potencia, del mismo modo que haría Wattch.

Utilizando la librería de opciones de SimpleScalar, y con una pequeña modificación conseguimos poder volcar las estadísticas de uso/actividad a un fichero, para posteriormente, y con otro ejecutable llamado `power-vatios` que forma parte de nuestra solución, poder recalcular potencia/energía utilizando otros modelos de potencia o calculando para otros puntos tecnológicos, sin tener que re-simular la ejecución temporal.

Se detecto una incompatibilidad entre este sistema y el modelo que tiene Wattch de calcular el consumo de la distribución de la señal de reloj, sin embargo en nuestro simulador hemos propuesto un modelo alternativo que genera resultados igualmente válidos.

Previo a la actualización de CACTI a la versión 4.2, se hizo un estudio y comparación de los resultados de las versiones 1.0 y 4.2 de esta herramienta, con el objetivo de detectar diferencias entre versiones y los límites de su uso. Además se corrigieron los fallos que presentaba Wattch, que provocaban que los resultados de las llamadas a CACTI se calcularan para una tecnología que no correspondía con la deseada. Con la posibilidad de escalado tecnológico que ofrece CACTI 4.2, ahora los resultados a las llamadas son respecto a la tecnología que realmente se quiere simular.

## 8.3. Resultados

Como resultado de los cambios en los modelos de potencia se ha obtenido esta tabla con ejemplos de los valores de potencia calculados, así como el consumo total del procesador (procesador por defecto de SimpleScalar). En esta tabla se compara la potencia pico para los siguientes simuladores: Wattch, `sim-testvatios` (simulador desarrollado para comprobar la compatibilidad de resultados con Wattch), `sim-vatios`, en el que se han corregido deficiencias en los modelos de potencia, principalmente debidas a utilizar CACTI con una escala de integración distinta a la simulada, y `sim-vatios` (Modelo 3) en el que la potencia calculada para unidades que contienen estructuras tipo cache, se calcula directamente mediante CACTI.

| Potencia Máxima (W) | Wattch | TestVatios | Vatios | Vatios (Cache Modelo 3) |
|---|---|---|---|---|
| Renombre | 0,42 | 0,42 | 0,24 | - |
| Cache de Datos | 6,12 | 6,12 | 6,35 | 2,02 |
| Otros | 39,5335 | 39,5335 | 30,6473 | 27,2529 |
| Total | 46,08 | 46,08 | 37,24 | 29,52 |

**Tabla 2 Tabla de resultados de sim-vatios**

En la Tabla 2 en primer lugar se puede observar la equivalencia entre usar Wattch o sim-testvatios. También se puede apreciar como para el renombre se reduce la potencia calculada, al pasar a Vatios, ya que las funciones de CACTI sobre las que se apoya Wattch devuelven sus resultados para la escala tecnológica adecuada. Dado que el renombre contiene estructuras que no se asemejan a una cache, no se ha implementado modelo 3 (utilizando CACTI para todo el cálculo de potencia).

Para la cache de Datos, el consumo aumenta al sustituir CACTI 1.0 por CACTI 4.2, ya que el sub-banking devuelve otro resultado. Sin embargo al calcular la potencia directamente con CACTI la predicción de la potencia disminuye, poniendo de manifiesto la inexactitud de Wattch.

En general, para el total del procesador, la potencia disminuye con Vatios y Vatios (Modelo 3) ya que muchos cálculos se realizan para la escala de integración correcta.

Otro de los resultados interesantes que pudimos comprobar es que gracias a la separación creada entre la simulación temporal y el cálculo de la potencia, reducíamos considerablemente el tiempo necesario para la ejecución de un juego de simulaciones.

En la Figura 6 se muestra un escenario de ejecución de Wattch contra Vatios. Con Vatios podemos obtener resultados para un abanico de configuraciones sin re-simular la ejecución temporal.

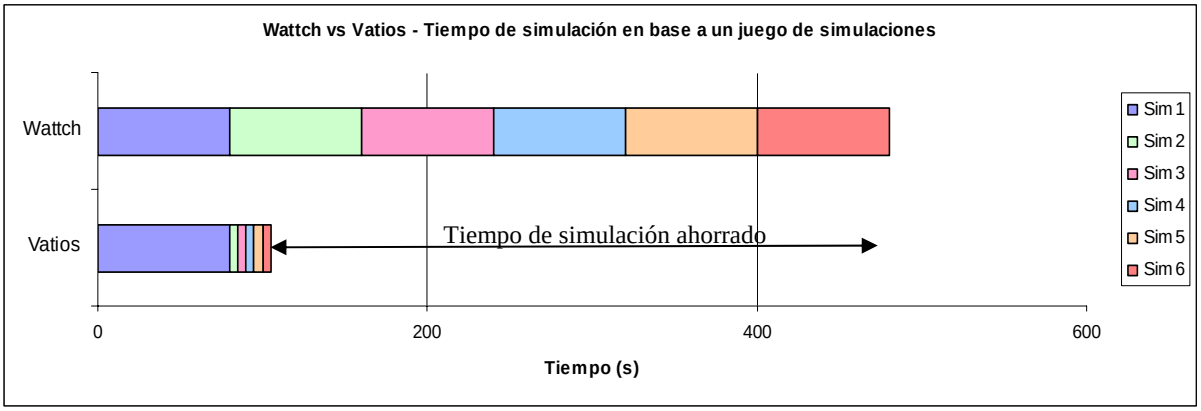



**Figura 6 Escenario de ejecución Wattch contra Vatios**

En la Tabla 3 se puede comprobar como la reducción de operaciones en punto flotante por ciclo de simulación reduce ligeramente el tiempo de ejecución de `sim-vatios` en comparación con Wattch al simular sobre dos trazas representativas de los Benchmark SPEC 2000 (`ammp-ref` y `eon-ref`). Sin embargo la principal fuente de ahorro de tiempo de ejecución se obtiene cuando se simula la misma traza para otras escalas tecnológicas o con otros modelos de potencia. Al evitar una nueva simulación temporal se reduce drásticamente el tiempo necesario para cubrir un abanico de simulaciones.

| Tiempo de simulación en segundos para 2 trazas representativas SPEC 2000 | | |
|---|---|---|
| 1ª Simulacion | sim-vatios | Wattch |
| ammp-ref | 460 | 471 |
| eon-ref | 471 | 484 |
| Simulaciones posteriores (sin alterar temporización) | | |
| ammp-ref | 0,056 | ~471 |
| eon-ref | 0,057 | ~484 |

**Tabla 3 Comparación de tiempos de simulacion Wattch - sim-vatios**

# 9. Sim-gaz

Como último objetivo de este proyecto se propuso crear un simulador de ámbito restringido, adaptado a las necesidades del grupo de Arquitectura de la Universidad de Zaragoza.

Este simulador esta creado de forma análoga a `sim-vatios`, pero implementa y modela una arquitectura más moderna. Una descripción más detallada del simulador `sim-gaz`, su arquitectura, implementación y el modelo de potencia se encuentra disponible en el Anexo D

## *9.1. Documentación*

A diferencia de Wattch o de `sim-vatios`, `sim-gaz` es un simulador con una arquitectura totalmente distinta, que modela de manera más representativa los procesadores actuales. Este objetivo implicó una fase de documentación acerca de las técnicas más modernas utilizadas en los procesadores, así como estudiar artículos de investigación de publicación reciente.

Una de las zonas críticas de los procesadores de hoy en día es la lógica de lanzamiento de instrucciones, por ello se estudiaron las diferentes alternativas a la hora de implementar esta lógica en varios procesadores [14][16]. El estudió de este mecanismo de lanzamiento de instrucciones implicó incluso llegar a tener que estudiar patentes.

Así mismo, tuvieron que estudiarse publicaciones que explicaban el segmentado de procesadores modernos [12][13], las etapas que lo componen, y que acciones se realizan en cada una de ellas, valga como ejemplo, el estudio del mecanismo estándar de renombre.

La jerarquía de memoria es otra de las zonas críticas de un procesador. Un sistema de memoria con una alta latencia perjudica seriamente el rendimiento de todo el procesador. Por otro lado el sistema de memoria que modela SimpleScalar (y por lo tanto Wattch y `sim-vatios`) es inexacto y poco representativo, comparado con el sistema de memoria de los procesadores de gama alta disponibles actualmente.

A la hora de implementar `sim-gaz`, se optó por un modelo de memoria tipo Itanium II, lo que obligó a hacer un minucioso estudio de la jerarquía de memoria de ese microprocesador [25][26][27][28][29].

`Sim-gaz` también debe implementar al menos un modelo de potencia para cada unidad, a ser posible utilizando CACTI 4.2. Esto conlleva una gran complejidad, ya que para crear un

modelo que calcule la potencia de una unidad es necesario comprender de forma clara su comportamiento, para posteriormente crear un modelo que refleje el consumo de esta unidad.

## *9.2. Requisitos*

Los requisitos deseados para el simulador `sim-gaz` fueron los siguientes:

Simulador basado en `sim-vatios`, que conserve las mejoras que este implementa. Además debe incluir los siguientes cambios en la arquitectura:

- Modelo estándar de renombre.
- Modelo de estaciones de reserva (RUU) sustituido por modelo de ventanas de lanzamiento (ROB + IQ)
- Revisión de las etapas del segmentado por las que pasa cada instrucción.
- Implementación de una jerarquía de memoria con latencia variable, tipo Itanium II
- La jerarquía tendrá memoria cache de nivel 1 separada para instrucciones y datos, los niveles 2 y 3 de memoria cache serán unificados.
- Esta jerarquía debería incluir un modelo de las principales unidades presentes en un sistema de memoria (STB, LDB, MAF, L2Q, MAF2)

Adicionalmente, y dado que el fin de este simulador es el de calcular potencia, se debería crear un modelo de potencia para las unidades más significativas no existentes en Wattch.

## 9.3. Implementación

La implementación de `sim-gaz` se realizó sobre SimpleScalar, además se ha contado con el modelo de renombre de Jesús Alastruey y las modelos de las estructuras de memoria de Luís Ramos.

Para simular correctamente el paso por el segmentado de las instrucciones, se implementó un sistema de colas con prioridad, más complejo que el que tiene SimpleScalar por defecto. A continuación se realizaron varias modificaciones generales para, posteriormente, modificar las etapas del segmentado, teniendo en cuenta las necesidades de la jerarquía de memoria que se iba a integrar posteriormente.

El siguiente paso en la construcción del simulador fue integrar el sistema de memoria y depurar su funcionamiento. El correcto funcionamiento del simulador fue probado mediante la ejecución de trazas representativas del Benchmark SPEC 2000.

Una vez implementada la parte funcional del simulador, se añadió la librería de potencia de `sim-vatios`. Añadida la librería y verificado que se integraba correctamente, se crearon los modelos de potencia para cada una de las nuevas unidades, al mismo tiempo que se añadieron en el código de cada una de las etapas del segmentado los contadores de uso/actividad correspondientes.

## 9.4. Resultados

Como resultado del trabajo realizado en este proyecto para crear `sim-gaz`, se ha conseguido un simulador, que funciona y esta siendo utilizado y validado por Enrique Torres. Este simulador será adoptado próximamente por el resto de miembros del grupo de arquitecturas como herramienta para sus trabajos de investigación.

# 10. Conclusiones

## *10.1. Aportación del proyecto - Objetivos Cumplidos*

Este proyecto se adentra en un campo dentro del área de los simuladores y la arquitectura de computadores apenas explorado. Cuando empecé el proyecto Wattch era el único referente en cuanto a simuladores que calcularan potencia o consumo, además era frecuentemente utilizado por grupos de investigación para validar sus trabajos. A pesar de ser el simulador más famoso y utilizado, la documentación que se puede encontrar acerca de Wattch fue muy escasa. Además pronto se pusieron de manifiesto errores y aspectos mejorables del simulador.

Con este proyecto se ha analizado la herramienta en profundidad, estudiado su funcionamiento interno, identificado fallos y virtudes y además se ha contribuido a mejorar la documentación disponible, ayudando a los posibles usuarios a comprender el funcionamiento y poniendo de manifiesto las posibles limitaciones.

Se ha creado una versión más general y flexible de la aplicación con la intención de que sea utilizada por la comunidad investigadora, ya que mejora en muchos aspectos la aplicación de referencia hasta el momento y que permite ahorrar tiempo a la hora de obtener resultados.

Se ha escrito la documentación necesaria, para que en caso de que sea necesario, la aplicación sea mejorada o ampliada en función de las necesidades futuras.

Por otro lado se ha desarrollado una versión específica, adaptada al modelo de procesador base del grupo de arquitecturas de la Universidad de Zaragoza, que podrá ser utilizada por el grupo para validar sus investigaciones. También servirá de base para el desarrollo de simuladores más específicos orientados a la investigación de nuevas soluciones dentro del campo de la arquitectura de computadores.

## 10.2. Posibilidades de continuación, mejora, ampliación

Este proyecto es válido en la situación actual de la tecnología y la arquitectura de computadores, sin embargo estamos seguros de que la situación cambiará y no sabemos muy bien hacia donde se dirigirá la investigación dentro de unos años. Quizás dentro de un tiempo algún avance tecnológico haga que la potencia disipada por un procesador no sea un factor crítico a la hora de diseñar un microprocesador, tal y como empresas como Intel buscan en sus investigaciones.

Sin embargo, a corto plazo esta claro que las escalas de integración aumentarán, fabricando transistores cada vez más pequeños y procesadores cada vez más densos. Adaptar este simulador a las nuevas escalas de integración puede ser un trabajo interesante, así como actualizar el uso de la biblioteca CACTI a las nuevas versiones que seguro se desarrollarán en los próximos años.

También sería igualmente interesante crear librerías que incluyan nuevos modelos de potencia más precisos para determinadas unidades, e incluso añadir nuevas unidades que pasen a formar parte indispensable de nuevas generaciones de procesadores.

Dado que en los últimos años se han introducido en el mercado nuevos procesadores multi-núcleo podría parecer interesante desarrollar una versión de Vatios que soporte la ejecución de varios hilos simultáneamente, e incluso soportando primitivas de sincronización, por ejemplo por memoria compartida.

## 10.3. Principales problemas

Los principales problemas que hemos encontrado tienen que ver con la herramienta Wattch. A pesar de ser la más popular y utilizada, no hemos encontrado apenas documentación sobre ella. Solamente hemos encontrado un simulador [24] basado en Wattch que lo mejorara, realizado por un grupo de la Universidad Politécnica de Madrid, cuyo código fuente no se encuentra disponible. Además las mejoras se centran exclusivamente en las unidades funcionales de cálculo.

Durante el desarrollo de la aplicación hemos detectado varios problemas que presenta Wattch y no pueden ser determinados a simple vista, por ejemplo que ciertos resultados no correspondan a la tecnología que se esta simulando, sino a otra distinta. También hemos tenido problemas a la hora de modelar en `sim-gaz` de manera genérica el consumo de algunas unidades, por ejemplo de la IQ.

## *10.4. Opinión personal*

Considero que este proyecto me ha servido para satisfacer cierta necesidad de conocimiento generada a lo largo de la carrera. Desde que empecé la carrera me ha gustado la arquitectura de computadores, y al empezar este proyecto lo vi como una magnífica oportunidad para ampliar mis conocimientos en el área.

Uno de los aspectos más interesantes de este proyecto ha sido descubrir la problemática que hay detrás de la investigación en arquitectura de computadores, o a la hora de diseñar un procesador que va a salir al mercado. Considero que si bien en las asignaturas se pueden aprender las bases del funcionamiento de un procesador, quedan muchos aspectos que pueden parecer secundarios a primera vista, pero que en realidad influyen y pueden condicionar el resultado final.

Es la primera vez que afronto un proyecto de esta magnitud, trabajando sobre una herramienta real, y no creada con fines académicos. Además me satisface haber hecho un proyecto que pueda colaborar con la comunidad investigadora. Así mismo he notado que hay una escasez de herramientas a disposición pública que permitan desarrollar de manera rápida y eficiente los trabajos de investigación dentro de esta área.

# 11. Bibliografía y referencias

[1] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In 27th Annual International Symposium on Computer Architecture, June 2000.

[2] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set Version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin--Madison, May 1997.

[3] An Enhanced Access and Cycle Time Model for On-Chip Caches.Wilton, Steven J.E; Jouppi, Norman P. WRL-93-5 Jul 1994

[4] CACTI 2.0: An Integrated Cache Timing and Power Model. Reinman, Glen; Jouppi, Norman P. WRL-2000-7 Feb 2000

[5] "CACTI 3.0: An Integrated Cache Timing, Power and Area Model," Premkishore Shivakumar and Norman P. Jouppi , Western Research Lab (WRL) Research Report 2001/2

[6] CACTI 4.0 Tarjan, David; Thoziyoor, Shyamkumar; Jouppi, Norman P. HPL-2006-86 20060606

[7] "Energy/power breakdown of pipelined nanometer caches (90nm/65nm/45nm/32nm)." Samuel V. Rodriguez and Bruce Jacob. Proc. International Symposium on Low Power Electronics and Design (ISLPED 2006), pp. 25-30. Tegernsee Germany, October 2006.

[8] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects. Technical Report CS-2003-05, University of Virginia

[9] J. A. Butts and G. S. Sohi. A static power model for architects. In Micro-33, pages 191--201, December 2000. http://citeseer.ist.psu.edu/butts00static.html

[10] Subbarao Palacharla, Norman P. Jouppi and J.E. Smith. "Complexity-Effective Superscalar Processors". In 24th Annual Internacional Symposium on Computer Architecture, Denver, 1997, pp. 206-218.

[11] Subbarao Palacharla, Norman P. Jouppi, and James E. Smith. Quantifying the complexity of superscalar processors. Technical Report CS-TR-96-1328, University of Wisconsin, Madison, 19 November 1996.

