INSTITUTO POLITÉCNICO NACIONAL

CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN

"DISEÑO E IMPLEMENTACIÓN EN HARDWARE DE UN ALGORITMO BIO-INSPIRADO"

T E S I S

QUE PARA OBTENER EL GRADO DE
MAESTRO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A

ING. ALEJANDRO LEÓN JAVIER

DIRECTORES DE TESIS

DRA. NARELI CRUZ CORTÉS
DR. MARCO A. MORENO ARMENDÁRIZ

MÉXICO, D. F. **2009**

1153

SIP-14

INSTITUTO POLITECNICO NACIONAL

SECRETARIA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REVISIÓN DE TESIS

En la Ciudad de México, D.F. siendo las 12:00 horas del día 4 del mes de noviembre de 2009 Se reunieron los miembros de la Comisión Revisora de Tesis designada por el Colegio de Profesores de Estudios de Posgrado e Investigación del:

Centro de Investigación en Computación

para examinar la tesis de grado titulada:

"DISEÑO E IMPLEMENTACIÓN EN HARDWARE DE UN ALGORITMO BIO-INSPIRADO"

LEÓN	**JAVIER**	**ALEJANDRO**
Apellido paterno	materno	nombre(s)

Con registro:

B	0	7	1	4	5	2

aspirante al grado de: ***MAESTRÍA EN CIENCIAS DE LA COMPUTACIÓN***

Después de intercambiar opiniones los miembros de la Comisión manifestaron ***SU APROBACIÓN DE LA TESIS***, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISIÓN REVISORA

Presidente

Dr. Sergio Suárez Guerra

Secretario

Dr. Carlos Fernando Aguilar Ibáñez

Primer vocal (Director)

Dr. Marco Antonio Moreno Armendáriz

Segundo vocal (Director)

Dra. Nareli Cruz Cortés

Tercer Vocal

Dr. Carlos Artemio Coello Coello

Suplente

Dr. Víctor Hugo Ponce Ponce

EL PRESIDENTE DEL COLEGIO

INSTITUTO POLITECNICO NACIONAL
CENTRO DE INVESTIGACION
EN COMPUTACION
DIRECCION

Dr. Jaime Álvarez Gallegos

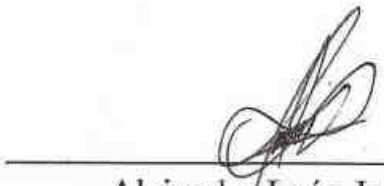
INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

CARTA CESIÓN DE DERECHOS

En la Ciudad de **México** el día **9** del mes de **noviembre** del año **2009**, el que suscribe **Alejandro León Javier** alumno del Programa de **Maestría en Ciencias de la Computación** con número de registro **B071452**, adscrito al **Centro de Investigación en Computación**, manifiesta que es autor intelectual del presente trabajo de Tesis bajo la dirección de la **Dra. Nareli Cruz Cortés y el Dr. Marco Antonio Moreno Armendáriz** y cede los derechos del trabajo intitulado **Diseño e implementación en hardware de un algoritmo bio-inspirado**, al **Instituto Politécnico Nacional** para su difusión, con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin permiso expreso del autor y/o director del trabajo. Este puede ser obtenido escribiendo a la siguiente dirección **ljavierb07@sagitario.cic.ipn.mx**. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.



Alejandro León Javier

AGRADECIMIENTOS

A la **Dra. Nareli Cruz Cortés** y al **Dr. Marco Antonio Moreno Armendáriz** por su guía y todo el apoyo brindado durante el desarrollo de la presente tesis.

Al **Dr. Carlos Artemio Coello Coello** por sus consejos.

Al **Dr. Francisco González Henríquez** por su aportación en la realización de esta tesis.

A

MI FAMILIA

Por todo su apoyo y cariño

A

MIS ASESORES

Por su paciencia y consejos

Índice general

Índice de figuras

Índice de tablas

Resumen

Existen una gran cantidad de algoritmos bio-inspirados, sin embargo, se aplican en situaciones diferentes, es decir, algunos sirven para realizar optimización numérica y/o optimización combinatoria, otros para realizar aproximaciones de funciones o modelado, etc. Todos ellos están inspirados en la biología, por ejemplo, los algoritmos de colonia de hormigas, están inspirados en la forma en que estos insectos encuentran la ruta más corta para llegar al alimento, por lo que se emplean para la búsqueda de rutas óptimas en grafos.

Dado un problema de optimización numérica global mono-objetivo, podemos elegir algún algoritmo bio-inspirado para atacar el problema. Sin embargo, existen problemas cuya optimización requiere ser realizada en tiempo real, generalmente esto es requerido para aplicaciones de ingeniería. En particular, hay algunas aplicaciones donde se requiere que los algoritmos sean ejecutados sobre un hardware de propósito específico, este es el caso del algoritmo bio-inspirado que se implementó en este trabajo de tesis.

El Algoritmo Genético Compacto (AGc) es una técnica de búsqueda que imita el comportamiento de un Algoritmo Genético Simple (AGS) con cruza uniforme. Su característica principal es la simplicidad del mismo y el manejo de un vector de probabilidad que emula la población.

En este trabajo se realiza el diseño y la implementación de un Algoritmo Genético Compacto sobre un *Field Programmable Gate Arrays* (FPGA). El diseño se hace a través de un Lenguaje de Descripción de Hardware (HDL) para su posterior implementación sobre un FPGA de la marca Altera.

Los resultados muestran que tener un algoritmo de búsqueda como un Algoritmo Genético Compacto en hardware, puede ser muy útil, ya que esto permite realizar optimización numérica global en tiempo real, y además permite incorporarlo en un chip que forme parte de una aplicación de mayor escala.

Abstract

There exist a great quantity of bio-inspired algorithms, however, they are used in different situations, that is, some them are useful for accomplishing numerical and/or combinatorial optimization, other ones to accomplish approximation of functions or modeling, etc. All of them are inspired in biology, that is, for example, the Ant Colony Optimization algorithms (ACO) are inspired in the way that these insects find the shorter way to arrive to food, this is why the ACO is used for searching optimal routes in graphs.

For a given numerical single-objective optimization problem, we can select a bio-inspired algorithm to solving it. However, there exist some problems where the optimization must be accomplished in real time, this is generally required for engineering applications. In particular, there are some applications where it is necessary to execute the algorithm on a specific purpose hardware platform, this is the case of the bio-inspired algorithm implemented in this work.

The Compact Genetic Algorithm (cGA) is a searching technique that mimics the behavior of a Simple Genetic Algorithm (SGA) with uniform crossover. Its main characteristic is its simplicity and a probability vector that emulates the population.

In this work the design and implementation of a Compact Genetic Algorithm on a Field Programmable Gate Arrays (FPGA) are accomplished. The design is done with the Hardware Description Language (HDL) for its later implementation on a FPGA of Altera.

According to results, it is a viable option to have a searching algorithm like the Compact Genetic Algorithm on a FPGA, due that it can be used in real time applications. The proposed design allows its easy integration into a chip or module for building a larger-scale application.

Capítulo 1

Introducción

1.1 Objetivo general

El objetivo general de este trabajo de tesis es realizar el diseño e implementación de un Algoritmo Genético Compacto estándar y su versión con elitismo y mutación en **VHDL** (Lenguaje de Descripción de Hardware de Circuitos Integrados de Muy Alta Velocidad, donde la **V** es de Very High Speed Integrated Circuit (VHSIC) y **HDL** de Hardware Description Language). La implementación se realiza sobre un FPGA (Field Programmable Gate Array) de la marca Altera, el Cyclone II EP2C70F896C6.

1.2 Objetivos particulares

Los objetivos particulares de este trabajo son:

a) Diseñar e implementar en hardware un Algoritmo Genético Compacto (AGc).

b) Diseñar e implementar en hardware el Algoritmo Genético Compacto con Elitismo y Mutación (AGcEM).

c) Diseñar en VHDL un conjunto de funciones de prueba clásicas para algoritmos evolutivos.

d) Realizar pruebas de desempeño de los algoritmos sobre un FPGA y también sobre software.

1.3 Motivación

La motivación principal del diseño e implementación de un AGc sobre un FPGA radica en que existen aplicaciones en ingeniería donde es necesario realizar la optimización de algunos parámetros en línea.

Una opción para realizar estas optimizaciones es recurrir a los algoritmos evolutivos clásicos, sin embargo, la mayoría de ellos consumen gran cantidad de recursos para el cómputo, debido a que tienen que aplicar una serie operadores sobre un conjunto de posibles soluciones (alrededor de cien), lo que hace muy difícil tener uno de ellos a disposición para realizar optimización en tiempo real. Dentro de los algoritmos evolutivos existe un tipo especial llamado Algoritmo Genético Compacto, el cual tiene características interesantes que lo hacen viable para su implementación en hardware, y sobre todo, para aplicaciones que requieran optimización en línea. A lo largo de esta tesis explicaremos más a detalle el Algoritmo Genético Compacto y plantearemos una arquitectura que será diseñada en VHDL.

El AGc es un tipo de *Estimation of Distributiom Algorithm* (EDA) debido a que emplea estimaciones de distribuciones. Existe otro algoritmo llamado *Population-based Incremental Learning* (PBIL) [7] que es similar al AGc e igualmente es un EDA. Se ha optado por realizar el diseño del AGc y no otro EDA porque el AGc tiene más ventajas sobre hardware que otros algoritmos de esta familia. Por ejemplo, el PBIL no puede lograr el mismo nivel de compresión de memoria del AGc, esto es porque genera un número grande de individuos en base al vector de probabilidades, a diferencia del AGc que sólo genera dos individuos por generación.

Otra razón para la elección de este algoritmo es que ha sido utilizado en aplicaciones de ingeniería exitosamente, y por tanto existe más documentación al respecto. Además en la literatura especializada se encuentran otras implementaciones del AGc en hardware y del PBIL no existen hasta el momento.

El AGc es un algoritmo que ha sido estudiado ampliamente y sobre el cual se han propuesto modificaciones que ayudan a mejorar su capacidad de búsqueda y la calidad sus soluciones.

1.4 Trabajos publicados

Actualmente tenemos las siguientes publicaciones y trabajos presentados:

1) Alejandro León Javier, Marco A. Moreno Armendáriz y Nareli Cruz Cortés, "Optimizing the Compact Genetic Algorithm Design with Cellular Automata".

Research in Computing Science, Special issue: Advances in Pattern Recognition. Vol. 44, ISSN 1870-4069. Series Editorial Board. México D.F. Noviembre del 2009.

2) Alejandro León Javier, Marco A. Moreno Armendáriz, Nareli Cruz Cortés y Sandra Orantes Jiménez, "Finding Minimal Addition Chains with a Particle Swarm Optimization Algorithm". Proceedings MICAI 2009: Advances in Artificial Intelligence. Lectures Notes in Artificial Intelligence. Subseries of Lecture Notes in Computer Science. ISSN 1867-8211. Springer-Verlag Berlin Heidelberg 2009.

3) Alejandro León Javier, Marco A. Moreno Armendáriz y Nareli Cruz-Cortés, "Designing a Compact Genetic Algorithm with Minimal FPGA Resources". Advances in Computational Intelligence. Advances in Intelligent and Soft Computing. ISSN 1867-5662. Springer-Verlag Berlin Heidelberg 2009.

4) Alejandro León-Javier, Marco A. Moreno-Armendáriz and Nareli Cruz-Cortés, "Artificial Neural Network used as an Objective Function for a Compact Genetic Algorithm with Persistent Elitism and Mutation", CIC 2008: 17th International Conference on Computing, Poster, diciembre 2008.

Capítulo 2

Fundamentos sobre algoritmos bio-inspirados

2.1 Algoritmos evolutivos

Los algoritmos evolutivos (AEs) son procedimientos de búsqueda y optimización que tienen sus orígenes e inspiración en el mundo biológico. Se caracterizan por imitar procesos adaptativos de los procesos naturales y se basan en la supervivencia del mejor individuo, siendo un individuo una solución potencial del problema que se implementa como una estructura de datos. Trabajan sobre poblaciones de soluciones que evolucionan de generación en generación mediante operadores genéticos adaptados al problema [1].

De acuerdo a la literatura especializada, existen muchas variantes de los algoritmos evolutivos. La esencia detrás de estas técnicas es la misma: dada una población de individuos (posibles soluciones del problema), la influencia del ambiente ocasiona selección natural (la supervivencia del más apto), misma que da lugar a un incremento en la aptitud de la población. Dada una función de calidad a ser maximizada, podemos crear aleatoriamente un conjunto de soluciones candidatas, es decir, elementos del dominio de la función, y aplicar la función de calidad como una medida de aptitud abstracta, mientras más alta mejor. Basados sobre esta aptitud, algunos de los mejores candidatos son seleccionados para pasar a la siguiente generación y aplicarles recombinación y/o mutación. La recombinación es un operador aplicado a dos o más candidatos seleccionados (llamados padres) y resulta en uno o más nuevos candidatos (hijos). La mutación es aplicada a un candidato y resulta un nuevo candidato. Realizando la recombinación y la mutación se conduce a un conjunto de nuevos candidatos (hijos) que compiten – basados en su aptitud (y posiblemente en la edad) – con los viejos candidatos por un lugar en la siguiente generación. Este proceso puede ser iterativo

hasta que un candidato con suficiente calidad (una solución) sea encontrado o que algún criterio de finalización establecido sea previamente alcanzado [2].

Durante este proceso hay dos fuerzas fundamentales que forman la base de los sistemas evolutivos:

a) Los operadores de variación (recombinación y mutación) crean la diversidad necesaria y consecuentemente facilitan lo novedoso.

b) La selección actúa como una fuerza que impulsa la calidad.

La combinación de los operadores de variación y la selección generalmente permiten mejorar los valores de aptitud en poblaciones consecutivas. Es fácil (aunque un poco engañoso) ver tal proceso como si la evolución fuera optimización, o por lo menos "aproximación", acercándose cada vez más a valores óptimos sobre su curso. En otros casos, la evolución es vista como un proceso de adaptación. Desde esta perspectiva, la aptitud no es vista como una función objetivo a ser optimizada, sino como una expresión de condiciones ambientales. Combinando estas condiciones más de cerca, implica un incremento en la viabilidad, reflejada en un número más grande de hijos. El proceso evolutivo hace que la población mejore más y más, adaptándose al ambiente.

Notemos que muchos componentes de tal proceso evolutivo son estocásticos. Durante la selección, los individuos más aptos tienen una probabilidad más alta de ser seleccionados que los menos aptos, pero generalmente hasta los individuos débiles tienen una oportunidad de llegar a ser padres o sobrevivir. Para la recombinación, la selección de los individuos que serán recombinados es aleatoria. De igual forma para la mutación, las partes que serán mutadas dentro de una solución candidata, y las nuevas partes reemplazándolas, son seleccionadas aleatoriamente. El esquema general de un algoritmo evolutivo en pseudocódigo es mostrado en la Fig. 2.1, la Fig. 2.2 muestra un diagrama del mismo.

Es fácil ver que el esquema cae en la categoría de algoritmos de generación y prueba. La función de aptitud representa una estimación de la calidad de la solución, y el proceso de búsqueda es conducido por los operadores de variación y selección.

```
INICIO
   INICIALIZAR la población con soluciones candidatas aleatorias;
   EVALUAR cada candidato;
   REPETIR HASTA (CONDICIÓN DE TERMINACIÓN sea alcanzada) HACER
       1 SELECCIONAR padres;
       2 RECOMBINAR parejas de padres,
       3 MUTAR los hijos resultantes;
       4 EVALUAR los nuevos candidatos;
       5 SELECCIONAR los individuos de la siguiente generación
   FIN DE REPETIR
   SALIDA: El mejor individuo de la población final
FIN
```

Fig. 2.1 Esquema general de un algoritmo evolutivo en pseudocódigo.

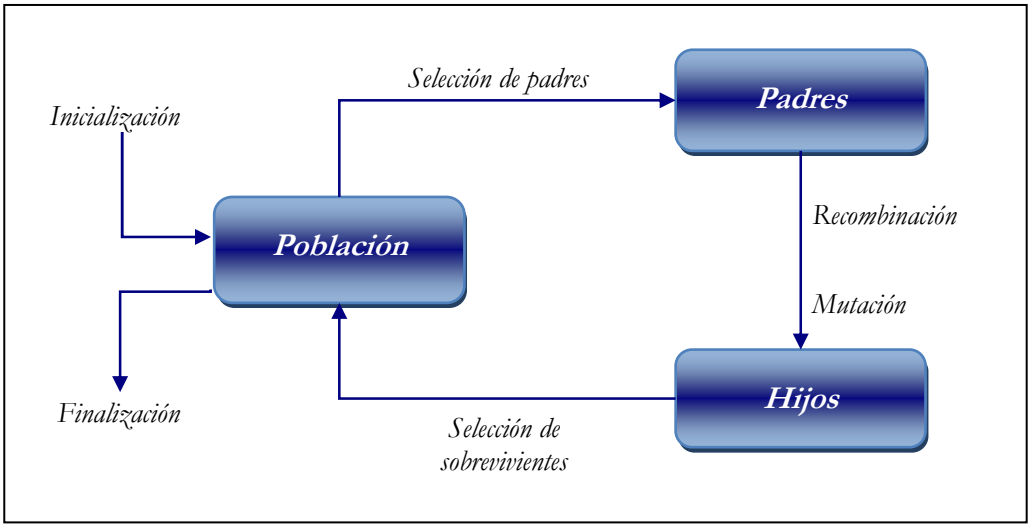


Fig. 2.2 Esquema general de un algoritmo evolutivo en diagrama de flujo.

Los algoritmos evolutivos poseen ciertas características que pueden ayudar a ubicarlos dentro de la familia de métodos de generación y prueba:

- Los AEs son poblacionales, es decir, procesan una colección entera de soluciones candidatas simultáneamente.
- Los AEs muchas veces utilizan la recombinación para mezclar información de muchas soluciones candidatas en una nueva solución.
- Los AEs son estocásticos.

Aunque existen una gran cantidad de algoritmos evolutivos, se suele hablar de tres paradigmas principales: Algoritmos Genéticos, Estrategias Evolutivas y Programación Evolutiva.

Término	***Concepto computacional (AEs)***
Cromosoma	*Estructura de datos que contiene una cadena de parámetros de diseño o genes. Esta estructura de datos puede almacenarse, por ejemplo, como una cadena de bits o un arreglo de enteros.*
Gene	*Es una subsección de un cromosoma que (usualmente) codifica el valor de un solo parámetro.*
Genotipo	*Codificación (por ejemplo, binaria) de los parámetros que representan una solución del problema a resolverse.*
Fenotipo	*Es la decodificación del cromosoma. Es decir, los valores obtenidos al pasar de la representación (binaria) a la usada por la función objetivo.*
Individuo	*Es un solo miembro de la población de soluciones potenciales a un problema. Cada individuo contiene un cromosoma que representa una solución posible del problema.*
Aptitud	*Valor que se asigna a cada individuo y que indica qué tan bueno es éste con respecto a los demás.*
Generación	*Cada iteración de la medida de aptitud y a la creación de una nueva población por medio de operadores de reproducción.*
Operador genético o de reproducción	*Es aquel mecanismo que influencia la forma en que se pasa la información genética de padres a hijos.*
Cruza	*Operador que forma un nuevo cromosoma combinando partes de cada uno de sus cromosomas padres.*
Mutación	*Operador que forma un nuevo cromosoma a través de alteraciones (usualmente pequeñas) de los valores de los genes de un solo cromosoma padre.*
Elitismo	*Mecanismo utilizado para asegurar que los cromosomas de los miembros más aptos de una población se pasen a la siguiente generación sin ser alterados por los operadores genéticos.*

Tabla 2.1 Terminología utilizada por los AEs [3].

A pesar de que existen estos paradigmas dentro de los AEs, es importante mencionar que para cada paradigma existen una gran diversidad de variantes del mismo, es decir, existe un

algoritmo clásico o estándar del cual se han hecho mejoras, o cambios para poder aplicarlos a cierto tipo de problemas, ocasionando que surjan muchas variantes, lo cual ha provocado que los AEs se utilicen cada vez más para resolver problemas de optimización.

Una característica importante de los AEs es la elección adecuada de los parámetros del mismo, así como el tipo de operador y la representación. Otro punto interesante es definir el mecanismo de selección más adecuado. Debido a que los AEs son técnicas de búsqueda heurísticas, el resultado será muy bueno o cercano al óptimo (con los parámetros correctos del algoritmo), y generalmente serán distintos para cada búsqueda.

En el área de la computación evolutiva es común emplear terminología de la biología, ya que los algoritmos están inspirados en la naturaleza. En la Tabla 2.1 presentamos un resumen de la terminología empleada en el área y su correspondiente concepto computacional.

Cada paradigma de la computación evolutiva fue desarrollado independientemente y con motivaciones distintas.

2.1.1 Algoritmos Genéticos

Los algoritmos genéticos (AG's) son quizás los algoritmos evolutivos más utilizados. Estos fueron inicialmente concebidos por John H. Holland como un método de estudio del comportamiento adaptativo [2].

Los AG's enfatizan la importancia de la cruza sexual (operador principal) sobre el de mutación (operador secundario), y usa selección probabilística. El algoritmo básico es el siguiente:

a) Generar (aleatoriamente) una población inicial.

b) Calcular la aptitud de cada individuo.

c) Seleccionar (probabilísticamente) en base a aptitud.

d) Aplicar operadores genéticos (cruza y mutación) para generar la siguiente población.

e) Ciclar a partir de b) hasta que cierta condición se satisfaga.

La representación tradicional es la binaria, como se muestra en la Fig. 2.3. A la cadena binaria se le llama "cromosoma". A cada posición de la cadena se le denomina "gene" y al valor dentro de esa posición se le llama "alelo".

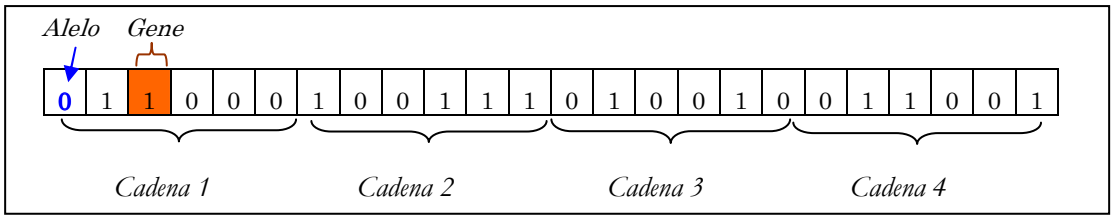


Fig. 2.3 Ejemplo de codificación binaria utilizada en AG's.

Para poder aplicar el algoritmo genético se requiere de cinco componentes básicos, que son los siguientes:

1) Una representación de las soluciones potenciales del problema.
2) Una forma de crear una población inicial de posibles soluciones (normalmente un proceso aleatorio).
3) Una función de evaluación que juegue el papel del ambiente, clasificando las soluciones en términos de su "aptitud".
4) Operadores genéticos que alteren la composición de los hijos que se producirán para las siguientes generaciones.
5) Valores para los diferentes parámetros que utiliza el algoritmo genético (tamaño de la población, probabilidad de cruza, probabilidad de mutación, número máximo de generaciones, etc.)

Existen muchas formas para la representación de las soluciones o individuos, sin embargo, para los AG's la más usual es la binaria. Para un problema de optimización numérica global, debemos saber cuántas variables se van a optimizar y sobre que intervalo se va a trabajar para cada una, además de la precisión de cada variable (número de dígitos después del punto que se van a tomar en cuenta). Estos parámetros nos van a permitir definir el tamaño de cada variable en bits y la longitud del cromosoma. Por ejemplo, supongamos que nuestro problema requiere optimizar dos variables ($x, y \in [-100,100]$), con una precisión de 3 dígitos

después del punto para cada una. Para determinar el tamaño en bits de cada variable utilizamos la siguiente expresión:

$$TamVar = (\text{int} \uparrow) \log_2 ((LimSup - LimInf) * 10^{Prec}) \qquad (2.1)$$

Donde $Var \in [LimInf, LimSup]$ con una precisión *Prec*. El término $(\text{int} \uparrow)$ indica que si el resultado obtenido es un número decimal, debemos redondearlo al entero próximo superior (Si obtenemos 2.2, *TamVar* será 3). *TamVar* debe ser un entero, ya que indica la longitud en bits de la variable. Para calcular la longitud del cromosoma basta con sumar las longitudes de las variables del problema, es decir:

$$LongCrom = TamVar_1 + TamVar_2 + TamVar_3 + ... + TamVar_n \qquad (2.2)$$

Para el ejemplo mencionado anteriormente la longitud de cada variable y del cromosoma se obtienen de la siguiente manera:

$$Tam_x = (\text{int} \uparrow) \log_2 ((100 - (-100)) * 10^3) = (\text{int} \uparrow) \log_2 (200000) = (\text{int} \uparrow) 17.6096 = 18\ bits$$
$$Tam_y = (\text{int} \uparrow) \log_2 ((100 - (-100)) * 10^3) = (\text{int} \uparrow) \log_2 (200000) = (\text{int} \uparrow) 17.6096 = 18\ bits$$

Por tanto, los individuos de nuestra población tendrán el formato mostrado en la Fig. 2.4, siendo *n* el tamaño total de la población con una longitud del cromosoma de 36 bits.

Cromosoma de uno de los individuos de la población

0	1	1	0	0	1	0	0	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	1	1	1	0	1	1	0	0	1	0	1	0	1
																		x																y			

Población de individuos (posibles soluciones)

Ind.	x																		y																	
1	0	1	1	0	0	1	0	0	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	1	1	1	0	1	1	0	0	1	0	1	0	1
2	0	0	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	1	0	0	0	1	1	0	0
3	1	0	1	1	0	1	0	1	1	1	0	0	0	0	0	1	1	1	0	0	0	0	1	0	0	1	0	1	1	1	0	1	0	1	1	1
4	0	1	1	1	1	1	1	0	1	0	1	0	0	1	1	0	0	0	0	0	1	1	0	1	1	0	0	0	1	0	1	0	1	1	1	1
...	0	1	1	0	1	0	0	0	0	1	1	1	1	1	0	1	0	0	0	1	0	0	0	0	0	0	1	1	0	1	1	0	1	1	0	0
n	1	1	1	0	0	1	1	1	0	1	1	0	0	0	0	0	1	0	1	0	1	1	1	0	1	1	0	0	0	0	0	0	1	0	1	0

Fig 2.4 Ejemplo de cromosoma y población.

Como ya se mencionó anteriormente, la inicialización de la población es aleatoria. Para realizar el cálculo de la aptitud de cada individuo, se debe evaluar cada individuo en la función objetivo. Para evaluar cada individuo debemos decodificarlo a una forma que se pueda evaluar en la función objetivo, por ejemplo, un número real. La decodificación se lleva a cabo con la siguiente expresión:

$$Var = LimInf + decimal(subcadena) * \frac{LimSup - LimInf}{2^{TamVar} - 1} \qquad (2.3)$$

Donde *Var* es la variable decodificada, *LimInf* es el límite inferior del intervalo para la variable, *decimal(subcadena)* es el valor decimal de la subcadena binaria de la variable, *LimSup* es el límite superior del intervalo para la variable y *TamVar* es el tamaño en bits de la subcadena binaria de la variable. En la Fig. 2.5 se muestra la decodificación de un individuo compuesto por dos variables (x & y) de 18 bits cada una, donde $x, y \in [-100,100]$.

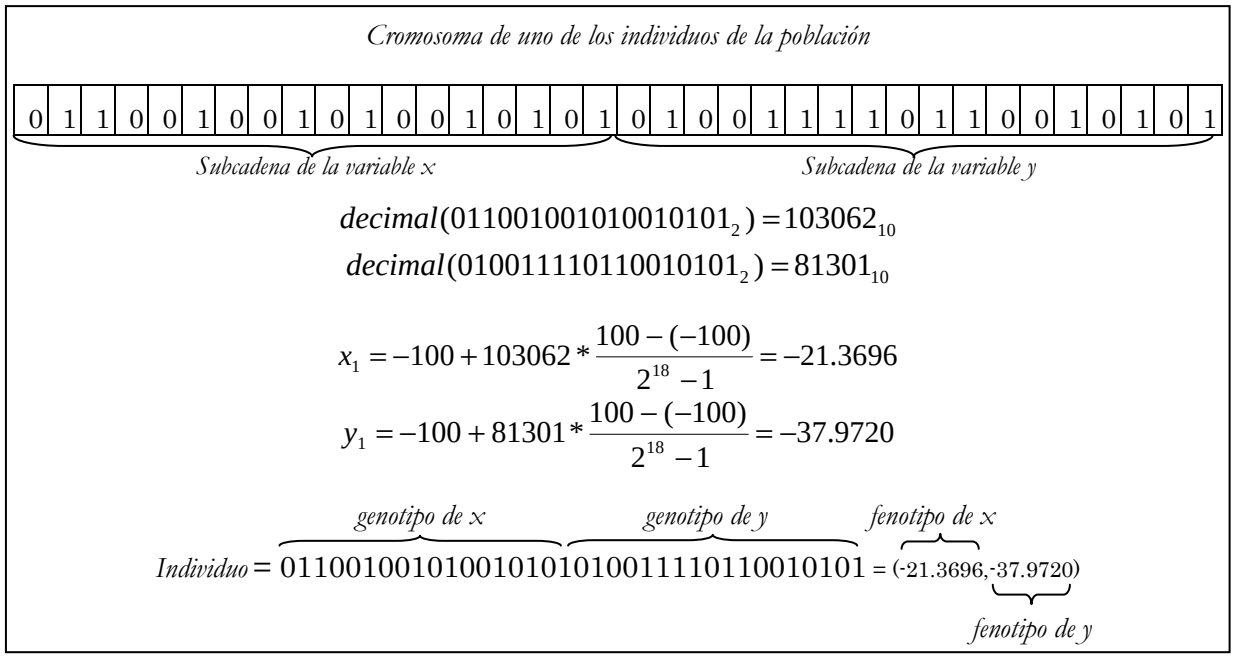


Fig. 2.5 Decodificación de un individuo.

Para calcular la aptitud de un individuo, solo se sustituyen los valores del fenotipo en la función objetivo. Suponiendo que la función objetivo para nuestro problema es $f(x) = x^2 + y^2$, el cálculo de aptitud se realiza de la siguiente manera:

Individuo = 01100100101001010101001111010110010101 = (-21.3696,-37.9720)

$$\text{Aptitud} = f(x,y) = x^2 + y^2 = (-21.3696)^2 + (-37.9720)^2 = 1898.5325$$

El cálculo de la aptitud se debe realizar para toda la población. El tamaño de la población suele ser de 100 individuos, aunque es un dato que puede ser definido por el diseñador del algoritmo. Para la población de la Fig. 2.4, se muestran los valores de aptitud en la Tabla 2.2.

Individuo	*fenotipo*	*aptitud*
1	*(-21.3696, -37.9720)*	*1898.5325*
2	*(-74.9144, -32.7054)*	*6681.8105*
3	*(41.9980, -92.6093)*	*10340.3144*
4	*(- 1.0555, -57.6788)*	*3327.9580*
…	*(-18.3682, -49.3314)*	*2770.9777*
n	*(80.7639, 46.1019)*	*8648.1927*

Tabla 2.2 Aptitud de la población.

Fig. 2.6 Clasificación de las técnicas de selección.

La selección es un mecanismo esencial de los AG's, ya que permite elegir a los individuos que se reproducirán, para dar lugar a los hijos. Comúnmente la selección en los AG's es probabilística, es decir, que cualquier individuo puede ser padre, aunque los individuos con menor aptitud tienen menos probabilidad, a diferencia de la selección de tipo extintiva, en la cual solo los individuos más aptos tienen la oportunidad de ser padres. En la Fig. 2.6 se muestra una clasificación de las técnicas de selección utilizadas por los AG's.

Como se puede apreciar en la Fig. 2.6, las técnicas de selección pueden clasificarse en 3 grandes grupos: selección proporcional, selección mediante torneo y selección de estado uniforme.

En el grupo de técnicas de selección proporcional se eligen a los individuos de acuerdo a su contribución de aptitud con respecto al total de la población. Dentro del grupo de selección proporcional podemos encontrar técnicas como la ruleta, sobrante estocástico, universal estocástica y muestreo determinístico; y en los cuales podemos utilizar los siguientes aditamentos: escalamiento sigma, jerarquías y selección de Boltzmann.

Dentro del grupo de selección mediante torneo podemos encontrar 2 técnicas: la determinística y la probabilística. El algoritmo de estas técnicas es el siguiente:

- Barajar los individuos de la población.
- Escoger un número p de individuos (normalmente 2).
- Compararlos con base en su aptitud.
- Seleccionar al ganador de acuerdo a la versión que se utilice: el ganador del "torneo" es el individuo más apto (versión determinística), o se aplica *flip(p)* (esta función devuelve cierto con una probabilidad p) y si el resultado es cierto, se selecciona al más apto, de lo contrario, se selecciona al menos apto (versión probabilística).
- Debe barajarse la población un total de p veces para seleccionar N padres (donde N es el tamaño de la población).

Las técnicas pertenecientes al grupo de selección de estado uniforme se utilizan en AG's no generacionales (donde solo unos cuantos individuos son reemplazados en cada generación). El algoritmo básico para este tipo de técnicas de selección es el siguiente:

- Sea *G* la población original de un AG.
- Seleccionar *R* individuos (*1*≤*R*<*M*) de entre los más aptos. Por ejemplo, *R*=2.
- Efectuar cruza y mutación a los *R* individuos seleccionados. Sean *H* los hijos.
- Elegir al mejor individuo en *H*. (o a los μ mejores).
- Reemplazar a los μ peores individuos de *G* por los μ mejores individuos de *H*.

Existen otras técnicas de selección de menor uso como: selección disruptiva, jerarquías no lineales y competitivas.

La selección por ruleta es una de las más empleadas en AG's, así que emplearemos este mecanismo para la población de la Fig. 2.4, donde los valores de aptitud se localizan en la Tabla 2.2. Para poder aplicar la ruleta se requiere calcular el valor esperado *Ve* de cada individuo con la siguiente expresión:

$$Ve_i = \frac{f_i}{\sum_{j=1}^{n} f_j} * n \qquad (2.4)$$

Donde f_i es el valor de aptitud del individuo i y n el tamaño de la población. El valor esperado de cada individuo de la población de la Fig. 2.4 se muestra en la Tabla 2.3.

Individuo	***fenotipo***	***Aptitud (f)***	***Ve***
1	*(-21.3696, -37.9720)*	*1898.5325*	*0.3383*
2	*(-74.9144, -32.7054)*	*6681.8105*	*1.1907*
3	*(41.9980, -92.6093)*	*10340.3144*	*1.8427*
4	*(- 1.0555, -57.6788)*	*3327.9580*	*0.5930*
…	*(-18.3682, -49.3314)*	*2770.9777*	*0.4938*
n	*(80.7639, 46.1019)*	*8648.1927*	*1.5412*
		$\sum_{j=1}^{n} f_j = 33667.7858$	$\sum_{j=1}^{n} Ve_j = 6$

Tabla 2.3 Valor esperado de la población.

La Ruleta se aplica de la siguiente manera:

- Calcular *Ve* para cada individuo de la población.
- Repetir *n* veces (*n* es el tamaño de la población):

- Generar un número real aleatorio *r* entre *0.0* y *n*.
- Ciclar a través de los individuos de la población sumando los *Ve* hasta que la suma sea mayor o igual a *r*.
- El individuo que haga que esta suma exceda el límite es seleccionado.

Un ejemplo de selección por ruleta se muestra a continuación de acuerdo a los valores esperados de la Tabla 2.3:

Sea r=3.42
Ciclando:

Individuo (i)	***Ve***	$\sum_{j=1}^{i} Ve_j$	
1	*0.3383*	*0.3383*	
2	*1.1907*	*1.5290*	
3	*1.8427*	***3.3717≥3.42***	***Individuo 3 es seleccionado***
4	*0.5930*	.	
…	*0.4938*	.	
n	*1.5412*	.	

Fig. 2.7 Ejemplo de selección por ruleta.

Una vez seleccionados los *n* padres, se procede a la cruza o recombinación de los mismos.

La cruza o recombinación, es el proceso por el cual un nuevo individuo es creado de la información contenida en 2 (o más) padres, y es considerada por muchos como una de las más importantes características en AG's. Los operadores de recombinación son usualmente aplicados probabilísticamente de acuerdo a un **porcentaje de cruza** p_c, el cual comúnmente se encuentra en el intervalo [*0.5, 1.0*]. Generalmente 2 padres son seleccionados y un número aleatorio es generado entre [*0.0, 1.0*), el cual es comparado con p_c. Si el valor es menor, 2 hijos son creados por recombinación de los padres; de otra modo ellos son creados asexualmente, es decir, los padres son copiados [2].

