ESTIMACIÓN DE POTENCIA Y ENERGÍA EN MICROPROCESADORES A TRAVÉS DE ANÁLISIS ESTÁTICO DE CÓDIGO

Por

Oscar Acevedo Patiño

Tesis sometida en cumplimiento parcial de los requerimientos para el grado de

MAESTRÍA EN CIENCIAS

en

INGENIERÍA ELÉCTRICA

UNIVERSIDAD DE PUERTO RICO
RECINTO UNIVERSITARIO DE MAYAGÜEZ

Diciembre, 2005

Aprobada por:

Rogelio Palomera, Ph.D Miembro, Comite Graduado	Fecha
Isidoro Couvertier, Ph.D Miembro, Comite Graduado	Fecha
Manuel Jimenez, Ph.D Presidente, Comite Graduado	Fecha
Francisco Maldonado, Ph.D Representante de Estudios Graduados	Fecha
Isidoro Couvertier, Ph.D Director del Departmento	Fecha

Abstract of Dissertation Presented to the Graduate School
of the University of Puerto Rico in Partial Fulfillment of the
Requirements for the Degree of Master in Sciences

MICROPROCESSOR POWER AND ENERGY ESTIMATION USING STATIC ANALYSIS OF CODE

By

Oscar Acevedo Patiño

December 2005

Chair: Manuel Jimenez, Ph.D.
Major Department: Electrical and Computer Engineering

Current methodologies for software-level estimation of power and energy consumption use a power model developed for the microprocessor along with specialized tools that profile the program under study to extract the model´s parameters. These tools commonly rely on real-time execution or simulations of the analyzed program which require real run-time data. This work presents an alternative methodology for power and energy estimation, in which the usage of such specialized tools is deemphasized. It is proposed instead the use of static code analysis to study and predict a program behavior. This, in combination with the microprocessor power model, allows the developed methodology to estimate power and energy for a program execution with only a small amount of run-time data. The new methodology first performs a static analysis of the program and then computes time, power, and energy costs for each node of its control flow graph (CFG) representation. Subsequently, these costs are combined with statistical program path information, obtained by analyzing the CFG, arriving at estimated power and energy costs for the program. We present power and energy estimation results for a set of 5 representative benchmark

embedded-system programs. Results show that the new methodology, besides the advantages of static code analysis and reduced run-time data requirements, can produce competitive results, with errors less than 20 % for energy estimates and 12 % for power estimates, with respect to experimentally measured values.

Resumen de Disertación Presentado a Escuela Graduada
de la Universidad de Puerto Rico como Requisito Parcial de los
Requerimientos para el Grado de Maestría en Ciencias

ESTIMACIÓN DE POTENCIA Y ENERGÍA EN MICROPROCESADORES A TRAVÉS DE ANÁLISIS ESTÁTICO DE CÓDIGO

Por

Oscar Acevedo Patiño

Diciembre 2005

Consejero: Manuel Jimenez, Ph.D.
Departamento: Ingeniería Eléctrica y Computadoras

Las metodologías actuales para la estimación del consumo de potencia y energía a nivel de programas utilizan un modelo de potencia desarrollado para el microprocesador bajo estudio en conjunto con herramientas especializadas que caracterizan el programa bajo prueba con el fin de extraer la información requerida por dicho modelo. Estas herramientas normalmente realizan una ejecución en tiempo real ó una simulación del programa bajo análisis, para lo cual, se requieren datos reales. Este trabajo presenta una metodología alterna, donde no se requiere el uso de estas herramientas. Se propone el uso de herramientas de análisis estático de código para analizar y predecir el comportamiento del programa bajo prueba. Esto, combinado con el modelo de potencia del microprocesador bajo estudio, permite obtener un valor estimado del consumo de potencia y energía al ejecutar un programa, con una mínima cantidad de datos adicionales. El método presentado realiza primero el análisis estático del programa, luego calcula los costos de tiempo, potencia y energía para cada nodo de la grafica de control de flujo del programa. Luego se combinan los resultados con las diferentes rutas del programa, obteniendo el costo estimado

de potencia y energía total del programa. Se presentan pruebas realizadas con un conjunto de programas de comparación de rendimiento, compuesto por 5 programas representativos del área de sistemas empotrados. Los resultados obtenidos indican que la metodología, además de las ventajas del análisis estático, produce resultados competitivos, con error menor al 20 % para estimados de energía y menor al 12 % para estimados de potencia respecto a las medidas reales de potencia y energía

Dedico este esfuerzo a Dios, a mi esposa Wilma, a mi familia y mis amigos.

AGRADECIMIENTOS

Quiero expresar mi gratitud al profesor Manuel Jiménez por su dedicación y ayuda en este trabajo. Agradezco al profesor Rogelio Palomera y al profesor Isidoro Couvertier por su colaboración y apoyo durante mi tiempo en la universidad.

Mi gratitud es también con el Recinto Universitario de Mayaguez por permitirme tener esta experiencia.

Agradezco por su amistad y apoyo a mis amigos, en especial a Jose Joaquin, Rafael Medina, Irvin Ortiz, Rafael Arce, Sandra Ordoñez, Alberto Quinchanegua, Cesar Aceros y Frank Vennemeyer. Quiero que sepan, que esté donde esté, los recordare siempre.

Agradezco mucho a mi esposa Wilma por su amor, paciencia, comprensión y apoyo durante todo este tiempo. A mi familia que desde la distancia me ha apoyado y siempre ha querido que salga adelante.

También agradezco a mis amigos de Colombia por su apoyo.

Gracias Dios por haber permitido tener esta experiencia.

Índice general

pagina

ABSTRACT ENGLISH II

RESUMEN EN ESPAÑOL IV

AGRADECIMIENTOS VIII

Índice de Tablas XII

Índice de Figuras XIV

1. INTRODUCCIÓN 1

2. TRABAJO PREVIO 4

2.1. Estimación del Consumo de Potencia y Energía de Microprocesadores 4
2.1.1. Potencia a Nivel de Circuito 5
2.1.2. Potencia a Nivel de Bloques Funcionales 8
2.1.3. Potencia a Nivel de Instrucciones 10
2.1.4. Metodología para la Generación de Estadisticas de los Parámetros de un Modelo 17
2.2. Analisis Estático de Código 18
2.2.1. Gráfica de Control de Flujo 18
2.2.2. Predicción de Saltos 19
2.2.3. Estimación de las Iteraciones de un Ciclo 22
2.2.4. Estimación del Tiempo de Ejecución 24
2.3. Discusión 26

3. OBJETIVOS Y METODOLOGíA 29

3.1. Objetivos del Proyecto 29
3.2. Metodología 30
3.2.1. Perfil de Potencia a Nivel de Instrucciones 31
3.2.2. Analizador Estático de Código 32
3.2.3. Estimador de Potencia y Energía 33

4. PERFIL DE POTENCIA DE LAS INSTRUCCIONES 35

4.1. Modelo para la Estimación de la Potencia y la Energía 35
4.1.1. Corriente Base 36
4.1.2. Corriente de Conmutación 37

4.1.3. Corriente Extra . . . 39
4.1.4. Estimación de la Potencia y Energía . . . 42
4.2. Clasificación de las Instrucciones para la Medida . . . 43
4.3. Medidas Preliminares . . . 45
4.4. Implementación del Sistema de Medida . . . 46
4.4.1. Componente *Hardware* . . . 48
4.4.2. Componente *Software* . . . 49

5. ANALISIS ESTATICO DE PROGRAMAS . . . 54

5.1. Gráfica de Control de Flujo (CFG) . . . 55
5.2. Predicción Estática de Saltos . . . 58
5.3. Análisis de Ciclos . . . 63
5.3.1. Detección de Ciclos . . . 63
5.3.2. Estimación de las Iteraciones de un Ciclo . . . 64
5.4. Rutas de Ejecución . . . 73

6. METODOLOGÍA DE ESTIMACIÓN DE POTENCIA Y ENERGÍA . . . 77

6.1. Estimación del Tiempo de Ejecución . . . 78
6.1.1. Unidad Lectora . . . 79
6.1.2. Unidad IQ . . . 79
6.1.3. Unidad Despacho . . . 80
6.1.4. Unidad BPU . . . 80
6.1.5. Unidades de Ejecución . . . 81
6.1.6. Unidad Temporal . . . 81
6.1.7. Unidad CQ . . . 82
6.1.8. Unidad Completar . . . 82
6.1.9. Unidad Registros Arquitecturales . . . 82
6.1.10. Programa General . . . 83
6.2. Estimación del Costo de Energía para cada Nodo del Programa . . . 83
6.2.1. Extracción de Instrucciones . . . 83
6.2.2. Costo de Corriente Base y Conmutación . . . 83
6.2.3. Costo de Corriente por Efectos Extra . . . 84
6.2.4. Estimación de la energía del nodo . . . 85
6.3. Estimación de Potencia y Energía del Programa . . . 85
6.3.1. Potencia y Energía de un Ciclo . . . 88
6.3.2. Potencia y Energía del Programa . . . 90
6.3.3. Resumen . . . 91

7. RESULTADOS Y ANÁLISIS . . . 93

7.1. Perfíl de Potencia de las Instrucciones . . . 93
7.2. Hipótesis . . . 93
7.3. Sistema Experimental . . . 95
7.4. Programas de Prueba . . . 96
7.4.1. FFT . . . 97

7.4.2. FIR . . . 97
7.4.3. Raíz cuadrada entera . . . 98
7.4.4. Controlador de una fuente de poder . . . 98
7.4.5. Control de un cargador de batería . . . 99
7.5. Resultados y Análisis . . . 99
7.5.1. FFT . . . 99
7.5.2. Filtro FIR . . . 100
7.5.3. Programa Raíz Cuadrada . . . 101
7.5.4. Programa Controlador Fuente de Poder . . . 102
7.5.5. Control de un cargador de batería . . . 103
7.6. Prueba de Hipótesis . . . 104
7.7. Limitaciones del Proyecto . . . 105

8. CONTRIBUCIONES . . . 107

9. CONCLUSIONES Y TRABAJO FUTURO . . . 109

9.1. Conclusiones . . . 109
9.2. Trabajo Futuro . . . 110

BIBLIOGRAFÍA . . . 112

APENDICES . . . 116

A. VALORES DE CORRIENTE ASIGNADOS A LAS INSTRUCCIONES . 117

B. COSTOS DE CORRIENTE POR CONMUTACIÓN . . . 122

C. INSTRUCCIONES NO INCLUIDAS EN EL PERFIL DE POTENCIA . 130

Índice de Tablas

Tabla pagina

2–1. Comparación entre los diferentes modelos de potencia. 28

4–1. Modos de direccionamiento. 45

5–1. Valores de probabilidad asignados por cada heurística. 60

7–1. Resultados para el programa FFT. 100

7–2. Resultados para el programa filtro FIR. 101

7–3. Resultados para el programa raíz cuadrada. 101

7–4. Resultados para el programa controlador de fuente de poder (FSM). . 103

7–5. Resultados para el programa de control de cargador de batería. 104

A–1. Medidas de corriente base para las instrucciones enteras del PPC603e. 117

A–2. Medidas de corriente base para las instrucciones enteras del PPC603e.(Cont.) 118

A–3. Medidas de corriente base para las instrucciones punto flotante del PPC603e. 119

A–4. Medidas de corriente base para las instrucciones load-store del PPC603e. 120

A–5. Medidas de corriente base para las instrucciones de sistema del PPC603e. 121

A–6. Medidas de corriente base para las instrucciones de salto del PPC603e. 121

B–1. Diferencia de corriente por ejecutar el par de instrucciones respecto al costo base (1) 122

B–2. Diferencia de corriente por ejecutar el par de instrucciones respecto al costo base (2) 123

B–3. Diferencia de corriente por ejecutar el par de instrucciones respecto al costo base (3) 124

B–4. Diferencia de corriente por ejecutar el par de instrucciones respecto al costo base (4) 125

B–5. Diferencia de corriente por ejecutar el par de instrucciones respecto al costo base (5) 126

B–6. Diferencia de corriente por ejecutar el par de instrucciones respecto al costo base (6) 127

B–7. Diferencia de corriente por ejecutar el par de instrucciones respecto al costo base (7) 128

B–8. Diferencia de corriente por ejecutar el par de instrucciones respecto al costo base (8) 129

C–1. Instrucciones no consideradas para la medida. 130

Índice de Figuras

Figura pagina

3–1. Mapa cognitivo de la metodología del proyecto 30
4–1. Estructura empleada para la medida de corriente de una instrucción . 47
4–2. Diagrama en bloques del sistema de medida 48
4–3. Modelo de programación del PowerPC 603e. 51
4–4. Mapa de memoria del sistema Excimer extraído del manual de usuario. 51
5–1. Esquema de análisis estático del proyecto 55
5–2. Ejemplo de CFG generado a partir de código ensamblador 58
5–3. Ejemplo de identificación de ciclo y nodos iterbranch. 65
5–4. Ejemplo de rutas obtenidas a partir del algoritmo presentado. 75
6–1. Esquema del estimador de potencia y energía 78
6–2. Diagrama en bloques del modelo funcional del PowerPC 603e 79
6–3. Ejemplo de un CFG 86
7–1. Esquema del estimador de potencia y energía 95

Capítulo 1
INTRODUCCIÓN

Debido a los requerimientos de las nuevas aplicaciones, los desarrolladores de sistemas embebidos se han visto en la necesidad de optimizar sus diseños en tiempo, espacio y energía, de modo que puedan proveer soluciones que respondan de una forma aceptable a dichos requerimientos. Optimizar el consumo de potencia y energía tradicionalmente ha sido orientado hacia el *hardware*, sin embargo, actualmente también se orienta hacia el *software*, debido a la gran cantidad de sistemas que utilizan microprocesadores y muchas de las funciones son implementadas como programas. Es así que se han desarrollado una serie de técnicas para la optimización de código para el ahorro de energía y de técnicas para estimar el consumo de potencia y energía del microprocesador. Tradicionalmente la estimación de energía se realiza a nivel de componentes básicos como transistores y compuertas, sin embargo, para un microprocesador, este proceso se convierte en una tarea muy tediosa, dedibo a la gran cantidad de dispositivos que contiene y a la necesidad de contar con los datos reales para determinar la actividad de los diferentes componentes a lo largo del programa [1]. Así, se han desarrollado nuevas metodologías que realizan abstracciones de los diferentes componentes del microprocesador, facilitando la estimación de la potencia y la energía, a expensas de un aumento en el error del valor estimado [2] [3] [4]. Normalmente estas técnicas generan modelos de energía del microprocesador y se apoyan en las instrucciones del programa que ejecuta el microprocesador para obtener la información de la actividad de los componentes de interés para el modelo, y así poder determinar el consumo de potencia y energía. En

general, para aplicar estas técnicas, se requiere del uso de simuladores especializados que ejecuten el programa y lleven cuenta de las acciones ocurridas, o de sistemas de evaluación en *hardware*, que permitan obtener la información deseada.

Este trabajo presenta una metodología alterna, que permite estimar el consumo de energía de modo que no se requiera la ejecución del programa en un entorno especializado o en un *hardware* especifíco. El objetivo de este trabajo consiste en utilizar un modelo de energía del microprocesador, combinado con técnicas de análisis estático de código para predecir el comportamiento del programa, con lo cual se minimiza la necesidad de datos reales y la necesidad de ejecución en tiempo real o simulación con herramientas especializadas, del programa bajo medida.

El proyecto consta de tres componentes fundamentales. El primer componente es un modelo de potencia del microprocesador utilizado en este proyecto. El segundo componente es un conjunto de herramientas de análisis estático de código, que permiten estimar el comportamiento del programa con base en métodos probabilisticos. También permiten determinar otros parámetros importantes como la cantidad de iteraciones de los ciclos del programa. El tercer componente integra la información de los componentes anteriores para realizar un estimado del tiempo de ejecución y energía a nivel de las rutas del programa. Finalmente, la combinación de los costos por ruta permite calcular la potencia y energía consumida por el microprocesador al ejecutar el programa.

Los resultados obtenidos empleando esta metodología muestran que es posible estimar el consumo de energía con un error menor al 20 % y la potencia con un error menor al 12 %. Adicionalmente, la cantidad de datos suministrados al estimador son mínimos y no se requiere el uso de herramientas especializadas adicionales para obtener el valor estimado de potencia y energía. Finalmente, gracias a la forma como se desarrolló esta metodología, es posible integrar esta herramienta en forma directa y eficiente en otras herramientas de diseño de sistemas empotrados, donde el código

de un programa es modificado por medio de diferentes técnicas de optimización y es necesario estimar el consumo de potencia y energía en reiteradas ocasiones, para evaluar la efectividad de las técnicas aplicadas.

La estructura de este documento es la siguiente. El capítulo 2 presenta el trabajo previo realizado a nivel de estimación de potencia en microprocesadores desde el punto de vista del código ejecutado y presenta trabajos en técnicas de análisis de código orientadas a estimar comportamiento o características de un programa. El capítulo 3 presenta los objetivos y la metodología adoptada para el desarrollo de la herramienta de estimación de consumo de potencia y energía. El capítulo 4 presenta el modelo de potencia y energía adoptado en este proyecto. El capítulo 5 presenta las técnicas de análisis estático de código utilizadas para estimar el comportamiento del programa y otros aspectos importantes. El capítulo 6 presenta cómo se utiliza el modelo de energía y el análisis de código para estimar el consumo de potencia de un programa bajo prueba. El capítulo 7 presenta los resultados obtenidos. El capítulo 8 presenta los aportes de este trabajo y el capítulo 9 presenta las conclusiones y trabajo futuro del proyecto.

Capítulo 2
TRABAJO PREVIO

Este capítulo presenta trabajos publicados que están relacionados con la estimación de la potencia consumida por un microprocesador al ejecutar un programa. Este capítulo se divide en tres partes. La primera parte describe trabajos reportados sobre la estimación del consumo de potencia y energía en microprocesadores. La segunda parte describe métodos de análisis de algoritmos para estimar el comportamiento del programa. La combinación de estas partes conforma la base del proyecto que se presenta en este documento. La última parte realiza una discusión sobre la relevancia del trabajo presentado con relación a los trabajos analizados.

2.1. Estimación del Consumo de Potencia y Energía de Microprocesadores

En general, los métodos para estimación de potencia y energía en microprocesadores se basan en la generación de un modelo del dispositivo bajo estudio, el cual se complementa con medidas reales de su consumo. Una vez realizado dicho modelo, se toma el programa bajo medida y se procede a obtener estadísticas de los parámetros del modelo, con lo cual se puede obtener el valor estimado de consumo de potencia y energía que tendrá el microprocesador cuando ejecute dicho programa. En general, se tienen los modelos de potencia a nivel de circuito, a nivel de bloques funcionales y a nivel de instrucciones. Para generar las estadísticas a partir del programa bajo prueba normalmente se utiliza simulación. A continuación se describen dichos métodos.

2.1.1. Potencia a Nivel de Circuito

Esta metodología se basa en la estimación de potencia y energía a nivel de los componentes de *Hardware*. Esto quiere decir que se determina el consumo de energía para cada circuito que compone el microprocesador. Para tal efecto, normalmente se utiliza un simulador de circuitos como SPICE. Los resultados que se obtienen de esta metodología son los mas precisos en comparación con los otros modelos, sin embargo, la simulación requiere de un computador poderoso y normalmente puede consumir mucho tiempo para estimar el valor de energía consumido para un programa dado. A continuación se presenta un resumen de algunos trabajos realizados.

Mehta y otros proponen un modelo de simulación circuital llamado modelo de caja negra (black box model)[5]. En este modelo, celdas básicas (ejemplo, sumadores completos en un circuito de suma, flip-flops en un registro, etc.) son caracterizadas respecto al consumo de potencia. Con base en esto, se calcula el consumo de potencia para unidades de circuito mayores, como sumadores, ALUs, registros, etc. El modelo de consumo de potencia se complementa con la interacción de las diferentes unidades circuitales que componen el microprocesador. Es importante anotar que los autores incluyen como valores de entrada las instrucciones, las constantes y los datos inmediatos, pues según estos valores, diferentes circuitos serán activados. Esto dificulta el cálculo de potencia y energía en dicho modelo, pues debe incluir en su modelo la relación datos de entrada y circuito activado.

La implementación de este proyecto se hizo con un modelo experimental de microprocesador, capaz de ejecutar 16 instrucciones. El programa estima el consumo de energía en función de las microinstrucciones, estado de la unidad de control, estado de la ALU y otras unidades y los datos actuales y anteriores almacenados en registros y en los buses. Se realizaron pruebas con tres programas: el máximo divisor común, un multiplicador y un generador de números de Fibonacci. Mehta y asociados reportaron un error en sus estimados entre 9 % y 13 %.

Una variante de esta metodología es utilizar una descripción en lenguaje de *hardware* del microprocesador. En este caso, el simulador determina parámetros como la actividad de conmutación, con lo cual se puede estimar el consumo de potencia y energía. Al igual que el en caso de circuito, este proceso consume mucho tiempo y gran cantidad de recursos del computador. En adición, se tiene que la síntesis de la descripción del microprocesador puede variar según la herramienta que se utilice para la síntesis. A continuación se presentan algunos trabajos que han tomado esta metodología.

Chakrabarti y Gaitonde desarrollaron un modelo del microprocesador bajo prueba a nivel de compuertas lógicas y lenguaje de descripción de *hardware* [6]. Para estimar el consumo de potencia y energía, se utilizaron varias herramientas ofrecidas por el fabricante del microprocesador. Los datos utilizados como argumentos para las instrucciones fueron elegidos aleatoriamente y en otros casos estaban correlacionados. A partir de esta información, los autores concluyeron que para la gran mayoría de las instrucciones, no existe variación de energía notoria si los datos están correlacionados. Los resultados de este trabajo, muestran que la energía promedio no presenta dependencia con los datos de entrada y se obtuvo un error no mayor del 12 % en las pruebas realizadas.

Caldari y otros presentan un modelo que calcula la energía consumida por el microprocesador con base en la estimación del número promedio de nodos de entrada y salida del circuito, la actividad de conmutación y la frecuencia de operación del dispositivo [1]. Este método utiliza como entrada la descripción funcional del microprocesador en VHDL y utiliza una técnica que estima la actividad de conmutación promedio para el circuito. Es importante destacar que se hace un análisis dividido en dos partes: la primera es la estimación de conmutación para los circuitos combinacionales y la segunda es la estimación de conmutación para los flip-flops.

El algoritmo presentado trata de generar nodos adicionales en la descripción funcional del circuito para lograr el valor deseado. Según la información presentada en este trabajo, a medida que aumenta el número de nodos del circuito, el error en la estimación de la energía consumida disminuye.

Este método fue aplicado a un dispositivo manejador de bus I2C. Se compararon los resultados obtenidos en PSpice con los del modelo y con la información presentada por el sintetizador del circuito. El error obtenido cambia de 25 % con cero nodos adicionales a menos de 5 % con mas de diez nodos adicionales.

Abrar presenta un trabajo que es combinación de un modelo de conmutación y un modelo de consumo de corriente a nivel de instrucciones [7]. A partir de estas medidas se genera un macromodelo que permite estimar el consumo de energía del microprocesador. La información que alimenta al modelo es obtenida de la ejcución de programa por medio de una herramienta comercial. El error presentado es del 10 % respecto al valor actual de consumo de energía.

En resumen, para esta metodología y los ejemplos presentados, la principal ventaja es la precisión en sus resultados. Adicionalmente, el modelo desarrollado puede ser aprovechado para realizar observaciones de los diferentes componentes del microprocesador y realizar optimizaciones antes de iniciar su fabricación. Por último, se puede estudiar el efecto de diferentes implementaciones del mismo microprocesador en cuanto al consumo de potencia y energía.

Las desventajas fundamentales de esta metodología incluyen la lentitud del proceso de estimación y la gran cantidad de recurso requerido para computar los valores de potencia y energía. También se tiene que es necesario generar un modelo diferente para cada nuevo microprocesador que se utilice, lo cual normalmente consume una gran cantidad de tiempo. Lógicamente, los resultados que se obtienen con esta metodología, solo aplican a los microprocesadores modelados.

2.1.2. Potencia a Nivel de Bloques Funcionales

Esta metodología crea abstracciones de la arquitectura de un microprocesador a nivel de bloques funcionales. Para obtener la información de energía consumida, estos bloques se caracterizan con información real del microprocesador bajo prueba. Normalmente se tienen dos variantes de esta metodología. Una implementa bloques funcionales a partir de los componentes de *hardware* del microprocesador. La otra crea bloques funcionales a partir de las acciones que realiza el microprocesador para ejecutar dicha instrucción. A continuación se presentan algunos trabajos relacionados con esta metodología.

Laurent y otros presentan una metodología donde dividen al microprocesador en bloques funcionales basados en la arquitectura del microprocesador [2]. Luego se seleccionó una serie de parámetros con los cuales se puede caracterizar la energía consumida. Se generaron dos tipos de parámetros, algorítmicos y arquitecturales. Los parámetros algoritmicos dependen del programa ejecutado, como la razón acierto-desacierto de la memoria cache, mientras que los arquitecturales, dependen de la configuración del microprocesador, como la frecuencia del reloj.

Una vez definidos los bloques y los parámetros, se procede mediante metodos para ajuste de curvas a obtener expresiones que puedan relacionar los parámetros con la potencia consumida. El error presentado por los autores al comparar los resultados con medidas reales está alrededor del 2.4 %.

Brandolese y otros presentan una metodología donde generan bloques funcionales de un microprocesador basado en las actividades llevadas a cabo por un microprocesador genérico, para ejecutar una instrucción [3]. El proceso de generación del modelo se realiza en tres pasos. El primero es descomponer al microprocesador en funcionalidades que sean independientes entre si. Se proponen como funcionalidades típicas las operaciones aritméticas, acceso a registros, acceso a memoria, los saltos y leer y decodificar instrucciones. El segundo paso es identificar la correspondencia

entre las instrucciones y las diferentes funcionalidades del microprocesador; es decir, cuáles funcionalidades se requieren para procesar la instrucción. El tercer paso consiste en estimar el consumo de potencia y energía para cada funcionalidad del microprocesador. En este paso se utilizó el método de mínimos cuadrados, dando como resultado un sistema lineal de ecuaciones. Este sistema se utiliza para realizar el calculo de energía y potencia de un programa dado. El error obtenido, comparado con la medida actual es alrededor del 9 %.

Los autores argumentan que esta metodología se puede utilizar para generar un modelo genérico de microprocesador, utilizando instrucciones de referencia. El error obtenido al probar esta idea con tres microprocesadores fue alrededor del 20 %.

Chakraborty presenta un artículo donde se describe una metodología para estimar el consumo de potencia de dispositivos IP (*Intellectual Property*), como microprocesadores y DSPs, en sistemas en un circuito integrado (SOC) [8]. Este modelo determina la potencia como una función de eventos de conmutación en los límites del dispositivo IP y los estados internos del mismo. Gracias a esto, se puede estimar el costo de potencia del dispositivo IP en el contexto del funcionamiento del sistema en un chip. El error reportado para esta metodología oscila entre un 20 % y 30 %, pero tiene la ventaja que se puede utilizar a varios niveles de abstracción, desde un nivel de sistema electrónico hasta *layout*.

Yunsi y otros presentan una metodología aplicada a microprocesadores expandibles, donde se generan macromodelos a partir de parámetros a nivel de instrucciones para las partes fijas del microprocesador y modelos estructurales para las partes no fijas del microprocesador [9]. Los macromodelos son obtenidos utilizando analisis de regresión. Los datos para que el macromodelo genere sus estimados de potencia y energía son obtenidos de la simualción del programa de prueba. El error publicado es del 3.3 % respecto a una herramienta comercial que trabaja sobre una versión RTL del microprocesador.

Natarajan y otros desarrollaron un modelo de un microprocesador que puede presentar consumo de potencia en las diferentes etapas del microprocesador [10]. Para esto, se utilizó una herramienta comercial donde se implementaron modelos de los diferentes componentes del microprocesador. Así, la potencia estimada es a nivel de ciclo. A partir de este modelo se obervó el comportamiento del microprocesador al ejecutar un programa y se evaluaron dos técnicas implementadas en el microprocesador para mejorar su funcionamiento. El trabajo reporta que se pierde un 17 % de energía en componentes subutilizados y un 6 % de la energía en predicciones erroneas de la cpu.

En resumen, para esta metodología y los ejemplos presentados, se tiene como principal ventaja la reduccion del tiempo de caracterizacion del microprocesador bajo estudio. Esta metodología es la mas rápida de las tres que se discuten en este capítulo.

Las desvetaja de esta metodología es el aumento en el error relativo, el cual depende de cómo se implementa el modelo del microprocesador bajo estudio. Al igual que las otras metodologías, los resultados que se obtienen, sirven principalmente para los microprocesadores modelados, sin embargo, como lo indican Brandolese y otros, es posible obtener cierta generalización con esta metodología.

2.1.3. Potencia a Nivel de Instrucciones

Esta metodología representa al microprocesador como una caja negra y genera un modelo que estima un valor de potencia promedio para cada instrucción ejecutable. La base de esta idea se presenta a contiuación. Las ecuaciones 2.1 y 2.2 son las expresiones para determinar la energía y potencia promedio consumida por un sistema. A partir de estas ecuaciones, se puede obtener la ecuación 2.3, de donde se puede deducir que una forma de estimar el consumo de energía en un microprocesador es midiendo la potencia promedio consumida durante el tiempo que este toma

en ejecutar el programa bajo prueba.

$$E(t) = \int_{t_0}^{t_0+T} P(t)dt \tag{2.1}$$

$$P_{ave} = \frac{1}{T}\int_{t_0}^{t_0+T} P(t)dt \tag{2.2}$$

$$E = T \cdot P_{ave} \tag{2.3}$$

La potencia promedio puede ser determinada con la expresión 2.4.

$$P_{ave} = V_{cc} \cdot I_{ave} \tag{2.4}$$

En un circuito digital, el valor de la fuente de polarización se supone constante, con lo cual, la potencia promedio queda en función de la corriente promedio que consume el microprocesador durante la ejecución de un programa. Así, estimar la potencia promedio se reduce a obtener la corriente promedio consumida por el microprocesador.

Si se considera al microprocesador un dispositivo discreto, cuyo componente atómico son las instrucciones, se tiene que para cada instrucción que ejecute el microprocesador se puede obtener un valor promedio del consumo de corriente, proporcional a la actividad que realiza dicho microprocesador para procesar una instrucción dada. Este valor de corriente promedio estará relacionado con los circuitos que se activan en el microprocesador para realizar la función indicada. A continuación se presenta un resumen de algunos trabajos realizados.

Tiwari y asociados plantearon el primer método para relacionar la energía consumida con el conjunto de instrucciones ejecutadas en un programa [11] [12]. En este modelo se consideró las instrucciones como la interfaz que conecta la funcionalidad de un sistema basado en microprocesador y el *hardware* que lo compone.

