Diseño, validación e implementación de una arquitectura RISC

por

Luciano César Natale

Tesis presentada para optar al Título de

Ingeniero Electrónico

por la

Facultad de Ingeniería de la Universidad de Buenos Aires

Director:

_____

Ing. Nicolás Alvarez

Co-Director:

_____

Ing. Octavio Alpago

Miembros del Jurado:

_____

Ing. XXXXXXXXXXXXXXXXXXXX

_____

Ing. XXXXXXXXXXXXXXXXXXXX

_____

Ing. XXXXXXXXXXXXXXXXXXXX

Calificación: _____ Fecha: _____

Universidad de Buenos Aires, Facultad de Ingeniería

Diseño, valdiación e implementación de una arquitectura RISC

por

Luciano César Natale

Resúmen

El presente trabajo constituye la Tesis de Grado necesaria para obtener el título de Ingeniero Electrónico de la Facultad de Ingeniería de la Universidad de Buenos Aires.

El objetivo de este trabajo es el diseño, la validación y la implementación de una arquitectura RISC con el objetivo de generar un núcleo de procesamiento, sintetizable en FPGA, altamente configurable y suficientemente flexible y sencillo para ser utilizado en distintas aplicaciones dentro del ámbito de la investigación en los Laboratorios de Microelectrónica y de Sistemas Embebidos. Se presenta la teoría e historia necesaria para poder explicar las decisiones de diseño adoptadas en el desarrollo de la arquitectura. Se presentan vectores de prueba para las validaciones posteriores. Se verifica el diseño generado mediante emuladores. Se implementa el diseño en un lenguaje de descripción de hardware. Se sintetiza en FPGA el diseño y se contrastan los resultados con las emulaciones realizadas. Finalmente se analizan los resultados obtenidos, se contrastan los mismos contra trabajos similares, y se proponen trabajos futuros.

Agradecimientos

Agradecimientos Lucho.

Dedicatoria Lucho

Índice

1. Introducción al trabajo de tesis **1**

1.1. Objetivo 2

1.2. Alcance 3

1.3. Organización del trabajo 4

2. Marco teório: arquitectura de procesadores **5**

2.1. Perspectiva histórica 6

2.1.1. La “Máquina de Turing” 7

2.1.2. La revolución digital 8

2.1.3. Silicio, dispositivos semiconductores y transistores 10

2.1.4. Circuitos integrados y tencología CMOS 10

2.1.5. Microprocesadores 11

2.1.5.1. CADC 12

2.1.5.2. Four-Phase Systems AL1 13

2.1.5.3. Pico/General Instruments 13

2.1.5.4. Intel 4004 15

2.2. Evolución hacia las arquitecturas modernas 16

2.2.1. Clasificación de las arquitecturas según el acceso a la memoria de instrucciones y datos 16

2.2.1.1. Arquitectura von Neumann 16

2.2.1.2. Arquitectura Harvard 19

2.2.1.3. Harvard vs. von Neumann y Harvard modificada 21
2.2.1.3.1. Arquitectura de caché separado 21
2.2.1.3.2. Acesso a instrucciones como datos 21
2.2.1.3.3. Ejecución de datos 22
2.2.1.3.4. Características de las arquitecturas Harvard modificadas 22
2.2.1.4. Aplicaciones de esta clasificación en la actualidad 23
2.2.2. Taxonomía de Flynn 23
2.2.2.1. SISD: Simple Instruction stream, Simple Data stream . . 24
2.2.2.2. SIMD: Simple Instruction stream, Multiple Data stream . 24
2.2.2.3. MISD: Múltiple Instruction stream, Simple Data stream . 25
2.2.2.4. MIMD: Multiple Instruction stream, Multiple Data stream 25
2.2.2.5. Actualización de la taxonomía de Flynn 25
2.2.2.5.1. SPMD: Single Program, Multiple Data streams . 25
2.2.2.5.2. MPMD: Multiple Programs, Multiple Data streams 27
2.2.3. Clasificación de las arquitecturas según el manejo de los datos y cantidad de operandos 27
2.2.3.1. Modos de direccionamiento 27
2.2.3.2. Accumulator 28
2.2.3.3. Stack 29
2.2.3.4. Load / Store 29
2.2.3.5. Register / Memory 30
2.2.4. Clasificación de las arquitecturas según su conjunto de instrucciones 30
2.2.4.1. CISC: Complex Instruction Set Computer 30
2.2.4.2. RISC: Reduced Instruction Set Computer 31
2.2.5. Elecciones en el diseño propuesto 31
2.3. Desempeño; mediciones y optimizaciones 32
2.3.1. Ecuación de desempeño 33
2.3.2. Mediciones de desempeño 34

2.3.3. Optimizaciones y Ley de Amdahl 35
2.4. Paralelismo 37
2.5. Técnicas de optimización en el diseño de arquitecturas RISC 39
2.5.1. Explotando ILP 39
2.5.1.1. Pipeline 39
2.5.1.1.1. Depedencias de recursos 41
2.5.1.1.2. Dependencias de control 41
2.5.1.1.3. Dependencias de datos 46
2.5.1.2. Forwarding 46
2.5.1.3. Predicción de salto 46
2.5.2. Explotando ThLP 47
2.5.3. Multiple Issue 47
2.5.4. Scalar - Superscalar 47
2.5.5. Multicore 47
2.6. Jerarquía de memoria 48
2.7. Diseño del XFire 48

Índice de Tablas

2.1. Taxonomía de Flynn. 24

Índice de Figuras

2.1. Esquema simplificado de un “sistema de procesamiento”. 6

2.2. Primer transistor desarrollado por John Bardeen y Walter Brattain bajo la dirección de William Shockley en los Laboratorios Bell de AT&T en el año 1947. 9

2.3. Primer circuito integrado presentado por Texas Instruments, Inc. el 12 de Septiembre de 1958. 12

2.4. Grumman F14 Tomcat, avión de combate de la US Navy cuyos sistemas de vuelo eran controlados por la CADC. 13

2.5. Imágen microscópica de la AL1 desarrollada por Four-Phase Systems Inc.. 14

2.6. Imágen microscópica del PICO1/GI250- Desarrollo colaborativo entre Pico y General Instruments de *chip* único para la Monroe/Litton Royal Digital III Calcultator. 14

2.7. Encapsulado del Intel 4004; que es considerado el primer microprocesador comercial en ser introducido en el mercado. 15

2.8. Carátula del escrito de von Neumann First Draft of a Report on the EDVAC. 17

2.9. Cinta de papel perforado con instrucciones de la Mark I. 20

2.10. Diagramas de la taxonomía de Flynn. 26

2.11. Esquema simplificado de un predictor de segundo órden. 45

2.12. Esquema simplificado de un predictor global (2,2). 45

2.13. Esquema simplificado de un búfer de predicción de saltos. 47

Capítulo 1

Introducción al trabajo de tesis

El presente trabajo se encuentra enfocado en el contexto del diseño de hardware digital. El mismo fue motivado por la necesidad de contar un con núcleo de procesamiento altamente configurable y suficientemente flexible y sencillo para distintas aplicaciones dentro del ámbito de la investigación en los Laboratorios de Microelectrónica y de Sistemas Embebidos; sintetizable en FPGA[1]. La arquitectura a desarrollar será del tipo RISC[2].

Desde la aparición de los microprocesadores a mediados de los años 70, la tendencia fue el aumento de la complejidad de las arquitecturas, generando un efecto de "bola de nieve", al ir superponiendo capas sobre un núcleo central. Existió, entonces, una reacción adversa a esta tendencia. Por ejemplo, la arquitectura experimental de IBM 801; y también en Berkeley, Patterson y Ditzel fueron los primeros en acuñar el término RISC, para descibir una nueva clase de arquitectura que deshacía el camino del resto de las arquitecturas hasta el momento, conocidas, en contraposición, como CISC[3]. A partir de este antecedente, los principales fabricantes de microprocesadores han lanzado al mercado sus propias implementaciones basadas en los principios establecidos en IBM y Berkeley.

El concepto de las arquitecturas RISC se basa, principalmente, en el hecho de que al simplificar la lógica necesaria para la ejecución de una instrucción permite aumentar la frecuencia de operación de las compuertas que componen la lógica. Además, es posible

[1] "Field Programmable Gate Array" (Arreglo de compuertas lógicas reprogramables): dispositivo electrónico de lógica programable utilizado para prototipar diseños lógicos, así como también en producciones que no justifican la fabricación de millones de circuitos idénticos.

[2] "Reduced Instruction Set Computer" (Computadora de conjunto de instrucciones reducido): técnica de diseño de unidades de procesamiento basada en el hecho de que un conjunto de instrucciones simple provee una mayor performance al ser combinado con una arquitectura capaz de ejecutar dichas instrucciones en algunos pocos ciclos de máquina.

[3] "Complex Instruction Set Computer" (Computadora de conjunto de instrucciones complejo): técnica de diseño de unidades de procesamiento basadas en el hecho de que el conjunto de instrucciones debe ser lo más poderoso posible.

dividir la ejecución de las instrucciones en etapas sencillas y consecutivas, permitiendo de esta manera implementar fácilmente optimizaciones como, por ejemplo, una arquitectura de *pipeline*[4]. Es por esto que el conjunto de instrucciones es sencillo, permitiendo solamente operaciones básicas entre registros internos del microprocesador. El trabajo realizado por cada instrucción, en general, es menor que el generado por una instrucción CISC, pero se hace de manera sencilla y rápida. Es importante notar que no solamente la ganancia radica en poder aumentar la frecuencia de operación de la lógica, sino que estas condiciones facilitan el desarrollo de diseños de bajo consumo, característica muy valorada en el nicho de los sistemas embebidos.

El mercado de los sistemas embebidos es excesivamente amplio y está inserto en todas las industrias. En un automóvil, por ejemplo, podemos encontrar microprocesadores en el sistema de frenos, en la central de inyección electrónica, en el sistema de entretenimiento y navegación, etc. La otra arista de vital importancia para el mercado de los sistemas embebidos, es el de los dispositivos móviles, donde se vuelve vital el requerimiento de bajo consumo. Estamos viviendo la revolución de IoT[5], que se trata básicamente de sistemas embebidos autónomos que estan conectados a "la nube" y pueden ser monitoreados y controlados remotamente a través de Internet.

Dentro del universo de las arquitecturas RISC, actualmente se destacan dos: MIPS y ARM. La primera, fue desarrollada por un grupo de investigadores de la Universidad de Stanford (entre ellos John L. Hennessy, pionero del concepto RISC junto a David Patterson, coautores de la bibliografía más relevante del área). Esta arquitectura, por su sencillez, es la predilecta al momento del desarrollo de cursos enfocados en la enseñanza de arquitectura de computadoras. Si bien MIPS posee gran relevancia académica, es muy popular en el mercado de los microprocesadores en sistemas embebidos como equipos de telecomunicaciones, decodificadores de TV digital, y consolas de entretenimiento, con ejemplos muy conocidos como *Nintendo* y *PlaySation*. ARM, por otro lado, ha ganado una importante porción del mercado de los sistemas embebidos (con un gran aporte de los dispositivos móviles), basando su modelo de negocios en la venta de la propiedad intelectual (IP, *intellectual property*) del diseño de los microprocesadores a las empresas que finalmente producen el microprocesador.

1.1. Objetivo

La Tesis tiene como objetivo principal el diseño, la validación e implementación de una arquitectura RISC y su conjunto de instrucciones.

[4] "Pipeline": técnica de optimización en el diseño de arquitecturas de computadoras en la que se segmenta la ejecución de las instrucciones en múltiples etapas, permitiendo que múltiples instrucciones estén ejecutándose en paralelo. Se desarrollará en profundidad en 2.5.1.1

[5] "Internet of Things" (Internet de las cosas): es un concepto que se refiere a la interconexión digital de objetos cotidianos con internet.

El enfoque de la tesis se basará en un desarrollo teórico del conjunto de instrucciones y de las características de la arquitectura; y en el desarrollo práctico del emulador y la implementación en lenguaje descriptor de hardware.

El concepto central detrás del desarollo será el de **ortogonalidad**. Esto implica, por una parte, que los bloques constructivos de la arquitectura que se repiten sean independientes e indiferenciables entre sí. Por otra parte, los formatos de las instrucciones, en la medida de lo posible, se diseñaran de manera tal que se pueda mantener el mismo ancho de campo para los datos inmediatos y los desplazamientos (excepto en los casos donde es explícitamente conveniente agrandarlos sin penalizar la complejidad del diseño).

El objetivo perseguido va a ser el de mantener la sencillez y la ortogonalidad, favoreciendo así la simplificación de la implementación. Se trabajará en el desarrollo de la definición de la arquitectura y su conjunto de instrucciones en favor de este objetivo. Se definirá la interfaz física para la conectividad con periféricos, los tipos de datos que maneja la arquitectura, la cantidad y tipos de registros internos, el acceso a memoria de programa y de datos con su organización y modo de direccionamiento, la interfaz con la ALU[6] y la FPU[7], mecanismos de manejos de excepeciones e interrupciones, modos de operación y manejo de periféricos. Luego se definirá el conjunto de instrucciones que ejecutará la arquitectura.

Una vez definida la arquitectura y su conjunto de instrucciones, se prodecerá a diseñar los vectores de prueba para poder validar las implementaciones. Se desarrollará un emulador de la arquitectura que deberá validar los vectores de prueba diseñados. Una vez concluida esta etapa, se implementará a nivel RTL el diseño en *Verilog*. Este diseño será validado mediante simulaciones y utilizando dispositivos programables. Se validará también contra los vectores de prueba. Se analizarán los recursos utilizados en dispositivos FPGA. Se realizará un análisis comparativo entre la arquitectura desarrollada y otras arquitecturas RISC.

1.2. Alcance

Como resultados a obtener de la tesis se tienen los siguientes:

- Especificación completa de la arquitectura
- Vectores de prueba
- Emulador de la arquitectura

[6] "Arithmetic Logic Unit" (Unidad aritmético-lógica): bloque constructivo encargado de realizar las operaciones aritméticas y lógicas sobre los datos.

[7] "Floating Point Unit" (Unidad de punto flotante): bloque constructivo encargado de realizar las operaciones en punto flotante sobre los datos.

- *IP Core* codificado en el lenguaje *Verilog* de la arquitectura completa
- Resultado de los vectores de prueba tanto en el emulador como en el *IP Core*
- Análisis comparativo entre la arquitectura desarrollada y otras arquitecturas RISC
- Proposición de trabajos futuros y/o mejoras.

1.3. Organización del trabajo

En esta sección se describe la organización de la presente tesis. Con el objetivo de que la misma sea autocontenida, los primeros capítulos se ocupan de presentar las bases o conocimientos necesarios para comprender la totalidad del trabajo.

El desarrollo de la tesis se organiza de la siguiente forma:

- En el capítulo 2 se presentará la teoría general de las arquitecturas de procesadores y una revisión histórica sobre el tema. Se estudiará la diferenciación entre los universos de procesadores CISC y RISC y se justificará la elección de diseñar una arquitectura RISC para la tesis. Se presentarán las técnicas de diseño de arquitecturas estudiadas. Además se presentarán reseñas de otras arquitecturas actuales y sus decisiones de diseño, para luego contrastarlas con los objetivos perseguidos por el presente trabajo.
- En el capítulo 3 se presentará la especificación completa de la arquitectura diseñada, explicitando los criterios y las decisiones de diseño tomadas. Además se presentará el diseño de los vectores de prueba que se utilizarán para validar las implementaciones de la arquitectura.
- En el capítulo 4 se desarrollarán las implementaciones del emulador de la arquitectura y del *IP Core* en RTL. Dicho RTL cumplirá con ciertas condiciones de portabilidad y legibilidad del código, para que el mismo sea efectivamente un IP core. Se evaluará y validará el *IP Core* utilizando simuladores. Se explicitarán las decisiones de diseño necesarias para pasar de la abstracción del diseño a la implementación real.
- En el caítulo 5 se validarán las implementaciones del capítulo 4 mediante los vectores de prueba diseñados para el capítulo 3. El *IP Core* será sintetizado para distintos dispositivos FPGA. Se analizará en cada caso el consumo de recursos utilizados, máxima frecuencia de operación y la potencia consumida
- En el capítulo 6 se extraerán las conclusiones pertinentes sobre los resultados obtenidos y se propondrán futuras mejoras de la arquitecturas a partir del análisis realizado.