[12] Kenneth C. Yeager, "The MIPS R10000 Superscalar Microprocessor," IEEE Micro, vol. 16, no. 2, pp. 28-40, Apr., 1996.

[13] The Alpha 21264 Microprocessor - IEEE Micro Volume 19 , Issue 2 (March 1999) R. E. Kessler

[14] Farrell, J.A.; Fischer, T.C. Issue logic for a 600-MHz out-of-order execution microprocessor. Solid-State Circuits, IEEE Journal of Volume 33, Issue 5, May 1998 Page(s):707 – 712

[15] D. Bailey and B. Benschneider. Clocking design and analysis for a 600 MHz Alpha microprocessor. IEEE Journal Solid-State Circuits, 33(11):1627--1633, November 1998.

[16] “Power- and complexity-aware issue queue designs” Abella, J. Canal, R. Gonzalez, A. Dept. d'Arquitectura de Computadors, Univ. Politecnica de Catalunya, Barcelona, Spain; IEEE Micro Sept.-Oct. 2003 Volume: 23, Issue: 5 On page(s): 50-58

[17] VanderWiel, S. P. and Lilja, D. J. (1999b). Data prefetch mechanisms. To be published in ACM Computing Surveys

[18] Y. Guo, S. Chheda, I. Koren, C. M. Krishna, and C. A. Moritz. Energy-aware data prefetching for general-purpose programs. In Power-Aware Computer Systems(PACS'04), Micro-37 Dec. 2004.

[19] Yao Guo , Saurabh Chheda , Israel Koren , C. Mani Krishna , Csaba Andras Moritz, Energy Characterization of Hardware-Based Data Prefetching, Proceedings of the IEEE International Conference on Computer Design (ICCD'04), p.518-523, October 11-13, 2004

[20] Yao Guo, Mahmoud Ben Naser, Csaba Andras Moritz: PARE: a power-aware hardware data prefetching engine. ISLPED 2005: 339-344

[21] N. Azizi, A. Moshovos, and F. N. Najm. Low-leakage asymmetric-cell sram. In Proc. the 2002.

[22] Michael Zhang and Krste Asanovic. Highly associative caches for low-power processors. Kool Chips Workshop, 33rd International Symposium on Microarchitecture, 2000. 12

[23] Power Reduction Techniques For Microprocessor Systems; Vasanth Venkatachalam and Michael Franz; ACM Computing Surveys, Volume 37, Issue 3, 2005.

[24] G. Miñana, O. Garnica, J.I. Hidalgo, J. Lanchares, J.M. Colmenar. Adaptación de un simulador de potencia para UFs en procesadores de alto rendimiento. Jornadas de Paralelismo. Actas del Primer Congreso Español de Informática. (CEDI 2005). Granada (Spain). September 2005.

[25] Itanium 2 Processor Microarchitecture Overview. Don Soltis, Mark Gibson, Cameron McNairy. Hot Chips 14 August 2002

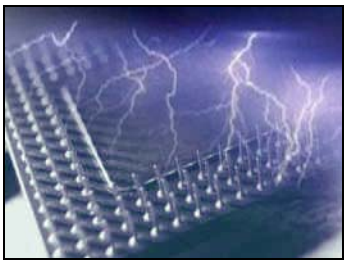
[26] Inside the Intel Itanium 2 Processor: an Itanium Processor Family member for balanced performance over a wide range of applications. A Hewlett Packard Technical White Paper July 2002

[27] Intel Itanium 2 Processor Hardware Developer’s Manual July 2002

[28] The High Bandwidth 256KB 2nd Level Cache on an Itanium Microprocessor. Tom Grutkowski, Reid Riedlinger. Intel Corporation and Hewlett Packard

[29] S. Naffziger et al., "The Implementation of the Itanium 2 Microprocessor," IEEE J. Solid-State Circuits, Nov. 2002, vol. 37, no. 11, pp. 1448-1460.

# Descripción de Wattch



Juan Antonio Victorio Ferrer

Proyecto Fin de Carrera

CPS- Universidad de Zaragoza

# Tabla de Contenido

# 1. Presentación

Una de las principales tareas de este proyecto ha sido estudiar en profundidad el funcionamiento de Wattch, para poder más adelante realizar cualquier tipo de mejora.

Wattch es un simulador de potencia/consumo programado en C, basado en Simplescalar. Puede ser descargado desde: http://www.eecs.harvard.edu/~dbrooks/wattch-form.html y el Report Oficial que le acompaña puede ser descargado desde: http://www.eecs.harvard.edu/~dbrooks/isca2000.ps.

Wattch nos permite simular la ejecución de un programa o traza de ejecución, del mismo modo que lo hace Simplescalar. Una vez terminada la simulación Wattch nos muestra las estadísticas calculadas normalmente por Simplescalar (numero de instrucciones simuladas, tiempo de simulación, número de saltos mal predichos, numero de referencias a memoria, tasas de fallo de las caches, etc.), más los consumos energéticos de todas las unidades que forman el procesador.

# 2. Funcionamiento interno

La manera en la que Wattch es capaz de calcular el consumo es la siguiente:

- Antes de la simulación, Wattch precalcula la potencia máxima (potencia de pico) de cada una de las unidades en función de parámetros de la arquitectura como por ejemplo tamaño de las caches, ancho de issue, tamaño del banco de registros, etc. Cabe destacar que calcula solamente la potencia dinámica y que por lo tanto el consumo estático del procesador debido a las corrientes de fuga no esta modelado.

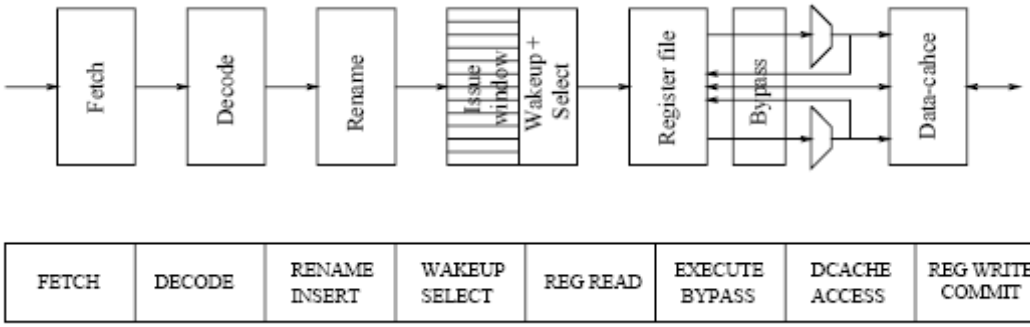
- Para cada ciclo de la simulación (equivalente a un ciclo de reloj en el procesador) Wattch mide en distintas partes del simulador el número de accesos (uso) y la actividad (proporción de 1’s y 0’s en los datos que circulan por el procesador) de todas las unidades que lo componen. Al final de cada ciclo utiliza esa medición, junto con el cálculo previo de potencia para calcular la energía consumida en ese ciclo.

- Por último, al final de la ejecución del programa, Wattch muestra las estadísticas de consumo al usuario.

# 3. Modelo de procesador

El modelo de procesador que modela Wattch es el mismo que el modelo de Simplescalar, es decir, un procesador fuera de orden, basado en estaciones de reserva RUU (las instrucciones guardan su información y también los operandos) ejecución especulativa de saltos, latencia de memoria fija.

El autor no ha modificado nada que relacionado con el núcleo del simulador. Sin embargo el modelo de unidades que hace se corresponde con un procesador distinto. En realidad el modelo parece sacado del artículo, ”Quantifying the Complexity of Superscalar Processors”, que el mismo cita como referencia en el artículo de Wattch.




Etapas segmentado del procesador descrito en el artículo ”Quantifying the Complexity of Superscalar Processors”

El segmentado que se presenta en este artículo es el indicado en la figura arriba. Que corresponde con un procesador algo más moderno, por ejemplo, incluyendo una etapa de renombre.

# 4. Tecnología

Wattch incluye un fichero en el que define sus parámetros tecnológicos como pueden ser tamaño de los transistores, capacidades de las puertas lógicas, etc. Estos parámetros son multiplicados por unos factores de escala propios para cada escala de integración.

Todos estos parámetros se definen mediante constantes y son preprocesados en tiempo de compilación del simulador, por lo tanto solamente se pueden variar recompilando el simulador.

Como ejemplo:

```
#define Wdecdrivep  (57.0 * LSCALE)
#define Wdecdriven  (40.0 * LSCALE)
#define Wdec3to8n   (14.4 * LSCALE)
…
```

Para cada escala se definen los factores de escala mediante constantes. Para cada posible escala tecnológica tendríamos un fragmento de código similar a este:

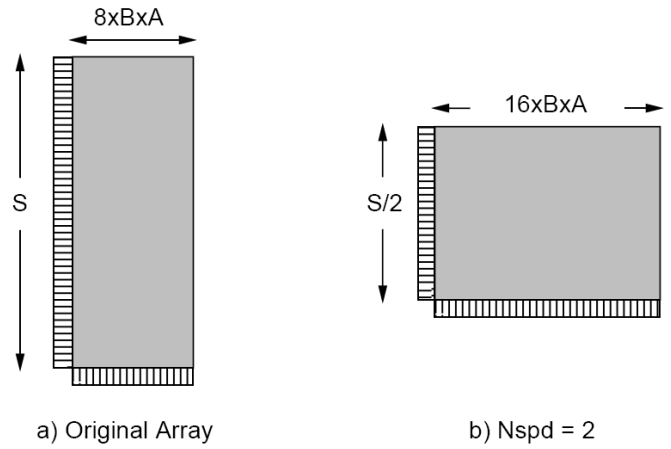
```
#elif defined(TECH_POINT18)
#define CSCALE (19.7172) /*wire capacitance scaling factor */
#define RSCALE (20.0000) /* wire resistance scaling factor */
#define LSCALE 0.2250    /* length (feature) scaling factor */
#define ASCALE (LSCALE*LSCALE)     /* area scaling factor */
#define VSCALE 0.4       /* voltage scaling factor */
#define VTSCALE 0.5046   /* threshold voltage scaling factor */
#define SSCALE 0.85 /* sense voltage scaling factor */
#define GEN_POWER_SCALE 1
#elif defined(TECH_POINT25)
```

# 5. CACTI

CACTI 1.0 es una herramienta que modela y calcula el tiempo de acceso para memorias cache. Fue publicada en 1994, y desde entonces han sido desarrolladas nuevas versiones hasta la versión actual 4.2.

Ya en la primera versión, que es la que utiliza Wattch, se asegura que las estimaciones del modelo están dentro del 10% de las estimaciones que haría H-Spice, por lo tanto se puede considerar que esta herramienta tiene una precisión suficiente.

Wattch utiliza CACTI para calcular la geometría de los bancos de cache (sub-banking), es decir, los valores `ndwl ndbl nspd` y `ntwl ntbl ntspd`. Los 3 primeros se refieren a los bloques de datos, y los siguientes al array de tags.




Muestra de una transformación de una memoria mediante el parámetro `Nspd`

`Ndwl * ndbl` nos indica el número de arrays en que se divide la cache (equivaldría al número de divisiones necesarias para optimizar la longitud de los wordlines y el número de divisiones necesarias para optimizar la longitud de los bitlines). `Nspd` es un factor que intenta optimizar la geometría de un array (intenta hacerla lo más cuadrada posible). Los otros tres parámetros significarían lo mismo, pero aplicado a los arrays de tags.

Wattch también utiliza unas funciones básicas de CACTI definidas en el fichero `time.c` que sirven para calcular la capacidad de los transistores empleados en sus modelos (anteriormente presentados). Estas funciones se usan dentro de otras funciones que define Wattch para el cálculo de la potencia de las líneas presentes en las memorias RAM y CAM. Las funciones son:

```
double gatecap(width,wirelength)
/* returns gate capacitance in Farads */
double gatecappass(width,wirelength)
/* returns gate capacitance in Farads */
double draincap(width,nchannel,stack)
/* returns drain cap in Farads */
```

Estas funciones, devuelven un valor que es función de los parámetros que obtienen como entrada y también de variables globales de CACTI.

Uno de las incorrecciones que hemos encontrado en el funcionamiento de Wattch es que dado que CACTI 1.0 esta diseñado para tecnología de 800 nm, los resultados de las llamadas de Wattch a CACTI no están calculados sobre la tecnología de Wattch (por defecto se compila a 350 nm) sino sobre tecnología de 800 nm.

# 6. Resultados

Wattch nos presenta al final de la ejecución los resultados de consumo. Para cada unidad, Wattch muestra 4 resultados distintos y para entender lo que significa debemos introducir una técnica muy utilizada a la hora de implementar procesadores, el clock-gating.

El clock-gating es una técnica que inhibe la señal de reloj a ciertas unidades funcionales que no van a ser utilizadas, con el propósito de que estas no conmuten y por lo tanto no disipen potencia dinámica ese ciclo.

De los 4 resultados que presenta Wattch, el primero de ellos considera que no se implementa clock-gating, todas las unidades reciben la señal de reloj en todos los ciclos, y por lo tanto todas ellas conmutan, consumiendo energía. El autor denomina este modelo de Clock Gating NCC.

El segundo de los resultados considera que se implementa clock-gating simple, en el que una unidad no recibe señal de reloj, si no va a ser utilizada en absoluto, por lo tanto, en ciclos que se de esta condición, esta unidad no consumirá energía. El autor denomina este modelo de Clock Gating CC1.

El tercero de los resultados correspondería con la implementación de un clock-gating más agresivo, en el que se considera que si solamente una porción de la unidad es utilizada, la potencia consumida es un porcentaje de la potencia máxima de esa unidad. Por ejemplo, para un banco de registros con cuatro puertos, si en un ciclo determinado solamente utilizamos 3, se considera que se ha consumido el 75% de la potencia máxima en ese ciclo, si solamente

utilizamos 1 puerto, consideraremos que se ha consumido el 25%. El autor denomina este modelo de Clock Gating CC2.

Por último Wattch presenta un 4º resultado, denominado CC3, que es exactamente igual que CC2, salvo que en el caso de que la porción de unidad utilizada sea el 0%, en cuyo caso se considera que existe un consumo mínimo que el autor estima en un 10% del consumo máximo de esa unidad.

## 7. Tipos de unidades modeladas

Los modelos que utiliza Wattch para cada una de las unidades pueden ser divididos en 4 grupos:

- Reloj: Modela el consumo de la red de distribución de la señal de reloj como la suma de tres factores: La capacidad de la línea, la capacidad de los buffers y la capacidad de carga de los registros. La capacidad de la línea se estima en base a una distribución del reloj en forma de H-tree a lo largo del procesador.

- Circuitos combinacionales: Para estructuras combinacionales que no pueden ser modelados como una estructura en array (RAM o CAM), Wattch debe utilizar un modelo específico, por ejemplo en el caso de las ALU's, cuyo consumo es una constante definida en el programa, o para la lógica de selección para las que calcula la potencia de un árbitro concreto y después la multiplica por el numero de árbitros por el ancho de issue.

- Estructuras en Array (RAM): Para estructuras que se pueden asemejar a una memoria RAM, Wattch define unas funciones que calculan la potencia, estas funciones toman como parámetros el numero de filas y columnas de la matriz, el número de puertos de lectura/escritura y si es o no una memoria cache (de cara a añadir amplificadores en la salida)

- Estructuras en Array (CAM): Para estructuras que se asemejen a una memoria CAM (Content Addressable Memory), de manera análoga, Wattch tiene funciones genéricas que calculan la potencia de una CAM en función del número de filas y columnas de la matriz y el numero de puertos de lectura/escritura.

Si quisiéramos por lo tanto añadir una unidad nueva dentro de Wattch, deberíamos asemejarla a una memoria RAM, memoria CAM o bien crearnos una función específica que calculara la potencia en función de los tamaños de la nueva unidad.

Cabe destacar que para ciertas unidades más sensibles a la actividad, Wattch divide la potencia modelada en dos, una que no es dependiente de la actividad (proporción de 0's / 1's en el resultado), y otra que si. Posteriormente, durante la ejecución del programa se cuenta el número de 1's y 0's en los resultados de esa unidad, consiguiendo una estimación más fina del consumo.

## 8. Modelo de unidades RAM

El consumo producido por una memoria RAM se puede dividir en 4 factores. El primero es la energía que gastamos en decodificar las direcciones. La salida de cada uno de estos decodificadores es una línea activa de entre todas las posibles líneas (tantas como direcciones o filas tenga la memoria)

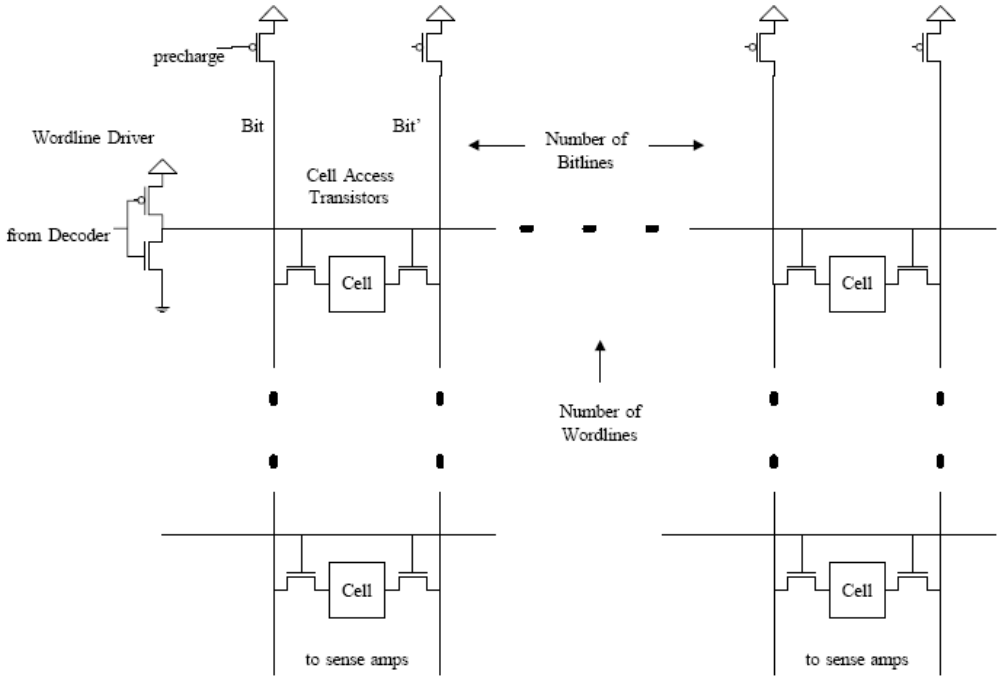
En Wattch para modelar la potencia de los decodificadores, hay definida una función:

```
double array_decoder_power(rows,cols,predeclength,rports,
wports,cache)
```

Los parámetros, tanto para esta, como para las siguientes funciones que van a ser descritas, son los siguientes:

`rows`: Número de direcciones o filas de la memoria.
`cols`: Número de bits por cada palabra de la memoria.
`rports`: Número de puertos de lectura.
`wports`: Número de puertos de lectura.
`cache`: Booleano con el que se indica si se trata de modelar una cache o un banco de registros. La diferencia es que las caches son modeladas como dual-ended y los bancos de registros como single-ended.