Para estimar el consumo de potencia, se midió la corriente promedio que consume el microprocesador por instrucción ejecutada. Para esto, conectó un amperímetro en serie con la línea de alimentación del microprocesador. Según Tiwari y asociados, el amperímetro puede realizar la medida promedio de corriente deseada, siempre y cuando dicha señal sea periodica y este periodo sea menor a la ventana de integración del amperímetro. Para obtener una medida estable, la instrucción bajo prueba fue puesta en un ciclo con una gran cantidad de iteraciones, donde hubiesen muchas instancias de la instrucción bajo prueba, para minimizar el efecto de otras instrucciones, como la de fin de ciclo.

El modelo propuesto está compuesto por dos componentes principales. La primera es el costo base de la instrucción, lo cual es el consumo de energía del microprocesador por ejecutar sólo esa instrucción. Es importante mencionar que el argumento de una instrucción puede variar dependiendo del programa en ejecución y los modos de direccionamiento empleado. Así, una instrucción de suma puede tener como argumento dos registros o un registro y un dato inmediato. Los registros pueden tener un valor que sólo se conoce en tiempo de ejecución. Esto implica que se deben realizar multiples pruebas a una misma instrucción para garantizar el valor promedio obtenido. El segundo componente es la actividad de conmutación. Esto es, el consumo de potencia por transiciones en los bits de los circuitos del microprocesador. Esta componente fue considerada sólo por parejas de instrucciones.

En adición a estas componentes, se consideró un consumo de energía adicional debido a efectos externos a la ejecución del programa. Estos efectos son los retardos producidos por desaciertos en la memoria cache, retrasos en la ejecución de instrucciones debido a que el recurso está ocupado y por desaciertos en la predicción de los saltos. Estos inconvenientes normalmente se traducen en consumo de tiempo, lo cual implica pérdidas en energía del sistema. Estos efectos se midieron y se agregaron al resultado final de la energía estimada del programa.

Tiwari y asociados tambien realizaron un estudio de consumo de potencia del subsistema de memoria. La memoria es un dispositivo de gran importancia en un sistema embebido debido a la información que guarda. Ella está relacionada con el consumo de energía de microprocesador, ya que existe retardo entre la solicitud de datos y la llegada de los mismos. Este retardo se traduce en perdida de energía en un sistema embebido.

El modelo fue aplicado a un microprocesador Intel 486DX2, a un microprocesador Fujitsu SPARC y un microprocesador orientado al procesamiento de señales. En los microprocesadores generales se concluyó que sólo el costo base es significativo al estimar el consumo de energía. El caso contrario sucedió en el microprocesador orientado al procesamiento de señales, donde el costo por actividad fue significativo.

Tiwari y asociados concluyen que su propuesta para estimar consumo de potencia y energía es valida para los procesadores que se han sometido a este proceso. Sin embargo, los resultados que se obtienen para un microprocesador dificilmente sirven para otro microprocesador, debido principalmente a la diferencia en la arquitectura empleada en su construcción.

Russell y Jacome realizaron un estudio basado en el trabajo presentado por Tiwari y asociados [13]. En este trabajo se propone asignar un único valor constante de consumo de potencia a cada instrucción del microprocesador bajo prueba. Esta idea se basa en que el consumo de energía debido a la actividad por conmutación es mínima y que el costo base por instrucción se mantiene aproximadamente constante para la familia de microprocesadores seleccionados.

Para medir la potencia consumida por el programa, se insertó una resistencia en serie con la fuente de poder del microprocesador y se realizó un cálculo indirecto de la potencia, tomando como medidas la diferencia de voltaje en la resistencia y el valor de la resistencia. Las medidas fueron realizas con un osciloscopio digital que podía

en el mismo instante que tomaba la medida, calcular y graficar la potencia instantanea del microprocesador. Las múltiples medidas obtenidas permiten determinar el valor promedio del consumo de potencia por instrucción. Este proceso es similar al planteado por Levy para medir el consumo de potencia de un microprocesador [14].

Con base en los resultados, los cuales presentaron una variación en consumo de potencia relativamente pequeña, se dedujo que no era necesario considerar las instrucciones en forma individual para estimar el consumo de energía y se asignó un unico valor de consumo de potencia por ciclo, reportando un error del 8 %.

Para probar la hipótesis, se realizaron medidas en dos microprocesadores de la familia Intel i960 la cual es una arquitectura RISC de 32 bits. En las pruebas no todas las instrucciones fueron tenidas en cuenta. Sólo se utilizaron las instrucciones más comúnes en los programas de usuario. Así, instrucciones que controlan la cache, que modifican el estado del microprocesador y otras, no se incluyeron en el perfil de potencia final. Para cada instrucción se hicieron variaciones en el uso de diferentes registros, diferentes datos inmediatos y otras opciones.

A partir de sus resultados, Russell y asociados dedujeron que el mejor método para optimizar el consumo de energía, es reducir el tiempo de ejecución de un programa. Una observación importante a este trabajo, Russell y asociados indican que no se presentó diferencia significativa entre las medidas obtenidas (potencia promedio) con el osciloscopio y con el multímetro, pero con el primero se puede obtener mayor información acerca de los valores instantaneos del consumo de potencia de las instrucciones.

Dongkun y asociados construyeron un sistema basado en la idea presentada por Tiwari y asociados [15]. La diferencia radica principalmente en dos aspectos. El primero es el método utilizado para medir la energía consumida por el procesador y el segundo es la arquitectura general del sistema de medida de consumo de energía del procesador.

El método de medida utilizado se basa en la carga y descarga de un capacitor. Se creó un sistema que permanentemente carga el capacitor con la fuente de poder y luego conecta en paralelo dicho capacitor con el microprocesador. La descarga del capacitor, reflejada en una caida de voltaje, está relacionada con la energía consumida por el microprocesador al ejecutar una instrucción. Así, este método permite medir directamente el consumo de energía por instrucción, pero no pemite visualizar las variaciones instantaneas de dicho consumo.

La arquitectura del sistema de medida esta basada en tres componentes. El primer componente se encarga de realizar la estimación del consumo de energía del microprocesador bajo prueba y de realizar la transmisión de la información recolectada al segundo componente del sistema. El segundo componente es el módulo de analisis de energía, el cual asocia el perfil de consumo de energía con el código del programa bajo prueba. El tercer componente es la interfaz de usuario que muestra gráficas, análisis de código y otras utilizades para el usuario.

Laopoulos y otros propusieron un método diferente para medir la corriente consumida por el microprocesador, utilizando como elemento sensor un espejo de corriente que está permanentemente conectado a la línea de polarización del microprocesador [16]. Este sistema influye menos en las medidas de lo que influye el método basado en resistencia serie. En adición, permite un acople para los instrumentos sin perturbar la polarización del microprocesador.

Este método es empleado como medio para generar la metodología para estimar el consumo de energía del microprocesador ARM7TDMI [4]. El modelo propuesto consta de tres componentes principales. El primero es el costo base de la instrucción. El segundo son los diferentes argumentos de la instrucción y el tercero es el efecto de inter-instrucción. Adicional a estos componentes, se agrega un costo adicional constante para compensar atrasos generados por instrucciones de salto que no se pueden resolver.

El primer componente, costo base, se modela como un valor constante para cada instrucción. El segundo componente, los argumentos de la instrucción, se modelan como ecuaciones lineales que dependen de la cantidad de unos (1) que contienen dichos argumentos. El tercer componente, inter-instrucción, se modela como un valor obtenido directamente de las medidas y está en un valor medio de 15 % del costo base.

Los resultados presentados por Laopoulos y asociados muestran un error del 6 % respecto a medidas directas del consumo de energía para algunos programas de prueba.

Talarico y otros presentan un sistema para estimación de potencia a partir de modelos de alto nivel, en lugar de modelos basados en compuertas o transistores [17]. La metodología presentada emplea un modelo para el microprocesador, el sistema de memoria y los periféricos del sistema empotrado. La información para los modelos se extrae de la simulación del programa, y los costos de potencia se determinan midiendo el consumo de corriente por instrucción ejecutada. La eficiencia de la metodología se evaluó con la ejecución de 20 programas de prueba, de donde obtuvo un error medio de 20 % con una desviación estándar de 8 %.

Haid y otros presentan una metodología alterna para estimar el consumo de energía de un microprocesador [18]. Se trata de un coprocesador que permanentemente monitorea al microprocesador. Para esto, mide el consumo de corriente del microprocesador cuando éste ejecuta un programa. El fin último de este dispositivo es ser un indicador para el microprocesador de su consumo de energía, de modo que pueda aplicar políticas de ahorro al programa en ejecución. El error presentado es del 5 % del valor actual de la medida de energía.

En general, para esta metodología y para los ejemplos presentados, se tiene que la principal ventaja es la precisión en sus resultados, aunque dicha precisión es

menor que la obtenida con el método de circuito. La otra ventaja es la reducción en el tiempo requerido para obtener dichos resultados, respecto al método de circuito.

La desvetaja de esta metodología es el requerimiento de caracterización de la gran mayoría de las instrucciones del microprocesador, lo cual consume tiempo. Aunque, al igual que las metodologías anteriores, es un proceso que se realiza una sola vez.

2.1.4. Metodología para la Generación de Estadisticas de los Parámetros de un Modelo

Las tres secciones anteriores explican cómo se genera un modelo de potencia y energía para estimar el consumo que tiene un microprocesador cuando procesa un programa. Para utilizar estos modelos se hace necesario obtener información específica del programa bajo prueba, la cual se puede utilizar con los modelos anteriores para la estimaciín de la energía. Algunos ejemplos de la información requerida son.

- Circuitos que se activan por instrucción.
- Datos introducidos al programa.
- Cantidad de veces que se repite una instrucción.
- Trazas del programa.

En forma general, el método adoptado por los investigadores para obtener dicha información es la simulación del programa, donde el simulador utilizado registra la información necesaria para los modelos de potencia. En algunos casos se tiene que los simuladores que ofrecen los fabricantes de microprocesadores son suficientes para obtener la información necesaria. En otros casos se modifican dichos simuladores y en algunos casos, se crea el simulador que responda a las necesidades de la metodología utilizada para la estimación de potencia y energía.

2.2. Analisis Estático de Código

Como se ha expresado en la seccion anterior, la mayoria de los sistemas que estiman consumo de potencia y energía se apoyan en las estadistiscas obtenidas de un simulador del microprocesador, cuyos resultados son los datos de entrada para los modelos de energía desarrollados. A continuación se presentan trabajos realizados en análisis estático de código, los cuales originalmente no tienen como fin la estimación de potencia o energía consumida por el microprocesador, pero son la base para la metodología propuesta en este proyecto.

En general, las herramientas de analisis estático de código se dividen en cuatro grupos. La primera herramienta convierte código a un modelo gráfico, el cual se presta para la realización de analisis del comportamiento del programa. La segunda herramienta analiza los saltos en un programa. La tercera analiza los ciclos de un programa y la última herramienta busca realizar un estimado del tiempo de ejecución de un programa. Es importante recalcar que la idea base de estas herramientas es obtener toda la información posible del comportamiento de un programa, con solo analizar las instrucciones que lo componen y algunos valores adicionales.

2.2.1. Gráfica de Control de Flujo

La grafica de control de flujo (CFG) es una técnica para representar el comportamiento de un programa, de una forma independiente al lenguaje en el cual ha sido escrito dicho programa. Esto garantiza que el proceso posterior de análisis sea aplicable a cualquier microprocesador que haya sido empleado para procesar el programa.

El uso de graficas de control de flujo es muy común en los compiladores. Dicha gráfica agrupa porciones de código en componentes llamados nodos y luego los interconecta con aristas, dependiendo de la trayectoria que siga el programa. En general, un nodo tiene como máximo dos aristas salientes y una o más aristas entrantes a

dicho nodo. A partir de esta gráfica se pueden identificar las diferentes rutas de ejecución de los programas, los ciclos y los diferentes caminos que pueden tomar los saltos en los programas.

Aho y otros en su libro Compiladores presentan en forma muy detallada como desarrollar las gráficas de control de flujo, a partir de algoritmos que identifican características especiales para los nodos y aristas pertenecientes ciclos y decisiones [19]. Es importante recalcar que estas técnicas son aplicadas a programas escritos en lenguaje ensamblador o similar.

2.2.2. Predicción de Saltos

La predicción de los saltos busca dar respuesta a cúal ruta seguirá un programa cuando se encuentre con una instrucción de decisión. Esto es necesario ya que al no tener los datos reales y los contenidos anteriores de las variables envueltas en el salto, no hay posibilidad de determinar exactamente la dirección de dicho salto. De acuerdo a un estudio realizado por Ball y Larus, se demostró que es posible hacer una predicción de cúal ruta tomará un programa cuando se encuentre con una decisión [20]. A continuación se presenta un resumen de su trabajo y de otros investigadores del tema.

Ball y Larus reportaron un conjunto de heurísticas para determinar la dirección de un salto, basadas en las instrucciones del programa y en el tipo de salto a realizar [20]. Para construir estas heurísticas, se recolectó información de los diferentes tipos de salto que puede ejecutar un microprocesador, referente al comportamiento de dichas instrucciones en muchos programas. Esta información se integró en un perfíl de dirección con los valores de probabilidad obtenidos. La información recolectada mostró que en muchos casos es posible hacer una correcta predicción del salto utilizando como base el perfíl de los tipos de salto. Esto se debe a que Ball y Larus observaron que los saltos tienden a tomar la misma dirección de salto con alta probabilidad y

esta dirección es la misma en diferentes programas. El mejor ejemplo para presentar es la instrucción de salto que forma parte de la decisión de un ciclo. En este caso, lo normal es que un ciclo tienda a ejecutarse muchas veces, asi que la dirección del salto que es más probable de ser tomada es aquella que se mantiene dentro del ciclo. Esto lo confirma el perfíl de los saltos, donde se ha encontrado un 90 % de aciertos en esta observación. Al aplicar las heurísticas a un programa, Ball y Larus encontraron que en muchos casos, más de una heurística puede ser aplicada a un salto. La solución propuesta fue la de aplicar las heurísticas en un orden preestablecido y detener el proceso al encontrar la primera heurística aplicable al salto. Para determinar dicho orden, se hizo una gran cantidad de experimentos, con diferentes combinaciones y se estimó el orden que mejores resultados presentase. Los resultados publicados por Ball y Larus presentan un error promedio del 26 %.

Larus y Wu realizaron mejoras a la metodología presentada por Ball y Larus [21]. En primer lugar, se optimizaron las heurísticas al separar las diferentes opciones de salto que se tienen en un ciclo. Así, se tienen heurísticas óptimizadas para el inicio de un ciclo y para finalizar un ciclo. Además, se eliminaron algunas heurísticas que no presentan un resultado significativo en su predicción.

En segundo lugar, se modificó el método de búsqueda y aplicación de heurísticas a un salto bajo análisis. La nueva metodología considera a todas las heurísticas que pueden ser aplicables a un salto, y las combina en un solo resultado que será el asignado a dicho salto. Para combinar las heurísticas, se utiliza el método estadístico de Demster-Shafer [21]. Esta teoría se resume en las siguientes ecuaciones presentadas por Larus y asociados en su trabajo.

$$P_{Salto} = P_{Actual-Salto} \oplus P_{Nueva-Salto} = \frac{u_S \cdot v_S}{u_S \cdot v_S + (1 - u_S) \cdot (1 - v_S)} \tag{2.5}$$

$$P_{No-Salto} = P_{Actual-Nosalto} \oplus P_{Nueva-Nosalto} = \frac{(1-u_{NS}) \cdot (1-v_{NS})}{u_{NS} \cdot v_{NS} + (1-u_{NS}) \cdot (1-v_{NS})} \tag{2.6}$$

Donde u corresponde al valor actual de probabilidad y v al valor de la nueva heurística aplicada al salto. Los resultados de estas expresiones hacen que el valor de la probabilidad asignada a un salto varíe en proporción a las probabilidades individuales de las heurísticas.

En tercer lugar, Larus y asociados presentaron una metodología para estimar las frecuencias de un salto. Por frecuencias de salto se debe entender la cantidad de veces, en promedio, que se ejecuta una instrucción de salto en un programa. Para estimar esto, se plantea una metodología donde se determina la frecuencia de un nodo a partir de la suma de las frecuencias de repetición de las arístas que llegan a ese nodo. La frecuencia de repetición de una arísta se calcula como la frecuencia de repetición del nodo origen, multiplicado por la probabilidad de la arísta en análisis. Los resultados publicados por Larus y asociados muestran un acierto del 82 % en sus predicciones, aunque este valor decrece en algunos programas.

Wong presenta un desarrollo de heurísticas similares a las anteriores, pero que se aplican a lenguajes de alto nivel [22]. Otra diferencia es que las heurísticas presentadas no están basadas en perfiles de las instrucciones de salto. Estas heurísticas han sido realizadas con base en los comportamientos normales de salto de cada instrucción del microprocesador y no se trabaja con valores de probabilidad, sino con pesos asignados por las reglas de cada heuristica. A continuación se describen las heurísticas.

- **Línea Base.** En este caso, el salto se supone siempre como tomado.
- **Aleatorio.** Se asigna igual probabilidad a tomar el salto o no. La elección se hace aleatoriamente.

- **Heurística S.** Si la comparación en un salto es de una igualdad, una AND lógica, una desigualdad con un número máximo o mínimo, la acción es una orden de impresión, un llamado a subrutina, un retorno de subrutina, se asigna un peso de -1 al salto. Si la comparación contiene un OR logico en la función, se asigna un peso de 1 al salto. El salto más positivo será supuesto como tomado.
- **Heurística SF.** Es una extensión a las heuristicas anteriores, que incluye funciones aritméticas.
- **Heurística SFM.** Analizan las macro-instrucciones que se encuentran en el código de alto nivel.

Los resultados presentados por Wong presentan un acierto del 75 % promedio, lo cual es similar a los resultados presentados por otros autores como Ball y Larus.

Hank y otros utilizan un conjunto de heurísticas y la grafica de contol de flujo para determinar las rutas de un programa que pueden ser optimizadas para incrementar la ejecución paralela de instrucciones [23]. En este trabajo, las heurísticas presentadas son similares a las de Ball. La efectividad reportada de las heurísticas utilizadas es del 86 %, lo cual coincide con la efectividad indicada en trabajos similares.

2.2.3. Estimación de las Iteraciones de un Ciclo

Esta técnica busca determinar dos aspectos fundamentales. El primero es determinar qué elementos pertenecen al ciclo bajo análisis, en especial la cabecera del ciclo. El segundo es determinar cuántas veces itera el ciclo antes de terminar. La importancia de estimar la cantidad de iteraciones en un ciclo se debe a que esta información esta directamente relacionada con el tiempo de ejecución del programa. A continuación se presentan trabajos realizados por investigadores, en este tema.

De Alba y Kaeli desarrollaron un sistema en *hardware* que pertime estimar en tiempo real la cantidad de iteraciones de un ciclo [24]. Su sistema se basa en el mismo

principio de los predictores de saltos dinámicos, donde la respuesta futura depende de los resultados de saltos previos. Se establecieron tres mecanismos de predicción.

- **último Valor.** El último número de iteraciones de un ciclo dado es utilizado para predecir el número de iteraciones de la siguiente ocasion que sea ejecutado el ciclo.
- **Paso.** Se determina la diferencia entre el número de iteraciones de las dos últimas visitas al ciclo. Esta diferencia es sumada al número de iteraciones de la última visita para predecir el futuro número de iteraciones.
- **Frecuencia.** Mantiene un conjunto de conteos de frecuencia de las ultimas n visitas al ciclo. El valor predicho de la iteración es el conteo que mayor valor tenga.

Los resultados presentados por De Alba y Kaeli muestran una efectividad mayor al 90 % en la predicción de las iteraciones del ciclo a ejecutar. La primera técnica (último valor), es la que mayor efectividad presenta, la cual es superior al 95 %. De este trabajo se puede observar que los ciclos tienden a realizar la misma cantidad de iteraciones siempre que el flujo del programa llega a ellos. En adición, se puede considerar que un solo estimado es suficiente para determinar la cantidad promedio de iteraciones del ciclo. Esta observación es importante porque nos dice que un primer estimado de las iteraciones de un ciclo, puede ser suficiente para describir el tiempo de ejecucion de dicho ciclo.

Healy y asociados proponen una metodología para estimar las iteraciones de un ciclo en un programa [25]. El primer paso del método consiste en analizar las instrucciones del programa para detectar los nodos que pertenecen al ciclo. Principalmente se busca el nodo inicial del ciclo (cabecera) y el nodo que reinicia el ciclo (nodo con una arista que se devuelve). Una vez identificados los componentes del ciclo, se procede a construir un árbol que muestre todas las rutas posibles en el ciclo. El segundo paso es determinar los saltos dentro del ciclo y determinar cuándo estos cambios modifican la dirección de salto. En este paso se identifican las variables del ciclo, valores inicial y final de la variable contadora, el tipo de operación

que se realiza en la comparación y el incremento de la variable contadora. El tercer paso es relacionar los saltos individuales del ciclo para estimar las iteraciones mínimas y máximas del ciclo bajo analisis. Este proceso inicia desde las hojas del árbol hasta llegar a la raiz. Las iteraciones asignadas al nodo raiz son las mismas que se asignarán al ciclo. Para generar esta relación, Healy y asociados presentan un total de seis reglas para estimar las iteraciones en nodos y aristas del árbol. El anterior procedimiento es valido para ciclos que tienen definidos los límites del ciclo (valores inicial, paso y final) y sin tener en cuenta el operador de igualdad. Para incluír el operador de igualdad, se busca una serie de condiciones adicionales, las cuales permiten procesar el salto. Para ciclos que no poseen un límite definido (expresiones), se establece un procedimiento mediante el cual, el usuario informa sobre los posibles valores de las variables que intervienen en el ciclo. Esta información la hace disponible el usuario del programa a través de una nueva directiva incluida en el lenguaje de alto nivel. Esta directiva explora el programa bajo prueba y prepara la información para la herramienta de análisis. Los resultados publicados por Healy y asociados demuestran que si se tiene un buen estimado de las variables que influyen en el ciclo, las iteraciones estimadas son bastante cercanas a las iteraciones reales. Sin embargo, el trabajo no presenta un valor numérico de la efectividad de la metodología desarrollada.

2.2.4. Estimación del Tiempo de Ejecución

La estimación del tiempo de ejecución de un programa es el proceso más crítico del análisis estático de código, debido a la naturaleza dinámica de la ejecución de instrucciones por parte del microprocesador. No es suficiente apoyarse en las tablas de tiempo de ejecución dadas por los fabricantes para realizar el estimado, pues existen efectos externos al programa que pueden aumentar el tiempo de ejecución. Por ejemplo, la dependencia de datos entre instrucciones es un factor que limita la

ejecución de una instrucción por necesitar los resultados de otra instrucción para continuar. Un acceso a memoria que no este almacenado en la cache requiere tiempo extra mientras el dato es accedido desde la memoria principal. Una instrucción de salto que ha sido erroneamente predicha por la unidad de predicción de saltos del microprocesador conlleva un retraso debido a que se deben expulsar las instrucciones predichas erroneamente e iniciar el proceso de las instrucciones debidas. Así, una estimación del tiempo de ejecución debe incluir varios parámetros para determinar con el menor error posible el tiempo deseado. Como se mencionó en la sección anterior, los investigadores actualmente utilizan simuladores para obtener el tiempo de ejecución de un programa. A continuación se presentan trabajos relacionados al tema.

Rapaka y Marculescu presentan una metodología para estimar el tiempo de ejecución basada en un simulador de dos niveles [26]. El primer nivel es un simulador detallado del microprocesador, el cual genera grán cantidad de información para su posterior uso en la estimación de energía consumida por el microprocesador. El segundo nivel es un simulador que asigna tiempos de ejecución determinados a una porción de código. El tiempo que asigna el simulador de segundo nivel lo ha generado previamente el simulador de primer nivel. Existe un programa de control que se encarga de conmutar entre los diferentes niveles, basado en la identificación de código repetido. Por código repetido, se refiere al código que ya ha sido simulado en detalle y que por lo tanto, no es necesario repetir dicha simulación.

Para identificar en qué momento se tiene un código repetido, se utiliza el concepto de *Puntos Calientes* (*hotspot* en inglés). Un punto caliente es un conjunto de bloques de código que se ejecutan juntos, la mayor parte del tiempo. Cuando un programa entra a un punto caliente, el microprocesador ejecuta sólamente el código perteneciente a dicho punto y raras veces sale de él, con lo cual se mantiene una alta

localidad temporal, haciendo que el programa se comporte de una manera predecible mientras este dentro de dicho punto. Un ejemplo de un punto caliente son las instrucciones dentro de un ciclo. Dependiendo de la implementación utilizada para detectar los puntos calientes, se tiene que el error en la estimación de tiempo oscila entre 2 % y 18 %.

Beltrame y otros presentan una metodología para estimar el tiempo de ejecucion de un programa, el cual utiliza dos componentes de tiempo, relacionados con las instrucciones que componen el programa bajo análisis [27]. El primer componente es el tiempo de ejecución de cada instrucción, indicado por el fabricante del microprocesador. El segundo componente consiste en determinar secuencias de instrucciones que generaran retardos adicionales en la ejecución de las instrucciones. Se identifican tres clases de retardos adicionales para este componente, estructurales, datos y control. Al identificar dichas secuencias en una traza de instrucciones, se puede determinar que ocurrirá un evento que retrasará la ejecución normal de las instrucciones, aumentando el tiempo de ejecución del programa.

Para identificar las secuencias de instrucciones que generarán retardos adicionales, es decir, la segunda componente, se analizaron conjuntos de instrucciones, de cantidad normalmente no mayor al tamaño del *pipeline* del microprocesador y se determinó en qué momento puede causar un retardo adicional. Para determinar el valor de dicho retardo, se realizarón mediciones en un microprocesador comercial. El error publicado es 5 % en promedio.

2.3. Discusión

Como se ha descrito en las secciones anteriores, se tienen diversas opciones para generar un modelo de energía para un microprocesador. Con excepción de modelo de potencia a nivel de circuito, las diferentes metodologías tratan de implementar

abstracciones del microprocesador que permitan reducir el esfuerzo de caracterizar la relación consumo de energía y programa en ejecución.

En general se observa que el error en los resultados obtenidos es relativamente pequeño comparado con el método de circuito, por lo tanto es necesario considerar otros factores en la elección de un modelo de energía, como la disponibilidad de la descripción funcional del microprocesador, en cuyo caso, una buena opción puede ser el modelo de bloques funcionales de *hardware*. Si no se posee una descripción del microprocesador, el modelo a nivel de instrucciones puede ser una buena opción para implementar el estimador de energía.

Adicional a los trabajos presentados en este capítulo, existen trabajo similares, donde se realizan variantes de los modelos presentados. Así, se encuentra que muchos investigadores se dedican a perfeccionar el proceso de obtención de las medidas de potencia y energía del microprocesador. Otros tratan de crear modelos con abstracciones de mayor nivel, por ejemplo, caracterizar funciones o subrutinas, y luego analizar el programa utilizando los resultados ya obtenidos para dichas funciones. Tambien se trata de realizar estimados a partir de lenguajes de alto nivel. Sin embargo, el precio que se paga con este tipo de generalizaciones normalmente es el aumento en el error en las estimaciones de energía consumida.

Una vez se tiene un modelo de energía, es necesario proveer información del programa que va a ejecutar el microprocesador, para realizar el estimado de energía consumida. La tendencia es utilizar simuladores para obtener la información requerida. La Tabla 2–1 presenta un resumen comparativo de las ventajas y desventajas de los diferentes modelos de energía presentados en este capítulo.

El proyecto que se presenta trata de buscar una alternativa a utilizar un simulador para obtener la información necesaria para el calculo de la energía. Se propone el uso de herramientas de análisis estático, las cuales normalmente no requieren la ejecución del programa para obtener la información buscada. Aunque normalmente

Tabla 2–1: Comparación entre los diferentes modelos de potencia.

Modelo	**Ventajas**	**Desventajas**
Circuito	Alta precisión.	Gran recurso computacional. Gran cantidad de tiempo. Dependencia de datos.
Bloques	Diferentes modelos. Caracterización en corto tiempo	Error variable. Dependencia de datos.
Instrucciones	Conocimiento mínimo del hardware. Uso reducido de datos reales.	Largo proceso de caracterización.

el error obtenido con el análisis estático es mayor que el error obtenido con herramientas dinámicas, se considera que es valido el esfuerzo, si este tipo de análisis logra reducir el tiempo de estimación y si se logra implementar como una herramienta para la selección de politicas de ahorro de energía, como sería el caso de un compilador en un sistema empotrado.

Capítulo 3
OBJETIVOS Y METODOLOGÍA

Este capítulo presenta los objetivos del proyecto y una descripción de la metodología adoptada para la implementación del estimador de potencia y energía.

3.1. Objetivos del Proyecto

El objetivo central de este proyecto es implementar una metodología que permita estimar el consumo promedio de potencia y energía de un microprocesador al ejecutar un programa determinado, por medio de un perfíl de potencia a nivel de instrucciones y herramientas de análisis estático de código.

Para alcanzar esta meta, se debe cumplir con los siguientes objetivos específicos.

- Generar un perfíl de consumo de potencia a nivel de instrucciones para el microprocesador seleccionado.
- Implementar las herramientas de análisis estático de programas.
- Implementar el sistema para estimar el tiempo de ejecución del programa bajo prueba.
- Generar el programa estimador de potencia y energía.
- Validar los resultados obtenidos por el estimador de potencia y energía, utilizando una metodología de prueba de hipótesis.

3.2. Metodología

La Figura 3–1 presenta un mapa congnitivo que describe la metodología del proyecto.