Capítulo 2

Marco teório: arquitectura de procesadores

En líneas generales un procesador es un sistema que permite, por un lado ingresar instrucciones y datos obteniendo en consecuencia los resultados de operar lo indicado en las instrucciones sobre los datos. No es necesario para tal fin, definir ninguna tecnología que soporte este comportamiento; se trata más bien de un desarrollo teórico. Dicho desarrollo implica distinguir las distintas partes que lo componen. En un primera aproximación, en un sistema de procesamiento podemos distinguir cuatro componentes:

- Datos de entrada
- Instrucciones
- Unidad de procesamiento
- Datos de salida

Estos componentes se relacionan de la forma mostrada en la figura 2.1. Debe notarse que los datos de entrada y las instrucciones ingresan a la unidad de procesamiento, la cual genera datos de salida como resultado de operar las instrucciones sobre los datos de entrada.
Los datos de entrada representan el dominio sobre el cual puede operar la unidad de procesamiento. Para definir un sistema de procesamiento debemos especificar, entonces, cuál es el dominio que puede manejar. Dicha definición deberá especificar el formato y cantidad de datos que acepta la unidad de procesamiento, así como los mecanismos para ingresarlos al sistema.
Las instrucciones definirán las operaciones que la unidad de procesamiento puede realizar sobre los datos de entrada. Por lo tanto, para definir un sistema de procesamiento, debemos definir qué operaciones será capaz de realizar sobre el dominio de entrada del

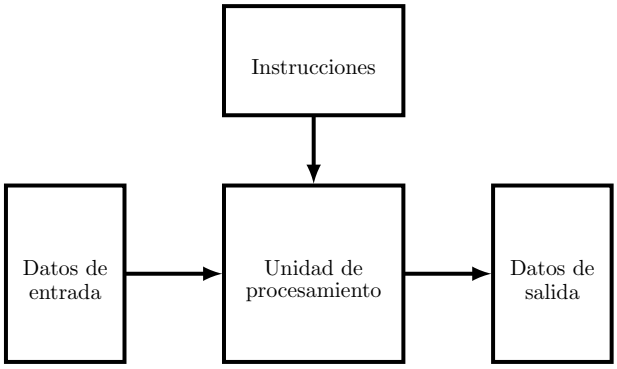


Figura 2.1: Esquema simplificado de un "sistema de procesamiento".

mismo. A su vez, se debe definir la salida esperada de las instrucciones; es por eso que al definir las instrucciones estamos definiendo intrínsecamente el dominio de salida del sistema. Es importante notar que la información sobre las instrucciones también representa una entrada para el sistema, es por eso que, como en el caso de los datos de entrada, se deberá especificar formato y cantidad que acepta la unidad de procesamiento, así como los mecanismos para ingresarlas.
Los datos de salida representan el dominio sobre el cuál las instrucciones vuelcan el resultado de las operaciones realizadas sobre los datos de entrada. El dominio de salida, como fue notado antes, queda definido al definir las instrucciones, pero debe definirse el formato y cantidad de los mismos, así como los mecanismos necesarios para extraerlos del sistema.
En este contexto la unidad de procesamiento es la encargada de recibir las instrucciones y datos, ejecutar las operaciones para finalmente presentar los resultados.
Un microprocesadores es —en efecto— una implementación de un sistema de procesamiento. El sustento físico de dicha implementación es la microelectrónica. En las siguientes secciones de este capítulo, se repasará el marco histórico en el que surgen los microprocesadores y las arquitecturas asociadas.

2.1. Perspectiva histórica

Desde épocas remotas, el hombre se ha destacado en el mundo animal por su capacidad de modificar su entorno para resolver problemas recurrentes. Es esta capacidad la fortaleza de la especie en la naturaleza. La película "2001: Odisea del esapcio"[1] narra, desde un enfoque particular, parte de la evolución del ser humano, o al menos la interpretación de los autores sobre la misma. En la misma, ubicándose temporalmente varios millones de años atrás, un clan de cavernícolas prehumanos intentan sobrevivir en condiciones

[1] "2001: Odisea en el espacio": es un film del año 1968 dirigida por Stanley Kubrick basada en la novela de Arthur C. Clarke.

extremas. Comen los pocos hierbajos que pueden encontrar en el desolado paisaje, hierbajos que para colmo han de compartir con una manada de tapires que habita la misma zona. La única fuente de agua del clan —un simple charco— les es arrebatada por un clan rival. Por si fuera poco, este desdichado clan vive permanentemente amenazado por un leopardo que domina la región y que de vez en cuando caza a alguno de sus miembros. En resúmen: este grupo de homínidos padece hambre, frío y miedo, y parecen condenados a una segura extinción. En ese contexto, y por motivos que no vienen al caso, aparece uno de los cavernícolas contemplando el esqueleto de un animal. Parece reflexionar sobre lo que tiene delante, como si estuviese viéndolo desde una nueva perspectiva. Hay algo nuevo en aquellos huesos. Algo que hasta entonces ni él ni ninguno de sus congéneres habían visto. Los huesos que hay tirados por el suelo pueden ser usados. El cavernícola toma el más robusto de los huesos y empieza a golpear el esqueleto; primero con precaución, más tarde con fuerza, hasta que termina consumido por un frenesí violento. Este cavernícola acaba de descubrir el primer arma —la primera herramienta— de la historia. O dicho de otro modo, acaba de aparecer el primer ser humano sobre la faz de la tierra. Gracias al uso del hueso —o de herramientas similares como palos o piedras— el clan que estaba a punto de extinguirse descubre que puede cazar a los tapires con los que convive y comérselos. Así que sus problemas de hambre han terminado. También gracias a sus armas pueden atacar al clan rival y recuperar el charco de agua, lo que soluciona también sus problemas de sed. Y deducimos que serán capaces incluso de defenderse del peligroso leopardo. Los miembros del clan ya no son prehumanos indefensos; ahora son humanos armados.
Esta cita cinematográfica pretende graficar la diferenciación del ser humano en la cadena alimenticia. Gracias al poder de la observación y el razonamiento hemos sido capaces de modificar nuestro entorno para asegurar la supervivencia de la especie en un mundo en el que la misma se encontraba en clara desventaja. En este contexto, el ser humano ha sido capaz de generar un desarollo tecnológico, que hoy en día es vertiginoso.

2.1.1. La "Máquina de Turing"

Alan Mathison Turing[2] es considerado el padre de la computación y la inteligencia artificial. Sus trabajos en el campo de la computación se remontan al año 1936 cuando publica un *paper* llamado "On computable Numbers, with an Application to the Entscheidungsproblem". En este contexto, Turing inventó la llamada Máquina de Turing; esta máquina se trata de un modelo abstracto que manipula símobolos ingresados a través de un medio infinito, de acuerdo a una tabla de reglas, o para ser más exactos, es la definición del modelo matemático de una máquina capaz de hacer dicho trabajo. A pesar de la simplicidad del modelo planteado por Turing, dado cualquier algoritmo computable, una

[2]Alan Mathison Turing (23 June 1912 – 7 June 1954): típicamente considerado el padre de las ciencias de la computación y de la inteligencia artificial, fue un biólogo, matemático, lógico, científico de la computación y de la criptografía inglés que formalizó los conceptos de algoritmo y computación mediante la "Máquina de Turing", que es un modelo de una computadora de propósito general

Máquina de Turing que lo resuelva puede ser construída. Con éste modelo, reformuló los resultados hallados por Kurt Gödel en 1931 sobre los límites de prueba y computabilidad, reemplazando el lenguaje formal planteado por Gödel, por su modelo de máquina. Turing demostró que cualquier problema que pudiera ser representado por un algoritmo, podía ser resuelto por una Máquina de Turing, a su vez probando que el "problema de decisión" (entscheidungsproblem) no tenía solución, al probar que el "Halting Problem" (Problema de Parada) para las Máquinas de Turing es indeterminado según el problema de decisión. En otras palabras, para un problema de este tipo, está indeterminado el hecho de que la máquina termine de computar. Es así que la Máquina de Turing fué capaz de demostrar las limitaciones del poder de cómputo. El modelo de acceso de datos e instrucciones era secuencial, lo cuál es minimalista, pero no apto para implementaciones prácticas. La teoría de autómatas es la rama de la ciencia que hoy se ocupa de estudiar estos modelos matemáticos de computabilidad. Se establece así la propiedad de que un sistema arbitrario de instrucciones sea "Turing-Completo".

2.1.2. La revolución digital

La revolución digital es considerada la tercera revolución industrial, dando orígen a la llamada "Era de la Información". Sus comienzos se remontan a fines de los años 50. La adopción y proliferación de las computadoras digitales y el mantenimiento de registros digitales de información son las características que la definen. De manera implícita, esta revolución está relacionada con los cambios radicales provocados por la computación y las tecnologías de telecomunicaciones. En el corazón de este proceso encontramos dos componentes tecnológicas. Por un lado está la teoría, que puede remontarse a los primeros análisis matemáticos de la lógica, como el álgebra de Boole introducida por primera vez en un pequeño folleto publicado en 1847 bajo el nombre *The Mathematical Analisis of Logic* y la posterior publicación del libro *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*, publicado en 1854. En estas publicaciónes George Boole[3] pretendió utilizar técnicas algebraicas para tratar expresiones de la lógica proposicional. En la actualidad, el álgebra de Boole se aplica de forma generalizada en el ámbito del diseño electrónico. Claude Shannon[4] fué el primero en aplicarla en el diseño de circuitos de conmutación eléctrica biestables, en 1948. Por el otro lado, se encuentra la revolución del silicio y la producción en masa y uso generalizado de circuitos lógicos digitales que permitieron el desarrollo en

[3]George Boole (2 de Noviembre de 1815 - 8 de Diciembre de 1864): Matemático, docente, filósofo y lógico Ingles. Trabajó en los campos de las ecuaciones diferenciales y el álgebra lógica. Su obra "The Laws of Thaught" (Las Leyes del Pensamiento) de 1854 describen lo que hoy conocemos como Álgebra de Boole, piedra fundamental de la "Era de la Información".

[4]Claude Elwood Shannon (30 de Abril de 1916 - 24 de Febrero de 2001): Matemático, ingeniero eléctrico y cryptografista nacido en Estados Unidos de América, es conocido como el padre de la teoría de la información. Con un *paper* publicado en 1948, describió lo que hoy conocemos como teoría de la información. Pero previamente, en 1937, también trabajó en lo que hoy conocemos como la teoría del diseño de circuitos digitales, al realizar su tesis de maestría aplicando las leyes del Álgebra de Boole a circuitos eléctricos, demostrando que podía construirse cualquier función lógica y numérica.

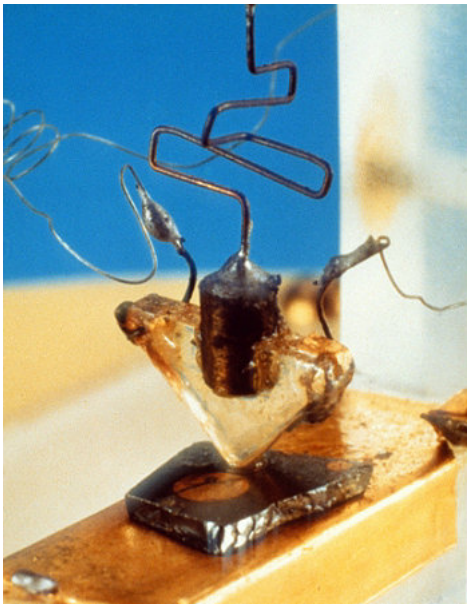


Figura 2.2: Primer transistor desarrollado por John Bardeen y Walter Brattain bajo la dirección de William Shockley en los Laboratorios Bell de AT&T en el año 1947.

gran escala de circuitos electrónicos que implementan funciones lógicas basados en los desarrollos de Boole. Aproximadamente cien años pasaron desde que Boole publicó su teoría, hasta que la tecnología encontró el camino para implementar esos conocimientos de forma práctica y útil. La invención del transistor data del año 1947. En la figura 2.2 se muestra una imágen de una réplica de este primer transistor de la historia. Este invento fué el que permitió la creación de equipos digitales avanzados. En el contexto de la lógica digital, los transistores son utilizados como llaves de conmutación que permiten o no el paso de una señal eléctrica al ser excitados por otra señal. Previo a los transistores, la lógica era implementada con componentes electromecánicos y válvulas termoiónicas de vacío; tecnologías que por su naturaleza eran de dimensiones y consumos energéticos elevados y poco fiables. Para poner esta problemática en perspectiva, comparamos la primera computadora de propósito general electrónica, llamada ENIAC[5] implementada con 18000 válvulas, con un consumo de 160 KW, capaz de realizar 5000 sumas/s, 385 multiplicaciones/s, con 5 millones de soldaduras y un peso de 30 Ton, contra un procesador Intel Core I7 de última generación que permite 177000 MIPS con un consumo de aproximadamente 100 W, es decir, 35 millones de veces más rápido. Para hacer la misma cantidad de sumas por segundo se requerirían 5 GW con la ENIAC; considerar que la capacidad actual instalada en argentina es de 30 GW.

[5] "Electronic Numerical Integrator and Computer" (Integrador numérico y computadora electróica): Considerada la primera computadora electrónica de propósito general. Era "Touring-complete", digital y podía resolver una gran cantidad de problemas numéricos al ser reprogramable.

2.1.3. Silicio, dispositivos semiconductores y transistores

Los semiconductores son materiales, que como bien indica su nombre, no son del todo conductores. En ellos, la capacidad de conducir una corriente eléctrica puede ser manipulada de diversas maneras. En 1931 Wolfgang Pauli —quien en 1945 fue premiado con el Nobel de Física— enunció: "Uno no debería trabajar en semiconductores, eso es un lío deleznable, quién sabe si realmente existen". Nada más alejado de la realidad que hoy nos rodea. Los materiales semiconductores permitieron la creación de los transistores y posteriormente los circuitos integrados, dispositivos que iniciaron la revolución digital. Los primeros dispositivos fueron fabricados sobre germanio y arsenurio de galio. Incluso, el primer circuito integrado, fue realizado en germanio. Pero fue el silicio el material que realmente revolucionó la industria, por su alta disponibilidad en la naturaleza y relativa fácil manipulación para la fabricación de dispositivos semiconductores en circuitos integrados. Puede afirmarse que el silicio es uno de los materiales mejor conocidos por el ser humano. Hace más de 50 años que se lo estudia en detalle para mejorar la tecnología, logrando importantes avances. También puede afirmarse que los transistores hoy en día conforman el bien más abundante en el mundo. Una comparativa del año 2012 muestra que durante año se produjeron en el órden de 10^{17} granos de arroz en la tierra, mientras que en el mismo período en se produjeron en el órden de 10^{19} transistores, según declaraciones de la *Semiconductor Industry Asociation* de los Estados Unidos de América. La clave de semejante número en la producción son los circuitos integrados.

2.1.4. Circuitos integrados y tencología CMOS

En la figura 2.3 puede verse el primer circuito integrado de la historia. Medía aproximadamente media pulgada de ancho e implementaba dos transistores montados en una barra de germanio. Nace así el concepto del *chip* o *microchip*: en inglés, *chip* significa "corte o fracción pequeño de un material duro" directamente relacionado con la técnica de fabricación de circuitos integrados donde, a grandes rasgos, una barra cilíndrica de cristal de silicio puro, es cortada en muy delgadas láminas en forma de discos, sobre las cuales se "imprimen" los circuitos integrados, repitiendo el mismo patrón múltiples veces en un mismo disco para finalmente cortar ese disco en diminutos fragmentos que contienen el diseño. La evolución de las técnicas de fabricación permitieron integrar en un mismo *chip* de silicio más de un dispositivo, permitiendo implementar circuitos relativamente complejos en una pequeña área de silicio, disminuyendo así los riesgos de fallas por interconexión entre dispositivos. A su vez, las técnicas de fabricación evolucionaron permitiendo escalar el tamaño de los dispositivos fabricados incrementando la cuenta de dispositivos integrados por unidad de área. Como consecuencia buscada de disminuir el tamaño de los dispositivos, se logró aumentar la velocidad máxima de conmutación que los mismos pueden lograr, redundando en generar lógica cada vez más compleja y rápida en un mismo *chip*. En simultáneo con estos avances se comenzaron a fabricar transistores

MOSFET[6], transistores de efecto de campo eléctrico, que como gran ventaja sobre los transistores bipolares de juntura, evitaban la disipación de potencia al mantenerse en un estado definido (encendido o apagado), es decir que sólo disipaban potencia al cambiar de estado. Es así que en el año 1978 aparece la tecnología CMOS[7] la cual permitió elevar el nivel de integración en forma masiva, manteniendo bajos niveles de disipación de potencia. Debido a la relativa sencillez geométrica del diseño de los transistores MOS, se pueden reutilizar diseños escalándolos para las nuevas generaciones de tecnología. El 19 de abril de 1965 Gordon Moore[8], cofundador de Intel Corporation[9], estableció de forma empírica que la cantidad de dispositivos integrados en un circuito integrado se duplicaría cada año. Más tarde, en 1975, modificó su propia ley al corroborar que el ritmo bajaría, y que la capacidad de integración no se duplicaría cada 12 meses sino cada 24 meses aproximadamente. Esta progresión de crecimiento exponencial, duplicar la capacidad de los circuitos integrados cada dos años, es lo que se denomina ley de Moore. Sin embargo, en 2007 el propio Moore determinó una fecha de caducidad: "Mi ley dejará de cumplirse dentro de 10 o 15 años", no obstante también aseveró que una nueva tecnología vendrá a suplir a la actual. El cumplimiento se ha podido constatar hasta la actualidad.