Es importante mencionar que las técnicas de cruzas son diferentes de acuerdo al tipo de representación que se esté utilizando en los individuos, esto es, existen técnicas de cruza para representación binaria, para representación entera, para representación en punto flotante e

incluso para representación de permutaciones, las cuales son muy diferentes para cada caso. Solo nos enfocaremos en la cruza para representación binaria, ya que ésta es la más empleada en AG's.

En el caso de la representación binaria, podemos realizar *cruza de un punto* (propuesta por Holland), la cual consiste en definir un punto de cruza en una posición aleatoria del cromosoma, el cual va a servir como divisor de las cadenas y definirá los segmentos a intercambiarse, gráficamente se puede apreciar en la Fig. 2.8.

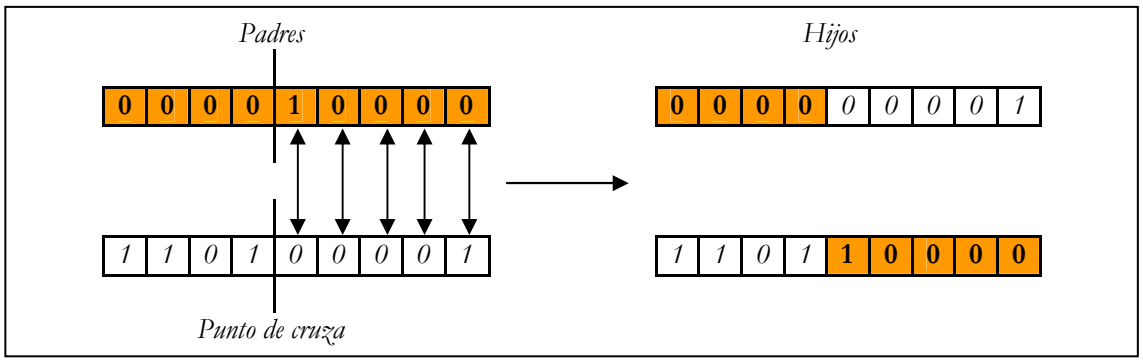


Fig. 2.8 Cruza de un punto.

Una versión generalizada de la cruza de un punto es la *cruza de n-puntos*, la cual consiste en definir *n* puntos de cruza en posiciones aleatorias a lo largo del cromosoma e intercambiar los segmentos de los padres alternadamente hasta obtener las nuevas cadenas binarias (hijos). Un ejemplo con n=2 (cruza de dos puntos) es mostrado en la Fig. 2.9.

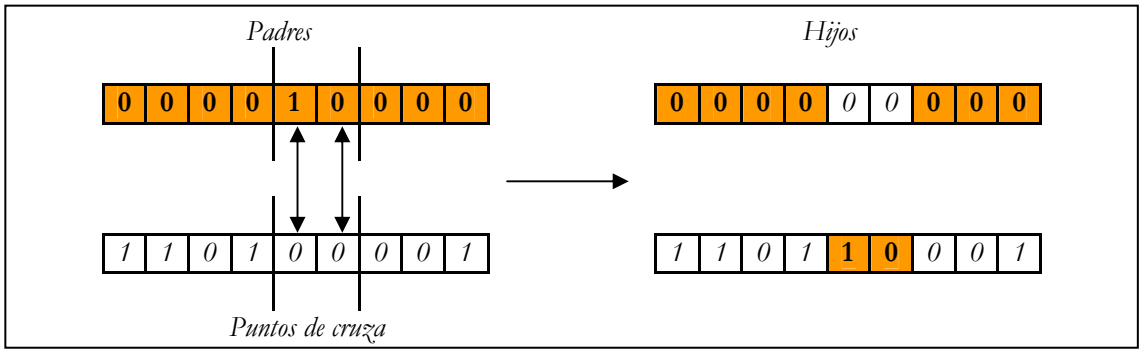


Fig. 2.9 Cruza de dos puntos.

La *cruza uniforme* se trata de una *cruza de n-puntos*, solo que el número de puntos de cruza no se fija previamente. Este tipo de cruza se implementa generando una cadena de *L* variables aleatorias (siendo *L* la longitud del cromosoma) con una distribución uniforme sobre [*0.0, 1.0*].

En cada posición, si el valor es menor al parámetro p (usualmente *0.5*), el gene es heredado del primer padre; de otro modo del segundo. El segundo hijo es creado usando el mapeo inverso. La Fig. 2.10 muestra un ejemplo de cruza uniforme.

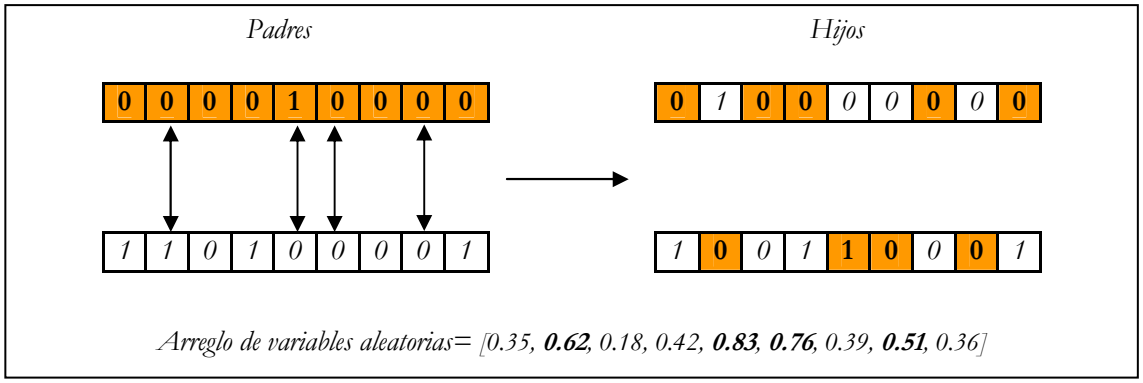


Fig. 2.10 Cruza uniforme.

Una vez efectuada la cruza, el siguiente operador a aplicar es la mutación. Al igual que las técnicas de cruza, existen técnicas de mutación para el caso de representación de permutaciones, representación entera, representación en punto flotante, etc., sin embargo solo trataremos la mutación para representación binaria.

La mutación es el nombre genérico dado a aquellos operadores de variación que utilizan solo un padre y crean un hijo aplicando algún tipo de modificación aleatoria a la representación (genotipo). El operador de mutación usado comúnmente para codificaciones binarias considera cada gene separadamente y permite a cada bit ser invertido (es decir, de *1* a *0* ó de *0* a *1*) con una pequeña probabilidad p_m. El número de valores modificados no es fijo, pero depende de una secuencia de números aleatorios, por tanto, para una codificación de longitud L, en promedio $L \cdot p_m$ valores serán cambiados. En la Fig. 2.11 se ilustra el caso donde el tercer, cuarto y octavo valor aleatorio generado son menores que la probabilidad de mutación establecida p_m [2].

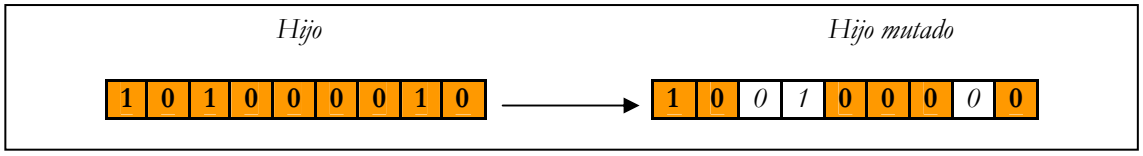


Fig. 2.11 Mutación para representación binaria.

Los operadores de cruza y mutación dotan a los AG's de la capacidad de explotación y exploración del espacio de búsqueda respectivamente, sin embargo, ha sido demostrado que un AG requiere elitismo (o sea, retener intacto al mejor individuo de cada generación) para poder converger al óptimo.

Cuando se está realizando una búsqueda, se denomina explotación al proceso de usar la información obtenida de los puntos visitados previamente para determinar qué lugares resultan más convenientes visitar a continuación. La exploración es el proceso de visitar nuevas regiones del espacio de búsqueda, para ver si se puede encontrarse algo prometedor. La exploración implica grandes saltos hacia lo desconocido y la explotación normalmente involucra movimientos finos. La explotación es buena para encontrar óptimos locales, y la exploración es buena para evitar quedar atrapados en óptimos locales [3].

El estudio de los AG's es amplio y existen muchas variantes del mismo, sin embargo, en este trabajo solo nos limitamos a tratar la versión con representación binaria, ya que es la más usual y además nos permitirá comprender más a fondo la versión conocida como Algoritmo Genético Compacto, la cual se pretende implementar en hardware. Aunque los AG's han sido y siguen utilizándose con mucho éxito, se han propuesto mejoras, e incluso versiones reducidas para reducir el costo computacional que ellos implican. También existen versiones combinadas de AG's con otros algoritmos bio-inspirados o heurísticos, enfocados ya sea a problemas de optimización numérica o combinatoria.

2.1.1.1 Algoritmos Genéticos Compactos

La población de un AG representa la información de las partes del espacio de búsqueda que ha visto con anterioridad. Los operadores de cruza y selección le dicen al AG como explotar esta información para generar soluciones nuevas y potencialmente buenas. A lo largo del tiempo, los investigadores han notado que la cruza tiende a disminuir la correlación de las dimensiones individuales (o genes) en las estructuras de la soluciones, mientras que la selección tiende a cambiar la naturaleza de la población recompensando a los genes más exitosos. De esta manera, nacieron un número de algoritmos que reemplazaron la población, la cruza y la selección con un número de acciones sobre distribuciones de probabilidad marginal en cada uno de los genes de la representación. La idea detrás de estos algoritmos se apoya en la

representación del estado actual de la búsqueda así como en la fracción de cada dimensión (o gene) en la población que tuvo un valor de "1". Usando solo ésta información, una nueva población puede ser generada imitando los efectos de muchas cruzas consecutivas. Alterando estas probabilidades de acuerdo a qué tan bien se desempeñaron ciertos genes en la competencia, estos algoritmos también podrían imitar el efecto de selección. El Algoritmo Genético Compacto (AGc) y el *Population-based Incremental Learning* (PBIL) son dos ejemplos de estos algoritmos simples, pero efectivos [4].

Los Algoritmos Genéticos Compactos son técnicas de búsqueda estocásticas robustas que imitan el comportamiento de un Algoritmo Genético Simple con cruza uniforme. Los AGc's representan la población mediante una distribución de probabilidad sobre el conjunto de posibles soluciones, es decir, un vector de probabilidad (*VP*). Este vector es procesado mediante reglas de actualización inspiradas en las operaciones típicas de selección y recombinación de los AG's [5]. El vector de probabilidad es inicializado a 0.5, y al finalizar el algoritmo solo debe contener 0's ó 1's, este vector representa la solución a la que ha llegado el AGc.

Parámetros del AGc:
n: tamaño de la población.
l: longitud del cromosoma.

1) Inicializar el Vector de Probabilidad (VP)
```
for i:=1 to l do VP[i]:=0.5;
```
2) Generar dos individuos del vector VP
```
a:=generar(VP);
b:=generar(VP);
```
3) Determinar el ganador y el perdedor
```
ganador,perdedor:=competencia(a,b);
```
4) Actualizar el VP con respecto al individuo ganador
```
for i:=1 to l do
   if ganador[i]≠perdedor[i] then
      if ganador[i]=1 then
         VP[i]:=VP[i]+1/n;
      else
         VP[i]:=VP[i]-1/n;
```
5) Verificar si el Vector VP ha convergido
```
for i:=1 to l do
   if VP[i]>0 and VP[i]<1 then
      return to step 2;
```
6) VP representa la solución final

Fig. 2.12 Pseudocódigo del AGc [7].

El pseudocódigo del AGc es presentado en la Figura 2.12. Los parámetros del AGc son: tamaño de la población (*n*) y longitud del cromosoma (*l*). La población es representada por un vector de probabilidades (*VP*) *l*-dimensional. El *VP[i]* es la probabilidad del bit de la i-ésima posición de un individuo, el cual será aleatoriamente tomado de la población. Primero, el *VP* es inicializado a (0.5,…, 0.5). Luego, los individuos a y b son generados de acuerdo al *VP*. Se determina cuál de los individuos es ganador y perdedor en base a sus valores de aptitud. Si ganador[i] ≠ perdedor[i], el *VP[i]* será actualizado de acuerdo al individuo ganador. Si ganador[i]=1 entonces el vector de probabilidad será incrementado en *1/n*. Si ganador[i]=0 entonces el vector de probabilidad será decrementado en *1/n*. Nótese que si ganador[i] = perdedor[i] entonces el vector de probabilidad no será actualizado. Este ciclo se repetirá hasta que cada elemento del *VP[i]* llegar a ser cero o uno. Finalmente, el *VP* representa la solución final [6].

Los AGc's son un tipo de *Estimation of Distribution Algorithm* (EDA) ó *Probabilistic Model Building Genetic Algorithm* (PMBA), los cuáles están englobados dentro del paradigma de la computación evolutiva. Como puede apreciarse en el AGc, los EDA's tienen la principal característica de trabajar con estimaciones de distribuciones.

Algunos investigadores han propuesto modificaciones al AGc con el fin de mejorar su capacidad de búsqueda, por ello, se ha propuesto incluir nuevas características en el algoritmo clásico, por ejemplo, la introducción de elitismo y mutación.

En la Fig. 2.13 se muestra el pseudocódigo de una versión del AGc que incluye Elitismo, Mutación y Muestreo. El elitismo consiste básicamente en mantener el mejor de los individuos generados para la siguiente generación, es decir, solo al inicio del algoritmo se generan dos individuos del vector de probabilidad, y para las siguientes generaciones, solo se genera uno (conservado el mejor individuo hasta el momento). La mutación consiste en realizar un pequeño cambio al individuo ganador y verificar si el ganador mutado es mejor, si esto se cumple, se sustituye al ganador por la versión mutada, de lo contrario, se mantiene al campeón. El muestreo consiste en reevaluar al individuo ganador cada cierto número de generaciones (tasa-muestreo), esto se aplica en el caso de que la función objetivo sea dinámica, es decir, que la función cambie en el tiempo, provocando que el óptimo se mueva. De éste pseudocódigo, emergen tres variantes del AGc: AGc con Elitismo (AGcE), AGc con Elitismo

y Mutación (AGcEM) y AGc con Elitismo, Mutación y Muestreo (AGcEMM). Para obtener el AGcE se debe omitir el punto 5) y el argumento tasa_muestreo en el punto 3). Para obtener el AGcEM solo se debe omitir el argumento tasa_muestreo. El pseudocódigo completo representa la versión AGcEMM.

1) Inicializar el Vector de Probabilidad (VP) – `l` es el número de bits del VP

```
for i:=1 to l do VP[i]:=0.5;
```

2) Generar dos individuos del vector VP

```
a:=generar(VP);
b:=generar(VP);
```

3) Determinar el ganador y el perdedor

```
ganador,perdedor:=evaluar(a,b,tasa-muestreo);
```

4) Actualizar el VP con respecto al individuo ganador - `n` es el número de individuos en la población simulada.

```
for i:=1 to l do
   if ganador[i]≠perdedor[i] then
       if ganador[i]=1 then
          VP[i]:=VP[i]+1/n;
       else
          VP[i]:=VP[i]-1/n;
```

5) Mutar al campeón y evaluar

```
if aptitud(a)> aptitud(b) then
    c:=mutar(a);evaluar(c);
    if aptitud(c)>aptitud(a) then a:=c;
else
    c:=mutar(b);evaluar(c);
    if aptitud(c)>aptitud(b) then b:=c;
```

6) Generar un individuo del vector

```
if ganador==a then b:=generar(VP);
else a:=generar(VP);
```

7) Verificar si el Vector VP ha convergido

```
for i:=1 to l do
   if VP[i]>0 and VP[i]<1 then
       return to step 3;
```

8) VP representa la solución final

Fig. 2.13 Pseudocódigo del AGc con Elitismo, Mutación y Muestreo (AGcEMM) [8].

Existen otras variantes que incorporan Elitismo no persistente, el cual consiste en reestablecer el individuo ganador cada cierto número de generaciones, es decir, olvidar el individuo elitista hasta el momento y generar otro a partir del vector de probabilidades (sin importar si es mejor o peor que el anterior).

Los AGc's tienen algunas ventajas, por ejemplo, no se necesita almacenar la población, sino solo un vector que simula esa población. Muchas de las operaciones se aplican a lo más a dos individuos y no a toda la población. Se tiene un criterio de convergencia definido, a diferencia de un AG, en el cual generalmente se propone un cierto número de iteraciones como criterio de convergencia. Debido a su simplicidad, es más fácil de implementar en hardware que un AG.

2.1.2 Estrategias Evolutivas

Las estrategias evolutivas (EE) fueron inventadas a principios de los 60s por Rechenberg y Schwefel, quiénes trabajaron en la Universidad Tecnológica de Berlín en una aplicación relacionada con la optimización de formas. Las EE ilustran una característica muy útil en la computación evolutiva: *auto-adaptación* de los parámetros de la estrategia. En general, la auto-adaptación significa que algunos parámetros del AE variarán durante una corrida de una manera específica: los parámetros están incluidos en los cromosomas y coevolucionan con las soluciones. Esta característica es inherente en las estrategias evolutivas modernas. Esto es, desde que el procedimiento fue detallado en 1977 muchas EE han sido auto-adaptables, y sobre los últimos diez años otros AEs han adoptado la auto-adaptividad [2]. La Tabla 2.4 muestra un resumen de las principales características de las EE.

Representación	*Vectores de valores reales*
Recombinación	*Discreta o intermedia*
Mutación	*Perturbación Gaussiana*
Selección de padres	*Aleatoria uniforme*
Selección de sobrevivientes	*(μ,λ) o $(\mu+\lambda)$*
Especialidad	*Auto-adaptación o mutación de los tamaños de paso*

Tabla 2.4 Resumen de las características de las EE.

La notación para las diferentes versiones de estrategias evolutivas es EE-$(\mu \overset{+}{,} \lambda)$, donde la letra μ es el número de padres, la λ el número de hijos y la EE señala que es una estrategia evolutiva.

La versión EE-$(\mu+\lambda)$ es aquella en la que no solamente un hijo es creado a la vez o en cada generación, sino que $\lambda \geq 1$ descendientes, y para mantener el tamaño de la población constante, los λ peores de todos los $\mu+\lambda$ individuos son descartados.

La versión EE-(μ, λ) es aquella en la cual la selección toma lugar entre los λ hijos solamente, mientras que sus padres son *olvidados* sin importar que tan buenos o malos hayan sido sus aptitudes con respecto a la nueva generación. Obviamente, esta estrategia depende de una descendencia excedente, es decir, sobre λ>μ en un sentido estricto darwiniano de selección natural [12].

La recombinación discreta es un operador para representaciones en punto flotante (reales). Se utiliza de manera análoga a los operadores para representación binaria, solo que en vez de considerar un bit como elemento mínimo para la recombinación, se utiliza un valor real (flotante). Si creáramos un hijo H_1 de los padres P_1 y P_2, los valores de los alelos para el gene i estarán dados por $H_{1\_i}=P_{1\_i}$ o P_{2_i} con igual probabilidad. La desventaja de este operador es que solo crea nuevas combinaciones de flotantes existente.

La recombinación intermedia es otro operador utilizado para la representación en punto flotante. En este caso, en cada gene, se crea un nuevo alelo, esto es, si suponemos que tenemos dos padres: $P_1=[v_1,\ldots,v_m]$ y $P_2=[w_1,\ldots,w_m]$, los cuales se cruzan en la posición k y con $a \in [0,1]$, entonces los hijos producidos serán:

$$H_1=[v_1,\ldots,v_k,w_{k+1}*a+v_{k+1}*(1-a),\ldots,w_m*a+v_m*(1-a)]$$

$$H_2=[w_1,\ldots,w_k,v_{k+1}*a+w_{k+1}*(1-a),\ldots,v_m*a+w_m*(1-a)]$$

La mutación no uniforme con distribución fija es utilizada en la representación en punto flotante. Esta es diseñada de manera que usualmente, pero no siempre, la cantidad de cambio introducida es pequeña. Esto se logra agregando al valor del gene actual una cantidad extraída aleatoriamente de una distribución Gaussiana con media cero y desviación estándar especificada por el usuario, y reduciendo luego el valor resultante al intervalo $[L_i, U_i]$ si es necesario. La distribución Gaussiana (o normal) tiene la propiedad de que aproximadamente dos tercios de las muestras extraídas caen dentro de una desviación estándar. Esto significa que muchos de los cambios realizados serán pequeños, pero hay una probabilidad distinta de cero de generar cambios muy grandes debido a que el extremo de la distribución nunca alcanza cero. Es normal aplicar este operador con una probabilidad de uno por gene, y en cambio el parámetro de mutación es usado para controlar la desviación estándar de la Gaussiana y consecuentemente la distribución de probabilidad de los tamaños de paso tomados [2].

La versión original EE-(1+1) usaba un solo padre y con él se generaba un solo hijo. Este hijo se mantenía si era mejor que el padre, o de lo contrario se eliminaba (selección extintiva). En la EE-(1+1), un individuo nuevo es generado usando:

$$\bar{x}^{t+1} = \bar{x}^{t} + N(0,\bar{\sigma}) \qquad (2.5)$$

Donde *t* se refiere a la generación (o iteración) en la que nos encontramos, y $N(0,\bar{\sigma})$ es un vector de números Gaussianos independientes con una media de cero y desviaciones estándar $\bar{\sigma}$.

Consideremos un ejemplo de tipo EE-(1+1) (estrategia evolutiva de dos miembros) para optimizar la siguiente función:

$$f(x_1, x_2) = 100(x_1^2 - x_2)^2 + (1 - x_1)^2$$

donde: $-2.048 \le x_1, x_2 \le 2.048$

Suponiendo que la población consiste del siguiente individuo (generado aleatoriamente):

$$(\bar{x}^{t}, \bar{\sigma}) = (-1.0, 1.0), (1.0, 1.0)$$

Y si las mutaciones fueran las siguientes:

$$x_1^{t+1} = x_1^t + N(0,1.0) = -1.0 + 0.61 = -0.39$$
$$x_2^{t+1} = x_2^t + N(0,1.0) = 1.0 + 0.57 = 1.57$$

Comparando el padre con el hijo:

Padre $f(x_t) = f(-1.0, 1.0) = 4.0$

Hijo $f(x_{t+1}) = f(-0.39, 1.57) = 201.416$

Dado que 201.416>4.0, el hijo reemplazará al padre en la siguiente generación (para maximización).

Una de las principales contribuciones de las estrategias evolutivas al campo de la computación evolutiva es la auto-adaptación. La auto-adaptación consiste en evolucionar no

sólo a las variables del problema, sino también a los parámetros mismos de la técnica (es decir, las desviaciones estándar).

Las desviaciones estándar se mutan usando la siguiente fórmula:

$$\sigma'(i) = \sigma(i) * e^{\tau' N(0,1) + \tau\, Ni(0,1)} \qquad (2.6)$$

donde τ y τ' son constantes de proporcionalidad que están en función de n.

Las estrategias evolutivas son interesantes, y existe mucha teoría acerca de ellas. Sin embargo utilizan operaciones complicadas desde el punto de vista de la implementación sobre hardware.

2.1.3 Programación Evolutiva

Lawrence J. Fogel propuso en los 60's una técnica denominada *programación evolutiva*, en la cual la inteligencia se ve como un comportamiento adaptativo [3]. La capacidad de predecir el ambiente fue considerada como un prerrequisito para la adaptividad, y por tanto para el comportamiento inteligente [2]. En la Tabla 2.5 se muestran las principales características de la programación evolutiva.

Representación	*Vectores de valores reales*
Recombinación	*Ninguna*
Mutación	*Perturbación Gaussiana*
Selección de padres	*Determinística (cada padre crea un hijo mediante mutación)*
Selección de sobrevivientes	*Probabilística ($\mu+\mu$)*
Especialidad	*Auto-adaptación de la mutación de los tamaños de paso (en meta programación evolutiva)*

Tabla 2.5 Reseña de la programación evolutiva.

La programación evolutiva enfatiza los nexos de comportamiento entre padres e hijos, en vez de buscar emular operadores genéticos específicos (como en el caso de los algoritmos genéticos).

El algoritmo básico de la programación evolutiva es el siguiente:

- Generar aleatoriamente una población inicial.
- Aplicar mutación a cada individuo de la población.

- Calcular la aptitud de cada hijo y se usa un proceso de selección mediante torneo (normalmente estocástico) para determinar cuáles serán las soluciones que se retendrán.

La programación evolutiva es una abstracción de la evolución al nivel de las especies, por lo que no se requiere el uso de un operador de recombinación (diferentes especies no se pueden cruzar entre sí). Asimismo, usa selección probabilística.

En el ejemplo clásico de programación evolutiva, los predictores fueron evolucionados en forma de máquinas de estados finitos. Una máquina de estados finitos es un transductor que puede ser estimulado por un alfabeto finito de símbolos de entrada y puede responder en un alfabeto finito de símbolos de salida. Este consiste de un número *S* de estados y un número de transiciones de estado. Las transiciones de estado definen funcionamiento de la máquina de estados finitos: dependiendo del estado actual y del símbolo de entrada actual, ellos definen un símbolo de salida y el próximo estado [2]. Un ejemplo de máquina de estados finitos es mostrado en la Fig. 2.14.

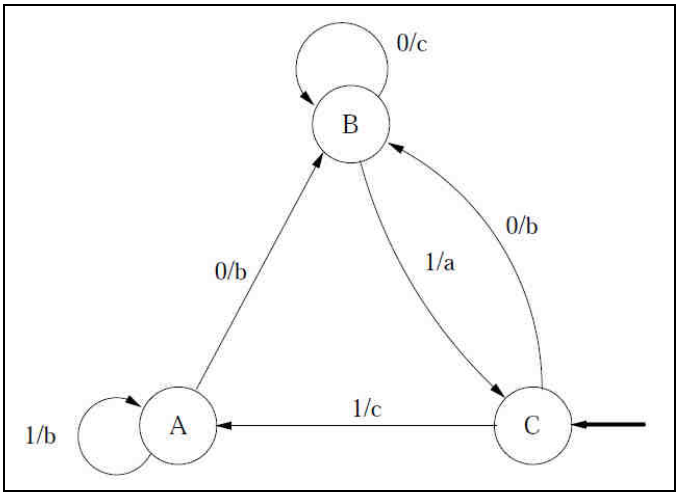


Fig. 2.14 Máquina de estados finitos de tres estados. Los símbolos a la izquierda de "/" son de entrada, y los de la derecha son de salida. El estado inicial es C.

La tabla de transiciones de la máquina de estados finitos de la Fig. 2.14 es la siguiente:

Estado actual	**C**	**B**	**C**	**A**	**A**	**B**
Símbolo de entrada	*0*	*1*	*1*	*1*	*0*	*0*
Estado siguiente	B	C	A	A	B	B
Símbolo de salida	b	a	c	b	b	c

En la máquina de estados finitos de la Fig. 2.14 pueden aplicarse cinco diferentes tipos de mutación: cambiar un símbolo de salida, cambiar una transición, agregar un estado, borrar un estado y cambiar el estado inicial. El objetivo es hacer que la máquina de estados finitos reconozca un cierto conjunto de entradas (o sea, una cierta expresión regular) sin equivocarse ni una sola vez [3].

Una simple tarea de predicción puede ser aprendida por una máquina de estados finitos para adivinar el siguiente símbolo de entrada en una cadena de entrada. Esto es, considerando n entradas, predecir el $(n+1)$-ésimo elemento, y acoplando esta predicción con el n-ésimo símbolo de salida. En este caso, el desempeño de una máquina de estados finitos es medida por el porcentaje de entradas donde $entrada_{n+1}=salida_n$. Claramente, esto requiere que el alfabeto de entrada y el de salida sean el mismo [2].

Algunas de las aplicaciones de la programación evolutiva son: predicción, generalización, juegos, control automático, problema del viajero, planeación de rutas, diseño y entrenamiento de redes neuronales, reconocimiento de patrones, etc. [3].

2.2 Otros

Además de los algoritmos evolutivos, existen otras técnicas de optimización que están inspiradas en otros aspectos biológicos, por ejemplo, la optimización por cúmulo de partículas, la cual está inspirada en el vuelo de las aves o la optimización por colonia de hormigas, la cual está inspirada en la forma en que estos insectos encuentran la ruta más corta para llegar a la comida. A continuación explicaremos brevemente algunos de los algoritmos bio-inspirados más comunes en el área de la computación.

2.2.1 Optimización por cúmulo de partículas

La optimización por cúmulo de partículas ó *particle swarm optimization* (PSO) en inglés, es un algoritmo de optimización muy popular, el cual fue introducido por primera vez en 1995 por Kennedy y Eberhart. El principio natural en el que se basa PSO es el comportamiento de una bandada de aves o de un banco de peces; supongamos que una de estas bandadas busca comida en un área y que solamente hay una pieza de comida en dicha área. Los pájaros no

saben dónde está la comida pero si conocen su distancia a las misma, por lo que la estrategia más eficaz para hallar comida es seguir al ave que se encuentre más cerca de ella. PSO emula este escenario para resolver problemas de optimización. Cada solución (partícula) es un "ave" en el espacio de búsqueda que está siempre en continuo movimiento y que nunca muere. En PSO, una posible solución es representada como la posición de una partícula, la trayectoria de cada partícula en el espacio de búsqueda es ajustada por variaciones dinámicas de la velocidad de cada partícula, de acuerdo a la mejor posición que ha encontrado la partícula hasta el momento y a la solución óptima global (o local, de acuerdo a un vecindario definido por el usuario) encontrada por el cúmulo o enjambre. Cada partícula se mueve siempre hacia la mejor solución encontrada por sí misma y a una solución óptima global (o local). Al final, el enjambre encuentra una buena solución para el problema [13, 14].

PSO puede utilizar dos tipos de representación para la posición de sus partículas, la binaria y la real. El pseudocódigo de PSO en su versión binaria es dado a continuación:

```
LOOP
   FOR i=1 TO número de partículas
      IF f(x_i)>f(p_i) THEN DO            //f() evalua la aptitud
         FOR d=1 TO dimensiones
            p_id = x_id                   // p_i es la mejor posición experimentada por la partícula i
         NEXT d
      END DO
      g=i;
      FOR j=índices de los vecinos
         IF f(p_j)>f(p_g) THEN g=j;       //g es el índice de la mejor partícula del vecindario
      NEXT j
      FOR d=1 TO dimensiones
         v_id(t) = v_id(t-1) + φ1(p_id - x_id(t-1)) + φ2(p_gd - x_id(t-1))
         v_id ∈ (-V_max, +V_max)
         IF ρ_id < s(v_id(t)) THEN x_id(t)=1; ELSE x_id(t)=0;
      NEXT d
    NEXT i
UNTIL Criterio de finalización
```

donde: $x_{id}(t)$ es el estado actual del *d*-ésimo bit de la partícula *i*, $v_{id}(t-1)$ es una medida de la inclinación para poner uno o cero (representa la velocidad de la partícula *i*), p_{id} es el estado actual del *d*-ésimo bit de la mejor posición experimentada por la partícula *i*, p_g es la mejor partícula del vecindario. φ representa un número aleatorio positivo con distribución uniforme

y un límite superior predefinido, a menudo se establece que la suma de las dos φ sumen 4.0. ρ_{id} es d-ésimo número aleatorio de un vector de números aleatorios *i* con distribución uniforme entre [0.0, 1.0]. $V_{\max}$ es un parámetro constante que puede ser definido al inicio del algoritmo y limita a v_{id}; a menudo es establecido a 4.0 y funciona de manera similar al porcentaje de mutación en los algoritmos genéticos. $s(v_{id}(t))$ representa la evaluación de $v_{id}(t)$ en la función sigmoide, esto es:

$$s(v_{id}) = \frac{1}{1+e^{-v_{id}}} \qquad (2.7)$$

La función sigmoide contrae su entrada a [0.0, 1.0] y sus propiedades la hacen adecuada para ser usada como umbral de probabilidad [15]. En PSO suelen usarse vecindarios donde se localiza la mejor partícula de cada uno, a este tipo de PSO se le conoce como PSO versión local. Si el vecindario lo extendiéramos a todo el cúmulo, estaríamos hablando de un PSO versión global.

PSO en su versión de números reales es algo parecido a la versión binaria, el pseudocódigo es el siguiente:

```
LOOP
   FOR i=1 TO número de partículas
      IF f(x_i)>f(p_i) THEN DO            //f() evalua la aptitud
         FOR d=1 TO dimensiones
            p_id = x_id                   // p_i es la mejor posición experimentada por la partícula i
         NEXT d
      END DO
      g=i;
      FOR j=índices de los vecinos
         IF f(p_j)>f(p_g) THEN g=j;       //g es el índice de la mejor partícula del vecindario
      NEXT j
      FOR d=1 TO dimensiones
         v_id(t) = v_id(t-1) + φ1(p_id - x_id(t-1)) + φ2(p_gd - x_id(t-1))
         v_id ∈ (-V_max, +V_max)
         x_id(t) = x_id(t-1) + v_id(t)
      NEXT d
   NEXT i
UNTIL Criterio de finalización
```

En este pseudocódigo, las posiciones de las partículas representan directamente una posible solución de un problema dado, por tanto, cada subíndice *d* representa el *d*-ésimo valor de un vector de números reales. A diferencia de PSO versión binaria, la actualización de la posición de la partícula se lleva a cabo directamente (sin aplicar la función sigmoide) mediante $x_{id}(t) = x_{id}(t-1) + v_{id}(t)$.

En PSO las posiciones y velocidades son usualmente inicializadas aleatoriamente. Diversos experimentos han mostrado que utilizar tamaños de cúmulo de entre 10 y 50 partículas, generalmente da buenos resultados. Para el tamaño de los vecindarios se puede utilizar del 10% al 20% de partículas del total del cúmulo (para el caso de PSO versión local).

Shi y Eberhart encontraron que la introducción de un peso inercial *w* tiene una mejora significante en el desempeño de PSO. Entonces el cálculo de la velocidad es modificado de la siguiente forma:

$$v_{id}(t) = wv_{id}(t-1) + \varphi_1(p_{id} - x_{id}(t-1)) + \varphi_2(p_{gd} - x_{id}(t-1)) \qquad (2.8)$$

w es el peso inercial de la velocidad y generalmente su valor es de 0.9 [13].

Actualmente existen algunas variantes de PSO, sin embargo, los algoritmos mostrados anteriormente son las versiones clásicas.

2.2.2 Sistema inmune artificial

El sistema inmune biológico es un sistema robusto, complejo y adaptativo que defiende al cuerpo de agentes patógenos extraños. Es capaz de clasificar todas las células (o moléculas) dentro del cuerpo como células propias o intrusas. Esto lo hace con ayuda de una fuerza de trabajo distribuida que tiene la inteligencia de tomar acción desde una perspectiva local y también global usando su red de mensajeros químicos para la comunicación. Hay dos ramas principales del sistema inmune. El sistema inmune innato es un mecanismo constante que detecta y destruye ciertos organismos invasores, mientras que el sistema inmune adaptativo responde previamente a células extrañas y elabora una respuesta que puede permanecer en el cuerpo por un largo periodo de tiempo. Esta valiosa información sobre el procesamiento de

los sistemas biológicos ha llamado la atención de las ciencias de la computación en años recientes [16].

El estudio y diseño de sistemas inmunes artificiales es un área relativamente nueva de la investigación que trata de construir sistemas computacionales inspirados en el sistema inmune biológico natural. Existen muchas características computacionales deseables en un sistema inmune biológico que pueden ser usados para resolver problemas computacionales. Un algoritmo/modelo de sistema inmune artificial implementa una o más de estas características. Las cuatro formas de un algoritmo de sistema inmune artificial reportadas en la literatura especializada son: modelo de red inmune, selección negativa, selección clonal y teoría del peligro.

Antes de continuar es necesario aclarar algunos términos usados en el área:

Células-B y células-T: Estas representan la mejor población de linfocitos(células encargadas de la inmunidad). Las células son producidas por la médula ósea y están inicialmente inertes, es decir, no son capaces de ejecutar sus funciones. Llegan a convertirse en componentes inmunes cuando experimentan un proceso de maduración. Para las células-B, el proceso de maduración ocurre en la médula ósea y para las células-T, tienen que migrar primero al timo, que es donde maduran. En general, un linfocito maduro puede ser considerado como un detector que puede detectar antígenos específicos. Hay billones de estos detectores circulando en el cuerpo constituyendo un sistema efectivo, anómalamente distribuido y de respuesta.

Discriminación de lo propio y lo no propio: Durante el proceso de maduración en el timo, la célula-T experimenta un proceso de selección que asegura que sea capaz de reconocer péptidos no propios.