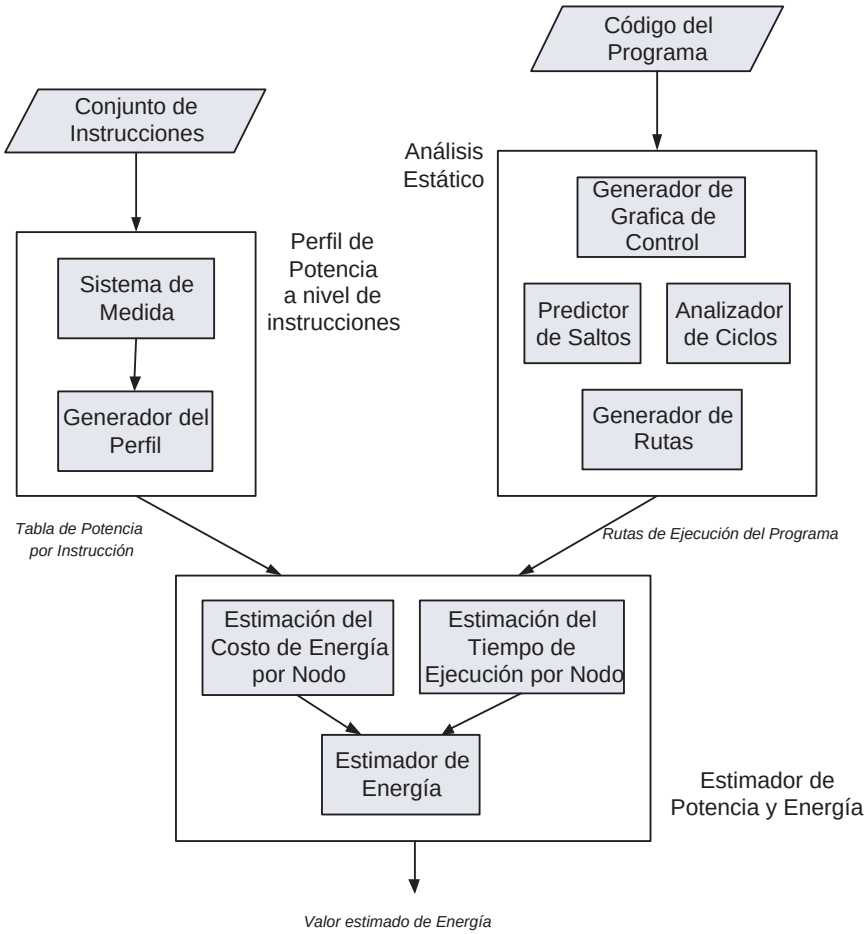

Figura 3–1: Mapa cognitivo de la metodología del proyecto

Se tienen dos entradas generales al sistema. La primera entrada es el conjunto de instrucciones del microprocesador bajo estudio. Esta entrada es necesaria dado que el modelo seleccionado en este proyecto utiliza las instrucciones como elemento base para estimar el consumo de potencia. La segunda entrada es el programa bajo prueba, el cual se divide en dos partes: La versión en alto nivel del programa y la versión en ensamblador del programa. La versión en alto nivel es necesaria para determinar los nombres asignados a variables de interés, como los limites de ciclo, contadores, etc. La versión en ensamblador es necesaria ya que todas las herramientas

de análisis estático están orientadas a este tipo de lenguaje. De hecho, es posible que no sea necesaria la versión en alto nivel, pero siempre será necesaria la versión en ensamblador. Una entrada que no se menciona en este mapa general son los valores de algunas variables del programa, las cuales pueden necesitar el analizar estático para estimar la cantidad de iteraciones de los ciclos del programa. Esta entrada será explicada con mas detalle en el Capítulo 5.

Como se puede observar en la Figura 3–1, los bloques perfíl de potencia y análisis estático presentan salidas intermedias dirigidas al tercer bloque. La salida del perfíl de potencia es una tabla que relaciona cada instrucción del microprocesador con un consumo de potencia. La salida del análizador estático son todas las posibles rutas de ejecución que tiene el programa bajo análisis. Estas rutas de ejecución contienen secuencias de instrucciones e información adicional como las probabilidades asignadas a cada ruta y la cantidad de iteraciones de los ciclos. Esta información es combinada en el tercer bloque para poder estimar el valor de consumo de potencia y energía del programa, el cual es la salida de todo el sistema. A continuación se explica con más detalle cada uno de los bloques que conforma el sistema presentado en la figura 3–1.

3.2.1. Perfil de Potencia a Nivel de Instrucciones

Este módulo tiene como función principal generar una relación entre el consumo de potencia promedio del microprocesador y cada instrucción que este puede procesar. La información de entrada para este módulo es el conjunto de instrucciones del microprocesador bajo estudio. Para implementar este perfíl, se construyó un sistema *hardware* que permitía realizar las medidas de potencia para cada instrucción del microprocesador. Este sistema es controlado por un sistema *software* que tiene como función principal seleccionar una por una las instrucciones a medir, implementar los programas de prueba, enviarserlos al sistema *hardware* para obtener la medida

y una vez se tiene suficiente cantidad de medidas, generar el valor promedio para dicha instrucción.

3.2.2. Analizador Estático de Código

Este módulo tiene como función principal realizar un análisis del programa a ser ejecutado por el microprocesador con el fin de proveer la información requerida por el estimador de potencia y energía para calcular el consumo promedio del microprocesador al ejecutar dicho programa. Este módulo se divide en tres partes principales, las cuales se explican a continuación.

El generador de la gráfica de control de flujo (CFG) tiene como fin convertir el programa escrito en lenguaje ensamblador en una gráfica compuesta por nodos y aristas, las cuales interconectan los diferentes nodos del programa. Para realizar esta conversión, se utilizó la técnica presentada por Aho y asociados [19]. La gráfica obtenida consiste de nodos, los cuales son bloques de instrucciones sin bifurcaciones, una o varias aristas de entrada a dicho nodo (excepto el nodo inicial) y una o dos aristas de salida, (solo los saltos tienen dos aristas de salida). La excepción es el nodo final de programa, que no tiene aristas de salida.

La segunda parte es el analizador de saltos y ciclos. El proceso se inicia identificando si hay ciclos en el programa. Si hay ciclos, se identifican cuáles nodos pertenecen a dichos ciclos. El interés especial está en determinar el nodo cabecera del ciclo, o sea el nodo que normalmente determina si el programa continúa o no en el ciclo. Una vez identificados los ciclos del programa, se procede a determinar si es posible estimar la cantidad de iteraciones del ciclo. Si no es posible determinar la cantidad de iteraciones debido a que se tiene una expresión como límite del ciclo, se procede a leer de un archivo los valores de iteración asignados por el usuario para cada ciclo. Luego de haber identificado los ciclos, se recorre el CFG para determinar dónde hay nodos que finalizan con una instrucción de salto y por lo tanto poseen

dos aristas de salida. Cuando se encuentran dichos nodos, se utilizan heurísticas de predicción de salto para asignar un valor de probabilidad a cada arista que sale del nodo. Esto se hace debido a que no se posee información de tiempo real sobre cual camino seguirá el programa.

La tercera parte es el generador de rutas del programa. Su objetivo es determinar todos los posibles caminos que puede seguir el microprocesador cuando ejecute el programa. Este análisis se realiza a partir de la gráfica de control de flujo. Gracias a los valores de probabilidad asignados a los saltos, se puede determinar cuál es la ruta del programa con mayor probabilidad de ejecución, y que, por lo tanto, será la que mayor influencia tendrá en la estimación de potencia y energía. Para implementar este componente se utilizan técnicas modificadas de análisis de estructuras de datos tipo árbol.

3.2.3. Estimador de Potencia y Energía

El último módulo utiliza la información generada por los dos módulos anteriores para estimar el consumo de potencia y energía del microprocesador. La primera parte toma las rutas de ejecución del programa y convierte cada nodo de dicha ruta al conjunto de instrucciones que representa. Esta secuencia de instrucciones es utilizada por otra parte del programa para computar los componentes de corriente para dicha ruta. Los componentes de corriente son el costo base y el costo por conmutación. Finalmente se computa el valor promedio de potencia multiplicando el valor de corriente promedio de cada ruta, por el voltaje de polarización del microprocesador. La potencia promedio total del programa se obtiene de sumar los productos de la potencia de cada ruta y el valor de probabilidad de dicha la ruta, valor que ha sido previamente asignado por el analizador estático de código. A este valor se le agregan los costos adicionales generados por fallas en la memoria Cache. Para

determinar cuánto es el aporte de esta componente se tiene un simulador de la Cache del microprocesador, a la cual se le aplican las diferentes rutas del programa.

Para calcular la energía se debe obtener primero el valor estimado del tiempo de ejecución del programa. El tiempo de ejecución se obtiene en dos pasos. El primer paso determina cuánto tiempo va a emplear el microprocesador en ejecutar cada ruta del programa. El segundo paso calcula el tiempo promedio de ejecución al sumar los productos tiempo de ejecución por el valor de probabilidad de cada ruta del programa. Finalmente del producto de potencia promedio y tiempo estimado de ejecución, se obtiene el valor estimado de consumo de energía del microprocesador al ejecutar el programa bajo análisis.

Capítulo 4
PERFIL DE POTENCIA DE LAS INSTRUCCIONES

Este capítulo describe la implementación del modelo de potencia utilizado en este proyecto. El microprocesador seleccionado para implementar el trabajo fue el PowerPC 603e de Motorola. La selección de dicho microprocesador se debe a las oportunidades que presenta su arquitectura para realizar estimación y optimización del consumo de energía a través del código que ejecuta. En adición, la complejidad de su arquitectura, junto con la grán cantidad de variantes de sistemas empotrados basados en este microprocesador, permite extender los conceptos presentados en este trabajo, a arquitecturas similares y proveer ideas para la generalización de una metodología de estimación de potencia y energía a nivel de instrucciones.

El capítulo está organizado de la siguiente forma: La sección 4.1 describe el modelo de potencia y energía empleado en el proyecto. La sección 4.2 describe el trabajo preliminar realizado con el conjunto de instrucciones del microprocesador bajo estudio. La sección 4.3 presenta medidas preliminares realizadas para determinar el influencia de ciertos factores en el consumo de corriente. Por último, La sección 4.4 describe la implementación física realizada para obtener las medidas de potencia.

4.1. Modelo para la Estimación de la Potencia y la Energía

El modelo adoptado en este proyecto es un consumo de potencia a nivel de instrucciones, considerando al microprocesador como una caja negra. Esto quiere

decir que el modelo es un perfil de potencia que relaciona un consumo de potencia promedio por ciclo, para cada instrucción que puede ejecutar el microprocesador bajo prueba.

El consumo de potencia promedio se determina indirectamente al medir la corriente promedio que consume el microprocesador al procesar las instrucciones del programa bajo prueba, dado que el voltaje se supone constante. La corriente consumida por el microprocesador se considera compuesta de tres componentes, llamadas corriente base I_B, corriente de conmutación I_O y corriente extra I_E. A continuación se describe cada uno de estos componentes y como se determinan estos valores para un conjunto de instrucciones.

4.1.1. Corriente Base

Este componente mide el consumo de corriente promedio de cada instrucción. Esto quiere decir que se mide el consumo de corriente por ejecutar únicamente esa instrucción, sin considerar la ocurrencia de otra instrucción o algún efecto externo. Debido a que los microprocesadores actuales contienen una estructura dividida en etapas, (*pipeline* en inglés), para medir este componente de corriente, el microprocesador bajo prueba debe procesar varias instancias de la instrucción bajo medida, con lo cuál se asegura que toda actividad realizada en el microprocesador en cada ciclo, se debe únicamente a la instrucción bajo prueba.

Una vez obtenida la corriente base para cada instrucción, se puede determinar el consumo promedio de corriente base para un conjunto de instrucciones, a partir de la siguiente expresión:

$$I_B = \frac{1}{C_T} \sum_i I_{Bi} \cdot C_i \tag{4.1}$$

donde:

I_B : Es la corriente base promedio de la secuencia de instrucciones.

I_{Bi} : Es la corriente base asignada a cada instrucción.

C_i : Es la cantidad de ciclos de cada instrucción.

C_T : Es la cantidad total de ciclos del conjunto de instrucciones.

La cantidad de ciclos de cada instrucción es un valor que está determinado en el manual de referencia del microprocesador bajo estudio [28].

4.1.2. Corriente de Conmutación

La corriente por conmutación es un componente de corriente que busca compensar el error que se obtiene por considerar sólo el costo base de corriente. Esto se debe a que la medida de la corriente base es realizada en un ambiente de mínimo cambio, pues la misma instrucción se encuentra en todas las etapas de microprocesador. Sin embargo, en un programa real es frecuente encontrar diferentes instrucciones en las etapas del microprocesador, lo cual aumenta la conmutación en dichas etapas, lo que conlleva a una variación en el consumo de corriente respecto al costo base.

Para determinar este componente de corriente, se compara la medida de corriente obtenida por ejecutar un par de instrucciones, con la medida estimada por considerar sólo el costo base de dichas instrucciones. La diferencia entre estas corrientes se toma como la corriente por conmutación del par bajo prueba.

$$I_{Ov} = I_{med-par} - I_{base-par} = I_{med-par} - \frac{I_{B1} \cdot C_1 + I_{B2} \cdot C_2}{C_1 + C_2} \tag{4.2}$$

donde:

I_{Ov} : Corriente por actividad de conmutación del par de instruciones.

$I_{med-par}$: Corriente medida al ejecutar el par de instrucciones.

I_1 : Costo base de la instrucción 1.

I_2 : Costo base de la instrucción 2.

C_1 : Ciclos instrucción 1.

C_2 : Ciclos instrucción 2.

Dado que la conmutación ocurre dos veces durante la ejecución del par de instrucciones, (de la instrucción uno a la dos y viceversa), se agrega un factor de corrección para compensar la diferencia de ciclos; así, la ecuación 4.2, se ajusta a:

$$I_{Ov} = [I_{med-par} - I_{base-par}] \cdot \frac{C_1 + C_2}{2} \tag{4.3}$$

Donde el 2 cuenta por las dos conmutaciones ocurridas durante los ciclos empleados por las dos instrucciones.

Se ha considerado el uso de pares de instrucciones, dado que el microprocesador bajo prueba puede enviar hasta dos instrucciones a las diferentes unidades de ejecución. Esto no quiere decir que puedan ocurrir casos donde se estén procesando más instrucciones en un momento determinado, pero se estima que el caso más común es la interacción de dos instrucciones en el microprocesador.

Para determinar el costo de corriente por conmutación de un conjunto de instrucciones, se consideran los pares consecutivos de instrucciones. Se ha considerado de esta forma, debido a la estructura por etapas de los microprocesadores actuales y porque el microprocesador va procesando las instrucciones de forma ordenada, según el programa bajo prueba. Así, el valor promedio de la corriente de conmutación para el conjunto de instrucciones, se determina con la siguiente expresion:

$$I_O = \frac{1}{C_T} \sum_i I_{Ov(i,i+1)} \tag{4.4}$$

Donde:

I_O : Es la corriente de conmutación promedio de la secuencia de instrucciones.

$I_{Ov(i,i+1)}$: Es la corriente de conmutación del par consecutivo de instrucciones.

C_T : Es la cantidad total de ciclos del conjunto de instrucciones.

4.1.3. Corriente Extra

En la ejecucion normal de un programa pueden ocurrir ciertas situaciones que pueden generar un aumento en el tiempo de ejecución del programa, y por lo tanto, el consumo de energía del microprocesador. Existen dos situaciones que son de gran importancia debido al costo que ellas generan. El costo asociado a un desacierto en la memoria cache y el costo asociado a una predicción erronea en la unidad de predicción de saltos.

El costo asociado a un desacierto en la memoria cache consiste en un tiempo de espera y un consumo de corriente generados cuando el dato no se encuentra en la cache. Esto se debe a que es necesario efectuar un acceso al sistema de memoria.

Para medir el costo asociado de corriente y tiempo, se utilizaron las características que presenta el microprocesador PowerPC 603e para la configuración de la memoria. Utilizando la opción de cache inhibida, todas las solicitudes de datos son dirigidas directamente a la memoria. Gracias a esta opción, se puede medir el retardo adicional y estimar el consumo de corriente al no tener ayuda de la memoria cache. Es importante recalcar que el tiempo de penalidad depende de la configuración del sistema de memoria, por lo tanto, este valor debe ser determinado para cada nuevo sistema bajo prueba.

El costo asociado a una predicción erronea del salto se describe de la siguiente forma. Cuando el microprocesador no puede resolver un salto condicional, este procede a predecir cuál va a ser la dirección del salto. Luego de realizar la predicción, procede a traer las instrucciones de la ruta predicha. Cuando el salto se puede resolver, se comprueba si la predicción fue cierta o falsa. Si la predicción fue cierta, se procede con la ejecución normal del programa, pero si fue falsa, se genera una acción de expulsión de las instrucciones traidas erroneamente y se procede a traer

las instrucción de la ruta correcta. Esto produce un retardo en el tiempo de ejecución del programa y un consumo adicional de corriente, el cual aumenta la energía consumida por el microprocesador.

La medida de esta penalidad se hace gracias a las características propias del microprocesador PowerPC 603e. La unidad de predicción de saltos puede ser modificada para que invierta su predicción en forma individual para cada instrucción de salto que encuentre. Así, se puede crear un programa que permita estimar este costo. A continuación se presenta un pseudocódigo donde se ilustra la idea para medir el costo de una predicción erronea.

```
        x = 4;
ciclo:  x = x + 0;
        compare x con 2;
        si son iguales, salte a etq1;
        instruccion 1;
etq1:   instruccion 2;
        salte a ciclo;
```

En este caso, la variable x tiene asignado un valor igual a 4. Este valor no cambia durante toda la ejecución del programa. Asi cuando se ejecuta la instrucción de salto, esta tendrá siempre la misma dirección (no salta). Sin embargo, debido a que la instrucción de comparación está contigua a la instrucción de salto, el microprocesador se ve obligado a predecir la dirección del salto. En modo normal el salto se predice como no tomado, esto quiere decir que no salta, y la próxima instrucción será *instrucción 1*. Así, las medidas adquiridas serán las de un éxito en la predicción. Si se modifica el valor de la predicción de la instrucción de salto y se ejecuta de nuevo el programa, el microprocesador predice que sí salta, de modo que en este caso, fallará su predicción, y las medidas obtenidas en este caso corresponderán al costo por fallar la predicción.

En general, para estimar el valor de la corriente extra y el tiempo adicional de un efecto externo, se realiza primero una medida de un conjunto de instrucciones

dadas, donde no se presenta el efecto bajo estudio. En este caso, la medida de corriente promedio obtenida esta dada por:

$$I_{NE} = \frac{I_1 \cdot C_1 + I_2 \cdot C_2 + \ldots + I_n \cdot C_n}{C_1 + C_2 + \ldots + C_n} \tag{4.5}$$

donde:

I_{NE} : Es la lectura de corriente obtenida por ejecutar la secuencia de instrucciones.

$I_1 \ldots I_n$: Es la corriente promedio por ejecutar cada instrucción.

$C_1 \ldots C_n$: Es la cantidad de ciclos de cada instrucción.

A continuación, se realiza otra medida en la misma secuencia de instrucciones, pero en este caso se agrega el efecto bajo medida. En este caso, se estima que la corriente medida y el tiempo varían de la siguiente forma:

$$I_{Ex} = \frac{I_1 \cdot C_1 + I_2 \cdot C_2 + \ldots + I_n \cdot C_n + I_E \cdot C_E}{C_1 + C_2 + \ldots + C_n + C_E} \tag{4.6}$$

donde:

I_{Ex} : Es la lectura de corriente que incluye el efecto bajo medida.

I_E : Es la corriente promedio por ejecutar el efecto extra.

C_E : Es la cantidad de ciclos por ejecutar el efecto extra.

A partir de estas dos medidas, se puede determinar el costo de corriente extra y de ciclos adicionales por ejecutar el efecto bajo medida. La cantidad de ciclos adicionales se determina a partir de la diferencia de los tiempos medidos en las dos pruebas:

$$C_E = C_{Ex} - C_{NE} \tag{4.7}$$

donde:

C_E : Son los ciclos del efecto extra bajo medida.

C_{NE} : Son los ciclos obtenidos de la medida sin el efecto extra.

C_{Ex} : Son los ciclos obtenidos incluyendo el efecto extra.

La corriente promedio por el efecto extra se puede estimar a partir de la expresión 4.8, donde todos los componentes ya son conocidos:

$$I_E = \frac{I_{Ex} \cdot C_{Ex} - I_{NE} \cdot C_{NE}}{C_E} \tag{4.8}$$

4.1.4. Estimación de la Potencia y Energía

Una vez determinados los diferentes componentes de corriente promedio para una secuencia de instrucciones, se puede determinar el consumo de potencia, como lo indican las siguientes ecuaciones:

$$P_{Base} = V_{cc} \cdot I_B \tag{4.9}$$

$$P_{Conmutacion} = V_{cc} \cdot I_O \tag{4.10}$$

$$P_{Extra} = V_{cc} \cdot I_E \tag{4.11}$$

Donde V_{cc} es el voltaje de polarización. Es importante resaltar que cada proceso extra tiene un consumo de corriente diferente, por lo tanto, es necesario deteminar el valor específico de potencia para cada efecto extra considerado.

La energía consumida por el microprocesador se puede determinar a partir de cada uno de los componentes de potencia expresados anteriormente. Para esto, es necesario determinar el tiempo de ejecución de la secuencia de instrucciones y el tiempo consumido por cada uno de los efectos externos que se presentan en la ejecución del programa. Así, se tiene el componente de energía base, expresado por:

$$E_{Base} = P_{Base} \cdot N \tag{4.12}$$

Donde N es el tiempo de ejecución de las instrucciones del programa. Este tiempo se puede determinar a partir de la cantidad de ciclos que requieren las instrucciones para su ejecución, multiplicado por el tiempo de cada ciclo, el cual corresponde al periodo de la señal de reloj del microprocesador.

De forma similar se puede determinar el consumo de energía por la componente de conmutación:

$$E_{Conmutacion} = P_{Conmutacion} \cdot N \tag{4.13}$$

Donde N es el tiempo de ejecución de las instrucciones del programa.

El componente de energía debido a un efecto externo se determina a partir del consumo de potencia del efecto y el tiempo requerido para procesar dicho efecto.

$$E_{Extra} = P_{Extra} \cdot T \tag{4.14}$$

Donde T es el tiempo de ejecución del efecto externo. Es importante recordar que cada efecto tiene un tiempo diferente. Para determinar la contribución de energía de esta componente en la ejecución del programa, es necesario determinar cuántas veces se presenta cada uno de los efectos durante la ejecución del programa, así, por cada ocasión que se presente el efecto, se debe agregar el costo de energía determinado por la expresión 4.14 al consumo de energía total del programa.

Finalmente, la energía consumida por el microprocesador al ejecutar un programa se obtiene a partir de la suma de las tres componentes de energía expresadas anteriormente.

4.2. Clasificación de las Instrucciones para la Medida

Como trabajo preliminar a la medida de potencia de cada instrucción, se hizo un examen del conjunto de instrucciones del microprocesador PowerPC 603e. De este examen se decidió que algunas instrucciones no serían incluidas en el perfíl de potencia. La razón principal es que estas instrucciones son utilizadas por el sistema

operativo para la configuración del microprocesador y sus componentes internos, o son utilizadas en un entorno de múltiples microprocesadores. Esto quiere decir que son instrucciones que normalmente no están disponibles para los programas de usuario. El apéndice C muestra las instucciones que no han sido consideradas en el perfíl de potencia de las instrucciones. Para no asignar un valor cero a la potencia promedio de estas instrucciones, se determinó qué instrucciones incluidas en el perfil de potencia tenían una funcionalidad similar o pertenecían a la misma unidad de ejecución. Una vez determinadas dichas instrucciones, se tomó un valor promedio y se asignó a las instrucciones no incluidas en el perfil de potencia. El razonamiento utilizado para esto es que las instrucciones ejecutadas por las mismas unidades funcionales tienden a consumir una potencia similar. El apéndice C también muestra las instrucciones asignadas como equivalentes. Notesé que en muchos casos, no se puede determinar una equivalencia, razón por la cual se asignó un valor promedio de la unidad de ejecución que las procesa.

En cuanto a los modos de direccionamiento, el PowerPC 603e tiene 3 modos para acceder datos de memoria: Registro indirecto con índice inmediato, registro indirecto con registro índice y registro indirecto. La estructura de las instrucciones del PowerPC 603e esta organizada de forma que solo las instrucciones de lectura y escritura acceden la memoria. El resto de instrucciones trabaja con los registros del microprocesador. Con base en esto, los modos de direccionamiento sólo se tuvieron en cuenta en las instrucciones de leer y guardar en memoria.

Por último, las instrucciones de salto tienen el modo de direccionamiento relativo, absoluto y con registro. Al igual que con las instrucciones de accceso a memoria, estos modos de direccionamiento están restringidos a este tipo de instrucción, por lo cúal, se pueden considerar estos modos de direccionamientos como propios de dicha instrucción. La tabla 4–1 muestra ejemplos de los modos de direccionamiento utilizados por las instrucciones.

Tabla 4–1: Modos de direccionamiento.

Instrucción	Función
ba address	salto absoluto
bc op,flag,address	Salto condicional relativo
lbz Rd,d(Ra)	Lectura memoria registro e indice inmediato
stb Rd,Ra,Rb	Escritura memoria con registro indice

4.3. Medidas Preliminares

El modelo adoptado de energía considera a las instrucciones como su componente base para la estimación del consumo de potencia y energía. Por lo tanto, es necesario considerar qué factores pueden hacer que el comportamiento de una instrucción cambie, de modo que pueda afectar el consumo de corriente de la instruccion bajo medida.

En general, se tienen factores de *hardware* y *software*. Los factores de *hardware* no han sido considerados en este trabajo, dado que el enfoque del modelo de energía es la estimación a través del análisis del código. Por lo tanto los factores de *hardware* como frecuencia de reloj y voltaje de polarización se mantuvieron constantes durante todas las pruebas. Tambien se procuró mantener un entorno de trabajo con características similares durante todas las pruebas realizadas.

En cuanto a los factores de *software*, se han considerado los argumentos que puede tener una instrucción. En general, las instrucciones pueden tener registros y datos inmediatos como argumentos. Así, antes de realizar las medidas de corriente, se realizaron un grupo de medidas preliminares para determinar su efecto en el consumo de corriente de la instrucción.

Para determinar la influencia del uso de registros, se consideró la instrucción de suma, la cual posee como argumento dos registros como fuente y un registro como destino de la operación. Para la prueba, se realizaron varias medidas con registros seleccionados de forma aleatoria. El resultado observado es que el uso de

diferentes registros no afecta en gran medida el consumo de corriente promedio. Asi se decidió utilizar registros aleatorios en las diferentes medidas realizadas.

Un caso similar sucedió con los valores inmediatos, donde no se presentó una variación notable en el consumo de corriente. Asi se decidió utilizar datos inmediatos aleatorios en las diferentes medidas realizadas.

Con base en las medidas obtenidas en estas pruebas preliminares, se observó que generalmente con una muestra de 100 medidas se obtiene un error estándar menor a 5 miliamperios con un intervalo de confianza de 95 %. Un recálculo de la cantidad de muestras indica que con 71 muestras se tiene un aumento leve en el error estandar, sin embargo, se decidió dejar 100 muestras para todas las pruebas, para compensar algún caso donde se necesiten más muestras que las obtenidas por el cálculo realizado anteriormente.

Para decidir si el uso de diferentes registros o datos inmediatos no afectaba en forma considerable el consumo de corriente, se utilizó el coeficiente de variación (CV) como referencia.

$$CV = \frac{DesviacionEstandar}{Media} \cdot 100\,\% \tag{4.15}$$

Un coeficiente de variación menor al 5 % sugiere que no hay diferencia considerable en las medidas obtenidas. En el caso de la prueba de registros y datos inmediatos, el coeficiente de variación fue de 3 % y 4 % respectivamente, por lo cual, durante las pruebas, sólo se consideró a la instrucción como unico factor de importancia en el consumo de corriente del microprocesador.

4.4. Implementación del Sistema de Medida

Para obtener el valor promedio de corriente para cada instrucción del microprocesador, se planteó el método de medida indicado en la Figura 4–1.

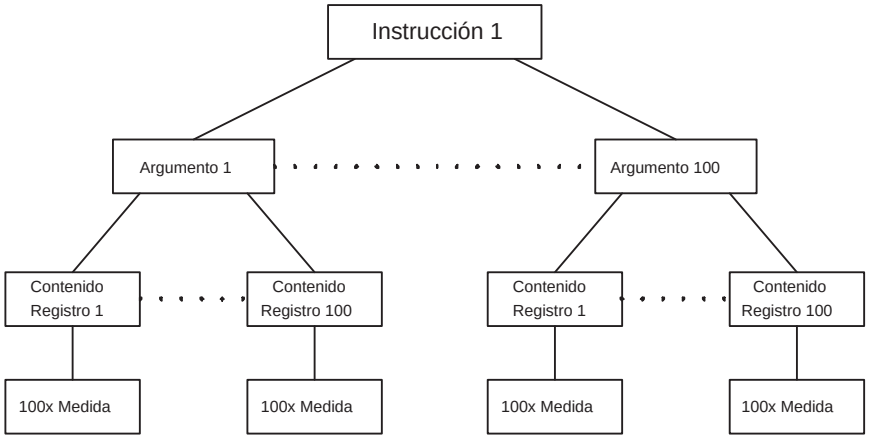

Figura 4–1: Estructura empleada para la medida de corriente de una instrucción

Para obtener el valor promedio del consumo de corriente, se han elegido 100 valores aleatorios para el argumento de cada instrucción. Como casi todas las instrucciones incluyen un registro, se ha elegido 100 valores aleatorios para los contenidos de los registros, garantizando que las medidas obtenidas puedan brindar un valor estadístico significativo [29].

Así, el proceso de medida se describe de la siguiente forma: primero se elige un argumento aleatorio para la instrucción bajo prueba. Luego se eligen contenidos aleatorios para los registros del argumento y se procede a medir. Finalizada la medida, se procede a generar un nuevo contenido aleatorio para los registros del argumento y se procede a medir nuevamente. Este proceso se repite hasta obtener 100 medidas para el argumento actual. El promedio de estas medidas se toma como el consumo de corriente de la instrucción con ese argumento. Luego se genera otro argumento aleatorio para la instrucción y se repite el proceso. Una vez finalizada la obtención los valores de corriente para los 100 argumentos, se puede obtener un valor promedio de corriente, el cual será la medida de corriente promedio para la instrucción bajo prueba.

La Figura 4–2 presenta el sistema general para la medida del consumo de corriente del microprocesador, cuando está ejecutando la instrucción bajo análisis.

El sistema se divide en un componente de *hardware* y un componente de *software*. Las secciones siguientes describen dichos componentes.

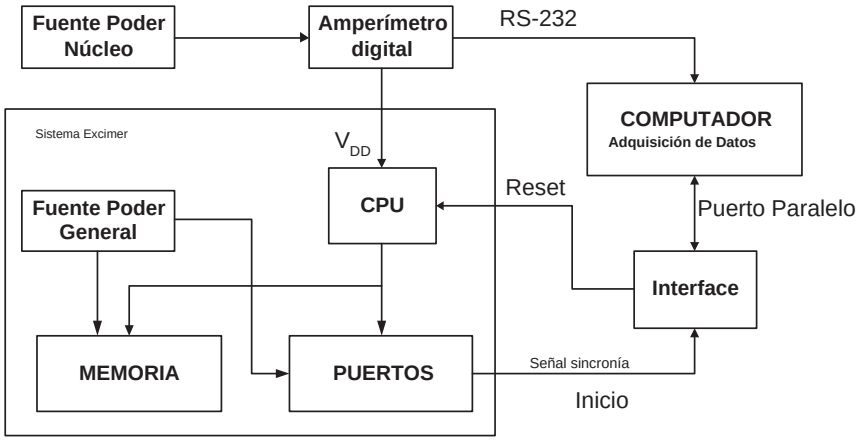

Figura 4–2: Diagrama en bloques del sistema de medida

4.4.1. Componente *Hardware*

En primer lugar, el microprocesador PowerPC 603e utilizado en este proyecto, está montado en una tarjeta de evaluación llamada Excimer [30] [31]. Esta tarjeta tiene la ventaja que posee una línea de polarización exclusiva para el núcleo del microprocesador y otra línea de polarización para el resto del circuito. Así, sólo es necesario modificar una fuente de poder. La medida de la corriente promedio se obtuvo por medio de un amperímetro conectado en serie con la línea de polarización del microprocesador. Para la automatización de la medida, el amperímetro cuenta con un sistema de comunicación en serie, que permite la lectura del valor de corriente desde un sistema remoto, que en este caso es un computador.