La ley de Moore, no es una ley en el sentido científico, sino más bien una observación del ritmo de avance de la industria de aquellos momentos. Al momento de publicar esas declaraciones, Moore trabajaba en los Laboratorios de Fairchild Semiconductor, donde trabajaba junto a Robert Noyce. Ellos fueron los fundadores de Intel en 1968. El ritmo de crecimiento de la insdustria de los semiconductores dió lugar así, entre otras cosas, a la creación de los microprocesadores.

2.1.5. Microprocesadores

Los microprocesadores surgen como consecuencia del alto nivel de integración y complejidad que la tencología de circuitos integrados permitieron alcanzar. Una computadora digital, podía ser construída en uno o algunos pocos *chips* de circuitos integrados. Los primeros diseños de microprocesadores propiamente dichos datan de fines de los años

[6] "Metal-Oxide-Semiconductor Field Effect Transistor" (Transistor de efecto de campo Metal-Óxido-Semiconductor): Es un tipo de transistor fabricado en silicio, donde se genera una estructura que superpone una capa metálica sobre un óxido sobre material semiconductor, generando una estructura capacitiva, con la habilidad de manejar la presencia de un canal conductivo al inducir un campo eléctrico en el material semiconductor. Éste tipo de transistores presenta diversas ventajas frente a los transistores bipolares de juntura para el funcionamiento en circuitos digitales.

[7] "CMOS": técnica de diseño de circuitos lógicos, que utiliza sólo transistores MOSFET-N y MOSFET-P para implementar cualquier función lógica.

[8] Gordon Earle Moore (Nacido el 3 de Enero de 1929): co-fundador de Intel Corporation, es el autor de la "Ley de Moore".

[9] Intel Corporation: fundada el 18 de Julio de 1968 por Goordon Moore y Robert Noyce, es hoy en día la empresea que hace punta en el diseño de circuitos integrados y tecnologías de fabricación. En sus comienzos, su principal negocio fue el desarrollo de circuitos de memorias SRAM y DRAM, pero a pesar de ser la empresa que creó el primer microprocesador comercial en 1971, fué luego de la aparición de las "PC" que el diseño de microprocesadores se volvió su principal negocio.

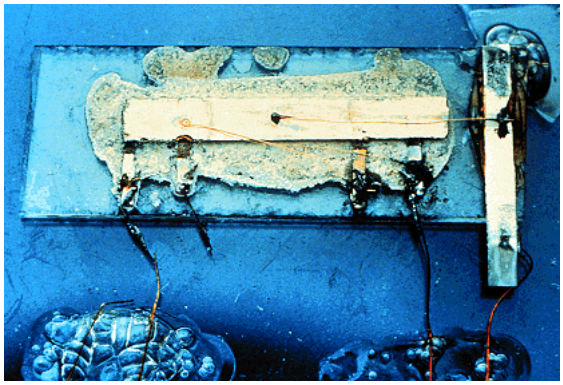


Figura 2.3: Primer circuito integrado presentado por Texas Instruments, Inc. el 12 de Septiembre de 1958.

60. Dentro de este contexto aparece el término *Central Processing Unit* (Unidad Central de Procesamiento) o CPU, que es la parte de la máquina encargada de tomar los datos de entrada y las intrucciones y generar los resultados, dentro del esquema del sistema de procesamiento planteado al principio de éste capítulo. El microprocesador integra además otros componentes y funcionalidades y se vale de ciertos periféricos que pueden estar o no integrados en el mismo circuito, como por ejemplo, memorias de sólo lectura (*Read Only Memory* o ROM) y memorias de acceso aleatorio (Random Access Memories o RAM) y dispositivos de entrada/salida (*I/O, input output devices*. El diseño de arquitecturas de microprocesarores, como todo proceso de innovación tecnológica, no estuvo exento de discusiones y distintas vertientes y prácticas a seguir. Los primeros diseños de los que se tenga registro en órden cronológico, fueron:

2.1.5.1. CADC

En 1968, la armada de los Estados Unidos de América le encomienda a la empresa Garret AiResearch[10] la producción de una computadora digital que pudiera competir con los sistemas electromecánicos que estaban bajo desarrollo para los sistemas de control de vuelo del nuevo avión de combate F-14 Tomcat. El diseño fue completado en 1970 y utilizaba un conjunto de chips (*chipset*) MOS como CPU, siendo aproximadamente 20 veces más pequeño que su contraparte electromecánica y a su vez, mucho más confiable. Este diseño fue utilizado en todos los primeros modelos de Tomcat. La armada se rehusó a publicar el diseño hasta 1997, es por eso que la CADC (Central Air Data Computer) y su *chipset* asociado MP944 no son muy conocidos y no son considerados en la mayoría de la bibliografía reelevante y es a partir de ese momento en que son reconocidos como los primeros diseños válidos de microprocesadores, y no sólo eso, si no que también son

[10]Garret AiResearch: fundada en 1936 en Los Ángeles por John Clifford "Cliff" Garret, fue una compañía dedicada al desarrollo de tecnologías relacionadas a la industria militar aérea.

Figura 2.4: Grumman F14 Tomcat, avión de combate de la US Navy cuyos sistemas de vuelo eran controlados por la CADC.

la primera especie de *Digital Signal Processor* (Procesador de Señales Digitales) o DSP[11] puesto que además de las funciones de microprocesador, implementaba la funcionalidad de medir variables como altitud, velocidad vertical y número de *mach* a partir de datos de sensores pitot, presión estática y temperatura. Los responsables del diseño fueron Ray Holt y Steve Geller.

2.1.5.2. Four-Phase Systems AL1

Un diseño de Lee Boysel que data del año 1969 dentro de Texas Instruments[12], era un componente de una implementación en partes de un microprocesador de 24 bits, compuesto por tres chips iguales: AL1. Durante un juicio por patentes, se demostró que una sóla AL1 junto con memorias ROM y RAM e I/O era capaz de funcionar como CPU.

2.1.5.3. Pico/General Instruments

En 1971, Pico y General Instruments (GI) colaboraron en el diseño de circuitos integrados con el objetivo de fabricar una implementación de un diseño de *chip* único para la calculadora Monroe/Litton Royal Digital III Calcultator. Integraba en el mismo *chip*, memoria ROM y RAM llamado PICO1/GI250. Pico era un emprendimiento de cinco ingenieros de diseño de GI, que tenían la visión de crear esta arquitectura en un único *chip*. Contaban con experiencia previa en diseños de *chipsets* tanto de GI como de Marconi-Elliot. Algunos de ellos habían trabajado para Elliot Automation para crear una computadora de 8 bits en tecnología MOS, y habían ayudado a establecer un laboratorio

[11] "Digital Signal Processor" (Procesaro de señales digitales): es un tipo de microprocesador especializado en el tratamiento digital de señales

[12] Nota sobre TI

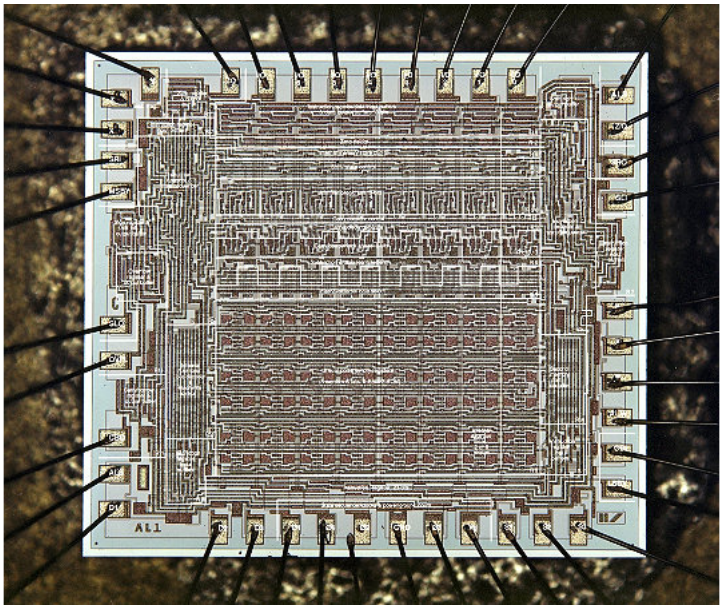


Figura 2.5: Imágen microscópica de la AL1 desarrollada por Four-Phase Systems Inc..

de investigación en tecnologías MOS en Escocia en el año 1967. El mercado de las calculadoras estaba en pleno auge, con lo cuál estos diseños supusieron un éxito comercial para Pico y GI. GI, por su parte continuó con la innovación en microprocesadores y microcontroladores y la división GI Microelectronics se convirtió en 1987 en Microchip PIC Microcontroller.

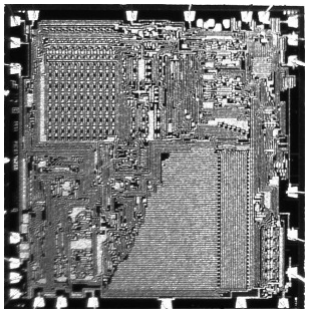


Figura 2.6: Imágen microscópica del PICO1/GI250- Desarrollo colaborativo entre Pico y General Instruments de *chip* único para la Monroe/Litton Royal Digital III Calcultator.

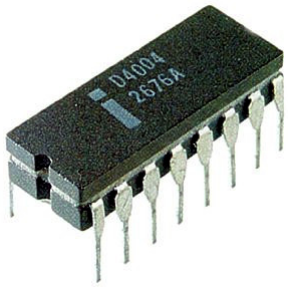


Figura 2.7: Encapsulado del Intel 4004; que es considerado el primer microprocesador comercial en ser introducido en el mercado.

2.1.5.4. Intel 4004

El Intel 4004 es generalmente el reconocido como el primer microprocesador comercial disponible en el mercado. El primer anuncio respecto de este dispositivo data del 15 de Noviembre de 1971, en la publicación Electronic News. El diseño nace a partir del requerimiento de una compañía japonesa fabricante de calculadoras llamado Busicom, que le encomienda a Intel la tarea de desarrollar un *chipset* para calculadoras de escritorio de alto rendimiento. El diseño original requerido por Busicom especificaba un *chipset* compuesto por 7 *chips* diferentes para la realización de una CPU de propósito específico cuyo programa estuviera almacenado en una ROM y sus datos guardados memoria de lectura y escritura implementada con registros de desplazamiento. Intel le asignó el proyecto a Ted Hoff, quien propuso simplificar el diseño utilizando almacenamiento en memoria RAM dinámica y una arquitectura de CPU de propósito general. El diseño propuesto por Hoff implementaba la solución en 4 *chips*: uno de ROM para almacenar el programa, uno de RAM dinámica para almacenar los datos, un dispositivo sencillo de I/O y una CPU de 4 bits. A pesar de no ser específicamente un diseñador de circuitos integrados, el supuso que se podía integrar todo el CPU en un único *chip*. Estas especificaciones surgieron de la interacción de Hoff con un empleado a su cargo, el ingeniero de software llamado Stanley Mazor y un ingeniero de Busicom llamado Masatoshi Shima. Este proyecto se llamó MCS-4 y no fué hasta que Intel contratara al italiano Federico Faggin que empezó a tomar su forma definitiva. Faggin venía de desarollar en 1968 en Fairchild Semiconductor una teconología de compuertas de silicio (SGT o *Silicon Gate Technology*), técnica que se sigue aplicando hoy en día. Faggin también era responsable del desarrollo de la técnica llamada *Random Logic* que permitió la síntesis de descripciones complejas de lógica en hardware sencillo, como compuertas AND y OR. En el momento en el que se estaba desarrollando el proyecto MCS-4, el responsable del área de diseño de MOS de Intel era Leslie L. Vadász, cuya atención estaba enfocada en el mercado de memorias. Fue gracias a esto que le otorgó el liderazgo del proyecto MCS-4 a Faggin, quien fue finalmente el responsable de llevar el proyecto hasta la concepción final del 4004 gracias a la aplicación de las técnicas de diseño y fabricación antes mencionadas. Las primeras unidades del 4004 fueron entregadas a Busicom en Marzo de 1971.

2.2. Evolución hacia las arquitecturas modernas

El concepto moderno de las arquitecturas se basa en las máquinas que almacenan tanto el programa que se ejecuta, como los datos que se manjan en memoria de lectura-escritura (*Stored-Program digital computer*). Esto las diferencia de las primeras implementaciones de los años 40, tales como Colossus[13] y la ENIAC.
En esta sección se estudian las distintas clasificaciones de arquitecturas existentes hoy en día. Luego se detallarán diversas arquitecturas existentes y se las enmarcará dentro de dichas clasificaciones.

2.2.1. Clasificación de las arquitecturas según el acceso a la memoria de instrucciones y datos

Las instrucciones que ejecuta el procesador, al igual que los datos del programa en ejecución, deben residir en la memoria. De este hecho se desprende que existen —al menos— dos tipos de memorias: de instrucciones, y de datos. El acceso a estas dos memorias puede realizarse a través del mismo *bus*[14] o con *buses* separados. Los nombres que la historia les ha provisto a estos dos posibles enfoques, son *von Neumann* para las arquitecturas de *bus* único y *Harvard* para aquellas de *buses* separados.

2.2.1.1. Arquitectura von Neumann

También conocida como modelo de von Neumann y arquitectura Princeton, este tipo de arquitectura debe su nombre a John von Neumann[15], quién en el año 1945 describió en el documento inconcluso, cuya carátula puede verse en la figura 2.8 y conocido como *First Draft of a Report on the EDVAC*[1], una arquitectura de computadora electrónica digital para ser implementada con válvulas de vacío. Ésta fue la primera publicación dónde se describe el diseño lógico de una computadora que utilizase el concepto de *stored-program* o programa almacenado. Este diseño, tal como fue llamado por el propio von Neumann, *very high speed automatic digital computing system* o sistema automático digital de cómputo de muy alta velocidad, estaba dividido en 6 componentes:

- CA: *central arithmetic* o aritmética central
- CC: *central control* o control central
- M: *memory* o memoria

[13] Colossus era el nombre de una serie de computadoras desarrolladas por los británicos para desencriptar las comunicaciones enemigas durante la segunda guerra mundial. Implementadas con válvulas termoiónicas, resolvían mediante lógica booleana operaciones de cálculo.
[14] Bus: definir bus
[15] Nota sobre el chango

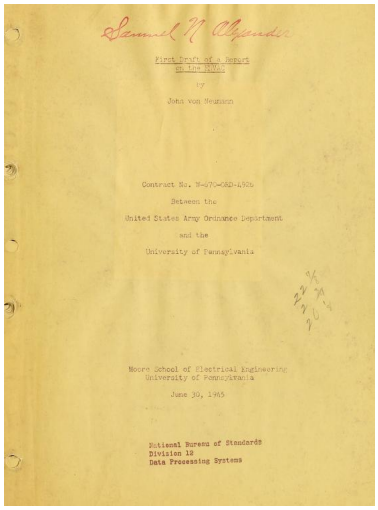


Figura 2.8: Carátula del escrito de von Neumann First Draft of a Report on the EDVAC.