El ***modelo de red inmune*** fue propuesto por Jerne. Esta teoría propone que el sistema inmune mantenga una red idiotípica de células interconectadas para reconocimiento de antígenos. Estas células estimulan y suprimen cada una de tal forma que conducen a la estabilización de la red. La formación de tal red es posible debido a la presencia de parátopos (partes de las células-B que se encargan de la detección de elementos alcanzables por la célula) e idiótopos (elementos de una célula que son detectados por parátopos de otras células) sobre cada célula del anticuerpo. El parátopo presente sobre una célula-B es reconocido por otros

idiótopos de células-B por lo que cada célula reconoce y es reconocida. En esta red dos células son conectadas si las afinidades que comparten exceden un cierto umbral y la fuerza de las conexiones es directamente proporcional a la afinidad que comparten.

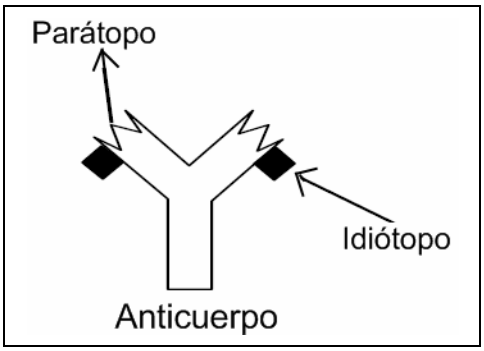


Fig. 2.15 Presencia de parátopo e idiótopo sobre un anticuerpo.

Desde el punto de vista de formación de redes hay dos cosas que son muy importantes: unión antígeno-anticuerpo y unión anticuerpo-anticuerpo. Esta red idiotípica puede también ser considerada como la disposición de capacidades cognitivas que la hacen semejante a las redes neuronales.

El propósito de la ***selección negativa*** es proporcionar tolerancia a las células propias. Se trata de la habilidad del sistema inmune para detectar antígenos desconocidos mientras que no reaccionen a células propias. Durante la generación de células-T, los receptores son creados a través de un proceso de re-ordenamiento genético pseudo-aleatorio. Luego, ellos experimentan un proceso censurado en el timo, llamado selección negativa. Allí, las células-T que reaccionan en contra de proteínas propias son destruidas; de esta manera, solo aquellas que no se unan a proteínas propias se les permiten salir del timo. Aquellas células-T maduras circulan a lo largo y ancho de todo el cuerpo para desempeñar funciones inmunológicas y proteger al cuerpo contra agentes extraños. Este algoritmo fue dado por Forest et al. cuyos pasos principales son:

1) En la fase de generación, los detectores son generados por algún proceso aleatorio y censurado tratando de detectar muestras propias como es mostrado en la Fig. 2.16.
2) Aquellos candidatos que detectan son eliminados y el resto son guardados como detectores.

3) En la etapa de detección, la colección de detectores (o conjunto de detectores) es utilizada para verificar si una instancia de datos de entrada es propia o no propia como es mostrado en la Fig. 2.17.

4) Si se encuentra algún detector, este es reclamado como anómalo o no propio.

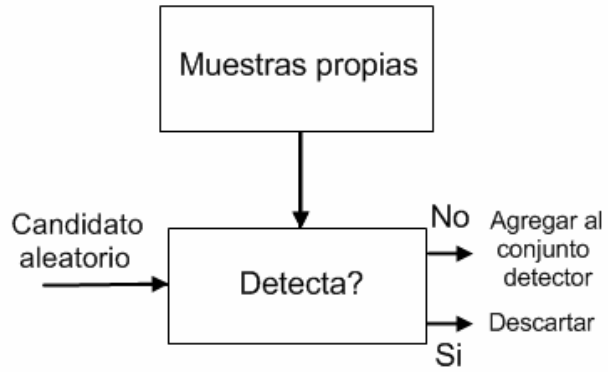


Fig. 2.16 Fase de generación.

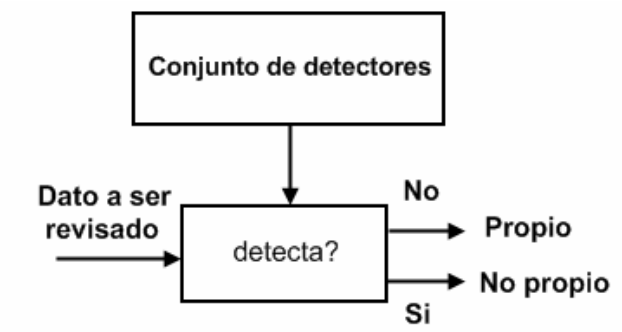


Fig. 2.17 Etapa de monitoreo.

El principio de ***selección clonal*** de un sistema inmune artificial describe como las células inmunes eliminan un antígeno extraño y es un algoritmo de aproximación simple pero eficiente para alcanzar soluciones óptimas. El algoritmo básico fue aplicado primeramente por Charsto et al. para resolver problemas de optimización. Los pasos involucrados en la selección clonal son:

1) Inicializar un número de anticuerpos (células inmunes) los cuales representan la población inicial.

2) Cuando un antígeno o patógeno invade el organismo; un número de anticuerpos que reconoce aquellos antígenos sobrevive. En la Fig. 2.18 solo el anticuerpo C es capaz de reconocer el antígeno "3" tal que su estructura se ajusta a una porción del patógeno. Así la aptitud del anticuerpo C es más alta que la de los otros.

3) Las células inmunes reconocen los antígenos experimentando reproducción celular. Durante la reproducción, las células somáticas se reproducen en forma asexual, es decir, no hay cruza de material genético durante la mitosis de una célula. Las nuevas células son copias (clones) de sus padres como es mostrado en el anticuerpo C de la Fig. 2.18.

4) Una porción de células clonadas experimentan un mecanismo de mutación el cual es conocido como hipermutación somática.

5) La afinidad de cada célula con otras es una medida de similitud entre ellas. Es calculada por la distancia entre dos de ellas. Los anticuerpos exhiben en una memoria la respuesta que tiene en promedio una afinidad más alta que aquella respuesta primaria anterior. Este fenómeno es llamado maduración de la respuesta inmune. Durante el proceso de mutación, la aptitud tanto como la afinidad de los anticuerpos es cambiada. En cada iteración después de clonar y mutar aquellos anticuerpos que tienen aptitud más alta y afinidad más alta, son elegidos para entrar a la piscina de células eficientes. Aquellas células con baja afinidad o receptores auto-reactivos deben ser eliminadas.

6) En cada iteración entre células inmunes eficientes, algunas llegan a ser células efectoras (células de plasma), mientras que otras son mantenidas como células de memoria. Las células efectoras secretan anticuerpos y las células de memoria tienen un tiempo de vida más largo así que actúan más rápido o más efectivamente en un futuro cuando el organismo es expuesto al mismo patógeno.

7) El proceso continua hasta que una condición de finalización es alcanzada, de lo contrario los pasos del 2 al 7 son repetidos.

El algoritmo de selección clonal tiene muchas características interesantes tales como un tamaño de población que es dinámicamente ajustable, exploración del espacio de búsqueda,

localización de varios óptimos, capacidad de mantener soluciones óptimas locales y un criterio de paro definido.

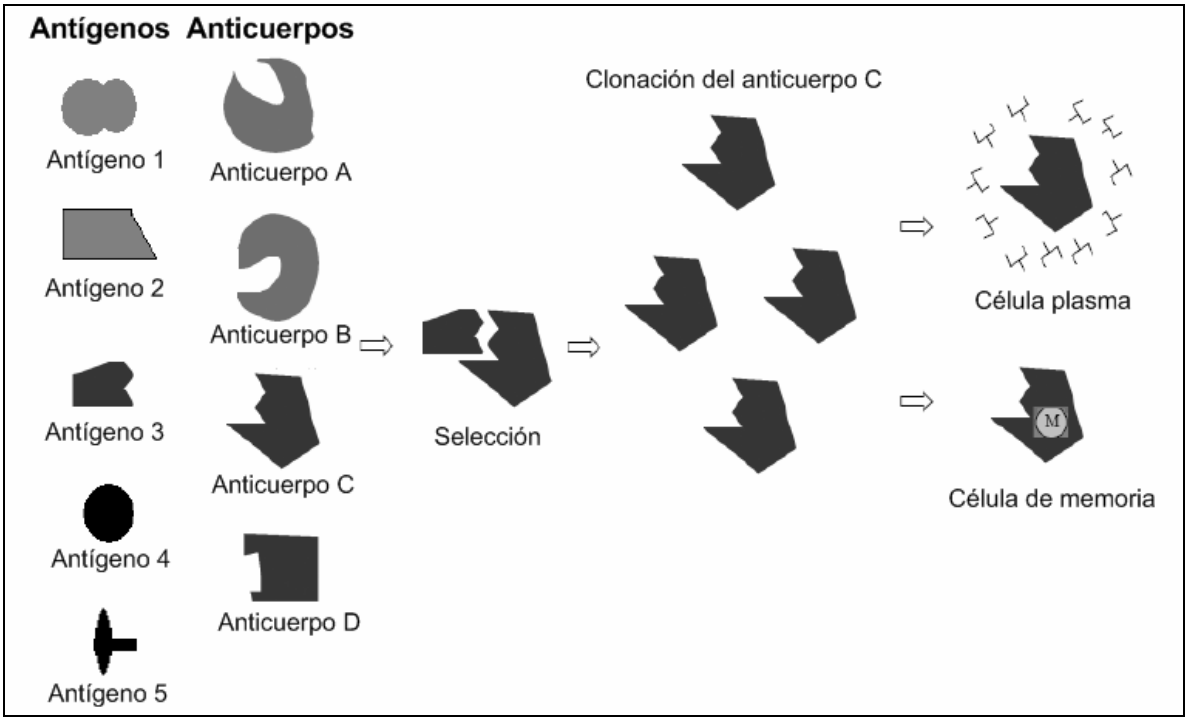


Fig. 2.18 Algoritmo de selección clonal básico

La ***teoría del peligro*** fue dada por Matzinger en 1994. Para que pueda funcionar apropiadamente el sistema inmune, es muy importante que solo las células "correctas" sean detectadas, de otro modo esto podría conducir a una reacción auto-inmune auto-destructiva. La inmunología clásica estipula que una respuesta inmune es disparada cuando el cuerpo encuentra algo no propio o extraño. No se ha entendido del todo como esta discriminación propia-impropia se lleva a cabo, pero muchos inmunólogos creen que la diferencia entre ellas es adquirida en la vida temprana. Se pensó que el proceso de maduración juega un rol importante para lograr auto-tolerancia eliminando aquellas células-B y células-T que reaccionan en sí mismas. Además una señal de confirmación es requerida, es decir, para la activación de la célula-B o célula-T (asesino), un linfocito-T (auxiliar) debe ser activado también. Esta activación dual es una protección más contra el cambio de reacción accidental en sí mismo.

La teoría del peligro toma cuidado de los invasores "no propios pero inofensivos" y de "propios pero dañinos" en nuestro sistema. La idea central es que el sistema inmune no

responda a los no propios a excepción de que dañen. Prácticamente no se necesita atacar todo lo extraño, algo que parece ser soportado por los contraejemplos anteriores. En esta teoría, el peligro es medido por el daño a las células indicado por señales de alarma que son enviadas cuando las células mueren por muerte natural. Como se muestra en la Fig. 2.19, una célula que está al borde de la muerte envía una señal de alarma, después de lo cual los antígenos del vecindario son capturados por células presentadoras de antígenos como los macrófagos, los cuales luego viajan al nodo linfático local y presentan los antígenos a los linfocitos. Esencialmente, las señales de peligro establecen que la detección de antígenos dentro de la zona de peligro es estimulada y experimentan un proceso de expansión clonal. Aquellos que no detectan o están muy lejos no son estimulados.

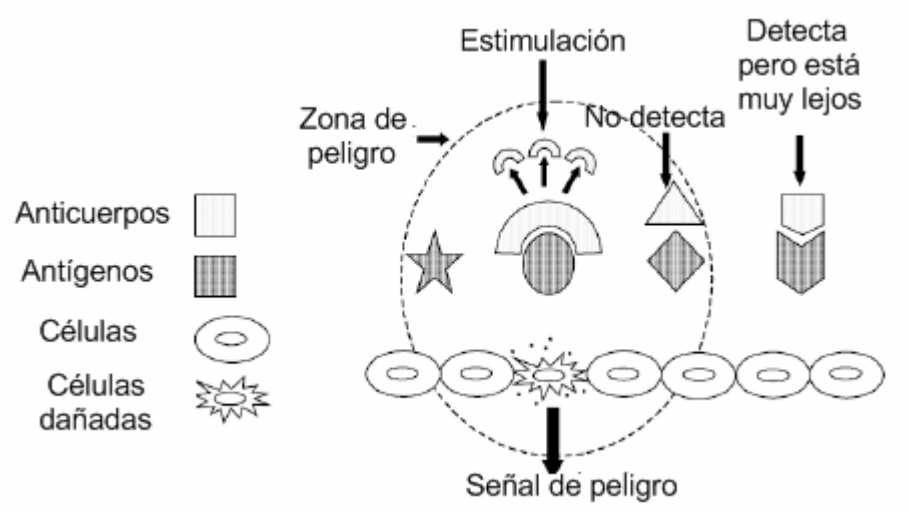


Fig. 2.19 Principio de la teoría del peligro

De acuerdo a la teoría del peligro de Bretscher y Chon se propusieron dos modelos de señales.

Señal 1: esta es usada para reconocimiento de antígenos. Básicamente para determinar la célula que es una célula extraña.

Señal 2: esta es usada para co-estimulación. Esto se refiere a que la célula es realmente peligrosa. Por tanto, de acuerdo al modelo de la señal dos de la teoría del peligro se llevan a cabo tres pasos:

1. Llega a ser activada si se reciben las señales una y dos juntas. Muere si se recibe la señal uno en ausencia de la señal dos. Ignora la señal dos sin la señal uno.
2. Acepta la señal dos solo de células presentadoras de antígenos. La señal uno puede venir de cualquier célula.
3. Después de la activación se regresa al estado de reposo después de un corto periodo de tiempo.

El reto es claramente definir una señal de peligro adecuada. La señal de peligro ayuda a identificar que subconjunto de vectores característicos es de interés.

Algunas áreas de aplicación del sistema inmune artificial incluyen problemas de optimización, seguridad computacional, diseño de detección de intrusos, detección de fallas, tolerancia a fallas, reconocimiento de patrones, redes de sensores, diseño de sistemas de recomendación, aprendizaje distribuido, etc. [17].

2.2.3 Algoritmo de colonia de hormigas

La metáfora natural sobre la cual los algoritmos de hormigas están basados es en las colonias de hormigas. Las hormigas reales son capaces de encontrar el camino más corto desde la fuente de alimento hasta su nido sin usar pistas visuales mediante la explotación de la información de la feromona. Mientras caminan, las hormigas depositan feromona sobre el suelo, y persiguen probabilísticamente la feromona depositada previamente por otras hormigas. La forma en que las hormigas explotan la feromona para encontrar el camino más corto entre dos puntos es mostrado en la Fig. 2.20.

Considere las Fig. 2.20A: Las hormigas llegan a un punto donde deben decidir si ir por la derecha o la izquierda. Como no tienen pista alguna de cual es la mejor elección, la escogen aleatoriamente. Se puede esperar en promedio, que la mitad de las hormigas se vayan por la izquierda y el resto por la derecha. Esto ocurre para las hormigas que se mueven de izquierda a derecha (aquellas marcadas con L) y para las que se mueven de derecha a izquierda (marcadas con R). Las Figs. 2.20B y 2.20C muestran que ocurre en los siguientes instantes, suponiendo que todas las hormigas caminan a la misma velocidad. El número de líneas punteadas es aproximadamente proporcional a la cantidad de feromona que las hormigas han depositado

sobre el suelo. Como el camino inferior es más corto que el superior, más hormigas lo visitarán en promedio, y por tanto la feromona se acumulará más rápido. Después de un periodo corto de transición, la diferencia en la cantidad de feromona de ambos caminos es suficientemente grande como para influir en la decisión de las nuevas hormigas durante su recorrido en el sistema (esto es mostrado en la Fig. 2.20D). De ahora en adelante, las nuevas hormigas preferirán probabilísticamente la elección del camino inferior, desde que al ubicarse en el punto de decisión perciben una cantidad más alta de feromona sobre el camino inferior. Ésta se incrementa, con un efecto de retroalimentación positivo donde cierto número de hormigas seleccionan el camino inferior que es más corto, luego, todas las hormigas usarán el camino más corto [19].

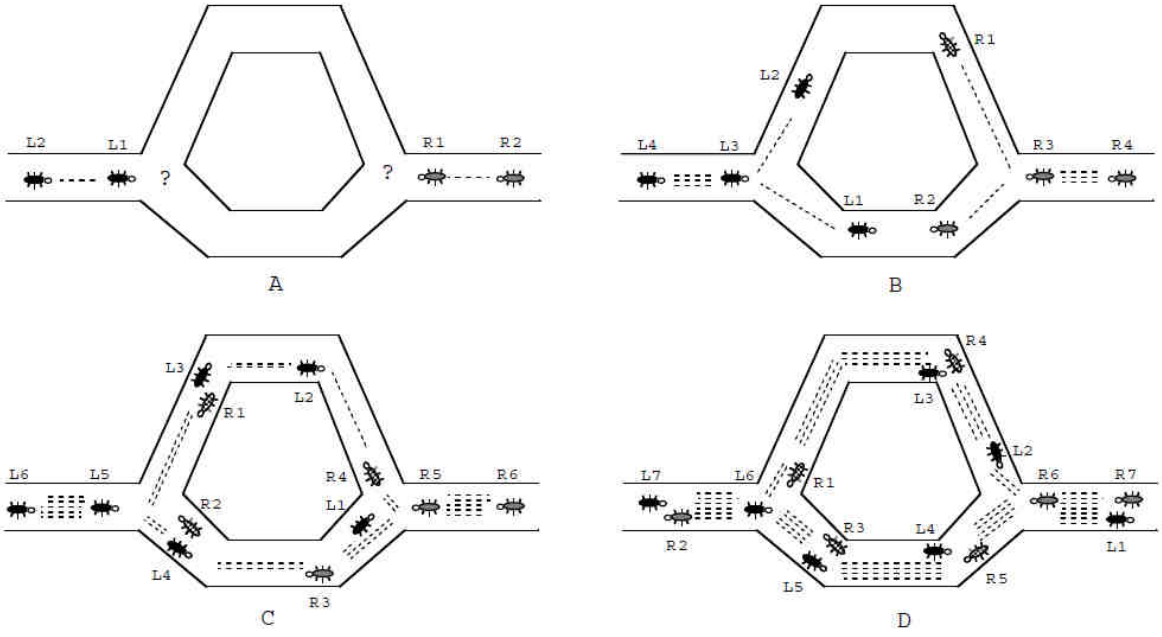


Fig. 2.20 Comportamiento de las hormigas reales.

El comportamiento anteriormente descrito de las hormigas reales ha inspirado a ***Ant System*** (AS), un algoritmo en el cual un conjunto de hormigas artificiales cooperan con la solución del problema intercambiando información vía feromona depositada sobre las aristas de un grafo. *Ant System* ha sido aplicado a problemas de optimización combinatoria tales como el problema del agente viajero y el problema de asignación cuadrática. Dado un conjunto de ciudades conectadas entre sí (grafo completo), con un costo asociado a cada arista, el problema del agente viajero consiste en encontrar la ruta con el menor costo saliendo de una ciudad y

regresando a las misma visitando todas las ciudades (sin que se repitan). Existen dos versiones del problema del agente viajero: la versión simétrica, donde el costo que implica ir de una ciudad A a una ciudad B ($\delta(A,B)$) es el mismo que ir de la ciudad B a la ciudad A ($\delta(B,A)$), esto es, $\delta(A,B)=\delta(B,A)$; y la versión asimétrica, que es donde el costo de ir de la ciudad A a la ciudad B es diferente que ir de la ciudad B a la ciudad A, es decir, $\delta(A,B)\neq\delta(B,A)$.

Ant System opera de la siguiente manera: Cada hormiga genera un recorrido completo seleccionando las ciudades de acuerdo a *una regla de transición de estados* probabilística: las hormigas prefieren moverse a ciudades conectadas por aristas más cortas con una cantidad más alta de feromona. Una vez que todas las hormigas completaron sus recorridos, una *regla de actualización global de feromona* es aplicada: una fracción de feromona se evapora sobre todas las aristas (las aristas que no son actualizadas llegan a ser menos deseables) y luego, cada hormiga deposita una cantidad de feromona sobre las aristas que pertenecen a su recorrido en proporción a que tan corto fue su recorrido (en otras palabras, las aristas que pertenecen a caminos cortos son las que reciben más feromona). Luego, el proceso es repetido.

La regla de transición de estados utilizada en *Ant System* obtiene la probabilidad de que una hormiga *k* en una ciudad *r* se mueva a la ciudad *s*:

$$p_k(r,s)=\begin{cases}\dfrac{[\tau(r,s)]\cdot[\eta(r,s)]^{\beta}}{\sum_{u\in J_k(r)}[\tau(r,s)]\cdot[\eta(r,s)]^{\beta}} & si\ s\in J_k(r)\\ 0 & en\ caso\ contrario\end{cases} \qquad (2.9)$$

donde $\tau(r,s)$ es la cantidad de feromona depositada entre la ciudad *r* y la *s* (arista (*r*,*s*)); $\eta(r,s)$ es el inverso del costo de ir de una ciudad *r* a una ciudad *s* ($\delta(r,s)$), esto es, $\eta(r,s)=1/\delta(r,s)$; $J_k(r)$ es el conjunto de ciudades que faltan ser visitadas por la hormiga *k* estando en la ciudad *r*; y β es un parámetro que determina la importancia relativa de la feromona contra la distancia ($\beta>0$).

La regla de actualización global se aplica sobre todas las aristas una vez que todas las hormigas han construido sus recorridos, la regla de actualización es:

$$\tau(r,s) \leftarrow (1-\alpha)\cdot\tau(r,s) + \sum_{k=1}^{m} \Delta\tau_k(r,s)$$

$$donde \quad \Delta\tau_k(r,s) = \begin{cases} 1/L_k & si\ (r,s) \in recorrido\ hecho\ por\ la\ hormiga\ k \\ 0 & en\ caso\ contrario \end{cases} \qquad (2.10)$$

donde $0 < \alpha < 1$ es un parámetro de decaimiento de feromona, L_k es el costo total del recorrido realizado por la hormiga *k*, y *m* es el número de hormigas [19].

Aunque *Ant System* fue útil para encontrar soluciones buenas u óptimas para configuraciones pequeñas del problema del agente viajero (hasta 30 ciudades), el tiempo requerido para encontrar tales resultados la hacen infactible para problemas más grandes.

Con el fin de compensar algunas debilidades de *Ant System*, se han propuesto algunas variantes, por ejemplo *Ant Colony System* (ACS), la cual difiere de *Ant System* en tres aspectos principales: la regla de transición de estados proporciona una forma directa de balance entre la exploración de nuevas aristas y la explotación de conocimiento acumulado y a priori acerca del problema, la regla de actualización global es aplicada solo a las aristas que pertenecen al recorrido de la mejor hormiga, y mientras las hormigas construyen una solución, una *regla de actualización local* de feromona es aplicada. Otra variante propuesta fué *Max-Min Ant System* cuyas características principales son: explotar las mejores soluciones encontradas durante una iteración o durante la ejecución del algoritmo, después de cada iteración solo una hormiga (la mejor hormiga de la iteración o la mejor desde el inicio) agrega feromona; para evitar el estancamiento de la búsqueda, los rastros de feromona son limitados a un intervalo $[\tau_{\min}, \tau_{\max}]$ y adicionalmente, se inicializan los rastros de feromona en $\tau_{\max}$, logrando de esta forma una mayor exploración de las soluciones desde el inicio del algoritmo [19, 20].

2.2.4 Evolución diferencial

La evolución diferencial es un algoritmo evolutivo propuesto para resolver problemas de optimización, principalmente en espacios de búsqueda continuos. La evolución diferencial comparte similitudes con los algoritmos evolutivos tradicionales. Sin embargo, esta no usa codificación binaria como un algoritmo genético simple y tampoco utiliza una función de

densidad de probabilidad para auto-adaptar sus parámetros como en las estrategias evolutivas. En lugar de eso, la evolución diferencial utiliza mutación basada sobre la distribución de las soluciones en la población actual. De esta forma, las direcciones de búsqueda y los tamaños de paso dependen de la localización de los individuos seleccionados para calcular los valores de mutación.

Existe una nomenclatura desarrollada para hacer referencia a las diferentes variantes de evolución diferencial. La más popular es llamada "DE/rand/1/bin", donde DE significa *Differential Evolution*, la palabra *rand* indica que los individuos seleccionados para calcular los valores de mutación son seleccionados aleatoriamente, "1" es el número de parejas de soluciones seleccionadas y "bin" significa que se utiliza recombinación binomial.

Existe un parámetro llamado "*CR*", el cual controla la influencia del padre en la generación del hijo. Valores más altos implican menos influencia del padre. El parámetro "*F*" controla la influencia del conjunto de parejas de soluciones seleccionadas para calcular el valor de mutación. Es importante notar que al incrementar el tamaño de la población o el número de parejas de soluciones para calcular los valores de mutación, también incrementará la diversidad de posibles movimientos, promoviendo la exploración del espacio de búsqueda, sin embargo, la probabilidad de encontrar la dirección de búsqueda correcta disminuye considerablemente. Por tanto, el balance entre el tamaño de población y el número de diferencias determina la eficiencia del algoritmo.

El pseudocódigo para la evolución diferencial "DE/rand/1/bin" se muestra en la Fig. 2.21, donde $\bar{x} \in \Re^n$ es un vector solución $\bar{x} = [x_1, x_2, ..., x_n]$, y cada $x_i, i = 1,...,n$ está delimitada por el intervalo $L_i \leq x_i \leq U_i$; *randint(min,max)* es una función que regresa un número entero entre *min* y *max*; *D* es la dimensionalidad del problema; *G* es la generación actual; *GEN_MAX* es el número máximo de generaciones; *NP* es el tamaño de la población; j_{rand} es un número entero entre *1* y *D*; $u_{i,j,G}$ es un valor mutado de $x_{i,j,G}$; *i* representa el i-ésimo vector de una colección de vectores; *j* señala la posición de un elemento de un vector; *rand[0,1)* es una función que regresa un número real entre *0* y *1*; r_1, r_2, r_3 son individuos seleccionados aleatoriamente para este caso (rand) y diferentes entre sí, en el caso de una versión *best*, r_1 debe ser el mejor individuo, mientras el resto debe ser escogido aleatoriamente (todos diferentes).

```
BEGIN
   G=0
   Crear una población inicial aleatoriamente  x_{i,G}  ∀i, i = 1,..., NP
   Evaluar  f(x_{i,G}) ∀i, i = 1,..., NP
   FOR G=1 TO GEN_MAX DO
      FOR i=1 TO NP DO
          Seleccionar aleatoriamente  r_1 ≠ r_2 ≠ r_3
          j_rand = randint(1, D)
          FOR j=1 TO D DO
             IF  (rand_j[0,1) < CR ó j = j_rand)  THEN
                  u_{i,j,G+1} = x_{r_3,j,G} + F*(x_{r_1,j,G} − x_{r_2,j,G})
             ELSE
                  u_{i,j,G+1} = x_{i,j,G}
             END IF
          END FOR
          IF  f(u_{i,G+1}) ≤ f(x_{i,G+1})  THEN
               x_{i,G+1} = u_{i,G+1}
          ELSE
               x_{i,G+1} = x_{i,G}
          END IF
      END FOR
      G=G+1
   END FOR
END
```

Fig. 2.21 Pseudocódigo para evolución diferencial versión DE/rand/1/bin.

El valor de CR debe ser un número real entre [0,1] y el parámetro CR se recomienda que esté entre [0.3,0.9] y se debe generar un valor nuevo en cada generación. En la Tabla 2.6 se muestran algunas variantes del algoritmo de evolución diferencial. j_r es un número entero aleatorio generado de entre [1,*D*], siendo D el número de variables del problema. $U_j(0,1)$ es un número real generado aleatoriamente entre 0 y 1. Ambos números se generan con una distribución uniforme [21].

Nomenclatura	Variante
DE/rand/p/bin	$u_{i,j} = \begin{cases} x_{r_3,j} + F \cdot \sum_{k=1}^{p} (x_{r_1^p,j} - x_{r_2^p,j}) & Si\ U_j(0,1) < CR\ ó\ j = j_r \\ x_{i,j} & en\ otro\ caso \end{cases}$
DE/rand/p/exp	$u_{i,j} = \begin{cases} x_{r_3,j} + F \cdot \sum_{k=1}^{p} (x_{r_1^p,j} - x_{r_2^p,j}) & Si\ U_j(0,1) < CR\ ó\ j = j_r \\ x_{i,j} & en\ otro\ caso \end{cases}$
DE/best/p/bin	$u_{i,j} = \begin{cases} x_{r_{best},j} + F \cdot \sum_{k=1}^{p} (x_{r_1^p,j} - x_{r_2^p,j}) & Si\ U_j(0,1) < CR\ ó\ j = j_r \\ x_{i,j} & en\ otro\ caso \end{cases}$
DE/best/p/exp	$u_{i,j} = \begin{cases} x_{r_{best},j} + F \cdot \sum_{k=1}^{p} (x_{r_1^p,j} - x_{r_2^p,j}) & Si\ U_j(0,1) < CR\ ó\ j = j_r \\ x_{i,j} & en\ otro\ caso \end{cases}$
DE/current-to-rand/p	$\vec{u}_i = \vec{x}_i + K \cdot (\vec{x}_{r_3} - \vec{x}_i) + F \cdot \sum_{K=1}^{p} (\vec{x}_{r_1^p} - \vec{x}_{r_2^p})$
DE/current-to-best/p	$\vec{u}_i = \vec{x}_i + K \cdot (\vec{x}_{best} - \vec{x}_i) + F \cdot \sum_{K=1}^{p} (\vec{x}_{r_1^p} - \vec{x}_{r_2^p})$
DE/current-to-rand/p/bin	$u_{i,j} = \begin{cases} \vec{x}_{i,j} + K \cdot (\vec{x}_{r_3,j} - \vec{x}_{i,j}) + F \cdot \sum_{k=1}^{p} (\vec{x}_{r_1^p,j} - \vec{x}_{r_2^p,j}) & Si\ U_j(0,1) < CR\ ó\ j = j_r \\ x_{i,j} & en\ otro\ caso \end{cases}$
DE/rand/2/dir	$\vec{v}_i = \vec{v}_1 + \frac{F}{2}(\vec{v}_1 - \vec{v}_2 + \vec{v}_3 - \vec{v}_4)\ donde\ f(\vec{v}_1) < f(\vec{v}_2)\ y\ f(\vec{v}_3) < f(\vec{v}_4)$

Tabla 2.6 Algunas variantes del algoritmo de evolución diferencial.

Capítulo 3

Fundamentos sobre FPGA's

3.1 Conceptos

Los arreglos de compuertas programables en campo ó *Field Programmable Gate Arrays* (FPGA's) son circuitos integrados digitales que contienen bloques lógicos configurables (programables) junto con interconexiones configurables entre esos bloques. Los ingenieros de diseño pueden configurar (programar) tales dispositivos para realizar una variedad impresionante de tareas.

Dependiendo de la forma en que están implementados, algunos FPGA's solo pueden ser programados una sola vez, mientras que otros pueden ser programados una y otra vez.

La porción "Field Programmable" de los FPGA's se refiere al hecho de que su programación toma lugar en el campo "*Field*" (lo opuesto a dispositivos cuya funcionalidad interna es diseñada por el fabricante). Esto puede significar que los FPGA's son configurados en laboratorio, o puede referirse a la modificación de la función de un dispositivo residente en un sistema electrónico que ha sido ya implementado en el mundo exterior. Si un dispositivo es capaz de ser programado mientras el resto reside en un sistema de un nivel más alto, es llamado *in-system programmable* (ISP).

Existen muchos tipos diferentes de circuitos integrados digitales, incluyendo *jelly-bean logic* (pequeños componentes que contienen algunas funciones lógicas), dispositivos de memoria y microprocesadores (μP's). Algunos ejemplos son los dispositivos lógicos programables ó *Programmable Logic Devices* (PLD's), los circuitos integrados de propósito específico ó *Application-Specific Integrated Circuits* (ASIC's), los productos estándar de propósito específico ó *Application-Specific Standard Parts* (ASSP's) y desde luego los FPGA's.

Cuando el primer FPGA apareció en escena a mediados de los 80's, los FPGA's fueron ampliamente usados para implementar *glue logic* (término usado para referirse a las

relativamente pequeñas cantidades de lógica simple que son usadas para conectar dispositivos, funciones o bloques lógicos más grandes), máquinas de estado de complejidad media, y tareas de procesamiento de datos relativamente limitados. A principios de los 90's, como el tamaño y complejidad de los FPGA's empezó a incrementar, sus grandes mercados en ese entonces eran las áreas de las telecomunicaciones y los sistemas de redes, los cuales involucraban procesamiento de grandes bloques de datos y su salida al exterior. Más tarde, hacia finales de los 90's, el uso de los FPGA's en aplicaciones para el consumidor, el sector automotriz e industrial experimentaron un crecimiento enorme.

Los FPGA's son típicamente usados para modelar diseños ASIC o para proporcionar una plataforma de hardware sobre la cual verificar la implementación física de nuevos algoritmos. Sin embargo, su bajo costo de desarrollo y su corto tiempo para comercializar suponen que son cada vez más fáciles de encontrar en productos finales (algunos de los vendedores de FPGA's avanzados actualmente tienen dispositivos que compiten directamente con los ASIC's).

A principios del 2000, los FPGA's de alto desempeño contienen millones de compuertas y son accesibles. Algunas de estos dispositivos exhiben núcleos de microprocesadores embebidos, interfaces de alta velocidad de entrada/salida, y al gusto. El resultado final es que los FPGA's de hoy en día pueden ser usados para implementar cualquier cosa, incluyendo dispositivos de comunicación, radar, imagen, y otras aplicaciones de procesamiento digital de señales ó *digital signal processing* (DSP); todas en forma de componentes de un sistema sobre un chip ó *system-on-chip* (SoC) que contienen elementos de hardware y software.

Para ser un poco más específicos, los FPGA´s son actualmente empleados en cuatro áreas del mercado avanzado: ASIC y silicio personalizado, DSP, aplicaciones de microcontroladores embebidos, y chips de comunicación de capa física. Además, los FPGA´s han creado un nuevo mercado sobre sí mismos: cómputo reconfigurable ó *reconfigurable computing* (RC).

- **ASIC y silicio personalizado:** Los FPGA´s de hoy son cada vez más usados para implementar una variedad de diseños que podrían previamente haber sido realizados usando solo ASIC's y silicio personalizado.

- **Procesamiento digital de señales:** El DSP de alta velocidad ha sido tradicionalmente implementado usando específicamente microprocesadores hechos a la medida llamados procesadores digitales de señales ó *digital signal processors* (DSP's). Sin embargo, los FPGA´s de hoy contienen multiplicadores embebidos, ruteo aritmético dedicado, y una gran cantidad de RAM sobre el chip, los cuales facilitan operaciones de DSP. Cuando estas características son combinadas con paralelismo masivo proporcionado por los FPGA's, el resultado es que funciona mejor que el DSP más rápido por un factor de 500 ó más.

- **Microcontroladores embebidos:** funciones pequeñas de control han sido típicamente manejadas por procesadores embebidos de propósito especial llamados microcontroladores. Estos dispositivos de bajo costo contienen un programa sobre el chip y memoria de instrucciones, temporizadores, y periféricos de entrada/salida plegados alrededor de un núcleo de procesador. Los precios de los FPGA´s están bajando, sin embargo, y al igual que los dispositivos más pequeños ahora tienen más que suficiente capacidad para implementar un núcleo de procesador combinado con una selección de funciones personalizadas de entrada/salida. El resultado final es que los FPGA´s están siendo cada vez más atractivos para aplicaciones de control embebido.

- **Comunicaciones de capa física:** Los FPGA´s han sido ampliamente usados para implementar *glue logic* interconectando los chips de comunicación de la capa física y las capas del protocolo del nivel de red. El hecho es que los FPGA´s pueden contener múltiples transceptores de alta velocidad que implican que funciones de red y de comunicación pueden ser consolidadas en un solo dispositivo.

- **Cómputo reconfigurable:** Esto se refiere a la explotación inherente del paralelismo y la reconfigurabilidad proporcionada por los FPGA´s para hardware acelerado de algoritmos de software. Varias compañías están actualmente construyendo enormes

motores de cómputo reconfigurables basados en FPGA´s para tareas de clasificación de simulación de hardware con análisis criptográfico para descubrir nuevos métodos [22].

3.2 El origen de los FPGA´s

Para poder desarrollarse los FPGA´s, la tecnología tuvo que evolucionar desde un punto clave, el transistor, y pasar por una serie de miniaturización e incorporación de nuevas tecnologías basadas en semiconductores para llegar hasta tal punto, donde existen características del hardware que permiten la implementación de diseños más grandes y complejos con ventajas notables sobre sus predecesores. En la figura 3.1 se muestra la ubicación temporal de los FPGA´s sobre otras tecnologías.