Para obtener una medida estable en el amperímetro, se introdujeron 200 instancias de la instrucción bajo prueba y se encerraron en un ciclo, a cuya variable contadora le fue asignado el máximo valor que puede almacenar, $2^{32} - 1$, garantizando una señal periódica que se repite suficiente cantidad de veces para obtener la cantidad de medidas deseadas. Para elegir la cantidad de instancias a incluír en el ciclo, se tuvo en cuenta que se minimizara el efecto de las instrucciones de salto del

ciclo y que no se excediera el tamaño de la cache para evitar desaciertos. También se tuvo en cuenta el número de instancias propuesto por Tiwari y Russell, que oscila entre 100 y 120 instancias de la instrucción bajo prueba [13] [12].

Debido a que se requería la ejecución de un programa de carga y configuración previo a la ejecución del programa de prueba, se utilizó una señal de sincronización (llamada inicio en la Figura 4–2) para la lectura de corriente. Esta señal proviene de un puerto de la tarjeta de evaluación Excimer y se dirige a una línea de entrada del puerto paralelo del computador. Así, la lectura de corriente está condicionada a la activación de la señal de sincronización.

4.4.2. Componente *Software*

Para coordinar las medidas se utilizó un programa de control, el cual se encargaba de realizar todos los pasos necesarios para la selección, procesamiento y medida de corriente de una instrucción. A continuación se presenta el algoritmo de dicho programa.

Algoritmo 1. *Controlador de medida de corriente*

Entrada: Conjunto de instrucciones.
Salida: Costo promedio de corriente para cada instrucción.

1. **Para** cada instrucción **hacer:**
   - 1.1. $Arg \leftarrow$ Número total de argumentos;
   - 1.2. **Mientras** $Arg > 0$ **hacer:**
     - 1.2.1. Llamar al generador de programas;
     - 1.2.2. Descargar programa y ejecutarlo;
     - 1.2.3. Llamar al programa recolector de datos;
     - 1.2.4. $Arg = Arg - 1$;
   - 1.3. Calcular promedio;
   - 1.4. Calcular varianza;
   - 1.5. Salvar resultado;
2. **Fin**;

El generador de programas es un software que se encarga de componer el programa que se descarga al microprocesador. Este programa se construyó utilizando un Script hecho en PERL. A continuación se presenta su algoritmo.

Algoritmo 2. *Generador de programas*

Entrada: Instrucción bajo prueba.
Salida: Programa en ensamblador para descargar en la tarjeta de evaluación.

1. Leer instrucciones a probar;
2. Leer el programa-base;
3. **Para** cada argumento en la instrucción **hacer:**
   - 3.1. **Si** argumento es registro **hacer:**
     - 3.1.1. $R \leftarrow$ rand(32);
     - 3.1.2. $Argumento \leftarrow$ R;
   - 3.2. **Si** argumento es dato inmediato **hacer:**
     - 3.2.1. $R \leftarrow$ rand(2^{32});
     - 3.2.2. $Argumento \leftarrow$ R;
   - 3.3. **Si** argumento es campo comparación **hacer:**
     - 3.3.1. $R \leftarrow$ rand(7);
     - 3.3.2. $Argumento \leftarrow$ R;
4. Guardar instruccion modificada en programa-base;
5. Ensamblar programa base;
6. **Fin**;

Este programa toma una instrucción, identifica cuáles son sus argumentos y los reemplaza con valores aleatorios, según los límites de cada tipo de argumento. En el caso de registros, es necesario almacenar valores aleatorios en ellos para garantizar la veracidad de las medidas realizadas. Esta función la realiza el programa que se descarga a la tarjeta Excimer. La cantidad de registros disponibles para el usuario son treinta y dos registros de proposito general y treinta y dos de punto flotante, como lo muestra la Figura 4–3, sin embargo, los primeros siete registros tienen un proposito específico, como almacenar el valor del apuntador de pila y parámetros entre subrutinas, razón por la cual no se utilizaron en las pruebas. En total, 25 registros se utilizaron para las diferentes pruebas de instrucciones que requerían registros enteros y los treinta y dos registros de punto flotante fueron utilizados en las instrucciones que lo requerían.

En el caso de los datos inmediatos, estos son números de 32 bits, asi que existe un gran rango para la selección. Las direcciones de memoria tienen la limitación que impone la implementación física de la tarjeta de evaluación utilizada, como lo indica

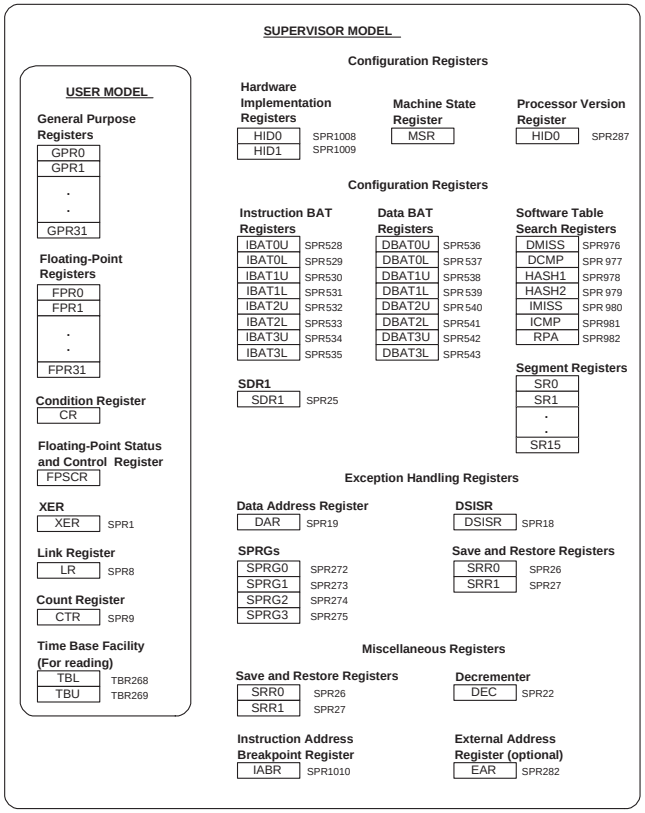

Figura 4–3: Modelo de programación del PowerPC 603e.

el mapa general de memoria de la Figura 4–4. El sistema está diseñado para alojar

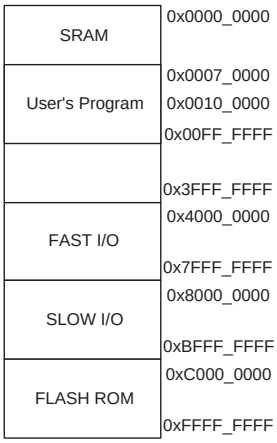

Figura 4–4: Mapa de memoria del sistema Excimer extraído del manual de usuario.

los programas de usuario en el rango 0x70000 hasta 0xFFFFFF. Sin embargo, la versión actual sólo implementa 1M de memoria, por lo tanto, la memoria de usuario se limita hasta la dirección 0x100000 [30] [32].

Esto limita la cantidad de números que se puede utilizar para la prueba de instrucciones. Adicionalmente, no todas las direcciones existentes se pueden utilizar para guardar datos, ya que el programa de control y el propio programa de prueba pueden verse afectados por una escritura aleatoria del programa.

El programa base es una plantilla que se utiliza para poder realizar la medida de corriente de la instrucción bajo prueba. A continuación se presenta su pseudocódigo.

```
Activar Memoria Cache;
Retardo para sincronizar con computador;
Cargar Semilla para Numeros Aleatorios;
Cargar variable Contador con repeticiones_maximas;
Lazo:  Decrementar Contador;
       Generar Numeros Aleatorios;
       Cargar variable Repeticion con valor maximo;
       Activar Seal de Sincronia;
       Inicio de Ciclo:
              200x Instruccion bajo prueba;
              Decremente Repeticion;
       Si repeticion no es cero, volver a ciclo;
       Desactivar Seal de Sincronia;
       Retardo de sincronizacion;
Si contador no es cero, volver a Lazo;
fin;
```

La primera parte del programa realiza la iniciación de la memoria cache y asigna valores a las variables del programa. La variable *contador* contiene la cantidad de valores aleatorios que se asignarán a los registros de la instrucción. Luego inicia un ciclo donde se generan valores aleatorios para los registros de la instrucción, se activa la señal de sincronía y se ejecutan en un ciclo interno las instancias de la instrucción bajo prueba. La variable *repetición* contiene cuántas veces se repite este ciclo interno. Una vez ejecutadas las instancias de la instrucción, se procede a generar un nuevo grupo de valores aleatorios para los registros de la instrucción, y se repite el ciclo.

Los algoritmos mostrados fueron empleados en la medida del costo base de corriente para las instrucciones del microprocesador. Para obtener la medida de

corriente de conmutación, se modificó el generador de programas, de modo que tomara 2 instrucciones consecutivas de un listado de instrucciones. Esta lista fué construida con un nuevo programa escrito en PERL, cuya función principal fue tomar las instrucciones a medir y generar las diferentes combinaciones de pares de instrucciones posibles [33]. Adicionalmente se implementó otro programa cuya función fue calcular el valor de la corriente de conmutación para cada uno de los pares medidos, utilizando la ecuación 4.3.

En el caso de la medida de corriente extra para los efectos externos, se utilizaron secuencias de instrucciones predefinidas, donde se pudiese generar el efecto extra bajo medida. Por tal razón, en esta prueba no fue utilizado el generador de programas y la plantilla fue modificada para tener la secuencia de instrucciones usada para cada efecto externo. En el caso del costo de un desacierto de la cache, se utilizó una secuencia de instrucciones de lectura de memoria, donde la primera medida utilizaba la memoria cache en modo normal y la segunda medida utilizaba la memoria cache en modo inhibido. En el caso de la predicción erronea del predictor de saltos, se utilizó una instrucción de salto seguida de una secuencia de instrucciones, donde el resultado del salto siempre es el mismo, con lo cual, la misma secuencia de instrucciones se repite. A partir de esto se realizó una medida con una predicción acertada y luego una medida donde se invirtió el resultado de la predicción para esa instrucción de salto.

Resumen

Este capítulo presentó el modelo de potencia adoptado para el microprocesador PowerPC 603e, el cuál esta basado en asignar un valor promedio de corriente a cada instrucción que puede ejecutar el microprocesador. Esta corriente está divida en tres componentes, la corriente base, la corriente de conmutación y la corriente extra. Con el valor de corriente asignado a cada instrucción, se puede determinar la potencia consumida por el microprocesador, dado que el voltaje se supone constante.

Capítulo 5
ANALISIS ESTATICO DE PROGRAMAS

El análsis estático es una técnica que permite inferir propiedades y comportamientos de un programa sin tener que realizar una ejecución completa, en tiempo real o por simulación, del programa bajo prueba. Así, es posible aplicar análisis estático para caracterizar un programa y obtener información que se pueda combinar con el modelo de potencia utilizado en este proyecto y generar un estimado del consumo de potencia y energía del microprocesador al ejecutar un programa dado.

En el caso del modelo de potencia utilizado en este proyecto, la información requerida es la secuencia de instrucciones que va a ejecutar el microprocesador. Debido a que no se conoce la secuencia real de instrucciones, (a menos que se ejecute el programa), se utilizan técnicas de análisis estático para determinar todas las posibles secuencias de instrucciones del programa bajo prueba y se asigna un valor de probabilidad que indique cuál es posibilidad que tiene cada secuencia de instrucciones de ser ejecutada. Una vez generada esta información se procede a utilizar el modelo de potencia para generar un valor estimado de consumo de potencia y energía del programa bajo prueba. Para generar todas las secuencias de instrucciones, se ha planteado el esquema de análisis estático mostrado en la Figura 5–1.

El generador de gráfica de control de flujo (CFG) convierte las instrucciones del programa ensamblador en una representación gráfica de nodos y aristas. Esta gráfica es enviada a los demás componentes para su procesamiento. El analizador de ciclos determina si existen ciclos en el programa y cuáles nodos del CFG pertenecen a cada ciclo encontrado. Para determinar la cantidad de iteraciones de cada ciclo, es

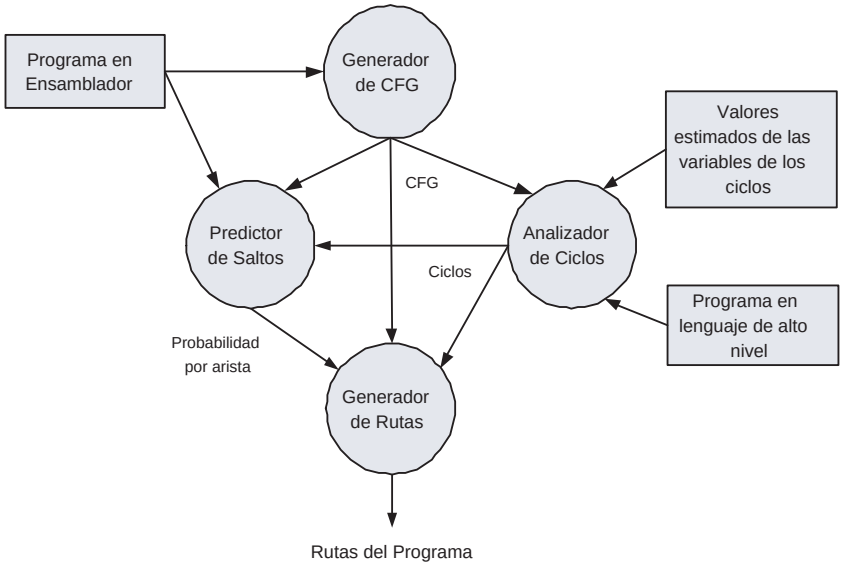

Figura 5–1: Esquema de análisis estático del proyecto

posible que se requieran valores estimados de las variables del ciclo y el código en alto nivel, si el programa está originalmente escrito en dicho lenguaje. La información de los ciclos es enviada al predictor de saltos y al generador de rutas. El programa predictor de saltos realiza un anásis del CFG para asignar valores de probabilidad a cada una de las aristas de la gráfica, para lo cual se ayuda de la información provista por el analizador de ciclos, heurísticas de decisión de salto y del código en lenguaje ensamblador. Los valores de probabilidad asignados son enviados al generador de rutas para que pueda estimar la probabilidad para cada ruta del programa. El programa generador de ruta utiliza la información generada por el analizador de ciclos, el predictor de saltos y el CFG para determinar todas las posibles rutas que puede seguir el programa bajo prueba. En las secciones siguientes, se explica en detalle el funcionamiento de cada uno de estos componentes y como fue implementado en el proyecto.

5.1. Gráfica de Control de Flujo (CFG)

Una gráfica de control de flujo es un grafo dirigido donde cada nodo que lo compone representa un conjunto secuencial de instrucciones, conocidas como bloque

básico, y cuyas aristas muestran hacia qué nodos se puede dirigir el control del programa en un momento dado de su ejecución.

La ventaja de utilizar un CFG es que se obtiene una representación general del programa, independiente del formato de las instrucciones y que posee una gran cantidad de herramientas con las que se pueden realizar análisis y estimaciones del programa bajo prueba.

Para construir el CFG, se deben observar las instrucciones de salto, especialmente los saltos condicionales, pues son estas las que modifican el flujo secuencial de un programa.

Antes de presentar el algoritmo, se definen los siguientes elementos constitutivos de un CFG:

- **Bloque**. Representa un conjunto de instrucciones que no cambian el flujo secuencial del programa, con excepción de la última instrucción, la cual puede ser una instrucción de salto.
- **Arista**. Representa la transición entre dos bloques. Las aristas tiene un solo sentido.

Basado en las definiciones anteriores, se tienen las siguientes reglas que delimitan un bloque básico de instrucciones:

- La primera instrucción del programa bajo prueba es el inicio de un bloque.
- La última instrucción del programa bajo prueba es el final de un bloque.
- Un bloque termina con la primera instrucción de salto que se encuentre.
- Un bloque inicia después de una instrucción de salto.
- La instrucción que es destino de un salto, es el inicio de un bloque y por lo tanto, la instrucción anterior a esta, es el final de un bloque.

Las reglas para definir una arista son:

- Existe una arista entre dos bloques si el segundo bloque es el destino de la instrucción de salto del primer bloque. Esta arista se designa como *Tomada*.

- Existe una arista entre dos bloques contiguos, si el segundo es el bloque inmediato al primero, el cual termina en una instrucción de salto condicionado. Esta arista se designa como *no tomada*.
- Existe una arista entre dos bloques si estos están contiguos y el primero no termina en una instrucción de salto.
- Existe una única arista entre un bloque y otro si el primer bloque termina con una instrucción de salto incondicional.

Las reglas anteriores se implementan en un algoritmo que puede generar el CFG deseado. A continuación se presenta el algoritmo implementado en este trabajo para generar el CFG, el cual ha sido adaptado de Aho y otros [19].

Algoritmo 3. *Generador de Grafica de Control de Flujo*

Entrada: Programa en lenguaje ensamblador.
Salida: Grafica en formato nodos y aristas.

1. Formar Bloques Básicos:
   1.1. Identificar instrucción inicial del bloque;
   1.2. Identificar instrucción final del bloque;
2. **Para** cada bloque **Hacer**
   2.1. **Si** la última instrucción del bloque es un salto **Entonces**
      2.1.1. **Si** es un salto incondicional **Entonces:**
         2.1.1.1. Identificar bloque destino;
         2.1.1.2. Generar arista;
      2.1.2. **De lo contrario, Si** es un salto condicional **Entonces**
         2.1.2.1. Identificar bloque destino;
         2.1.2.2. Generar arista *Tomada*;
         2.1.2.3. Identificar bloque siguiente al actual;
         2.1.2.4. Generar arista *NO Tomada*;
   2.2. **Si** la última instrucción del bloque NO es un salto **Entonces**
      2.2.1. Identificar bloque siguiente al actual;
      2.2.2. Generar arista;
   2.3. **Si** la última instrucción del bloque es FIN de programa **Entonces**
      2.3.1. Marcar bloque como FINAL;
3. Salvar información;
4. **Fin.**

Ejemplo 1. *La Figura 5–2 presenta un ejemplo de código representado en bloque y aristas. Las instrucciones de salto están identificadas con b o bc, para salto incondicional y condicional respectivamente.*

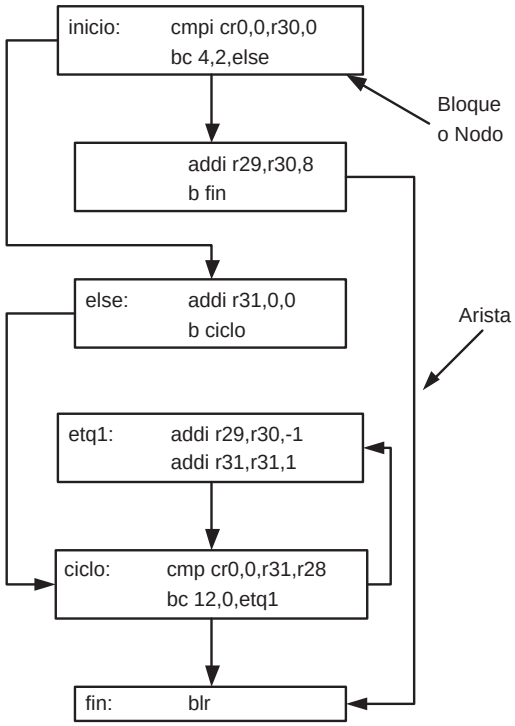

Figura 5–2: Ejemplo de CFG generado a partir de código ensamblador

5.2. Predicción Estática de Saltos

La predicción estática de saltos consiste en asignar un valor de probabilidad a cada arista de una instrucción de salto condicional, que expresa cúal será el posible comportamiento de la instrucción de salto cuando esta sea ejecutada por el microprocesador.

La metodología empleada para asignar los valores de probabilidad consiste en el uso de heurísticas que evaluan un tipo específico de condición de salto. En caso de encontrar una instrucción de salto que contiene la misma condición especificada por una heurística, se asigna el valor de probabilidad estático de dicha heurística a las aristas del bloque que contiene la instrucción de salto.

Las heurísticas fueron propuestas por Ball y asociados en base a los comportamientos en tiempo real de las instrucciones de salto de una gran cantidad de programas bajo diferentes datos de entrada [20]. A continuación se presenta el conjunto de heurísticas que se han implementado en este trabajo.

- Heurística de salto de ciclo (LBH). Predice que un salto que debe elegir entre salir de un ciclo o devolver al inicio de dicho ciclo, tiende a devolverse al inicio del ciclo.
- Heurística de código de operación (OH). Una comparación de una variable con menor, menor o igual a cero (0) o igual a una constante, tiende a fallar.
- Heurística de salida de un ciclo (LEH). Si un salto dentro del ciclo debe elegir entre mantenerse dentro del ciclo o salir, tiende a permanecer en el ciclo.
- Heurística de inicio de ciclo (LHH). Si un salto debe elegir entre entrar a un ciclo o no, normalmente elige entrar al ciclo.
- Heurística de llamado de subrutina (CH). Si una arista del bloque que contiene la instrucción de salto conduce a un llamado a subrutina, esta ruta probablemente no será tomada.
- Heurística de retorno de subrutina (RH). Si una arista del bloque que contiene la instrucción de salto conduce a un retorno de subrutina, esta ruta probablemente no será tomada.
- Heurística de salvar en memoria (SH). Si una arista del bloque que contiene la instrucción de salto conduce a una instrucción de almacenar información en memoria, esta ruta probablemente no será tomada.

La Tabla 5–1 presenta los valores de probabilidad de cada heurística. Estos valores fueron generados a partir de la frecuencia de predicciones correctas de cada heurística, medidas en una gran cantidad de programas de prueba [20].

Para implementar cada una de estas heurísticas, se desarrolló un programa en PERL. Dicho programa analiza el código del programa bajo prueba en busca de las

Tabla 5–1: Valores de probabilidad asignados por cada heurística.

Heurística	**Probabilidad**
Salto de ciclo	0.88
Código de operación	0.84
Salida de un ciclo	0.80
Inicio de ciclo	0.75
Llamado de subrutina	0.78
Retorno de subrutina	0.72
Salvar en memoria	0.55

instrucciones de salto condicional y aplica cada una de las heurísticas para determinar si las condiciones de cada heurística se cumplen. A continuación se presenta los algoritmos utilizados para implementar cada una de las heurísticas, las cuales han sido adaptadas de Larus y Wu [21].

Algoritmo 4. *Heurística de salto de ciclo LBH*

Entrada: Programa en lenguaje ensamblador, CFG.
Salida: Probabilidades asignadas a saltos.

1. **Para** cada nodo con instrucción de salto **Hacer**
   1.1. **Si** nodo es cabecera **OR** nodo tiene arista de retroceso **Entonces**
      1.1.1. **Si** una arista sale del ciclo **Entonces**
         1.1.1.1. Heurística Encontrada;
         1.1.1.2. Arista que NO sale del ciclo ← probabilidad;
      1.1.2. **De lo contrario**
         1.1.2.1. Heurística NO Aplica;
2. **Fin.**

Algoritmo 5. *Heurística de código de operación OH*

Entrada: Programa en lenguaje ensamblador, CFG.
Salida: Probabilidades asignadas a saltos.

1. **Para** cada nodo con instrucción de salto **Hacer**
   1.1. **Si** compara con $\{<, \leq\}$ a cero **OR** $\{=\}$ a constante **Entonces**
      1.1.1. Heurística Encontrada;
      1.1.2. Arista que falla la comparación ← probabilidad;
   1.2. **De lo contrario**
      1.2.1. Heurística NO Aplica;
2. **Fin.**

Algoritmo 6. *Heurística de salida de un ciclo LEH*

Entrada: Programa en lenguaje ensamblador, CFG.
Salida: Probabilidades asignadas a saltos.
1. **Para** cada nodo con instrucción de salto **Hacer**
   1.1. **Si** $nodo \in \{ciclo\}$ **Entonces**
      1.1.1. **Si** $sucesor_A \notin \{ciclo\}$ **AND** $sucesor_B$ no es cabecera **Entonces**
         1.1.1.1. Heurística Encontrada;
         1.1.1.2. $Sucesor_B \leftarrow$ probabilidad;
      1.1.2. **De lo contrario**
         1.1.2.1. Heurística NO Aplica;
2. **Fin.**

Algoritmo 7. *Heurística de inicio de ciclo LHH*

Entrada: Programa en lenguaje ensamblador, CFG.
Salida: Probabilidades asignadas a saltos.
1. **Para** cada nodo con instrucción de salto **Hacer**
   1.1. **Si** $sucesor_A \in \{ciclo\}$ **AND** $sucesor_B \notin \{ciclo\}$ **Entonces**
      1.1.1. $X \leftarrow sucesor_A$
      1.1.2. **Si** $X \notin postdominantes\,\{Nodo_{Actual}\}$
         1.1.2.1. Heurística Encontrada;
         1.1.2.2. Arista hacia $Sucesor_A \leftarrow$ probabilidad;
      1.1.3. **De lo contrario**
         1.1.3.1. Heurística NO Aplica;
2. **Fin.**

Algoritmo 8. *Heurística de llamado de subrutina CH*

Entrada: Programa en lenguaje ensamblador, CFG.
Salida: Probabilidades asignadas a saltos.
1. **Para** cada nodo con instrucción de salto **Hacer**
   1.1. **Si** solo un sucesor llama a una subrutina **Entonces**
      1.1.1. $X \leftarrow$ Sucesor que llama la subrutina;
      1.1.2. **Si** $X \notin postdominantes\,\{Nodo_{Actual}\}$
         1.1.2.1. Heurística Encontrada;
         1.1.2.2. Arista hacia X $\leftarrow$ probabilidad;
      1.1.3. **De lo contrario**
         1.1.3.1. Heurística NO Aplica;
2. **Fin.**

Algoritmo 9. *Heurística de retorno de subrutina RH*

Entrada: Programa en lenguaje ensamblador, CFG.
Salida: Probabilidades asignadas a saltos.
1. **Para** cada nodo con instrucción de salto **Hacer**
   1.1. **Si** solo un sucesor incluye instrucción de retorno **Entonces**

1.1.1. Heurística Encontrada;
1.1.2. X ← sucesor que *NO* incluye instrucción de retorno;
1.1.3. Arista hacia X ← probabilidad;
1.2. **De lo contrario**
1.2.1. Heurística NO Aplica;
2. **Fin.**

Algoritmo 10. *Heurística de salvar en memoria SH*

Entrada: Programa en lenguaje ensamblador, CFG.
Salida: Probabilidades asignadas a saltos.
1. **Para** cada nodo con instrucción de salto **Hacer**
1.1. **Si** solo un sucesor salva datos en memoria **Entonces**
1.1.1. $X \leftarrow$ Sucesor que salva datos en memoria;
1.1.2. $Y \leftarrow$ Sucesor que *NO* salva datos en memoria;
1.1.3. **Si** $X \notin postdominantes\,\{Nodo_{Actual}\}$
1.1.3.1. Heurística Encontrada;
1.1.3.2. Arista hacia Y ← probabilidad;
1.1.4. **De lo contrario**
1.1.4.1. Heurística NO Aplica;
2. **Fin.**

Es probable que más de una heurística aplique a una instrucción de salto. En este caso se utiliza un método que combina los valores de probabilidad de las heurísticas que aplican, generando un único valor que será el asignado a cada arista del bloque que contiene el salto. A continuación se presenta la ecuación utilizada.

$$P_{Salto} = P_{Actual-Salto} \oplus P_{Nueva-Salto} = \frac{u_S \cdot v_S}{u_S \cdot v_S + (1 - u_S) \cdot (1 - v_S)} \tag{5.1}$$

$$P_{No-Salto} = P_{Actual-Nosalto} \oplus P_{Nueva-Nosalto} = \frac{(1 - u_{NS}) \cdot (1 - v_{NS})}{u_{NS} \cdot v_{NS} + (1 - u_{NS}) \cdot (1 - v_{NS})} \tag{5.2}$$

Donde u corresponde al valor actual de probabilidad y v al valor de la nueva heurística aplicada al salto. Los resultados de estas expresiones hacen que el valor de la probabilidad asignada a un salto varíe en proporción a las probabilidades individuales de las heurísticas.

5.3. Análisis de Ciclos

El objetivo de esta herramienta es detectar los ciclos que contiene el programa bajo prueba. La detección se realiza a partir del CFG y está dividida en dos partes. La primera parte determina si existe un ciclo en el programa. De ser así, se procede a detectar que nodos o bloques básicos pertenecen a este ciclo y cuales de ellos son el nodo cabecera de ciclo y el nodo que contiene la arista de retroceso. La segunda parte analiza las instrucciones que componen el nodo cabecera de ciclo para determinar cuantas iteraciones realiza el ciclo. A continuación se detalla como se realizaron estas dos partes.

5.3.1. Detección de Ciclos

Antes de presentar el algoritmo de detección de ciclos, es necesario observar las siguientes definiciones.

- Un nodo v domina a otro nodo w si todas las rutas desde el nodo inicial hasta el nodo w, deben pasar primero por el nodo v.
- Un nodo w postdomina a un nodo v si todas las rutas desde el nodo v eventualmente pasan por el nodo w.
- Si el sucesor de un nodo lo domina, se ha encontrado un ciclo, donde el sucesor es la cabecera del ciclo y el nodo actual es el nodo que contiene la arista de retroceso.
- Todos los nodos que pueden alcanzar al nodo con arista de retroceso sin pasar por la cabecera de ciclo, forman parte del ciclo.

Basado en las definiciones anteriores, se plantea el siguiente algortimo para la detección de ciclos en un CFG, el cual han sido adaptado de Aho y asociados [19].

Algoritmo 11. *Detección de Ciclos en un CFG*

Entrada: Grafica de Control de Flujo (CFG).
Salida: Listado de nodos pertenecientes a un ciclo.