Para esta función `array_decode_power()` hay un parámetro `predeclength`, que curiosamente no se utiliza en la función.




En esta figura se muestra la estructura de una memoria RAM. Faltaría el decodificador previo a los "wordlines" y el amplificador al final de los "bitlines" de las caches.

El segundo factor que influye en el consumo de una memoria RAM es la potencia disipada en los "wordlines". Se tratan de líneas de metal relativamente largas que deben ser activadas para leer/escribir cada una de las palabras. Las líneas que serán activadas en las lecturas/escrituras vienen determinadas por la salida del decodificador mencionado antes. Además de la propia potencia disipada por la capacidad de la línea se deben añadir consumos generados por las capacidades de carga de los transistores a los que se conecta la línea.

La potencia que disipan se calcula en Wattch mediante la función:

```
double array_wordline_power(rows,cols,wordlinelength,rports,
wports,cache)
```

El parámetro wordlinelength sirve para indicarle a la función la longitud de la línea. Esta es calculada previamente en función del número de columnas o bit por palabra, el número de puertos y constantes dependientes de la technología como `RegCellWidth` y `BitlineSpacing`

| | |
|---|---|
| Regfile Wordline Capacitance = | $C_{diff}(WordLineDriver)$ $+ \ C_{gate}(CellAccess) * NumBitlines$ $+ \ C_{metal} * WordLineLength$ |

En esta ecuación se muestra la fórmula utilizada para el cálculo de la capacidad de las líneas Wordline en una memoria RAM

El tercer factor en el consumo de una memoria RAM es la potencia disipara en los bitlines. Estas son las líneas metálicas que atraviesan verticalmente la memoria, y van a parar a la salida de ésta, o si se trata de una memoria cache, a los amplificadores. Además de la capacidad de las líneas en si, en Wattch se cuentan además la capacidad del transistor de precarga que hay al principio de la línea, así como la capacidad de los transistores de acceso a celda, que se encuentran en cada una de las filas que atraviesa el bitline.

La potencia disipada por las líneas de wordline se calcula en Wattch mediante la función:

```
double array_bitline_power(rows,cols,bitlinelength,rports,
wports,cache)
```

Del mismo modo que con los wordlines, un parámetro que necesita la función es `bitlinelength,` que representa la longitud del bitline, y se calcula en función del número de filas o número de palabras, el número de puertos y constantes dependientes de la tecnología como `RegCellHeight` y `WordlineSpacing.`

| | |
|---|---|
| Regfile Bitline Capacitance = | $C_{diff}(PreCharge)$ $+ \ C_{diff}(CellAccess) * NumWdlines$ $+ \ C_{metal} * BLLength$ |

En esta ecuación se muestra la fórmula utilizada para el cálculo de

la capacidad de las líneas Bitline en una memoria RAM

Cabe destacar que la capacidad de estas líneas depende de si el dato leído es un 0 o un 1, ya que en un caso se produce descarga de la capacidad de la línea, y en el otro no. En Wattch, la potencia disipada por estas líneas es multiplicada por un factor de actividad, que mide la proporción entre 0's y 1's en los resultados. Mientras que en CACTI, se asume un factor de actividad de 0.5.

Por ultimo, y solamente para memorias cache, Wattch calcula la potencia disipada por los Amplificadores colocados a la salida de una memoria cache. Este cálculo se realiza mediante la función: `double senseamp_power(int cols)` que simplemente multiplica una constante por Vdd (voltaje de alimentación) por el número de amplificadores (número de columnas de la matriz).

## 9. Modelo de unidades CAM

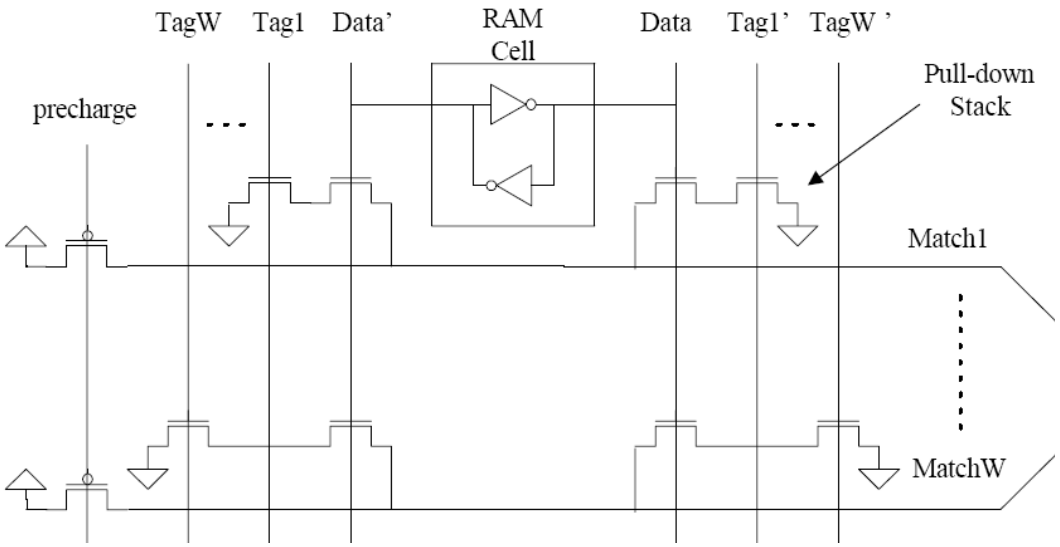
Las memorias CAM almacenan palabras en celdas dispuestas en forma de matriz, igual que con las memorias RAM, sin embargo, estas palabras no son direccionadas, sino que la búsqueda es por contenido. Si a la entrada de una memoria CAM ponemos una etiqueta, se activará como resultado una salida que indica que en esa fila de la matriz esta almacenado esa misma etiqueta.

Wattch utiliza dos funciones para calcular la potencia disipada por las memorias CAM. La primera de ellas calcula la potencia disipada por las líneas que llevan la etiqueta desde la entrada, pasando por todas las etiquetas de la matriz. La función es:

```
double cam_tagdrive(rows,cols,rports,wports)
```

| | |
|---|---|
| CAM Tagline Capacitance = | $C_{gate}(CompareEn) * NumberTags$ $+ \ C_{diff}(CompareDriver)$ $+ \ C_{metal} * TLLength$ |

Ecuación utilizada para calcular la capacidad de las líneas de etiquetas "Tagline" de una memoria CAM




En esta figura se puede ver la estructura interna de una memoria CAM.

La segunda función utilizada para calcular la potencia disipada por las memorias CAM, calcula la potencia que se disipa en las líneas que indican si se ha producido un acierto o no. Estas líneas se denominan “matchline”. En Wattch la potencia que disipan se calcula mediante la función:

```
double cam_tagmatch(rows,cols,rports,wports)
```

| | |
|---|---|
| CAM Matchline Capacitance = | $2 * C_{diff}(CompareEn) * TagSize$<br>$+ \ C_{diff}(MatchPreCharge)$<br>$+ \ C_{diff}(MatchOR)$<br>$+ \ C_{metal} * MLLength$ |

Ecuación utilizara para calcular la capacidad de las líneas de etiquetas Matchline

## 10. Unidades modeladas

En esta sección vamos a hacer un repaso por todas las unidades que modela Wattch y como las modela.

### *10.1. Renombre (y decodificación)*

Wattch modela la potencia del mecanismo estándar de renombre, en el que se incluye la tabla de traducción de registros, y la lógica de chequeo de dependencias. No se modela el consumo del hardware requerido para gestionar los registros físicos libres.

A pesar de que SimpleScalar no posee unidad de renombre, Wattch la modela. Además modela que para cada instrucción que pasa por la etapa de Dispatch, se realiza un acceso a la unidad de renombre.

El autor, en el fichero `power.c` considera la potencia disipada por renombre como una contribución de tres factores.

El primero de ellos es la potencia disipada por la RAT (Register Alias Table) que modela como una memoria RAM de `MD_NUM_IREGS` filas, `npreg_width` columnas, `2*ruu_decode_width` puertos de lectura y `ruu_decode_width` puertos de escritura.

El segundo factor que contribuye a la potencia de renombre es la potencia de DCL (Dependency Check Logic), que el modela como `(ruu_decode_width - 1) * ruu_decode_width` comparadores de `nvreg_width` bits.

Siendo:

```
nvreg_width = (int)ceil(logtwo((double)MD_NUM_IREGS));
npreg_width = (int)ceil(logtwo((double)RUU_size));
```

Tambien se añade a la potencia de renombre la potencia del decodificador de instrucciones, que modela como una simple RAM de `opcode_length` filas, una columna, un puerto de lectura y uno de escritura.

El autor considera también que como máximo se puede acceder `ruu_decode_width` veces por ciclo a la unidad de renombre.

## 10.2. Predictor de saltos

Wattch modela el consumo de un predictor de saltos convencional. El autor cuenta un acceso al predictor de saltos por cada instrucción que pasa por Commit o por WriteBack, en función de donde se actualice el predictor (es un parámetro de simulación). Sin embargo no cuenta un acceso al predictor cuando este es leído en las primeras etapas del segmentado.

El autor modela la potencia del predictor de saltos como la suma de las potencias de varias pequeñas unidades:

```
power->bpred_power = power->btb + power->local_predict + power->global_predict + power->chooser + power->ras;
```

Todas las componentes del predictor de saltos se modelan como memorias RAM del tamaño especificado al lanzar la simulación.

## 10.3. Ventana de Instrucciones

Wattch divide le consumo de la ventana de instrucciones en tres componentes: Lógica de despertado, lógica de selección. y Estaciones de reserva.

Para la lógica de despertado, se modela una memoria CAM de `RUU_size` filas, `npreg_width` columnas y `ruu_issue_width` puertos de lectura y otros tantos de escritura.

Igualmente, `nvreg_width = (int)ceil(logtwo((double)MD_NUM_IREGS));`

Se cuenta acceso a la lógica de despertado una vez por cada instrucción que inicia la ejecución (issue).

Existe una función definida explícitamente para calcular la potencia de la lógica de selección. Esta función calcula el número de árbitros necesarios (a partir de árbitros de 4 peticiones) para servir las peticiones de una ventana de ejecución de tamaño `RUU_size`. Después multiplica por el número de instrucciones que se pueden servir en un ciclo (`ruu_issue_width`) y por la potencia disipada por cada árbitro.

Se cuenta acceso a la lógica de selección una vez por cada instrucción que pasa por la etapa de Issue.

Las estaciones de reserva se modelan como una memoria RAM de `RUU_size` filas por `data_width` columnas, con `2*ruu_issue_width` puertos de lectura y `ruu_issue_width` puertos de escritura.

Se cuenta 1 acceso a las estaciones de reserva por cada instrucción que pasa por writeback y 2 accesos por casa instrucción que lanzamos en la etapa de issue. En la etapa de dispatch, se cuentan 2 lecturas si `OPERANDS_READY(rs)` o bien 1 si `ONE_OPERANDS_READY(rs).`

## *10.4. Cola Load Store (LSQ)*

El autor divide el consumo de la Cola Load Store en dos componentes: Un banco de registros (RAM) más la lógica de despertado (CAM), `lsq_rs_power` y `lsq_wakeup_power` respectivamente. Ambas con `LSQ_size` filas, `data_width` columnas y `res_memport` puertos de lectura y otros tantos de escritura.

El acceso al banco de registros se cuenta en la etapa de Issue, para las operaciones tipo Load y Store

Mientras que la lógica de despertado se cuenta en la misma etapa, pero solamente para instrucciones del tipo Load

## *10.5. Banco de registros*

El banco de registros se modela en Wattch como una memoria RAM de `MD_NUM_IREGS` registros de `data_width` bits cada uno, `2*ruu_issue_width` puertos de lectura y `ruu_issue_width` puertos de escritura.

Wattch cuenta accesos al banco de registros en Commit si la instrucción es distinta de Store y en Dispatch una vez por cada operando que no se lea de una entrada que ya esta en la RUU.

Además se cuenta el número de 0’s en el valor del operando leído/escrito ya que la potencia del banco de registros depende del factor de actividad.

## 10.6. Cache de Instrucciones

Se modela por un lado el TLB de instrucciones y la cache en si.

El TLB de instrucciones se modela como una CAM + RAM, de un puerto de lectura y otro de escritura.Con `itlb->nsets` filas y `va_size – log2 (itlb->bsize)` columnas.

A su vez, la cache en si se modela como dos RAM, una de etiquetas y otra de datos. Todas estas estructuras con un puerto de lectura y otro de escritura.

Se utiliza CACTI para calcular la mejor opción de sub-banking (cambiar la geometría de un array de datos para que las longitudes de los wordlines y los bitlines sean lo más parecidas posibles).

En fetch, por cada instrucción decodificada se cuenta un acceso a la cache de instrucciones, pero a la hora de calcular potencia solamente se toma en cuenta uno, ya que se supone que en cada ciclo solo se hace fetch de una línea.

## 10.7. Cache de Datos

Al igual que en la cache de instrucciones, se modela TLB de datos y la cache en si.

El TLB de datos se modela como una CAM + RAM, de un puerto de lectura y otro de escritura. Con `dtlb->nsets` filas y `va_size – log2 (dtlb->bsize)` columnas.

La cache de datos en si se modela como dos RAM, una para las etiquetas y otra para los datos. Ambas con 1 puerto de lectura y otro de escritura. Del mismo modo que con la cache de instrucciones, se utiliza CACTI para el subbanking.

Se cuenta acceso a cache de datos en Commit para Stores y en Issue para Loads. Al final del ciclo se considera que se pueden hacer `res_memport`.

## 10.8. Cache Nivel 2

La cache de nivel 2 se modela como dos RAM, una para TAGs y otra para datos. Ambas con un puerto de lectura y otro de escritura. Igualmente, se utiliza CACTI para calcular la mejor

opción de sub-banking. Sin embargo, para la cache de nivel 2 solamente se considera la potencia de uno solo de los sub-bancos, ya que el autor presupone que el resto pueden estar desconectados cuando se produce un acceso a esta cache.

Se cuenta acceso a cache de nivel 2 en las rutinas de "miss handler" de SimpleScalar tanto de cache de instrucciones como de datos de nivel 1. (Se supone cache de nivel 2 compartida).

## 10.9. Alu's

La potencia que disipa una alu en Wattch es modelada como una constante. Por lo tanto la potencia de las alus en Wattch se modelan como:

```
res_ialu * I_ADD
```

(Numero de alu's enteras por la potencia de una alu entera)

```
res_fpalu * F_ADD
```

(Numero de alu's coma flotante por la potencia de una alu coma flotante)

En Wattch original el consumo en las ALU's no distingue el tipo de operación. Sin embargo existe una modificación de Wattch llamada FU-Wattch, que mejora el modelo de las ALU's cuya publicación se encuentra en la siguiente dirección:

http://atc.dacya.ucm.es/atc/descargar.php?file=cedi2005guada.pdf.

El código fuente de esta modificación no se encuentra disponible.

El uso de las ALU's se cuenta en la etapa de Issue, una vez por cada instrucción que no sea Load/Store y que necesite una unidad funcional.

## 10.10. Resultbus

Para calcular la potencia disipada por el Result Bus (Red de cortocircuitos) Wattch tiene definida una función Ad-hoc que se llama:

`double compute_resultbus_power()`

Wattch cuenta el uso del resultbus por cada instrucción que pasa por writeback. Se considera que como máximo pueden producirse `ruu_issue_width` accesos a la red de cortocircuitos. Además Wattch mide el factor de actividad de los resultados escritos en el resultbus mediante llamadas a la funcion `pop_count()`.

### *10.11. Clock*

Para la red de distribución de la señal de reloj, Wattch tambien tiene definida una funcion Ad-hoc que calcula la potencia máxima que disipa. Se identifican 3 fuentes de consumo de potencia relacionadas con el reloj: Global Clock Metal Lines, Global Clock Buffers, Clock Loading (Lineas que distribuyen la señal de reloj, bufferes que ayudan a distribuir la señal correctamente y capacidad por registros conectados al reloj)

La longitud de las líneas la calcula suponiendo que se distribuyen en forma de H-tree.