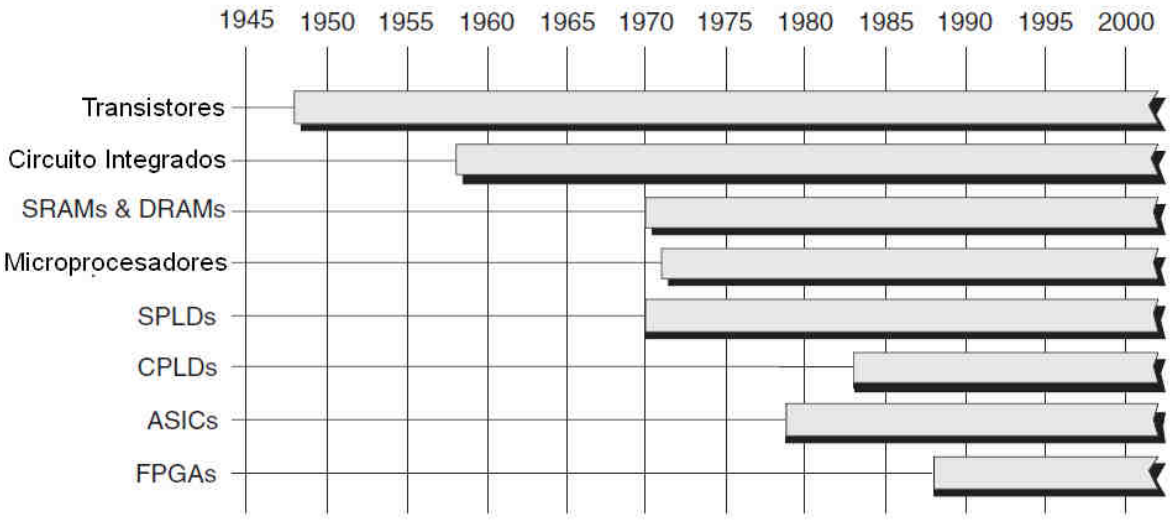


Fig. 3.1 Línea del tiempo de la tecnología.

Los primeros transistores fueron desarrollados en 1947 por W. Shockley, W. Brattain y J. Bardeen en los laboratorios Bell en Estados Unidos, los cuáles consistían de un dispositivo de punto-contacto formado de germanio. En 1950 se introdujo un componente más complejo llamado transistor de unión bipolar, el cual era más fácil, barato de construir y más confiable. A finales de los 50's, los transistores eran fabricados de silicio en su mayoría, debido a su bajo costo y docilidad. Al conectar transistores de unión bipolar de cierta forma, resultaron compuertas lógicas digitales, conocidas como lógica transistor-transistor ó *transistor-transistor*

lógic (TTL). Luego, se desarrolló la tecnología de semiconductores metal-óxido complementarios ó *Complementary Metal-Oxide Semiconductor* (CMOS) cuyo consumo de potencia era extremadamente bajo, la cual está basada en transistores de efecto de campo de metal-óxido ó *metal-oxide semiconductor field-effect transistors* (MOSFET's).

Los primeros transistores fueron suministrados como componentes discretos que fueron individualmente empaquetados en pequeños contenedores de metal. Con el tiempo, la gente comenzó a pensar que podría ser buena idea fabricar circuitos enteros en una sola pieza de semiconductor. Fue hasta 1958 cuando J. Kilby, quien trabajaba en Texas Instruments, que tuvo éxito en la fabricación de un oscilador con cambio de fase compuesto de 5 componentes de una sola pieza de semiconductor. Al mismo tiempo que Kilby trabajó en su proyecto, dos de los fundadores de Fairchild Semiconductor inventaron las técnicas litográficas ópticas que ahora son usadas para crear transistores, capas aisladoras e interconexiones sobre circuitos integrados modernos. A mediados de los 60's se introdujeron una gran variedad de circuitos integrados llamados la serie 54xx y la serie 74xx, las cuales eran para uso militar y comercial respectivamente. Luego, estas series fueron implementadas en TTL por Texas Instruments. En 1968, RCA introdujo el equivalente CMOS llamado la serie 4000.

A finales de los 60's y principios de los 70's hubo nuevos desarrollos en la tecnología de circuitos integrados. En 1970, Intel anunció la primera memoria dinámica de acceso aleatorio ó *Dynamic Random-Access Memory* (DRAM) de 1024 bits (la 1103) y Fairchild introduce la primera memoria estática de acceso aleatorio ó *Static Random-Access Memory* (SRAM) de 256 bits (la 4100). En 1971, Intel introdujo el primer *microprocesador* (μP) del mundo, el 4004, el cual fue concebido y creado por M. Hoff, S. Mazor y F. Faggin. También referido como "computadora sobre un chip", el 4004 contenía alrededor de 2300 transistores y podía ejecutar 60,000 operaciones por segundo.

Los primeros circuitos integrados programables fueron generalmente conocidos como dispositivos lógicos programables ó *programmable logic devices* (PLD's). Los componentes originales entraron en escena en 1970 en forma de Memorias Programables de solo Lectura ó *Programmable Read-Only Memories* (PROM's), fueron bastante simples, pero cada uno es digno de mención. Luego surgieron unos nuevos dispositivos llamados *Complex PLD's* (CPLD's). A menudo suele encontrarse el término *Simple PLD* (SPLD), que muchas veces es usado como

sinónimo de PLD, sin embargo, otros consideran los PLD's como un superconjunto que contiene los SPLD's y los CPLD's. En la Fig. 3.2 se muestran algunas variantes de los PLD´s.

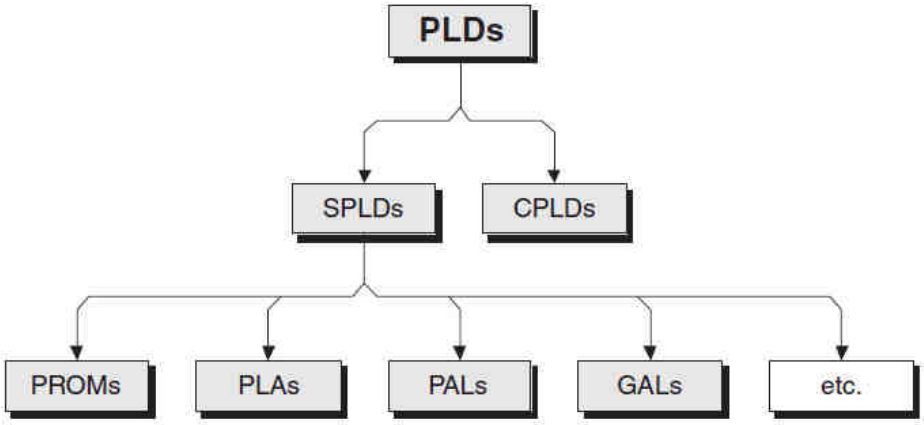


Fig. 3.2 Variantes de los PLD´s.

Existen también las versiones EPLD (programables eléctricamente y borrables mediante rayos UV), E^2PLD (eléctricamente programables y borrables byte por byte) y FLASH (eléctricamente programables y borrables por sector) de muchos de estos dispositivos.

Los primeros PLD´s simples fueron las PROM´s, las cuales aparecieron alrededor de 1970. Una forma de visualizar estos dispositivos consistía en verlos como un arreglo fijo de funciones AND controlando un arreglo programable de funciones OR, por ejemplo, la Fig. 3.3 muestra una PROM de 3-entradas y 3-salidas. Estas PROM´s fueron originalmente usadas para almacenar instrucciones de programas y datos constantes. Sin embargo, los ingenieros de diseño también las usaron para implementar funciones lógicas simples como tablas de búsqueda ó *LookUp Table* (LUT) y máquinas de estado.

Para ocuparse de las limitaciones impuestas por la arquitectura PROM, el siguiente paso en la evolución de los PLD's fueron los arreglos lógicos programables ó P*rogrammable Logic Arrays* (PLA's), los cuales llegaron alrededor de 1975. Estos dispositivos fueron mayormente usados como PLD´s configurables porque los arreglos AND y OR eran programables. La Fig. 3.4 muestra un PLA de 3-entradas y 3-salidas. A finales de los 70's y principios de los 80's empezaron a surgir PLD's más sofisticados conocidos como CPLD´s. El gran brinco ocurrió en 1984, cuando ALTERA introdujo un CPLD basado en una

combinación de tecnologías CMOS y EPROM. EL uso de CMOS permitió a Altera conseguir enormes densidades y complejidades, con un consumo relativamente bajo de energía.

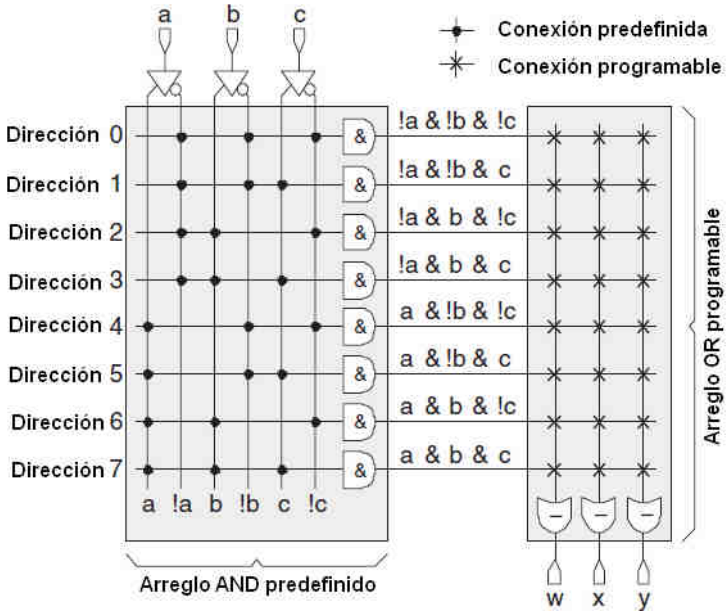


Fig. 3.3 PROM no programada

Altera hizo el brinco conceptual usando un arreglo de interconexiones centrales con menos del 100% de conectividad. Esto incrementó la complejidad de las herramientas de software de diseño, pero esto mantuvo la velocidad, potencia y costo de estos dispositivos escalables.

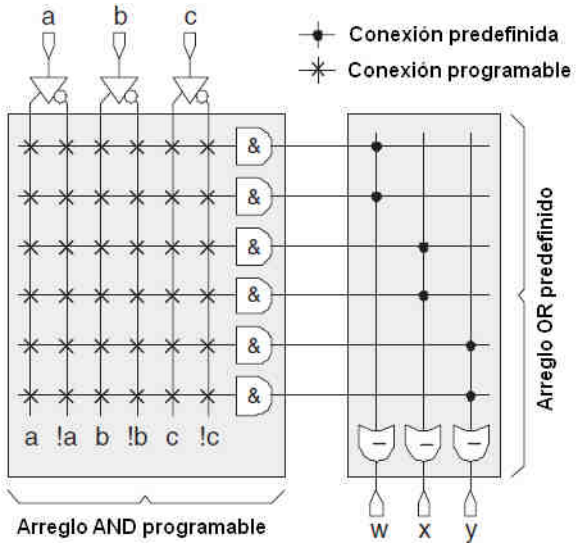


Fig. 3.4 PAL no programada.

En la Fig. 3.5 se muestra una representación de alto nivel de un CPLD. En realidad, todas estas estructuras están formadas sobre una misma pieza de silicio. Por ejemplo, la matriz de interconexión programable puede contener muchos alambres (dícese 100), pero esto es más de lo que puede ser manejado por los bloques individuales de SPLD's, los cuales son capaces de ser alojados en un número limitado (dícese 30). Así que los bloques SPLD son interconectados usando una matriz con alguna forma de multiplexor programable (Fig. 3.6).

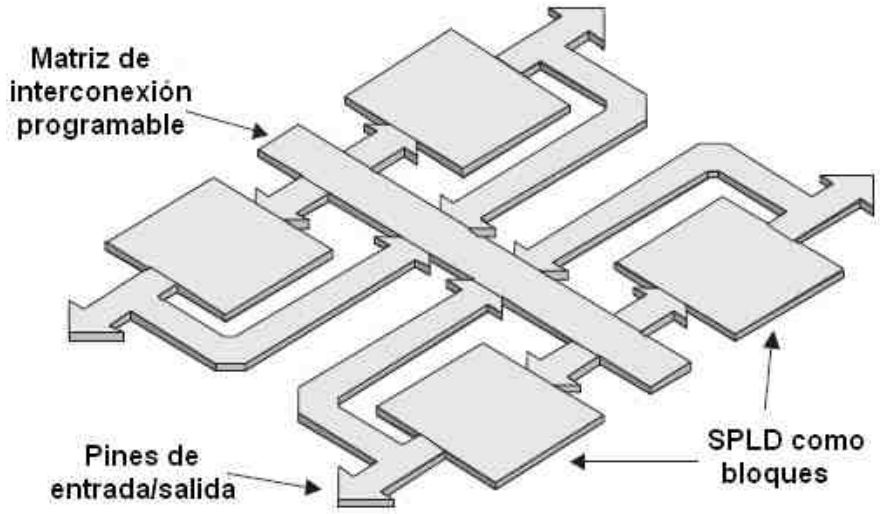


Fig. 3.5 Estructura de un CPLD genérico

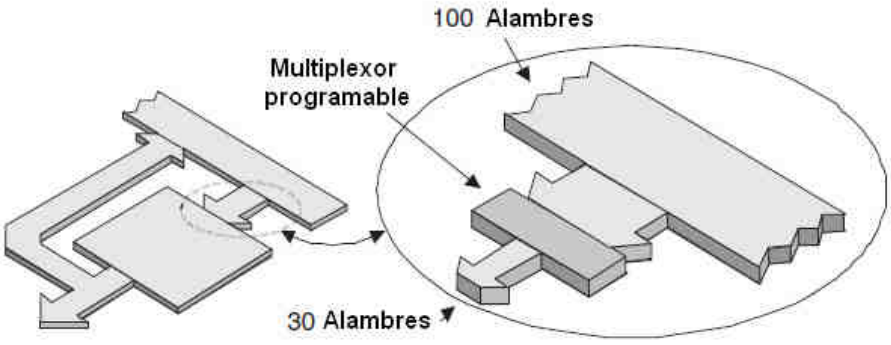


Fig. 3.6 Uso de multiplexores programables.

Dependiendo del fabricante y de la familia, los switches de los CPLD´s programables pueden ser basados en EPROM (Memoria eléctricamente programable y borrable mediante rayos UV), E^2PROM (Memoria eléctricamente programable y borrable byte por byte), FLASH (Memoria eléctricamente programable y borrable por sector) o celdas SRAM. En el caso de

dispositivos basados en SRAM, algunas variantes incrementan su versatilidad permitiendo a las celdas SRAM asociadas con cada bloque SPLD ser usadas como switches programables o como una pieza de memoria.

Conforme fue incrementando la complejidad, surgieron cuatro clases de ASIC, estos son: arreglos de compuertas, ASIC's estructurados, dispositivos de celdas estándar y chips completamente personalizados.

En el caso de dispositivos completamente personalizados, los ingenieros de diseño tienen completo control sobre la capa de la máscara usada para fabricar los chips de silicio. Los vendedores de ASIC no prefabrican cualquier componente de silicio y no proporcionan bibliotecas de funciones y compuertas lógicas predefinidas. El diseño de dispositivos completamente personalizados es altamente complejo y consume tiempo, pero los chips resultantes contienen la máxima cantidad de lógica con un gasto mínimo de silicio.

El concepto arreglo de compuertas fué originado en compañías como IBM y Fujitsu a finales de los 60's. Sin embargo, estos dispositivos inicialmente estuvieron disponibles solo para consumo interno y no fué sino hasta mediados de los 70's que el acceso a arreglos de compuertas basados en CMOS estuvieron disponibles para cualquiera que pudiera pagar por ellos. Los arreglos de compuertas están basados en la idea de una celda básica consistente de una colección de transistores y resistores no conectados. Cada vendedor de ASIC determina la mezcla óptima de componentes proporcionados en la celda básica (Fig. 3.7). Los vendedores de ASIC comienzan prefabricando chips de silicio conteniendo arreglos de estas celdas básicas. En el caso de arreglos de compuertas acanalados, las celdas básicas son generalmente presentadas como arreglos de una sola columna o doble columna; las áreas libres entre los arreglos son conocidas como canales (Fig. 3.8).

Los arreglos de compuertas ofrecen ventajas considerables en el costo de prefabricación de los transistores y otros componentes, así que sólo las capas de metal necesitan ser personalizadas. La desventaja es que muchos diseños dejan cantidades significativas de recursos internos inutilizables, la ubicación de las compuertas es restringida y el ruteo de las pistas internas es menos óptimo. Todos estos factores impactan negativamente en el desempeño y consumo de potencia del diseño.

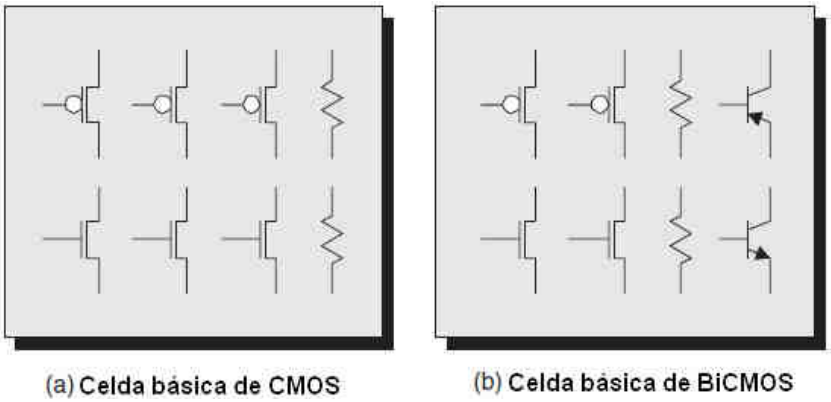


Fig. 3.7 Ejemplos de celdas básicas en un arreglo de compuertas.

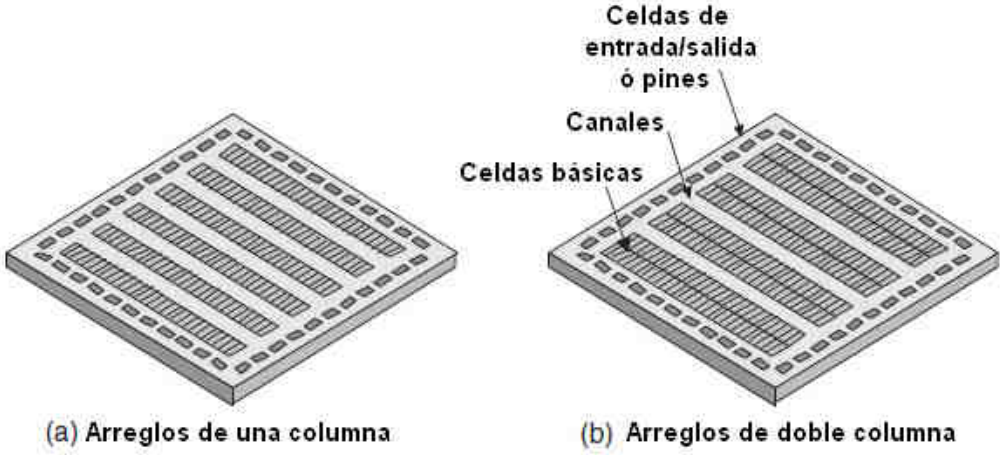


Fig. 3.8 Arquitecturas de arreglos de compuertas acanalados.

Para resolver los problemas asociados con los arreglos de compuertas, los dispositivos de celdas estándar llegaron ha estar disponibles a principios de los 80's. Estos componentes tienen muchas similitudes con los arreglos de compuertas. El concepto de celda estándar permite a cada función lógica ser creada utilizando el mínimo número de transistores con componentes no redundantes, y las funciones pueden ser posicionadas para facilitar las conexiones entre ellas. Los dispositivos de celdas estándar, sin embargo, proporcionan una utilización más cerca del óptimo de silicio que los arreglos de compuertas.

Los ASIC's estructurados aparecieron a inicios de los 90's. La idea es que estos dispositivos pueden ser personalizados solo en las capas de metal (justo como un arreglo de

compuertas). La diferencia es que, debido a la gran complejidad de los bloques de ASIC estructurado, muchas de las capas de metal están predefinidas.

A inicios de los 80's, al parecer hubo un vacío en la continuidad de los circuitos integrados. A finales de esta época, hubieron dispositivos programables como SPLD's y CPLD's, los cuales eran altamente configurables y de rápido diseño, pero no soportaban funciones grandes o complejas. También hubo ASIC´s, pero estos no podían soportar funciones extremadamente grandes y complejas, pero fueron dolorosamente caros para diseñar. Luego, aparecieron los primeros FPGA's desarrollados por Xilinx y estuvieron disponibles en el mercado alrededor de 1984. Los primeros dispositivos estuvieron basados en el concepto de bloques lógicos programables, los cuales consistían de una tabla de búsqueda (LUT), un registro que podría actuar como un flip-flop o latch, y un multiplexor. La Fig. 3.9 muestra un bloque lógico programable simple (los bloques lógicos en los FPGA's modernos pueden ser significativamente más complejos). Cada FPGA contiene un número grande de estos bloques lógicos programables.

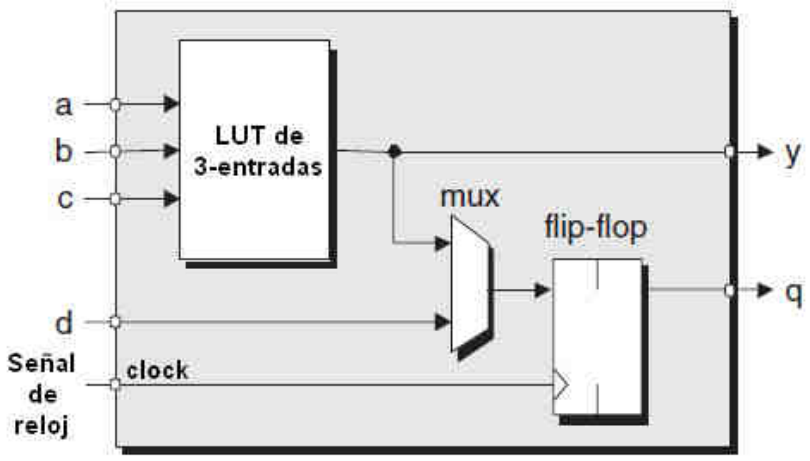


Fig. 3.9 Elementos clave formando un bloque lógico programable simple.

Un FPGA completo contiene un gran número de bloques lógicos programables "islas" cercadas por un "mar" de interconexiones programables (Fig. 3.10). Como es usual, esta ilustración de alto nivel es solo una representación abstracta. En realidad, todo está implementado con transistores e interconexiones sobre una misma pieza de silicio usando técnicas de creación estándar de circuito integrados.

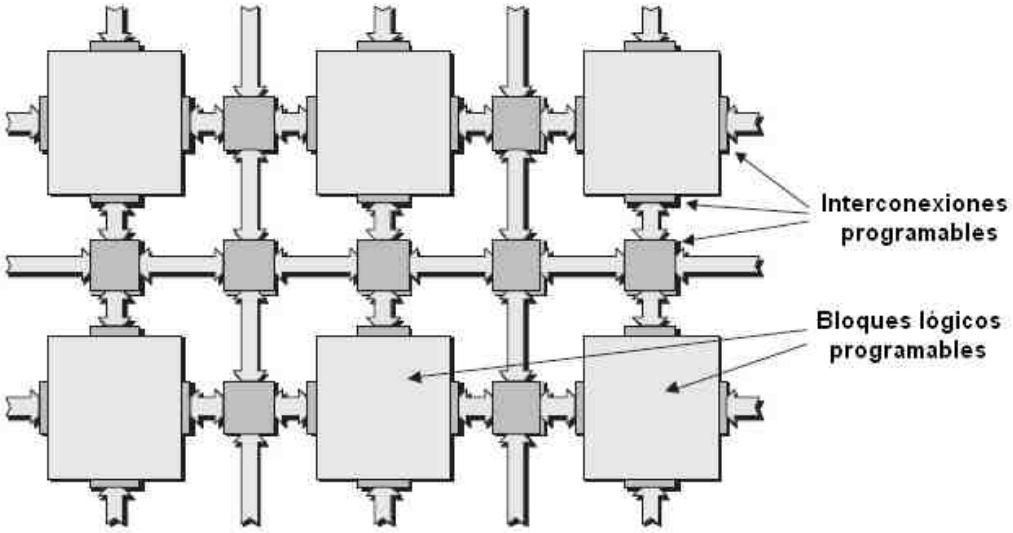


Fig. 3.10 Vista simple de una arquitectura genérica de un FPGA.

Finalmente los FPGA´s resultaron ser el puente del vacío provocado entre los PLD's y los ASIC's. De una u otra manera, ellos fueron altamente configurables y gozaron de rapidez en el diseño y tiempo de modificación asociados con PLD's. Igualmente, podrían ser usados para implementar funciones enormes y complejas que estuvieron previamente en el dominio de los ASIC's [22].

3.3 Arquitectura de los FPGA´s

La mayoría de los FPGA's están basados en el uso de celdas de configuración SRAM, lo que significa que pueden ser configurados una y otra vez. Las principales ventajas de esta técnica son que los diseños pueden ser rápidamente implementados y probados, mientras evolucionan los estándares y protocolos pueden ser reubicados de una manera relativamente fácil. Además, cuando el sistema es activado por primera vez, el FPGA puede inicialmente ser programado para desempeñar una función tal como auto-prueba o prueba tarjeta/sistema, y puede posteriormente ser reprogramado para desempeñar su tarea principal. Otra gran ventaja de las estrategias basadas en SRAM es que estos dispositivos están a la vanguardia en tecnología. Los vendedores de FPGA's pueden pensar en el hecho de que muchas otras compañías especializadas en dispositivos de memoria gastan enormes recursos en investigación

y desarrollo de esta área. Además, las celdas SRAM son creadas usando exactamente la misma tecnología CMOS como el resto del dispositivo.

En el pasado, los dispositivos de memoria fueron usados para calificar los procesos de fabricación asociado con un nuevo nodo de tecnología. Recientemente, la mezcla de tamaño, complejidad y regularidad asociada con las últimas generaciones de FPGA's han resultado en que estos dispositivos sean usados para esta tarea. Una ventaja del uso de los FPGA´s sobre dispositivos de memoria para calificar el proceso de fabricación es que, si hay un defecto, la estructura del FPGA es tal que es más fácil de identificar y localizar el problema.

Desafortunadamente, un aspecto negativo de los dispositivos basados en SRAM es que tienen que ser reconfigurados cada vez que el sistema sea activado. Esto requiere el uso de un dispositivo especial de memoria externo (el cual tiene un costo asociado y consumo sobre la tarjeta) o de un microprocesador sobre la tarjeta.

Otra desventaja de los dispositivos basados en SRAM es que puede ser difícil proteger tu propiedad intelectual en la forma de tu diseño. Esto es porque el archivo de configuración usado para programar el dispositivo es almacenado en un dispositivo de memoria externo. Existe una buena noticia a pesar de esto, algunos FPGA's basados en SRAM soportan el concepto de *encriptamiento de trama de bits*. En este caso, el dato de configuración final es encriptado antes de que sea almacenado en el dispositivo de memoria. La clave de encriptamiento es cargada en un registro especial del FPGA. En conjunción con alguna lógica asociada, esta clave permite la configuración encriptada de la trama de bits para ser desencriptada cuando esté siendo cargada en el dispositivo.

A diferencia de los dispositivos basados en SRAM, los cuales son programados mientras residen en el sistema, los dispositivos basados en *antifuse* son programados fuera de línea usando un dispositivo programador especial.

Los promotores de los FPGA's basados en *antifuse* están orgullosos de apuntar a una variedad (no significativa) de ventajas. Primero que todo, estos dispositivos son no volátiles (sus datos de configuración permanecen mientras el sistema está inactivo), lo que significa que hay disposición inmediata cuando el sistema es iniciado (encendido). Su no volatilidad implica

que no requieren de memoria externa para almacenar sus datos de configuración, lo cual ahorra costos en componentes adicionales y conserva el estado de la tarjeta.

Una notable ventaja de los FPGA's basados en *antifuse* es el hecho de que su estructura interconectada es naturalmente "tolerante a la radiación", lo que significa que son relativamente inmunes a los efectos de la radiación. Una vez que un *antifuse* ha sido programado, este no puede ser alterado.

Los FPGA's basados en E^2PROM o FLASH son similares a sus contrapartes basados en SRAM en que sus celdas de configuración está conectadas juntas en una larga cadena de registros de desplazamiento. Estos dispositivos pueden configurados fuera de línea usando un dispositivo programador. Algunas versiones son programables en sistema, pero su tiempo de programación es aproximadamente tres veces el de un componente basado en SRAM.

Una vez programado, los datos que contienen son no volátiles, por tanto estos dispositivos funcionaran tan luego sean activados. Con respecto a la protección, algunos usan el concepto de clave multi-bit, la cual puede tener desde 50 bits hasta algunos cientos de bits en tamaño. Una vez que se ha programado el dispositivo, se puede cargar la clave definida por el usuario para proteger los datos de configuración. Después de que la clave ha sido cargada, la única forma de leer los datos del dispositivo, o escribir nuevos datos en él, es cargar una copia de la clave vía JTAG (puerto de comunicación del FPGA). El hecho de que el puerto JTAG en los dispositivos de hoy corra a aproximadamente 20MHz implica que tomaría billones de años encontrar la clave probando con todos los valores posibles.

Actualmente, algunos fabricantes ofrecen combinaciones extrañas de tecnología de programación. Por ejemplo, considere un dispositivo donde cada elemento de configuración está formado por la combinación de una celda FLASH (o E^2PROM) y una celda SRAM. En este caso, los elementos tipo FLASH pueden ser reprogramados. Luego, cuando el sistema es activado, el contenido de las celdas FLASH es copiado en paralelo masivamente a sus correspondientes celdas tipo SRAM. Esta técnica proporciona la no volatilidad asociada con los dispositivos *antifuse*, lo que significa que este dispositivo está inmediatamente disponible cuando el sistema es activado. Además, se puede reconfigurar el dispositivo usando sus celdas

FLASH mientras permanezca en el sistema o fuera de línea mediante un dispositivo programador.

En la Tabla 3.1 se muestra un resumen de las características principales de las tecnologías de programación descritas anteriormente.

Características	***SRAM***	***Antifuse***	***E²PROM/FLASH***
Reprogramable	*Si (en sistema)*	*No*	*Si (en sistema o fuera de línea)*
Velocidad de reprogramación (incluyendo borrado)	*Rápido*	---	*3 veces más lento que las SRAM*
Volatilidad (que deba ser programado mientras el sistema esté activo)	*Si*	*No*	*No (pero puede ser si se así desea)*
Requiere archivo de configuración externo	*Si*	*No*	*No*
Bueno para prototipos	*Si (muy bueno)*	*No*	*Si (razonablemente)*
Disposición inmediata después del encendido	*No*	*Si*	*Si*
Seguridad (protección intelectual)	*Aceptable (especialmente cuando se usa encriptamiento)*	*Muy buena*	*Muy buena*
Tamaño de la celda de configuración	*Grande (seis transistores)*	*Muy pequeña*	*Medianamente pequeña (2 transistores)*
Consumo de energía	*Medio*	*Bajo*	*Medio*
Resistente a la radiación	*No*	*Si*	*No relamente*

Tabla 3.1 Resumen de las tecnologías de programación

Es común clasificar a los FPGA's como de *grano fino* o *grano grueso*. En el caso de la arquitectura de grano fino, cada bloque lógico puede ser usado para implementar solo una función muy simple. Por ejemplo, podría ser posible configurar el bloque para actuar como cualquier función de 3-entradas, tal como una compuerta lógica (AND, OR, NAND, etc.) o un elemento de almacenamiento (flip flop tipo-D, latch tipo-D, etc.).

Por otro lado, para implementar *glue logic* y estructuras irregulares como máquinas de estados, las arquitecturas de grano fino son particularmente eficientes cuando ejecutan algoritmos sistólicos (funciones que se benefician de las implementaciones masivamente paralelas). Se dice que estas arquitecturas ofrecen algunas ventajas con interés particular en tecnología de síntesis lógica, la cual es convertida hacia arquitecturas ASIC de grano fino.

En el caso de arquitecturas de grano grueso, cada bloque lógico contiene una cantidad relativamente grande de lógica comparado a sus contrapartes de grano fino. Por ejemplo, un bloque lógico podría contener cuatro LUT's de 4-entradas, cuatro multiplexores, cuatro flip flops tipo-D, y alguna lógica rápida de acarreo.

Una consideración importante con respecto a la granularidad de la arquitectura es que las implementaciones de grano fino requieren un número relativamente grande de conexiones dentro y fuera de cada bloque comparado con la cantidad de funcionalidad que pueden ser soportados por aquellos bloques. Como la granularidad de los bloques incrementa con grano medio ó más alto, la cantidad de conexiones dentro de los bloques decrementa comparado con la cantidad de funcionalidad que pueden soportar. Esto es importante porque la interconexión de interbloques programables cuenta para la gran mayoría de retardos asociados con señales a medida que se propagan a través de un FPGA.

Hay dos encarnaciones fundamentales de bloques lógicos programables usados para formar arquitecturas de grano medio: las basadas es MUX (multiplexores) y las basadas en LUT (*lookup table*).

Basadas en MUX: Como ejemplo de aquellas basadas en MUX, considere una forma en la que la función de 3-entradas $y=(a\&b)|b$ podría ser implementada usando un bloque que contiene solo multiplexores (Fig. 3.11). El dispositivo puede ser programado tal que cada entrada al bloque es representada con un '0' lógico ó un '1' lógico, o verdadero y la versión inversa de una señal (*a*, *b*, ó *c* en este caso) viniendo desde otro bloque o desde una entrada primaria al dispositivo. Esto permite a cada bloque ser configurado en miles de formas para implementar una gran cantidad de funciones posibles (En la Fig. 3.11 la 'x' señala una entrada irrelevante, es decir, puede ser cero ó uno, no importa).

Basadas en LUT: El concepto fundamental de las LUT's es relativamente simple. Un grupo de señales de entrada es usado como un índice (apuntador) a una tabla de búsqueda. El contenido de esta tabla es organizada tal que la celda apuntada por cada combinación a la entrada contiene el valor deseado. Suponiendo que la LUT está formada de celdas SRAM (podrían ser celdas *antifuse*, E^2PROM ó FLASH), una técnica común es usar las entradas para seleccionar la celda SRAM usando una cascada de compuertas de transmisión como se muestra

en la Fig. 3.12. Si una compuerta de transmisión está habilitada (activa), esta pasa la señal recibida a la salida. Si la compuerta está deshabilitada, la señal de salida es eléctricamente desconectada del alambre. El símbolo de la compuerta de transmisión con un pequeño círculo indica que esta compuerta estará activa con un cero lógico sobre la entrada de control.

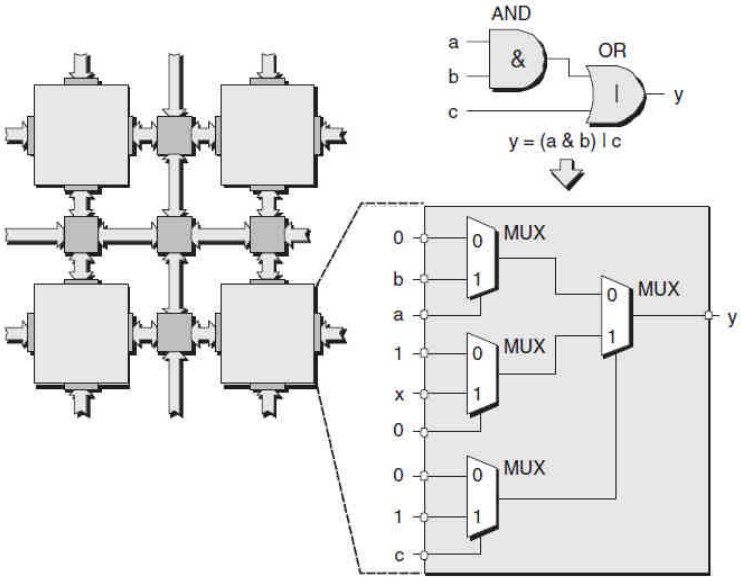


Fig. 3.11 Bloque lógico basado en MUX (multiplexores).