- I: *input* o entrada
- O: *output* o salida
- R: *external memory* o memoria externa

La memoria almacena tanto números (datos) como órdenes (instrucciones). La aritmética central podía realizar sumas, restas, multiplicaciones, divisiones y raices cuadradas. Otras operaciones matemáticas, como logaritmos y funciones trigonométricas se debían realizar utilizando tablas de búsqueda e interpolaciones, posiblemente bicuadráticas. Una decisión de diseño establecida en el documento estableció que multiplicaciones y divisiones podían realizarse mediante tablas logarítmicas, pero para poder mantener dichas tablas pequeñas debería utilizarse interpolaciones, lo cuál a su vez necesita de multiplicaciones, aunque de menor precisión.
Los números se represetaban mediante notación binaria. Estimó que 27 dígitos binarios[16] deberían ser suficientes, pero redondéo a 30 dígitos, más uno de signo y otro para diferenciar los números de las órdenes, resultando en palabras de 32 dígitos binarios. La aritmética utilizada era complemento a dos para simplificar la operación de resta. Para la multiplicación y la división, propuso ubicar el punto binario luego del bit de signo, lo que implica que el dominio de los operandos y resultados está en el rango -1 a 1, y por lo tanto, los datos y resultados de los programas deberían ser escalados acordemente.
En cuanto al diseño de los circuitos, estableció que deberían utilizarse válvulas de vacío, dejando de lado los relé, dada a la mayor velocidad provista por las primeras. Otras sugerencias involucraban mantener al sistema de cómputo lo más sencillo posible, evitando

[16]En el documento se hace meción a dígitos binarios y no a *bits*, término que fue acuñado en 1948 por Claude Shannon

cualquier tipo de optimización de performance mediante la superposición de operaciones. Las operaciones aritméticas debían realizarse de a un dígito binario a la vez. Estimó el tiempo de la suma de dos dígitos binarios en un microsegundo, por lo tanto la multiplicación de dos números representados por 30 dígitos binarios deberí realizarse en aproximadamente 30^{20} microsegundos; lo cuál era mucho más rápido que el tiempo de cualquier dispositivo de cálculo del momento. El diseño estaba basado en lo que von Neumann llamó "elemento E", basándose en el modelo biológico de las neuronas, pero implementado de forma digital planteando que podrían ser fabricados mediante una o dos válvulas. En términos modernos, la forma más sencilla del "elemento E" es una compuerta *AND* de dos entradas, con una de sus entradas invertidas, llamada "entrada de inhibición". Al agregar más entradas a este dispositivo, se establecía un nivel de umbral, el cual al ser superado por la suma de una determinada cantidad de entradas generaba una salida en tanto y en cuanto no se excitara la entrada de inhibición. También estableció que dichos elementos de múltiples entradas podían ser construidos utilizando combinaciones de la versión elemental, pero recomendaba implementarlos por completo utilizando así menos válvulas de vacío con el objetivo de lograr circuitos más sencillos y rápidos, principio que hoy en día se mantiene vigente.
Toda función lógica arbitraria, podía implentarse, entonces, a partir de dichos "elementos E". Demostró en este trabajo, cómo implementar circuitos que implementaban las funciones aritméticas, así como elementos de memoria y circuitos de control, sin referirse en ningún momento al término de "lógica binaria".
Los ciruitos debían ser sincrónicos, obteniendo la señal de reloj a partir de un circuito oscilador implementado con válvulas y posiblemente controlado mediante cristales osciladores. Los diagramas lógicos incluían los tiempos de demora representados mediante una flecha. Estimó que la velocidad a la que se movía un pulso eléctrico en un cable era de $300 mts/microsegundo$, por lo tanto no debería generar problemas hasta obtener velocidades de reloj de $100 MHz$. Tambíen menciona pero no desarrolla la necesidad de contar con mecanismos de detección y correción.
En cuanto a la memoria, que es donde entra la clasificación discutida en esta sección, von Neumann estableció que la memoria es uniforme, conteniendo tanto los números (datos) como las órdenes (instrucciones). Citando su trabajo:

> El dispositivo requiere una memoria considerable. A pesar de que pareciese que distintas partes de la memoria tiene que realizar funciones que difieren en su naturaleza y considerablemente en su propósito, resulta tentador tratar a toda la memoria con un sólo órgano, y hacer que sus partes sean tan intercambiables como sea posible para todas las funciones que deban realizar. [1, sección, 2.5]

> Lás órdenes recibidas por CC vienen de M, es decir, el mismo lugar donde se almacenan los datos numéricos.[1, sección, 14.0].

Él concluyó que la memoria sería el subsistema más grande y abarcativo de la máquina. Basándose en diversos problemas matemáticos, incluyendo la resolución de ecuaciones diferenciales parciales y ordinarias, ordenamientos y experimentos probabilísticos, estimó que se necesitaba espacio para almacenar 8192 palabras de 32 dígitos binarios para los datos, y algunos cientos de palabras para almacenar las órdenes. Para la implementación de memoria, propuso dos tipos de memoria rápida, *delay line memory* e *iconoscope*. Con estas implementaciones planteó que la memoria debía ser direccionable por palabras y para sortear los problemas de demora asociados a la lectura de estas memorias, organizó las mismas en 256 conjuntos de 1024 dígitos binarios, o sea 32 palabras, logrando así direccionar los conjuntos con 8 dígitos binarios y las palabras con 5 utilizando, entonces, 13 dígitos binarios en total para el direccionamiento completo.
En su trabajo también estableció el formato de las órdenes, al cual llamó "código". Los tipos de órdenes incuyeron las operaciones aritméticas básicas, así como el movimiento de palabras entre CA y M (análogas a las instrucciones *load* y *store* de hoy en día que veremos más adelante), una órden que elegía entre dos números basado en el signo del resultado de una operación previa (análoga a una instrucción *branch*), órdenes para controlar la entrada y salida de datos y para indicarle a la CC que debía tomar instrucciones desde otra sección de M (análoga a un *jump*). Determinó, asimismo, la cantidad de dígitos binarios necesarios para necesarios para los distintos tipos de órdenes, sugirió lo que llamó "órdenes inmediatas", donde la siguiente palabra se trata del operando (lo cual también tiene una anagolía con ciertos conjuntos de instrucciones actuales) y dejó planteado si era deseable dejar dígitos sin especificar para propósitos futuros y mayor direccionamiento de memoria.

2.2.1.2. Arquitectura Harvard

Este tipo de arquitectura, ve sus orígenes en un desarrollo propuesto por el Dr. Howard Aiken[17] en la universidad de Harvard en el año 1937 basado en el trabajo llamado *Analytical Engine*[18] de Charles Babage[19]. El desarrollo fue llevado al cabo por IBM[20] y se trataba de una máquina de cálculo llamada *Automatic Sequence Controlled Calculator - Harvard Mark I* entregado a la universidad el 24 de agosto de 1944. A diferencia de la máquina propuesta (posteriormente) por von Neumann, ésta era una máquina electromecánica construida con llaves, relés, engranajes y embreagues. Constaba de 765000 componentes electromecánicos, cientos de millas de cableado, un peso de 4.5 Ton. y un consumo de potencia de 3.7 kW. Para el ingreso de datos, contaba con 60 conjuntos de 24 llaves rotativas que permitían ingresar números decimales. Podía almacenar hast 72 números de hasta 23 dígitos decimales. En un segundo era capaz de realizar 3 sumas o restas. La multiplicación tardaba 6 segundos y una división 13.5 segundos. El calculo de funciones logarítmicas o trigonométricas utilizaba más de un minuto. Las instrucciones

[17]Nota sobre Aiken
[18]Nota sobre Analytical Engine
[19]Nota sobre el tipo
[20]Nota sobre IBM

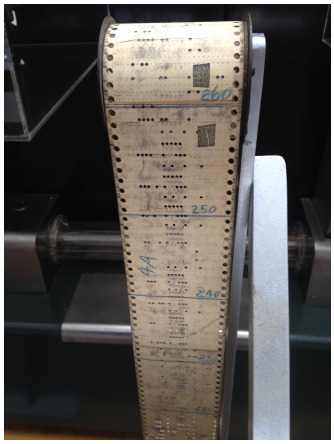


Figura 2.9: Cinta de papel perforado con instrucciones de la Mark I.

eran ingresadas a través de cintas perforadas de 24 canales (en la figura 2.9 se puede ver una cinta con instrucciones de esta máquina). Otra cinta podía ser utilizada para el ingreso de datos, pero utilizando un formato distinto al de las instrucciones; estas no podían ser ejecutadas desde los registros de almacenamiento. Esta separación entre datos e instrucciones es lo que se conoce como arquitectura Harvard y que la diferencia de la arquitectura von Neumann. El formato de las instrucciones definía tres campos en ocho canales separados. Cada acumulador, cada conjunto de switches, y los registros asociados a la entrada y salida y las unidades aritméticas tenían asignado un índice único. Estos números eran representados en formato binario en la cinta perforada de control, siendo el primer campo el índice del resultado de la operación, el segundo el orígen del dato para la operación y el tercero el código que representa la operación a realizar. Una particularidad de ésta primera versión es que no tenía instrucción para el salto condicional en el programa. Por lo tanto, los programas complejos debían ser largos y los "loops" se implementaban uniendo físicamente el final de la cinta perforada del programa con el principio.

La Mark I fue sucedida por la Mark II en el año 1948, la Mark III en septiembre de 1949 y la Mark IV en el año 1952, todos estos proyectos fueron de Aiken. La Mark II fue una mejora sobre la Mark I, pero aún basada en componentes electromecánicos. La tercer versión utilizaba en su mayoría componentes electrónicos tales como válvulas de vacío y diodos cristalinos, pero utilizaba cilindros magnéticos para almacenamiento, y relés para la transferencia de datos entre ellos. La cuarta versión fue la primera en ser completamente electrónica, reemplazando los cilindros magnéticos de almacenamiento con memorias de núcle magnético; el tipo de memoria de acceso aleatorio predominante entre 1955 y 1975.

2.2.1.3. Harvard vs. von Neumann y Harvard modificada

Como fue establecido al principio de esta sección, lo que diferencia estas dos arquitecturas es el acceso a los datos y a las instrucciones. La arquitectura von Neumann puede tratar a los datos como instrucciones y viceversa, dado que para dicha arquitectura, la memoria es exactamente la misma, con el mismo tipo de acceso y el mismo direccionamiento, contrariamente a lo que es una arquitectura puramente Harvard, cuyas memorias de datos e instrucciones se acceden por diferentes medios y cuyas direcciones pueden estar sobrepuestas (es decir que la misma dirección de memoria no representa lo mismo si se trata de datos o intrucciones) y por lo tanto, los datos y las intrucciones no pueden ser intercambiados en el tratamiento que se realiza dentro de la unidad de cómputo. Dadas estas condiciones, las principales desventajas son que, la arquitectura von Neumann no puede acceder simultáneamente a datos y a instrucciones y la máquina de Harvard no puede ni leer ni escribir en el espacio de memoria de las instrucciones, denegando por lo tanto cualquier posibilidad de código auto-optimizable o inteligente que reescriba sus instrucciones, así como tambíen la carga de los programas a partir de medios de almacenamiento de datos.
Hoy en día, las máquinas puramente Harvard no son más que una especialidad. Típicamente, en las máquinas modernas, se implementas arquitecturas Harvard modificadas. Estas modificaciones pueden ser de diversos tipos:

2.2.1.3.1. Arquitectura de caché separado Es la modificación más común sobre la arquitectura Harvard, se construye una jerarquía de memoria donde el nivel más cercano al núcleo de procesamiento (típicamente llamado caché de nivel 1) está separado en datos e instrucciones, y unificando estas memorias en un nivel de jerarquía superior. Con esta técnia, se puede relajar el problema de acceder a la memoria de instrucciones como datos, tanto para la lectura como para la escritura al unificar toda la memoria en un solo espacio de direccionamiento, emulando así el comportamiento de una arquitectura de von Neumann, pero manteniendo la ventaja de poder acceder a datos e instrucciones de forma simultánea. Esta característica será transparente para la mayoría de los programadores, excepto aquellos casos en los que es necesario el manejo de técnicas como por ejemplo, la coherencia de caches.

2.2.1.3.2. Acesso a instrucciones como datos Un arquitectura que provea esta solución matiene la naturaleza de espacios de direcciones separados de la arquitectura Harvard, pero incluye operaciones especializadas en acceder al contenido de la memoria de instrucciones como si fueran datos. Como los datos no pueden ser ejecutados driectamente como instrucciones, estas implementaciones no siempre son vistas como Harvard "modificadas".
Esta técnica puede presentarse con dos variantes. Por un lado, el acceso puede ser de sólo lectura, provee la capacidad de copiar contenido embebido en el código cargado en la memoria de instrucciones a la memoria de datos cuando el programa comienza o, mejor

aún, si los datos no van a cambiar (como es el caso de constantes aritméticas o cadenas de texto predefinidas), estos datos que están en la memoria de instrucciones pueden ser directamente leídos en tiempo de ejecución por el programa sin tomar espacio de la memoria de datos (que puede ser escasa en las implementaciones que aplican esta técnica). Por otro lado, dicho acceso puede ser de lectura/escritura, suele darse cuando es necesaria una capacidad de reprogramación de la máquina en cuestión. Pocas computadoras hoy en día están basadas únicamente en ROM para su funcionamiento. Puede ser el caso de algunos microcontroladores que tienen operaciones para escribir en su memoria "Flash" (en la cual se almacenan las instrucciones). Esta capacidad provee medios de acceso para actualizaciones de software/firmware y reemplaza a las EEPROM.

2.2.1.3.3. Ejecución de datos Algunas arquitecturas pueden obtener sus instrucciones desde cualquier segmento de memoria con la limitación de que no podrá acceder simultáneamente para leer instrucciones y datos de un mismo segmento de memoria. Para lograr este comportamiento, se direccionan los buses tanto de datos como de instrucciones sobre distintos segmentos físicos de una misma memoria.

2.2.1.3.4. Características de las arquitecturas Harvard modificadas Tres características pueden ser utilizadas para diferenciar a este tipo de arquitectura de las Harvard puras y de las von Neumann:

2.2.1.3.4.1. Las memorias de datos e instrucciones ocupan diferentes espacios de direccionamiento Para las máquinas puramente Harvard, hay una dirección "cero" de instrucciones en el espacio de direcciones de las instrucciones y una dirección "cero" de datos en el espacio de direcciones de los datos que referencian a un lugares de almacenamiento distintos. En contraste, las máquinas von Neumann y algunas Harvard modificadas (como las de caché separado), almacenan tanto datos como instrucciones en un espacio de direcciones unificado, por lo tanto, la dirección "cero" hace referencia a una sóla cosa y lo almacenado allí puede ser la representación binaria de un código de instrucción o de un dato, dependiendo de cómo fue escrito el programa en ejecución. Sin embargo, tal como las máquinas puramente Harvard, las Harvard modificadas con espacios de direcciones separados, pero con instrucciones específicas que permiten leer y escribir las instrucciones como datos, tienen una direcciónes solapadas para los respectivos espacios de direccionamiento, por lo tanto esto no distingue las máquinas puramente Harvard de este último tipo de Harvard modificadas.

2.2.1.3.4.2. Las memorias de datos e instrucciones tienen conexiones físicas separadas hacia el núcleo de procesamiento Este es el motivo principal por el que existen las arquitecturas Harvard (tanto puras como modificadas), y por qué coexisten con las más generales y flexibles von Neumann: los caminos separados físicamente para

los datos y las instrucciones permiten que las instrucciones sean cargadas en la unidad de cómputo al mismo tiempo que los datos, aumentando así el rendimiento. Las máquinas puramente Harvard tienen caminos separados con espacios de direcciones separados. Las máquinas de caché separado proveen este acceso separado para la unidad de procesamiento pero presentan un espacio de direccionamiento unificado al ascender en la jerarquía de memoria. Una máquina von Neumann puede tenera sólamente un espacio de direcciones unificado. Para el punto de vista del programador, una arquitectura de cache separado es tratada de forma equivalente a una von Neumann (al menos hasta el punto en el que el tratamiendo de coherencia de cache se convierta en significativo), como puede ser el caso de código auto-optimizable o la carga de un programa en memoria desde un medio de almacenamiento externo. Otras arquitecturas Harvard modificadas se comportan como Harvard puras en este ámbito.

2.2.1.3.4.3. Las memorias de datos e instrucciones pueden ser accedidas de diferentes formas Las máquinas Harvard puras, al tener distintos espacios de memoria para datos e instrucciones, pueden acceder a dichas memorias de distintas maneras. Puede ser el caso de una arquitectura donde la memoria de instrucciones se direccione con una cantidad de bits distinta a la de datos. Esto no es posible en una arquitectura von Neumann ni en una Harvard modificada de cache separado, dado que ambos espacios de memoria deben estar superpuestos.

2.2.1.4. Aplicaciones de esta clasificación en la actualidad

En áreas específicas donde un memoria cache es prohibitiva (puede ser el caso de los DSP por el determinismo necesario para las diversas operaciones, o de microcontroladores por el bajo costo), pueden darse arquitecturas Harvard puras o von Neumann. En modelos donde el uso es de propósito general, típicamente se utilizan arquitecturas Harvard modificadas con caché separado debido a que tener espacios de direccionamiento separados genera dificultados con ciertos lenguajes de programación de alto nivel que no soportan la noción de tener datos de sólo lectura cargados en un espacio de direccionamiento distinto al de los datos comunes, con la necesidad de utilizar distintas instrucciones para leerlos. El lenguaje de programación “C” soporta este comportamiento a través de extensiones propietarias, aunque hoy en día ya existen estandarizaciones para los llamados procesares embebidos.