1. **Para** cada nodo del CFG **Hacer**
   1.1. Generar nodos dominantes;
2. Vaciar lista;

3. **Para** cada nodo del CFG **Hacer**
   3.1. $y \leftarrow$ {nodos que dominan al nodo actual};
   3.2. $x \leftarrow$ {nodos sucesores del nodo actual};
   3.3. **Si** algún elemento de $x \in y$ **Entonces**
      3.3.1. Marcar nodo sucesor que $\in y$ como cabecera;
      3.3.2. Marcar nodo actual como nodo de retroceso;
      3.3.3. lista$\leftarrow$ lista $\cup$ {nodo cabecera, nodo de retroceso};
4. **Para** cada nodo cabecera en lista **Hacer**
   4.1. Vaciar la Pila;
   4.2. ciclo$\leftarrow$ nodo cabecera;
   4.3. pila$\leftarrow$ nodo de retroceso;
   4.4. **Mientras** la pila no este vacía **Hacer**
      4.4.1. Tomar un nodo de la pila;
      4.4.2. pila$\leftarrow$ predecesores del nodo actual;
      4.4.3. ciclo$\leftarrow$ ciclo $\cup$ {nodo predecesor};
5. **Fin.**

5.3.2. Estimación de las Iteraciones de un Ciclo

Una vez identificados los ciclos en un programa, se procede a determinar la cantidad de iteraciones de cada ciclo. Para obtener esta información se procede con la metodología propuesta por Healy y asociados [25]. El primer paso consiste en determinar cuales nodos con instrucciones de salto pueden afectar la cantidad de iteraciones de un ciclo. Luego se procede a determinar la condición en la cuál cada salto sale del ciclo. Después se determina el posible rango de iteraciones del ciclo donde el salto puede cambiar de dirección. Por último, se determina el número de iteraciones totales para el ciclo bajo análisis. A continuacíon se explican cada uno de estos pasos.

Detección de Nodos

La primera parte consiste en determinar los nodos que poseen instrucciones de salto que pueden afectar la cantidad de iteraciones de un ciclo. Normalmente estos nodos tienen una arista que sale del ciclo o se dirige a la cabecera del ciclo o a un nodo que es postdominado por la cabecera del ciclo. Para diferenciar estos nodos, se le asigna el nombre *iterbrach* a los nodos que cumplan con estas condiciones.

A continuación se presenta el algoritmo que detecta dichos nodos, el cual han sido adaptado de Healy y otros [25].

Algoritmo 12. *Detección de nodos Iterbranch*

Entrada: Nodos de un ciclo.
Salida: Listado de nodos que pueden afectar la cantidad de iteraciones de un ciclo.

1. Vaciar lista de nodos iterbrach;
2. **Mientras** lista iterbranch sea modificada **Hacer**
   2.1. **Para** cada nodo del ciclo **Hacer**
      2.1.1. **Si** el nodo tiene 2 sucesores s1 y s2 **Entonces**
         2.1.1.1. **Si** s1 o s2 $\notin$ ciclo **OR**
         s1 o s2 $\in$ Postdominadores(cabecera) **OR**
         s1 o s2 $\in$ Postdominadores(algun nodo de lista iterbranch) **Entonces**
            2.1.1.1.1. lista iterbranch$\leftarrow$ lista iterbranch $\cup$ nodo actual;
3. **Fin.**

Ejemplo 2. *La Figura 5–3 presenta un ejemplo de identificación de ciclos y nodos iterbranch. En este caso, el nodo cabecera del ciclo es el nodo 3. Al mismo tiempo este nodo es identificado como nodo iterbranch, pues este nodo puede salir del ciclo.*

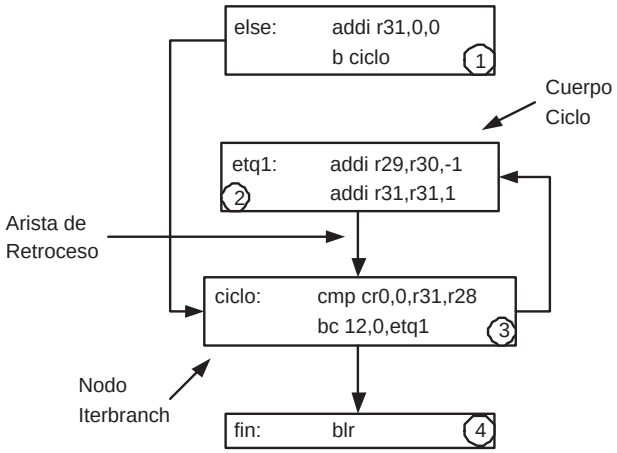

Figura 5–3: Ejemplo de identificación de ciclo y nodos iterbranch.

Una vez identificados los nodos *iterbrach*, se procede a generar una gráfica que muestra el orden de precedencia de los diferentes nodos iterbranch en el ciclo. Se agregan dos nodos extra para indicar cuándo una arista sale del ciclo o se dirige a la cabecera del ciclo. Estos nodos son llamados *continue* y *break* respectivamente.

A continuación se presenta el algoritmo que genera esta gráfica, el cual ha sido implementado a partir de la información provista por Healy y otros [25].

Algoritmo 13. *Generador grafica nodos iterbranch*

Entrada: nodos de un ciclo y lista de nodos iterbranch.
Salida: Grafica que relaciona los nodos iterbranch.

1. Generar Tabla con nodos iterbranch y sucesores;
2. **Para** cada nodo iterbranch **Hacer**
   - 2.1. **Si** algún sucesor no pertenece al ciclo **Entonces**
     - 2.1.1. Reemplazarlo con nodo *break*;
   - 2.2. **Si** algún sucesor no es un nodo iterbranch **Entonces**
     - 2.2.1. Reemplazarlo con nodo el primer nodo iterbranch que lo postdomine;
   - 2.3. **Si** algún sucesor es cabecera del ciclo **Entonces**
     - 2.3.1. Reemplazarlo con nodo *continue*;
   - 2.4. Generar gráfica para tabla resultante;
3. **Fin.**

Cambio de Dirección

La segunda parte consiste en determinar cuándo una instrucción de salto puede cambiar de dirección, con lo cual afecte el número de iteraciones. En este caso, debe analizarse la codificación de las instrucciones de comparación y de salto, para identificar los siguientes elementos:

- **Contador**. Es la variable que lleva el conteo de las iteraciones del ciclo.
- **Limite**. Es un valor o una expresión con la cual la variable contadora es comparada.
- **Operador**. Operador de relación utilizado en la comparación.
- **Inicial**. Valor inicial de la variable contadora.
- **Incremento**. Valor por el cual se incrementa o decrementa la variable contadora.
- **Ajuste**. Compensación en la cantidad de iteraciones debido al operador de relación utilizado. El valor del ajuste es uno o cero.

Si estos elementos logran definirse para cada instrucción de salto, se dice que el salto es *conocido*. En tal caso, el valor en el cuál la instrucción de salto cambia de

dirección esta dada por la ecuación 5.3.

$$N = \left\lfloor \frac{Limite - Inicial}{Incremento} \right\rfloor + Ajuste \tag{5.3}$$

En el caso de no identificar alguno de los elementos, se dice que el salto es *desconocido*. En tal caso, no es posible calcular el valor de cambio del salto.

El algoritmo para detectar cada uno de los elementos anteriores se divide en dos etapas. la primera etapa examina el programa en busca de los valores limites de los ciclos del programa. Si estos resultan ser variables, el programa genera una tabla con los nombres de las variables, los registros en el programa ensamblado que han sido asignados a dichas variables y el valor estimado de dicha variable. El valor estimado debe ser introducido por el usuario antes de iniciar el análisis del programa. La segunda etapa analiza el programa en lenguaje ensamblador, a partir del nodo *iterbrach*, en busca de los elementos descritos en la ecuación 5.3. Cuando encuentra variables, el programa recurre a la tabla generada por la primera etapa para leer el valor estimado. A continuación se explica con mas detalle las dos etapas de este programa.

El programa de la primera etapa realiza los siguientes cuatro pasos para generar la tabla de asignación:

- Generar el CFG del programa a partir del lenguaje de alto nivel.
- Generar el CFG del programa a partir del lenguaje ensamblador.
- Realizar una *igualación* de las dos graficas, en busca de los nodos cabecera de ciclo.
- Analizar las instrucciones de los nodos para extraer la variables limite y guardar la información en la tabla.

Por igualación se debe entender la identificación de los nodos que hacen la misma función en las dos graficas. Este método está basado en la idea que para un mismo programa, las gráficas generadas a partir del lenguaje de alto nivel y a partir del

lenguaje ensamblador son similares, pues la lógica del programa no cambia independiente de su codificación. Otra razón para utilizar este método es que el ensamblador no genera la información requerida y no era posible modificarlo para obtener los datos deseados. Por último, es común que el ensamblador reutilice registros en algunas partes del programa, lo cuál puede generar confusión en la asignación de registros a nombres de variables. Gracias a la identificación de las dos gráficas, se puede generar información adicional que evite confusiones. A continuación se presenta el algoritmo para esta etapa, es cual es producto propio de este proyecto.

Algoritmo 14. *Identificador de Variables*

Entrada: CFG y código de alto nivel.
Salida: Asignación de variables y registros.

1. **Proceso** Identificar estructuras en programa de alto nivel;
   - 1.1. Detectar palabras clave: IF, FOR, WHILE, ELSE, {, };
   - 1.2. Identificar llaves ({, }) relacionadas;
   - 1.3. Relacionar llaves con palabras clave;
   - 1.4. Relacionar IF con ELSE;
2. **Proceso** Crear Estructura (tipo,nodo inicial,nodo final);
   - 2.1. **Si** la estructura es un IF **Entonces**
     - 2.1.1. Crear nodos THEN, ELSE, ENDIF;
     - 2.1.2. Asignar arista entre los nodos;
     - 2.1.3. Relacionar nodos con instrucciones del programa;
     - 2.1.4. Llamar recursivamente a: Crear Estructura (tipo,nodo THEN,nodo ENDIF);
     - 2.1.5. Llamar recursivamente a: Crear Estructura (tipo,nodo ELSE,nodo ENDIF);
     - 2.1.6. Hacer nodo ENDIF igual a nodo final;
   - 2.2. **Si** la estructura es un FOR o WHILE **Entonces**
     - 2.2.1. Crear nodos PREHEADER, HEADER, BODY, END;
     - 2.2.2. Asignar arista entre los nodos;
     - 2.2.3. Relacionar nodos con instrucciones del programa;
     - 2.2.4. Llamar recursivamente a: Crear Estructura (tipo,nodo BODY,nodo END);
     - 2.2.5. Hacer nodo END igual a nodo final;
3. **Programa** principal;
   - 3.1. Llamar proceso: Identificar estructuras en programa de alto nivel;
   - 3.2. Crear nodo inicial y final del CFG;
   - 3.3. **Para** cada estructura del programa **Hacer**

3.3.1. Llamar proceso: Crear Estructura (tipo estructura, nodo inicial, nodo final);
3.3.2. Actualizar nodo inicial y final;
3.4. Leer CFG de bajo nivel;
3.5. Realizar recorrido del CFG de alto nivel;
3.5.1. Si el nodo visitado en una cabecera, marcarlo;
3.6. Realizar recorrido del CFG de bajo nivel;
3.6.1. Si el nodo visitado en una cabecera, marcarlo;
3.6.2. **Si** nodo actual tiene la misma marca de un nodo de alto nivel **Entonces**
3.6.2.1. Identificar variables de alto nivel;
3.6.2.2. Leer registros de bajo nivel;
3.6.2.3. Guardar información como nombres equivalentes;
3.7. Buscar en la información de usuario, los valores de variables;
3.8. salvar infromación.
4. **Fin.**

La segunda etapa se encarga de identificar todos los elementos para el cálculo del valor en el cual el salto cambia de dirección. A continuación se presenta el algoritmo, el cual ha sido implementado a partir de la información de Healy y otros [25].

Algoritmo 15. *Estimación de cambio de dirección de nodos iterbranch*

Entrada: Nodo iterbranch, nodos del ciclo, código del programa.
Salida: Valor de cambio (N).
1. **Para** cada nodo iterbranch **Hacer**
1.1. Determinar valor de comparación;
1.2. Identificar operador de relación;
1.3. **Para** cada instrucción del ciclo anterior al nodo iterbranch **Hacer**
1.3.1. Identificar la instrucción de incremento;
1.4. **Para** cada instrucción anterior al ciclo **Hacer**
1.4.1. Identificar la instrucción de inicialización;
1.5. $N \leftarrow 0$;
1.6. **Si** todos los valores fueron identificados **Entonces**
1.6.1. Calcular N;
1.7. **De lo Contrario**
1.7.1. Declarar N desconocido;
2. **Fin.**

Rango de Iteraciones

La tercera parte consiste en determinar un rango de valores de iteración del ciclo, donde cada uno de los nodos *iterbranch* puede cambiar de dirección. Esta

información se obtiene a partir de la gráfica de nodos iterbranch construida en la primera parte y los valores de cambio obtenidos en la segunda parte. El proceso consiste en recorrer la gráfica, calculando para cada nodo que se visita, un rango que es la unión de los rangos de las aristas que llegan a dicho nodo. Para esto se utilizan las siguientes reglas:

- El nodo raíz de la gráfica tiene un rango predeterminado $[1..\infty]$.
- Si el nodo *iterbranch* es conocido, el operador y el signo del incremento se utiliza para determinar la arista que toma el rango resultante de la intersección del rango del nodo actual y $[1..N_i]$, donde N_i es el valor de cambio de dicho nodo. La otra arista del nodo toma el rango resultante de la intersección del rango del nodo actual y $[N_i..\infty]$.
- Si el nodo *iterbranch* es desconocido, se asigna a las dos aristas el mismo rango del nodo.

A continuación se presenta el algoritmo que determina el rango deseado, el cual ha sido implementado a partir de la información de Healy y otros [25].

Algoritmo 16. *Estimación del rango de cambio de los nodos iterbranch*

Entrada: nodo iterbranch, Rango Entrada (RngIn).
Salida: Rango del nodo iterbranch de entrada.

1. **Si** el nodo existe **Entonces**
   1.1. **Si** el nodo no tiene rango asignado **Entonces**
      1.1.1. Rango nodo actual ← RngIn;
   1.2. **De lo Contrario**
      1.2.1. X← rango nodo actual ∪ RngIn;
   1.3. RangoA ← [1..N] ∩ rango nodo actual;
   1.4. RangoB ← $[N..\infty]$ ∩ rango nodo actual;
   1.5. **Si** nodo es conocido **Entonces**
      1.5.1. **Si** operador es $<$ o $\leq$ **Entonces**
         1.5.1.1. Rango arista taken ← RangoA;
         1.5.1.2. Rango arista No taken ← RangoB;
      1.5.2. **Si** operador es $>$ o $\geq$ **Entonces**
         1.5.2.1. Rango arista taken ← RangoB;
         1.5.2.2. Rango arista No taken ← RangoA;
   1.6. **De lo Contrario**
      1.6.1. Rango arista taken ← RngIn;

1.6.2. Rango arista No taken ← RngIn;
1.7. Llamada recursiva con argumentos (nodo taken, rango arista taken);
1.8. Llamada recursiva con argumentos (nodo NO taken, rango arista NO taken);
2. **Fin.**

Iteraciones Totales

La última parte del programa se encarga de determinar el valor final de las iteraciones del ciclo. Este proceso se realiza calculando valores mínimo y máximo de iteraciones para cada nodo y arista de la gráfica de nodos *iterbranch*, haciendo un recorrido desde el fondo de la gráfica hasta el nodo raíz. El valor asignado al nodo raíz es el valor de iteraciones de todo el ciclo. Las reglas para asignar valor mínimo y máximo a una arista son:

- Si la arista se dirige a un nodo *break*, entonces los valores mínimo y máximo de la arista son iguales al valor mínimo del rango de dicha arista.
- Si una arista se dirige a un nodo *continue*, entonces los valores mínimo y máximo de la arista son indefinidos.
- Si la arista se dirige a un nodo iterbranch, el valor mínimo y máximo se calculan en base a las siguientes reglas:
  - si el valor inferior del rango del nodo es menor que el valor inferior del rango de la arista, el valor asignado es el valor inferior de la arista.
  - Si el valor inferior del rango de la arista es menor o igual al valor del rango del nodo y este último valor es menor o igual que el valor superior del rango de la arista, entonces el valor asignado es el valor del nodo.
  - Si el valor superior del rango de la arista es menor que el rango del nodo, entonces el valor asignado es indefinido.

  Las reglas para asignar los valores mínimo y máximo a un nodo son:

- El valor mínimo del nodo es el menor de los valores mínimos de las aristas que salen del nodo. Si ambas aristas tienen valores mínimos indefinidos, ese mismo valor se asigna al nodo.
- Si el nodo es conocido, entonces el valor máximo del nodo es el menor de los valores máximos de las aristas del nodo. Si los valores de las aristas son indefinidos, este mismo valor se asigna al nodo.
- Si el nodo es desconocido, el valor máximo del nodo es el mayor de los valores máximos de las aristas del nodo. Si alguna de las aristas tiene valor máximo indefinido, ese valor se asigna al nodo.

A continuación se presenta el algoritmo de esta parte del programa, el cual ha sido implementado a partir de la información de Healy y otros [25].

Algoritmo 17. *Estimación de iteraciones de un ciclo*

Entrada: nodo iterbranch.
Salida: Valor estimado de iteraciones.

1. **Si** nodo existe y no es break o *continue* **Entonces**
   1.1. Llamar en forma recursiva con argumento (nodo taken);
   1.2. Llamar en forma recursiva con argumento (nodo No taken);
   1.3. **Para** cada sucesor del nodo actual **Hacer**
      1.3.1. **Si** el sucesor es un nodo *break* **Entonces**
         1.3.1.1. Mínimo arista = Máximo de arista = valor inferor rango arista;
      1.3.2. **Si** el sucesor es un nodo *continue* **Entonces**
         1.3.2.1. Mínimo arista = Máximo de arista = indefinido;
      1.3.3. **Si** el sucesor es iterbranch **Entonces**
         1.3.3.1. **Si** mínimo de nodo No taken $<$ inferior rango arista **Entonces**
            1.3.3.1.1. Mínimo arista= inferior rango arista;
         1.3.3.2. **Si** inferior rango arista $\leq$ mínimo nodo taken **AND** mínimo nodo taken $\leq$ superior rango arista **Entonces**
            1.3.3.2.1. Mínimo arista= mínimo nodo taken;
         1.3.3.3. **Si** superior rango arista $<$ mínimo nodo taken **Entonces**
            1.3.3.3.1. Mínimo arista= indefinido;
         1.3.3.4. **Si** máximo nodo taken $<$ inferior rango arista **Entonces**
            1.3.3.4.1. Máximo arista= inferior rango arista;
         1.3.3.5. **Si** inferior rango arista $\leq$ maximo nodo taken **AND** máximo nodo taken $\leq$ superior rango arista **Entonces**
            1.3.3.5.1. Máximo arista= máximo nodo taken;
         1.3.3.6. **Si** superior rango arista $<$ máximo nodo taken **Entonces**

1.3.3.6.1. Mínimo arista= indefinido;
1.4. **Si** mínimo arista taken < mínimo arista No taken **Entonces**
1.4.1. Mínimo nodo= mínimo arista taken;
1.5. **De lo contrario**
1.5.1. Mínimo nodo= mínimo arista No taken;
1.6. **Si** nodo es conocido **Entonces**
1.6.1. **Si** maximo arista taken < máximo arista No taken **Entonces**
1.6.1.1. Máximo nodo= máximo arista No taken;
1.6.2. **De lo contrario**
1.6.2.1. Máximo nodo= máximo arista taken;
1.7. **De lo contrario**
1.7.1. **Si** maximo arista taken > máximo arista No taken **Entonces**
1.7.1.1. Máximo nodo= máximo arista taken;
1.7.2. **De lo contrario**
1.7.2.1. Máximo nodo= máximo arista No taken;
2. **Fin.**

5.4. Rutas de Ejecución

Este programa tiene como función extraer todas las posibles rutas de ejecución que tiene el CFG del programa bajo prueba. De esta forma se puede analizar todos los posibles comportamientos del programa y se puede estimar el consumo de potencia y energía según la ruta que tome el programa bajo prueba.

El algoritmo utilizado construye primero un árbol a partir del CFG y luego utiliza un algoritmo de recorrido para generar las diferentes rutas del programa. Sin embargo, antes de crear el árbol es necesario eliminar las aristas de retroceso de ciclos, ya que estas pueden causar una recursión infinita en el programa generador del árbol. La consecuencia de esto es la generación de rutas adicionales que terminan con el nodo que posee la arista de retroceso del ciclo. Esto es conveniente porque se puede disponer de las rutas internas de cada ciclo, las cuales se pueden utilizar para el cálculo de potencia y energía de dicho ciclo.

A continuación se presentan las diferencias del programa generador de árbol utilizado en el proyecto respecto a un programa generador de árbol tradicional.

- Existe un orden preestablecido para ubicar los nodos del árbol.

- Un nodo tiene como máximo dos aristas de salida.
- Dos nodos diferentes pueden tener aristas a un mismo nodo destino.
- Los nodos requieren información adicional para evitar la exploración repetida de nodos.

A continuación se presentan los algoritmos utilizados para crear el árbol y la generación de las rutas del programa, los cuales están basados en algoritmos clasicos de estructuras de datos [34].

Algoritmo 18. *Generador de árbol*

Entrada: Tabla descriptiva de nodos con sucesores.
Salida: árbol.
1. Eliminar aristas de retroceso de la tabla;
2. Crear nodo Raiz de árbol con nodo inicial de la tabla;
3. lista ← {nodos de la tabla};
4. **Mientras** la lista no este vacia **Hacer**
   4.1. X ← lista de nodos en el árbol;
   4.2. Tomar nodo de lista;
   4.3. **Si** nodo ∈ X **Entonces**
      4.3.1. Guardar sucesores del nodo en el árbol;
5. **Fin.**

Algoritmo 19. *Recorrido de árbol*

Entrada: árbol.
Salida: Rutas del árbol.
1. **Si** nodo existe **Entonces**
   1.1. Guardar nodo en la pila;
   1.2. **Si** nodo es fin de ruta **Entonces**
      1.2.1. Ruta ← {nodos almacenados en la pila};
   1.3. **De lo contrario**
      1.3.1. Llamada recursiva con argumento (nodo sucesor izquierdo);
      1.3.2. Llamada recursiva con argumento (nodo sucesor derecho);
   1.4. Extraer nodo de la pila;
2. **Fin.**

Las rutas obtenidas del programa se complementan con la información adicional obtenida en los programas anteriores. Así, las rutas de un ciclo tienen información de cuantas veces se repiten y las probabilidades de transición de una nodo a otro.

Ejemplo 3. *La Figura 5–4 presenta un ejemplo de generación de rutas a partir de la información provista por las herramientas de análisis estático. La primera ruta se identifica como una ruta de ciclo, por terminar con el nodo que posee la arista de retroceso del ciclo. Las demás rutas son rutas generales del programa.*

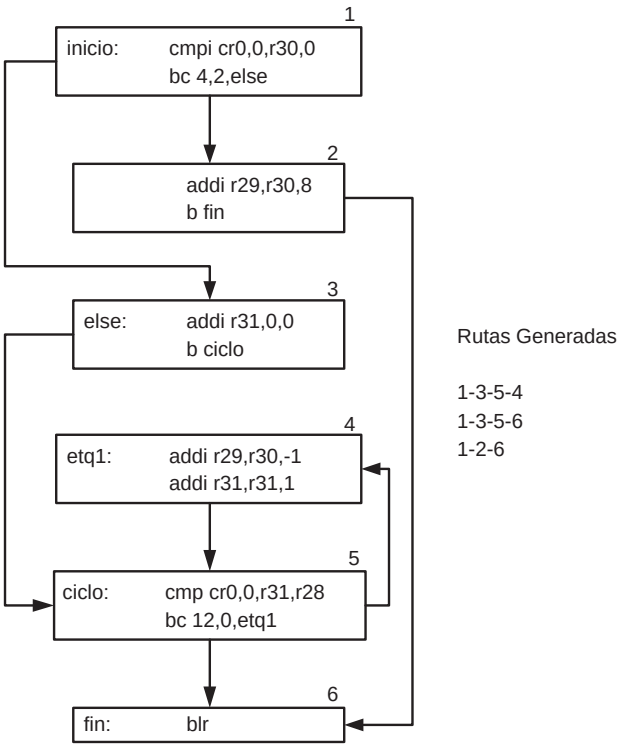

Figura 5–4: Ejemplo de rutas obtenidas a partir del algoritmo presentado.

Esta información se puede aplicar al modelo de energía para obtener el estimado de potencia y energía por la ejecución de un programa, como lo explica el capítulo siguiente.

Resumen

Este capítulo presentó el conjunto de herramientas de análisis estático empleadas en este trabajo. Estas herramientas buscan analizar los aspectos mas importantes de un programa y generan una predicción de su comportamiento basado en heurísticas implementadas en base a probabilidades. Esta información es utilizada por los demás componentes del sistema para generar el estimado de consumo de potencia y energía.

Capítulo 6
METODOLOGÍA DE ESTIMACIÓN DE POTENCIA Y ENERGÍA

El estimador de potencia y energía es un programa que utiliza la información provista por el perfil de potencia de las instrucciones y el análisis estático de programas para generar un valor estimado de potencia y energía para el programa bajo prueba.

Para calcular el estimado, se analiza cada uno de los nodos del programa bajo prueba, determinando su tiempo de ejecución y el costo de energía. Luego, se combinan los diferentes costos de los nodos, dependiendo de la ruta bajo análisis y la probabilidad de ejecución de dicha ruta, obteniendo el valor total estimado de potencia y energía para el programa bajo análisis. La Figura 6–1 presenta la metodología empleada para determinar los valores de potencia y energía. El bloque estimador de tiempo se encarga de determinar cuánto tiempo se requiere para ejecutar cada nodo del programa. El estimador de costo de energía se encarga de estimar el valor de energía para cada nodo del programa. El bloque estimador de ciclos analiza cada ciclo para determinar el costo por iteración. Este análisis lo realiza identificando las rutas que pertenecen al ciclo, las cuáles combina con la información de los costos por nodo para obtener el costo total del ciclo. El bloque final es el encargado de combinar la información de los bloques anteriores para determinar el costo total de potencia y energía del programa. Las secciones siguientes explican la implementación de este programa así: La sección 6.1 presenta el estimador de tiempo de ejecución implemetado en este proyecto. La sección 6.2 presenta la estimación de energía para

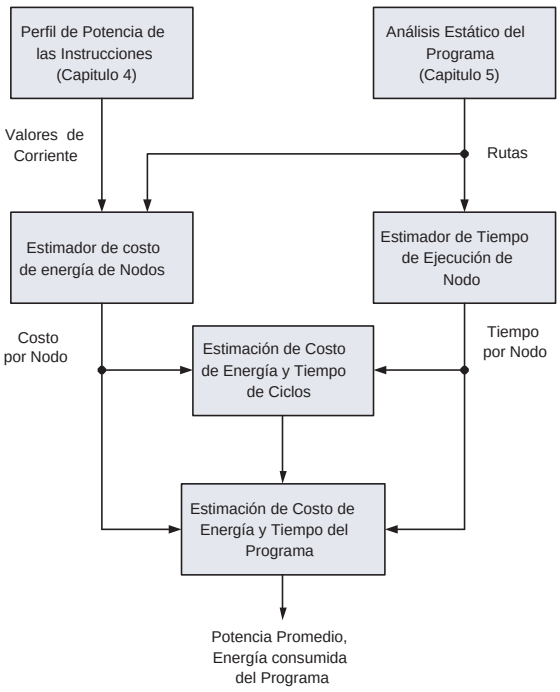

Figura 6–1: Esquema del estimador de potencia y energía

cada nodo del programa. La sección 6.3 presenta como se combinan los resultados anteriores para estimar el costo por iteración de cada ciclo y obtener el valor final de consumo de energía y potencia del programa.

6.1. Estimación del Tiempo de Ejecución

La estimación del tiempo de ejecución de los nodos de un CFG se basa en la implementación de un modelo del microprocesador PowerPC 603e, el cuál determina el comportamiento del microprocesador ciclo a ciclo, utilizando solo la información provista por el analizador estático de programas. La Figura 6–2 presenta su diagrama en bloques.

Cada uno de estos bloques representa un programa que simula el comportamiento de la unidad funcional indicada. La información sobre el comportamiento de cada unidad fue extraida del manual de usuario del microprocesador PowerPC 603e y de algunos artículos publicados por Motorola [35] [36]. A continuación se presenta una breve descripción de las unidades funcionales que componen este modelo.

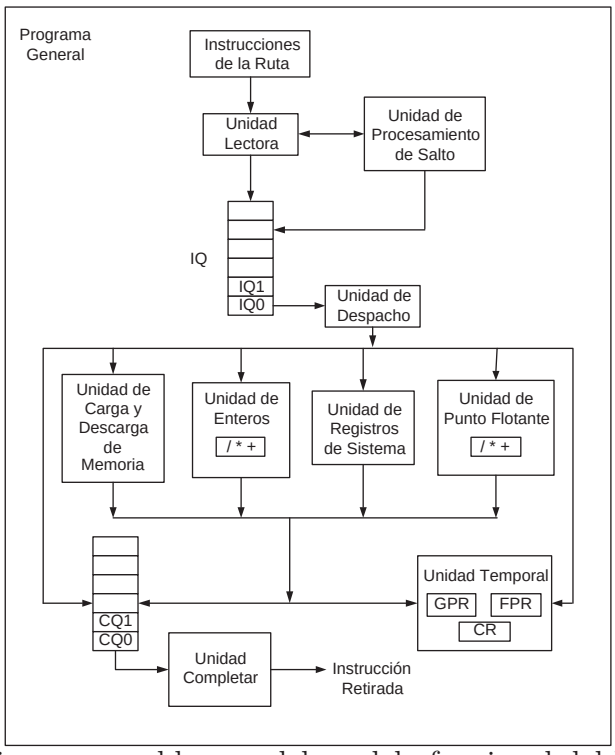

Figura 6–2: Diagrama en bloques del modelo funcional del PowerPC 603e

6.1.1. Unidad Lectora

La función de la unidad lectora es traer las instrucciones desde la memoria y almacenarlas en la unidad IQ. En este caso, las instrucciones son las indicadas por la ruta bajo análisis. Para implementar esta unidad se tiene una variable contadora de instrucción, cuya función es indicar cuál es la siguiente instrucción a traer. Adicionalmente puede marcar y desmarcar instrucciones como predichas, a partir de las indicaciones dadas por la unidad de procesamiento de saltos.

6.1.2. Unidad IQ

La función de la unidad IQ (Instruction Queue por sus siglas en inglés), es la de almacenar las instrucciones traidas por la unidad lectora. Para esto, cuenta con una memoria FIFO de 6 localidades y los métodos asociados a su manejo. En adición, esta unidad puede expulsar las instrucciones de la FIFO a partir de las ordenes dadas

por la unidad de procesamiento de saltos y tiene métodos para identificar ciertas instrucciones especiales como las de salto y comparación.

6.1.3. Unidad Despacho

La función de la unidad despacho es enviar instrucciones desde el IQ hacia las respectivas unidades de ejecución. Para esto, extrae la instrucción que esta en la posición cero de IQ, la decodifica para identificar a que unidad de ejecución debe ser asignada y verifica que existan los recursos necesario para la ejecución de la instrucción. Estos recursos son: La unidad de ejecución designada para esa instrucción debe estar libre. Debe existir suficientes registros temporales para almacenar los resultados de la instrucción. Debe existir una localidad disponible en la unidad CQ. Si estas condiciones se cumplen, entonces la instrucción puede ser despachada, con lo cuál se ocupa la unidad de ejecución adecuada y se reservan los registros temporales y el espacio en CQ para dicha instrucción. De no cumplirse alguna de las condiciones, la instrucción no es despachada y es devuelta a IQ.