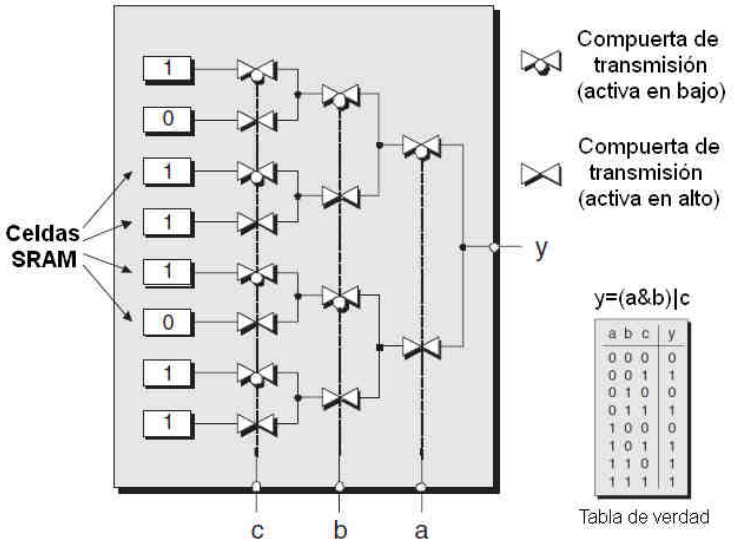


Fig. 3.12 LUT basada en compuertas de transmisión.

Un bloque lógico puede tener además de una o más LUT's, otros elementos tales como multiplexores y registros.

Cada vendedor de FPGA's nombra las cosas a su manera, por ello, el bloque de construcción del núcleo en un FPGA moderno de Xilinx es llamado celda lógica ó *logic cell* (LC). Entre otras cosas, una LC comprende una LUT de 4-entradas (que puede actuar como una RAM de 16*1 ó un registro de desplazamiento de 16-bits), un multiplexor y un registro (Fig. 3.13).

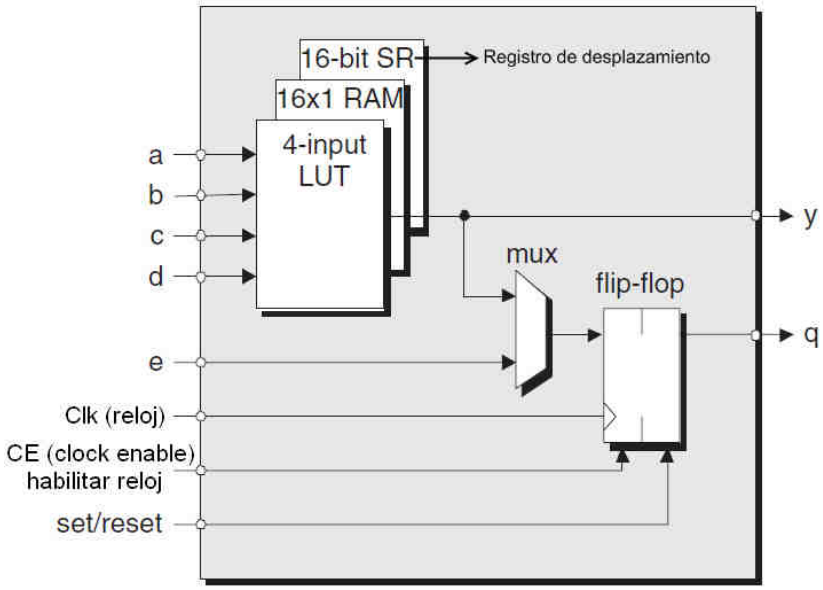


Fig. 3.13 Celda lógica simplificada de Xilinx.

El registro puede ser configurado para actuar como un flip flop ó como un latch. La polaridad del reloj (flanco de subida o flanco de bajada) puede ser configurada, así como la polaridad de las señales de habilitación del reloj y set/reset (activas en alto ó activas en bajo).

El equivalente del bloque de construcción del núcleo en un FPGA de Altera es llamado elemento lógico ó *logic element* (LE). Hay diferencias entre un LC de Xilinx y un LE de Altera, pero los conceptos son similares.

En un nivel de jerarquía más elevado encontramos los *slices* y los bloques lógicos configurables ó *configurable logic block* (CLB) de Xilinx. Los slices pueden contener dos ó más LC's y los CLB's pueden contener cuatro ó más slices. En el caso de Altera encontramos bloques de arreglos lógicos ó *Logic Array Blocks (LAB's)* que consisten de varios LE's.

Cada fabricante de FPGA´s incorpora elementos interesantes en sus arquitecturas, solo mencionaremos brevemente algunos.

RAM embebida: Muchas aplicaciones requieren el uso de memoria, por tanto los FPGA´s incluyen secciones relativamente grandes de memoria RAM embebida conocidas como *e-RAM* ó bloque RAM. Dependiendo de la arquitectura del componente, estos bloques pueden encontrarse alrededor del dispositivo, dispersos sobre el chip u organizados en columnas.

Multiplicadores embebidos, sumadores, MAC's: Algunas funciones como los multiplicadores son lentos si son implementados conectando un gran número de bloques lógicos programables. Como este tipo de funciones son requeridas para muchas aplicaciones, muchos FPGA's incorporan bloques multiplicadores embebidos. Estos son usualmente alojados en las cercanías de los bloques de memoria RAM embebidos. Una operación muy común en aplicaciones de procesamiento digital de señales ó *Digital Signal Proccessing* (DSP) es la llamada multiplica-y-acumula ó *multiply-and-accumulate* (MAC), esta operación multiplica dos números y suma el resultado a un total almacenado en un acumulador.

Pines de entrada/salida: Los FPGA´s de hoy pueden tener 1000 ó más pines, los cuales están organizados en un arreglo a través de la base del encapsulado. Cuando el chip de silicio se encuentra dentro del encapsulado, las estrategias de encapsulado flip-chip permiten a los pines de alimentación, tierra, reloj y de entrada/salida ser presentados a través de la superficie del chip.

Para finalizar, es importante mencionar que existen otros componentes embebidos además de los mencionados anteriormente, por ejemplo, núcleos de microprocesadores [22, 23, 24].

3.4 Lenguaje VHDL

Electronic Desgin Automation (EDA) es el nombre que se le da a todas las herramientas (tanto hardware como software) que sirven de ayuda en el diseño de sistemas electrónicos. Dentro del EDA existe el diseño asistido por computadora ó *Computer Aided Design* (CAD), el cual consiste en un proceso de diseño que emplea sofisticadas técnicas gráficas de ordenador,

apoyadas en paquetes de software para ayuda en los problemas analíticos, de desarrollo, de coste y ergonómicos asociados con el trabajo de diseño.

En el diseño de hardware se puede introducir la fase de simulación y comprobación de circuitos utilizando las herramientas CAD, de manera que no es necesario realizar físicamente el circuito para comprobar el funcionamiento del mismo, economizando así el ciclo de diseño. El ciclo de diseño de hardware se muestra en la Fig. 3.14.

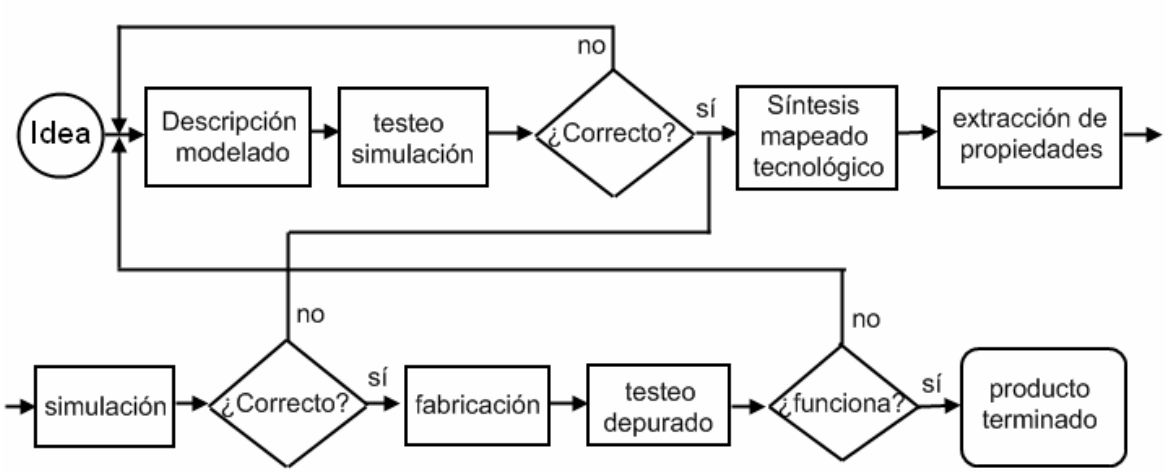


Fig. 3.14 Flujo de diseño para sistemas electrónicos y digitales.

Entre las herramientas CAD para el diseño de hardware tenemos: lenguajes de descripción de circuitos (lenguajes que permiten describir un circuito digital ya sea estructural o comportamentalmente), captura de esquemas (descripción mediante diagramas de sus componentes y sus interconexiones), grafos y diagramas de flujo (descripción gráfica comportamental mediante diagramas de flujo, redes de petri, máquinas de estado, etc.), simulación de sistemas, simulación digital (simulación que toma en cuenta los retrasos de las señales), simulación eléctrica (simulación de bajo nivel, por ejemplo, a nivel de transistores), realización de PCB's (permiten el trazado de pistas para la implementación en placas de circuito impreso), realización de circuitos integrados y realización de dispositivos programables (para programar desde PAL's hasta FPGA's).

El significado de las siglas VHDL es *VHSIC (Very High Speed Integrated Circuit) Hardware Description Lenguaje*, es decir, lenguaje de descripción de hardware de circuitos integrados de muy alta velocidad. VHDL es un lenguaje de descripción y modelado diseñado para describir,

en una forma en que los humanos y las máquinas puedan leer y entender la funcionalidad y organización de sistemas de hardware digitales, placas de circuitos y componentes.

VHDL fue desarrollado como un lenguaje para el modelado y simulación lógica dirigida por eventos de sistemas digitales, y actualmente se utiliza también para la síntesis automática de circuitos.

VHDL es un lenguaje con una sintaxis amplia y flexible que permite el modelado estructural, en flujo de datos y de comportamiento de hardware. VHDL permite el modelado preciso, en distintos estilos, del comportamiento de un sistema digital conocido y el desarrollo de modelos de simulación.

Otros de los usos de este lenguaje es la síntesis automática de circuitos. En el proceso de síntesis se parte de una especificación de entrada con un determinado nivel de abstracción y se llega a una implementación más detallada, menos abstracta. Por tanto, la síntesis es una tarea vertical entre niveles de abstracción, del nivel más alto en la jerarquía de diseño hacia el más bajo nivel de la jerarquía.

La síntesis a partir de VHDL constituye hoy en día una de las principales aplicaciones del lenguaje con una gran demanda. Las herramientas de síntesis basadas en el lenguaje permiten en la actualidad ganancias importantes en la productividad del diseño.

Algunas ventajas del uso de VHDL para la descripción de hardware son:

- VHDL permite diseñar, modelar y comprobar un sistema desde un alto nivel de abstracción bajando hasta el nivel de definición estructural de compuertas.
- Circuitos descritos utilizando VHDL, siguiendo unas guías para síntesis, pueden ser utilizados por diversas herramientas de síntesis para crear e implementar circuitos.
- Los módulos creados en VHDL pueden utilizarse en diferentes diseños, lo que permite la reutilización del código. Además la misma descripción puede emplearse para diferentes tecnologías sin tener que rediseñar todo el circuito.
- Al estar basado en un estándar (IEEE Std 1076-1987, IEEE Std 1076-1993) los ingenieros de toda la industria de diseño pueden usar este lenguaje para minimizar errores de comunicación y problemas de compatibilidad.

- VHDL permite diseño *Top-Down*, esto es, describir (modelar) el comportamiento de los bloques de alto nivel, analizarlo (simularlos) y refinar la funcionalidad en alto nivel requerida antes de llegar a niveles más bajos de abstracción de la implementación del diseño.
- Modularidad: VHDL permite dividir o descomponer un diseño de hardware y su descripción VHDL en unidades más pequeñas.

VHDL es interesante porque permite dos tipos de descripciones:

Estructura: VHDL puede ser usado como un lenguaje *Netlist* normal y corriente donde se especifican por un lado los componentes del sistema y por otro sus interconexiones.

Comportamiento: VHDL también se puede utilizar para la descripción funcional o comportamental de un circuito. Ésta es la diferencia con un lenguaje *Netlist*. Sin necesidad de conocer la estructura interna de un circuito es posible describirlo explicando su funcionalidad. Esto es especialmente útil en simulación, ya que permite simular un sistema sin conocer su estructura interna. Muchas veces la descripción comportamental se divide en dos, dependiendo del nivel de abstracción y del modo en que se ejecutan las instrucciones. Estas dos formas comportamentales de describir circuitos son la de flujo de datos y la algorítmica.

La sintaxis de VHDL no es sensible a mayúsculas o minúsculas, por lo que se puede escribir como se prefiera.

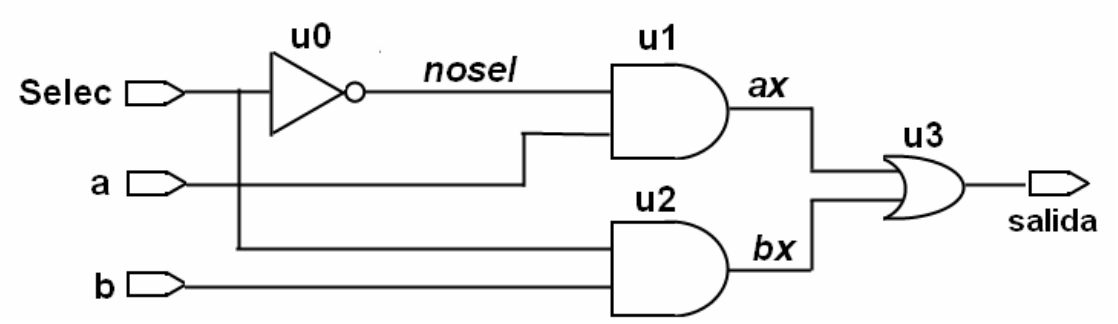


Fig. 3.15 Esquema de un ejemplo básico en VHDL.

Sin importar el tipo de descripción a utilizar, hay que definir el símbolo o entidad del circuito. En efecto, lo primero es precisar las entradas y salidas del circuito, es decir, la caja negra que lo define. Se llama entidad porque en la sintaxis de VHDL esta parte se declara con

la palabra reservada **ENTITY**. En el caso de la Fig. 3.15, se muestra un circuito que multiplexa dos líneas de entrada (*a* y *b*), a una sola línea (*salida*) también de un bit; la señal *selec* sirve para seleccionar la línea *a* (*selec='0'*) ó *b* (*selec='1'*), por lo que la declaración de la entidad para este circuito puede hacerse de la siguiente manera:

```
--Los comentarios empiezan con dos guiones
ENTITY mux IS
PORT (      a:   IN  bit;
            b:   IN  bit;
        selec:   IN  bit;
       salida:   OUT bit);
END mux;
```

Esta porción del lenguaje indica que la entidad *mux* (que es el nombre que se le ha dado al circuito) tiene tres entradas de tipo *bit* y una salida también del tipo *bit*. El tipo *bit* simplemente indica una línea que puede tomar los valores '0' ó '1'.

La entidad de un circuito es única, sin embargo, una sola entidad puede tener varias vistas, que en el caso de VHDL se llaman **arquitecturas**. Cada bloque de arquitectura, que es donde se describe el circuito, puede ser una representación diferente del mismo circuito. Por ejemplo, puede haber una descripción estructural y otra comportamental, las dos son descripciones diferentes, pero ambas corresponden al mismo circuito, símbolo o entidad. Se muestra a continuación la descripción comportamental del multiplexor:

```
ARCHITECTURE comportamental OF mux IS
BEGIN
    PROCESS(a,b,selec)
    BEGIN
        IF(selec='0') THEN
           salida<=a;
        ELSE
           salida<=b;
        END IF;
    END PROCESS;
END comportamental;
```

Un bloque *process* es una especie de subrutina cuyas instrucciones se ejecutan secuencialmente cada vez que alguna de las señales de la *lista sensible* cambia. Esta lista sensible es una lista de señales que se suele poner junto a la palabra reservada **PROCESS**, y en el caso del ejemplo es *(a,b,selec)*.

Esta descripción comportamental es muy sencilla de entender, ya que sigue una estructura parecida a los lenguajes de programación convencionales. Es por lo que se dice que

se trata de una descripción comportamental algorítmica. Lo que se está indicando es simplemente que si la señal *selec* es cero, entonces la salida es la entrada *a*, y si *selec* es uno, la salida es la entrada *b*.

Esta forma tan sencilla de describir el circuito permite a ciertas herramientas sintetizar el diseño a partir de una descripción comportamental como la que se acaba de mostrar. La diferencia con un *Netlist* es directa: en una descripción comportamental no se está indicando ni los componentes ni sus interconexiones, sino simplemente lo que hace, es decir, su comportamiento o funcionamiento.

VHDL posee una forma de describir circuitos que además permiten la paralelización de instrucciones, y que se encuentra más cercana a una descripción estructural del mismo, siendo todavía una descripción funcional. A continuación se muestran dos ejemplos de una descripción concurrente, también llamada de *flujo de datos* o de *transferencia entre registros*.

```
ARCHITECTURE flujo1 OF mux IS
SIGNAL nosel,ax,bx: bit;
BEGIN
     nosel  <=    NOT selec;
        ax  <=  a AND nosel;
        bx  <=  b AND selec;
    salida  <= ax OR  bx;
END flujo1;
```

```
ARCHITECTURE flujo2 OF mux IS
BEGIN
    salida<=a WHEN selec='0' ELSE
             b;
END flujo2;
```

En la descripción de la izquierda hay varias instrucciones todas ellas concurrentes, es decir, se ejecutan cada vez que cambia alguna de las señales que intervienen en la asignación. Este primer caso es casi una descripción estructural, ya que de alguna manera se están describiendo las señales (cables) y los componentes que la definen; aunque no es estructural, ya que en realidad se trata de asignaciones a señales y no una lista de componentes y conexiones. En el segundo caso (derecha) es también una descripción de flujo de datos, aunque basta una única instrucción de asignación para definir el circuito.

VHDL también puede ser usado como *Netlist* o lenguaje de descripción de estructura. Esta descripción estructural sirve también para la realización de diseños jerárquicos. La descripción estructural también se incluye dentro de un bloque de arquitectura, si bien la sintaxis interna es completamente diferente:

```
ARCHITECTURE estructura OF mux IS
SIGNAL ax,bx,nosel:bit
BEGIN
    u0: ENTITY   inv  PORT MAP (e=>selec, y=>nosel);
    u1: ENTITY  and2  PORT MAP (e1=>a,e2=>nosel,y=>ax);
    u2: ENTITY  and2  PORT MAP (b,selec,bx);
    u3: ENTITY   or2  PORT MAP (e1=>ax,e2=>bx,y=>salida);
END estructura;
```

Se puede observar que esta descripción es un poco más larga y mucho menos clara que las anteriores. En el cuerpo de la arquitectura se hace lo que en un *Netlist* normal, es decir, se ponen los componentes y sus interconexiones. Para los componentes se usarán entidades que estarán definidas en alguna biblioteca, y para las conexiones se usarán señales que se declararán al principio de la arquitectura.

Las señales y conexiones deben tener un nombre. En el esquema se han colocado nombres a las líneas de conexión internas del circuito. Estas líneas se deben declarar como **SIGNAL** en el cuerpo de la arquitectura y delante del **BEGIN**. Una vez declaradas las señales que intervienen se procede a conectar entre sí las señales y entidades que representan los componentes, para ellos la sintaxis es muy simple. Lo primero es identificar y poner cada componente, que es lo que comúnmente se conoce como replicación, es decir, asignarle a cada componente concreto un símbolo. El nombre puede ser cualquier identificador válido y la única condición es que no se repita (u1, u2, u3, son ejemplos de nombres). Luego de la palabra **ENTITY** se pone el nombre del componente y se realizan las conexiones poniendo cada señal en su correspondiente lugar con las palabras **PORT MAP**. Así los dos primeros argumentos en el caso de la compuerta *and2* son las entradas, y el último es la salida. De esta forma se va creando el *Netlist* o definición de la estructura.

Aunque los dos primeros ejemplos se ajustan al VHDL'87 y al VHDL'93, el último solo es válido para el VHDL'93. En la primera versión de VHDL no se permite la referencia directa a la entidad, por lo que se hace necesaria la definición de un componente y luego el enlace, en un bloque de configuración, del componente con la entidad que se quiera.

Como se mencionó anteriormente, existen dos versiones de VHDL. La primer versión apareció en 1987 y de ahí su nombre VHDL'87. Esta primera especificación fue estandarizada por la IEEE y su número de registro fue el 1076, así que la referencia correcta de esta versión es IEEE Std 1076-1987. En esta primera versión pronto se dieron algunas carencias,

especialmente en lo relacionado con la síntesis de circuitos. Durante varios años se recogieron las experiencias obtenidas del uso del primer estándar y se planteó un segundo que sigue vigente hoy en día. Fué en 1993 y se trata del IEEE Std-1076-1993, que es conocido comúnmente como VHDL'93 [25].

No profundizaremos más en lo que respecta al lenguaje VHDL, ya que no es el propósito de este trabajo, sin embargo mencionaremos que dentro de todo el conjunto de instrucciones que existen en VHDL, solo un subconjunto es sintetizable, es decir, que se pueden implementar en hardware, por ello, en nuestro diseño nos limitaremos a trabajar en este subconjunto debido a que la meta es poder implementar los diseños en un FPGA. Para finalizar, es posible utilizar los tipos de datos reales en VHDL, sin embargo, no pertenecen al subconjunto sintetizable, por lo que se debe crear este tipo de dato y diseñar los bloques que realicen operaciones para este nuevo tipo de dato; con lo que respecta al tratamiento de números enteros, estos si son sintetizables al igual que las operaciones entre ellos, por lo que, si es posible utilizarlos se recomienda trabajar con ellos (por facilidad).

Capítulo 4

Estado del arte

4.1 Algoritmos bio-inspirados en hardware

La mayoría de los algoritmos bio-inspirados se caracterizan por la gran cantidad de recursos consumidos y el tiempo de cómputo a la hora de ejecutarlos. Esto es debido a que tienen que procesar cierta cantidad de datos con el fin de encontrar una buena solución al problema. Como vimos en capítulos anteriores, los algoritmos bio-inspirados tienen diferentes usos, y en general, aquellos utilizados para resolver problemas de optimización global, ya sea numérica o combinatoria, con o sin restricciones y mono-objetivo o multi-objetivo, necesitan realizar una gran cantidad de cálculos para encontrar una buena solución (en el caso de multi-objetivo, un conjunto de buenas soluciones), que se ve traducido en un mayor consumo de tiempo en los ordenadores.

Han surgido versiones "micro" de algunos de los algoritmos bio-inspirados, cuyo objetivo es minimizar la cantidad de cálculos que realizan, con el fin de acelerar su procesamiento y obtener resultados más rápidamente. Sin embargo, con el surgimiento de tecnologías como los FPGA´s, algunos investigadores se han aventurado a desarrollar arquitecturas de estos algoritmos, con el fin de tener un dispositivo de propósito específico con una versión acelerada de dichos algoritmos. El desarrollo de tales arquitecturas ha sido ventajoso, ya que tener versiones aceleradas de estos algoritmos en hardware a permitido su aplicación en problemas de ingeniería que requieren procesamiento en tiempo real, con toda la portabilidad que proporcionan diversas tecnologías (como los FPGA's).

Los algoritmos evolutivos utilizados como motores de búsqueda e implementados en algún hardware programable, han creado un nuevo paradigma llamado *hardware evolutivo* [26].

El hardware evolutivo es la integración del cómputo evolutivo y los dispositivos de hardware programables. El objetivo del hardware evolutivo es la reconfiguración "autónoma"

de la estructura del hardware con el fin de mejorar el desempeño. La capacidad de reconfiguración autónoma del hardware evolutivo lo hace fundamentalmente diferente del hardware convencional, donde es casi imposible cambiar la función del hardware una vez que es fabricado. Mientras que dispositivos de hardware programables tales como los PLD's y los FPGA's, permiten algunos cambios en la funcionalidad después de ser instalados sobre una placa de circuito impreso, tales cambios no pueden ser ejecutados sin la intervención de los ingenieros de diseño (es decir, el cambio no es autónomo). Con el uso del cómputo evolutivo, sin embargo, el hardware evolutivo tiene la capacidad de cambiar autónomamente sus funciones de hardware [26].

En [27], B. Shackleford et al. propusieron la aceleración de un algoritmo genético implementándolo en un FPGA. Las características del AG implementado son: selección aleatoria de los padres, la cual mantiene el sistema de circuitos de selección; un modelo de memoria de estado permanente, el cual conserva el área del chip; supervivencia de los cromosomas hijos más aptos sobre los cromosomas padres menos aptos, lo que promueve la evolución. El AG implementado sobre el FPGA está organizado en un proceso segmentado (*pipeline*) de seis estados, donde a cada estado se le asigna el mismo tiempo de procesamiento igual a un ciclo de reloj. Para sus experimentos, consideraron dos problemas, el problema del cubrimiento de conjuntos y el problema de plegamiento de proteínas. En el caso del problema de cubrimiento de conjuntos, destinaron tres FPGA's para el AG segmentado y otros tres para la función objetivo de dicho problema. Su diseño fue implementado en una tarjeta de circuitos programables en campo *Aptix AXB-MP3* con seis FPGA´s. Para el problema de plegamiento de proteínas, utilizaron una memoria para la población de 512 palabras * 82 bits (70 bits del cromosoma + 12 bits del costo), un AG segmentado, una interfaz con la PC y una sola función de costo, las cuáles fueron implementadas sobre un FPGA Xilinx XCV300 con 6,144 LUT's.

Entre sus resultados, para el problema de cubrimiento de conjuntos, corriendo a 1 MHz, el prototipo generó un millón de cromosomas nuevos por segundo, lo cual resultó ser 2,200x veces más rápido que una estación de trabajo ejecutando el mismo algoritmo escrito en C. El prototipo diseñado para el problema de plegamiento de proteínas produjo 66 millones de cromosomas no evaluados por segundo. La función de costo tiene un intervalo de inicialización segmentado de 36, lo que resulta en un rendimiento del cromosoma evaluado de

1.83 millones de cromosomas por segundo. La aceleración producida sobre un Pentium II a 366 MHz ejecutando el mismo algoritmo en C es de 320x.

Un procesador de algoritmos genéticos más eficiente fué propuesto en [28] basado en un AG de estado permanente, un AG basado en sobrevivientes modificado y selección mediante torneo modificado. Para realizar este diseño, los autores emplearon paralelización segmentada eficiente y un protocolo de comunicación, casi el 50% de la velocidad de cómputo se puede lograr sobre el AG basado en sobrevivientes el cual corre un millón de cruzas por segundo (1MHz). Para los experimentos, abordaron el problema de cubrimiento de conjuntos y sus diseños fueron realizados en VHDL. Su prototipo fue implementado sobre una tarjeta PCUGEN10K con un dispositivo EPF10K100A y un modulo SRAM. Su propuesta ocupó el 71% de 4992 bloques lógicos en el dispositivo corriendo a 24.2 MHz.

Otro trabajo relacionado al tema fué propuesto en [29], en donde se presenta el diseño de un algoritmo genético en VHDL llamado HGA (Hardware-based Genetic Algorithm) con el objetivo de implementarlo en hardware. Debido a los procesos de segmentación, paralelización y al no llamado de funciones, un AG en hardware alcanza una velocidad significativa sobre un AG en software, lo que es especialmente útil cuando el AG es usado para aplicaciones de tiempo real, por ejemplo, planificación del disco y registro de imágenes. Desde que un AG de propósito general requiere que la función de aptitud sea fácilmente cambiada, la implementación en hardware debe explotar la reprogramabilidad de ciertos tipos de FPGA's. Mientras que los FPGA's no son tan rápidos como los ASIC's típicos, estos aún mantienen una gran ventaja en velocidad sobre funciones ejecutándose en software. De hecho, la aceleración de orden-1 ó 2 de magnitud ha sido observada con frecuencia en rutinas de software implementadas con FPGA's. El HGA propuesto fue basado en el AG simple de Goldberg [29]. Los módulos del HGA fueron diseñados para correlacionar con las operaciones del AG simple, para ser simple y fácilmente escalable, y tener interfaces que faciliten la paralelización. También fueron diseñados para operar concurrentemente, produciendo un proceso segmentado de grano grueso. En la Fig. 4.1 se muestra el flujo de datos de la arquitectura propuesta para el HGA.

Existen otros trabajos similares sobre implementaciones de diversas variantes del AG, sin embargo, sólo hemos mencionado algunas de ellas.

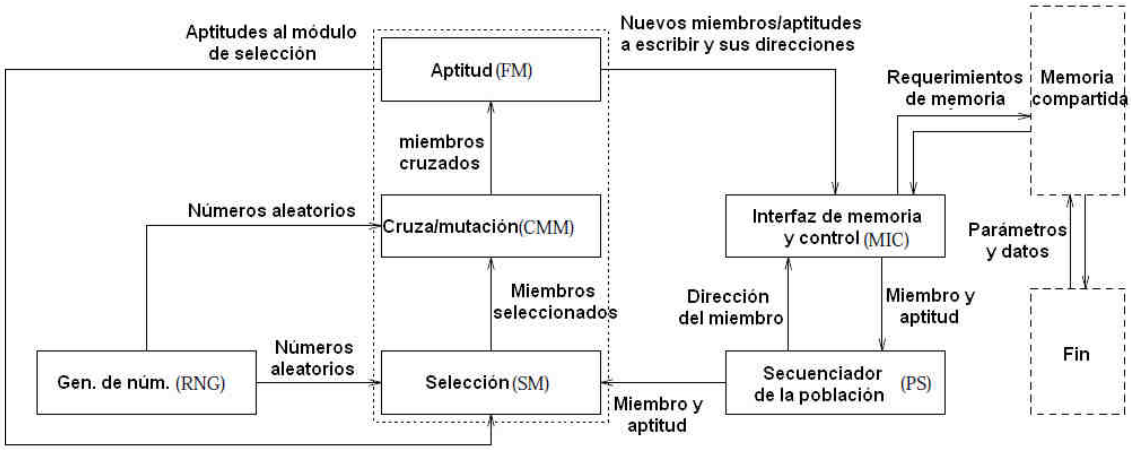


Fig. 4.1 Flujo de datos de la arquitectura del HGA.

Con el surgimiento del algoritmo genético compacto (AGc), los investigadores también se han dado a la tarea de proponer algunas arquitecturas, sobre todo por la simplicidad del algoritmo, ya que por su naturaleza, es más adecuado y fácil de implementar que un AG, pero haciendo énfasis, en que el AGc no es un sustituto del AG, sino una alternativa más para realizar optimización de una manera sencilla y económica (con respecto a los recursos) directamente sobre el hardware.

En [6], C. Aporntewan y P. Chongstitvatana proponen la implementación en hardware de un AGc. El diseño es realizado usando el lenguaje *Verilog* (lenguaje para descripción de hardware) y luego implementado en un FPGA. Sus resultados muestran que su diseño corre alrededor de mil veces más rápido que la versión en software ejecutándose sobre una estación de trabajo. Debido a que el AGc utiliza un vector *l*-dimensional, siendo *l* la longitud del cromosoma y manipulándolo en lugar de la población, esto reduce dramáticamente un número de bits requeridos para almacenar la población. Debido a que el algoritmo utiliza operaciones básicas: suma, resta y comparación, cada probabilidad puede ser actualizada en paralelo. Además, el AGc puede ser parcialmente traslapado. Esto permite un proceso segmentado que incrementa el desempeño del hardware. En su diseño proponen cinco módulos: un generador de números aleatorios, un registro de probabilidad, un comparador, un buffer y un evaluador de aptitud. La arquitectura que propusieron se muestra en la Fig. 4.2.

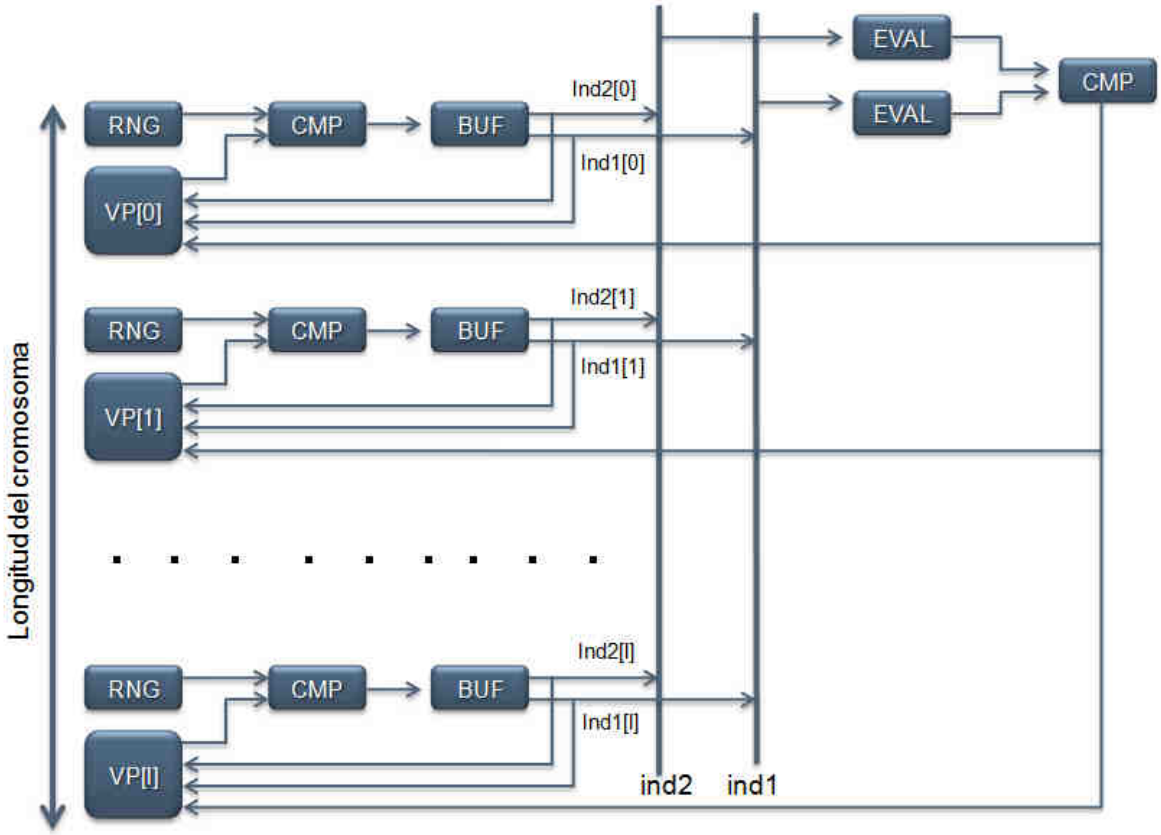


Fig. 4.2 Organización del hardware para el AGc [6].

Para evaluar el desempeño de su AGc sólo realizaron pruebas con el problema *max-one*, el cual consiste en maximizar el número de 1's en una cadena binaria. Como resultado, los autores destacan que el AGc ejecuta una generación por cada tres ciclos de reloj para el problema max-one. Como anotación, su diseño fue implementado en un FPGA de Xilinx a una frecuencia de 23.57 MHz, con un consumo de compuertas de 15,210 para el max-one de 32-bits con una población de 256.

J. C. Gallagher et al. proponen en [8] algunas modificaciones al AGc básico, lo cual mejora significativamente su capacidad de búsqueda sin incrementar el tamaño de la complejidad de la implementación en hardware. Este trabajo proporciona resultados comparativos demostrando el incremento en la eficacia y un diseño de flujo de datos/microcontrolador adecuado para la implementación en hardware digital. Entre sus modificaciones destacan la incorporación de elitismo y mutación al AGc, demostrando la eficacia de su propuesta con el conjunto de funciones de DeJong, las cuales son funciones

clásicas para evaluar el desempeño de algoritmos evolutivos. Los autores proponen igualmente, un diseño de flujo de datos que cubre toda la familia de AGc's propuestos por ellos. Su diseño es modular, y contiene los siguientes componentes:

1) Generador de números aleatorios (RND).
2) Registros de probabilidad B_PBR (elemento del vector de probabilidad) y M_PBR (probabilidad de mutación).
3) Buffer: RAM de 2 puertos que almacena los individuos.
4) Módulos INC/DEC: son los encargados de actualizar cada elemento del vector de probabilidad.
5) Registro E: Registro de un bit que indica cual individuo es el ganador.

Al igual que las otras arquitecturas, su diseño permite la paralelización de algunas secciones del algoritmo. En la Fig. 4.3 se muestra una parte de la arquitectura propuesta.