2.2.2. Taxonomía de Flynn

La taxonomía de Flynn es una clasificación de las arquitecturas basada en la cantidad de “hilos” de control y de datos que soporta el procesador de forma concurrente. Establecida

	Hilo simple de instrucciones	Hilo múltiple de instrucciones
Hilo simple de datos	SISD	MISD
Hilo múltiple de datos	SIMD	MIMD

Tabla 2.1: Taxonomía de Flynn.

en 1966 por Michael Flynn[21], estableciendo los efectos que pueden surgir de explotar el paralelismo tanto en los hilos de instrucciones como en los de datos. Se puede tener hilos únicos o múltiples para cada uno de ellos. Es estas definiciones, Flynn estableció que una máquina identifica los siguientes conceptos:

- CU: *contro unit* o unidad de control
- PE: *processing element* o elemento de procesamiento
- IS: *instruction stream* o hilo de instrucciones
- DS: *data stream* o hilo de datos

Al combinar hilos sencillos y múltiples de datos e instrucciones se da lugar, entonces, a cuatro tipo de arquitecturas (ver tabla 2.1).

2.2.2.1. SISD: Simple Instruction stream, Simple Data stream

Resultante de combinar un único hilo de control con un único hilo de datos, este tipo de arquitectura no explota ningún paralelismo en una máquina completamente secuencial. Una única CU obtiene un único IS de memoria. La CU genera las señales de control necesarias para excitar a un único PE sobre un único DS, es decir, una operación a la vez. Ver figura 2.10a.
Un ejemplo de esta arquitectura son los procesadores de las antiguas computadoras personales.

2.2.2.2. SIMD: Simple Instruction stream, Multiple Data stream

En este tipo de arquitecturas, sige existiendo un único hilo de control, pero los datos que afectan las instrucciones provienen de distintos hilos de datos. La CU trabaja con las instrucciones de un único IS y general las señales de control necesarias para que mútiples PE procesen los datos provenientes de múltiples DS. Es un tipo de máquina

[21] Michael J. Flynn (Nacido el 20 de Mayo de 1934): es un profesor emérito de la universidad de Stanford que propuso la llamada "Taxonomía de Flynn" para clasificar las arquitecturas de computadoras en el año 1966

que es naturalmente paralelizable. Ver figura 2.10b.
Ejemplos de este tipo de arquitecturas son los procesadores matriciales y las unidades de procesamiento gráficas (GPU). Cabe destacar que los antiguos procesadores de PC, en un determinado momento incorporaron instrucciones específicas con el que lograban este comportamiento, tales como las extensiones multimedia (MMX) que Intel incluyó en su línea de procesadores Pentium a mediados de los años '90.

2.2.2.3. MISD: Múltiple Instruction stream, Simple Data stream

Se trata de arquitecturas en el que diversos hilos de control actuan sobre un único hilo de datos. No se trata de un tipo común de arquitecturas, si no más bien de un tipo específico utilizado en sistemas tolerantes a fallas. Diversos sistemas heterogéneos operan sobre el mismo hilo de datos y deben arribar al mismo resultado. Es decir que diversas CU toman instrucciones de diversos IS controlando múltiples PE que actuan sobre un único DS. Ver figura 2.10c.
Ejemplos de este tipo de máquinas pueden ser las computadoras de control de vuelo de aeronaves y naves espaciales o cohetes; es decir aplicaciones de misión crítica que poseen sistemas redundantes y decisores.

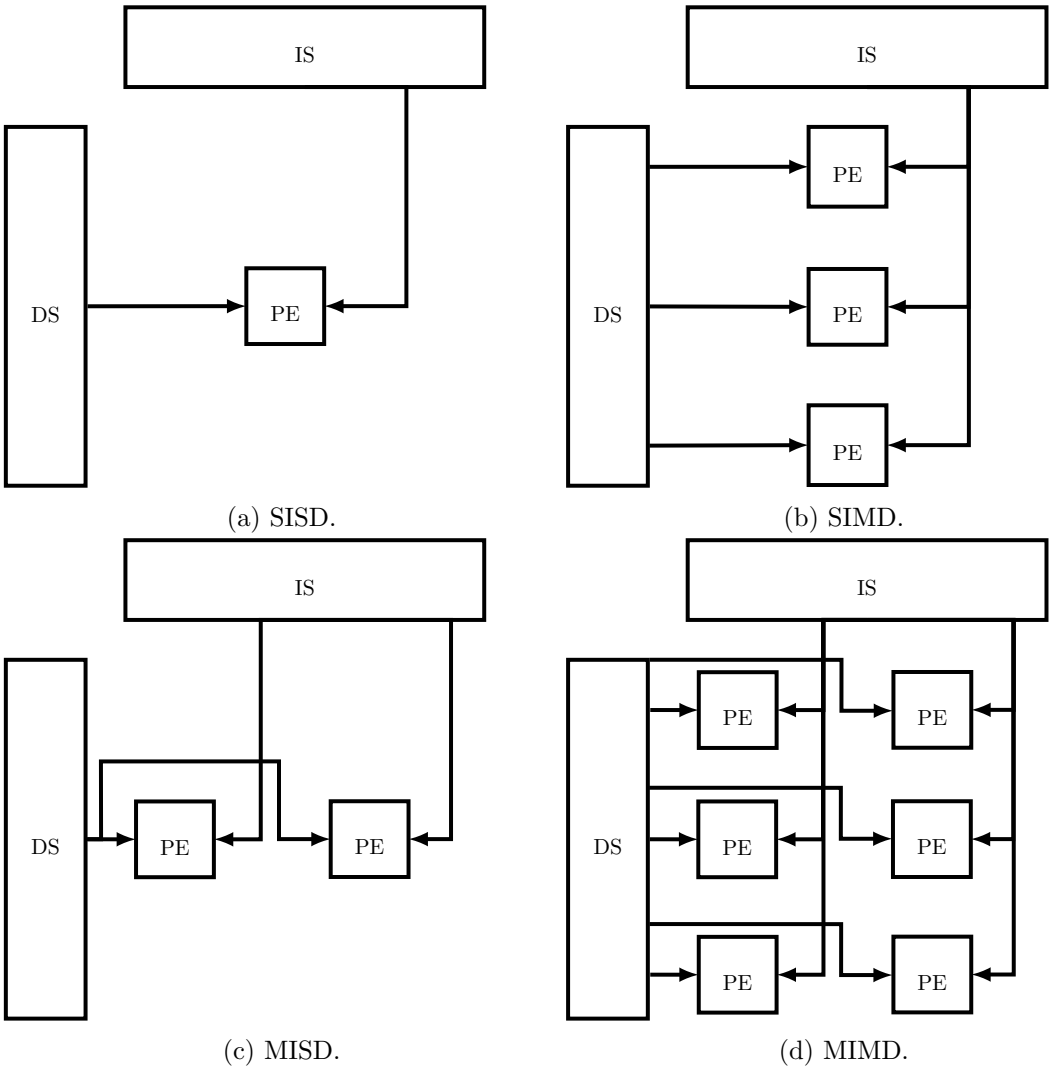
2.2.2.4. MIMD: Multiple Instruction stream, Multiple Data stream

Se trata de arquitecturas donde múltiples procesadores simultáneamente ejecutan diversos hilos de instrucciones sobre diversos hilos de datos. Es decir que diversas CU toman instrucciones de diversos IS para excitar a diversos PE que actúan sobre diversos DS. Ver figura 2.10d.
Los procesadores multinúcleo y los sistemas distribuidos son ejemplos de éste tipo de máquinas, y pueden utilizar tanto un espacio de memoria compartido, así como uno distribuído.

2.2.2.5. Actualización de la taxonomía de Flynn

Con el advenimiento de las arquitecturas multinúcleo —hoy en día la mayoría de las arquitecturas utilizadas en microprocesadores y microcontroladores son multinúcleo— la taxonomía de Flynn ha recibido una actualización para la categoría MIMD, basándose en el programa al que pertenencen los hilos de datos e instrucciones en cuestión. Estos grupos son conocidos como "Single Program, Multiple Data Streams" (SPMD) y "Multiple Programs, Multiple Data Streams" (MPMD).

2.2.2.5.1. SPMD: Single Program, Multiple Data streams Esta clasificación modela múltiples procesadores autónomos que simultánemente ejecutan el mismo pro-



(a) SISD. (b) SIMD.

(c) MISD. (d) MIMD.

Figura 2.10: Diagramas de la taxonomía de Flynn.

grama (pero en punto independientes, a diferencia de la naturaleza secuencial de SIMD) en diversos hilos de datos del programa. En sí, no es sólo la naturaleza de la arquitectura lo que da lugar a este tipo, si no que además involucra al modelo de programación utilizado.

2.2.2.5.2. MPMD: Multiple Programs, Multiple Data streams También se trata de un modelo que involucra tanto a la arquitectura como al diseño del software que se ejecuta. En este caso, múltiples procesadores ejectuan simultáneamente al menos dos programas independientes. Típicamente, estos sistemas eligen un nodo que administra como los demás nodos ejecutan el resto de los programas con diversas estrategias y modelos de programación.

2.2.3. Clasificación de las arquitecturas según el manejo de los datos y cantidad de operandos

Tal como fuese indicado al comienzo del presente capítulo, la unidad de procesamiento, recibe datos e intrucciones y genera resultados. Para poder realizar este trabajo, la arquitectura debe ser capaz de leer de la memoria datos y luego volcar los resultados a la misma. En este proceso, hay diversos mecanismos por el cual las instrucciones pueden obtenera esta información y devolverla luego del procesamiento para un posterior uso en el programa o generar salidas en el sistema. Inherentemente, se definen la cantidad de operandos que pueden tener las instrucciones de la arquitectura.
Es así que las instrucciones de la arquitectura, podrían operar directamente sobre los datos en memoria y escribir los resultados en la misma; hacer uso de estructuras de hardware intermedias para almacenar los datos temporalmente para su procesamiento y posterior almacenamiento o una combinación de estas alternativas.
Las ventajas, desventajas y particualridades que presentará cada una de las variantes deben ser sopesadas con la tecnología de los compiladores utilizados y las posibilidades de optimización que proveen.
Previo a analizar las alternativas de diseño que se presentan en cuanto al manejo de los datos dentro de la unidad de procesamiento, es conveniete estudiar los diversos modos de direccionamiento que podemos observar en las arquitecturas a través de la historia.

2.2.3.1. Modos de direccionamiento

Las instrucciones deben ser capaces de entregarle datos o referencias a los mismos a la unidad de procesamiento. Así mismo, la unidad de procesamiento debe poder enviar los resultados a alguna salida. Estos mecanismos se conocen como modos de direccionamiento. Para el ingreso de datos, estos pueden estar embebidos en el código de la instrucción, pueden ser representados por su dirección en memoria o estar contenidos en algún registro interno. Para la salida, también podremos enviarlos a alguna dirección

de meomoria representada de alguna manera en particular, o sencillamente almacenar el dato en algún registro interno.
No sólo son los datos los que deben ser direccionados para ingresar a la unidad de procesamiento; esta también debe ser capaz de obtener las instrucciones de memoria.
En el caso de las instrucciones, el direccionamiento cobra importancia al momento de analizar las instrucciones que modifican el flujo del programa y rompen con la secuencialidad del mismo, o sea las instrucciones de salto condicional e incondicional. Se reconocen principalmente tres variantes:

- Absoluta o directa: el código de la instrucción lleva embebido directamente la dirección de la instrucción a la que se debe saltar.
- Relativa: el código de la instrucción lleva embebido un desplazamiento respecto de la dirección de la siguiente instrucción a ejecutar, siendo que ese desplazamiento puede ser en ambos sentidos.
- Indirecta: el código de la instrucción lleva embebido el registro que indica a qué dirección debe saltar la ejecución del programa.

Al analizar los posibles modos de direccionamiento de los datos, surgen más alternativas básicas:

- Registro: es el utilizado en las operaciones en las cuales todos los operandos se obtienen y escriben desde registros internos.
- Desplazamiento: en este modo de direccionamiento las instrucciones que lo implementan indican la dirección de memoria a la cual acceder utilizando un como base una dirección contenida en un registro a la cual se le suma un desplazamiento que está embebido en la instrucción. Existen diversas variantes de este modo de direccionamiento, algunos de los cuales utilizan índices para recorrer estructuras de memoria más complejas.
- Inmediato: para este caso, la instrucción posee embebido en el código de la operación el dato.
- Implícito: las instrucciones que implementan este modo de direccionamiento sólamente actúan sobre algún dato específico dentro de algún registro interno.

2.2.3.2. Accumulator

Las arquitecturas Accumulator, o de acumulador, también llamadas máquinas de un operando, se caracterizan por tener un registro particular —típicamente llamado acumulador— en el que los resultados de las operaciones entre dicho registro y el operando indicado en la instrucción se almacena. El operando puede ser obtenido con los diversos modos

de direccionamiento soportado por tal arquitectura.
Esta arquitectura basa su funcionamiento en el hecho de que en general, los programas realizan una secuencia de operaciones sobre los datos. Por lo tanto, si no existiese el acumulador, cada vez que se realiza una operación, se debería guardar el resultado en memoria, para luego volver a obtenerlo para seguir operando. Dado que la tecnología con la que se implementan los registros internos permiten mayor velocidad de transferencia que la memoria principal (más larga y más lenta pero de menor costo), se utiliza este registro especial para explotar la localidad temporal de los datos en el software.

2.2.3.3. Stack

La estructura "stack" o pila, es una estructura lógica que permite ir apilando y desapilando datos. En una arquitectura de tipo stack, existe este tipo de estructura implementada con registros internos y las operaciones se realizan entre los dos elementos superiores de la pila y el resultado se almacena al tope. Los datos utilizados son removidos del tope en el proceso. Es evidente la necesidad de instrucciones específicas (o posibles pseudo-instrucciones) para mover datos entre la pila y la memoria. En este escenario, también existirán diversos modos de direccionamiento para los datos. Típicamente, el conjunto de instrucciones notará las operaciones en RPN (Reverse Polish Notation)[22] y en el código de la operación se elimina la necesidad de explicitar los operandos. Una variación de esta arquitectura puede implementar operaciones entre el tope de pila y otro operando indicado en la instrucción mediante los diversos modos de direccionamiento soportados por tal implementación.
Esta arquitectura tal como la de acumulador también explota la localidad temporal de los datos considerando la secuencialidad de las operaciones sobre los mismos. De hecho, puede considerarse a la arquitectura acumulador como un caso especial de pila con un único registro que opera contra los operandos indicados en la instrucción.

2.2.3.4. Load / Store

También conocidas como "Register / Register" estas arquitecturas implementan múltiples registros internos direccionables y sólo realizan operaciones entre ellos. Además se proveen instrucciones específicas de carga y almacenamiento para mover los datos entre memoria y los registros internos. Dichas instrucciones proveerán los diversos modos de direccionamiento de datos soportados por tal arquitectura.
En implementaciones de este tipo, se resigna por un lado las ventajas que presenta la localidad temporal de los datos en pos de explotar posibilidades del paralelismo a nivel de instrucción, que es tódo un tópico en sí mismo dentro de las optimizaciones posibles de las arquitecturas. Hoy en día, ésta es la elección clásica al momento de desarrollar una arquitectura RISC.

[22] Reverse Polish Notation (Notación Polaca Inversa): DESARROLLAR

2.2.3.5. Register / Memory

Estas arquitecturas, como lo indica su nombre, permiten que las operaciones sean realizadas en o desde memoria, así como también entre registros. Si además se permite que todos los operandos estén en memoria o en registros o en una combinación de ellos, se las llama "Register Plus Memory". También deben proveerse instrucciones que permitan la transferencia de datos de memoria a registros internos y viceversa. Todas las operaciones típicamente se proveerán con diversos modos de direccionamiento.
Claramente de todas las opciones mencionadas, ésta es la más flexible. El costo asociado a esta flexibilidad será el de complejizar el hardware, relegando así oportunidades de optimización del mismo. Típicamente es la elección de las arquitecturas CISC.

2.2.4. Clasificación de las arquitecturas según su conjunto de instrucciones

Esta clasificación quizá sea la más importante en el desarrollo del presente trabajo dado que el conjunto de instrucciones de la arquitectura define por sí mismo el modelo lógico de la misma.
El *Instruction Set Architecture* —o sencillamente ISA— de una arquitectura, puede pertenecer a dos categorías principales: CISC o RISC. Cabe destacar que no son sencillamente categorías, si no más bien se tratan de dos posiciones filosóficas opuestas en cuanto a lo que al diseño de arquitecturas concierne, tal como fuese anticipado al comienzo del capítulo 1. Estas posiciones pueden diferenciarse en cuanto al lugar donde está puesto el énfasis: la filosofía CISC enfatiza el poder de procesamiento del hardware y la filosofía RISC el del software a través de un hardware básico y sencillo, pero rápido.