Una vez se logra enviar la instrucción de la posición cero, despacho trata de enviar la instrucción en la posición uno, para lo cuál, realiza el mismo proceso de decodificación y verificación de recursos.

6.1.4. Unidad BPU

La unidad de procesamiento de saltos (Branch Processing Unit) es la encargada de analizar cada instrucción de salto que llega a IQ. Una se vez identifica una instrucción de salto, se trata de resolver dicha instrucción. Si la instrucción es salto incondicional, la unidad BPU puede resolver de inmediato el salto. Si el salto es condicional, se explora las instrucciones para determinar si alguna modifica el registro de condición con el cuál trabaja el salto. De no encontrarse alguna instrucción que modifique el registro de condición, el salto se puede resolver, de lo contrario, el

salto no se puede resolver y entonces se predice. Esta unidad utiliza una predicción estática de salto, indicando que un salto hacia una dirección anterior será tomada. Una vez generada una respuesta, la unidad BPU procede a indicarle a la unidad lectora de donde debe obtener las nuevas instrucciones. Una vez se ha procesado la instrucción de la que depende el salto, BPU procede a determinar si su predicción fué correcta o no. Si fué correcta, entonces se continua el flujo normal de las instrucciones. De no ser correcta, BPU indica a todas las unidades que deben realizar una expulsión de las instrucciones traidas erroneamente y le indica a lectora de donde debe traer las nuevas instrucciones.

6.1.5. Unidades de Ejecución

Se encargan de procesar cada instrucción. Se tienen cuatro unidades en esta parte. La unidad de enteros, la unidad de punto flotante, la unidad de leer y escribir en memoria y la unidad de sistema. La unidad de punto flotante está dividida en tres estaciones para procesar la instrucción. La unidad de leer y escribir en memoria esta dividida en dos estaciones, la primera para cálcular la dirección y la segunda para realizar la acción indicada por la instrucción. La unidad de enteros y de sistema son de una sola estación.

6.1.6. Unidad Temporal

Esta unidad contiene registros temporales que se utilizan para almacenar los resultados de las diferentes unidades de ejecución. Aunque los resultados que guarda esta unidad no son validos, dado que no se están procesando datos reales, si es importante la asignación de registros temporales que realiza, pues se puede causar un retraso en la ejecución de instrucciones debido a la cantidad limitada de registros temporales disponibles. Así, esta unidad incluye 5 registros temporales para datos

enteros, 4 para datos de punto flotante y registros para los indicadores de comparación y algunos registros adicionales. Además, incluye métodos para determinar si hay registros libres, para asignarlos a una instrucción en particular y almacenar los resultados indicados por la instrucción. También tiene indicadores para que otras unidades, como BPU puedan determinar si la instrucción de la que depende ya terminó su ejecución.

6.1.7. Unidad CQ

La unidad CQ (Completion Queue) es similar a IQ, en el sentido que es una memoria FIFO de 5 localidades, cuya función es almacenar el estado de cada instrucción que ha sido enviada por despacho a las diferentes unidades de ejecución. En general posee métodos similares a los implementados en IQ para expulsar y marcar instrucciones según ordenes enviadas por otras unidades.

6.1.8. Unidad Completar

Esta unidad retira de CQ las instrucciones que ya han sido procesadas y escribe en los registros arquitecturales los resultados de dichas instrucciones. Esta unidad puede retirar hasta 2 instrucciones en un solo ciclo, sin embargo, existen restricciones que evitan que esto suceda en algunas ocasiones, por ejemplo, las limitaciones del microprocesador para escribir en los registros arquitecturales. Sólo se puede escribir a un registro de punto flotante por ciclo.

6.1.9. Unidad Registros Arquitecturales

Son los registros que dispone el usuario para los programas. Aunque son funcionales, no se utilizan en la simulación por no utilizar datos reales.

6.1.10. Programa General

Las unidades anteriores se reunen en un sólo programa, el cuál realiza la interconexión de las diferentes unidades y las ejecuta en secuencia, simulando la acción total del microprocesador por cada ciclo de reloj. Para determinar el tiempo del nodo, se incluye un contador de ciclos de reloj que se va incrementando a medida que se avanza en el procesamiento de las instrucciones del nodo bajo análisis.

6.2. Estimación del Costo de Energía para cada Nodo del Programa

Para estimar el consumo de cada nodo del programa se ejecutan los siguientes pasos:

- Extracción de las instrucciones del nodo.
- Estimación del costo base de Corriente.
- Estimación del costo por conmutación de Corriente.
- Estimación del costo de corriente por efectos extra.
- Estimación de la energía del nodo.

A continuación se detallan los pasos mencionados anteriormente.

6.2.1. Extracción de Instrucciones

Cada nodo incluye un conjunto de instrucciones, las cuales han sido asignadas en el momento de generar el CFG del programa. Así, se implementó un programa que extrae el conjunto de instrucciones de cada nodo y las almacena en un archivo, en el orden indicado por el nodo, para ser posteriormente utilizado por los otros componentes del programa.

6.2.2. Costo de Corriente Base y Conmutación

La estimación del costo base de corriente y el costo por conmutación se realiza a partir de examinar cada instrucción que contiene el nodo bajo prueba. Para esto, se

aplica el modelo descrito en el Capítulo 4, donde se determina el costo de corriente promedio de un conjunto de instrucciones, utilizando la siguiente expresión:

$$I_B = \frac{1}{C_T} \sum_i I_{Bi} \cdot C_i \tag{6.1}$$

Donde:

I_B : Es la corriente base promedio de la secuencia de instrucciones.

I_{Bi} : Es la corriente base asignada a cada instrucción.

C_i : Es la cantidad de ciclos de cada instrucción.

C_T : Es la cantidad total de ciclos del conjunto de instrucciones.

El costo de corriente base de cada instrucción ya ha sido previamente determinado en el perfíl de potencia de las instrucciones. De forma similar, el costo por conmutación se determina al tomar parejas consecutivas de instrucciones y extraer el costo por conmutación asignado a dicho par del perfíl de potencia. Esta información se aplica a la siguiente expresión para determinar el costo por conmutación del nodo.

$$I_O = \frac{1}{C_T} \sum_i I_{Ov(i,i+1)} \tag{6.2}$$

Donde:

I_O : Es la corriente de conmutación promedio de la secuencia de instrucciones.

$I_{Ov(i,i+1)}$: Es la corriente de conmutación del par consecutivo de instrucciones.

C_T : Es la cantidad total de ciclos del conjunto de instrucciones.

La suma de los valores promedio de costo base y costo de conmutación dan la corriente promedio asignada al nodo.

6.2.3. Costo de Corriente por Efectos Extra

Para determinar la contribución de corriente por las predicciones erroneas de la unidad de procesamiento de saltos, es necesario identificar el nodo que es destino de

una instrucción de salto, para determinar si habrá o no un error de predicción. Debido a que se están analizando nodos individuales, no es posible determinar este costo en este momento. Así, este costo se determina cuando se esté analizando cada ruta del programa. En caso de estimar una predicción erronea, se agrega el costo de energía por error de predicción al nodo que incluya la instrucción de salto condicional.

Determinar la contribución de los desaciertos de la memoria cache es un proceso dificil para el analizador estático, debido al comportamiento dinámico de la memoria cache. En adición, no se conoce cuál es la secuencia real de instrucciones del programa y por lo tanto se desconoce cuáles instrucciones se encuentran, en un momento dado almacenadas en la memoria cache y finalmente porque no se cuentan con todos los datos reales que va a utilizar el programa. Así, la versión actual de este proyecto sólo considera casos donde no hay desaciertos de la memoria cache.

6.2.4. Estimación de la energía del nodo

Una vez determinado el consumo de corriente promedio para cada nodo del programa y el tiempo de ejecución, se puede determinar el consumo de energía del nodo utilizando la siguiente ecuación:

$$E = T \cdot V_{cc} \cdot I_{ave} \tag{6.3}$$

Donde I_{ave} es la corriente promedio del nodo, V_{cc} es el voltaje del microprocesador, el cual se supone constante a 2.5V y T es el tiempo de ejecución del nodo.

6.3. Estimación de Potencia y Energía del Programa

El costo de energía de un programa se determina como la suma del costo de cada nodo que compone el programa, multiplicado por la probabilidad que el programa llegue a cada uno de los nodos. De forma similar se puede determinar el tiempo de ejecución del programa. Una vez se determina la energía y el tiempo de ejecución, es

posible determinar la potencia promedio como la división de valor de energía sobre el tiempo de ejecución del programa. El ejemplo 4 ilustra como se puede determinar el costo de energía de un programa. La misma metodología se aplica para determinar el tiempo de ejecución e indirectamente el valor de potencia promedio.

Ejemplo 4. *La figura 6–3 presenta una grafica de un programa. El costo de energía*

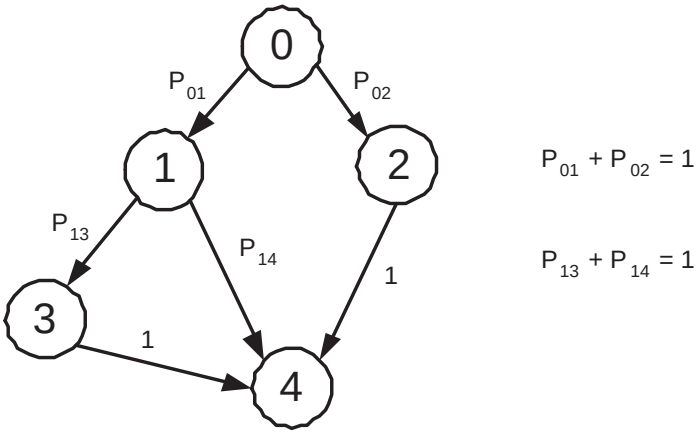

Figura 6–3: Ejemplo de un CFG

del programa está dado por la siguiente expresión:

$$E = E_0 + P_{01} \cdot E_1 + P_{02} \cdot E_2 + P_{01} \cdot P_{13} \cdot E_3 + E_4 \tag{6.4}$$

Si se consiera que un programa se ejecuta desde el nodo inicial hasta el nodo final, tomado una sola ruta del programa, se pueden estimar el costo del programa como un promedio ponderado del costo de cada ruta, multiplicado por la probabilidad que el programa ejecute cada ruta que lo compone. Si se aplica esta idea al ejemplo anterior se tiene:

Ejemplo 5. *A partir de la Figura 6–3 se tiene que el costo de energía del programa es:*

$$E = Ruta_1 \cdot P_{ruta1} + Ruta_2 \cdot P_{ruta2} + Ruta_3 \cdot P_{ruta3} \tag{6.5}$$

Al descomponer esta expresión, se obtiene lo siguiente:

$$\begin{aligned} E = &(E_0 + E_1 + E_3 + E_4) \cdot (P_{01} \cdot P_{13} \cdot 1) + \\ &(E_0 + E_1 + E_4) \cdot (P_{01} \cdot P_{14}) + \\ &(E_0 + E_2 + E_4) \cdot (P_{02} \cdot 1) \end{aligned} \tag{6.6}$$

Donde E_0 a E_4 son los costos de energía de cada nodo y P_{ij} son las probabilidades de transición asignadas a las aristas de cada nodo. Al descomponer esta expresión, se tiene que:

$$\begin{aligned} E = &E_0 \cdot (P_{01} \cdot P_{13} + P_{01} \cdot P_{14} + P_{02}) + \\ &E_1 \cdot (P_{01} \cdot P_{13} + P_{01} \cdot P_{14}) + \\ &E_2 \cdot (P_{02}) + E_3 \cdot (P_{01} \cdot P_{13}) + \\ &E_4 \cdot (P_{01} \cdot P_{13} + P_{01} \cdot P_{14} + P_{02}) \end{aligned} \tag{6.7}$$

Finalmente, utilizando las expresiones de la figura 6–3 para los valores de probabilidad, se obtiene lo siguiente:

$$E = E_0 + P_{01} \cdot E_1 + P_{02} \cdot E_2 + P_{01} \cdot P_{13} \cdot E_3 + E_4 \tag{6.8}$$

Esta expresión corresponde al costo del programa como el costo del nodo, multiplicado por la probabilidad de llegar a dicho nodo.

El ejemplo anterior muestra que es posible determinar el costo de un programa ya sea determinando la contribución individual de cada nodo o considerando las diferentes rutas de ejecución del programa. En este proyecto se implementa la opción de rutas debido a que esta forma permite la estimación de los costos de los ciclos en forma individual y facilita la implementación de los algoritmos de estimación de consumo de potencia y energía del programa.

A continuación se presenta la metodología empleada para estimar los costos de energía y potencia de un programa.

6.3.1. Potencia y Energía de un Ciclo

Un ciclo puede ser considerado como un programa que se repite una cantidad determinada de veces, donde el inicio del programa es la cabecera del ciclo y el final del programa es el nodo con la arista de retroceso. Por lo tanto, es posible analizar cada ciclo, utilizando las ideas expresadas anteriormente y obtener el costo esperado de una iteración del ciclo bajo estudio.

Dado que ya se tiene el consumo de energía, el tiempo estimado de nodo y las diferentes rutas que componen el ciclo, es posible determinar el costo de una iteración del ciclo con las siguientes expresiones:

$$T = \sum_{i=0}^{n} \alpha_i \cdot t_i \tag{6.9}$$

Donde:

T : Es el valor esperado de tiempo de una iteración.

t_i : Es el tiempo de ejecución de cada ruta del ciclo.

α_i : Es la probabilidad de ejecución de cada ruta. $\sum \alpha_i = 1$.

n : Es la cantidad de rutas del ciclo.

$$E = \sum_{i=0}^{n} \alpha_i \cdot E_i \tag{6.10}$$

Donde:

E : Valor esperado de energía de una iteración del ciclo.

E_i : Energía de cada ruta del programa.

α_i : Es la probabilidad asignada a cada ruta. $\sum \alpha_i = 1$.

n : Es la cantidad de rutas del ciclo.

$$P_{ave} = E/T \tag{6.11}$$

Donde:

E : Valor esperado de energía de una iteración del ciclo.

T : Valor esperado de tiempo de ejecución de una iteración del ciclo.

La energía de cada ruta se determina sumando las contribuciones individuales de energía de cada nodo que se incluye en la ruta. De forma similar, el tiempo de ejecución de la ruta se determina sumando los tiempos de los diferentes nodos que la componen. La potencia es la división de la energía entre el tiempo de ejecución. La probabilidad de la ruta se determina multiplicando los valores de probabilidad de las aristas incluidas en la ruta. Si existen ciclos anidados en el programa, se puede utilizar un proceso recursivo. Es decir, primero debe estimarse el aporte del ciclo interno en tiempo y energía y luego se determina el costo del ciclo externo que incluye al ciclo interno, considerando al último como un solo nodo. Es importante recordar que el valor que se calcula es el valor esperado de una iteración, por lo tanto, es necesario multiplicar por la cantidad de veces que se repite el ciclo, para determinar el aporte total del ciclo interno bajo análisis.

A continuación se presenta el algoritmo para determinar el valor esperado de energía y tiempo por iteración de cada ciclo.

Algoritmo 20. *Consumo de Energía y Tiempo de ejecución de los Ciclos de un Programa*

Entrada: Resultados del analizador estático, nodos caracterizados en energía y tiempo de ejecución.
Salida: Costo esperado de energía y tiempo de ejecución del ciclo.

1. $Tiempo \leftarrow 0$;
2. $Energia \leftarrow 0$;
3. $Potencia \leftarrow 0$;
4. **Para** cada ruta del ciclo **hacer:**
   - 4.1. $Tiempo_{ruta} \leftarrow 0$;
   - 4.2. $Energia_{ruta} \leftarrow 0$;
   - 4.3. **Para** cada nodo de la ruta **hacer:**
     - 4.3.1. $Tiempo_{nodo} \leftarrow 0$;
     - 4.3.2. $Ener_{nodo} \leftarrow 0$;
     - 4.3.3. Tomar un nodo de la ruta;
     - 4.3.4. **Si** nodo actual es ciclo interno **entonces:**

4.3.4.1. $Tiempo_{nodo} \leftarrow$ Tiempo estimado del ciclo interno;
4.3.4.2. $Ener_{nodo} \leftarrow$ Energía estimada del ciclo interno;
4.3.5. **De lo contrario, Si** el nodo contiene un salto **entonces:**
4.3.5.1. $Tiempo_{nodo} \leftarrow$ Tiempo estimado del nodo;
4.3.5.2. $Ener_{nodo} \leftarrow$ Energía del nodo;
4.3.5.3. $Tiempo_{nodo} \leftarrow Tiempo_{nodo}+$ costo de tiempo de predicción.
4.3.5.4. $Ener_{nodo} \leftarrow Ener_{nodo}+$ costo de energía predicción.
4.3.6. **De lo contrario**
4.3.6.1. $Tiempo_{nodo} \leftarrow$ Tiempo estimado del nodo;
4.3.6.2. $Ener_{nodo} \leftarrow$ Energía estimada del nodo;
4.3.7. $Tiempo_{ruta} \leftarrow Tiempo_{ruta} + Tiempo_{nodo}$;
4.3.8. $Energia_{ruta} \leftarrow Energia_{ruta} + Ener_{nodo}$;
4.4. $Tiempo \leftarrow Tiempo + Tiempo_{ruta} \cdot Probabilidad_{ruta}$;
4.5. $Energia \leftarrow Energia + Energia_{ruta} \cdot Probabilidad_{ruta}$;
4.6. $Potencia \leftarrow Potencia + (Energia_{ruta}/Tiempo_{ruta}) \cdot Probabilidad_{ruta}$;
4.7. Marcar ruta como procesada;
5. $Tiempo_{Ciclo} \leftarrow Tiempo \cdot Iteraciones_{Ciclo}$;
6. $Energia_{Ciclo} \leftarrow Energia \cdot Iteraciones_{Ciclo}$;
7. $Potencia_{Ciclo} \leftarrow Potencia$;
8. Salvar información;
9. **Fin;**

6.3.2. Potencia y Energía del Programa

Una vez se tiene estimado los costos de los ciclos de un programa, se proceder a estimar el valor esperado de tiempo, energía y potencia del programa. La metodología utilzada es similar a la empleada con los ciclos, en el sentido que se determina la probabilidad de cada ruta y se multiplica por el costo de dicha ruta. En caso de encontrar un ciclo, este se considera como un sólo nodo, cuyo costo en tiempo y energía ya han sido determinados en la etapa anterior.

A continuación se presenta el algoritmo para calcular el valor esperado de energía, tiempo de ejecución y potencia de todo el programa. Este programa no debe ser ejecutado hasta que todos los ciclos hayan sido procesados.

Algoritmo 21. *Estimación de Potencia y Energía de un Programa*

Entrada: Resultados del analizador estático, nodos caracterizadas en energía y tiempo de ejecución, costos por cada ciclo.
Salida: Valor esperado de tiempo de ejecución, potencia y energía del programa.

1. $Tiempo \leftarrow 0$;
2. $Energia \leftarrow 0$;
3. $Potencia \leftarrow 0$;
4. **Para** cada ruta del programa **hacer:**
   - 4.1. $Tiempo_{ruta} \leftarrow 0$;
   - 4.2. $Energia_{ruta} \leftarrow 0$;
   - 4.3. **Para** cada nodo de la ruta **hacer:**
     - 4.3.1. $Tiempo_{nodo} \leftarrow 0$;
     - 4.3.2. $Ener_{nodo} \leftarrow 0$;
     - 4.3.3. Tomar un nodo de la ruta;
     - 4.3.4. **Si** nodo actual es ciclo **entonces:**
       - 4.3.4.1. $Tiempo_{nodo} \leftarrow$ Tiempo estimado del ciclo interno;
       - 4.3.4.2. $Ener_{nodo} \leftarrow$ Energía estimada del ciclo interno;
     - 4.3.5. **De lo contrario, Si** el nodo contiene un salto **entonces:**
       - 4.3.5.1. $Tiempo_{nodo} \leftarrow$ Tiempo estimado del nodo;
       - 4.3.5.2. $Ener_{nodo} \leftarrow$ Energía del nodo;
       - 4.3.5.3. $Tiempo_{nodo} \leftarrow Tiempo_{nodo}+$ costo de tiempo de predicción.
       - 4.3.5.4. $Ener_{nodo} \leftarrow Ener_{nodo}+$ costo de energía predicción.
     - 4.3.6. **De lo contrario**
       - 4.3.6.1. $Tiempo_{nodo} \leftarrow$ Tiempo estimado del nodo;
       - 4.3.6.2. $Ener_{nodo} \leftarrow$ Energía estimada del nodo;
     - 4.3.7. $Tiempo_{ruta} \leftarrow Tiempo_{ruta} + Tiempo_{nodo}$;
     - 4.3.8. $Energia_{ruta} \leftarrow Energia_{ruta} + Ener_{nodo}$;
   - 4.4. $Tiempo \leftarrow Tiempo + Tiempo_{ruta} \cdot Probabilidad_{ruta}$;
   - 4.5. $Energia \leftarrow Energia + Energia_{ruta} \cdot Probabilidad_{ruta}$;
   - 4.6. $Potencia \leftarrow Potencia + (Energia_{ruta}/Tiempo_{ruta}) \cdot Probabilidad_{ruta}$;
   - 4.7. Marcar ruta como procesada;
5. $Tiempo_{programa} \leftarrow Tiempo$;
6. $Energia_{programa} \leftarrow Energia$;
7. $Potencia_{programa} \leftarrow Potencia$;
8. Salvar información;
9. **Fin;**

6.3.3. Resumen

Se ha presentado una metodología para estimar la potencia y energía de un programa, donde el valor determinado corresponde al valor esperado del consumo de potencia y energía promedio del microprocesador por ejecutar el programa. La metodología determina primero los costos de los nodos individuales y luego combina estos resultados para cada ruta del programa. En caso que el programa tenga ciclos,

estos deben ser procesados primero para que al momento de estimar el costo de la ruta que contiene un ciclo, todos los valores requeridos ya estén calculados. Finalmente se procesan las rutas generales del programa de donde se obtiene el costo deseado de potencia y energía.

Capítulo 7
RESULTADOS Y ANÁLISIS

Este capítulo presenta los resultados obtenidos para el sistema de estimación de potencia y energía desarrollado. La sección 7.1 comenta sobre los valores de corriente promedio asignados a cada instrucción del microprocesador. La sección 7.2 presenta la hipótesis que se busca validar en el experimento desarrollado. La sección 7.3 describe el sistema *hardware* utilizado para obtener las medidas de potencia y energía de programas de prueba. La sección 7.4 describe los programas elegidos para realizar las pruebas. La sección 7.5 presenta las medidas obtenidas y la comparación con las medidas generadas por el estimador desarrollado en este proyecto. La sección 7.6 presenta la prueba de la hipótesis planteada en la sección 7.2. La sección 7.7 presenta las limitaciones del proyecto.

7.1. Perfíl de Potencia de las Instrucciones

Debido a la extensión de los datos obtenidos, los valores de corriente asignados para cada instrucción se listan en el Apéndice A y B. La lista completa se encuentra junto con la copia electrónica de este documento.

7.2. Hipótesis

El análisis de trabajos previos presentado en el Capítulo 2 muestra dos tipos de estratégias. El primer tipo está relacionado con la estimación de potencia y energía en microprocesadores utilizando diferentes modelos energéticos. Los resultados presentan un rango de error que va desde el 3 % hasta el 20 %, con una tendencia hacia

el 10 %. Este es el caso de los trabajos presentados por Russell, Brandolese, Mehta, Laopoulos y Chakrabarti [13] [3] [5] [16] [6]. Estos trabajos se caracterizan por presentar técnicas dinámicas, como ejecución en tiempo real o simulación del programa, para obtener la información relevante al modelo. El segundo tipo de trabajo está relacionado con el uso de análisis estático para estimar el comportamiento del programa. Algunos de estos trabajos presentar un error promedio entre el 18 % y 25 % para las predicciones realizadas, como es el caso de las heurísticas de predicción de salto presentadas por Ball y Larus [20] [21]. En adición, Ball comenta en su trabajo que las técnicas de análisis estático generalmente presentan un factor de dos en el error respecto a los valores estimados con técnicas dinámicas [20].

En base a esta información, se puede puede presentar una hipótesis sobre la efectividad esperada de la herramienta deasarrollada en este proyecto. Dado que se utilizan técnicas estáticas para estimar el consumo de potencia y energía y considerando la proporción de error entre este tipo de técnicas al compararlas con técnicas dinámicas y la tendencia hacia el 10 % de error de las técnicas dinámicas, se plantea la siguiente hipótesis: La herramienta desarrollada en este proyecto estima el consumo de potencia y energía de un microprocesador al ejecutar un programa dado con un error promedio menor al 20 % respecto a las medidas reales de consumo de potencia y energía.

Esta hipótesis se considera con un intérvalo de confianza del 95 %, ($\alpha = 0{,}05$). La hipótesis nula y la alternativa se establecen de la siguiente forma:

$$H_0\text{: } Error = 20\,\%$$

$$H_1\text{: } Error < 20\,\%$$

Las secciones siguientes describen el proceso realizado para verficar la hipótesis planteada.

7.3. Sistema Experimental

Para validar los resultados obtenidos por la herramienta desarrollada, se realizaron medidas de consumo de potencia y energía en una selección de programas de aplicación típica en sistemas embebidos. Para tal fin, Se implementó un sistema de medida y se diseñó un experimento en base a los programas seleccionados para obtener información confiable de los datos adquiridos. A continuación se describe la organización del sistema de medida.

Para medir la corriente promedio se empleó una resistencia de 1Ω con tolerancia 0.5 %, la cuál fué conectada en serie con la fuente de poder que alimenta al microprocesador. El sistema de fuente de poder de la tarjeta Excimer consta de una fuente de 5V en cascada con un regulador ajustado a 2.5V exclusivo para el microprocesador. Debido a los valores de corriente esperados, se decidió colocar el resistor de prueba entre la fuente de 5V y el regulador de 2.5V. Para compensar el error introducido por el consumo extra de corriente del regulador de 2.5 voltios, se determinó el consumo de corriente del regulador y se restó del valor medido en el resistor de prueba. Ver Figura 7–1.

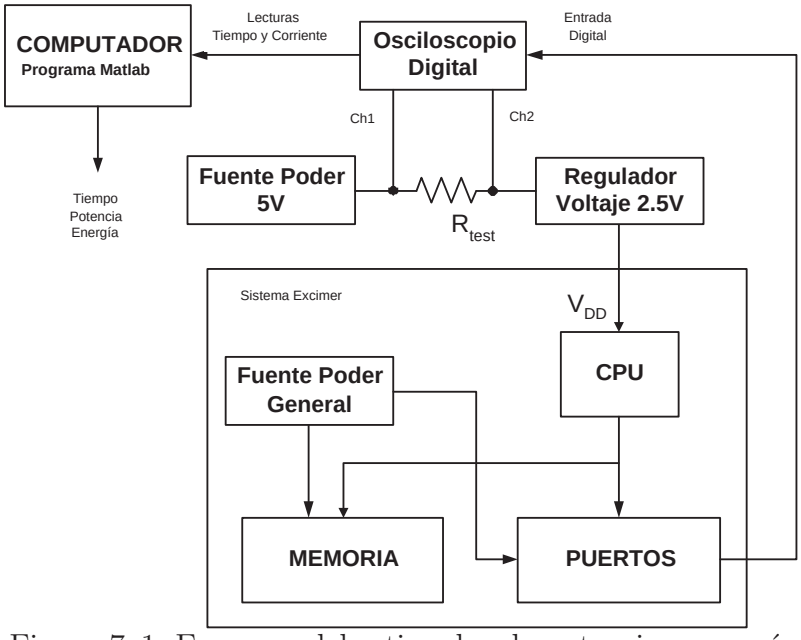

Figura 7–1: Esquema del estimador de potencia y energía

El voltaje de la resistencia fué medido por un osciloscopio digital de 100 MHz, el cuál determinaba el voltaje restando la lectura de los dos canales de entrada, los cuáles estaban conectados a los extremos de la resistencia de prueba. Una entrada digital del osciloscopio fué utilizada para sincronizar el inicio de las lecturas. La señal de sincronía es necesaria para iniciar la adquisición de los datos debido a que se requiere ejecutar otros programas en el sistema microprocesador antes de iniciar la ejecución del programa a medir. Esta señal de sincronía proviene de un puerto de la tarjeta que contiene al microprocesador y permanece activa solo durante la ejecución del programa bajo prueba. Se desarrolló un programa en Matlab para procesar las señales del osciloscopio, el cuál elimina las medidas de corriente que no pertenecen a la ejecución del programa y luego calcula la energía consumida integrando la señal de corriente en el intervalo de tiempo de ejecución del programa. El voltaje de polarización se supone constante a 2.5V.

El osciloscopio fué configurado para disparo en modo normal no automático y adquisición simple. Esto significa que el osciloscopio no inicia lecturas válidas hasta que recibe la señal de disparo indicada y una vez inicia la lectura, solo adquiere datos hasta que llena la memoria, ignorando si se genera otra señal de disparo válida. Esto permite con una sola ejecución o una cantidad fija de ejecuciones del programa se pueda obtener simultaneamente la información de corriente y de tiempo consumido por el microprocesador al ajecutar el programa.

7.4. Programas de Prueba

En la elección de los programas de prueba se consideró que estos fuesen representativos de aplicaciones actuales de microprocesadores empotrados. Así, se consideró programas para el procesamiento de señales, programas de control y programas de cálculo matemático. Para realizar las pruebas se seleccionaron los siguientes programas:

- FFT.
- Filtro FIR.
- Raíz cuadrada entera.
- Controlador de una fuente de poder basado en máquina de estados finitos.
- Control de un cargador de batería.

7.4.1. FFT

Este programa es una versión en lenguaje C de la transformada rápida de Fourier (FFT radix 2, decimation in time). Requiere como datos de entrada el tamaño de la FFT, un apuntador a los coeficientes de la FFT y un apuntador a la señal a procesar. La restricción está en el tamaño de la FFT, la cual debe ser potencia de 2. La distribución de la señal de entrada depende de la aplicación que utilice la FFT, por lo tanto, se decidió utilizar una distribución uniforme para generar los datos aleatorios de entrada. De acuerdo a ejecuciones preliminares, se espera que se recorran todas las rutas del programa en cada ejecución del mismo. También se espera un comportamiento normal del programa. Esto quiere decir que para un tamaño definido de FFT, se espera que el consumo de potencia y energía tengan valores similares, aunque se tengan diferentes valores de señales de entrada.