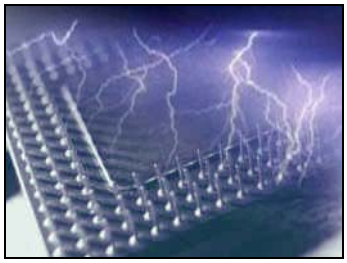
Para calcular la capacidad de los registros conectados a la señal de reloj, lo hace estimando el número de etapas por el número de bits que tiene cada registro inter-etapa, a lo que le suma un 50% debido a señales de control.

A la hora de calcular la energía por la distribución del reloj, Wattch supone que la proporción de ahorro en consumo por clock gating del reloj es la misma que la del total del resto de unidades del procesador. Por eso para calcular el CC1, CC2 y CC3 utiliza la proporción del total del resto de unidades, es decir:

clock_cc1 = clock_ncc * total_energy_cc1 / total_energy_ncc e igualmente para cc2 y cc3. Siendo total_energy la suma de los consumos del resto de unidades.

# Vatios 1.0

# User Manual



Juan Antonio Victorio Ferrer

University of Zaragoza

# Table of Contents

# 1. Description

Vatios is the Spanish word for "*Watts*". In this context, Vatios is a set of tools for computer architects that was developed as a part of a project at the Computer Architecture Group (gaZ) at the University of Zaragoza http://webdiis.unizar.es/gaz/. Our start point was Wattch, developed by David Brooks http://www.eecs.harvard.edu/~dbrooks/

Wattch is a tool based on SimpleScalar that simulates the execution of a program or an execution-trace and, at the end of the simulation, gives power and energy consumption of the different units that make up the processor.

Vatios is a set of tools, written in C that can be used to get power and energy consumption values from super-scalar processors. It's based on Wattch and not only provides the same functionality as, but also gives more flexibility and new interesting features.

Vatios is compounded of `sim-vatios`, which is the simulator similar to Wattch, and `power-vatios`, which is a tool to calculate power/consumption from previous simulations.

# 2. Differences

Wattch has been a great development because a computer architect can take design-decisions based on power and energy predictions made by Wattch at early development stages. With Wattch, you don't need a detailed circuit level description of the processor to get the energy consumption prediction.

Wattch's main problem is its lack of flexibility. All the technology parameters used during the simulation are directly written in the code, and the user is not able to modify them without re-compiling the tool. We have written the Vatios tool set in order to avoid those problems and create a more flexible and generic tool.

The Vatios' main simulator `sim-vatios` uses a more efficient algorithm that saves some floating-point operations every cycle of the simulation, calculates the Energy prediction as well and allows the user to dump in a file use/access distribution statistics of every processor's unit, and later, reuse those statistics to generate new predictions for different tech points or power models.

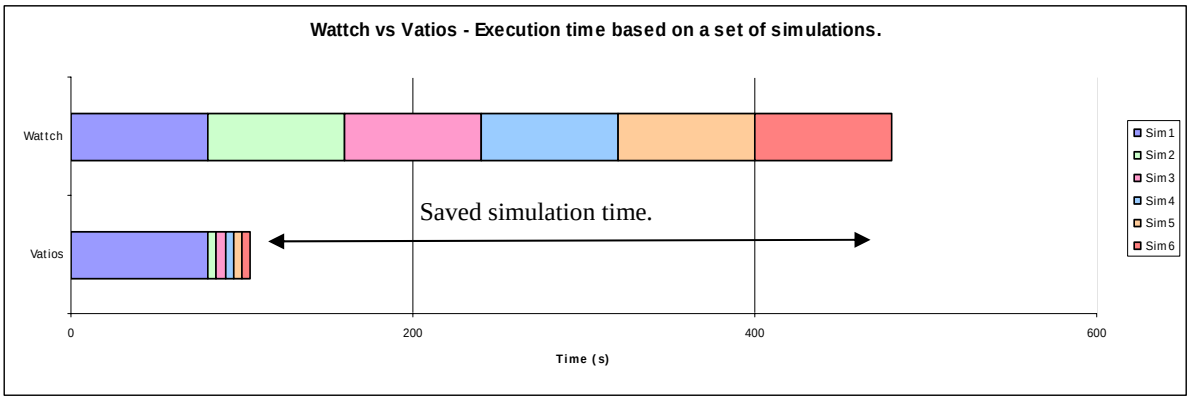
Instead of counting number of access and at the end of the cycle, generating the power/energy prediction every simulation's cycle, `sim-vatios` counts the number of accesses and generates use/access distributions statistics. With those statistics `sim-vatios` calculates the energy/power predictions at the end of the simulation, not at the end of the cycle. Moreover, this allows us to dump those statistics to a file and reuse them with `power-vatios`.

We have developed a program named `power-vatios` that with a dumped file, that includes the unit's architectural parameters and the usage statistics, can generate the same power and energy predictions that Wattch would generate without re-compiling the simulator or re-simulating the temporal execution.

We have also updated the CACTI 1.0 tool to the CACTI 4.2 tool.

## 3. Simulation Speedup

The main advantage is that this program allows you to change different technology parameters as well as the implementation of the power models (the algorithm used to calculate the peak power of a unit based on architectural parameters) and generate new energy/power predictions without re-simulating the program or execution trace if the unit's timing and use don't change.




Let's consider this example: We have to simulate an execution trace, with 2 different power models and each of these, in 3 different technology points.

With Wattch we have to simulate the execution:

3 technology points * 2 different power models = 6 times.

Moreover, we are not considering the compilation time, because with Wattch, each time that we are changing the technology point, we have to compile the simulator.

With Vatios we have to simulate one full temporal execution with `sim-vatios`. Then, since we have the use/access statistical distribution in a dumped file, and assuming that the timing of the simulation doesn't change if we move to another technology point, we have to execute `power-vatios` 5 extra times to calculate the energy/power prediction without re-simulating the full temporal execution. So, we have saved 5 unnecessary full temporal simulations, what depending of the length of the trace, can mean several hours.

## 4. Requisites

We have developed the Vatios tool-set in C under Linux on a x86 machine, and we don't use any special library, so the requisites are:

The Vatios tar.gz file
The GCC compiler
The Make tool
The standard library

You can download the Vatios tar.gz file from http://webdiis.unizar.es/gaz/vatios/ and in the case that you don't have any of the other tools, you should install or tell the administrator to install them.

Vatios is based on SimpleScalar, so there would be no difficult to compile it on any system that can use SimpleScalar.

## 5. Compilation

If you meet the requisites, you should extract the tar.gz file to your selected directory and then access the directory and type `make` in the command like. Everything else should be done automatically. This process will generate two executable files `sim-vatios` and `power-vatios`.

# 6. Use `sim-vatios`

First of all, you should simulate the executable or the execution trace. You have to do this the same way you would do it with SimpleScalar or with Wattch. This time the name of the executable will be `sim-vatios`. But there are some parameters that you have to take care of.

## *6.1. The –dumpconfig parameter*

The first parameter you should take care of is `-dumpconfig myDumpedFile`. This option already appears in SimpleScalar, but, if you are using it with `sim-vatios`, it will dump not only the config (architectural parameters used by the simulation like cache sizes, pipe-stages widths, etc…), but also the statistical information (use/activity of ech unit) needed to calculate the energy/power prediction.

If you don't specify the `–dumpconfig` option, the execution of `sim-vatios` will be similar to a normal Wattch execution.

## *6.2. The –tech parameter*

To calculate the energy/power prediction, the program needs the technology point you are simulating at, and the energy model you are using for each unit.

Therefore, we have defined the parameter `-tech` which defines the technology point. Available tech points are: 800, 400, 350, 250, 180 and 100 in nm. If you specify an unavailable tech point, the simulator will report an error.

## *6.3. The –freq parameter*

Wattch has been developed simulating a 600Mhz CPU. All the power is calculated at this clock-speed. We have programmed Vatios considering the CPU speed as a parameter. By default, if you don't specify this parameter, it will be 600Mhz, as in Wattch, but you can specify the speed as an integer number in Mhz. For example, for a 2Ghz CPU you have to add this option: `-freq 2000`

## *6.4. Power Models*

An energy model is an algorithm that calculates the peak power of a unit. Wattch defines only one power model for each unit. With Vatios there are several power models for each unit, and they are identified by numbers. By default, this version of Vatios runs with the same power models as Wattch (power model 1)

### 6.4.1. The power model #0.

but we have added the 0 power model, that means that this unit consumes 0 power/energy. You can use the 0 power model in the case that you want to ignore some units in your predictions. Units modelled with the 0 power model will consume no energy, and therefore, they will have no influence in the total energy/power predictions.

To specify the power model you are going to use for a unit (only if it's not the default), you have to invoke `sim-vatios` with and option that looks like:

```
-unitName:model powerModelNumber
```

For example:

```
-rename:model 0 -bpred:model 0
```

### 6.4.2. The power model #2 (User defined peak power of the unit).

In Vatios, if you want, you can't directly specify the peak power of a Unit. This can be done using the power model 2. In this case you will have to add an additional parameter (the power this unit consumes).

As an example:

```
-rename:model 2 –rename:power 2.5
```

This line means that want to directly specify the peak power for the rename unit (model 2) and that this unit consumes 2.5 Watts.

### 6.4.3. The power model #3

In sim-vatios, we have updated the CACTI 1.0 tool to the CACTI 4.2 tool. This version of CACTI is able to calculate not only the subbanking, but also the power of a cache-like unit.

We have added a new power model (model 3) that uses CACTI 4.2 to calculate the peak power of the Instruction Cache (and Intruction TLB), Data Cache (and Data TLB), Level 2 Data Cache and Regfile.

## *6.5. Names of the Units*

The names of the units are specified at `vatios_strings.h` and by default are these:

```
-rename
-bpred
-regfile
-icache
-dcache
-dcache2
-ialu
-falu
-resultbus
-window_preg
-window_selection
-window_wakeup
-lsq_wakeup
-lsq_preg
```

By default, if you don't specify the power model, the power model 1 will be used (the same as Wattch)

If you need to simulate with other tech points, or you need to add new power models to Vatios, please refer to the "Vatios Modification Manual".

# 7. Use `power-vatios`

Imagine that you want to simulate the energy consumption of one processor in several technologies or using several power models. At first glance, it seems unnecessary to simulate the execution every time, if the timing is going to be the same.

If you have already simulated one execution and you have dumped a file with the use/access information, you can use this file to calculate power/energy predictions without re-simulating the entire execution again. Starting from a use/activity file dumped with `sim-vatios`, you only have to invoke `power-vatios` with the appropriate arguments.

The first argument you have to specify is the name of the dumped file. This time with the option `-config myDumpedFile`

You also have to specify the tech point, frequency and the power models. This is done the same way as with `sim-vatios`.

# 8. Results

If you are not familiar with Wattch, it displays 4 kinds of results, depending on the clock gating that simulates. It displays the peak power of the units as well.

The first one (NCC) simulates no clock gating, the second one (CC1) simulates simple clock gating, the third one (CC2) simulates aggressive ideal clock gating, ant the last one (CC3) simulates aggressive non-ideal clock gating. If you need a more detailed explanation of these results, please read the original Wattch paper.

With Vatios we calculate the same results as with Wattch, the only difference is that we cannot detect the cycle when the energy consumption is maximum. We have also changed the way that we calculate the clock energy. It’s important to note that results with Vatios will be different from results with Wattch, since the Cacti tool has been updated to version 4.2 and this tool is used to get the results.

With Wattch:

Clock energy CCX = Clock energy NCC * Total Energy CCX / Total Energy NCC

Where Total Energy is the energy dissipated by the rest of the units that cycle.

With Vatios, since with our system we cannot calculate the energy in a specific cycle, we use this formula but taking the total energy of the whole simulation.

Here can be seen some examples of the results generated by Vatios:

```
#========================================#
#VATIOS - Power/energy calculator tool V1.0#
#========================================#
Rename Accesses Power AF Independent: 0
Rename Accesses Energy NCC: 0
Rename Accesses Energy CC1: 0
Rename Accesses Energy CC2: 0
Rename Accesses Energy CC3: 0
Bpred Accesses Power AF Independent: 18.6886
Bpred Accesses Energy NCC: 973957
Bpred Accesses Energy CC1: 122625
Bpred Accesses Energy CC2: 69007.7
Bpred Accesses Energy CC3: 154168
Regfile Accesses Power AF Independent: 0.677213
Regfile Accesses Power AF Dependent: 7.49981
Regfile Accesses Energy NCC: 426145
Regfile Accesses Energy CC1: 181133
Regfile Accesses Energy CC2: 43714.1
Regfile Accesses Energy CC3: 65263.8
```

# 9. Examples

In this section we are going to comment a pair of invocations to Vatios in order to help understanding how Vatios works.

## *9.1. Example 1*

With this invocation

```
./sim-vatios –tech 400 tests-alpha/eio/test-math.eio
```

we are simulating the test-math trace, without dumping the access statistics to a file, calculating and displaying the energy/power predictions at the end of the simulation, the prediction is calculated for a tech point of 400 nm , with the same power models as Wattch (default power models).

## *9.2. Example 2*

With this invocation

```
./sim-vatios -dumpconfig myFile.txt tests-alpha/eio/test-
math.eio
```

We simulate the test-math trace, dumping the access statistics to “myFile.txt”, and calculating the energy/power prediction for a tech point of 350 nm (default tech point) and the same power models as Wattch (default power models).

With this invocation

```
./power-vatios -config myFile.txt
```

we are calculating the power/energy predictions for the architecture and access statistics saved “in myFile.txt” for a tech point of 350 nm (default tech point) and the same power models as Wattch (default power models).

With this invocation

```
./power-vatios -config myFile.txt -tech 400
```

We are specifying that the tech point we want to simulate is 400, not the default 350.

With this invocation

```
./power-vatios -config myFile.txt -rename:model 0
```

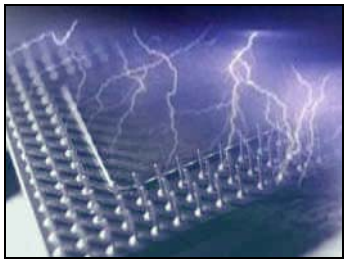
We are specifying that we want to ignore (power model 0) the rename energy/power in this simulation.

# 10. Further development

We hope to further develop Vatios, add new power models and technology points. However, as it's free software, you can improve it as you want. You can also send your request by email.

# Vatios 1.0

# Programmer's Manual



Juan Antonio Victorio Ferrer
University of Zaragoza


# Table of contents

# 1. Presentation

Vatios is a free simulator, written in C that can be used to get power and energy consumption values from super-scalar processors. It implements the same architecture as Wattch and not only provides the same functionality, but also gives more flexibility and new interesting features.

Vatios has been developed trying to allow the advanced-user to modify it with few effort. In order to do that, it's recommended to be familiar with the SimpleScalar simulator.

As Vatios and Wattch are based on SimpleScalar, if you have a modified simulator based on SimpleScalar you can easily add power/energy prediction, only including some headers, their correspondent files in the directory and some code in your simulator.

Moreover, if you have a simulator that isn't based on SimpleScalar, you can generate a file with the same structure and required information as the dump files that `sim-vatios` generates and then, you can process it with `power-vatios`, calculating the predictions.

This manual has been written to help you understanding how Vatios works and will guide you if you try to modify Vatios. If you are not familiar with the simulator or you haven't read the user manual, we strongly recommend doing that first.

Basically, there are 4 ways you would like to modify Vatios.

- Add another technology point.
- Add a new power model for a unit (a new way of calculating the peak power of a unit)
- Add a new unit, with its power model.
- Add the Vatios library to an existing SimpleScalar based simulator.
- Use the Vatios library to calculate power/energy based on the statistics generated by a non-SimpleScalar based simulator.

# 2. Simulator Software Architecture

This simulator is based on SimpleScalar, thus it contains the same files as SimpleScalar, but there are 2 files that have been modified:
- `options.c` has been slightly modified to allow dump use/access info.
- `sim-outorder.c` has been renamed to `sim-vatios.c` and we have added the code to measure the use of the units and dump that use to a file.

We have replaced the CACTI 1.0 tool with CACTI 4.2, therefore, you will find a directory called `cacti4_2.` We use CACTI to calculate subbanking of cache memories and to directly calculate the power of cache-like arrays.