2.2.4.1. CISC: Complex Instruction Set Computer

Dado que en la filosofía CISC, el enfoque está centrado en el poder del hardware de la arquitectura, el objetivo se centra en crear un conjunto de instrucciones poderoso y abarcativo. Tal conjunto de instrucciones permite realizar operaciones complejas a bajo nivel, es decir, directamente en el lenguaje de la máquina. Es así que el lenguaje debe dar soporte a múltiples operaciones sobre distintos tipos de datos con diversos modos de direccionamiento; con la consecuente complejidad del hardware subyacente para soportar dichas operaciones. Dada la muliplicidad de operaciones, tipos de datos y modos de direccionamiento que debe soportar este tipo de arquitecturas, su lenguaje será un reflejo de esta complejidad, es por eso que típicamente, el ancho de palabra natural de la arquitectura no es suficiente para incluir todas las instrucciones que soporta, en consecuencia, suelen utilizarse formatos de instrucciones en los que una instrucción puede estar distribuida a lo largo de múltiples palabras; y esto suma a la complejidad del hardware necesario para decodificar dicho tipo de instrucciones. Típicamente, en un hardware de este tipo las instrucciones pueden demandar varios ciclos de reloj para

generar un resultado, pero siendo este resultado producto de un trabajo complejo sobre los datos. En consecuencia, el código a nivel *assembler*[23] generado por un compilador de un lenguaje de más alto nivel, tenderá a ser compacto dado el poder de cómputo elevado que tienen las instrucciones de este tipo de arquitecturas.

2.2.4.2. RISC: Reduced Instruction Set Computer

Cómo fuese mencionado antes, la filosofía RISC pone el enfoque en el poder del software, dejando que el hardware sólo se encargue de operaciones básicas sobre los datos. El conjunto de instrucciones, en este caso, será reducido y soportará operaciones sencillas sobre tipos de datos compatibles entre sí y en general se contará con algunos pocos modos de direccionamiento. Es así que el hardware necesario para implementar las operaciones de una ISA RISC será sencillo, transfiriendo así la responsabilidad de la generación de resultados de operaciones más complejas al software. Es por esto que los programas en lenguaje de máquina tenderán a ser más extensos que su contraparte CISC. Es aquí que cobra importancia la tecnología de los compiladores para poder soportar optimizaciones razonables en este tipo de arquitecturas. El beneficio que trae la implementación de este tipo de arquitecturas radica en la sencillez del hardware, algo que quizá a simple vista parezca trivial, pero que en la práctica acarrea consecuencias deseables para la aplicación de mejoras en el hardware.

2.2.5. Elecciones en el diseño propuesto

Los objetivos prácitcos del presente trabajo de tésis incluyen la necesidad de generar un IP Core con un núcleo de procesamiento sencillo y configurable. Será entonces la sencillez un parámetro con elevado peso específico al momento de tomar decisiones de diseño; es por ésto que la primera, consta en seguir la filosofía RISC para el presente desarrollo.
Tal como marcan las tendencias actuales en el campo de las arquitecturas RISC, se elige el formato Load / Store, siendo ésta la segunda decisión de diseño.
Es oportuno mencionar que el diseño que se propondrá no exigirá definir la cantidad de hilos de instrucciones que soportará esta arquitectura; dejando así librado a la implementación en particular la posibilidad de incluír múltiples núcleos, pero no se soportará el trabajo sobre múltiples hilos de datos por parte de un único elemento de procesamiento; evidenciando que con el mismo IP Core podrán implementarse tanto arquitecturas SISD como MIMD.
Siendo la configurabilidad otro aspecto importante, se eligirá el formato de arquitectura Harvard en lo que al manejo de memoria concierne; una vez más, dejando a merced de la implementación particular las decisiones que conciernen a la jerarquía de memorias; flexibilizando así las posibilidades que pueda entregar el IP Core pudiendo implementar

[23] Assembler o lenguaje ensamblador: es el lenguaje de máquina de una determinada arquitectura.

tanto arquitecturas Harvard puras, como modificadas si fuese necesario aprovechar las bondades del modelo de von Neumann.
Tal como fue indicado en 2.2.4.2, la filosofía RISC pone el énfasis en las posibilidades del software y la tecnología de los compiladores, pero esto no implica que se relaje la complejidad del diseño de la arquitectura en sí, sino que más bien obliga a generar una arquitectura sencilla a la que se le pueda aplicar la mayor cantidad de optimizaciones posibles. No debe confundirse en este punto, sencillez en el diseño con sencillez en el trabajo de diseñar, por el contrario, obtener una arquitectura sencilla obliga a trabajar más en el diseño.

2.3. Desempeño; mediciones y optimizaciones

Es necesario en este contexto, introducir una medida del desempeño o performance de una máquina que procesa datos. Debe notarse que el desempeño de una máquina varía según el contexto de los programas en ejecución. No es la misma "clase" de performance la esperada para un servidor de bases de datos, que para una computadora de escritorio. Es así que queda en evidencia que existen diversos tipos de computadoras, con distintos objetivos, por más de que todas sean consideradas de propósito general. Podemos distinguir en un primer nivel de abstracción, tres grandes grupos:

- computadoras personales o estaciones de trabajo,
- servidores,
- nodos de clústeres,
- sistemas embebidos,
- dispositivos personales móviles.

Claramente, el trabajo que se espera, realice cada una de estas variantes, es distinto. Las primeras, deberían ser las más generalistas, mientras que los servidores pueden especializarse en diversas áreas, como por ejemplo, almacenamiento masivo de archivos, servidor web, servidor de base de datos o hypervisor[24], o ya como otro conjunto de este universo, los nodos de clústeres de cómputo; sistemas que interconentan gran cantidad de equipos para proveer infraestructura redunante y de alta disponibilidad para las modernas plataformas de servicios en la nube, que aunque pueden implementarse con computadoras personales o servidores, también existen versiones especializadas. Los sistemas embebidos son los mas dificiles de definir de forma general, puesto que la particularidad del sistema —en sentido amplio— definirá las características del trabajo que debe realizar, como puede ser detección de huellas dactilares en un sistema de control de acceso, tarea que demandará alta capacidad de cómputo para el procesamiento de imágenes, o la sencilla

[24]Definir hypervisor

gestión del HMI[25] de una máquina industrial, así como los procesadores que controlan electrodomésticos o los sistemas de un automóvil o aeronave; y también combinaciones de estas funciones y muchas otras más, como pueden encontrarse en un teléfono celular moderno. Dentro del universo de los sistemas embebidos, el de los dispositivos móviles es todo un área por sí sola, es por ésto que se los clasifica aparte como dispositivos personales móviles. El principal motivo para diferenciar estos últimos —muchas veces considerados sistemas embebidos— es la posibilidad de ejecutar programas desarrollados por terceras partes, y no sólamente el firmware que da el fabricante.
No obstante resulta imprescindible contar con un método preciso y definido para poder comparar arquitecturas. Sin embargo, no deben perderse de vista los objetivos principales que persiguen los diseñadores de arquitecturas y las relaciones de compromiso que pueden resumirse como maximizar la performance minimizando el costo y el consumo energético. Los motivos para minimizar el costo son hartos conocidos por la humanidad y el sistema económico que en líneas generales rige en el mundo, pero en la cualquiera de los tipos de computadoras debe minimizar el consumo energético, por dos principales motivos: el costo operativo —una vez más motivación económica— y maximizar la autonomía en el caso de dispositivos móviles. Aunque resumidos aquí, son temas extensos que se abordarán a lo largo del trabajo.
A lo largo de la sección se irán mencionando distintas técnicas de optimización aplicables al diseño de arquitecturas que serán desarrolladas con mayor profundidad en la sección 2.5.

2.3.1. Ecuación de desempeño

La métrica definida de forma histórica para tal fin es el tiempo de ejecución que emplea la arquitectura para un determinado programa o tarea. Se define entonces el desempeño (D) de la máquina como la inversa del tiempo de ejecución (t_{ex}) de un determinado programa.

$$D = \frac{1}{t_{\text{ex}}} \tag{2.1}$$

Es de especial interés definir en éste punto al tiempo de ejecución, dado que existen diversas interpretaciones del mismo. La definición más directa es el del tiempo del reloj de pared, también llamado tiempo de respuesta o tiempo transcurrido; en otras palabras, la latencia necesaria para completar la tarea incluyendo las demoras por acceso a disco y memoria, actividades de entrada y salida de datos y el *overhead* asociado al sistema operativo. El problema que existe con ésta definición directa del tiempo de ejecución es que típicamente en una máquina el proceso que estaremos midiendo no será el único proceso que se está ejecutando en un determinado intervalo de tiempo; lo más probable es que exista un sistema operativo que está gestionando múltiples procesos de usuario además de los propios que entre otras cosas será responsable de gestionar el tiempo de procesamiento que se le dedica a cada proceso en ejecución. En esa gestión el sistema

[25] Definir HMI

operativo irá suspendiendo procesos que quedan a la espera de respuestas de eventos externos o de periféricos y lo más probable es que en dichas condiciones no se minimize el tiempo de pared necesario para ejecutar el programa que estamos utilizando como patrón. Se define entonces el tiempo de CPU, que es el tiempo que la máquina está efectivamente computando sobre el programa en cuestión. Claramente, el tiempo que percibe el usuario no es el de CPU. El problema se solucionará entonces, comparando iguales cargas de trabajo en distintas arquitecturas y midiendo el tiempo de CPU de cada una de ellas. Utilizando la ecuación 2.1 obtenemos una poderosa herramienta para comparar arquitecturas. Si estuviésemos comparando la máquina A con la máquina B, la ecuación 2.2 implica que la máquina A es n veces más rápida que la máquina B. Referido en términos de desempeño se transforma en la ecuación 2.3 y decimos que la arquitectura A es n veces más performante que la arquitectura B.

$$\frac{t_{\mathrm{ex_B}}}{t_{\mathrm{ex_A}}} = n \tag{2.2}$$

$$n = \frac{t_{\mathrm{ex_B}}}{t_{\mathrm{ex_A}}} = \frac{\frac{1}{D_{\mathrm{B}}}}{\frac{1}{D_{\mathrm{A}}}} = \frac{D_{\mathrm{A}}}{D_{\mathrm{B}}} \tag{2.3}$$

Ahora bien, para que la comparación sea válida, el programa que ejecutan ambas arquitecturas debe ser el mismo y en las mismas condiciones, entendiéndose por ésto que:

- deben generar el mismo resultado a partir del mismo conjunto de datos de entrada,
- el algoritmo implementado para el procesamiento de los datos debe ser el mismo,
- las condiciones de entorno deben ser las mismas (sistema operativo si existiese, carga del sistema, cantidad de software en ejecución en paralelo si lo hubiese, etc).

Para lograr tal condición será necesario partir de un programa escrito en un lenguaje de más alto nivel que el ensamblador, ya que será el compilador quién decida que instrucciones utilizar a nivel código de máquina para llegar al objetivo.

2.3.2. Mediciones de desempeño

A lo largo de la historia, se han diseñado diversas pruebas para comparar máquinas denominados bancos de prueba —*benchmarks* en inglés—. Antiguamente, estos se implementaban utilizando "nucleos" —*kernels*— que son pequeñas piezas claves de programas reales, programas maqueta como implementaciones básicas de algoritmos comunes, y los llamados benchmarks sintéticos, que son programas escritos específicamente para emular el comportamiento de aplicaciones reales. Todas estas opciones han quedado desacreditadas puesto que fue posible (y ha sucedido) ajustar las teconologías de los compiladores

para que las arquitecturas ejecutasen estas pruebas de forma rápida, sin que ello reflejara necesariamente el comportamiento frente a aplicaciones reales. A pesar de ello, el benchmark sintético "Dhrystone"[26] es uno de los más nombrados a la hora de exponer las virtudes en el desempeño de arquitecturas del mercado de los procesadores embebidos. Como fuese mencionado en 2.3.1, las condiciones de entorno en las cuales se deben ejecutar estas pruebas deben estar de cierta forma normalizadas. Las implicancias que esto acarrea afectan por un lado a la compilación de los programas que componen a los bancos de prueba, debido al uso de *flags* o banderas de compilación que pueden generar transformaciones ilegales en algunos programas o disminuir el desempeño en otros. Para restringir estas diferencias y normalizar los resultados de las pruebas, los desarrolladores de *benchmarks* establecen el compilador a utilizar y un set de *flags* común a todos los programas escritos en C[27] o C++[28] —que son los lenguajes de programación utilizados de forma estándar para estas pruebas junto con Fortran[29] en algunos casos—. Por otro lado otra pregunta que surge es si se permite la modificación del código fuente. Hay tres enfoques en relación a esta cuestión:

- no se permite ninguna modificación del código fuente,
- se permiten modificaciones pero son impracticables; por ejemplo, las pruebas de base de datos funcionan sobre programas específicos con millones de líneas de código, cuyos desarrolladores no modifican el código con el objetivo de mejorar el desempeño en alguna arquitectura en particular,
- se permiten modificaciones, en tanto y en cuanto la versión modificada genere exactamente el mismo resultado que la orginal.

En este contexto, la cuestión clave que deben afrontar los desarrolladores, es si dichas modificaciones reflejarán la realidad y producirán resultados certeros como predictores de la performance real de los equipos teniendo en cuenta el mercado objetivo de las arquitecturas que se desean relevar.
No se profundizará en la temática del diseño de *benchmarks* puesto que escapa al alcance de este trabajo, pero sí se desea remarcar la importancia de la necesidad de medir la performance de las arquitecturas con el objetivo de compararlas.

2.3.3. Optimizaciones y Ley de Amdahl

Dado que se cuenta con un método objetivo y estándar para lograr mediciones de performance se pueden cuantificar las mejoras que se pueden aplicar a una determinada arquitectura; lo que conforma una poderosa herramienta para el diseñador. Como se

[26] Referencia a Dhrystone
[27] Lenguaje C
[28] Lenguaje C++
[29] Lenguaje Fortran

estableció en 2.3, el desmpeño se relaciona con el tiempo de ejecución de los programas. Ahora bien, si se implementa una mejora en el hardware, el objetivo de la misma será el de disminuir el tiempo de ejecución de los mismos. El término utilizado en inglés es el *speedup* (SU) que podría traducirse como "aceleración". Definimos de forma genérica el *speedup* en la ecuación 2.4 considerando los tiempos de ejecución antes (viejo, indicado como "v" en las ecuaciones) y después (nuevo, indicado como "n" en las ecuaciones) de la mejora:

$$SU = \frac{t_{\mathrm{ex_v}}}{t_{\mathrm{ex_n}}} \tag{2.4}$$

Considerando que la ecuación 2.4 resulta válida al aplicarla a la ejecución de un mismo programa en una versión con y sin mejora, y siendo obvio que su valor debe ser mayor a 1 puesto que si efectivamente se mejoró la arquitectura, el tiempo nuevo de ejecución debe ser menor al viejo. Ahora bien, una mejora (m) en particular puede afectar a la ejecución de todo un programa, como podría ser elevar la velocidad de reloj de la arquitectura que en consecuencia ejecutaría el programa en cuestión en menos tiempo, o puede afectar sólo a una parte del programa. Considerando que una mejora afecta a una sola parte de un determinado software que se usará como patrón se puede establecer que el *speedup* global (g) es el de la ecuación 2.5 donde f_{m} es la fracción del tiempo orignal de ejecución (viejo) en donde estará activa la mejora:

$$\begin{aligned} SU_{\mathrm{g}} &= \frac{t_{\mathrm{ex_v}}}{t_{\mathrm{ex_n}}} \\ &= \frac{t_{\mathrm{ex_v}}}{t_{\mathrm{ex_v}}\left(1 - f_{\mathrm{m}} + \dfrac{f_{\mathrm{m}}}{SU_{\mathrm{m}}}\right)} \\ &= \frac{1}{1 - f_{\mathrm{m}} + \dfrac{f_{\mathrm{m}}}{SU_{\mathrm{m}}}} \end{aligned} \tag{2.5}$$

$$\begin{aligned} SU_{\mathrm{g}} &= \frac{1}{1 - f_{\mathrm{m}} + \dfrac{f_{\mathrm{m}}}{SU_{\mathrm{m}}}} \\ SU_{\mathrm{g_{MAX}}} &\xrightarrow[SU_{\mathrm{m}} \to \infty]{} \frac{1}{1 - f_{\mathrm{m}}} \end{aligned} \tag{2.6}$$

$$
\begin{aligned}
SU_{\mathrm{g}} &\geqslant SU_{\mathrm{g_{ob}}} \\
\frac{1}{1 - f_{\mathrm{m}} + \dfrac{f_{\mathrm{m}}}{SU_{\mathrm{m}}}} &\geqslant SU_{\mathrm{g_{ob}}} \\
1 - f_{\mathrm{m}} + \frac{f_{\mathrm{m}}}{SU_{\mathrm{m}}} &\leqslant \frac{1}{SU_{\mathrm{g_{ob}}}} \\
1 - \frac{1}{SU_{\mathrm{g_{ob}}}} &\leqslant f_{\mathrm{m}} \left(1 - \frac{1}{SU_{\mathrm{m}}}\right) \\
\frac{1 - \dfrac{1}{SU_{\mathrm{g_{ob}}}}}{1 - \dfrac{1}{SU_{\mathrm{m}}}} &\leqslant f_{\mathrm{m}} = f_{\mathrm{m_{min}}}
\end{aligned}
\tag{2.7}
$$

La ecuación 2.5 es conocida como la Ley de Amdahl[30] y es una fórmula matemática que permite expresar las mejoras de rendimiento en una arquitectura. Podemos extraer dos rápidas conclusiones analisando la expresión: por un lado, dada una parte de código que puede ser optimizado, podemos encontrar el máximo *speedup* global que podremos obtener suponiendo que el *speedup* de la mejora en particular es infinito, es decir, suponemos que la mejora se ejecuta en tiempo cero (ver ecuación 2.6); por el otro dada una mejora en particular, podemos averiguar la mínima fracción de tiempo que necesitamos aplicarla para poder cumplir con algún *speedup* global objetivo (ob) (ver ecuación 2.7). En el caso de que existan diversas mejoras y bajo la condición de que las mejores no funcionen en simultáneo en la ejecución del programa, la ecuación 2.5 puede reescribirse como en 2.8.

$$
\begin{aligned}
SU_{\mathrm{g}} &= \frac{t_{\mathrm{ex_v}}}{t_{\mathrm{ex_n}}} \\
&= \frac{t_{\mathrm{ex_v}}}{t_{\mathrm{ex_v}} \left[1 - \sum\limits_{\mathrm{m}_i} \left(f_{\mathrm{m}_i} + \dfrac{f_{\mathrm{m}_i}}{SU_{\mathrm{m}_i}}\right)\right]} \\
&= \frac{1}{1 - \sum\limits_{\mathrm{m}_i} \left(f_{\mathrm{m}_i} + \dfrac{f_{\mathrm{m}_i}}{SU_{\mathrm{m}_i}}\right)}
\end{aligned}
\tag{2.8}
$$

2.4. Paralelismo

El concepto de paralelismo retoma las ideas de la taxonomía de Flynn vista en 2.2.2 y es una problemática que se aborda en distintos niveles. El software tiene básicamente dos tipos de paralelismos:

- paralelismo a nivel de datos o *data-level parallelism* (DLP): surge dado que múltiples datos pueden ser operados al mismo tiempo,

[30] Nota sobre Amdahl

- paralelismo a nivel de tarea o *task-level parallelism* (TaLP): surge debido a la capacidad de creación de cargas de trabajo que pueden operar independientemente en un entorno controlado por un sistema operativo.