7.4.2. FIR

Este programa es una versión en lenguaje C de un flitro FIR. Los datos de entrada más importantes que requiere el programa son los coeficientes del filtro y la señal a filtrar. Ambos son datos tipo entero. Esta información se pasa al programa como dos variables que indican los tamaños y dos apuntadores, que indican donde se encuentra almacenada la información. Los valores de los coeficientes del filtro dependen del tipo de filtro a implementar. Los valores de la señal dependen de la aplicación que utilice el filtro. Para esta prueba, se asignaron datos aleatorios

con una distribución normal para los coeficientes del filtro y datos aleatorios con distribución uniforme para la señal de entrada. Basado en ejecuciones preliminares, se espera que el programa recorra todas las rutas en una sola ejecución del mismo. También se espera un comportamiento normal del programa. Esto quiere decir que para un tamaño definido de filtro y señal, se espera que el consumo de potencia y energía tengan valores similares, aunque cambien los valores de la señal de entrada.

7.4.3. Raíz cuadrada entera

Este programa determina la raíz cuadrada de un valor entero. El resultado se presenta como dos valores, la raíz y el residuo. La información dada al programa es un número entero, al cual se le obtendrá la raíz. Se le asignó una distribución uniforme al dato de entrada. Basado en ejecuciones preliminares, se espera que el programa tenga un comportamiento normal respecto al dato de entrada. También se estima que se recorrerán todas las rutas del programa en cada ejecución del programa.

7.4.4. Controlador de una fuente de poder

Este programa es el control principal de una fuente de poder. Está basado en una máquina de estados finitos, donde cada estado describe los diferentes comportamientos del control. Los datos de entrada para este programa son el voltaje de entrada, voltaje de disparo, temperatura de los elementos de potencia y corriente de salida. Basado en la información obtenida, se espera que estos datos de entrada tengan una distribución normal, donde la mayoría de los valores deben estar entre los límites designados por el controlador. Un exámen del programa revela que el comportamiento del programa depende del estado actual y los valores de la variables de entrada. Se espera que cuando la fuente esté bajo funcionamiento regular, el control

esté en un estado llamado *normal*. Sólo en casos de falla o apagado el control cambiará de estado. Para ejercitar todos los estados, los datos de entrada deben tener suficiente variabilidad. La sección 7.5 explica cómo se determinaron estos valores.

7.4.5. Control de un cargador de batería

Este programa es el control principal de un cargador de baterías. Este cargador está conectado durante un largo tiempo a la batería para asegurar una recarga completa. Se puede considerar como un control realimentado donde el voltaje actual de la batería se utiliza para determinar la señal de recarga. La entrada a este programa son dos señales de voltaje, el voltaje de la batería y un voltaje de referencia. Basado en la descripción del programa, se espera que ambos voltajes varíen en un rango de valores prederminado. Para la prueba de este programa, se eligieron datos con una distribución normal, donde los datos deben exceder los rangos preestablecidos por el programa, de tal forma que se puedan ejecutar todas las rutas de programa. La sección 7.5 explica cómo se determinaron estos valores.

7.5. Resultados y Análisis

A continuación se presenta los resultados obtenidos para cada programa y los valores estimados por el programa desarrollado en este proyecto.

7.5.1. FFT

El consumo de potencia y energía de este programa esta directamente relacionado con la longitud de la FFT utilizada, la cuál es una potencia de dos. Debido a las limitantes del sistema, se eligieron valores específicos de tamaños de FFT. En este caso el estimador requiere como dato auxiliar para realizar el estimado el tamaño de la FFT y dos variables auxiliares. La Tabla 7–1 presenta la comparación de las medidas obtenidas con los valores estimados para el programa FFT.

Tabla 7–1: Resultados para el programa FFT.

Medida	Cantidad	Valores Medidos	Valor Estimado	Error (%)
FFT 128	Potencia (W)	2.294580777	2.150585144	6.27
	Energía (J)	4.49E-04	0.000499873	11.37
FFT 256	Potencia (W)	2.266288952	2.152005911	5.04
	Energía (J)	0.001	0.001134699	13.47
FFT 512	Potencia (W)	2.337398374	2.153044286	7.89
	Energía (J)	0.0023	0.002537471	10.32
FFT 1024	Potencia (W)	2.34375	2.153839749	8.10
	Energía (J)	0.0051	0.005608963	9.98
FFT 2048	Potencia (W)	2.311365807	2.154472945	6.79
	Energía (J)	0.0121	0.012283549	1.52

Como se puede observar, el error en el estimado del consumo de energía se mantiene en un valor aceptable. Al igual que en el caso del filtro FIR, se considera como principal fuente de error el tiempo asignado a cada iteración de los diferentes ciclos del programa.

7.5.2. Filtro FIR

Este programa tiene 2 elementos de los cuales depende su comportamiento. La cantidad de coeficientes del filtro y el tamaño de la señal a filtrar. Para la prueba se eligieron dos filtros de 20 y 35 puntos. Los valores fueron elegidos de distribuciones normales con media 0 y 50. Los valores se limitaron a un rango entre -127 y 128, para conservar las indicaciones dadas en el programa de ejemplo. También se eligieron dos señales de entrada de 1000 y 2000 puntos, a partir de distribuciones uniformes en el rango -127 a 127. El estimador requiere como dato adicional los tamaños del filtro y la señal para poder realizar el estimado de consumo de potencia y energía. La Tabla 7–2 presenta la comparación de las medidas obtenidas con los valores estimados para el programa filtro FIR.

Como se puede observar de los resultados, el error oscila entre el 10 % y 15 %, con una tendencia al 10 %. Se considera como principal fuente de error el valor promedio

Tabla 7–2: Resultados para el programa filtro FIR.

Medida	Cantidad	Valores Medidos	Valor Estimado	Error (%)
FIR 20 pts	Potencia (W)	2.278820375	2.040793247	10.44
Señal 1000 pts	Energía (J)	0.0034	0.002873582	15.48
FIR 20 pts	Potencia (W)	2.27996647	2.040796406	10.49
Señal 2000 pts	Energía (J)	0.0068	0.005746882	15.49
FIR 35 pts	Potencia (W)	2.31092437	2.044416978	11.53
Señal 1000 pts	Energía (J)	0.0055	0.004866479	11.52
FIR 35 pts	Potencia (W)	2.306079665	2.044418955	11.35
Señal 2000 pts	Energía (J)	0.011	0.009732675	11.52

de tiempo que asigna el estimador a las iteración de los ciclos del programa bajo prueba.

7.5.3. Programa Raíz Cuadrada

Este programa tiene como única entrada el valor del cual se obtiene la raíz. Para esta variable, se utilizó un generador de números aleatorios enteros con distribución uniforme con límites 0 y $2^{32} - 1$. El programa fué ejecutado 100 veces con datos aleatorios de entrada. En el caso del estimador, este no requiere información adicional al código para determinar el consumo de potencia y energía. La Tabla 7–3 presenta los resultados obtenidos para este programa y los compara con los obtenidos por el estimador desarrollado.

Tabla 7–3: Resultados para el programa raíz cuadrada.

	Prom. Valores Medidos	Valor Estimado	Error (%)
Potencia (W)	2.23	1.948877644	12.64
Energía (J)	7.2054E-06	5.82E-06	19.26

Como se puede observar de los resultados, hay una diferencia considerable en el valor de energía calculada por el estimador, respecto al valor real de energía. Se considera que la diferencia se debe a los valores de probabilidad asignados a una decisión dentro del ciclo principal, los cuáles no corresponden con la frecuencia que ocurre en el programa real, lo que afecta el valor estimado del tiempo de ejecución

del programa y por lo tanto, la energía consumida. A pesar de esto, el valor de la potencia tiene un error menor al 13 %. Esto se atribuye a que el consumo de corriente de las diferentes instrucciones del programa tienen valores similares, con lo cuál el consumo de potencia tiende a ser similar al obtenido en la realidad.

7.5.4. Programa Controlador Fuente de Poder

Este programa tiene cuatro variables de entrada: temperatura de la etapa de potencia, corriente de saida, voltaje de disparo y voltaje de entrada. El comportamiento de la máquina de estados se define en función de la comparación de estas variables con unos parámetros internos de funcionamiento de la fuente de poder. Así, a partir de los parámetros, se designaron distribuciones normales para cada variable de entrada, donde el valor medio estuviese dentro del rango indicado por los parámetros. Se designó que la desviación estándar alcanzara el valor límite de los parámetros, de modo que se tenga la mayoría de los datos aleatorios dentro de los rangos prestablecidos (66 % de los datos) y el resto de datos fuera del rango. Con esto se espera que se puedan ejecutar todas las rutas del programa.

Ejemplo 6. *El rango de temperatura para los elementos de potencia se definió entre 40 y 50 grados. Así, la variable de entrada temperatura se eligío como una distribución normal con media igual a 45 grados y una desviación estándar de 5 grados. En base a esto, se espera que 66 % de los datos elegidos aleatoriamente estén entre 40 y 50 grados.*

Es importante recalcar que dependiendo del estado que se encuentre el programa, algunas variables tienen prioridad sobre otras para decidir el comportamiento del programa. Este programa fué ejecutado 1000 veces con el fin de obtener una medida representativa del valor esperado de energía y potencia. La Tabla 7–4 presenta la comparación de las medidas obtenidas con los valores estimados para el programa controlador de fuente de poder.

Tabla 7–4: Resultados para el programa controlador de fuente de poder (FSM).

	Prom. Valores Medidos	Valor Estimado	Error (%)
Potencia (W)	2.12	2.046399255	3.4
Energía (J)	8.2663E-07	7.36E-07	10.91

Como se puede observar, el error en potencia y energía disminuyó respecto al programa anterior. Se tiene que la principal fuente de error es la estimación del tiempo de ejecución del programa, pues el valor medido es de 390.2 nS mientras que el valor estimado es de 359.3 nS. Se considera que la principal diferencia entre los valores estimados de tiempo radica en la estimación de tiempo por nodos separados. Es posible que exista un consumo de tiempo adicional entre los diferentes nodos del programa que en este momento no se tiene en cuenta por la herramienta desarrollada.

7.5.5. Control de un cargador de batería

Este programa requiere como datos de entrada 2 voltajes. Uno es el voltaje de la batería y el otro es un voltaje de referencia. Dado que el cargador tiene límites de voltaje preestablecidos para su operación (12.5V y 13.8V), se eligió para el voltaje de la batería una distribución de datos normal, con media de 13.15V y una desviación estándar de 0.65V. Con esto se tiene que el 66 % de los datos generados se encuentran en el rango de carga y el resto de valores se encuentra fuera de los límites impuestos por el controlador. De acuerdo a la información del programa, se espera que el voltaje de referencia tenga una variación similar al voltaje de la batería, pero con un valor promedio de 1.2V. Con esta información se generó la cantidad suficiente de datos para ejecutar 100 veces el programa, de modo que se tuviese un valor medio aceptable. En este caso, el estimador no requiere información adicional para generar el estimado de potencia y energía del programa. La Tabla 7–5 presenta la comparación de las medidas obtenidas con los valores estimados para el programa de control.

Tabla 7–5: Resultados para el programa de control de cargador de batería.

	Prom. Valores Medidos	Valor Estimado	Error (%)
Potencia (W)	2.120141343	2.103036706	0.81
Energía (J)	0.0012	0.000960964	19.92

Este programa presenta un error muy grande en la estimación de energía, el cuál contrasta con el error mínimo en el estimado del consumo de potencia del programa. La principal fuente de error está en la estimación del tiempo de ejecución del programa, pues la medida real es de $566\mu S$ mientras que el estimador calcula un valor de $456{,}94\mu S$. Se considera que la diferencia en tiempo se debe a la gran cantidad de subrutinas que tiene el programa, lo cual dificulta la estimación del tiempo de ejecución del programa.

7.6. Prueba de Hipótesis

En la sección 7.2 se presentó la siguiente hipótesis sobre la efectividad del estimador de potencia y energía, considerando un intérvalo de confianza del 95 %, ($\alpha = 0{,}05$).

$$H_0\text{: } Error = 20\,\%$$

$$H_1\text{: } Error < 20\,\%$$

Para probar esta hipótesis, se tienen 5 programas de muestra, de donde se obtiene valores promedio para el error. Debido a la cantidad reducida de muestras, (n= 5), y una varianza desconocida, se utiliza una prueba *t-statistic* para verificar la hipótesis. A continuación se presentan los resultados obtenidos por el programa *Minitab* para los errores en el estimado de la energía.

```
Test of mu = 20 vs < 20
                                                    95%
                                                  Upper
Variable  N     Mean    StDev  SE Mean     Bound      T      P
energia   5  14.5853   4.8120   2.1520   19.1730  -2.52  0.033
```

Como se puede observar, el *p-value* es menor que el valor dado de α, con lo cuál se rechaza la hipótesis nula (Ho). Sin embargo, se observa que el error está muy cerca del límite establecido. Un estudio con más programas puede proveer un mejor punto de vista sobre la efectividad de la herramienta presentada.

Para la potencia se tienen los siguientes resultados:

```
Test of mu = 20 vs < 20
                                                  95%
                                                Upper
Variable  N     Mean    StDev  SE Mean     Bound      T      P
potencia  5  6.92125  4.96974  2.22253  11.65936  -5.88  0.002
```

Como se puede observar, el resultado del *p-value* es menor que el α seleccionado, con lo cuál se puede rechazar la hipótesis nula. De hecho, se observa que el error límite superior es de 12 %, lo cuál indica que la efectividad de la herramienta para estimar consumo de potencia puede ser similar a la efectividad que presentan las herramientas dinámicas de estimación de potencia.

7.7. Limitaciones del Proyecto

El programa presentado en este proyecto es un prototipo inicial el cuál debe ser refinado en todas sus etapas antes de poder ser utilizado en otras aplicaciones.

Existen estructuras del lenguaje de alto nivel que no pueden ser reconocidas por el programa. Esto se debe a la forma como se implementaron algunos de los componentes del analizador estático de código.

El tamaño de los datos y el programa bajo prueba no deben exceder el tamaño de la cache.

La estimación de iteraciones de un ciclo está restringida a ciclos donde el límite superior sea una variable y el valor inicial del contador no cambie durante diferentes ejecuciones del programa. En otras palabras, la herramienta desarrollada asigna un

valor fijo de iteraciones a cada ciclo del programa, aunque en realidad este pueda realizar un número variable de iteraciones.

Debido al comportamiento totalmente dinámico de la memoria cache, la herramienta desarrollada solo puede realizar estimados en un entorno de aciertos en la cache. Esto puede ser aceptable para aplicaciones embebidas de bajo y mediano nivel, donde normalmente se ejecutan uno o pocos programas.

Este trabajo no incluye el efecto de *interrupciones* en la estimación de potencia y energía debido a su comportamiento totalmente dinámico.

Un factor de error en la estimación de las iteraciones de un ciclo son los valores de las variables límite, los cuales son asignados por el usuario.

De acuerdo a la información del manual de ususario del microprocesador PowerPC 603, existe un sistema automático de manejo de potencia de los diferentes componentes del microprocesador. Sin embargo, no es posible controlar de alguna forma este sistema y por tanto, de determinar su efecto en la ejecución de programas.

La herramienta desarrollada no puede trabajar con programas recursivos, ni programas que estén compuestos por otros programas. El trabajo con librerías está limitado a que el usuario provea las instrucciones de cada función de librería utilizada en su programa. Esto se debe a que la herramienta no forma parte de un compilador, por lo tanto, no se tiene acceso a la mayoría de la información para poder trabajar.

Capítulo 8
CONTRIBUCIONES

Este trabajo presenta una metodología alterna para estimar el consumo de potencia y energía en microprocesadores. Se propone el uso de herramientas de análisis estático de código para determinar el comportamiento del programa, lo cuál, combinado con un modelo de energía del microprocesador y una cantidad mínima de datos adicionales, permite obtener la información necesaria para determinar estimados de consumo de potencia y energía. Las metodologías actuales utilizan simuladores especiales, donde se realiza la ejecución del programa y se obtiene la información requerida, como es el caso de los trabajos publicados por Kavvadias y Laurent [4] [2]. Adicionalmente, estos simuladores requieren de todos los datos reales de la aplicación para poder estimar con precisión los parámetros deseados.

Este proyecto presenta un prototipo de simulador de dos niveles para estimar el tiempo de ejecución de un programa. El primer nivel consiste en un modelo del microprocesador bajo estudio, el cuál caracteriza el comportamiento del programa a nivel de los nodos que lo componen, determinando cuantos ciclos de reloj se requieren para procesar las instrucciones de cada nodo, sin necesidad de utilizar datos reales del programa. El segundo nivel lo proporciona las herramientas de análisis estático, donde ya se tiene caracterizado el programa a nivel de rutas de ejecución, junto con valores de probabilidad y estimados de las iteraciones de cada ciclo del programa. Esta información es combinada con los resultados obtenidos en el primer nivel para realizar un estimado del tiempo total de ejecución de un programa. Los simuladores

actuales de dos niveles realizan simulación detallada de las instrucciones cuando se procesan por primera vez y luego utilizan la información obtenida cuando se identifica que se va a volver a simular la misma secuencia de instrucciones, como es el caso del trabajo presentado por Rapaka y Marculescu [26].

Este trabajo presenta una herramienta para generar equivalencias entre variables de alto y bajo nivel. Esta herramienta construye una gráfica de control de flujo (CFG) a partir del lenguaje de alto nivel, la cuál es comparada con la gráfica previamente generada del lenguaje ensamblador. Al realizar la comparación, se puede determinar qué nodos son correspondientes en ambas gráficas. Una vez se encuentran dichos nodos, se procede a extraer las variables directamente de las instrucciones de alto nivel y bajo nivel. Esta información se almacena en una tabla de asignación. Esta herramienta reconoce estructuras básicas como decisiones y ciclos *FOR*.

Capítulo 9
CONCLUSIONES Y TRABAJO FUTURO

9.1. Conclusiones

Este trabajo presenta una metodología para estimar el consumo de potencia y energía de un microprocesador a partir del análisis de las instrucciones que conforman el programa que ejecuta el microprocesador.

Se utilizaron técnicas de análisis estático de código para predecir el comportamiento del programa, con lo cuál se evita la simulación del programa bajo estudio y adquirir un sistema *hardware* adicional para determinar los estimados de potencia y energía.

Debido a sus características, la herramienta desarrollada en este proyecto puede ser incorporada en otro tipo de programas que requieran realizar estimados preliminares del consumo de potencia y energía de sus prototipos, por ejemplo, los optimizadores de código orientados a la reducción del consumo de potencia y energía. También puede ser utilizado como elemento de apoyo para las decisiones de diseño *hardware-software* de sistemas empotrados.

Al considerar que los valores estimados de potencia y energía son obtenidos sin el uso de un hardware especializado, que no se requiere agregar instrucciones al programa para obtener información extra que ayude al estimador y que se requiere una cantidad mínima de datos, se puede considerar el error promedio obtenido aceptable

para algunas aplicaciones de sistemas empotrados, en especial en las primeras etapas de implementación de los algoritmos de la aplicación en desarrollo.

Las técnicas de análisis estático de código han demostrado tener potencial para la estimación del consumo de potencia y energía de microprocesadores. Sin embargo, en los casos donde el funcionamiento del programa tiene una alta dependencia de los datos de entrada, las técnicas estáticas tienen a fallar. Un punto intermedio entre las técnicas estáticas y el análisis dinámico del programa puede puede conducir a mejores estimados de consumo de potencia y energía.

9.2. Trabajo Futuro

Para que esta herrramienta pueda ser utilizada en otras aplicaciones, se requiere la evolución de sus componentes, de modo que se pueda superar las limitaciones actuales. También es necesario considerar la expansión de la herramienta para que pueda cubrir aspectos adicionales de sistemas empotrados, como la memoria y sistemas periféricos. Con base en esto, se presenta la siguiente lista de actividades futuras para este proyecto.

- Es necesario combinar esta herramienta con un compilador, de modo que sea posible realizar una mejor detección de todas las estruturas, parámetros y variables del programa bajo prueba. Esto permitiría analizar programas de mayor complejidad, por ejemplo, programas orientados a objetos.
- Refinar y aumentar la cantidad de heurísticas de predicción de las instrucciones de salto, para mejorar el estimado del comportamiento del programa bajo prueba.
- Mejorar la técnica para estimar las iteraciones de los ciclos de un programa. En especial cuando el ciclo tiene una cantidad variable de iteraciones.
- Realizar un estudio que determine estrategias para incluír el efecto de los desaciertos en la memoria cache en un entorno no dinámico.

- Realizar estudios para incluir l efecto de *interrupciones* en la estimación de potencia y energía.
- Experimentar con otros modelos de potencia y energía para microprocesadores e iniciar estudios en otros microprocesadores para generar una librería de componentes. Esto permitiría expandir las posibles aplicaciones del proyecto presentado.
- Expandir la estimación de potencia y energía a otros componentes de sistemas embebidos, en especial el sistema de memoria.
- Combinar esta herramienta con técnicas de optimización de código a nivel de potencia y energía, de modo que se puedan obtener una forma de medir la eficiencia de las diferentes técnicas de optimización y se pueda seleccionar la más adecuada para una aplicación específica.

Bibliografía

[1] M. Caldari, M. Conti, P. Crippa, G.Ñuzzo, S. Orcioni, and C. Turchetti. Instruction based power consumption estimation methodology. *9th International Conference on Electronics, Circuits and Systems*, Vol. 2:721 – 724, 2002.

[2] J. Laurent, N. Julien, E. Senn, and E. Martin. Functional level power analysis: An efficient approach for modeling the power consumption of complex processors. *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, Vol. 1:666 – 667, 2004.

[3] C. Brandolese, F. Salice, W. Fornaciari, and D. Sciuto. Static power modeling of 32-bit microprocessors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 21:1306 – 1316, 2002.

[4] N. Kavvadias, P.Ñeofotistos, S.Ñikolaidis, C. Kosmatopoulos, and T. Laopoulos. Measurements analysis of the software-related power consumption in microprocessors. *IEEE Transactions on Instrumentation and Measurement*, 53:1106 – 1112, 2004.

[5] H. Mehta, R. Owens, and M. Irwin. Instruction level power profiling. *IEEE Conference Proceedings International Conference on Acoustics, Speech, and Signal Processing, ICASSP-96.*, Vol. 6:3326 – 3329, 1996.

[6] C. Chakrabarti and D. Gaitonde. Instruction level power model of microcontrollers. *Proceedings IEEE International Symposium on Circuits and Systems, ISCAS '99.*, Vol. 1:76 – 79, 1999.

[7] S.S. Abrar. Cycle-accurate energy model and source-independent characterization methodology for embedded processors. *Proceedings 17th International Conference on VLSI Design*, pages 749 – 752, 2004.

[8] Pradeep Chakraborty. Power verification. http://www.edn.com/article/CA609105.html?industryid=23439, June 2005.

[9] Y. Fei, S. Ravi, A. Raghunathan, and N.K. Jha. A hybrid energy-estimation technique for extensible processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23:652 – 664, May 2004.

[10] K.Ñatarajan, H. Hanson, S.W. Keckler, C.R. Moore, and D. Burger. Microprocessor pipeline energy analysis. *Proceedings of the 2003 International Symposium on Low Power Electronics and Design. ISLPED '03.*, pages 282 – 287, Aug. 2003.

[11] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 2:437 – 445, 1994.

[12] V. Tiwari and M. Tien-Chien. Power analysis of a 32-bit embedded microcontroller. *Proceedings of the Design Automation Conference, ASP-DAC '95.*, pages 141 – 148, 1995.

[13] J. Russell and M. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. *Proceedings International Conference on Computer Design: VLSI in Computers and Processors, ICCD '98.*, pages 328 – 333, 1998.

[14] Levy Markus. Processor measure up to the power challenge. EDN Magazine, July 1999.

[15] S. Dongkun, S. Hojun, J. Yongsoo, Y. Han-Saem, K. Jihong, and C.Ñaehyuck. Energy-monitoring tool for low-power embedded programs. *IEEE Design and Test of Computers*, 19:7 – 17, 2002.

[16] T. Laopoulos, P.Ñeofotistos, C.A. Kosmatopoulos, and S.Ñikolaidis. Measurement of current variations for the estimation of software-related power consumption. *IEEE Transactions on Instrumentation and Measurement*, 52:1206

– 1212, 2003.

[17] C. Talarico, J.W. Rozenblit, V. Malhotra, and A. Stritter. A new framework for power estimation of embedded systems. *Computer. IEEE Computer Society*, vol. 38:71 – 78, Feb. 2005.

[18] J. Haid, G. Kaefer, Ch. Steger, and R. Weiss. Run-time energy estimation in system-on-a-chip designs. *Proceedings of the ASP-DAC 2003 Design Automation Conference, 2003*, pages 595 – 599, Jan. 2003.

[19] A. Aho, R. Sethi, and J. Ullman. Compiladores. principios, tecnicas y herramientas. 1998.

[20] T. Ball and J. Larus. Branch prediction for free. *SIGPLAN Conference on Programming Language Design and Implementation*, 1993.

[21] J. Larus and Y. Wu. Static branch frequency and program profile analysis. *27th IEEE/ACM Inter'l Symposium on Microarchitecture (MICRO-27)*, 1994.

[22] W. Wong. Source level static branch prediction. *The Computer Journal*, Vol. 42:142 – 149, 1999.

[23] R.E. Hank, S.A. Mahlke, R.A. Bringmann, J.C. Gyllenhaal, and W.W. Hwu. Superblock formation using static program analysis. *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 247 – 255, 1993.

[24] M. De Alba and D. Kaeli. Path-based hardware loop prediction. *4th International Conference on Control, Virtual Instrumentation and Digital Systems*, pages 29–38, 2002.

[25] C. Healy, M. Sjodin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. *Proceedings Real-Time Technology and Applications Symposium*, pages 12 –21, 1998.

[26] V. Rapaka and D. Marculescu. Pre-characterization free, efficient power-performance analysis of embedded and general purpose software applications. *Design, Automation and Test in Europe Conference and Exhibition*, Vol. 1:504

– 509, 2003.

[27] G. Beltrame, C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto, and V. Trianni. An assembly-level execution-time model for pipelined architectures. *IEEE/ACM International Conference on Computer Aided Design ICCAD 2001*, Vol. 1:195 – 200, 2001.

[28] Motorola. Programming environments manual for 32-bit implementations of the powerpc architecture. Motorola Literature Distribution, 2001.

[29] Vera Sit. Analyzing anova designs. B.C. Ministry of Forests. Research Branch, 1995.

[30] Motorola. Excimer users manual. Motorola RISC Applications, 1999.

[31] Gary Milliorn. Designing a minimal powerpc system. Motorola RISC Applications, 1999.

[32] C. Corley, J. Quinones, N. Serrano, W. Guiot, L.Ñarvaez, and E. Montalvo. Excimer laboratory manual. Motorola RISC Applications, 1999.

[33] David Medinets. Perl 5 by example. http://www.webbasedprogramming.com/Perl-5-By-Example/, 1996.

[34] Deitel P.J. and Deitel H.M. C++ how to program. Prentice Hall, 1994.

[35] Motorola. Mpc603e risc microprocessor users manual. Motorola Literature Distribution, 2002.

[36] Top Changwatchai, Skipper Smith, and Nasr Ullah. Optimizing instruction execution in the powerpc 603e superscalar microprocessor. Presentation of Motorola RISC Microprocessor Division, 1998.

APENDICES

Apéndice A
VALORES DE CORRIENTE ASIGNADOS A LAS INSTRUCCIONES

Tabla A–1: Medidas de corriente base para las instrucciones enteras del PPC603e.

Instruccin	**Corriente** (mA)	**Instruccin**	**Corriente** (mA)
add	788.756515909091	add.	729.183
addc.	735.93555	addco.	735.94395
addco	737.410275	addc	727.861575
adde.	732.74355	addeo.	730.4619
addeo	740.89155	adde	726.239325
addme.	728.429100000001	addmeo.	731.348625000001
addmeo	729.525825	addme	729.034425
addo.	739.082925	addo	788.840849999999
addze.	719.691524999999	addzeo.	719.6427
addzeo	730.052399999999	addze	721.858725
and.	723.20325	andc.	730.39155
andc	737.925825000001	and	734.030325000001
cntlzw.	716.92425	cntlzw	722.224125
divw.	719.694675	divwo.	721.614075
divwo	717.405675	divw	715.362375
divwu.	717.820425	divwuo.	717.949575
divwuo	714.753375000001	divwu	707.435400000001
eqv.	740.624850000001	eqv	733.8135
extsb.	725.524799999999	extsb	732.5241
extsh.	725.14785	extsh	731.46465
mulhw.	794.301375000001	mulhw	782.856375000001
mulhwu.	778.79445	mulhwu	778.200675
mullw.	794.228925	mullwo.	792.000825
mullwo	784.790475	mullw	792.085875
nand.	733.016025	nand	725.72955
neg.	708.27855	nego.	726.405750000001

Tabla A–2: Medidas de corriente base para las instrucciones enteras del PPC603e.(Cont.)

Instruccin	**Corriente** (mA)	**Instruccin**	**Corriente** (mA)
nego	722.548575	neg	718.196325
nor.	736.49835	nor	728.210174999999
or.	740.520374999999	orc.	727.646325000001
orc	733.59615	or	733.714274999999
subf.	732.601275	subfc.	718.518675
subfco.	738.12165	subfco	728.56875
subfc	730.743825000001	subfe.	729.083775
subfeo.	725.290125	subfeo	730.553774999999
subfe	729.16935	subfme.	718.296075
subfmeo.	722.246175	subfmeo	729.6639
subfme	724.7709	subfo.	737.630775
subfo	736.118774999999	subf	735.2121
subfze.	719.842724999999	subfzeo.	723.41535
subfzeo	718.820025	subfze	722.981174999999
xor.	722.449875	xor	739.7796
addi	785.0169075	addic.	736.2216855
addic	734.9448225	addis	787.104645000001
andi.	731.847564	andis.	734.172222
mulli	815.824737	ori	736.960157142857
oris	739.283664	subfic	734.051989285714
xori	741.281646	xoris	739.66473
rlwimi	736.934520000001	rlwimi.	727.898661000001
rlwinm	736.391838	rlwinm.	736.084986
rlwnm	730.223536411764	rlwnm.	727.571038352941
slw	730.324429058823	slw.	723.119475235294
sraw	733.36606782353	sraw.	730.31748382353
srawi	741.617132117648	srawi.	739.79340482353
srw	731.702746352941	srw.	726.006738470589
tw	735.004977	twi	735.004977

Tabla A–3: Medidas de corriente base para las instrucciones punto flotante del PPC603e.