We have created a new executable `power-vatios` created from `power-vatios.c` that uses files dumped by `sim-vatios` to calculate power/energy consumption (allowing the user to modify power models, frequency, technology…)

We have created `access_vatios.c` and `access_vatios.h`. In these files we define the number of units that the processor is made of, and we define functions to count the use of units. These functions are used by `sim-vatios.c`

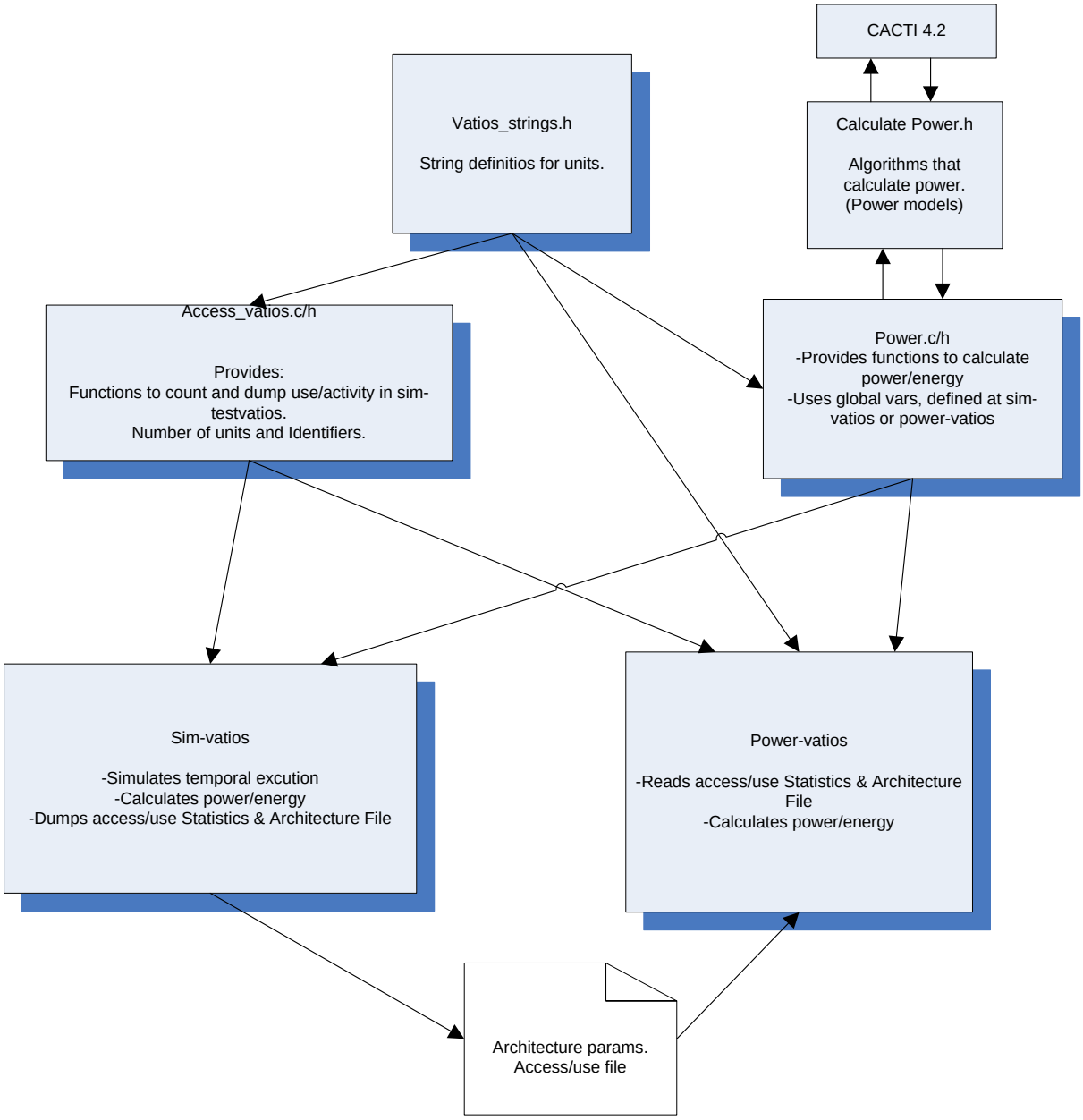
In `vatios_strings.h` we have written the strings used to refer to the units and used to display information to the user.

At the file `calculatePower.h` we have written the code (power models) to calculate the peak power of a unit, which will be escalated to calculate the actual energy consumption. In this file we call CACTI when necessary. This file is included in `power.c`.

In `power.c` and `power.h` you will find generic functions to calculate the power of a unit. You will find the calculate function, which calculates and displays all the energy/power results and is called by `power-vatios` and by `sim-vatios`.

Once you have compiled the simulator, 2 executable files will be generated: `sim-vatios` and `power-vatios.`

This diagram shows the simulator software architecture:



# 3. Adding another technology point

A technology point is the definition of some parameters determined by your fabrication process. Vatios comes with some technology point defined, but if you want to make simulations in a technology point not defined in Vatios, you can add the information needed to simulate in that point.

Wattch's author developed some functions to calculate the power of several kinds of structures (for example RAM, CAM…). These functions use some variables, whose initial values depend on some parameters: CSCALE, RSCALE, LSCALE, ASCALE, VSCALE, VTSCALE, SSCALE, GEN_POWER_SCALE, which are factors relative to the 0.80um fabrication process that represent the wire capacitance, wire resistance, length , area, voltage, threshold voltage, sense voltage and power.

So, let's see how we would add a new technology point:

In the file `power.c` there is a function called:

```
void init_vatios_tech_params(double techPoint)
```

At the beginning of this function, there's a `switch` clause. You have to add your technology point there. As an example, you can see how the tech point for 90nm is defined (Numbers are not real!). The 8 scaling factors you have to define for your tech point are relative to the 800nm fabrication process.

```
case 180:
     CSCALE    =(19.7172)/* wire capacitance scaling factor */;
     RSCALE    =(20.0000)/* wire resistance scaling factor */;
     LSCALE    =0.2250   /* length (feature) scaling factor */;
     ASCALE    =(LSCALE*LSCALE)    /* area scaling factor */;
     VSCALE    =0.4      /* voltage scaling factor */;
     VTSCALE   =0.5046   /*threshold voltage scaling factor */;
     SSCALE    =0.85     /* sense voltage scaling factor */;
     GEN_POWER_SCALE = 1;
     break;
case 90:
     CSCALE    =(35.232)/* wire capacitance scaling factor */;
     RSCALE    =(37.5000)/* wire resistance scaling factor */;
```

```
     LSCALE    =0.1150   /* length (feature) scaling factor */;
     ASCALE    =(LSCALE*LSCALE)    /* area scaling factor */;
     VSCALE    =0.3      /* voltage scaling factor */;
     VTSCALE   =0.5046   /*threshold voltage scaling factor */;
     SSCALE    =0.64     /* sense voltage scaling factor */;
     GEN_POWER_SCALE = 1;
     break;
case 400:
…
…
```

## 4. Adding new power models

First of all we want to introduce what a power model is. In Vatios, a power model is simply an algorithm that calculates a unit's peak power. For one unit, can be defined several power models, for example, we can have a cache and two power models, the first one that uses the functions defined by Wattch, and the second one that uses the latest version of CACTI.

When you are using your simulator, you can select the model for the unit introducing this parameter in the command line:

`-unit_name:model power_model (Example: –dcache:model 3)`

You may want to add new power models to improve or to give different choices when calculating the energy of a unit. The power model calculates the power of the unit and returns a `double` value.

First of all, you have to write the C code that calculates the maximum power that this unit can consume in one cycle. There's a file in the main program directory whose name is `calculatePower.h`. For every unit there's a switch instruction and at least "case 0:" and "case 1:" and "case 2". What you have to do is to add another "case #:"where # is the number you are going to use for this model.
Let's see an example:

This would be the code after adding my new model.

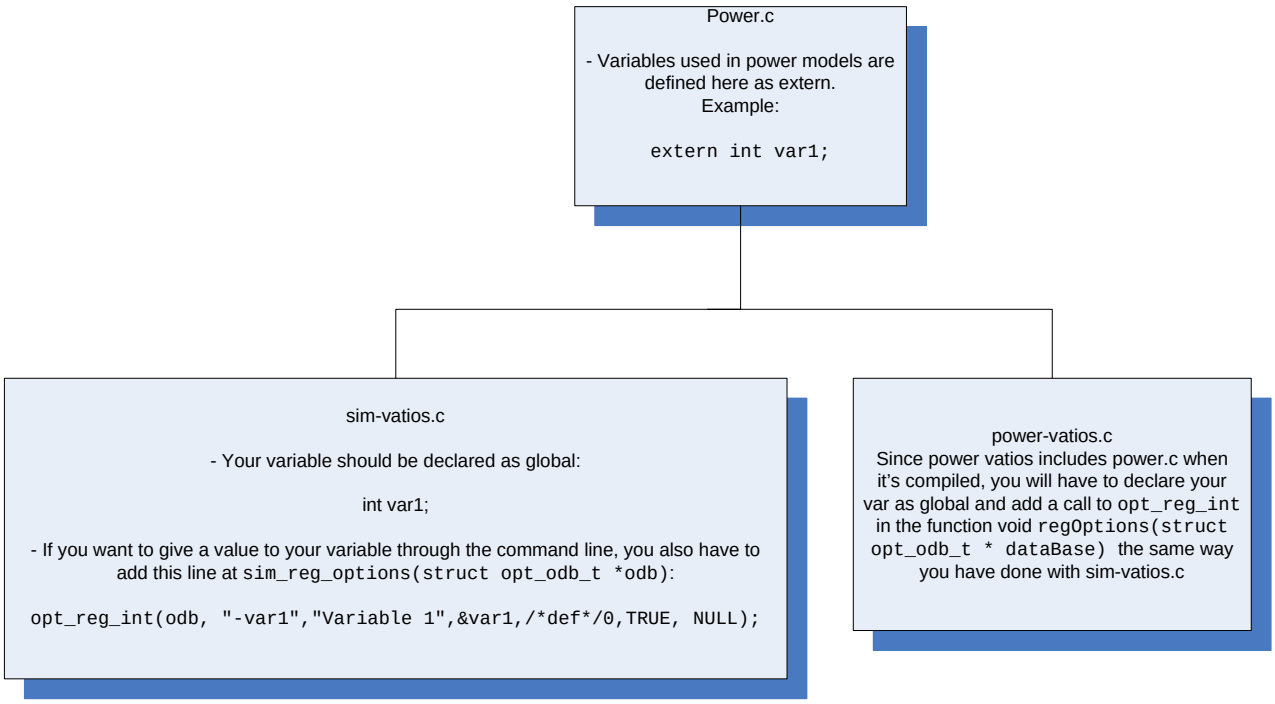
```
switch(energyModels[RENAME]){
     case 0: /*Don't' count this unit*/
          power.powerAFIndependent[RENAME] = 0.0;
          break;
     case 1:/* Default Wattch model*/
          power.powerAFIndependent[RENAME] = rat_power +
          dcl_power + inst_decoder_power;
          break;
     case 2:
          power.powerAFIndependent[RENAME] =
               userPowerAFIndependent[RENAME];
          break;
     case 3:
          power.powerAFIndependent[RENAME] = myNewRenamePower();
          break;

     default:
          fprintf(stderr,"Invalid energyModel for %s\n",
          varUnitNames[RENAME]);
          exit(-1);
}
```

Inside your algorithm you will need some architectural parameters, for example, if you are modelling the power of a cache, you will probably need the cache size, the cache assoc, the block size, of if you are modelling the ALU unit, you will probably need the number of ALUs.

If the parameter you need is present in another power model, you will be able to read the value and that would be enough (and you don't have to read the rest of this chapter), but if you creating a more detailed power model, it could be possible that you need a new parameter. Then you have to understand how architectural parameters are defined and used in Vatios.

This is the variable architecture in `sim-vatios`




Now, we are going to give a more detailed explanation of this.

Due to constraint in the options.c/h library, the architectural variables should be declared as global variables in `sim-vatios.c` and you have to register them, like other parameters in the function:

```
void sim_reg_options(struct opt_odb_t *odb)
```

Make sure that the `opt_reg_XXXXX` call has the print parameter set to TRUE, because otherwise, it won't be dumped to the config file. Note: `XXXX` can be `double, uint, int, string`,… If you need the full list of functions, please read `options.h`

You have to do the same at `power-vatios.c`. You have to declare the variables as global in a section where you can find the comment:

```
/****************** GLOBAL VARIABLE DECLARATIONS *************/

…
…
int opcode_length = 8;
int inst_length = 32;
int ruu_issue_width = 4;
int RUU_size = 8;
int myNewVar = 10;
…
…
```

You also have to register the variables in a function called:

```
void regOptions(struct opt_odb_t * dataBase)

{

  opt_reg_int(dataBase, "-myNewVar ","My new Var",&myNewVar
,MY_NEW_VAR_DEFAULT_VAL,/* print */TRUE, /* format */NULL);
```

Be careful, there are a lot of `opt_reg_XXXXX` functions, so use the appropriate one depending on the type of your new variable.
Finally, you have to declare your variable as extern at the beginning of the power.c file.

```
…
…
extern int data_width;
extern int res_ialu;
extern int res_fpalu;
extern int res_memport;
extern int myNewVar;
…
…
```

That's all. If you have some trouble, try to understand the compiler messages and how are defined other variables used in other power models. One of the easiest power models is the one for the IALU, you can follow this unit's model as an example. This model uses a parameter called `res_ialu`.

# 5. Adding a new unit, with its power model.

Adding a new unit means that you are going to model a processor with another piece of hardware, for example, we can think of adding a L3 cache structure. First of all, we have to modify our simulator to include this L3 cache structure. If we only do this, this would be transparent for Vatios. Therefore, we also have to tell Vatios that we are going to calculate the power/energy for this unit, and how we are going to do this.

We have divided this process in these steps:

- Declaring the unit
- Adding auxiliary strings
- Counting the number of accesses
- Creating the power models
- Adding the call to the function that calculates power/energy.

## *5.1. Declaring the Unit*

First of all, you will have to modify the `access_vatios.h` file. You will have to increment the `NUM_UNITS` constant, you will also have to add a new identifier in the first `enum` for example:

```
#define NUM_UNITS 15
…,LSQ_WAKEUP,LSQ_PREG,MY_UNIT};
```

The name used here is not relevant. Be careful to use an unused identifier.

## *5.2. Adding auxiliary strings*

Once you have done that, you will have to add new strings for your unit at the file `vatios_strings.h`, this strings will be used by the user to refer to this unit, and by the simulator to display information.

```
char * varUnitNames[] = {
"-rename",
"-bpred",
…
…
```

```
"-lsq_wakeup",
"-lsq_preg",
"-my_unit"
};

char * comentars[] = {
"Rename Accesses",
"Bpred Accesses",
…
…
"Lsq wakeup Accesses",
"Lsq preg Accesses",
"My unit Accesses"
};
```

Attention: The order of these strings IS relevant. It should match with the order of the units declared in the enum at the previous section.

### *5.3. Counting the number of accesses*

Wattch (and `sim-vatios`) models the Clock-gating technique to calculate energy consumption. The different clock-gating techniques are explained at Wattch's Report. To use this technique to calculate energy consumption, we need to know how many times one unit is accessed. Therefore, during the temporal execution of the simulator, we have to count the number of accesses. Specifically, we have to count the number of times that one unit is used every cycle, and then, increment one position of a vector.

Our goal is to create a vector for each unit, which has this structure:

Number of cycles that this unit has been accessed …

`[0 times, 1 time, 2 times, …]`

If you think that your unit is going to be used 30 o more times per cycle, please increment this constant defined at `access_vatios.h`:

```
#define MAX_ACCESS_CYCLE 30
```

To achieve this goal these steps are necessary:

There are some things you have to add at `access_vatios.c`

First of all, as global variables:

```
…
counter_t ialu_access;
```

```
counter_t falu_access;
counter_t resultbus_access;
counter_t my_unit_access;
```

These variables should be also declared as `extern` in the `sim-vatios.c` file, as you are going to modify them in that program to count accesses.

```
At the function clear_access_stats()
…
…
  ialu_access=0;
  falu_access=0;
  resultbus_access=0;
  my_unit_access=0;
```

`clear_access_stats` is called at the beginning of every cycle.

You also have to add one line at the function:

```
void update_access_stats(access_data_t * access_data){

…

access_data->accesses[LSQ_PREG][lsq_preg_access]++;
access_data->accesses[MY_UNIT][my_unit_access]++;
```

This function is called at the end of every cycle, and increments the correspondent position of the vector (previously explained).

You will have to count the accesses to your unit, you will have to know where your unit is used (at file `sim-vatios` or the name of your simulator main file) and every time you use your unit you have increment the correspondent counter:

```
my_unit_access++;
```

## 5.4. Creating the power models

Then, you have to create a section at the file `calculatePower.h` that calculates at least one power model for the new unit. If you need help adding power models, please read the previous section in this document. Here there is an example of the section for a unit:

```
/******************** MY UNIT ************************/

//Power model selection

switch(energyModels[MY_UNIT]){
      case 0: /*Don't count this unit*/
            power.powerAFIndependent[MY_UNIT] = 0.0;
            break;
      case 1: /*Default power model*/
            power.powerAFIndependent[MY_UNIT] = myUnitPowerModel();
      case 2:
            power.powerAFIndependent[MY_UNIT] =
                  userPowerAFIndependent[MY_UNIT];
            break;
      default:
            fprintf(stderr,"Invalid energyModel for %s\n",
            varUnitNames[MY_UNIT]);
            exit(-1);
}
```

### *5.5. Adding the call to the function that calculates power/energy*

The last step is to add one line in a function at `power.c`

The name of the function is:

```
void calculate(int accesses[][MAX_ACCESS_CYCLE],double
activity[][MAX_ACCESS_CYCLE],int energyModels[NUM_UNITS]){
```

You have to add one line, next to the last call to the function:

```
calculateEnergyAndDisplay(…)
```

And the line you have to add looks like:

```
calculateEnergyAndDisplay(accesses,MY_UNIT,power,my_unit_max_a
ccess,FALSE, NULL,0,totalEnergy);
```

`MY_UNIT` is the identifier you have declared at `access_vatios.h` and

`my_unit_max_access` is the max number of accesses to your unit at a cycle.

# 6. Adding the Vatios library to an existing SimpleScalar based simulator.

Basically, Vatios is composed of a standard SimpleScalar simulator plus some files and some modifications to `sim-outorder.c`. These modifications are only made to counter use and activity of the units, not to modify the behaviour of the simulator. In this chapter we are going to explain what has to be done to add Vatios to another SimpleScalar based simulator.

First of all you have to add to your project the `cacti4_2` directory, the `access_vatios.c/.h`, `power.c/.h`, `calculatePower.h` and `vatios_strings.h`. Additionally, the `options.c` file is slightly modified, so you will have to replace the original.

Now you will have to modify your simulator to link it with the Vatios library.
At the beginning of your simulator (usually `sim-outorder.c`), you will have to include the `access_vatios` library:

```
/* added for Vatios */
#include "access_vatios.h"
```

Then, as global variables, you will have to declare these variables:

```
/* Vatios*/
extern char * dumpConfigFileName;
access_data_t access_data;
int energyModels[NUM_UNITS];

int va_size = 48;
int  technologyPoint = 350;
double crossover_scaling = 1.2;
double turnoff_factor = 0.1;
int opcode_length = 8;
int inst_length = 32;
extern char * varUnitNames[];
extern char * varAfNames[];
extern char * comentars[];
extern char * afComentars[];
char modelOption[NUM_UNITS][128];
```

In the function that registers simulator –specific options, you will have to include these new options.