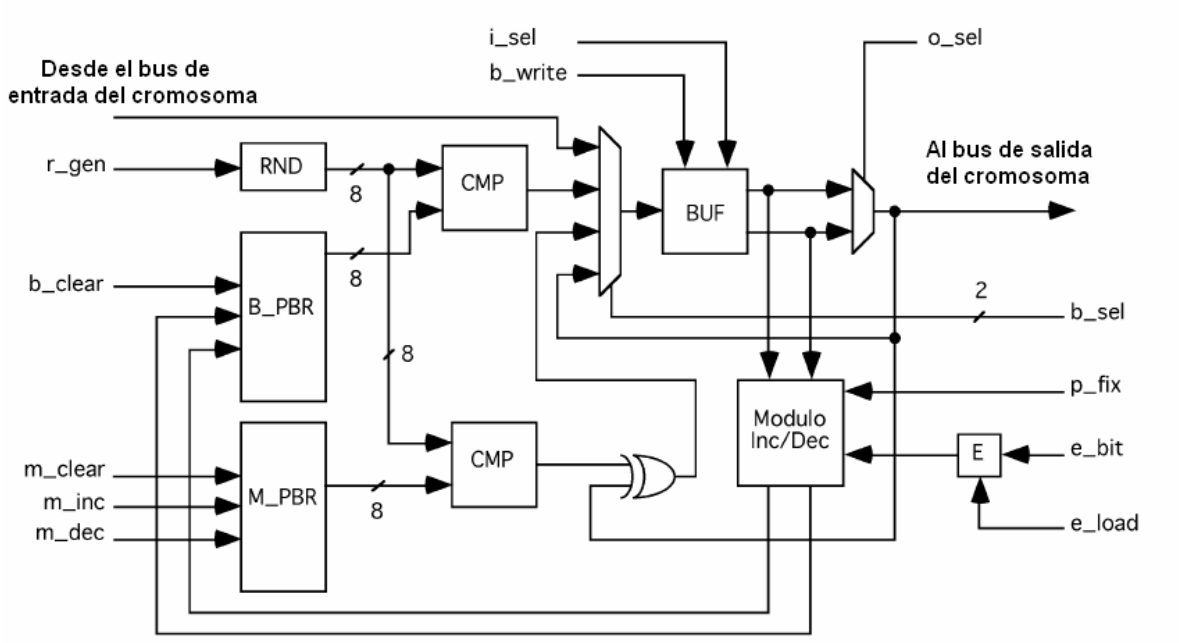


Fig. 4.3 Módulo para un bit de la familia de AGc's.

En la Fig. 4.3 destacan un grupo de señales que sirven de comunicación con el exterior. Estas señales son de control, ya que el funcionamiento del algoritmo en su totalidad es

sincronizado por un microcontrolador. El diagrama de flujo del microcontrolador para el AGc con elitismo y mutación es dado en la Fig. 4.4.

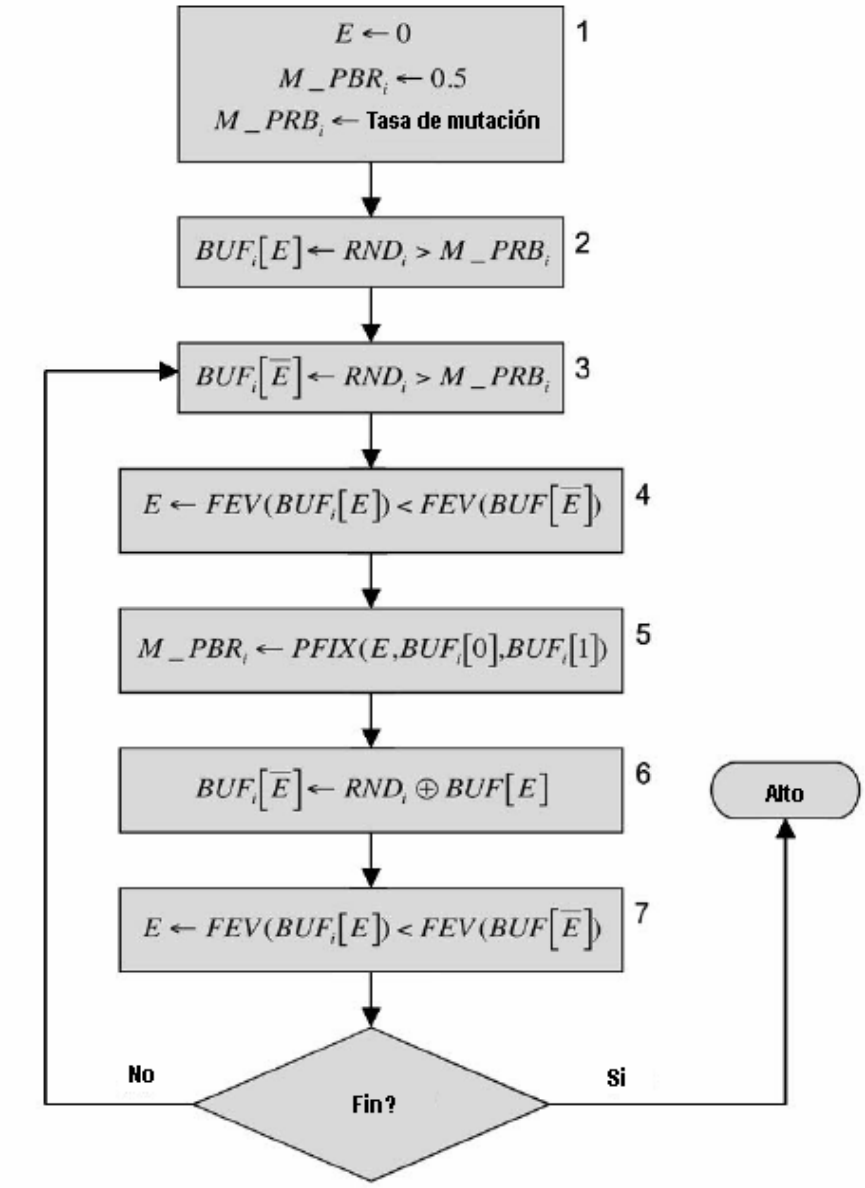


Fig. 4.4 Diagrama de flujo del microcontrolador del AGc con elitismo y mutación.

Sus diseños fueron realizados en VHDL implementando sus propuestas en una tarjeta XC4000 BORG y una Virtex xc2v1000. El problema tratado fue el max-one de 32-bits con una población simulada de 255 individuos. Las comparaciones realizadas son con respecto al consumo de recursos con la arquitectura propuesta en [6], mostrando que su propuesta ocupa

ligeramente más recursos, sin embargo, de acuerdo a sus pruebas de hardware, se tiene una mejor calidad de las soluciones.

Aprovechando la concurrencia de estas tecnologías, existen otros trabajos donde proponen AGc's paralelos, es decir, existen varios módulos en el FPGA conteniendo cada uno un AGc, los cuales pueden adoptar cualquier modelo de paralelización conocido, sea centralizado, global, de grano fino, de grano grueso, etc.

4.2 Aplicaciones de algoritmos bio-inspirados en hardware

Como habíamos mencionado anteriormente, un paradigma surgido de la combinación del cómputo evolutivo con los dispositivos de hardware programables, ha sido el hardware evolutivo. La evolución del hardware digital es la categoría más activa de la investigación sobre hardware evolutivo. Puede hacerse una diferenciación de dos tipos: algoritmos genéticos sobre hardware evolutivo y máquinas bio-inspiradas.

En [30], I. Kajitani et al. proponen el diseño de un motor en hardware de un AG y su aplicación a un controlador para la prótesis de una mano y su implementación en un chip *Large Scale Integration* (LSI). El motor en hardware de un AG es una implementación en hardware de las operaciones de un AG. Por tanto esto habilita la ejecución a alta velocidad de la búsqueda del AG así como su implementación en forma compacta.

El controlador mioeléctrico interpreta las intenciones de controlar del operador reconociendo señales mioeléctricas. Este tipo de controladores ha sido típicamente aplicado al control de prótesis eléctricas. La ventaja más notable de usar controladores mioeléctricos es su capacidad para utilizar las funciones musculares residuales de personas misnuválidas. Por ejemplo, en el caso de una prótesis de una mano, los controladores mioeléctricos habilitan al amputado para utilizar las funciones residuales de los músculos remanentes en su muñón.

Aunque el control mioeléctrico de prótesis de manos eléctricas ha sido investigado desde principios de los 60's, y algunas prótesis de manos ya están disponibles comercialmente, muchas son sistemas de una sola función, es decir, solo son capaces de desempeñar una sola función tal como abrir o cerrar, y son de esta manera de utilidad limitada en la vida cotidiana.

Consecuentemente, ha habido llamados para desarrollar sistemas multifuncionales, capaces de ejecutar más de una función.

En respuesta a esta demanda, recientemente, muchas investigaciones se han conducido sobre las prótesis del antebrazo multifuncionales, aplicando métodos de clasificación de patrones para determinar las acciones de la mano.

Mientras los controladores con redes neuronales son capaces de clasificar patrones con alta precisión, el tamaño es el principal obstáculo para la implementación compacta, la cual es requerida en este tipo de aplicaciones, donde la prótesis de la mano tiene que ser implementada para ser más pequeña y ligera que una mano normal. Debido a este problema, en [30] se propone utilizar un circuito lógico para clasificación de patrones mioeléctricos para realizar una prótesis de la mano multifuncional. Sin embargo, las señales mioeléctricas varían dependiendo de los individuos y del tiempo para un mismo individuo, por ello no es posible determinar las especificaciones exactas del circuito de clasificación. En consecuencia, un chip de hardware evolutivo es desarrollado para adoptar la clasificación de patrones mioeléctricos.

Como información adicional, los autores implementaron el hardware diseñado sobre un FPGA para validar su funcionalidad. El hardware fue diseñado usando un simulador del lenguaje Verilog y una herramienta de desarrollo para FPGA's (Quartus II de Altera). La tarjeta utilizada para su implementación fue la Stratix EP1S10F780C6. El tamaño del circuito implementado fue de 5,082 elementos lógicos y 69,504 bits de memoria. Su máxima velocidad de operación del reloj fue estimada para ser 97.32 MHz.

Otras aplicaciones prácticas han surgido en la actualidad, por ejemplo, existen métodos de compresión de datos sin pérdidas para imágenes de dos niveles utilizando hardware evolutivo. También se han implementado en FPGA's filtros de imágenes evolutivos, los cuáles están habilitados para evolucionar un filtro de imágenes en unos pocos segundos si imágenes originales corruptas son proporcionadas por el usuario [26].

En [9] y [31] se realizan implementaciones de algoritmos genéticos compactos sobre microcontroladores para realizar la optimización en línea de un controlador PI para un motor de inducción, con el fin de realizar control de velocidad y control de posición/velocidad respectivamente. Los autores implementaron sus esquemas de control sobre una tarjeta de

tiempo real dSPACE 1104, equipada con un microcontrolador Motorota PPC de 250Mhz para ambas implementaciones. Se destaca que el AGc realiza el ajuste de las ganancias proporcional e integral en tiempo real, esto es, el AGc aplica directamente las ganancias sobre el esquema de control, obteniendo la respuestas inmediatamente a través de sensores que le permiten realizar la evaluación de la función objetivo propuesta por los autores. Esta técnica permite al sistema ser robusto ante ruido cuando se realiza el ajuste de las ganancias, al aplicar directamente las ganancias en tiempo real sobre el sistema a controlar, evitando también errores de modelado que pueden surgir de la optimización fuera de línea realizada con un modelo obtenido matemáticamente de dicho sistema.

En [8] utilizan igualmente el AGc con algunas modificaciones propuestas por los autores para utilizarlo como un componente en un chip de control neuromórfico VLSI integrado. Su experimento consiste en hacer evolucionar un controlador de locomoción con redes neuronales recurrentes de tiempo contínuo (CTRNN).

Prácticamente, la mayoría de las aplicaciones de los algoritmos bio-inspirados utilizados como motores de optimización es orientada hacia el área de control, en donde se requiere hacer optimización de algún parámetro en línea o sobre aplicaciones de hardware evolutivo.

Capítulo 5

Desarrollo de la propuesta

5.1 Introducción

Como se ha mencionado desde el principio, el objetivo de este trabajo es realizar la implementación de un algoritmo genético compacto y sus versiones con elitismo y mutación sobre un FPGA. El diseño propuesto tiene las siguientes características:

- Modularidad: El diseño completo del AGc está divido en módulos o componentes que realizan una tarea específica que pueden fácilmente ubicarse en la concepción original del algoritmo, claro, con algunas variantes debido a las características particulares del hardware.
- Concurrencia: Debido a la naturaleza del algoritmo, se puede observar que existen partes paralelizables del mismo, esto es, debido a que se procesa un vector de probabilidades, cada actualización es independiente en cada elemento del vector, permitiendo que todo el vector sea actualizado paralelamente. De igual forma, en la generación de individuos, como cada bit que forma parte del individuo, se genera independiente uno del otro (a menos que use la misma fuente de generación de números aleatorios, entonces su generación sería secuencial), esta sección es igualmente paralelizable, es decir, los individuos pueden generarse en un sólo paso.
- Mínimo consumo de recursos: En la propuesta se experimentan algunas técnicas para ciertos módulos, las cuales permitirán el ahorro de recursos del dispositivo programable (FPGA).
- Ejecución en tiempo real: Gracias a las características de la tecnología a utilizar, es posible utilizar el diseño para aplicaciones en tiempo real.

- Escalabilidad: el propósito de tener un diseño modular es con el fin de que nuestro diseño sea escalable a cualquier tipo de problema, aprovechando la reprogramabilidad de los FPGA's.

El primer paso en el desarrollo de la propuesta, es definir los componentes a utilizar en nuestro diseño, los cuales serán descritos en el siguiente punto. Cabe mencionar, que el lenguaje utilizado para nuestros diseños es VHDL.

5.2 Componentes propuestos

Con el objetivo de aprovechar al máximo las características proporcionadas por los FPGA's, en cuanto a concurrencia, hemos propuesto los siguientes cinco componentes para el algoritmo genético compacto clásico ó estándar: un generador de números pseudos-aleatorios (una versión con autómatas celulares y otra basada en la reglas de Park y Miller), un generador de individuos, un evaluador de aptitud, un actualizador del vector de probabilidades y un verificador de dicho vector, los cuales serán descritos a continuación.

1) ***Generador de números pseudo-aleatorios ó Pseudo-Random Number Generator (PRNG):*** Este componente será el encargado de generar un número pseudo-aleatorio, y es esencial para la generación de los individuos. Es posible utilizar cualquier algoritmo conocido para la generación de números pseudo-aleatorios, sin embargo, el tipo de algoritmo utilizado impactará fuertemente en el consumo de recursos del FPGA al ser sintetizado el diseño completo, esto es, porque son necesarios varios módulos de este tipo. Para el componente *PRNG* hemos considerado dos algoritmos que son el generador estándar mínimo y el generador basado en autómatas celulares que a continuación se explicarán:

   Generador estándar mínimo: Este generador está basado en las reglas de Park y Millar [32] para la generación de secuencias pseudos-aleatorias, en combinación con el algoritmo de Schrage, el cual permite el uso de este algoritmo en arquitecturas de 32 bits. Este generador básicamente recibe una semilla I_j, la cual es un valor de tipo entero de 32 bits con signo (la semilla será positiva, diferente de cero y menor a

2147483647) y genera otro número I_{j+1} del mismo tipo positivo. La regla utilizada para la generación del siguiente número es:

$$I_{j+1} = aI_j \bmod m = \begin{cases} a(I_j \bmod q) - r[I_j / q] & Si\ I_{j+1} \geq 0 \\ a(I_j \bmod q) - r[I_j / q] + m & en\,caso\,contrario \end{cases} \qquad (5.1)$$

Con $a = 7^5 = 16807$, $m = 2^{31} - 1 = 2147483647$, $q = 127773$ y $r = 2836$. La operación $[I_j / q]$ indica que sólo se utiliza la parte entera del resultado de la división. Este generador tiene una distribución uniforme y su periodo es de $2^{31} - 2 \approx 2.1 * 10^9$. La generación de números es sobre el intervalo (0, 2147483647) [32]. La idea es retroalimentar el valor generado como semilla para el cálculo del siguiente elemento de la secuencia. Este componente será referenciado como *PRNG* a partir de ahora y el diseño propuesto del componente es mostrado en la Fig. 5.1.

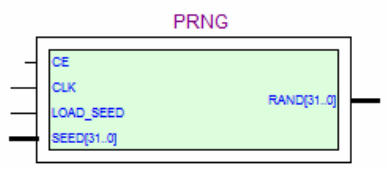


Fig. 5.1 Configuración del PRNG.

Para todos los componentes mostrados, del lado izquierdo se muestran las entradas y del lado derecho las salidas. Este componente cuenta con una entrada *CE* (Chip Enable), que habilita al componente para su funcionamiento; una señal de reloj *CLK* (que sincroniza la operación del componente); una entrada *SEED[31..0]* que es un entero de 32 bits con signo (la semilla) y una señal *LOAD_SEED* asíncrona que carga la semilla en el *PRNG*. Este componente se encuentra siempre activo y cada vez que se detecta un flanco de subida del reloj, genera un número pseudo-aleatorio. La arquitectura de este componente se muestra en la Fig. 5.2.

Generador basado en autómatas celulares ó Cellular Automata based Pseudo-Random Number Generator (CAPRNG): La idea de utilizar autómatas celulares (AC's) como generadores de números pseudo-aleatorios fue propuesta en [33]. Los autómatas celulares básicamente consisten en un arreglo de celdas de memoria, que

guardan el estado actual de cada célula. Las células contienen reglas, las cuales dependen del estado actual de la célula ó de algunos vecinos, que definen el estado siguiente del autómata. Formalmente, un autómata celular puede ser definido como una 5-tupla $C = \langle S, s_0, G, d, f \rangle$, donde S es un conjunto finito de estados, $s_0 \in S$ son los estados iniciales del AC, G es el vecindario celular, $d \in \mathbf{Z}^+$ es la dimensión de C y *f* es regla de interacción local, también llamada función de transición. Las células actualizan su estado simultáneamente de acuerdo a la regla local dada [34].

```
architecture comp of RNG is
signal   RN      :integer;
constant a       :integer:=16807;
constant m       :integer:=2147483647;
constant q       :integer:=127773;
constant r       :integer:=2836;
begin
   process(CE,CLK,RN,LOAD_SEED,SEED)
   variable randaux: integer;
   begin
     if CE='1' then
        if(clk='1' and clk'event) then
           randaux:=RN;
           randaux:=(a*(randaux-((randaux/q)*q)))-(r*(randaux/q));
           if randaux<0 then
              randaux:=randaux+m;
           end if;
           RN<=randaux;
        end if;
      end if;
      if LOAD_SEED='1' then RN<=conv_integer(SEED);end if;

    end process;
    RAND<=conv_std_logic_vector(RN,32);
end comp;
```

Fig. 5.2 Código en VHDL de la arquitectura del PRNG.

El autómata celular utilizado es un autómata unidimensional, compuesto de *n* células. Cabe mencionar que los estados del autómata son "0" y "1", lo que quiere decir que los números generados son en notación binaria. En [33] proponen un conjunto de *CAPRNG's* con longitudes desde 4 hasta 53 bits. En la tabla 5.1 se muestra la propuesta de los autores para construir tales *CAPRNG's*, donde se indica que reglas existen entre las células. Es importante mencionar que en este contexto se hace alusión a dos reglas conocidas como *Regla 90* y *Regla 150* (nombradas así de acuerdo a Wolfram [33]). La regla 90 es $a_i(t+1) = a_{i-1}(t) \oplus a_{i+1}(t)$ y la regla 150 es

$a_i(t+1) = a_{i-1}(t) \oplus a_i(t) \oplus a_{i+1}(t)$, donde $a_i(t+1)$ es el *estado siguiente* de la célula *i*, $a_{i-1}(t)$ es el *estado presente* de la célula *i-1* (vecino de la izquierda), $a_i(t)$ es el *estado presente* de la célula *i* (célula actual) y $a_{i+1}(t)$ es el *estado actual* de la célula *i+1* (vecino de la derecha).

n	***Regla***	***n***	***Regla***
4	0101	**29**	10101001010111001010001000011
5	11001	**30**	11101000100110110010101000111101
6	010101	**31**	01001100101011011111101110011000
7	1101010	**32**	01000110000010011011101111010101
8	11010101	**33**	00001100010011100111001011000005101
9	110010101	**34**	00111100001011010000110001101110 10
10	0101010101	**35**	01010111101111011001110101001010011
11	11010101010	**36**	10100110010010001111101011000010 0011
12	010101010101	**37**	00100101100111101011010110000101100 11
13	1100101010100	**38**	00011100101011110110011001111000010011
14	01111101111110	**39**	11010001011111011011110011001110110 1100
15	100100010100001	**40**	00001110110010101011111001000010111001 01
16	1101010101010101	**41**	01101011111110100001011001100011110000111
17	01111101111110011	**42**	00100111111011001110010100100110011110 0110
18	010101010101010101	**43**	00111010111000101110001000010110101100 10010
19	011010011011000 1001	**44**	00111100111101110101101110000100101011000010
20	11110011101101111111	**45**	00110100101100110110100100010011000110100 1101
21	011110011000000111011	**46**	00010010100110010100011010001011001110110 10110
22	0101010101010101010101	**47**	00111001011111100111001010100100010111000001101
23	11010111001110100011010	**48**	00011000011011111001001010011101000111100000 1111
24	11111101001011010101 0110	**49**	00101101111011001000110010111110001011101100 11001
25	10111101010100111100100	**50**	10011010011011000000110001101000101100100010010110
26	01011010110100010111011000	**51**	00010000101110101010000101101001110100010100001 0111
27	00001111100000110010000 1101	**52**	00110010001101111011101111111000100011110101110001 10
28	01010101010101010101010101	**53**	10000111001010001000001001001100101110111110110010101

Tabla 5.1 Reglas para diversas configuraciones del CAPRNG.

En la tabla 5.1, *n* indica el número de estados del *CAPRNG* y las reglas mostradas están dadas de la siguiente forma: Regla=0101, significa que para la célula '1' se debe usar la regla 90, para la célula '2' se debe usar la regla 150, para la célula '3' la regla 90 y para la célula '4' la regla 150; es decir, donde hay un "cero" se usa la regla 90, y donde hay un "uno" la regla 150, o viceversa (es igualmente válido).

Para las células de los extremos, como no existen vecinos a la izquierda o a la derecha, su estado es considerado como "0". En la Fig. 5.3 se muestra la configuración de un *CAPRNG* de 4 bits.

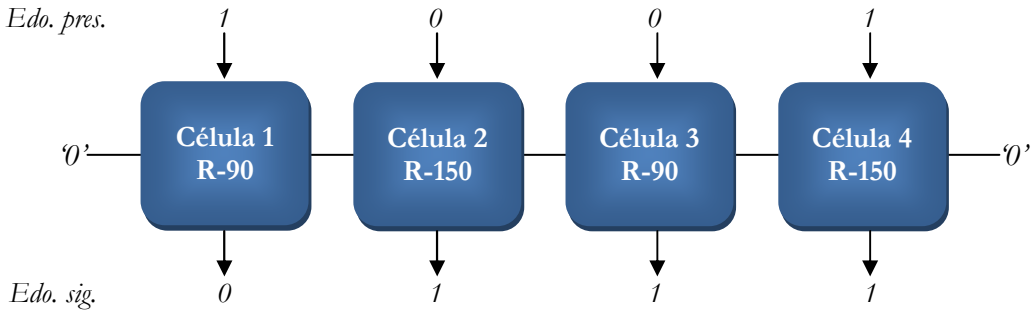


Fig. 5.3 Configuración de un CAPRNG de 4 bits.

En el caso de la célula '1', la regla 90 ($0 \oplus 0$) da un estado siguiente de '0'. Para la célula '2', la regla 150 ($1 \oplus 0 \oplus 0$) da un estado siguiente de '1', y así sucesivamente. En general, los *CAPRNG's* son capaces de generar secuencias de $2^n - 1$ elementos (siendo *n* el número de estados del CAPRNG). Con este tipo de generadores podemos emular poblaciones de $2^n - 2$ individuos (si realizamos incrementos y decrementos de una unidad sobre cada elemento del vector de probabilidades). En nuestro caso emularemos una población de 62 individuos, es decir, utilizaremos un *CAPRNG* de 6 bits, ya que es el más próximo a 50 individuos. El uso de los *CAPRNG's* en nuestra arquitectura propuesta, reduce solamente el tamaño de bus entre algunos componentes, por lo que la arquitectura no sufre alteración significativa alguna. La memoria de una célula (lo que almacena el estado del autómata) de un *CAPRNG* se puede implementar con un flip-flop síncrono tipo D, y cada regla se implementa con una LUT o con una función combinacional simple. En nuestro diseño, cada célula del *CAPRNG* está formada por una función combinacional y un elemento de memoria (flip-flop síncrono tipo D). En la Figs. 5.4 y 5.5 se muestran las células utilizadas para nuestra propuesta.

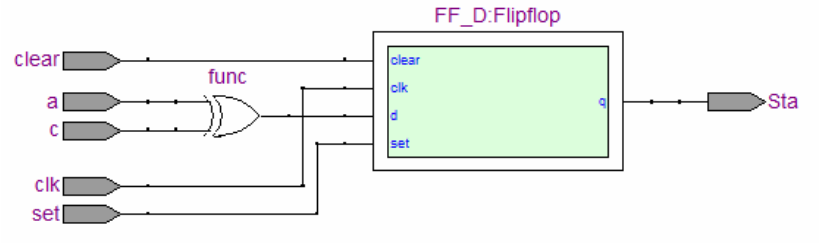


Fig. 5.4 Estructura interna de una célula con la regla 90.

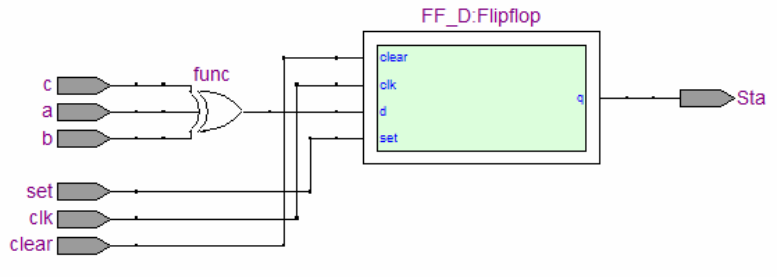


Fig. 5.5 Estructura interna de una célula con la regla 150.

A partir de ahora denominaremos *célula-90* a aquellas células conteniendo la regla 90 y *célula-150* a las que contengan la regla 150. Ya disponiendo de las células-90 y 150, la construcción de un CAPRNG de 6 bits se realiza de acuerdo a las reglas de la tabla 5.1, por lo que el diseño de nuestro CAPRNG de 6 bits se muestra en la Fig. 5.6.

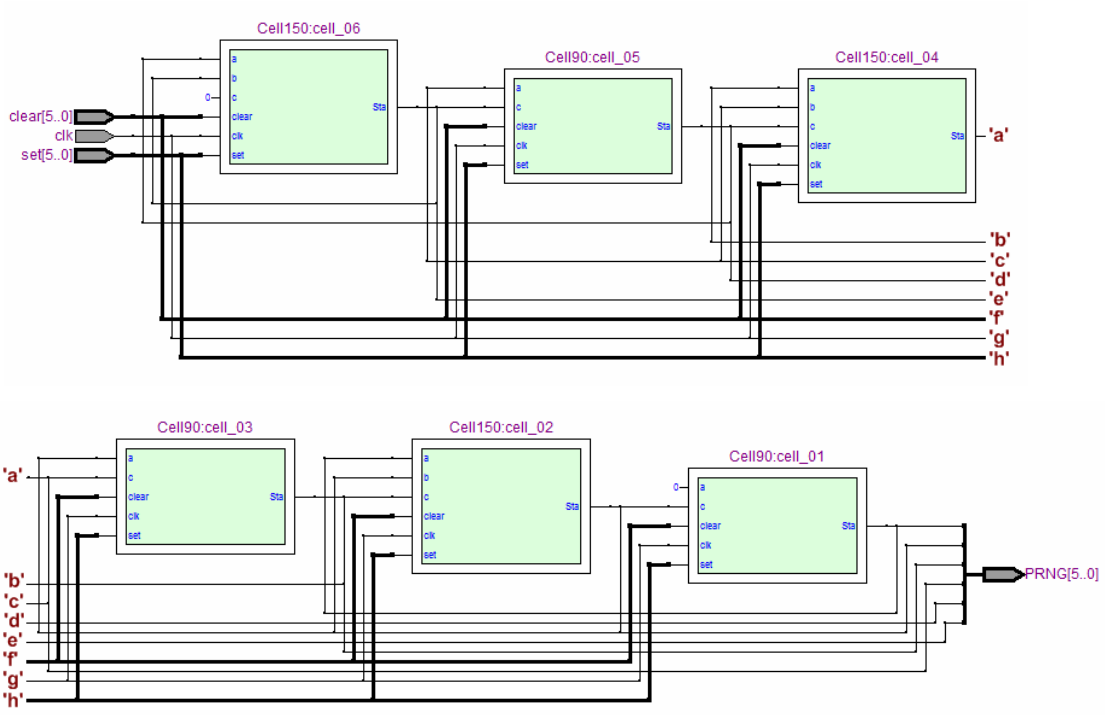


Fig. 5.6 Estructura interna de un CAPRNG de 6 bits.

El módulo *CAPRNG* visto como un componente que puede ser utilizado en cualquier otro diseño se muestra en la Fig. 5.7.

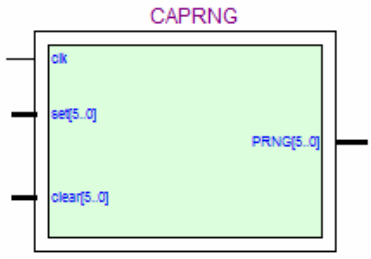


Fig. 5.7 Componente CAPRNG de 6 bits.

El funcionamiento de este componente es simple, cada vez que se detecta un flanco de subida de la señal de reloj (*clk*), se genera un nuevo número aleatorio (cadena o arreglo binario de 6 bits de longitud para este diseño). Es importante mencionar, que las entradas *set* y *clear*, forman parte de los flip-flops de cada célula interna del *CAPRNG*, por lo que para la generación de los números pseudo-aleatorios, estas entradas deben ser cero, sin embargo, en la etapa de inicialización, la semilla es configurada de acuerdo a estas entradas, es decir, si la semilla para el *CAPRNG* de 6 bits es "100101", las entradas deben ser configuradas como *set*="100101" y *clear*="011010", y posteriormente deben ser configuradas como "000000" para ambas entradas. Esta inicialización es asíncrona (se actualiza automáticamente sin depender del flanco de subida de la señal de reloj). En resumen, la entrada *clear* debe ser el inverso de la entrada *set* en la etapa de inicialización de los *CAPRNG's*.

2) ***Generador de individuos o Individuals Generator (IndGen):*** Este componente se encarga de la generación de un par de bits, esto es, un bit para cada individuo, por tanto, si los individuos tienen una longitud de 10 bits, tendremos diez componentes de este tipo. El componente propuesto es mostrado en la Fig. 5.8.

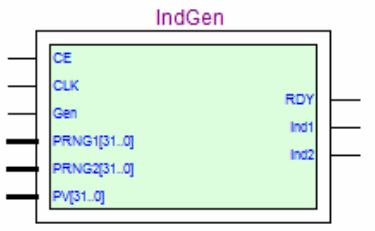


Fig. 5.8 Componente IndGen con entradas de 32 bits.

Este componente tiene las siguientes entradas y salidas:

CE: Señal de entrada que habilita al componente para funcionar y que cuando es puesta a '0' resetea la señal *RDY* (Ready) además de deshabilitar dicho componente.

CLK: Señal de reloj.

GEN: Señal de entrada que activa la generación del par de bits.

PRNG1, PRNG2: Buses de entrada para los números pseudo-aleatorios de 32bits.

PV: Bus de entrada del i-ésimo elemento del vector de probabilidades de 32 bits.

RDY: Señal de salida que indica que la generación de los bits ha terminado.

Ind1, Ind2: Señales de salida para el par de bits generados.

La arquitectura de este componente es mostrada en la Fig. 5.9.

```
architecture comp of IndGen is
begin
  process(CE,CLK)
  variable rng1   :integer;
  variable rng2   :integer;
  variable vp     :integer;
  begin
    if CE='1' then
       if(clk='1' and clk'event) then
          if Gen='1' then
             rng1:=conv_integer(PRNG1);
             rng2:=conv_integer(PRNG2);
               vp:=conv_integer(PV);
            if rng1<=vp then ind1<='1';
            else ind1<='0';
            end if;
            if rng2<=vp then ind2<='1';
            else ind2<='0';
            end if;
            RDY<='1';
          end if;
        end if;
    else
       RDY<='0';
    end if;
  end process;
end comp;
```

Fig. 5.9 Código en VHDL de la arquitectura del IndGen.

La forma en que opera este componente es sencilla. Su función consiste en comparar un número pseudo-aleatorio recibido (número entero de 32 bits) y compararlo con el elemento del vector de probabilidades. Si el número pseudo-aleatorio es menor o igual al elemento del vector de probabilidades, el bit generado es '1', de lo contrario es '0'. Esto se hace para ambos números pseudos-aleatorios, con el fin de generar los bits *ind1* e *ind2*. El componente descrito funciona con números pseudo-aleatorios de 32 bits.

Para el caso de números pseudo-aleatorios de 6 bits, como en el caso de nuestro *CAPRNG*, el componente debe ser modificado, como en la Fig. 5.10.

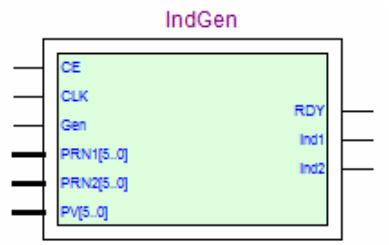


Fig. 5.10 Componente IndGen con entradas de 6 bits.

Las entradas y salidas tienen la misma función, a diferencia de los números pseudo-aleatorios *PRN1* y *PRN2*, y la probabilidad *PV* que ahora son de 6 bits. La arquitectura para esta variante es mostrada en la Fig. 5.11.

```
architecture comp of IndGen is
begin
  process(CE,CLK)
  variable rn1    :integer range 0 to 63;
  variable rn2    :integer range 0 to 63;
  variable Vect   :integer range 0 to 63;
  begin
    if CE='1' then
       if(clk='1' and clk'event) then
          if Gen='1' then
             rn1:=conv_integer(PRN1);
             rn2:=conv_integer(PRN2);
             vect:=conv_integer(PV);
            if rn1<=vect then ind1<='1';
            else ind1<='0';
            end if;
            if rn2<=vect then ind2<='1';
            else ind2<='0';
            end if;
            RDY<='1';
          end if;
        end if;
    else
       RDY<='0';
    end if;
  end process;
end comp;
```

Fig. 5.11 Código en VHDL de la arquitectura del IndGen para un CAPRNG de 6 bits.

3) ***Evaluador de aptitud ó Fitness Evaluator (FEv):*** Este componente se utiliza en el caso de que se realice la evaluación de la función objetivo sobre el hardware (FPGA). El componente propuesto para desempeñar tal función se muestra en la Fig. 5.12.

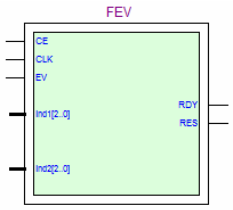


Fig. 5.12 Componente FEv.

Este componente puede variar radicalmente dependiendo de la función objetivo a implementar y además, puede consumir más tiempo de acuerdo a la complejidad de la función objetivo. Se puede optimizar este bloque de acuerdo a la habilidad del diseñador para implementar funciones complejas segmentadas o paralelas. En nuestro caso, presentaremos el diseño de una función objetivo que realiza el conteo de 1's en una cadena binaria (cada individuo). Las entradas de este componente son: *CE* que habilita al *FEv*; una señal de reloj *CLK*; una entrada *EV* que activa el cálculo de la aptitud y dos vectores de entrada *Ind1* e *Ind2*, que son los individuos a evaluar. Como salidas se tienen una señal *RDY* que indica que la evaluación y la comparación han terminado, y una señal *RES*, la cual si es '1' indica que la aptitud del individuo 1 es mayor o igual que la del individuo 2, si es cero, indica que la aptitud del individuo 2 es mayor. La arquitectura propuesta para la función objetivo señalada anteriormente es mostrada en la Fig. 5.13.

4) ***Actualizador del vector de probabilidades ó Probability Vector Updater (PVU):*** La tarea que realiza este componente es simplemente sumar o restar cierta cantidad a cada elemento del vector de probabilidades. Si utilizamos el *CAPRNG*, solo se suma o se resta '1' a cada elemento del vector según sea el caso. Si empleamos el *PRNG* de 32 bits, el factor puede variar, de acuerdo a la población que se esté emulando. En la Fig. 5.14 se muestran los componentes para ambos casos.