El hardware puede explotar estos paralelismos para acelerar la ejecución de los programas, y las maneras que tiene de hacerlo son principalmente cuatro:

- paralelismo a nivel de instrucción o *instruction-level parallelism* (ILP): explota el DLP en baja medida con técnicas de optimización en la compilación del software a través de mecanismos como *pipeline* y *branch-delay slot*, o en mayor medida con técnicas como ejecución especulativa y predicción de saltos,
- arquitecturas vectoriales y procesadores gráficos o *graphic processor units* (GPUs): explotan el DLP al aplicar instrucciones simples a colleciones de datos en paralelo,
- paralelismo a nivel de hilos o *thread-level parallelism* (ThLP): explota tanto DLP como TaLP mediante la utilización de modelos de hardware con mecanismos específicos que permiten la interacción entre hilos de ejecución paralelos,
- paralelismo a nivel de solicitud o *request-level parallelism* (RLP): explota el paralelismo del software especificando tareas completamente independientes implementadas por el programador o por el sistema operativo.

Cuando Michael Flynn estudión los mecanismos del paralelismo en el software y el hardware elaboró la clasificación conocida como taxonomía de Flynn (2.2.2) que seguimos utilizando hoy en día. Dicha clasificación surgió de la observación del paralelismo en los hilos de datos e instrucciones. Relacionando la taxonomía con las clasificaciones de paralelismo vistas en esta sección, podemos analizar cada una de las categorías planteadas por Flynn:

- SISD: esta categoría es el uniprocesador. Desde la óptica del programador se trata de una máquina estándar secuencial, pero puede explotar el ILP, con técnicas como *pipeline*, *Superscalar* y *speculative execution*,
- SIMD: la misma instrucción es ejecutada por múltiples unidades de procesamiento sobre diversos hilos de datos. Estas computaodores explotan el DLP aplicando la misma operación a múltiples datos en paralelo. Cada unidad maneja sus datos independientemente, o sea, su memoria (de ahí el MD del SIMD), pero existe una única memoria de instrucciones y unidad de control que obtiene instrucciones y las ejecuta. Las arquitecturas que explotan de esta manera el paralelismo, son los GPUs y de forma híbrida, las arquitecturas clásicas con conjuntos de instrucciones extendidos para multimedia por ejemplo,
- MISD: no existen ejemplos de multiprocesadores comerciales de este tipo, si no más bien son de aplicaciones específicas como fuese mencionado en 2.2.2.3,

- MIMD: es el caso general hoy en día, con las arquitecturas paralelas. ada procesador obtiene sus instrucciones de un hilo propio y opera sobre sus datos; tratándose, entonces, de una forma de explotar TaLP. Típicamente, MIMD es más flexible que SIMD y por ende más general y aplicable, pero inherentemente más costoso en cuanto a recursos de hardware, por necesitar replicar múltiples veces circuitos lógicos, para su disponibilidad para cada núcleo de procesamiento. Estas arquitecturas poseen la habilidad de operar múltiples hilos de procesamiento en paralelo, cuyo objetivo puesde ser cooperativo, lo cual indica que explota el ThLP. Además, los *clústers* de computadoras se ubican también dentro de esta clasificación donde máquinas independientes, pueden ejecutar múltiples tareas independientes mediante poca comunicación, explotando así el RLP.

No debe perderse de vista que las arquitecturas modernas en general son híbridos de esta clasificación —combinando las bondades de SISD, SIMD y MIMD—.

2.5. Técnicas de optimización en el diseño de arquitecturas RISC

Durante esta sección estudiaremos las diversas optimizaciones implementables en el diseño del hardware de las arquitecturas RISC. Algunas de ellas fueron mencionadas en 2.4, otras serán directamente introducidas en esta sección. Debido a ciertas bondades que propone la filosofía se facilita la implementación de las optimizaciones.
Es importante no perder de vista que estas optimizaciones se desarrollaron con el objetivo de explotar algún tipo de paralelismo. Comenzaremos analizando las técnicas que explotan el paralismo a nivel de instrucción.

2.5.1. Explotando ILP

Dentro del universo del ILP encontraremos como básico y principal componente a la técnica de *pipelining* y luego se analizarán distintas técnicas que permiten mejorar el desempeño de una arquitectura *pipeline*.

2.5.1.1. Pipeline

Todos los procesarodres desde aproximadamente 1985 implementan *pipelining* para solapar y paralelizar la ejecución de instrucciones con el objetivo de mejorar el rendimiento. Esta potencial superposición de la ejecución de las instrucciones es lo que llamamos ILP, dado que ciertas partes de las mismas puede ejecutarse en paralelo. Es importante notar que se menciona “partes de las instrucciones”, ya que la ejecución de las mismas puede dividirse en etapas, en los que los recursos internos de la arquitectura que se utilizan

son independientes. El caso común es hacer la analogía con una línea de montaje en una fábrica, por ejemplo, de automóviles: primero debe fabricarse el chasis, una vez finalizado debemos montar y soldar los paneles de la carrocería para luego pintarlos, luego montar motor, caja de cambios y trenes de rodaje (que probablemente se han fabricado en sendas lineas de montaje independientes), luego se podrán montar paneles internos, tapizados, faros, ruedas, vidrios, cableados y demás componentes menores. Debe ser claro que las etapas descriptas, son independientes en lo que a recursos respecta, es decir, montar componentes menores no utiliza las mismas herramientas de la fábrica que la etapa de pintura o soldadura. Con la ejecución de las instrucciones dentro del microprocesador existe la misma estructura: no es el mismo hardware que se utiliza para obtener una instrucción de la memoria, que el utilizado para realizar, por ejemplo, la suma de dos registros o decidir un salto en el flujo del programa. Así como pasa en la fábrica de autos, que diversas etapas pueden utilizar la misma herramienta, en la ejecución de instrucciones puede suceder que diversas etapas utilicen los mismos recursos de hardware; en tal caso, deberemos duplicar el hardware para evitar situaciones en la que una etapa quede a la espera de que otra libere los recursos utilizados. Siguiendo con la analogía de la fábrica de automóviles mencionamos que, por ejemplo, motor y caja de cambios pueden provenir de otra línea de montaje. En el caso del microprocesador, la etapa del cálculo de las operaciones aritmético-lógicas puede ser procesada dentro de otro *pipeline*.
Debe resultar natural preguntarse por qué esta técnica mejora la performance. Cabe aclarar que tiempo de ejecución de una instrucción individual no se ve afectado, es decir, la latencia de ejecutar una dada instrucción se mantendrá —idealmente— constante. Lo que se ve optimizado es la tasa de instrucciones ejecutadas en una unidad de tiempo (llamado *throughput* en inglés), o sea, la cantidad de instrucciones que puede ejecutar la arquitectura por, digamos, segundo. Es importante destacar cómo debe dividirse en etapas la ejecución de las instrucciones. Idealmente, debe cumplirse que:

- las etapas del *pipepline* están perfectamente balanceadas,
- el *pipeline* siempre tiene todas sus etapas ocupadas.

Esto es lo que se llama *pipeline* ideal. El balance de las etapas implica que cada una de las etapas demora exactamente lo mismo en ejecutarse. Bajo estas condiciones, podemos calcular el *speedup* de un *pipeline* ideal como lo calculado en la ecuación 2.9 donde N es la cantidad de etapas en la que se divide la ejecución de las instrucciones.

$$\begin{aligned} SU_{\text{pipeline}} &= \frac{t_{\text{ex}_\text{v}}}{t_{\text{ex}_\text{n}}} \\ &= \frac{t_{\text{ex}_\text{v}}}{\frac{t_{\text{ex}_\text{v}}}{N}} \\ &= N \end{aligned} \tag{2.9}$$

Esto parecería implicar que en cuanto más etapas dividamos la ejecución de las instrucciones, mayor optimización obtendremos. Es cierto; pero debe tenerse en cuenta que la

implementación deberá aproximarse lo máximo posible al pipeline ideal, y además, cuanto más etapas se generen, puede complejizarse el hardware y aumentar la necesidad de multiplicar el mismo, lo que va en contra de mantener acotados los costos y el consumo energético; y más dificil será mantener el balance entre los tiempos de ejecución de las etapas.
Analizaremos ahora las diversas fuentes de problemas que apartarán el diseño de la idealidad que los llamaremos "peligros" o *hazards*. Específicamente, se pueden diferenciar diversas clases de peligro relacionadas con tres tipos de dependencias existentes entre las instrucciones:

- dependencias de recursos,
- dependencias de control,
- dependencias de datos.

2.5.1.1.1. Depedencias de recursos Las depenedencias de recursos surgen de la utilización de ciertos bloques constructivos básicos del hardware por parte de distintas etapas de ejecución de las instrucciones. Este tipo de dependencias generan los peligros homónimos y conforman un problema que debe ser atacado en etapas tempranas del diseño de la arquitectura, debido a que no es posible evitarlas a posteriori en tiempo de compilación y/o ejecución (como sí sucederá con los otros tipos de peligros); básicamente debe pensarse el conjunto de instrucciones de manera tal que exista la mínima cantidad posible de superposición en la utilización de recursos entre las etapas de ejecución de las instrucciones. Si fuese imposible eliminarlas, al momento de la implementación el diseñador se verá obligado a duplicar recursos circuitales, debido a que el costo será menor que la baja en rendimiento asociada a la lógica de control y demoras extras que sería necesario introducir con el objetivo de la correcta ejecución del software.

2.5.1.1.2. Dependencias de control El código máquina que se ejecuta en los microprocesadores es esencialmente secuencial, más allá de las abstracciones y los paradigmas de programación utilizados en los lenguajes de programación de nivel superior. Típicamente se ejecuta una instrucción a la vez y al finalizar la ejecución de la misma se continúa con la siguiente en la memoria de instrucciones, avanzando secuencialmente instrucción por instrucción el programa. Este escenario representaría una situación óptima para una implementación con *pipeline*, dado que siempre podríamos ingresar al mismo instrucciones que *deben* ser ejecutadas, acercándonos así al aspecto de *pipeline* ideal en cuanto a tener todas las etapas de nuestra línea de montaje siempre ocupadas. Es evidente que si este fuese el único mecanismo para "avanzar" en un programa, el programador se vería limitado en las posibilidades. Es por eso que típicamente, cualquier arquitectura moderna provee instrucciones de salto condicionales e incondicionales con el objetivo de interrumpir y modificar el flujo del código ejecutado; lo que da lugar a la aparición de las dependencias de control y sus peligros asociados.

La cuestión clave de los peligros de control radica en la etapa en la que una determinada instrucción de salto (condicional o incondicional) resuelve si se debe interrumpir o no el flujo del programa. Si la decisión se tomase en la primera etapa, no existiría este tipo de peligros, dado que al terminar la ejecución de esa primera etapa, el hardware ya tendría resuelto cuál es la siguiente instrucción a ejecutar y esa sería la que entra en el pipeline. Pareciese entonces, que sería óptimo entonces decidir si saltar o no en la primera etapa; pero esto no será posible sin sacrificar a cambio diversas relaciones de compromiso, como por ejemplo la posibilidad de dividir la ejecución de las instrucciones en más etapas. Claramente, el problema de decidir el salto en posteriores etapas radica en qué será lo que sucede con las instrucciones que entran al *pipeline* desde que se está ejecutando la instrucción de salto, hasta que se decide el mismo, ya que esas instrucciones pueden ser o nó trabajo útil, pero además, pueden alterar el flujo correcto del programa.
Este tipo de peligros se resolverá, en parte, a partir del diseño del set de instrucciones, trabajando en pos de que los saltos sean resueltos lo más tempranamente posible. No obstante, al momento de la implementación en particular, será muy probable que dicha decisión el hardware la realice en la segunda, tercera o cuarta etapa del pipeline, según el tipo y cantidad de divisiones del *pipeline*; lo que deberá ser resuelto, o bien en tiempo de compilación, o en tiempo de ejecución. Cabe destacar que para aprovechar al máximo el diseño de la implementación en particular, el compilador deberá conocer las particualridades de la misma.
Existe también, en el mundo de los microprocesadores, otra fuente de alteración en el órden de ejecución de las instrucciones, que son las excepciones y las interrupciones; si bien puede considerarse a estas últimas como un caso especial de las primeras. Las excepciones pueden considerarse condiciones anormales que al ser detectadas por la arquitectura deben ser atendidas o manejadas. La definición formal de la arquitectura deberá describir el tipo de excepciones que pueden generarse y las herramientas que el hardware proveerá para su correcto tratamiento. La naturaleza de las mismas es asincrónica, es decir que pueden suceder en cualquier momento de la ejecución de un programa, dada por condiciones evaluadas dinámicamente; como por ejemplo la detección de división por cero, o la detección de *carry* u *overflow* en la ALU. Es esta naturaleza asincrónica la que las convierte en otro peligro de control que debe ser tomado en cuenta. Las excepciones tendrán su detección determinada también en alguna etapa del pipeline, pero las fuentes de interrupción externa serán completamente ajenas al funcionamiento interno de la arquitectura; es por esto que la arquitectura deberá proveer hardware específico para detectar las señalizaciones correspondientes y dar lugar al tratamiento de las mismas dentro del ciclo de funcionamiento del hardware.
Nos ocuparemos ahora del tratamiento de estos peligros en el diseño de las arquitecturas. Una instrucción de salto condicional debe realizar alguna verificación sobre el estado en particular de algún registro o elemento de señalización interno del microprocesador para decidir si debe saltar a la sección de código objetivo o no. Típicamente una instrucción de este tipo tiene dos operandos, la condición, y algún tipo de dato que indique a donde debe saltar la ejecución del programa en el caso asertivo. Si finalmente la evaluación de la condición da como resultado que no debe interrumpirse el flujo normal del programa,

la instrucción que deberá ejecutarse es la subsiguiente en memoria de instrucciones. Pero como ya fue discutido, al momento de decidir si saltar o no, es probable que una o más etapas ya estén ocupadas con las instrucciones subsiguientes a la de salto. Si el salto finalmente no es tomado, la solución a este problema es trivial, y no existe peligro, ya que se continúa con la ejecución normal del programa. En cambio, si el salto es tomado debe decidirse que sucede con las instrucciones que ya ingresaron al *pipeline* y no deben ser ejecutadas.
En principio, podría parecer que el esfuerzo de diseño que insumirá implementar soluciones a estos problemas podría ser no justificado. Es por esto que existen estadísticas en las que se miden la cantidad porcentual de tipos de instrucciones que se ejecutan, en este caso, saltos que realiza un programa promedio y podemos citar como ejemplo CONSEGUIR UN EJEMPLO DE CANTIDAD DE BRANCHES EN UN SOFT.