The function will look like:

```
/* register simulator-specific options */
Void sim_reg_options(struct opt_odb_t *odb){
     //Vatios
     int j;
     for(j=0;j<NUM_UNITS;j++){
          sprintf(modelOption[j],"%s:model",varUnitNames[j]);
          opt_reg_int(odb,modelOption[j],"Energy Model",
                              &energyModels[j],1,FALSE,NULL);
          //We use 1 as default value,
          //because in Vatios 0 means
          // that this unit consumes 0 energy.
     }

opt_reg_double(odb, "-tech","Tech point.(Double in nm)”
     ,&technologyPoint,350,FALSE, NULL);
```

The next step is to add the code to show the results and to dump all the access/use info to a file. You have to find the `sim_aux_stats` function and modify it so that it begins like:

```
/* dump simulator-specific auxiliary simulator statistics */
void
sim_aux_stats(FILE *stream)             /* output stream */
{

//Vatios
if(dumpConfigFileName != NULL)
               dump_access_vectors(stream,&access_data);
calculate(access_data.accesses,access_data.activity,energyMode
ls);
```

The last step is to add at the main simulator, one call to **`clear_access_stats();`** and one call to **`update_access_stats(&access_data);`** at these points of the main loop:

```
  for (;;)
    {
      /* RUU/LSQ sanity checks */
      if (RUU_num < LSQ_num)
     panic("RUU_num < LSQ_num");
      if (((RUU_head + RUU_num) % RUU_size) != RUU_tail)
     panic("RUU_head/RUU_tail wedged");
      if (((LSQ_head + LSQ_num) % LSQ_size) != LSQ_tail)
     panic("LSQ_head/LSQ_tail wedged");

      /* added for Wattch to clear hardware access counters */
```

```
      clear_access_stats();
```

```
…
      else
     ruu_fetch_issue_delay--;

  /* Added by Wattch to update per-cycle power statistics */
      update_access_stats(&access_data);

      /* update buffer occupancy stats */
      IFQ_count += fetch_num;
…
```

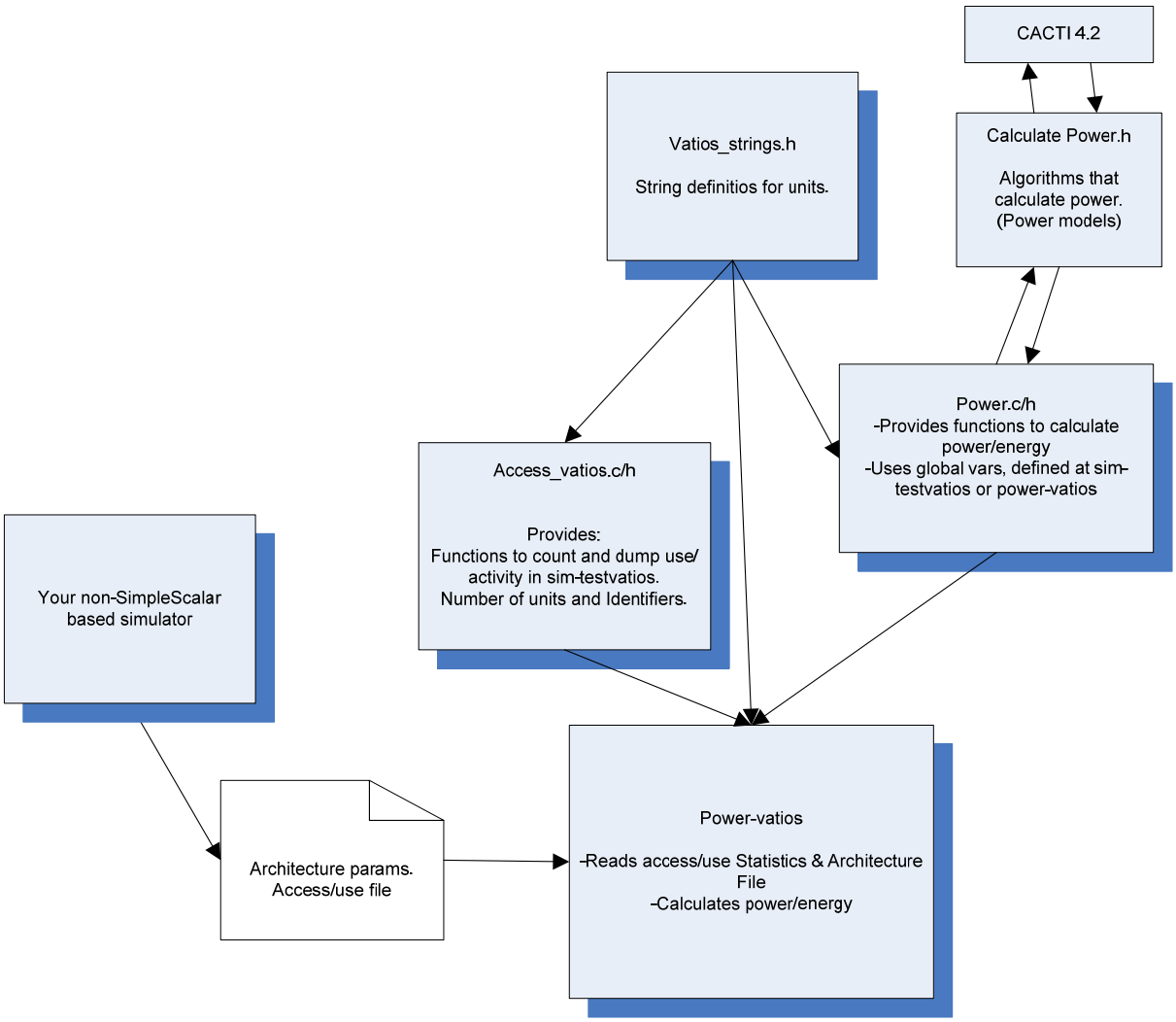
The rest of the work depends on the units in your simulator and your power models. It's recommended to understand the differences between `sim-vatios.c` and the original sim-outorder.c before including Vatios to an existing SimpleScalar based simulator.

To compile the project, you will have to modify the Makefile. You can take the `Makefile` distributed with Vatios as an example.

# 7. Adding Vatios to a non-SimpleScalar based simulator.

If you are using a simulator that is not based on SimpleScalar, you won't have a sim-outorder file. Despite that, you could use the Vatios library and `power-vatios`. You only have to modify your simulator to dump some stats and architectural parameters to a file with a certain format.

This would be the simulator architecture:




It's highly recommended to be familiar with Vatios and how to add new power models/units. First of all, you should generate a file, readable by the SimpleScalar `options.h` library, with all the information needed by `power-vatios.`

Any option should look like:

```
-optionName value
```

Lines that begin with '#' are considered as comments.

For example:

```
# instruction decode B/W (insts/cycle)
-decode:width                    4
# instruction issue B/W (insts/cycle)
-issue:width                     4
# run pipeline with in-order issue
-issue:inorder                false
# issue instructions down wrong execution paths
-issue:wrongpath               true
# instruction commit B/W (insts/cycle)
-commit:width                    4
# register update unit (RUU) size
-ruu:size                       16
# load/store queue (LSQ) size
-lsq:size                        8
# l1 data cache config, i.e., {<config>|none}
-cache:dl1             dl1:128:32:4:l
```

For every unit you are going to calculate energy predictions, you will have to write access and use information using the function `void dump_access_vectors(FILE * stream, access_data_t * access_data)` included in access_vatios.h, or with this format:

```
-unitName:accesses       [Cycles0,Cycles1,Cycles2,…,CyclesN]
```

Where Cycles0 is the number of cycles when this unit has been accessed 0 times, Cycles1 is the number of cycles when this unit has been accessed 1 time, and so on…

For units that depends on an Activity factor (that can vary between 0 and 1), you will have to write the Activity factor information with this format:

```
-unitName_af         [term0,term1,term2,…,termN]
```

Where termX is the addition of all the activity factors resulting from cycles where the number of accesses to this unit have been X.

The activity factor can be calculated using `pop_count` function. You have predefined functions to do all of this in the `acces_vatios.h` library.

An example of the access and use information is:

```
#=========================================#
#               ACCESS VECTORS            #
#=========================================#

-rename:accesses         [34095,2756,3321,2438,9505]
-bpred:accesses          [45568,5738,780,29]
-icache:accesses         [31714,20401]
-dcache:accesses         [41927,7021,2854,285,28]
-dcache2:accesses        [49057,3032,23,2,1]
…
…
-lsq_af [0.0,2546.171875,494.617188,326.916667,42.101562]
-resultbus_af [0.0,4727.468750,5078.125,5485.276042,3266.125]
```

Of course in the dumped file, you will have to include all the information needed by your power models. `power-vatios` depends on all this libraries, so you will have to include them from the Vatios tar.gz file from the Web. Don't forget to include the .c files as well.

```
access_vatios.h
vatios_strings.h
options.h
power.h
cache.h
misc.h
```

Note: the function `dump_access_vectors(FILE * stream, access_data_t * access_data)` depends on a extern variable declarated as:

```
extern char * dumpConfigFileName;
```

You have to make sure that this variable has the name of the file where all other information has been dumped when you call the function.

Now, you only have to modify the files you have included to map the units in your simulator. The guide to this process can be found at previous sections of this manual.

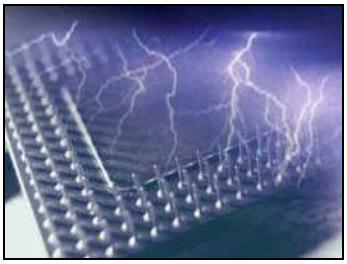
Note:

There are some units whose power depends on an activity factor. These units should be treated specially, because not only the number of accesses should be count, but also the activity factor of the results.

Regfile
Resultbus
Window
LSQ

If you are going to add a new unit and you want to measure the activity factor, please, take a look at how these units are modelled. The process is similar, but you will have to add extra code in some points.

# Sim-gaz



Juan Antonio Victorio Ferrer

Proyecto Fin de Carrera

CPS- Universidad de Zaragoza

# Tabla de contenidos

# 1. Introducción

`Sim-gaz` es un simulador creado para el grupo de Arquitectura de Computadores de Zaragoza (gaZ). La idea de crear este simulador surgió debido a las diferencias de la arquitectura del simulador SimpleScalar (que es la misma de Wattch y de `sim-vatios`) comparada con la arquitectura de los procesadores actuales.

Este documento va dirigido a miembros del gaZ y en él se describe la implementación de este simulador, así como los modelos de potencia que se incluyen. Se trata de un documento técnico que sirve como guía para comprender la implementación del simulador, y como referencia para futuras modificaciones. Previo a su lectura se recomienda estar familiarizado con SimpleScalar, Wattch y `sim-vatios`.

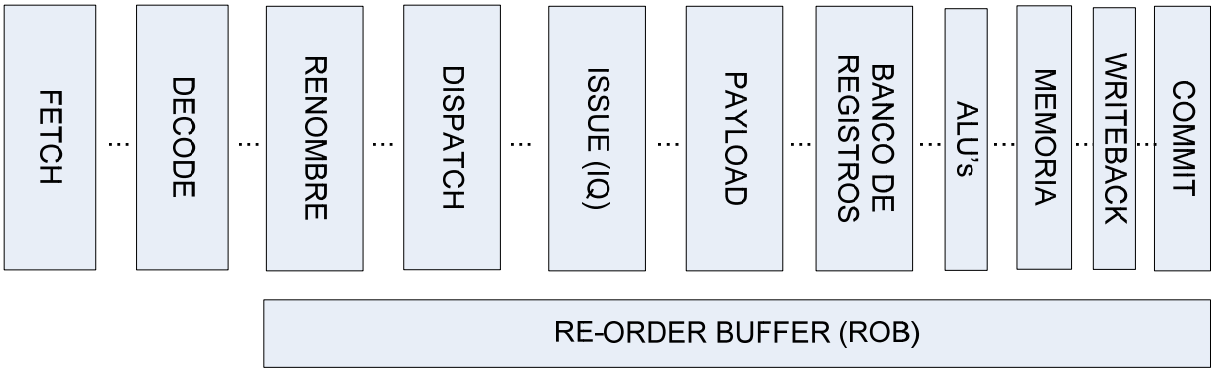
En primer lugar se presentará la arquitectura del simulador, después se detallará la implementación. A continuación se describirán las etapas y temporización del front-end (primeras etapas del segmentado), del back-end (últimas etapas del segmentado), el sistema de memoria y por último se describirán los modelos de potencia, tanto de las estructuras del núcleo de ejecución como de las del sistema de memoria.

# 2. Arquitectura del simulador.

En esta sección se va a ofrecer una primera visión de la arquitectura de `sim-gaz`.

Se modela un procesador superescalar, con ejecución fuera de orden, basado en modelo de ventanas de lanzamiento (ROB + IQ) con IQ's separadas para enteros y coma flotante. Se ha modificado la prioridad de inicio de ejecución de las instrucciones de tal modo que en este simulador las instrucciones se inician en estricto orden de edad.

El front-end es en orden y esta compuesto por las etapas de fetch, decode, renombre y dispatch. La latencia de cada etapa es parametrizable. Se modela un front-end agresivo, es decir, se puede controlar la velocidad relativa del front-end del procesador con respecto al back-end mediante un parámetro. `Sim-gaz` implementa el mecanismo estándar de renombre.



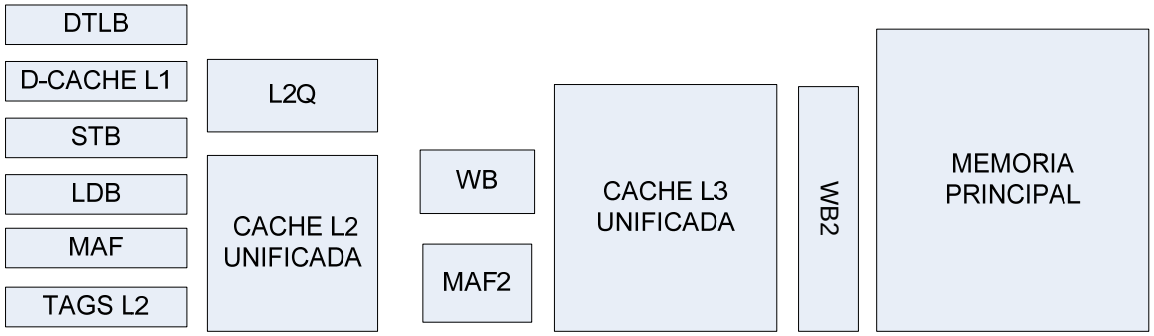

Etapas del segmentado del simulador `sim-gaz.`

El back end es fuera de orden, se modela un procesador con predicción de latencia de instrucciones Load. Las instrucciones dependientes se despiertan con antelación para enlazar su ejecución inmediatamente después del cálculo de sus operandos fuente. Las instrucciones permanecen en la IQ hasta confirmar la predicción. En caso de fallo en la predicción se ha implementado un mecanismo de anulación y reejecución de instrucciones.

Se modela un Store Buffer y un Load Buffer, así como un contador de instrucciones totales de memoria en vuelo. Este contador simula etiquetas asignadas a las instrucciones de memoria en la etapa de renombre, que además es cuando las instrucciones entran en el ROB.

Se permite la ejecución en desorden entre Store-Load. Se ha implementado un predictor ideal (Oráculo). Cuando un Load depende de un Store que va a la misma dirección, el Load no inicia la ejecución hasta que el Store se haya ejecutado.

La arquitectura de la memoria que se ha modelado es similar al Itanium II, con Miss Address File (MAF), MAF2 entre la cache de nivel 2 y la cache de nivel 3 o memoria principal. El acceso al array de datos de L2, a L3 y a Memoria principal se hace a través de una cola llamada L2Q. Se implementa un buffer de escritura (WB) de L2 a L3 y otro (WB2) de L3 a Memoria principal. El acceso a la memoria cache de instrucciones se conserva igual que el de SimpleScalar.




Organización de la jerarquía de memoria del simulador `sim-gaz`.

# 3. Implementación

A continuación se van a describir aspectos importantes de la implementación de `sim-gaz.`

Para evitar que el fichero principal que contiene el código del simulador fuera demasiado extenso, se han creado dos ficheros para modularizar el simulador. `sim-gaz.h y sim-gaz.defs.h.` En `sim-gaz.h` se encuentran la declaración de variables globales del simulador, mientras que en `sim-gaz.defs.h` se encuentran las definiciones de estructuras como `RUU_station`, `RSlink` y algunas macros del núcleo de ejecución del simulador. Para dar soporte al mecanismo de renombre, se han incluido los siguientes ficheros: `estadosreg.base.c/h` y `renombre.base.c/h`.

Los ficheros en los que se incluye la implementación de la jerarquía de memoria de `sim-gaz` son `memoria.c/h`. Además, la función que utiliza SimpleScalar para acceder a memoria se llama `cache_access()` y está definida en los ficheros `cache.h/.c`. En `sim-gaz`, además se utiliza una modificación de esta función llamada `luisma_cache_access()` y que se encuentra en los mismos ficheros.

Adicionalmente se han incluido en `sim-gaz` todos los ficheros relacionados con el cálculo de potencia. El listado de estos ficheros se puede encontrar en el manual del programador de `sim-vatios`.

Para que la simulación se ejecute con las caches y los predictores de salto precalentados, antes de la simulación con detalle se simula una ejecución funcional *fast-forward*, el código de este *fast-forward* se encuentra en la función `sim_main`, previo al bucle principal del simulador. Una vez terminado el *fast-forward* se re-inicializan las estadísticas de caches y predictor de saltos. Para que el simulador funcionara correctamente con ficheros de traza (.eio), la variable `sim_num_commit` sustituye a `sim_num_insn` en el cálculo de estadísticas.