   En la Fig. 5.14 se puede apreciar que ambos componentes se diferencian en la salida *PV*, esto es porque depende del generador de números pseudo-aleatorios. *PV* será de 32 bits en el caso del *PRNG* y de 6 bits en el caso del *CAPRNG*. Al igual que los componentes anteriores, cuenta con una señal RDY que indica que la actualización ha terminado.

```
architecture comp of FEv is
begin
   process(CE,CLK)
   variable fit1,fit2  :integer;
   begin
      if CE='1' then
         if(clk='1' and clk'event) then
            if EV='1' then
               fit1:=0;fit2:=0;
               for i in 2 downto 0 loop
                  if ind1(i)='1' then fit1:=fit1+1; end if;
                  if ind2(i)='1' then fit2:=fit2+1; end if;
               end loop;
               if fit1>=fit2 then RES<='1';
                else RES<='0'; end if;
                RDY<='1';
            end if;
          end if;
       else
          RDY<='0';
       end if;
   end process;
end comp;
```

Fig. 5.13 Código en VHDL de la arquitectura de la función objetivo propuesta.

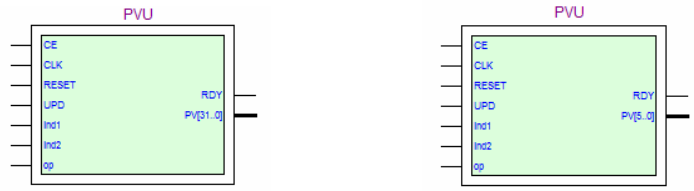


Fig. 5.14 Componentes PVU.

El componente *PVU* tiene las siguientes entradas: *CE* que habilita al componente, una señal de reloj *CLK*, una señal de *RESET* para reinicializar el vector de probabilidades, una señal *UPD* que activa la actualización de cada elemento del vector, un par de entradas *ind1* e *ind2* que son los elementos correspondientes de los individuos necesarios para actualizar el elemento del vector y una entrada *op* que indica cual de los individuos es el mejor. La actualización se realiza en paralelo, todos los elementos del vector a la vez. Para que se lleve a cabo una actualización sobre un elemento del vector, las entradas *ind1* e *ind2* deben ser diferentes. Un aspecto contemplado en el diseño son los límites de cada elemento del vector, es decir, en el caso de un elemento del vector de 6 bits, su máximo valor a llegar es 63, si se le hace un incremento, se desborda el dato, así que se agregaron unas condiciones para evitar esto en ambos extremos del intervalo.

La arquitectura para este componente es mostrada en la Figs. 5.15 y 5.16.

```
architecture comp of PVU is
signal   oe        :std_logic;
signal   rdydata   :std_logic;
signal   vect      :integer:=1073741823;
begin
  process(CE,CLK,RESET)
  begin
    if CE='1' then
      if(clk='1' and clk'event) then
        if UPD='1' then
          if (oe='1' and ind1='1') then
            if (op='1' and vect<2147483623) then
              vect<=vect+42949672;
            elsif (op='0' and vect>23) then
              vect<=vect-42949672;
            end if;
          elsif oe='1' and ind1='0' then
            if (op='1' and vect>23) then
              vect<=vect-42949672;
            elsif (op='0' and vect<2147483623) then
               vect<=vect+42949672;
            end if;
           end if;
          RDY<='1';
         end if;
        end if;
      else
      RDY<='0';
    end if;
    if RESET='0' then vect<=1073741823; end if;
  end process;
  PV<=conv_std_logic_vector(vect,32);
  oe<=ind1 xor ind2;
end comp;
```

Fig. 5.15 Código en VHDL de la arquitectura del PVU para 32 bits.

En el código de la Fig. 5.15 se puede notar la señal de *RESET*, la cual es necesaria para reinicializar cada elemento del vector de probabilidades. Se cuenta con una señal *oe*, que básicamente indica si los bits *ind1* e *ind2* son diferentes, ya que esta es la condición para realizar una actualización sobre el elemento correspondiente del vector de probabilidades. En este caso, definimos los pasos de incremento o decremento de cada elemento del vector de probabilidades de 42949672, esto para que se pueda emular la población de 50 individuos. Esta cantidad fue definida de la siguiente manera, se divide el máximo valor que puede alcanzar cada elemento del vector entre el número de individuos, 2147483647/50= 42949672.94, redondeando este valor al entero próximo menor, es decir, 42949672. Cada elemento del vector es inicializado con 1073741823,

ya que es el punto medio del intervalo. Por tanto, avanzando 25 pasos más se llega a 2147483623 (máximo valor) y decrementando 25 pasos se obtiene 23, por ello estos valores son los límites y se considerarán como el '1' ó '0' a converger.

```
architecture comp of PVU is
signal   oe        :std_logic;
signal   rdydata   :std_logic;
signal   vect      :integer range 0 to 63:=32;
begin
  process(CE,CLK,RESET)
  begin
    if CE='1' then
      if(clk='1' and clk'event) then
        if UPD='1' then
          if (oe='1' and ind1='1') then
            if (op='1' and vect<63) then
              vect<=vect+1;
            elsif (op='0' and vect>1)  then
              vect<=vect-1;
            end if;
          elsif oe='1' and ind1='0' then
            if (op='1' and vect>1) then
              vect<=vect-1;
            elsif (op='0' and vect<63) then
              vect<=vect+1;
            end if;
          end if;
          RDY<='1';
        end if;
      end if;
    else
    RDY<='0';
    end if;
  if RESET='0' then vect<=32; end if;
  end process;
  PV<=conv_std_logic_vector(vect,6);
  oe<=ind1 xor ind2;
end comp;
```

Fig. 5.16 Código en VHDL de la arquitectura del PVU para 6 bits.

En el caso del *PVU* con elementos de 6 bits en el vector de probabilidades, la población emulada será de 62 bits, así que los pasos serán de una unidad.

5) ***Verificador del vector de probabilidades ó Probability Vector Checker (PVC):*** Este componente verifica si un determinado elemento del vector ha convergido. La Fig. 5.17 muestra el diseño propuesto para este componente. Este componente contiene las siguientes entradas: una señal *CE* que habilita al componente, una señal de reloj *CLK*, una señal *chk* que activa la verificación del elemento del vector y la entrada *PV* que recibe el elemento correspondiente del vector.

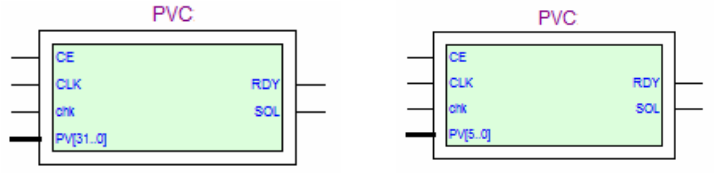


Fig. 5.17 Componentes PVC.

La arquitectura del *PVC* es mostrada en la Fig. 5.18 y 5.19. La Fig. 5.18 muestra el componente para verificar un elemento del vector de 6 bits y la Fig. 5.19 el diseño para verificar un elemento de 32 bits.

```
architecture comp of PVC is
begin
  process(CE,CLK,PV)
  variable vect :integer;
  begin
    vect:=conv_integer(PV);
    if CE='1' then
      if(clk='1' and clk'event) then
        if chk='1' then
          if vect=1 then SOL<='0';RDY<='1'; end if;
          if vect=63 then SOL<='1';RDY<='1'; end if;
        end if;
      end if;
    else
      RDY<='0';
    end if;
  end process;
end comp;
```

Fig. 5.18 Código en VHDL de la arquitectura del PVC para 6 bits.

```
architecture comp of PVC is
begin
  process(CE,CLK,PV)
  variable vect :integer;
  begin
    vect:=conv_integer(PV);
    if CE='1' then
      if(clk='1' and clk'event) then
        if chk='1' then
          if vect=23 then SOL<='0';RDY<='1'; end if;
          if vect=2147483623 then SOL<='1';RDY<='1'; end if;
        end if;
      end if;
    else
    RDY<='0';
    end if;
  end process;
end comp;
```

Fig. 5.19 Código en VHDL de la arquitectura del PVC para 32 bits.

Los componentes presentados anteriormente fueron utilizados para construir la arquitectura del Algoritmo Genético Compacto clásico (AGc), ahora describiremos los componentes diseñados para construir el Algoritmo Genético Compacto con Elitismo y Mutación (AGcEM), algunos de los cuales son variantes de los componentes anteriores. En el diseño del AGcEM se utilizan algunos componentes del diseño del AGc que son el *CAPRNG* de 6 bits y el *PVC*. Los componentes empleados para el AGcEM son el *CAPRNG* de 5 bits, el generador del individuo uno, la mutación del individuo elitista, el actualizador del individuo elitista y el actualizador del vector de probabilidades, los cuales se describen a continuación.

1) ***CAPRNG de 5 bits:*** Este CAPRNG se utiliza para la mutación del individuo elitista (*indE*). Debido a que en los experimentos se emplean individuos de 32 bits, la probabilidad de mutación utilizada fue de $P_m = 1/Longitud\ del\ cromosoma$, es decir, 1/32. Como el *CAPRNG*-5bits genera una secuencia de números aleatorios con un periodo de 31 (2^5-1), la mutación se puede realizar detectando un número elegido de tal secuencia, esto da una probabilidad de mutación de 1/31, que es cercano a lo deseado. La configuración del autómata celular para el *CAPRNG*-5bits fue tomada de la tabla 5.1. En la Fig. 5.20 se muestra el componente que contiene el *CAPRNG*-5bits y opera de manera similar al *CAPRNG* de 6 bits.

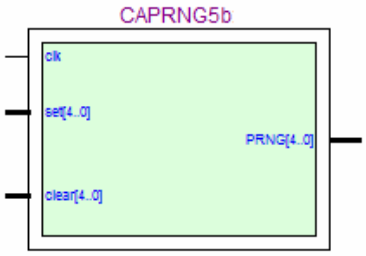


Fig. 5.20 Componente CAPRNG de 5 bits.

2) ***Generador del individuo uno ó Individual 1 Generator (Ind1Gen)*:** Este componente fue diseñado con el propósito de generar sólo un individuo. Utiliza la misma técnica para generar individuos que el componente ***IndGen*** para el AGc, sólo que en este caso se genera un individuo. El módulo propuesto se muestra en la Fig. 5.21. Este componente contiene una señal de entrada *Gen* que activa la generación del individuo 1, una señal *PRN* donde se recibe el número aleatorio generado por el

CAPRNG-6bits, una señal *PV* donde se recibe el valor actualizado del i-ésimo elemento del vector de probabilidades y una señal de salida *Ind1* que es el i-ésimo bit del individuo 1 generado a partir del vector de probabilidades. Se requieren añadir varios de estos componentes para generar el individuo en paralelo.

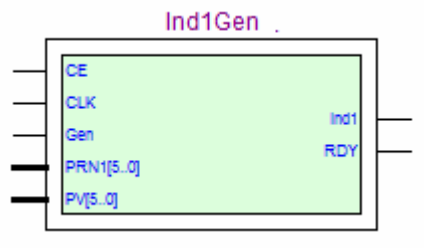


Fig. 5.21 Componente Ind1Gen.

3) ***Mutación del individuo elitista ó Elitist Individual Mutation (EIndMut):*** Este componente realiza la mutación sobre el individuo elitista (*indE*) a nivel de bits. El componente *EIndMut* es mostrado en la Fig. 5.22. Este componente contiene las siguientes entradas: *EInd* que recibe un bit del individuo elitista, *Mut* es la señal que activa la mutación y *PRNm* que es el número aleatorio recibido del componente *CAPRNG*-5bits. La mutación se lleva a cabo con la siguiente condición: *if unsigned(PRNm)=5 then MInd<=not EInd; else MInd<=EInd*; básicamente indica que si el número aleatorio es 5, se invierte el bit del individuo elitista y se guarda en el individuo mutado, de lo contrario, se guarda directamente. El componente tiene como salida un bit del individuo mutado (*MInd*).

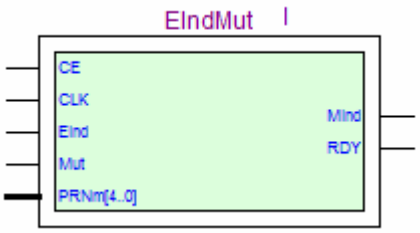


Fig. 5.22 Componente EIndMut.

4) ***Actualizador del individuo elitista ó Elitist Updater (EUpd):*** Este componente realiza la evaluación del individuo mutado y el individuo 1, y actualiza el individuo elitista con respecto a la aptitud de estos individuos. La actualización consiste en

comparar inicialmente la aptitud del individuo elitista (*fitE*) y la aptitud del individuo mutado (*fitM*), si *fitM* es mayor o igual que *fitE*, entonces se hace un intercambio entre los registros *indE* e *indM*, de lo contrario, permanecen iguales. Luego, se realiza otra comparación entre la aptitud del individuo 1 (*fit1*) y la aptitud del individuo elitista (*fitE*), y si *fit1*es mayor o igual que *fitE*, entonces se intercambian los registros *indE* e *ind1*, de lo contrario, permanecen iguales. No se realiza el cálculo de *fitE*, ya que es un dato de entrada al componente, al igual que *Ind1*, *MInd* e *EInd*. Como salida se actualiza el individuo elitista (*UpdEInd*), el individuo 1 (*UpdInd1*) y la aptitud del individuo elitista (*UpdFitEInd*). La Fig. 5.23 muestra el componente *EUpd*.

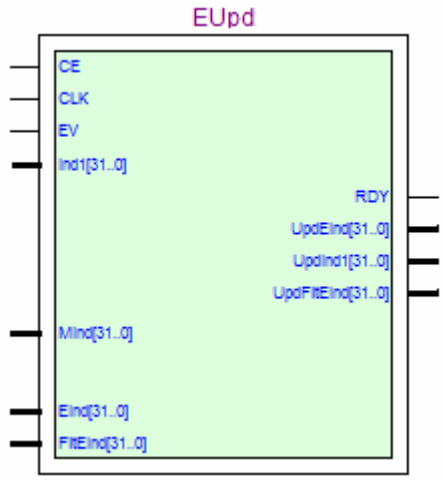


Fig. 5.23 Componente EUpd.

5) Actualizador del vector de probabilidades ó Probability Vector Updater (PVU): Este componente realiza la misma tarea que su similar para el AGc, sin embargo, se realizaron algunas modificaciones debido a la presencia del individuo elitista. Este componente realiza una operación XOR entre un bit del individuo 1 (*Ind1*) y un bit del individuo elitista (*IndE*), esto para saber si los bits son diferentes, ya que en consecuencia se debe realizar la actualización del elemento correspondiente del vector de probabilidades (*PV*) con respecto al individuo elitista (individuo ganador). Si el bit del individuo elitista es uno, entonces el elemento del vector de probabilidades se debe incrementar, de lo contrario se debe disminuir. El factor de incremento o disminución debe ser *1/n*, siendo *n* el tamaño de la población, sin embargo, cada elemento del

vector es un entero entre (0, 63), por lo que basta con que ese factor sea de una unidad para emular una población de 62 individuos. La Fig. 5.24 muestra el componente *PVU* empleado en el AGcEM. Las entradas y salidas del componente son similares al componente *PVU* diseñado para el AGc.

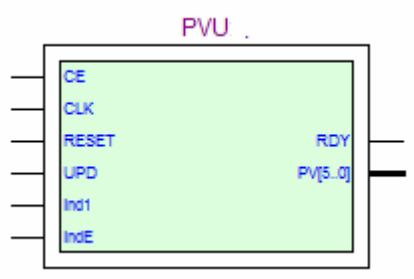


Fig. 5.24 Componente PVU.

Una vez construidos los componentes anteriores, el siguiente paso es diseñar la arquitectura del algoritmo genético compacto y su variante.

5.3 Organización del hardware propuesto

Ya diseñados los componentes de la sección anterior, se deben interconectar entre sí y sincronizarlos de tal forma que se pueda obtener la funcionalidad deseada, es decir, la del AGc y la del AGcEM. A continuación describiremos la arquitectura propuesta para el **AGc**.

Arquitectura del Algoritmo Genético Compacto

En la sección anterior se planteó el diseño de dos tipos de generadores de números pseudo-aleatorios, el *PRNG* (para números enteros de 32 bits) y el *CAPRNG* de 6 bits (para números enteros de 6 bits), por lo que se diseñaron dos arquitecturas, es decir, una para cada generador. La diferencia radica en el tamaño de cada elemento del vector de probabilidades y por ende, en los buses que interconectan a los números pseudo-aleatorios con los componentes que los requieren. Estas diferencias hicieron necesario ajustar los componentes de acuerdo al tamaño de dichos datos, por lo que en la descripción de los componentes del punto anterior se hizo alusión a dos variantes en el caso del AGc.

La arquitectura propuesta para el AGc estándar utilizando el generador *PRNG* para enteros de 32 bits se muestra en la Fig. 5.25 y en la Fig. 5.26 se muestra la arquitectura del AGc utilizando el generador *CAPRNG* de 6 bits.

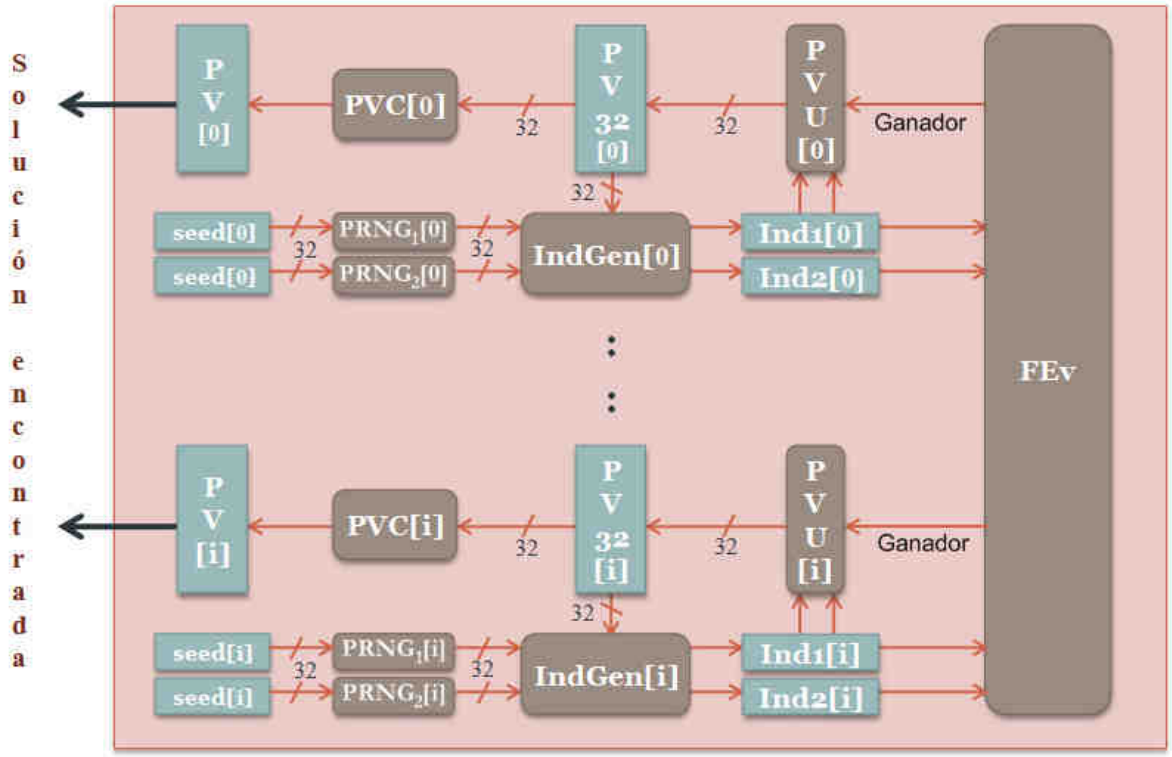


Fig. 5.25 Arquitectura propuesta para el AGc con PRNG's de 32 bits.

La Fig. 5.25 muestra un esquema resumido de la arquitectura empleada para el diseño del AGc que incorpora el *PRNG* basado en el generador estándar mínimo (Reglas de Park y Miller). Los componentes diseñados se muestran sobre un rectángulo con esquinas redondas. Los rectángulos muestran registros que almacenan el dato indicado. Para esta arquitectura, se tienen buses de 32 bits.

El registro *seed[i]* contiene la semilla para el componente *PRNG[i]* que genera un entero de 32 bits. El componente *IndGen[i]* recibe dos números enteros de 32 bits generados aleatoriamente y el elemento del vector de probabilidades correspondiente *PV[i],* y en consecuencia genera dos bits que son guardados en los registros *Ind1[i]* e *Ind2[i]*. Los registros *Ind1* e *Ind2* son actualizados en paralelo y son datos de entrada para el componente *FEv* que realiza el cálculo de la aptitud de cada individuo y determina quien es el ganador. El

componente *PVU* recibe el dato de cual individuo es el ganador, además del i-ésimo bit correspondiente a cada individuo (*ind1[i]* e *ind2[i])* y actualiza el i-ésimo elemento del vector de probabilidades *PV32[i]*, que es un registro de 32 bits. El componente *PVC* lee el dato almacenado en *PV32[i]* y verifica la convergencia del mismo, si tal dato ha convergido se guarda en el registro *PV[i]* (registro de un bit). Una vez que el AGc ha concluido, el resultado que se muestra a la salida es el dato almacenado en el registro *PV*.

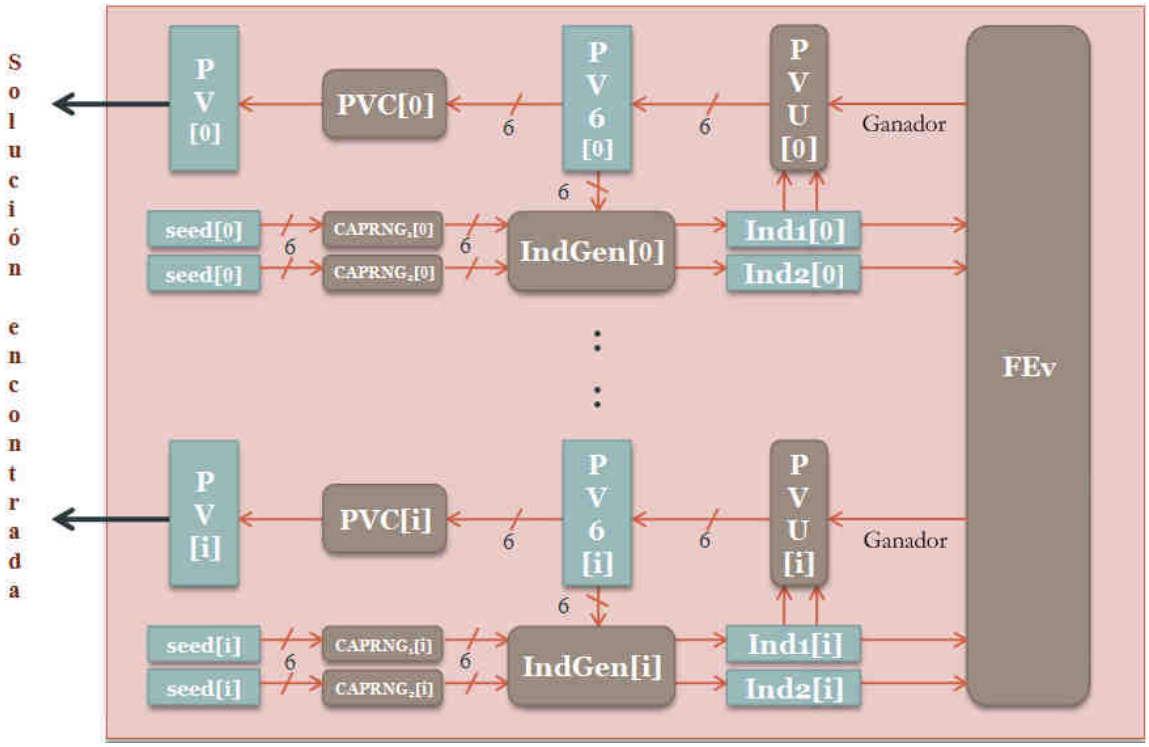


Fig. 5.26 Arquitectura propuesta para el AGc con CAPRNG´s de 6 bits.

La Fig. 5.26 muestra la arquitectura del AGc que incluye generadores de números pseudo-aleatorios basados en autómata celulares de 6 bits (CAPRNG-6bits). En este caso, los buses son de 6 bits en lugar de 32, debido a que los CAPRNG's generan números enteros de 6 bits. Los componentes operan de forma similar que en el diseño anterior para el AGc, con la diferencia de que el registro *PV32* ha sido sustituido por un registro *PV6*, es decir, datos enteros de 6 bits en lugar de 32 bits. En consecuencia, los componentes que requieren acceder a este registro han sido adaptados al nuevo tamaño.

Las arquitecturas tienen una característica interesante, las operaciones que se aplican a cada elemento de un cierto vector se realizan en paralelo. Por ello, cada componente cuenta con una señal *RDY* que indica que el componente ha finalizado su tarea. Como se requieren varios componentes para procesar el vector completo (en paralelo), las señales *RDY* autorizan el siguiente paso del algoritmo, es decir, todas deben valer uno.

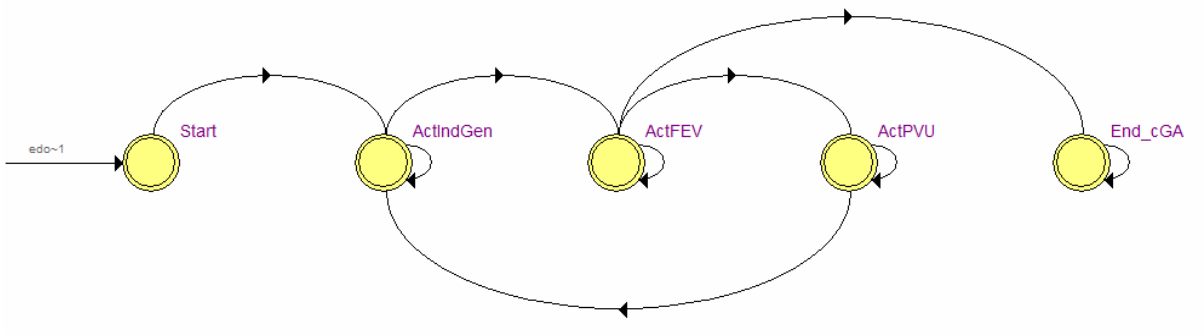


Fig. 5.27 Máquina de estados finitos para el AGc.

La sincronización de los componentes se realiza a través de una máquina de estados finitos ó *Finite State Machine* (FSM). Una máquina de estados finitos es un circuito secuencial con un número finito de estados [25]. La utilización de esta máquina garantizará el comportamiento correcto del diseño completo. Para nuestro diseño utilizamos una máquina de Mealy, la cual utiliza una señal de reloj para la sincronización de los estados. En la Fig. 5.27 se muestra la FSM utilizada en nuestra propuesta y en la tabla 5.2 se muestran las reglas de transición.

	Estado presente	**Estado siguiente**	**Regla**
1	Start	ActIndGen	
2	ActIndGen	ActIndGen	flag≠'0' and RDY_PVU≠"111"
3	ActIndGen	ActFEV	flag='0' or RDY_PVU="111"
4	ActFEV	ActFEV	RDY_IndGen≠"111"
5	ActFEV	ActPVU	RDY_IndGen="111" and RDY_PVC≠"111"
6	ActFEV	End_cGA	RDY_IndGen="111" and RDY_PVC="111"
7	ActPVU	ActPVU	RDY_FEV='0'
8	ActPVU	ActIndGen	RDY_FEV='1'
9	End_cGA	End_cGA	

Tabla 5.2 Reglas de transición para la máquina de estados finitos del AGc.

La FSM contiene cinco estados que son *Start, ActIndGen, ActFEV, ActPVU* y *End_cGA* y son descritos a continuación:

1) **Estado Start:** En este estado se realiza la inicialización de algunos componentes, esto es, el vector de probabilidades se establece al valor medio del intervalo (32 para la versión con *CAPRNG*'s de 6 bits y 1073741823 para la versión con *PRNG*'s de 32 bits) y las semillas de los generadores de números pseudo-aleatorios son inicializadas. Una vez hecho esto, se pasa al siguiente estado, *ActIndGen*.

2) **Estado ActIndGen:** En este paso se activan todos los componentes *IndGen* para generar un par de individuos a la vez. Una vez generados estos individuos, el siguiente estado en la secuencia es *ActFEV*. Durante este estado es igualmente activado el componente *PVC*, el cual verifica la convergencia del vector de probabilidades, esto ocurre a partir de la segunda iteración del algoritmo.

3) **Estado ActFEV:** En este estado, el componente *FEv* es activado y en consecuencia se realiza la evaluación de cada individuo y se determina cual es mejor. Una vez finalizada la operación, se continúa con el estado *ActPVU*. Cuando el estado *ActFEv* es activado, el generador de números pseudo-aleatorios genera el siguiente número (para ambas versiones).

4) **Estado ActPVU:** Aquí se realiza la activación del componente *PVU*, es decir, se lleva a cabo la actualización del vector de probabilidades en un solo paso (en paralelo). Una vez finalizada esta actualización, se regresa al estado *ActIndgen* si el vector no ha convergido, en caso contrario, la secuencia finaliza moviéndose al estado *End_cGA*.

5) **Estado End_cGA:** Este es el estado final de la FSM, aquí se detiene el funcionamiento de todos los componentes y se activa una señal *RDY* que indica que el AGc ha terminado.

Es importante mencionar que se dispone de una señal de RESET, la cual regresa al algoritmo al estado **Start**, inicializando de nuevo el vector de probabilidades al valor medio del intervalo.

El componente que contiene al algoritmo genético compacto para un vector de probabilidades de tres elementos es mostrado en la Fig. 5.28.

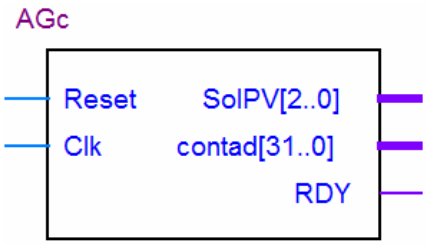


Fig. 5.28 Componente AGc.

En general, el diseño propuesto del AGc tiene las siguientes entradas: una señal de *Reset* que reinicia la operación del algoritmo y una señal de reloj *Clk* que sincroniza los estados del algoritmo. Como salidas se tienen el vector solución (vector de probabilidades compuesto solamente de ceros y unos), un contador *contad* que es un dato entero de 32 bits, el cual reporta el número de iteraciones alcanzadas por el algoritmo al momento de encontrar la solución (está limitado por el dato entero, si es sobrepasado la cuenta se reinicia, así que solo se agregó como dato puramente informativo) y una señal *RDY* que indica que el algoritmo ha finalizado. La Fig. 2.8 muestra el componente AGc en el caso de un vector de probabilidades de tres elementos. Si el vector de probabilidades contiene más elementos, se deben modificar los tamaños y el número de componentes que los procesan, es decir, escalar el diseño.

En los registros *seed* se deben tener valores diferentes para que se tengan diferentes secuencias en los generadores de números pseudo-aleatorios. Algo que mejoraría el desempeño del algoritmo sería incorporar una fuente externa de semillas para los generadores de números pseudo-aleatorios con el fin de tener diferentes resultados cada vez que se ejecute de nuevo el algoritmo sobre hardware.

La Fig. 5.29 muestra una parte del diagrama esquemático del diseño propuesto para el AGc con *PRNG*'s de 32 bits. Los componentes se caracterizan por tener señales de entrada *CE*, *CLK* y una señal que activa la operación del componente que sirven como señales de control para que la FSM pueda sincronizar los componentes.

La Fig. 5.30 muestra parte del diagrama esquemático del diseño de un AGc con *CAPRNG*'s de 6 bits. Las semillas son configuradas con las entradas *set1, set2, clear1* y *clear2*

durante la etapa inicial, luego estas entradas se ponen a cero para no reiniciar la secuencia de números pseudo-aleatorios. El resto del diseño es similar a la versión con *PRNG's* de 32 bits.

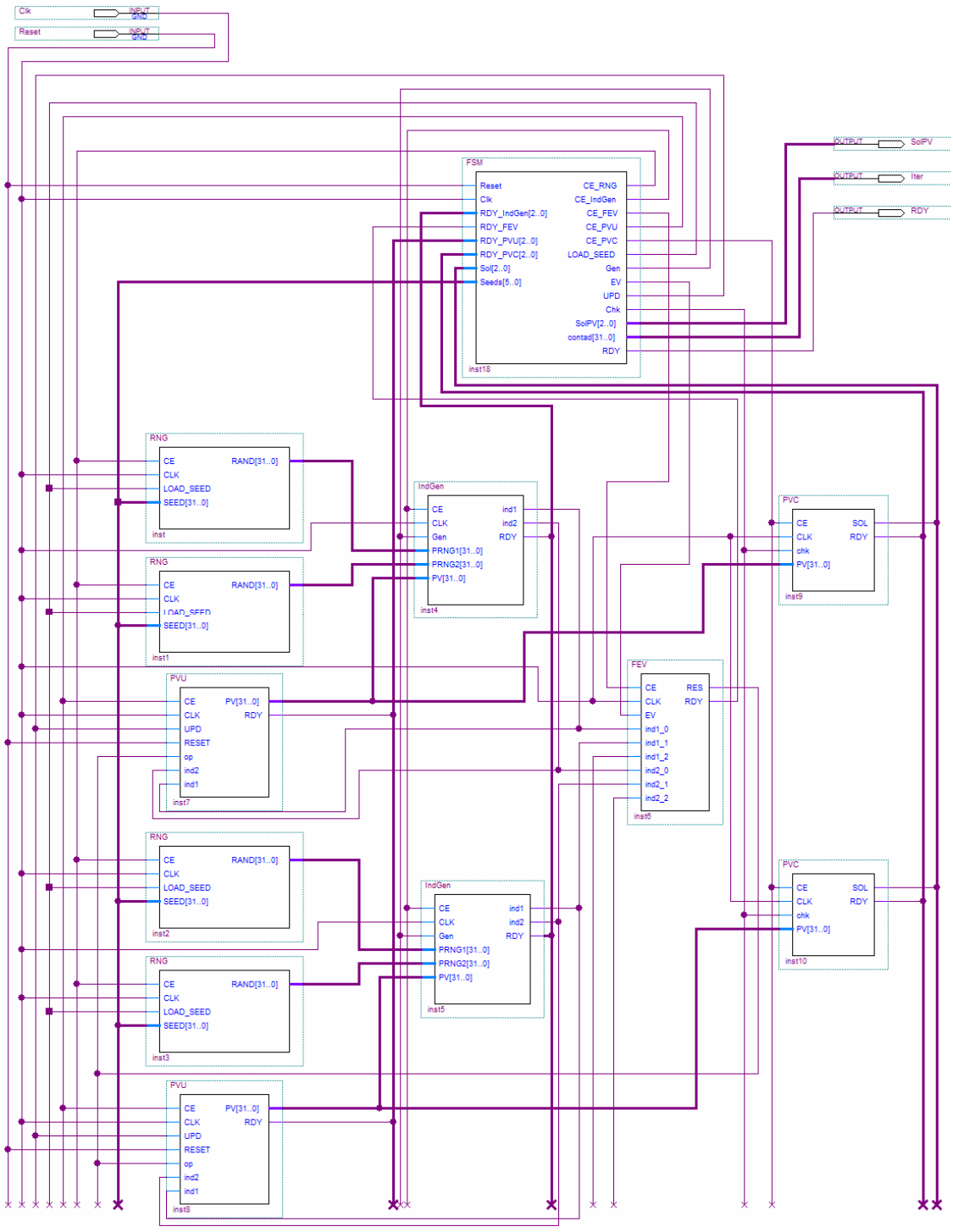


Fig. 5.29 Diagrama esquemático del diseño del AGc con PRNG's de 32 bits.