2.5.1.1.2.1. Detención del *pipeline* A nivel de hardware, la solución más básica y rudimentaria a los peligros de control sería detener el procesamiento de instrucciones hasta que el salto sea resuelto —algo fácilmente implementable introduciendo instrucciones *nop*, es decir *no operation*, que es un tipo especial de instrucción que suele estar provista en todas las arquitecturas con el objetivo de no realizar ningún trabajo— y recién allí con la dirección de la siguiente instrucción a ejecutar calculada se prosigue a cargar nuevas instrucciones en el *pipeline*. Claramente el problema que intruduce esta técnica es la merma en el rendimiento por consumir ciclos de máquina en no realizar ningún trabajo; pero se trata de una solución muy sencilla de implementar.

2.5.1.1.2.2. Predicción de saltos Siguiendo con las técnicas implementadas exclusivamente a nivel de hardware, encontramos las técnicas de predicciń de saltos, o en inglés *branch prediction*, surge como una posibilidad de no desaprovechar el trabajo realizado por el *pipeline*, al intentar predecir si se va a saltar o no ante una instrucción de *branch* con el objetivo de instroducir al *pipeline* las instrucciones correspondientes a la predicción. Si la predicción fué erronea, se descartará el trabajo realizado hasta el momento por las instrucciones ingresadas posteriores al salto. El descartar este trabajo puede ser no trivial, dado que dependiendo de la etapa en la que se encuentren las instrucciones pueden haber alterado registros o memoria de datos, cuestiones que deberán ser tenidas en cuenta al momento del diseño de la implementación; es decir, el hardware debe garantizar la correctitud de los datos contenidos por los registros internos y la memoria al momento de realizar lo que se llama un *flush* o vaciado del *pipeline*. Siempre, la idea dentro de este contexto será lograr la predicción durante la primera etapa del *pipeline*, con el objetivo de que en el siguiente ciclo de máquina se obtenga la instrucción del destino predicho. La versión estática más sencilla es asumir que los saltos condicionales no son tomados (en inglés *assume branch not taken*): el hardware predice que no se va a saltar, por lo tanto, al *pipeline* ingresan las instrucciones sucesivas a la instrucción de *branch*, y si finalmente se salta, se procederá al vaciado del *pipeline*.
La predicción dinámica de saltos es más compleja y será muy particular en cada imple-

mentación de la arquitectura en cuestión. La implementación más básica utiliza un búfer de predicción o tabla histórica: una pequeña porción de memoria indexada por la parte más baja de la dirección de la instrucción de salto en cuestión contiene un bit de estado que indica si el salto fue tomado o no la última vez que se accedió. La cantidad de filas de esta tabla o búfer será 2^n donde n es la cantidad de bits menos significativos tomados de la dirección de las instrucciones de salto, tratandose entonces de un parámetro del diseño, donde se valorará el costo de agregar hardware específico para esta tarea. Es el tipo más sencillo de tabla que podemos hallar; no sabemos de hecho si la predicción es correcta o no —puede haber sido colocada allí por otra instrucción de salto cuya dirección tiene los mismos bits menos significativos— pero esto no afecta la correctitud. La predicción es sólo una pista que el hardware asumirá correcta, por lo tanto las siguientes instrucciones a cargar en el *pipeline* serán las correspondientes al destino de la misma. Si la predicción resultase incorrecta, se procederá al *flush* y se comezará a cargar el *pipeline* con las instrucciones correctas, al mismo tiempo que se invertirá el bit en la tabla de predicción. Aunque sencilla, esta implementación —llamada predictor de primer órden— trae aparejado la incorrectitud de al menos uno pero potencialmente dos predicciones en cada bucle o *loop* de software —resulta fundamental aclarar que los *loops* son la principal fuente de saltos condicionales en cualquier software. Considérese un bucle de n iteraciones: al ingresar al mismo, se puede predecir correctamente o no el salto (depende del estado del bit de predicción de la tabla de la última instrucción de salto ubicada allí), luego durante todas las iteraciones intermedias se predecirá correctamente, pero para la última resulta claro que se va a fallar. Es por ello que el siguiente paso es agregar otro bit de estado a la tabla; para cambiar la predicción un salto debe ser predicho dos veces mal. Llamamos a esta solución, predictor de segundo órden. Se logra reducir entonces el impacto de los *loops* en la predicción. La lógica de un predictor de este tipo es sencillamente implementable mediante una máquina de estados. Podemos ver un esquema de un predictor de segundo órden en la figura 2.11.

Configuraciones más avanzadas, utilizan técnicas de predicción estadísticas, llamados predictores correlacionadores o globales o de dos niveles. Estos enfoques basan su principio de funcionamiento en la relación que puede existir entre distintas instrucciones de *branch*, por ejemplo, al tener dos o maś bucles de código anidados, existirá una relación entre ellos que permitirá optimizar la predicción. Se describe formalmente a un *correlating predictor* como un binomio (m, n), tal predictor utiliza el comportamiento de los últimos m saltos para elegir entre 2^m predictores, cada uno de los cuales es un predictor de n bits, consecuentemente utilizando la historia de saltos que acumula el software en ejecución con el objetivo de mejorar la predicción. El selector es sencillo de implementar con un registro de desplazamiento: cada vez que se toma o no un salto se ingresa esa información en el registro de desplazamiento de m bits, que indiza cuál de las tablas de de n bits debe utilizarse para predecir y luego actualizarse. En la figura 2.12 podemos ver un esquema de un predictor de tal tipo. Estudios estadísticos demuestran que con predictores globales se mejora la tasa de acierto de predicciones. Citando [2, sección, 3.3] podemos ver los resultados de distintas pruebas del *benchmark* SPEC98[31] comparando

[31] SPEC98: Explicar

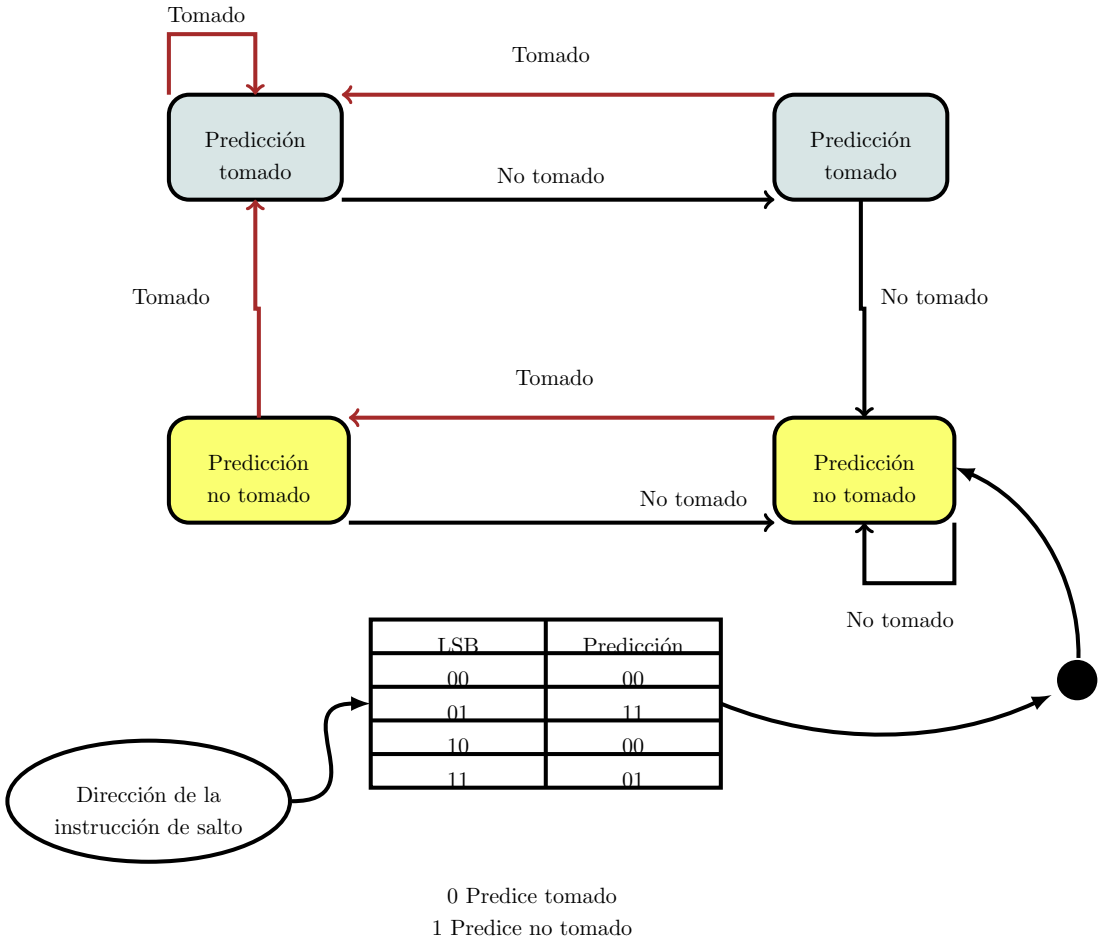


Figura 2.11: Esquema simplificado de un predictor de segundo órden.

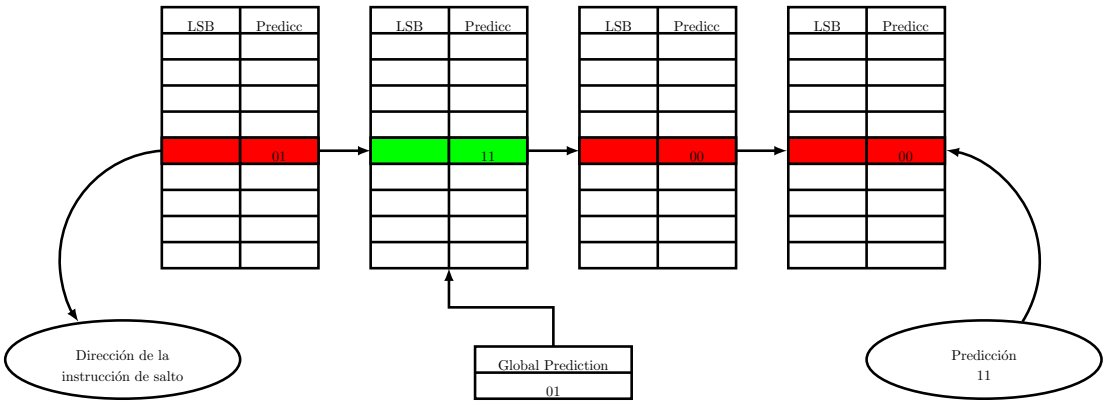


Figura 2.12: Esquema simplificado de un predictor global (2,2).

la tasa de acierto de distintos predictores, de la que podemos extraer como conclusión que no se justifica utilizar más de 4096 entradas de predictores, por ejemplo, en el caso del predictor global (2,2) utilizar tablas de 1024 entradas indizadas con los 10 bits menos significativos de la dirección de la instrucción de salto. Más allá de esa cantidad, el predictor deja de sumar eficiencia.
La motivación principal para utilizar predictores globales surge del estudio de que los locales de 2 bits fallan en condiciones que se presentan típicamente en el software y se puede lograr mejorar la performance del predictor agregando información global. Los llamados *tournament predictors* logran combinar de forma adaptativa los predictores locales y globales utilizando un sencillo contador saturante de 2 bits (la misma máquina de estados que se utiliza para el predictor de 2 bits) para cada salto, y así poder elegir entre dos diferentes predictores (locales, globales o incluso combinaciones) basándose en cuál fué mas efectivo en las predicciones más recientes. Con éste mecanismo que, aunque complejo en concepto sigue siendo sencillo de implementar a nivel de harwdare, se logra mejorar aún más la precisión de los mecanismos de predicción, elevando a su vez la cota de 4096 entradas. Estudios estadísticos demuestran que la eficiencia máxima de estos predictores se encuentra utilizando entre 8096 y 32768 entradas. Las mediciones estadísticas también las encontramos en la cita [2, sección, 3.3].
Para poder implementar eficientemente cualquiera de estas técnicas de predicción de saltos, debemos además considerar el cálculo eficiente de la próxima dirección de instrucción a introducir en el *pipeline* dada la predicción. La estructura de hardware que dará soporte físico a esta problemática se llama en inglés *branch target buffer*, es decir búfer de predicción de saltos. Se trata de una tabla que tendrá tres entradas y será indizada por bits menos significativos de la dirección de la instrucción de salto en cuestión. Las tres entradas serán: la dirección de la instrucción de salto, un bit de validez y la dirección predicha. En sí, se trata de un caché de predicciónes. En la figura 2.13 podemos visualizar gráficamente el esquema de una implementación de un *branch target buffer*.

2.5.1.1.2.3. Branch delay slot

2.5.1.1.3. Dependencias de datos

2.5.1.2. Forwarding

TODO.

2.5.1.3. Predicción de salto

TODO.

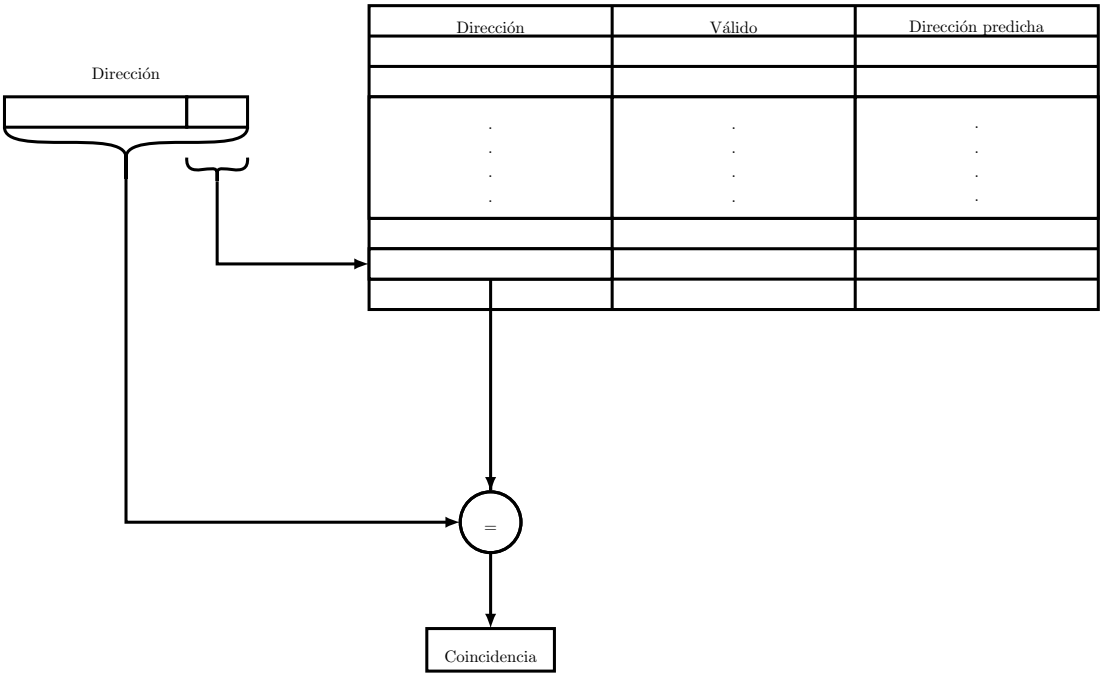


Figura 2.13: Esquema simplificado de un búfer de predicción de saltos.

2.5.2. Explotando ThLP

TODO.

2.5.3. Multiple Issue

TODO.

2.5.4. Scalar - Superscalar

TODO.

2.5.5. Multicore

Coherencia de cache? TODO.

2.6. Jerarquía de memoria

Localidad espacial y temporal de datos e instrucciones. TODO.

2.7. Diseño del XFire

La ISA define el modelo lógico: Tres cosas, los registros, el modelo de la memoria y como se relacionan los registros (operaciones).

IMPORTANTE:

El modelo formal del procesador queda definida por el ISA. El isa no es una tabla de instrucciones. Es el modelo lógico formal que describe la arquitectura.

Hablar de Alan Turing. Problemas Turing Computables -¿Investigar. Teorema de incompletitud de Gödel.

Referencias

[1] John von Neumann, “First Draft of a Report on the EDVAC”, 1945.

[2] John L. Hennessy & David A. Patterson, “Computer Architecture,” *A Quantitative Approach,* Morgan Kaufmann, Fifth Edition, 2012.

[3] David A. Patterson & John L. Hennessy, “Computer Organization and Desing,” *The Hardware Software Interface,* Morgan Kaufmann, Third Edition, 2005.

Luciano César Natale