Instruccin	**Corriente** (mA)	**Instruccin**	**Corriente** (mA)
fabs	775.729047619048	fabs.	778.41
fadd	838.768095238095	fadd.	835.76130952381
fadds	809.413333333333	fadds.	815.20619047619
fcmpo	794.899523809524	fcmpu	800.981904761905
fctiw	830.02619047619	fctiw.	839.343809523809
fctiwz	836.447142857143	fctiwz.	831.758571428571
fdiv	767.114285714286	fdiv.	770.610476190476
fdivs	757.315714285714	fdivs.	764.854285714285
fmadd	950.267619047619	fmadd.	997.58
fmadds	786.218930041152	fmadds.	789.720833333333
fmr	778.357619047619	fmr.	779.161904761905
fmsub	958.065714285714	fmsub.	984.217142857143
fmsubs	790.809523809524	fmsubs.	790.490564373898
fmul	947.962857142857	fmul.	951.480476190476
fmuls	772.729047619048	fmuls.	773.327619047619
fnabs	774.233333333333	fnabs.	781.715238095238
fneg	775.900476190476	fneg.	781.740952380952
fnmadd	976.741904761904	fnmadd.	995.411904761905
fnmadds	789.548148148148	fnmadds.	790.952380952381
fnmsub	973.53	fnmsub.	987.485714285714
fnmsubs	790.249047619048	fnmsubs.	793.680952380952
fres	751.768095238095	fres.	751.820105820106
frsp	804.653333333333	frsp.	800.46
frsqrte	796.487142857143	frsqrte.	807.460952380952
fsel	759.131904761905	fsel.	767.771428571429
fsub	830.097619047619	fsub.	835.083809523809
fsubs	808.248095238095	fsubs.	806.046190476191
mcrfs	668.918571428571	mffs.	672.863265306122
mffs	671.333333333333	mtfsb0	662.63492063492
mtfsb0.	666.113690476191	mtfsb1	662.820634920635
mtfsb1.	668.223214285714	mtfsf	702.583333333333
mtfsf.	701.725238095238	mtfsfi	675.425238095238
mtfsfi.	675.916402116402		

Tabla A–4: Medidas de corriente base para las instrucciones load-store del PPC603e.

Instruccin	**Corriente** (mA)	**Instruccin**	**Corriente** (mA)
lbz	849.210764285715	lbzu	833.007921428572
lbzux	867.841955357143	lbzx	844.22064047619
lfd	863.995	lfdu	852.041954761904
lfdux	883.651895238095	lfdx	867.874935714285
lfs	854.887	lfsu	845.639783068783
lfsux	879.597335978836	lfsx	859.715083333334
lha	846.54824047619	lhau	838.402242857143
lhaux	865.331080952381	lhax	855.63575952381
lhbrx	848.773435714285	lhz	847.477714285715
lhzu	831.525461904762	lhzux	868.568276190477
lhzx	844.621223809524	lwbrx	847.764447619048
lwz	846.585588095238	lwzu	833.41995
lwzux	872.338578571429	lwzx	851.449814285715
stb	843.735121428572	stbu	827.687492063492
stbux	857.8724	stbx	843.349597619048
stfd	837.43	stfdu	833.097073809524
stfdux	865.903041666666	stfdx	850.845626190476
stfiwx	853.459357142857	stfs	839.96
stfsu	832.822388095238	stfsux	867.700245238096
stfsx	850.82635	sth	842.171942857144
sthbrx	845.111561904762	sthu	833.344518518519
sthux	863.313707142857	sthx	842.219530952382
stw	843.715845238096	stwbrx	848.73669047619
stwu	823.972207142856	stwux	865.723164021165
stwx	843.9068	eciwx	851.449814285715
ecowx	843.9068	tlbie	849.615745
tlbld	849.615745	tlbi	849.615745
tlbsync	849.615745	dcbf	849.615745
dcbi	849.615745	dcbst	849.615745
dcbt	849.615745	dcbtst	849.615745
dcbz	849.615745	icbi	849.615745
stwcx.	874.115	lswx	857.0375
lswi	840.5925	stswx	868.296
stswi	850.01675	lmw	845.6525
stmw	845.526	lwarx	609.0975

Tabla A–5: Medidas de corriente base para las instrucciones de sistema del PPC603e.

Instruccin	**Corriente** (mA)	**Instruccin**	**Corriente** (mA)
eieio	670.13	isync	672.85
lmw	693.829166666667	lswi	695.122
mfcr	728.601499999999	mtcrf	743.4
mfmsr	727.556111111111	mfspr	742.731666666667
mftb	767.673	sync	665.975
crand	735.196576470588	crandc	736.393452941177
creqv	742.340241176471	cror	743.31487556561
crnor	715.05	crorc	740.3166
crxor	737.6544	mcrf	710.820227272727
cmp	702.879314851485	cmpi	707.147346534654
cmpl	704.556799009901	cmpli	705.471442574258
crnand	740.562830578513	crnor	740.228739669422
mfsr	696.675	mfsrin	690.375
mtsr	695.1525	mtsrin	683.655
mcrxr	703.5	mtmsr	448.7175
mtspr	751.7475	rfi	723.7503
sc	723.7503	mttb	730.8525

Tabla A–6: Medidas de corriente base para las instrucciones de salto del PPC603e.

Instruccin	**Corriente** (mA)	**Instruccin**	**Corriente** (mA)
b	678.335	ba	696.93
bl	691.92	bla	708.335
bc	683.626777777778	bcl	688.644166666666
bclr	805.35	bclrl	807.45
bcctr	805.665	bcctrl	720.3
bca	700.35	bcla	704.55

Apéndice B
COSTOS DE CORRIENTE POR CONMUTACIÓN

Tabla B–1: Diferencia de corriente por ejecutar el par de instrucciones respecto al costo base (1)

Par	**Dif. I** (A)	**Par**	**Dif. I** (A)	**Par**	**Dif. I** (A)
nand, nor	0.0505	neg, subfze	0.0505	subfc, subfe	0.0506
nand, or	0.0509	andis., xoris	0.0514	addic, xori	0.0519
mulli, srawi	0.0520	nand, xor	0.0530	eqv, xor	0.0531
addic, mulli	0.0531	andc, nor	0.0537	cntlzw, srawi	0.0538
addc, andc	0.0540	eqv, or	0.0540	mulli, extsh	0.0542
subfic, ori	0.0544	eqv, slw	0.0544	slw, srw	0.0546
addic, ori	0.0550	eqv, sraw	0.0552	and, nand	0.0557
nor, xor	0.0558	and, srw	0.0569	orc, xor	0.0583
adde, mulli	0.0588	subfc, nor	0.0591	eqv, orc	0.0591
andc, orc	0.0592	adde, orc	0.0597	eqv, nor	0.0606
eqv, nand	0.0608	rlwnm, sraw	0.0609	and, sraw	0.0614
and, nor	0.0615	oris, srawi	0.0617	mulli, extsb	0.0618
cntlzw, extsh	0.0626	adde, xor	0.0628	extsb, srawi	0.0629
mulli, ori	0.0635	and, slw	0.0635	mulli, xoris	0.0636
nor, or	0.0637	nand, srw	0.0638	mulli, oris	0.0647
ori, oris	0.0649	mulli, rlwinm	0.0651	orc, sraw	0.0653
addc, mulli	0.0653	adde, or	0.0657	xor, slw	0.0658
or, srw	0.0660	orc, srw	0.0661	rlwimi, rlwnm	0.0662
adde, and	0.0675	addc, sraw	0.0680	addze, cntlzw	0.0681
eqv, srw	0.0690	nor, slw	0.0693	mulli, orc	0.0695
xoris, srawi	0.0696	addze, extsh	0.0697	addc, subfe	0.0700
mulli, xori	0.0701	or, sraw	0.0703	mulli, neg	0.0714
mulli, xor	0.0716	adde, eqv	0.0716	neg, srawi	0.0717
oris, xori	0.0719	xor, srw	0.0722	mulli, and	0.0724
addic, or	0.0725	nand, sraw	0.0726	addc, srw	0.0728
extsb, xoris	0.0745	addc, nand	0.0746	nor, sraw	0.0747
mulli, subfze	0.0748	addc, xor	0.0750	mulli, andi.	0.0751
adde, ori	0.0751	addze, extsb	0.0752	addc, orc	0.0759
xori, xoris	0.0766	mulli, or	0.0767	adde, cntlzw	0.0770

Tabla B–2: Diferencia de corriente por ejecutar el par de instrucciones respecto al costo base (2)

Par	**Dif. I** (A)	**Par**	**Dif. I** (A)	**Par**	**Dif. I** (A)
cntlzw, mtfsfi	0.0501	neg, mffs	0.0501	subfc, fneg	0.0502
srawi, fneg	0.0502	and, mcrfs	0.0502	addi, fmadds	0.0502
subfe, mcrfs	0.0504	addme, mcrfs	0.0504	orc, mtfsfi	0.0504
and, fneg	0.0505	divwu, fsub	0.0506	rlwimi, mtfsfi	0.0507
eqv, mcrfs	0.0507	divwu, fneg	0.0507	divwu, fnmsub	0.0508
andi., mtfsfi	0.0509	andc, mtfsb1	0.0509	add, mtfsb0	0.0510
eqv, fneg	0.0510	or, fneg	0.0511	mulhwu, fmadds	0.0511
oris, fmuls	0.0511	oris, mffs	0.0512	mulhw, fmadds	0.0512
subfic, mcrfs	0.0512	rlwnm, mtfsfi	0.0512	divwu, fcmpu	0.0514
andc, mffs	0.0515	subf, mtfsb1	0.0515	oris, mtfsb1	0.0516
addic, mtfsb1	0.0516	xor, fneg	0.0516	xoris, fneg	0.0517
neg, mtfsb0	0.0517	andis., mtfsfi	0.0517	cntlzw, fneg	0.0518
adde, mtfsb0	0.0518	rlwinm, fmuls	0.0518	subfic, fneg	0.0520
xor, mtfsfi	0.0520	subfze, mffs	0.0520	divwu, fmuls	0.0521
subfe, fneg	0.0521	subf, mffs	0.0521	mulli, fsub	0.0522
addic, mffs	0.0522	addze, mtfsfi	0.0522	adde, mffs	0.0522
ori, fneg	0.0523	addc, fneg	0.0524	mulli, frsp	0.0524
add, mffs	0.0525	ori, mcrfs	0.0525	slw, fmuls	0.0525
extsh, fneg	0.0526	neg, mtfsb1	0.0526	adde, mtfsb1	0.0526
xori, mtfsfi	0.0526	divw, fcmpo	0.0527	rlwnm, fneg	0.0529
divw, fsubs	0.0530	subf, fmuls	0.0531	addze, mcrfs	0.0531
xori, fneg	0.0533	or, mcrfs	0.0533	nand, mtfsfi	0.0533
extsb, fneg	0.0533	add, fmadds	0.0533	sraw, mcrfs	0.0534
subfic, mffs	0.0534	subfze, mtfsb0	0.0535	sraw, fneg	0.0536
subf, mtfsb0	0.0537	addic, mtfsb0	0.0537	slw, mtfsb0	0.0538
extsb, mtfsfi	0.0539	and, mtfsb0	0.0539	extsh, mtfsfi	0.0539
andc, mtfsb0	0.0540	orc, fneg	0.0541	addme, mtfsb0	0.0541
addic, fmuls	0.0541	nand, mcrfs	0.0542	adde, fmuls	0.0542
slw, mffs	0.0543	rlwimi, fneg	0.0543	addc, mtfsb0	0.0544
sraw, mtfsfi	0.0545	addme, mffs	0.0546	andi., mcrfs	0.0546

Tabla B–3: Diferencia de corriente por ejecutar el par de instrucciones respecto al costo base (3)

Par	**Dif. I** (A)	**Par**	**Dif. I** (A)	**Par**	**Dif. I** (A)
xor, stswi	0.0505	nand, lwzu	0.0507	or, lwzu	0.0509
divw, lbzux	0.0509	divw, lfsux	0.0517	divwu, lbzu	0.0520
divw, lfdux	0.0523	divwu, lhau	0.0524	divw, lwzux	0.0524
divwu, lfsu	0.0527	divwu, lhzu	0.0533	divw, lwzu	0.0534
divwu, lfsx	0.0544	divwu, lhaux	0.0547	divw, lhzux	0.0548
divw, lhaux	0.0552	divwu, lfdx	0.0554	divwu, lwzu	0.0569
divwu, lfsux	0.0572	divw, stswi	0.0576	divwu, lbzux	0.0583
divwu, lfdux	0.0588	divwu, lwzux	0.0599	divwu, lhzux	0.0613
divwu, stswi	0.0644	divw, stswx	0.0653	divwu, stswx	0.0731
sraw, sync	0.0501	or, mfcr	0.0501	addze, cmpli	0.0501
mulhw, mtspr	0.0502	addme, mttb	0.0502	xoris, mfcr	0.0503
xori, sync	0.0503	rlwimi, sync	0.0504	subfc, mfcr	0.0505
addic, mfcr	0.0505	rlwnm, sync	0.0505	addze, mttb	0.0506
adde, mfcr	0.0506	nand, mtcrf	0.0508	mullw, mtcrf	0.0508
srawi, mfmsr	0.0508	slw, mfmsr	0.0509	andi., cror	0.0509
subfme, mtsrin	0.0511	subfme, mttb	0.0512	sraw, mfcr	0.0512
mullw, mfspr	0.0513	mulli, eieio	0.0514	xori, mfcr	0.0515
ori, mfcr	0.0515	xor, mfmsr	0.0517	xori, cmp	0.0517
subfe, mfcr	0.0518	mullw, mfcr	0.0518	orc, sync	0.0519
srawi, cmpi	0.0522	subfme, mtcrf	0.0522	andis., crorc	0.0523
mulli, mfspr	0.0523	addze, mtcrf	0.0526	extsb, mfcr	0.0526
cntlzw, mfcr	0.0527	xoris, mfmsr	0.0528	subfic, mfcr	0.0529
subfc, mfmsr	0.0531	mulli, mttb	0.0531	addic, mfmsr	0.0531
extsh, mfcr	0.0532	andis., crnor	0.0533	subfze, mfcr	0.0533
andi., crorc	0.0534	neg, mfcr	0.0534	addze, sync	0.0535
nor, sync	0.0535	or, mfmsr	0.0536	adde, mfmsr	0.0536
rlwimi, mfcr	0.0537	mulhwu, mtspr	0.0537	rlwnm, mfcr	0.0537
and, mfmsr	0.0538	mulli, cmp	0.0540	xori, mfmsr	0.0542
addme, mtcrf	0.0542	eqv, mfmsr	0.0543	andis., sync	0.0547
ori, mfmsr	0.0549	andi., crnor	0.0550	srw, mfcr	0.0550
sraw, mfmsr	0.0550	orc, mfcr	0.0551	andis., mcrf	0.0553

Tabla B–4: Diferencia de corriente por ejecutar el par de instrucciones respecto al costo base (4)

Par	**Dif. I** (A)	**Par**	**Dif. I** (A)	**Par**	**Dif. I** (A)
bcl, addc	0.0501	bcl, or	0.0501	bcl, add	0.0501
bcl, subfe	0.0502	bl, addc	0.0503	bcl, xor	0.0503
bl, subfic	0.0504	bcl, subf	0.0504	b, xori	0.0504
bcl, xori	0.0505	bcl, extsb	0.0507	bl, xor	0.0507
bl, subfe	0.0507	bcl, rlwinm	0.0508	bcl, addc	0.0509
bcl, subfic	0.0509	b, andc	0.0510	ba, andi.	0.0510
bcl, eqv	0.0511	bcl, orc	0.0512	bcl, extsh	0.0512
bcl, xor	0.0512	bcl, extsh	0.0513	bcl, addme	0.0513
bcl, addme	0.0513	bcl, subf	0.0514	bcl, neg	0.0514
bcl, neg	0.0514	bcl, rlwimi	0.0516	bl, and	0.0516
bl, or	0.0516	bcl, extsh	0.0517	bla, srw	0.0517
bcl, slw	0.0517	bcl, adde	0.0517	bcl, adde	0.0518
bcl, rlwinm	0.0518	ba, nor	0.0518	bcl, andis.	0.0519
ba, nand	0.0519	bcl, subfe	0.0521	bcl, orc	0.0522
bcl, addme	0.0522	ba, orc	0.0522	bcl, xor	0.0522
bcl, xor	0.0522	b, xor	0.0522	bcl, sraw	0.0523
bla, nand	0.0525	bl, eqv	0.0525	bcl, subfc	0.0525
bcl, xori	0.0525	bcl, subfme	0.0526	bcl, slw	0.0527
bcl, rlwnm	0.0527	bcl, andi.	0.0528	bcl, and	0.0530
bcl, xoris	0.0533	bcl, xoris	0.0533	bcl, xori	0.0533
bcl, subfme	0.0534	b, ori	0.0535	bcl, xori	0.0535
bl, addme	0.0537	b, rlwinm	0.0537	bcl, addc	0.0539
bcl, xoris	0.0539	bcl, and	0.0539	bcl, and	0.0539
bcl, andi.	0.0540	bcl, orc	0.0542	bcl, extsh	0.0542
bcl, andis.	0.0542	bl, andis.	0.0543	bcl, subfe	0.0543
bcl, addze	0.0544	bcl, subf	0.0544	bcl, subf	0.0544
bcl, subfc	0.0544	bcl, rlwinm	0.0544	bcl, subf	0.0545
bcl, addze	0.0546	bcl, addze	0.0547	bcl, slw	0.0547
bcl, subfe	0.0547	b, andis.	0.0548	bcl, rlwinm	0.0548
bcl, andis.	0.0549	bcl, andis.	0.0549	bl, subfc	0.0549

Tabla B–5: Diferencia de corriente por ejecutar el par de instrucciones respecto al costo base (5)

Par	**Dif. I** (A)	**Par**	**Dif. I** (A)	**Par**	**Dif. I** (A)
fmadds, fnmadds	0.0501	fsel, mtfsb0	0.0501	fmsubs, mtfsb0	0.0502
fmadd, fmr	0.0509	fmul, fnmadds	0.0509	fsel, mcrfs	0.0509
fmsub, fctiw	0.0519	fmsubs, mtfsb1	0.0521	fctiw, fabs	0.0531
fsel, mtfsb1	0.0532	fmul, fneg	0.0537	fsubs, fctiw	0.0538
fmsubs, frsp	0.0542	fmul, fnabs	0.0544	fadds, fmul	0.0551
frsqrte, fmadd	0.0551	fctiwz, fnabs	0.0556	fctiw, fcmpo	0.0557
fadd, fsub	0.0557	fmadd, fnabs	0.0570	fmuls, fmadds	0.0575
fmadd, fnmsubs	0.0593	fadds, fctiw	0.0602	fadd, fmr	0.0604
fnmsub, fctiw	0.0607	fmadd, fneg	0.0614	fmsub, frsp	0.0616
fmsubs, fnmadd	0.0629	fadd, fmsub	0.0629	fadds, fctiwz	0.0630
fadd, fnmsub	0.0632	fadd, fnabs	0.0634	fadd, frsqrte	0.0648
fmadds, fmsub	0.0674	fadd, fneg	0.0684	fadd, fabs	0.0687
fmsubs, fnmsub	0.0704	fmul, fsel	0.0709	fmadd, fmsubs	0.0720
fmuls, fnmadds	0.0728	fmsub, fnabs	0.0737	fadds, fmsub	0.0746
fmuls, fmsub	0.0753	fadds, fnmsub	0.0758	fnmadds, fnmsub	0.0764
fmadds, fnmsub	0.0767	fmsub, fmr	0.0779	fnmsub, frsp	0.0788
fadds, fmr	0.0801	fmul, fmsubs	0.0811	fadds, fsub	0.0812
fnmsub, fabs	0.0818	fsub, frsp	0.0826	fadds, fnabs	0.0840
frsqrte, fnmsub	0.0841	fmul, fnmsubs	0.0856	fnmsub, fneg	0.0861
fmsub, fneg	0.0869	fadds, fneg	0.0881	fmsub, fnmadds	0.0881
fadd, fsubs	0.0887	fadds, frsqrte	0.0890	fsel, fmsub	0.0891
fadds, fabs	0.0894	fmuls, fnmsub	0.0895	fadd, fcmpu	0.0911
fadds, fsubs	0.0921	fsel, fmadds	0.0923	fsel, fnmadds	0.0926
frsqrte, fsub	0.0958	fnmsub, fnabs	0.0973	fadd, fcmpo	0.0981

Tabla B–6: Diferencia de corriente por ejecutar el par de instrucciones respecto al costo base (6)

Par	**Dif. I** (A)	**Par**	**Dif. I** (A)	**Par**	**Dif. I** (A)
fmuls, stwbrx	0.0501	fsubs, lfsux	0.0502	mffs, lwbrx	0.0502
fmuls, sthu	0.0502	fcmpu, sthux	0.0502	fres, lwzux	0.0503
fnmsubs, lhbrx	0.0503	fmuls, sthux	0.0504	mffs, stswi	0.0504
mcrfs, lwbrx	0.0504	fabs, stfsu	0.0504	fcmpu, sthbrx	0.0504
fmsubs, lwbrx	0.0505	fmr, stfsx	0.0505	fmr, stfdux	0.0505
fnabs, stfdx	0.0505	fsel, lhzx	0.0505	fsubs, lfdux	0.0506
fmadds, lfdux	0.0506	fres, lbzux	0.0506	fmuls, stwx	0.0506
fadds, stwcx.	0.0507	fadds, stfsux	0.0507	fmsubs, lhzx	0.0507
fsub, stfsu	0.0507	mtfsfi, lhbrx	0.0508	fabs, stfdx	0.0508
fmr, stfsux	0.0508	fmadds, lfdx	0.0508	mcrfs, stswi	0.0508
mtfsb0, lfsx	0.0508	fabs, stfdux	0.0508	fnabs, stfdux	0.0509
fcmpu, stwx	0.0509	fadds, stfdx	0.0509	mtfsb1, lwzx	0.0510
mtfsb0, stmw	0.0510	fnmsubs, lwzux	0.0510	fnabs, stfsx	0.0510
mtfsb0, lwzx	0.0511	fnmsubs, lbzux	0.0511	fabs, stfsux	0.0511
fmadds, lfsx	0.0511	mffs, lhzx	0.0512	fsel, lwbrx	0.0513
fcmpu, stbux	0.0514	fcmpu, stfs	0.0515	fsubs, lfsu	0.0515
fnabs, stfiwx	0.0515	fmuls, stbux	0.0515	fcmpu, sthx	0.0516
fres, lfdx	0.0516	fmr, stfsu	0.0516	mcrfs, lhzx	0.0516
fmadds, stwux	0.0517	fmuls, sthx	0.0517	mtfsb1, lfsx	0.0517
fnmsubs, lwbrx	0.0517	fabs, stfiwx	0.0517	fnmsub, stwcx.	0.0518
fcmpu, stwu	0.0518	fnabs, stfsux	0.0519	fmsubs, stswx	0.0519
fadds, stfiwx	0.0519	fsel, lwzux	0.0520	fadds, stfdux	0.0520
fmsubs, lhau	0.0521	fres, stfsu	0.0522	fmsubs, lhzux	0.0522
fsel, lbzux	0.0523	fmsubs, lbzu	0.0523	mtfsb1, lhax	0.0523
fres, lfsx	0.0525	fres, lhzux	0.0525	fsub, stfiwx	0.0526
fmuls, stwcx.	0.0527	fneg, stfsx	0.0527	fneg, stfdux	0.0527
fsubs, sth	0.0528	mffs, lhbrx	0.0528	mtfsb0, lwbrx	0.0528
fres, lhaux	0.0529	mtfsb1, lbzx	0.0529	fmsubs, lhzu	0.0529
mtfsb1, stmw	0.0529	mcrfs, lhbrx	0.0530	fcmpo, stw	0.0530
fsubs, lfsx	0.0530	fneg, stfsux	0.0531	mtfsfi, stswx	0.0532

Tabla B–7: Diferencia de corriente por ejecutar el par de instrucciones respecto al costo base (7)

Par	**Dif. I** (A)	**Par**	**Dif. I** (A)	**Par**	**Dif. I** (A)
mtfsfi, cmpi	0.0501	fmadds, cmpl	0.0505	mtfsb1, mttb	0.0508
fneg, cmpli	0.0511	mtfsb1, mtcrf	0.0518	fmuls, cmpi	0.0520
fmr, eieio	0.0522	fabs, eieio	0.0526	mffs, mtcrf	0.0529
mtfsb0, mtcrf	0.0530	mtfsb0, cmp	0.0556	mffs, cmpl	0.0557
fmadds, cmpi	0.0559	mtfsb1, cmpl	0.0561	mffs, cmp	0.0561
mtfsfi, cmpli	0.0568	fcmpo, cmpl	0.0570	mtfsb0, cmpl	0.0573
mcrfs, cmpi	0.0579	fadd, eieio	0.0583	mtfsb1, cmp	0.0585
fnmadds, cmp	0.0589	fnmadds, cmpl	0.0600	fmuls, cmpli	0.0604
fcmpu, eieio	0.0607	mffs, cmpi	0.0613	mtfsb0, cmpi	0.0616
mtfsb1, cmpi	0.0622	fnmadds, cmpi	0.0629	fmsub, cmp	0.0636
fmadds, cmpli	0.0642	fmsub, cmpl	0.0643	mcrfs, cmpli	0.0653
fcmpu, cmpl	0.0655	fcmpo, cmp	0.0658	fmsub, eieio	0.0660
fnmsub, cmp	0.0662	fnmsub, cmpl	0.0666	mffs, cmpli	0.0685
fnabs, eieio	0.0686	mtfsb1, cmpli	0.0689	fsel, cmp	0.0692
fmsub, cmpi	0.0701	fsel, cmpl	0.0708	mtfsb0, cmpli	0.0710
fnmadds, cmpli	0.0718	fnmsub, cmpi	0.0727	fnmsub, eieio	0.0747
fsel, cmpi	0.0752	fcmpo, eieio	0.0762	fneg, eieio	0.0765
fsubs, eieio	0.0772	fnmsubs, crand	0.0772	fnmsubs, creqv	0.0775
fnmsubs, crandc	0.0779	fnmsubs, crxor	0.0783	fmsub, cmpli	0.0784
fnmsubs, crnand	0.0789	fmsubs, creqv	0.0791	fmsubs, crandc	0.0795
fmsubs, crand	0.0798	fmsubs, crxor	0.0802	fmsubs, crnand	0.0805
fnmsub, cmpli	0.0808	fnmsubs, cror	0.0813	fnmsubs, crnor	0.0818
fnmsubs, mcrf	0.0820	fmsubs, crnor	0.0836	fsel, cmpli	0.0836
fmsubs, cror	0.0839	fnmsubs, crorc	0.0840	fadds, eieio	0.0843
fnmsubs, mftb	0.0845	fmsubs, mcrf	0.0846	fnmsubs, mttb	0.0853
fnmsubs, mfspr	0.0854	fmsubs, mftb	0.0863	fmsubs, crorc	0.0865
fmsubs, mttb	0.0878	fmuls, eieio	0.0878	fmsubs, mfspr	0.0880
fnmsubs, mtcrf	0.0923	fmsubs, mtcrf	0.0948	fnmsubs, mfmsr	0.0955

Tabla B–8: Diferencia de corriente por ejecutar el par de instrucciones respecto al costo base (8)

Par	**Dif. I** (A)	**Par**	**Dif. I** (A)	**Par**	**Dif. I** (A)
sthbrx, mfsr	0.0502	sthbrx, mfsrin	0.0502	stfiwx, mfsr	0.0507
stfsux, mtsrin	0.0510	sthu, mtsrin	0.0513	sthbrx, mtsr	0.0523
stfiwx, mtsr	0.0523	stfsu, mfsrin	0.0525	stfdx, mtsrin	0.0529
stfsx, mtsrin	0.0530	stbx, mtsrin	0.0535	stwu, mfsrin	0.0535
stwbrx, mtsrin	0.0539	stbu, mtsrin	0.0549	stwx, mtsrin	0.0552
sthx, mtsrin	0.0560	sthbrx, mtsrin	0.0577	stwu, mtsrin	0.0577
mttb, mcrf	0.0504	creqv, mtcrf	0.0505	mtsr, cmpli	0.0506
crnor, cmp	0.0507	crandc, cmpi	0.0508	mttb, mfcr	0.0510
crand, cmpi	0.0511	crxor, cmpi	0.0511	mcrf, cmpl	0.0513
sync, cmpl	0.0513	eieio, crnor	0.0513	eieio, mcrf	0.0515
mcrf, cmp	0.0515	crnand, cmpi	0.0516	mfmsr, cmp	0.0516
mtspr, cmpli	0.0517	mtsrin, cmpli	0.0519	sync, cmp	0.0521
crnor, mtcrf	0.0533	eieio, mttb	0.0538	sync, cmpi	0.0541
cror, cmpi	0.0543	mfcr, cmpi	0.0563	mfmsr, mttb	0.0564
mttb, crnor	0.0564	crorc, cmpi	0.0567	mfspr, cmpli	0.0573
eieio, mtcrf	0.0578	mtcrf, cmp	0.0582	mfmsr, mtcrf	0.0585
mtcrf, cmpl	0.0585	mfmsr, cmpi	0.0588	crnor, cmpi	0.0588
mttb, cmpli	0.0590	cmpli, cmpi	0.0593	creqv, cmpli	0.0603
mfmsr, eieio	0.0604	mcrf, cmpi	0.0607	crand, cmpli	0.0610
crandc, cmpli	0.0613	crnand, cmpli	0.0617	crxor, cmpli	0.0617
sync, cmpli	0.0621	eieio, mfcr	0.0626	mftb, cmp	0.0637
cror, cmpli	0.0641	mfcr, cmpli	0.0661	crorc, cmpli	0.0665
mtcrf, cmpi	0.0672	mfmsr, cmpli	0.0676	crnor, cmpli	0.0705
mcrf, cmpli	0.0705	mtcrf, cmpli	0.0740	mfmsr, cmpl	0.0840
bcl, mfcr	0.0511	bcl, cmp	0.0518	bcl, cmpi	0.0523
bcl, cmpi	0.0523	bcl, cmpi	0.0523	bcl, cmpl	0.0525
bcl, cmpl	0.0525	bcl, cmpli	0.0551	bcl, cmpli	0.0551
bcl, cmpli	0.0551	bcl, cmpl	0.0555	bcl, cmpl	0.0555
bcl, mfmsr	0.0571	bcl, mftb	0.0581	bcl, cmpli	0.0581
bcl, mfspr	0.0584	bcl, mttb	0.0640	bcl, mtcrf	0.0670
bcl, cmp	0.0753	bcl, cmpi	0.0793	bcl, cmpl	0.0805

Apéndice C
INSTRUCCIONES NO INCLUIDAS EN EL PERFIL DE POTENCIA

Tabla C–1: Instrucciones no consideradas para la medida.

Instrucción	**Función**	**Asignación**
dcbz	Borrar un bloque de cache	Promedio SRU
dcba	cargar un bloque de cache	Promedio SRU
dcbf	expulsar un bloque de cache	Promedio SRU
icbi	Invalidar bloque de cache	Promedio SRU
mtsr	mover a registro de segmento (manejo de memoria)	mtspr
tw	captura de condiciones especiales	cmp
sc	llamado al sistema	b (salto)
lwarx	carga sincronizada	Promedio SRU
stwcx	escritura sincronizada	Promedio SRU
rfi	retornar de interrupción	b (salto)
mtmsr	mover al registro de estado de máquina	mtspr
tlbie	invalidar tabla de traducción de dirección	Promedio SRU
tlbia	invalidar todas las tabla de traducción de dirección	Promedio SRU
tlbsync	sincronizar tabla de traducción de dirección	Promedio SRU