La ejecución funcional de las instrucciones se realiza en la rutina `ruu_funcional()`, que es llamada dentro de `ruu_fetch()`. En la ejecución funcional se ejecuta propiamente la instrucción, y además se escribe la información de la instrucción en las estructuras necesarias (principalmente en la RUU) para poder simular el comportamiento temporal de esa instrucción a lo largo del segmentado. RUU pasa a ser una estructura únicamente de soporte a

la simulación, su tamaño es el total de las instrucciones en vuelo en el simulador. La estructura RUU ya no tiene relación con la arquitectura que se modela.

Se ha modificado la rutina `readyq_enqueue()`, que como su nombre indica, contiene a las instrucciones listas para inicial la ejecución, para que al insertarlas su orden sea estrictamente por edad de las instrucciones.

Se ha unificado la cola de eventos del núcleo de fuera de orden, es decir, ahora se implementa una sola cola, pero con etiquetas que indican a qué evento se refiere. Esta cola se ordena por ciclo en el que se produce el evento, tipo de evento y edad de la instrucción.

Como se ha dicho en la sección previa, se ha separado el ancho de lanzamiento (anteriormente `issue_width` ) en `issue_width_int` e `issue_width_fp`, además se ha eliminado la restricción de que fuera potencia de 2. Estos valores se parametrizan mediante las opciones "`-issue_int:width`" y "`-issue_fp:width`".

Loads y Stores pasan a ser una única instrucción (y no una de cálculo de dirección y otra de acceso a memoria, como se hace en SimpleScalar). Ahora estas instrucciones pasan por etapas de cálculo de dirección y de acceso a memoria. Este cambio afecta, por lo tanto, a la segmentación del procesador. La estructura Load Store Queue (LSQ) desaparece y aparecen Store Buffer (STB) y Load Buffer (LDB).

El número de instrucciones de memoria en vuelo se modela mediante un contador entero `colaLDST_num`, y unas funciones asociadas que se llaman `colaLDST_mete()` y `colaLDST_saca()`. Análogamente para el Store Buffer (STB) y para el Load Buffer (LDB). Estos contadores se utilizan en la etapa de renombre para verificar si se puede renombrar la siguiente instrucción. La condición es que `colaLDST_num`, `STB_num` y `LDB_num` sean menores que un valor máximo especificado por parámetro para cada uno de ellos.

Se ha implementado un sistema de detección de dependencias Store-Load mediante un filtro bloom. Esta estructura nos permite evitar búsquedas exhaustivas de dependencias Store-Load por cada nuevo Load que entre en nuestro segmentado, y por lo tanto incrementamos el rendimiento de nuestro simulador. Cada Store tiene una lista de dependencias de salida en la

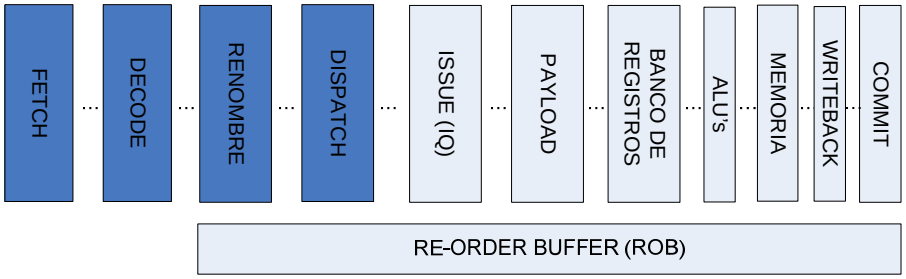
que se insertan los Loads que dependen de él. Se ha añadido una nueva dependencia de entrada extra para los Loads, que hace que no estén ready si existe un Store previo que va a escribir a la misma dirección.

Para evitar Live-Locks (por ejemplo dos o más instrucciones de memoria provocan expulsiones de bloques de cache mutuamente, de modo que ninguna de ellas consigue terminar su ejecución) se ha establecido un umbral máximo de fallos que puede producir una instrucción que acceda a memoria. Una vez superado este umbral, la instrucción pasa obligatoriamente a Writeback y se cuenta el número de veces que se produce esta situación. Este tipo de situaciones es muy poco común, pero ocurre incluso en procesadores reales y la forma de solucionarlo es que el procesador pase a un modo seguro en el que ejecuta las instrucciones en orden.

# 4. Front-End. Etapas y temporización.

En `sim-gaz`, todas las instrucciones pasan por las siguientes etapas en orden:

fetch, decode, rename y dispatch .




En caso de que se defina un predictor de saltos, la búsqueda (`fetch`) de bloques con varias instrucciones de salto tomado la controla el parámetro entero `fbranch` en lugar de `fetch_speed`. El valor de `fbranch` se parametriza por medio de "`-fetch:fbranch`". `fbranch` controla únicamente este comportamiento. Ahora `fetch_speed` representa la velocidad relativa del front-end del procesador respecto del núcleo de ejecución. Esta variable controla el ancho real de los bucles principales de `ruu_fetch()`, `ruu_decode()` `ruu_rename()` y `ruu_dispatch()`, ya que es un factor que multiplica el ancho propio para cada una de las etapas (Existe un `ruu_decode_width` y un `ruu_rename_width`).

En la etapa de fetch (`ruu_fetch()`) se accede a memoria cache de instrucciones, se consulta el predictor de saltos y se realiza la ejecución funcional de la instrucción con llamadas a la rutina: `ruu_funcional()`. Esta rutina se encarga además de insertar la instrucción en la estructura de datos de simulación RUU y de crear las dependencias con otras instrucciones, tanto las dependencias normales de datos por registros, como las dependencias a través de memoria Store-Load a la misma dirección. Se pueden buscar hasta `ruu_decode_width * fetch_speed` instrucciones por ciclo de simulación, pudiéndose encontrar hasta un máximo de `fbranch` instrucciones de salto predichas como tomadas.

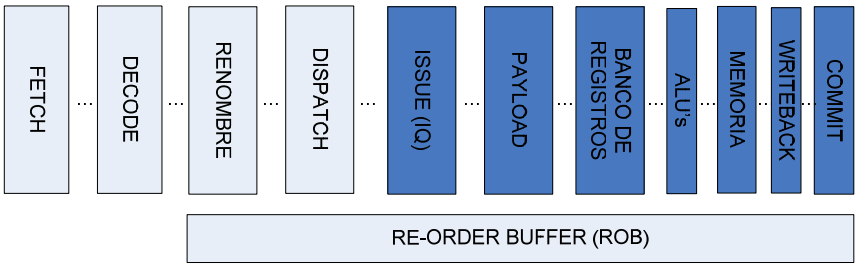
La etapa de decode (`ruu_decode()`) se mantiene para que las instrucciones de salto actualicen el predictor. Esta etapa tiene una latencia configurable por el parámetro `ciclos_decode` (al menos 1 ciclo).

En la etapa de renombre (`ruu_rename()`) se toman las instrucciones que ya hayan sido decodificadas, y se renombran si existen registros físicos disponibles. En caso contrario se bloquea el segmentado. Las instrucciones se meten en el ROB y en caso de que sean de memoria se les asigna una etiqueta (`colaLDST_mete()`) y paralelamente se meten en el Load Buffer (`LDB_mete()`) o Store Buffer (`STB_mete()`). Esta etapa tiene una latencia configurable por el parámetro `ciclos_rename` (al menos 1 ciclo).

En la etapa de dispatch (`ruu_dispatch()`) se sacan de renombre las instrucciones que hayan acabado con esa etapa. Se meten en la IQ correspondiente (`IQint_mete()` o `IQfp_mete()`) y si tienen los operandos disponibles se insertan en la cola de instrucciones listas (`readyq`). Si no tienen todos los operandos listos, se irán despertando conforme instrucciones más viejas calculen los operandos que necesitan. Si no hay sitio en IQ se bloquea el segmentado.

# 5. Back-End. Etapas y temporización.

El Back-end del procesador comprende las etapas desde Issue hasta Commit. Esto incluye las etapas de lectura de Payload, banco de registros, el paso por las ALU's y acceso a memoria en caso de que la instrucción sea de memoria.




Este simulador implementa ejecución fuera de orden con predicción de latencia, por lo tanto tiene que soportar ejecución especulativa de instrucciones, así como la anulación/re-ejecución. El desorden de las instrucciones empieza en la etapa de issue, y a partir de ahí las instrucciones van pasando por ciertos estados y son tratadas por ciertas funciones antes de llegar a Writeback, que es la última etapa en desorden. El siguiente gráfico muestra el diagrama de estados, que iremos comentando a continuación.

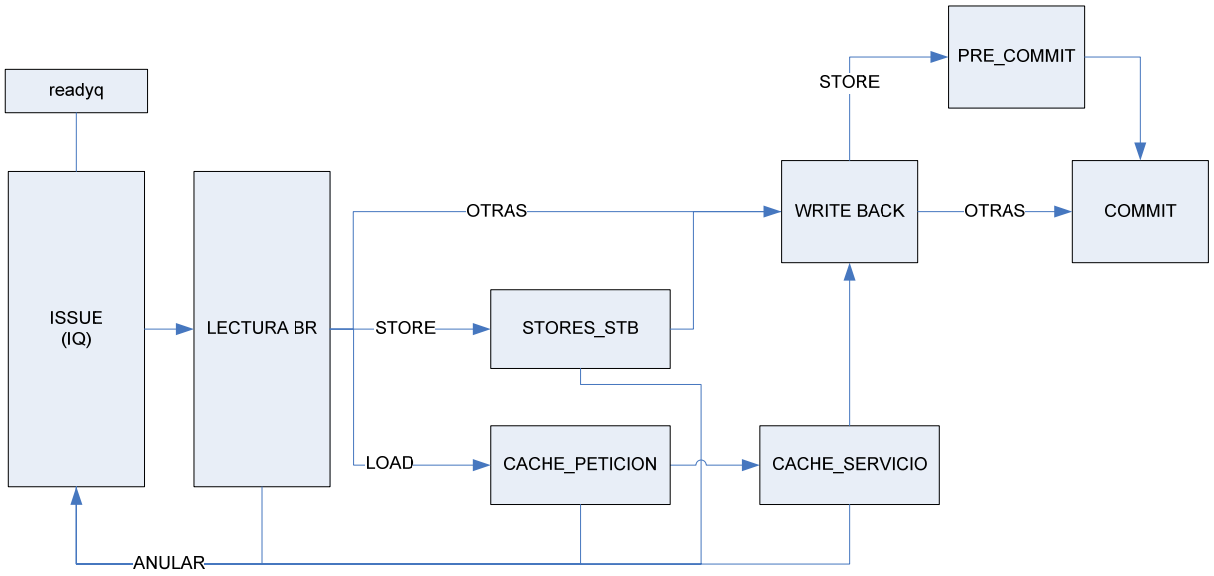



Diagrama de estados por los que pasa una instrucción en el Back-end del procesador.

El paso por los diferentes estados se realiza a través de eventos. Un evento se programa con una llamada a `cola_queue()` y se sirve con la función `cola_next()`.

La primera rutina que trata instrucciones en el fuera de orden es la rutina `ruu_issue()`. Mientras el número de instrucciones tratadas no supere el ancho definido para esa etapa (`ruu_issue_width_int/fp`), esta rutina toma instrucciones listas de la `readyq`, que como ya se ha comentado contiene instrucciones listas en orden de edad .En caso de que haya recursos para esa instrucción, se inicia su ejecución. En primer lugar se programa un evento con la rutina que coincide con el último ciclo de acceso al banco de registros (llamado `LECTURA_BR`). La latencia de este evento es configurable (LAT_REG) y puede ser 0. En estos ciclos se modelan todas las acciones desde que la instrucción inicia su ejecución hasta que llega a la unidad funcional (Lectura BR, Payload, etc.). Además marca la disponibilidad de resultados en función de la latencia de la unidad funcional, con la programación del evento `DESPERTAR`. Para las instrucciones Load, esta rutina supone la latencia de acierto en primer nivel.

Como se ha dicho, la rutina que da servicio al evento DESPERTAR marca la disponibilidad de resultados de manera especulativa. Esta rutina se llama `ruu_despertar()` y para todas las instrucciones dependientes, marca el operando como disponible, y en caso de que sea el último operando que espera la instrucción, la añade a la cola `readyq.`

La siguiente rutina por las que pasan todas las instrucciones en el núcleo de ejecución es la rutina `ruu_lectura_br()`. Esta etapa modela el último ciclo de lectura del banco de registros. Aquí, en función del tipo de instrucción, el comportamiento difiere. En caso de que la instrucción no sea de memoria, se programa su evento de Writeback en función de su latencia de ejecución. En caso de que la instrucción sea un Store se programa un evento llamado `STORES_STB`, que modela el ciclo en el que los Stores, después de calcular la dirección, acceden al STB. En caso de que la instrucción sea un Load, se programa un evento llamado `CACHE_PETICION` que modela el primer ciclo de acceso a memoria cache después de calcular la dirección.

La rutina `ruu_stores_stb()` trata los eventos `STORES_STB`, que como se ha dicho, modelan el acceso de los Stores al STB. Todos los Stores que pasan por esta rutina son sacados de la IQ y se programa para el ciclo siguiente su evento de Writeback.

La rutina `cache_petición()` trata los eventos `CACHE_PETICION`, que modelan el primer ciclo de acceso a memoria de instrucciones Load. En esta rutina se acceden a las

estructuras de memoria, como se comentará en el próximo capítulo y se programa un evento llamado `CACHE_SERVICIO`, que modela el último ciclo de acceso a memoria. Estos dos eventos pueden llegar a coincidir en el mismo ciclo, ya que la latencia de DL1 es configurable mediante el parámetro `-cache:dl1lat.`

La rutina `cache_servicio()` trata el evento `CACHE_SERVICIO`, que modela el ultimo ciclo de acceso a memoria de instrucciones Load. En esta rutina se programa el evento de Writeback del Load para el siguiente ciclo en caso de acierto en el primer nivel. En caso de fallo en el primer nivel, se anula la instrucción y sus dependientes creando una cadena de anulación de dependientes. Si además se conoce la latencia a la que debe ser despertado el load, se programa también ese evento, llamado `DESPERTARSE`. En caso de que esta latencia sea desconocida, será el sistema de memoria quien se encargue de programar dicho evento.

Por último, la rutina de servicio al evento `DESPERTARSE`, llamada `ruu_despertarse()`, simplemente inserta las instrucciones (que serán de tipo Load) en la `readyq`, en caso de que los operandos estén disponibles.

En el siguiente dibujo se puede ver cuándo se produce el cortocircuito para una dependencia Load-Uso:

| ISS | BR | BR | @ | Mem | … | Mem | WB | CT |
|---|---|---|---|---|---|---|---|---|
| | | | | ISS | BR | BR | … | |

Nota: El número de ciclos de acceso a memoria es configurable mediante (`-cache:dl1lat`).

Y para una dependencia Aritmética- Uso:

| ISS | BR | BR | ALU | … | ALU | WB | CT | |
|---|---|---|---|---|---|---|---|---|
| | | | ISS | BR | BR | … | | |

Nota: El número de ciclos de la etapa ALU depende de la latencia de esa operación. El último ciclo de BR de la dependiente coincide con el último ciclo de ALU.

Los Stores utilizan la misma temporización que las aritméticas, salvo que la etapa ALU es la de cálculo de dirección. Las únicas instrucciones que dependen de un Store son los Loads que van a parar a la misma dirección de memoria.

Independientemente del tipo de instrucción, todas las instrucciones acaban encolándose en la cola de instrucciones que van a la etapa de Writeback mediante una llamada a `eventq_queue_event()`. Esta etapa marca las instrucciones como completadas, y por lo tanto sus resultados son firmes. También se recupera el estado del procesador en caso de salto mal predicho.

La siguiente etapa dentro el segmentado es la etapa `ruu_precommit()`, que se ha introducido para que los Stores, una vez se sabe con certeza que son firmes, esto es, ninguna instrucción previa ha causado excepciones y pueden ser retirados, consoliden su estado (WriteThrough en L2). Esto se realiza con un cierto ancho por ciclo de Stores máximo, definido en la constante `STORE_COMMIT_WIDTH`. Los detalles de cómo acceden los Stores a L2 se detallan en el siguiente apartado.

La última etapa del segmentado es `ruu_commit()`. En esta etapa se elimina la instrucción del ROB, se liberan los registros físicos y se retira la instrucción del segmentado y del simulador.

# 6. Sistema de memoria

El sistema de memoria de datos de SimpleScalar se ha sustituido por un sistema de latencia variable, con tres niveles de cache, tipo Itanium II. La memoria de instrucciones, sin embargo, permanece igual que la de SimpleScalar.

Se han utilizado las estructuras del sistema de memoria del simulador de Luís Ramos, sin embargo por simplicidad se han eliminado las prebúsquedas. También se ha cambiado por completo el interfaz del simulador con el sistema de memoria.

En `sim-gaz`, el interfaz del simulador con memoria se define en las rutinas `cache_peticion()` para Loads y `ruu_precommit()` para Stores. Adicionalmente se han incluido las llamadas a las siguientes funciones dentro del bucle principal del simulador:

- `MAF2_ciclo()`
- `L2Q_ciclo()`
- `l1d()`
- `L2Q_BL2release()`

Estas funciones realizan acciones que tienen que ser ejecutadas ciclo a ciclo en el sistema de memoria.

Todo Load pasa por la rutina `cache_peticion()` en su primer ciclo de acceso a memoria, después de sus etapas de banco de registros y calculo de dirección. En el último ciclo de acceso a memoria, los Loads pasan por `cache_servicio()`.

En `cache_peticion()` se accede a TLB, se accede a DL1. También se reserva entrada en STB, LDB, MAF, L2Q, MAF2 y WB. El resultado se indica mediante un campo en la estructura `RUU_station` llamado `load_latencia`.

En caso de acierto en TLB y acierto en DL1 se asigna 0 al campo `load_latencia`, y la rutina `cache_servicio()` envía el Load a Writeback.

En caso de fallo de TLB o falta de recursos, el Load y sus dependientes tendrán que ser reejecutados. Se indica en `load_latencia` el número de ciclos en los que el Load deberá ser despertado para reejecución. Este campo es leído por `cache_servicio()`, quien anula las dependientes y programa el evento `DESPERTARSE`.

En caso de que sea Fallo en DL1, Se distingue entre fallo primario o secundario (ya estaba ese bloque asignado en la MAF). En caso de fallo secundario, si el refill ya esta realizándose, el Load recircula. En el resto de casos, se indica a `cache_servicio()` que será el sistema de memoria el que despertará al Load asignando a `load_latencia` el valor `-1`. En este caso, `cache_servicio()` solamente anula al Load y a las dependientes.

Cuando se satisface una petición de la MAF, el sistema de memoria se encarga de despertar todos los Loads que esperen datos de ese bloque. El despertado de los Loads se programa en la función `MAF_despierta()`, para que el acceso a memoria coincida con el refill a DL1.

En caso de que se produzca un fallo secundario y la MAF ya haya despertado a las instrucciones que esperan ese bloque, el Load recircula, esto es, se vuelve a iniciar su ejecución inmediatamente. A continuación se muestra una tabla resumen de las posibles situaciones en el acceso a memoria por un Load.

| Resultado de `cache_peticion()` | Acción en `cache_servicio()` |
|---|---|
| Hit TLB y hit en primer nivel (DL1 o STB) | Enviar instrucción a Writeback |
| Fallo TLB | Anular el Load, las dependientes y despertar al Load a la latencia del TLB |
| Hit TLB, fallo en primer nivel, Hit MAF y refill en camino. | Recircular (anular el Load, las dependientes y despertar el Load a latencia 1) |
| Hit TLB, fallo en primer nivel, Hit MAF y el refill no esta en camino. | Añadir a lista de Loads para ese bloque. El Load será despertado por la MAF |
| Hit TLB, fallo en primer nivel, Fallo MAF y no hay recursos | Recircular |
| Hit TLB, fallo en primer nivel, Fallo en MAF, hay recursos y hit en L2 | Insertar en MAF y L2Q. El Load será despertado por la MAF |

| | |
|---|---|
| Hit TLB, fallo en primer nivel, Fallo en MAF, hay recursos y miss en L2 | Insertar en MAF, L2Q y MAF2. El Load será despertado por la MAF |

Los Stores acceden al sistema de memoria en la etapa `ruu_precommit()`. Como se ha dicho antes, esta etapa simplemente trata Stores con un determinado ancho definido en la constante `STORE_COMMIT_WIDTH`.

Los Stores acceden directamente a L2. Se hace Lookup en etiquetas de L2 y en caso de hit se añade el Store a la cola L2Q. En caso de Miss en etiquetas de L2 Se inserta el Store en MAF2 y en L2Q. A la vez que se accede a L2, se accede a la MAF, y si se encuentra el bloque se inhibe el refill pendiente. Adicionalmente, los Stores son marcados por el sistema de memoria como “commitables”, que es un campo booleano que se ha introducido en la estructura RUU_station. Esto quiere decir que ya pueden ser retirados en la etapa `ruu_commit()`, al haber consolidado su estado en L2.

El interfaz de L2 a L3 o memoria principal permanece igual que el del simulador de Luís Ramos. Las peticiones de bloques a L2 se realizan en la rutina `L2Q_ciclo()`. El servicio a esas peticiones se trata en la rutina `MAF2_ciclo()`.

NOTA: El modelo de Stores es funcionalmente correcto, pero no se modelan todos los accesos ni se cuentan todos en el consumo o en el dimensionado de estructuras.

# 7. Modelos de Potencia

`sim-gaz` incluye la librería de potencia de `sim-vatios`, por lo tanto calcula la potencia y el consumo del mismo modo, sin embargo las unidades que se modelan en `sim-gaz` son distintas de las que se modelan en `sim-vatios`. Esto obligó a crear nuevos modelos de potencia para ciertas unidades.

Para cada una de las unidades se ha incluido al menos un modelo de potencia, es decir, un modo de calcular la potencia de pico de esa unidad (el máximo consumo que se puede producir), para luego escalarla en función del uso, tal y como hacen Wattch y `sim-vatios`.

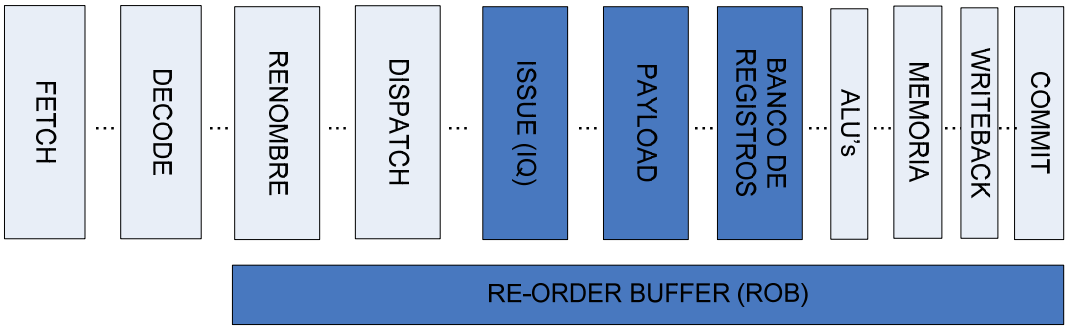
Para seguir manteniendo la semántica de `sim-vatios`, siempre se incluye el modelo 0 (no se cuenta el consumo de esa unidad para el cómputo total) y el modelo 2 (el usuario introduce la potencia a través de un parámetro en la línea de comando). Además hemos asignado el número 1 a los modelos heredados de Wattch, presentes en unidades que no han cambiado en `sim-gaz`, y el número 3 a los modelos cuya potencia se calcula únicamente con CACTI. Hay unidades para las que no se ha programado el modelo 1, ya que no estaban modeladas por el procesador de Wattch.

Los modelos de potencia de unidades como ROB, IQ, Payload, Cache L2, STB, LDB, MAF han sido creados desde cero, ya que Wattch no modelaba esas unidades. Algunos modelos han sido levemente modificados, como es el caso del Resultbus o del Clock. Otros modelos de potencia como los de Renombre, Predictor de salto, ALU's se han mantenido iguales a los de Wattch.

A continuación se van a repasar todas las unidades cuyo modelo presente cambios significativos y se va a describir brevemente el modelo de potencia que se ha programado para ellas. También se indicará donde se cuentan los accesos para posteriormente aplicar el escalado a causa del clock gating.

## 7.1. Modelos de Potencia – Núcleo del Procesador

En primer lugar vamos a describir los modelos de potencia programados para las estructuras del núcleo, en orden de ejecución (salvo las estructuras para instrucciones de memoria):




Etapas que componen el núcleo de la ejecución.

### 7.1.1. Reorder buffer (ROB)

El ROB es una estructura que mantiene el orden de las instrucciones. Las instrucciones entran en la etapa de Renombre y salen en Commit. Dado que la carga de información de la instrucción se guarda en el Payload, el ROB sirve simplemente para conservar el orden de las instrucciones y guardar el estado en el que se encuentran.

Las entradas del ROB se escriben en bloques del ancho del procesador. El ROB se ha modelado como una memoria RAM de `ROB_size` entradas de `ROB_width` bits por bloque. Nota: En el código, el número de bits se divide por 8 para pasarlo a bytes. `Rob_width` es una constante definida con el número de bits estimado para la información que se guarda en el ROB por cada instrucción.

Se han modelado `fetch_speed` puertos de escritura, ya que escribimos ese número de bloques y 0 puertos de lectura, ya que del ROB solamente se leen bits de las entradas, y no palabras completas. Los accesos al ROB se cuentan en la etapa de renombre (`ruu_rename()`).

### 7.1.2. Payload

El Payload es una memoria que contiene la carga de información necesaria para que la instrucción se ejecute, por ejemplo el código de la operación, registros fuente, registro destino, etc.… Esta memoria se ha modelado utilizando CACTI. El modelo es una memoria de mapeo directo de (`IQfp_size+IQint_size`) entradas de `PAYLOAD_WIDTH` bits por entrada con (`ruu_issue_width_int + ruu_issue_width_fp`) puertos de lectura. Se consideran tantos puertos de escritura como el ancho de dispatch.

Los accesos al Payload se cuentan en la rutina `ruu_issue()` y en la rutina ruu_`dispatch()` mediante la variable `payload_access`.

### 7.1.3. IQ

En sim-gaz se han modelado lad IQ'd como 2 memorias CAM de nº de registros bits de longitud con `IQint_size y IQfp_size` entradas respectivamente. Se consideran tantos puertos de escritura como el ancho de dispatch, y un puerto de lectura (broadcast de disponibles).

Además se ha añadido la potencia de la lógica de selección utilizando la función `selection_power()` heredada de Wattch.

El acceso a la IQ se cuenta en la rutina `ruu_issue()` mediante la variable `iq_access`. Para la IQ no se considera posibilidad de clock-gating.

### 7.1.4. Banco de Registros (Regfile)

Para el banco de registros se han incluido dos modelos. El modelo 1 es el heredado de Wattch. El modelo 3 utiliza CACTI para modelar dos bancos, uno de enteros y otro de coma flotante, cuya potencia se suma. Para el modelo 1 se cuenta factor de actividad a través de `pop_count()` (cuenta de 1's/ 0's en el resultado), sin embargo para el modelo 3 no tenemos esto en cuenta.

Se cuentan accesos al banco de registros en la rutina `ruu_lectura_br()` y en la rutina `ruu_writeback()`.

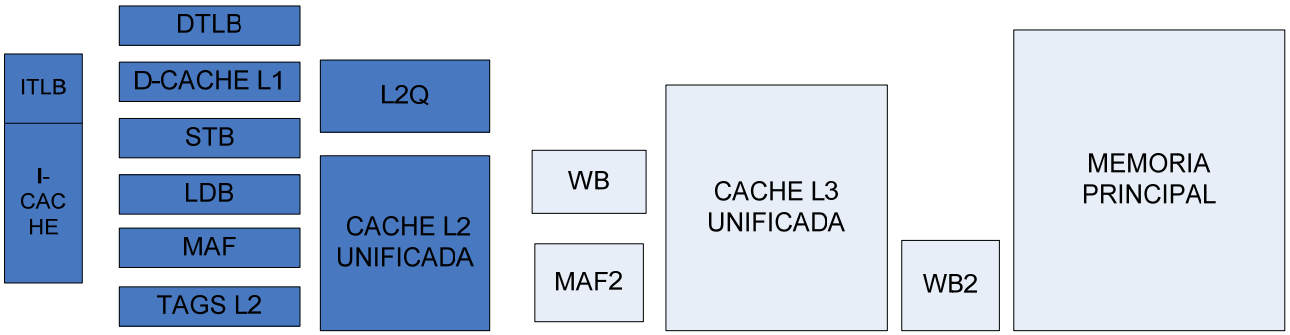
### 7.1.5. Bus de resultados (Resultbus)

El modelo de potencia para el Resultbus es el mismo de Wattch (modelo1), pero ligeramente modificado. En primer lugar se calcula la potencia para dos buses, uno de enteros y otro de coma flotante. Además se ha modificado la función `compute_resultbus_power()` para que reciba dos parámetros, el número de registros y el ancho de issue.

Los accesos al bus de resultados se cuentan en la rutina `ruu_writeback()` mediante la variable `resultbus_access`. Dado que la potencia del Resultbus se escala en función del factor de actividad, se realiza una cuenta de número de 1's en el resultado mediante la llamada a la función: `pop_count(rs->val_rc).`

## *7.2. Modelos de Potencia – Memoria*

A continuación vamos a describir los modelos de potencia programados para las estructuras de las instrucciones de memoria, primero las de memoria de instrucciones y luego las de memoria de datos. Desde las estructuras más cercanas al núcleo hasta las más alejadas.




Estructuras modeladas en la memoria del simulador sim-gaz

### 7.2.1. Instruction TLB

A diferencia de Wattch, en `sim-gaz` contamos los accesos al TLB de instrucciones de manera independiente de los accesos a la cache de instrucciones. Aunque el número de accesos que se cuenten sea el mismo, permite utilizar modelos de potencia diferentes para estas unidades.

Para el TLB de instrucciones se han incluido 3 modelos de potencia, el modelo1 que utiliza Wattch para el TLB, el modelo 3 basado en CACTI y lógica completamente asociativa CAM

y el modelo 4 basado en CACTI, pero con la asociatividad indicada por el parámetro que define el TLB de instrucciones (`"-tlb:itlb"`).El acceso al TLB se cuentan en la rutina `ruu_fetch()`.

### 7.2.2. Instruction Cache

Para la cache de instrucciones se han incluido dos modelos de potencia. El modelo 1, heredado de Wattch, y el modelo 3 utilizando CACTI para modelar una cache estándar con los parámetros de tamaño y asociatividad correspondientes. El acceso a la cache de instrucciones también se cuenta en la rutina `ruu_fetch()`.

### 7.2.3. Store Buffer (STB)

Para el Store Buffer hemos modelado una memoria CAM, completamente asociativa, de `STB_size` entradas de `STB_ENTRY_SIZE` bytes por entrada. Como en este tipo de memorias la operación que consume la mayor parte de la energía, la hemos modelado con `res_memport` puertos de lectura.

El acceso al STB se cuenta con la variable `store_buffer_access` en la rutina `cache_peticion()`, por la que pasan las instrucciones de tipo Load. El acceso de los Stores al STB no se cuenta, ya que es un acceso en escritura y la operación que tiene el consumo significativo es la lectura CAM.

### 7.2.4. Load Buffer (LDB)

Para el Load Buffer hemos modelado igualmente una memoria completamente asociativa, de `LDB_size` entradas de `LDB_ENTRY_SIZE` bytes por entrada. Igualmente se han modelado `res_memport` puertos de lectura. Los accesos se cuentan por medio de la variable `load_buffer_access` y se cuentan tanto en la rutina `cache_peticion()` para los accesos producidos por Loads como en la rutina `ruu_stores_stb()` para los accesos producidos por los Stores. La escritura al Load Buffer por parte de Loads no se tiene en cuenta, ya que son escrituras en RAM y el consumo significativo se produce en las lecturas CAM.

### 7.2.5. Miss Address File (MAF)

Se modela la MAF como una memoria completamente asociativa, de `MAF_size` entradas de `MAF_ENTRY_SIZE` bytes por entrada. Se modelan también `res_memport` puertos de escritura. Se cuenta un acceso a MAF en la variable `maf_access` cada vez que un Load para por la rutina `cache_peticion()`.

### 7.2.6. Data TLB

Del mismo modo que para el TLB de Instrucciones, para el TLB de datos se han incluido tres modelos. El modelo 1 heredado de Wattch, el modelo 3 utilizando CACTI y modelando lógica completamente asociativa y el modelo 4, basado en CACTI, pero con la asociatividad indicada por el parámetro que define el TLB de datos ("`-tlb:dtlb`").

El número de puertos de lectura para el DTLB es `res_memport`. El acceso al TLB de instrucciones se cuenta en la rutina `cache_peticion()`.

### 7.2.7. Data Cache

Para la cache de datos de nivel 1 se han incluido dos modelos de potencia. El modelo 1 es el mismo que incluye Wattch. El modelo 3 utiliza CACTI para calcular la potencia. En el modelo 3 hemos considerado `res_memport` puertos para esta cache.

El acceso de la cache de datos se cuenta en la rutina `cache_peticion()`.

### 7.2.8. Level 2 Cache (Tags)

Para los tags de L2 se ha modelado una cache del tamaño correspondiente. A continuación se han extraído del resultado las parte de la potencia que corresponde a los tags (Es posible sacarlo del `struct` que devuelve como resultado CACTI).

Los accesos a las etiquetas de L2 se cuentan en la rutina `cache_peticion()`, ya que es en esta primera etapa de memoria cuando se consultan las etiquetas de L2 en paralelo con el resto de estructuras de memoria. No se tienen en cuenta los accesos a tags de L2 por parte de los Stores, ni los accesos cuando se produce un refill a L2.

### 7.2.9. Level 2 Cache (Data)

Para modelar los datos de la cache de nivel 2 se ha dividido el tamaño total de la cache en tantos bancos como indica la constante `DCACHE2_NUM_BANKS`. El modelo de potencia calcula, utilizando CACTI, la potencia para un banco del tamaño correspondiente, 1 puerto de lectura/escritura. A continuación, para calcular la potencia total de los datos de l2 se multiplica la potencia de una de esas caches por `DCACHE2_NUM_BANKS`.

Los accesos a los datos de L2 se cuentan en el fichero `memoria.c`. Se cuentan 2 accesos (2 bancos) para cada refill de l2 a l1 y 8 accesos para cada refill de l3 a l2.

### 7.2.10. Cola Cache L2 (L2Q)

La L2Q se modela en `sim-gaz` a través de CACTI. Se modela como una memoria de mapeo directo (RAM) de `L2Q_size` entradas de tantos bits por entrada como indica la constante `L2Q_WIDTH`. Se modelan tantos puertos de lectura como indica la constante `MAX_LANZADOS_L2Q` para las peticiones que se leen y tantos puertos de escritura como indica `res_memport` para las peticiones que vienen del procesador.

Los accesos a L2Q se cuentan con la variable `l2q_access` tanto en la rutina `L2Q_ciclo()` dentro de `memoria.c`, como en la rutina `cache_peticion()` en `sim-gaz.c`.