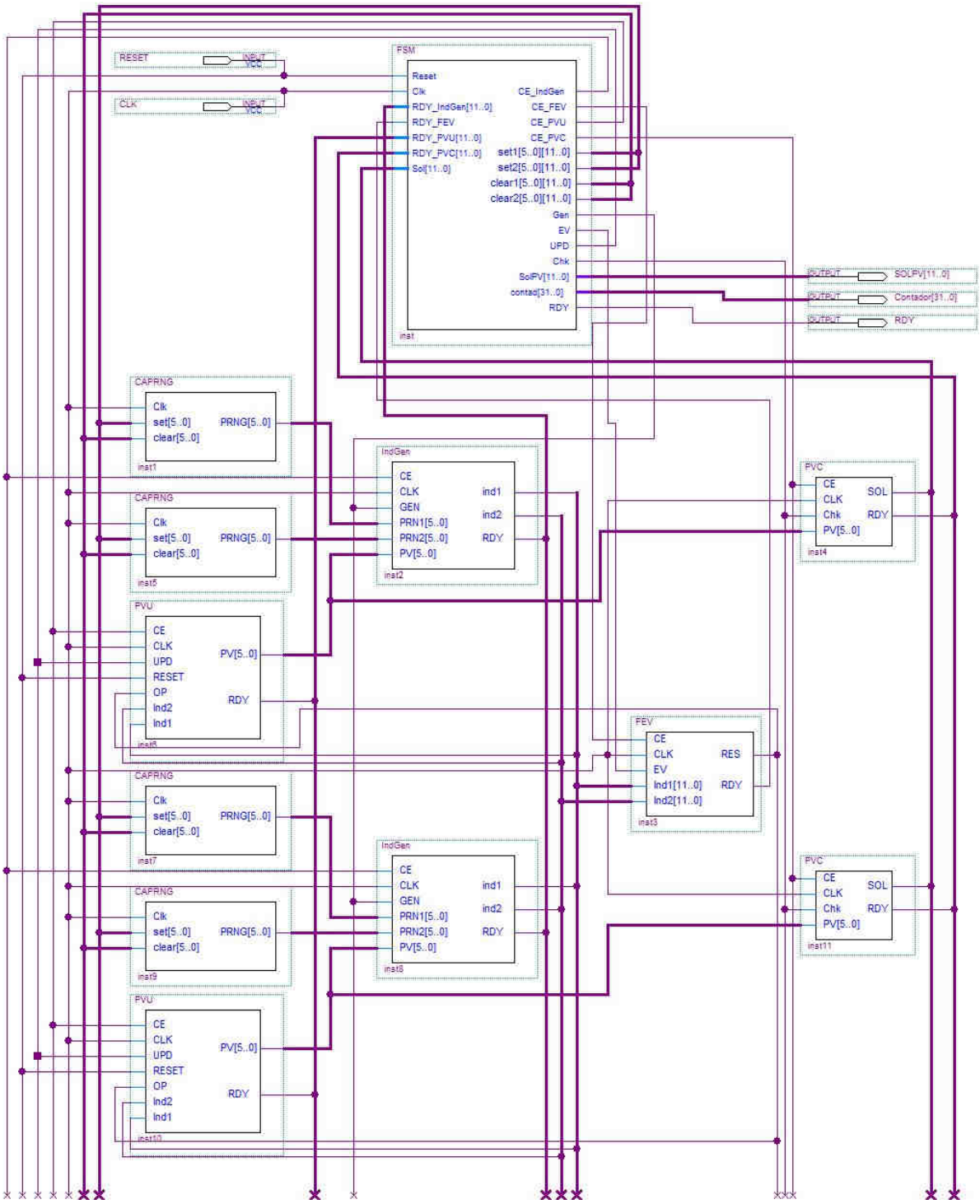


Fig. 5.30 Diagrama esquemático del diseño del AGc con CAPRNG's de 6 bits.

Arquitectura del Algoritmo Genético Compacto con Elitismo y Mutación

En el caso de la arquitectura propuesta para el **AGcEM**, solo se realizó el diseño con generadores de números pseudo-aleatorios basados en autómatas celulares. Este diseño difiere del AGc ya que incorpora elitismo y mutación. En la Fig. 5.31 se muestra parte del diagrama esquemático de un diseño del AGcEM con un vector de probabilidades de 32 bits.

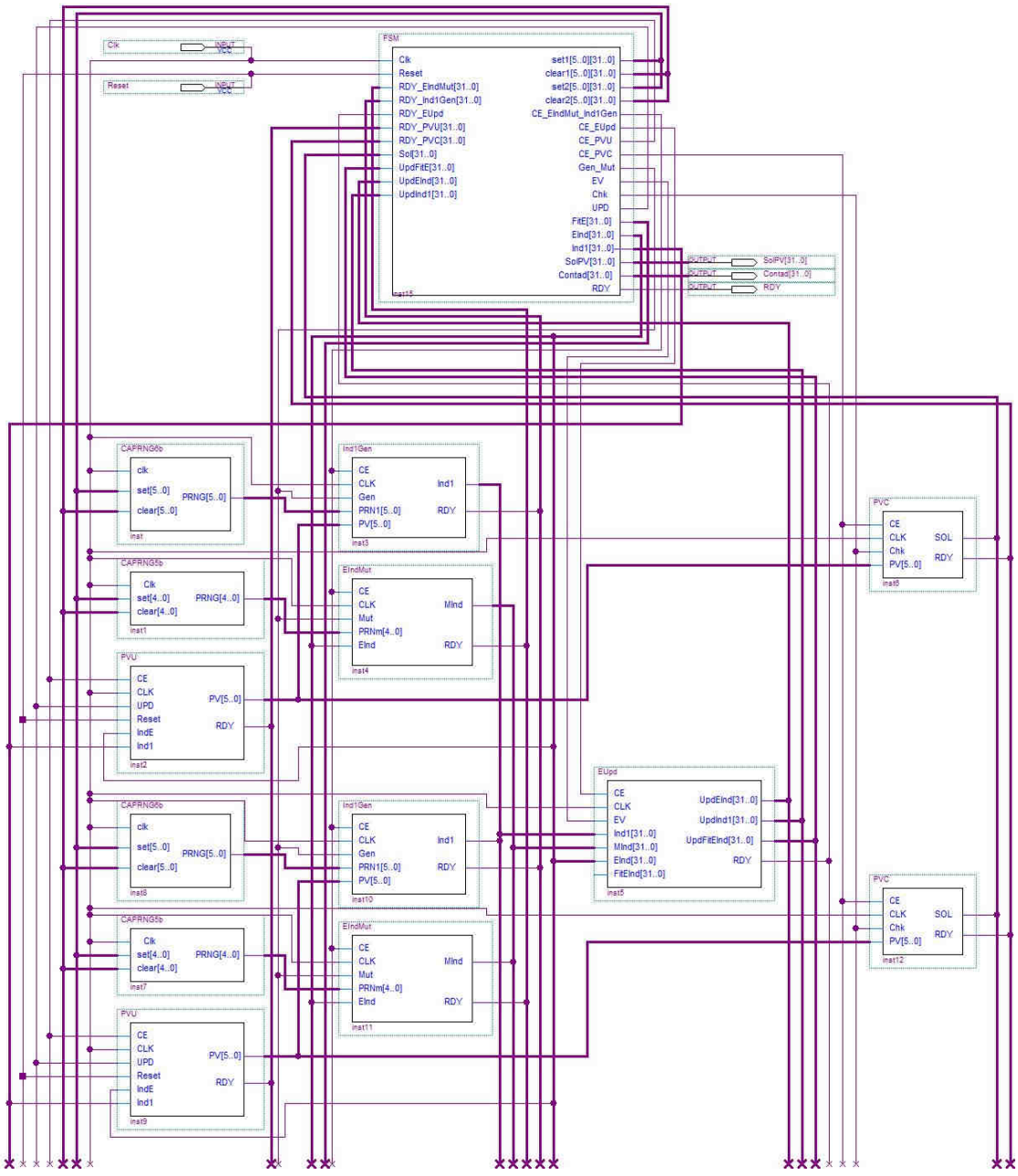


Fig. 5.31 Diagrama esquemático del diseño del AGcEM con CAPRNG's.

La arquitectura contiene componentes *CAPRNG6b* y *CAPRNG5b*, que son generadores de números pseudo-aleatorios de seis y cinco bits de longitud respectivamente. El número generado por el *CAPRNG6b* se utiliza para generar el individuo "1" en el componente *Ind1Gen*. El número generado por el *CAPRNG5b* se utiliza para decidir si se muta un bit del individuo elitista con una probabilidad de 1/31 ($1/2^5$-1), el cual es guardado en el registro *MInd*. El componente *EUpd* recibe el individuo 1 (*Ind1*), el individuo mutado (*MInd*), el individuo elitista (*EInd*) y la aptitud del individuo elitista (*FitEInd*), calcula la aptitud de *Ind1*y *MInd*, las compara con *FitEInd* y determina los nuevos valores para *EInd*, *Ind1* y *FitEInd*. El componente *PVU* realiza la actualización del vector de probabilidades en base al individuo ganador (que siempre será *IndE*), y este vector será inspeccionado por el componente *PVC* para verificar la convergencia del mismo.

La máquina de estados que sincroniza los componentes del AGcEM consta de cinco estados al igual que el AGc. La Fig. 5.32 muestra la FSM utilizada para el AGcEM y la Tabla 5.3 muestra las reglas de transición de dicha FSM.

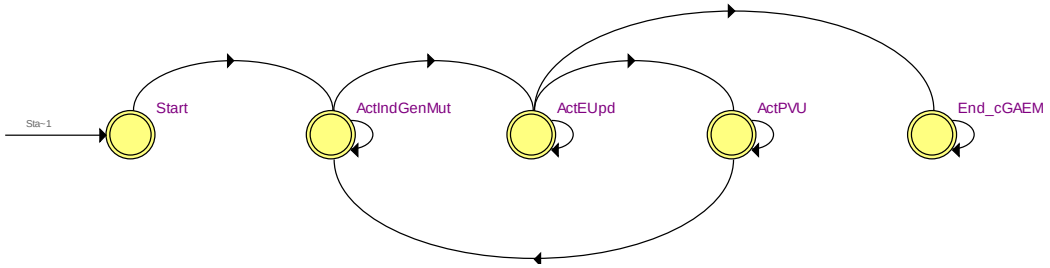


Fig. 5.32 Máquina de estados finitos para el AGcEM.

	Estado presente	Estado siguiente	Regla
1	Start	ActIndGenMut	
2	ActIndGenMut	ActIndGenMut	flag≠'0' and RDY_PVU≠"111.."
3	ActIndGenMut	ActEUpd	flag='0' or RDY_PVU="111.."
4	ActEUpd	ActEUpd	RDY_EIndMut≠"111.." and RDY_Ind1Gen≠"111.."
5	ActEUpd	ActPVU	RDY_Ind1Gen="111.." and RDY_PVC≠"111.." and RDY_EIndMut="111.."
6	ActEUpd	End_cGAEM	RDY_IndGen="111.." and RDY_PVC="111.." and RDY_EIndMut="111.."
7	ActPVU	ActPVU	RDY_EUpd='0'
8	ActPVU	ActIndGenMut	RDY_EUpd='1'
9	End_cGAEM	End_cGAEM	

Tabla 5.3 Reglas de transición para la máquina de estados finitos del AGcEM.

La FSM contiene cinco estados que son *Start, ActIndGenMut, ActEUpd, ActPVU* y *End_cGAEM* y son descritos a continuación:

1) **Estado Start:** En este estado se realiza la inicialización de algunos componentes, esto es, el vector de probabilidades se establece al valor medio del intervalo "32", las semillas de los generadores de números pseudo-aleatorios son establecidas (para el *CAPRNG5b* y *CAPRNG6b*) y se deben inicializar los registros que contienen el individuo elitista y su valor de aptitud. Una vez realizado esto, se pasa al siguiente estado, *ActIndGenMut*.

2) **Estado ActIndGenMut:** En este paso se activan todos los componentes *Ind1Gen* y *EIndMut* para generar el individuo uno y aplicar la mutación sobre el individuo elitista a la vez. Una vez finalizada su tarea, el siguiente estado en la secuencia es *ActEUpd*. Durante este estado es igualmente activado el componente *PVC*, el cual verifica la convergencia del vector de probabilidades, esto ocurre a partir de la segunda iteración del algoritmo.

3) **Estado ActEUpd:** En este estado, el componente *EUpd* es activado y en consecuencia se realiza la evaluación del individuo mutado y del individuo uno, se calcula la aptitud de cada uno y se compara con la aptitud del individuo elitista para determinar el nuevo individuo elitista y su aptitud, además del individuo uno. Una vez finalizada la operación, se continúa con el estado *ActPVU*. Cuando el estado *ActEpd* es activado, los generadores de números pseudo-aleatorios generan el siguiente número en su secuencia.

4) **Estado ActPVU:** Aquí se realiza la activación del componente *PVU*, es decir, se lleva a cabo la actualización del vector de probabilidades en un solo paso (en paralelo) con respecto al individuo elitista. Una vez finalizada esta actualización, se regresa al estado *ActIndGenMut* si el vector no ha convergido, en caso contrario, la secuencia finaliza moviéndose al estado *End_cGAEM*.

5) **Estado End_cGAEM:** Este es el estado final de la FSM, aquí se detiene el funcionamiento de todos los componentes y se activa una señal *RDY* que indica que el AGcEM ha terminado.

El componente AGcEM presenta a la salida el vector de probabilidades como un arreglo de bits (en lugar de un arreglo de números enteros), es decir, cada elemento con su correspondiente valor convergido. El componente es igual al mostrado en la Fig. 5.28, sólo que esta versión incluye elitismo y mutación.

Un aspecto a considerar para este diseño es que se debe inicializar el registro que contiene el individuo elitista y su valor de aptitud con algún valor inicial en la etapa de inicialización, ya que en este diseño no se propuso un estado para generarlo, ni para calcular su valor de aptitud. En consecuencia, se obtiene la misma cantidad de estados que el AGc y por tanto el AGcEM resulta ser igual de veloz.

5.4 Alcances y limitaciones

El diseño propuesto tiene algunas limitantes, así que, dependiendo del hardware se pueden encontrar algunas otras limitantes adicionales, por ejemplo, a pesar de que el algoritmo debe aprovechar la aleatoreidad de los PRNG's ó CAPRNG´s, esto puede ocasionar problemas dependiendo de la función objetivo a optimizar, ya que si las semillas son las mismas, el algoritmo arrojará el mismo resultado cada vez que inicie el cálculo o la búsqueda en el caso de una función objetivo estática. Una solución a esto es tener una fuente externa de semillas para proporcionar esa aleatoreidad, o en su caso disponer de una memoria con un arreglo de semillas en el mismo hardware. Otra opción sería crear la semilla a partir de un reloj como en los ordenadores. Esto podría ser un problema en algunos casos, sin embargo, si la evaluación de la función objetivo es dinámica (que cambia con el tiempo) y recibe un conjunto de parámetros externos por la misma razón, tal vez no sea necesario, ya que esta sería una fuente de aleatoreidad, y proporcionaría resultados diferentes.

Otra limitante es el problema a resolver, ya que se pueden encontrar funciones objetivo muy complejas, ocasionando un mayor consumo de recursos y agotando tal vez recursos necesarios para el resto del diseño (si esta propuesta formara parte de una aplicación de mayor escala).

Una ventaja de nuestro diseño, es que se presta para aplicaciones en tiempo real, ya que su ejecución resulta ser muy rápida (del orden de microsegundos en el caso del problema max-one), aunque esto podría ser afectado por el tipo de función objetivo a optimizar.

Debido al diseño modular, sus componentes se pueden reutilizar para la construcción de otros algoritmos de la misma naturaleza.

Podría considerarse una desventaja el hecho de que para el AGcEM se deban inicializar los registros del individuo elitista y su aptitud, sin embargo se gana en velocidad.

Capítulo 6

Experimentos y resultados

6.1 Diseño del conjunto de funciones de prueba

Para evaluar el desempeño del algoritmo genético compacto en hardware (AGc y AGcEM) propusimos cinco tipos de experimentos que requieren optimizar las funciones Max-One, Min-One, HammingInverso-1431655765, HammingInverso-858993459 y HammingInverso-477218588. Estos experimentos se detallan a continuación:

Problema Max-One

El problema Max-One consiste en la maximización de unos en una cadena binaria. Formalmente puede ser descrito como la búsqueda de una cadena $\vec{x} = \{x_1, x_2, ..., x_N\}$, con $x_i \in \{0,1\}$, que maximiza la siguiente ecuación:

$$F(\vec{x}) = \sum_{i=1}^{N} x_i \qquad (6.1)$$

En nuestro caso, la longitud de la cadena fué de 32, es decir, N=32. La gráfica de la función Max-One para cadenas de 32 bits es mostrada en la Fig. 6.2. En el eje de las abscisas se indica el valor decimal de la cadena binaria de 32 bits y en el eje de las ordenadas se muestra la cantidad de unos contenidos en la cadena binaria correspondiente. La gráfica muestra básicamente el espacio de búsqueda donde aplicamos el AGc. El óptimo global para esta función es 32 (sobre el eje de las ordenadas), es decir, todos los bits establecidos a uno sobre el eje de las abscisas (4294967295_{10}=" $11111111111111111111111111111111_2$").

Problema Min-One

El problema Min-One consiste en minimizar la cantidad de unos en una cadena binaria. Al igual que el Max-One, puede ser descrito como la búsqueda de una cadena $\vec{x} = \{x_1, x_2, ..., x_N\}$, con $x_i \in \{0,1\}$, que minimiza la ecuación 6.1.

La longitud de las cadenas binarias empleadas fue de 32 bits al igual que en el problema Max-One. La Fig. 6.3 muestra la gráfica de la función Min-One, donde el eje de las abscisas muestra el valor decimal de las cadenas binarias de 32 bits y el eje de las ordenadas indica el conteo de unos de la cadena correspondiente. El óptimo global para esta función es 0 sobre el eje de las ordenadas, y con todos los bits en ceros para el eje de las abscisas (0_{10}="$00000000000000000000000000000000_2$").

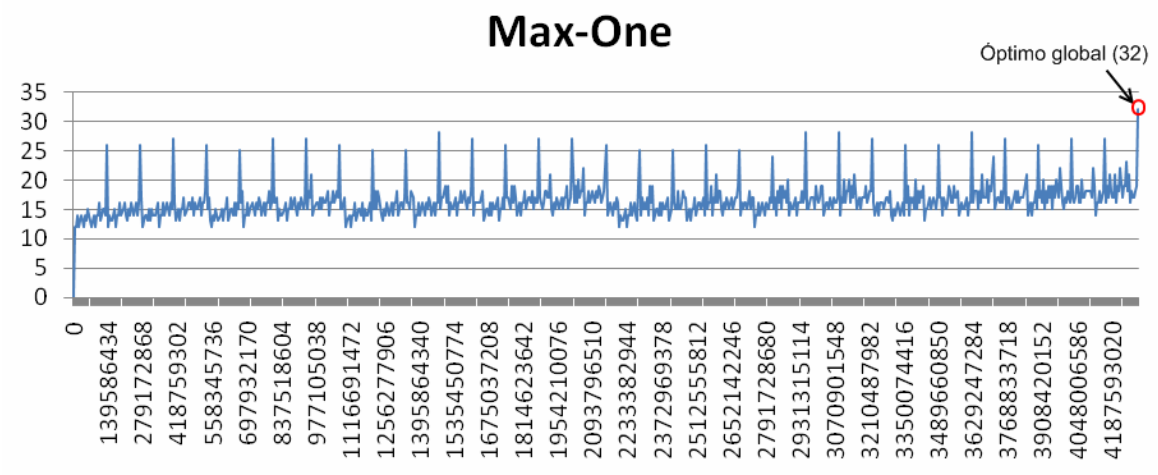


Fig. 6.1 Gráfica de la función Max-One.

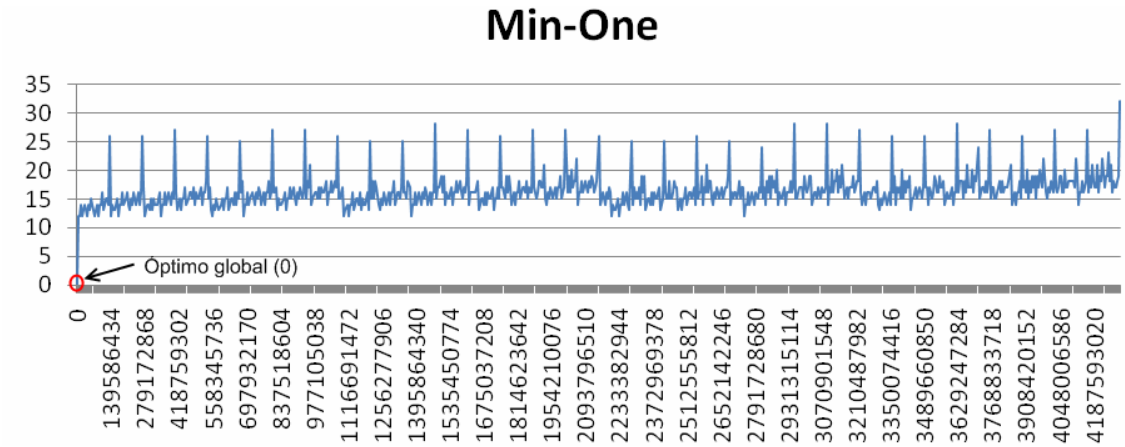


Fig. 6.2 Gráfica de la función Min-One.

Problema HammingInverso-1431655765

El problema HammingInverso-1431655765 consiste en maximizar el inverso de la distancia de Hamming de una cadena binaria de 32 bits a la cadena "01010101010101010101010101010101", cuyo valor decimal es 1431655765. La distancia de Hamming se define como el número de bits que tienen que cambiarse para transformar una palabra de código válida (cadena binaria) en otra palabra de código válida (cadena binaria), y la distancia de Hamming inversa se refiere al número de bits iguales en dos cadenas binarias. A continuación se muestran algunos ejemplos de distancia de Hamming entre cadenas binarias de 6 bits:

La distancia de Hamming entre "**1**0**0010**" y "**0**0**1101**" es 5.

La distancia de Hamming entre "**1**011**10**" y "**0**011**01**" es 3.

La distancia de Hamming entre "001101" y "001101" es 0.

En nuestro caso, el problema HammingInverso-1431655765 fue planteado como un problema de maximización, es decir, maximizamos la distancia inversa de Hamming. Así que la distancia inversa de Hamming para los ejemplos anteriores son:

La distancia inversa de Hamming entre "1**0**0010" y "0**0**1101" es 1.

La distancia inversa de Hamming entre "1**011**10" y "0**011**01" es 3.

La distancia inversa de Hamming entre "**001101**" y "**001101**" es 6.

La gráfica para el problema HammingInverso-1431655765 se muestra en la Fig. 6.3. El eje de las abscisas muestra el valor decimal de la cadena binaria de 32 bits y el eje de las ordenadas muestra la distancia inversa de Hamming de la cadena correspondiente a la cadena "01010101010101010101010101010101". El valor óptimo se encuentra en la cadena 1431655765_{10}="$01010101010101010101010101010101_2$" sobre el eje de las abscisas, con un valor de 32 sobre el eje de las ordenadas.

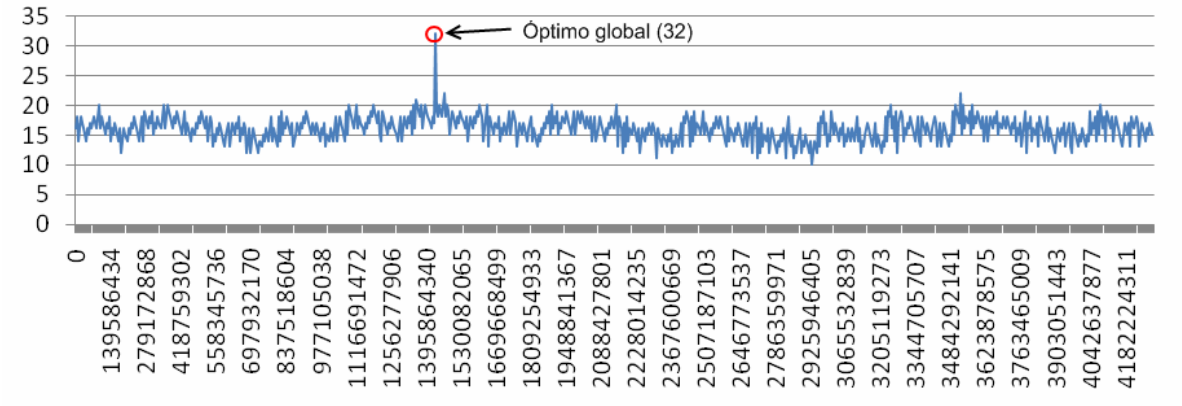


Fig. 6.3 Gráfica de la función HammingInverso-1431655765.

Problema HammingInverso-858993459

El problema Hamming-858993459 consiste en maximizar la distancia inversa de Hamming de una cadena binaria de 32 bits a la cadena "00110011001100110011001100110011", cuyo valor decimal es 858993459.

La gráfica para el problema Hamming-858993459 se muestra en la Fig. 6.4. El eje de las abscisas muestra el valor decimal de la cadena binaria de 32 bits y el eje de las ordenadas muestra la distancia inversa de Hamming de la cadena correspondiente a la cadena "00110011001100110011001100110011". El valor óptimo se encuentra en la cadena 858993459_{10}="$00110011001100110011001100110011_2$" sobre el eje de las abscisas, con un valor de 32 sobre el eje de las ordenadas.

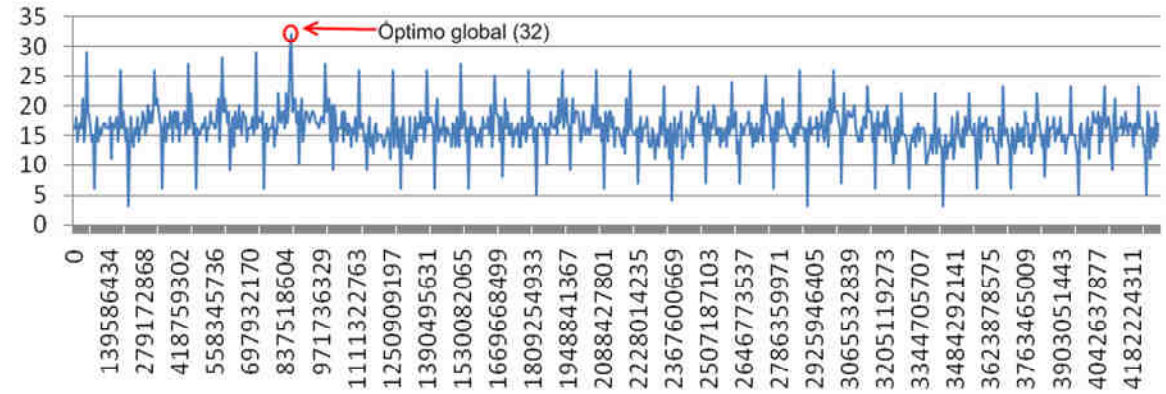


Fig. 6.4 Gráfica de la función HammingInverso-858993459.

Problema HammingInverso- 477218588

El problema HammingInverso-477218588 consiste en maximizar la distancia inversa de Hamming de una cadena binaria de 32 bits a la cadena "00011100011100011100011100011100", cuyo valor decimal es 477218588.

La gráfica para el problema HammingInverso- 477218588 se muestra en la Fig. 6.5. El eje de las abscisas muestra el valor decimal de la cadena binaria de 32 bits y el eje de las ordenadas muestra la distancia inversa de Hamming de la cadena correspondiente a la cadena "00011100011100011100011100011100". El valor óptimo se encuentra en la cadena 477218588_{10}="$00011100011100011100011100011100_2$" sobre el eje de las abscisas, con un valor de 32 sobre el eje de las ordenadas.

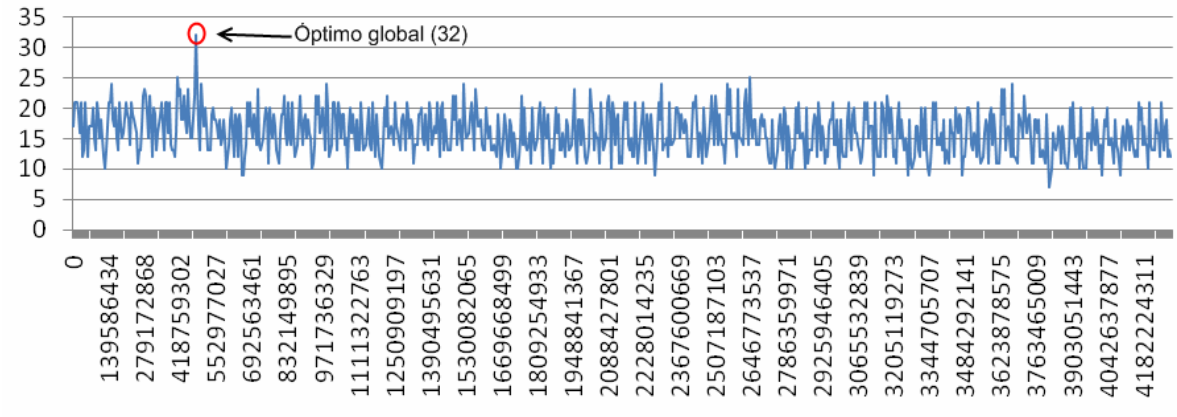


Fig. 6.5 Gráfica de la función HammingInverso-477218588.

Los problemas HammingInverso fueron propuestos por nosotros, y el resto son funciones utilizadas en otras propuestas de AGc's sobre hardware, las cuales servirán para comparar los diseños.

6.2 Eficiencia del algoritmo en hardware

Con el conjunto de experimentos propuesto anteriormente hemos realizado pruebas del AGc y el AGcEM en hardware con el *PRNG* basado en el generador estándar mínimo y con el *CAPRNG* en el caso del AGc y sólo con el *CAPRNG* para el AGcEM. Es importante mencionar que los experimentos fueron realizados en un **Cyclone II EP2C70F896C6 de Altera** con un reloj de **50 MHz**, y la herramienta utilizada para los diseños fue Quartus II de

Altera utilizando el lenguaje **VHDL**. Los resultados reportados en cuanto a tiempo de ejecución e iteraciones para el AGc y AGcEM sobre hardware son sólo de **simulación**. La Tabla 6.1 muestra los resultados obtenidos por el AGc para el problema max-one con vectores de probabilidades de 8 y 12 bits en sus dos versiones, es decir, con dos tipos de generadores de números pseudo-aleatorios.

Max-one		**AGc con PRNG basado en el generador estándar mínimo**	**AGc con CAPRNG**
8-bits	Tiempo de ejecución:	40.28μs	41.86μs
	Iteraciones:	335	348
	Funciones combinacionales:	18796/68416	496/68416
	Registros lógicos dedicados:	826/68416	226/68416
	Multiplicadores embebidos de 9-bits:	192/300	0/300
12-bits	Tiempo de ejecución:	41.48μs	43.06μs
	Iteraciones:	345	358
	Funciones combinacionales:	28149/68416	703/68416
	Registros lógicos dedicados:	1214/68416	314/68416
	Multiplicadores embebidos de 9-bits:	288/300	0/300

Tabla 6.1 Comparación de diseño del AGc para el problema max-one.

Los resultados muestran que la versión del AGc con el PRNG basado en el generador estándar mínimo converge más rápido (menos iteraciones), sin embargo, tiene un mayor consumo de recursos, debido a la naturaleza del PRNG, ya que consume demasiados multiplicadores embebidos, lo que limita su uso para el FPGA especificado anteriormente. Con respecto al consumo de recursos, el AGc con el CAPRNG resultó ser más económico en cuanto al consumo de recursos, esto debido a la naturaleza de los autómatas celulares. Este diseño tardó un poco más en converger, sin embargo el ahorro de recursos de hardware es enorme comparado con el primer diseño.

En la Tabla 6.2 se muestran los datos de síntesis de los generadores de números pseudo-aleatorios utilizados en la propuesta del AGc. En el caso del CAPRNG, los números generados son de 6 bits de longitud, por ellos requiere de 6 registros para almacenar cada bit (estados del autómata celular).

Componente		Datos de síntesis
PRNG basado en las reglas de Park & Miller	Funciones combinacionales: Registros lógicos dedicados: Multiplicadores embebidos de 9-bits:	1108/68416 826/68416 192/300
CAPRNG	Funciones combinacionales: Registros lógicos dedicados: Multiplicadores embebidos de 9-bits:	24/68416 6/68416 0/300

Tabla 6.2 Resultados de síntesis sobre el FPGA para los generadores de números pseudo-aleatorios.

Las Tablas 6.3-6.7 corresponden al conjunto de problemas planteados en el punto anterior sobre cadenas de 32 bits. En las tablas se reportan los datos de síntesis para la versión en hardware del **AGc**, así como las iteraciones y el tiempo consumido tanto para la versión en hardware como la de software.

Los resultados de software corresponden al **AGc** programado en C++, presentando los resultados de un promedio de 20 ejecuciones del algoritmo (AGc) sobre el CPU especificado. En el caso de la versión en hardware se presenta el resultado de solo una ejecución. Los resultados para cada problema se muestran a continuación:

Problema Max-One

Hardware			Software	
Total de elementos lógicos	1,785/68,416	3%	*Versión del AGc programado en C++ y ejecutado en un ordenador con procesador AMD Athlon(tm) 64 X2 Dual Core 4000+ a 2.09 GHz con 1 GB de RAM con S.O. Ubuntu 8.04*	
- Total de funciones combinacionales	1,764/68,416	3%		
- Registros lógicos dedicados	754/68,416	1%		
Total de registros	754			
Multiplicadores de 9-bits embebidos	0/300	0%		
Iteraciones	697		*Iteraciones (promedio)*	605.15
Tiempo de ejecución (simulación)	83.72 μs		*Tiempo de ejecución (promedio)*	4.2 ms

Tabla 6.3 Comparación del AGc en hardware sobre la versión en software para el problema max-one.

Problema Min-One

Hardware			Software	
Total de elementos lógicos	1,785/68,416	3%	*Versión del AGc programado en C++ y ejecutado en un ordenador con procesador AMD Athlon(tm) 64 X2 Dual Core 4000+ a 2.09 GHz con 1 GB de RAM con S.O. Ubuntu 8.04*	
- Total de funciones combinacionales	1,764/68,416	3%		
- Registros lógicos dedicados	754/68,416	1%		
Total de registros	754			
Multiplicadores de 9-bits embebidos	0/300	0%		
Iteraciones	851		*Iteraciones (promedio)*	865.5
Tiempo de ejecución (simulación)	102.2 μs		*Tiempo de ejecución (promedio)*	4.9 ms

Tabla 6.4 Comparación del AGc en hardware sobre la versión en software para el problema min-one.

Problema HammingInverso-1431655765

Hardware			Software	
Total de elementos lógicos	1,786/68,416	3%	*Versión del AGc programado en C++ y ejecutado en un ordenador con procesador AMD Athlon(tm) 64 X2 Dual Core 4000+ a 2.09 GHz con 1 GB de RAM con S.O. Ubuntu 8.04*	
- Total de funciones combinacionales	1,764/68,416	3%		
- Registros lógicos dedicados	754/68,416	1%		
Total de registros	754			
Multiplicadores de 9-bits embebidos	0/300	0%		
Iteraciones	653		*Iteraciones (promedio)*	626.45
Tiempo de ejecución (simulación)	78.45 μs		*Tiempo de ejecución (promedio)*	4.45 ms

Tabla 6.5 Comparación del AGc en hardware sobre la versión en software para el problema HammingInverso-1431655765.

Problema HammingInverso-858993459

Hardware			Software	
Total de elementos lógicos	1,785/68,416	3%	*Versión del AGc programado en C++ y ejecutado en un ordenador con procesador AMD Athlon(tm) 64 X2 Dual Core 4000+ a 2.09 GHz con 1 GB de RAM con S.O. Ubuntu 8.04*	
- Total de funciones combinacionales	1,764/68,416	3%		
- Registros lógicos dedicados	754/68,416	1%		
Total de registros	754			
Multiplicadores de 9-bits embebidos	0/300	0%		
Iteraciones	718		*Iteraciones (promedio)*	645.4
Tiempo de ejecución (simulación)	86.24 μs		*Tiempo de ejecución (promedio)*	4.55 ms

Tabla 6.6 Comparación del AGc en hardware sobre la versión en software para el problema HammingInverso-858993459.

Problema HammingInverso- 477218588

Hardware			Software	
Total de elementos lógicos	1,785/68,416	3%	*Versión del AGc programado en C++ y ejecutado en un ordenador con procesador AMD Athlon(tm) 64 X2 Dual Core 4000+ a 2.09 GHz con 1 GB de RAM con S.O. Ubuntu 8.04*	
- Total de funciones combinacionales	1,764/68,416	3%		
- Registros lógicos dedicados	754/68,416	1%		
Total de registros	754			
Multiplicadores de 9-bits embebidos	0/300	0%		
Iteraciones	698		*Iteraciones (promedio)*	639.75
Tiempo de ejecución (simulación)	83.84 μs		*Tiempo de ejecución (promedio)*	4.4 ms

Tabla 6.7 Comparación del AGc en hardware sobre la versión en software para el problema HammingInverso-477218588.

En el caso del **AGcEM**, se realizaron los mismos experimentos sobre cadenas de 32 bits. Las Tablas 6.8-6.12 muestran los resultados obtenidos para la versión en hardware y software del AGcEM.

Problema Max-One

Hardware			Software	
Total de elementos lógicos	1,543/68,416	2%	*Versión del AGc programado en C++ y ejecutado en un ordenador con procesador AMD Athlon(tm) 64 X2 Dual Core 4000+ a 2.09 GHz con 1 GB de RAM con S.O. Ubuntu 8.04*	
- Total de funciones combinacionales	1,541/68,416	2%		
- Registros lógicos dedicados	791/68,416	1%		
Total de registros	791			
Multiplicadores de 9-bits embebidos	0/300	0%		
Iteraciones	358		*Iteraciones (promedio)*	570.95
Tiempo de ejecución (simulación)	43.04 μs		*Tiempo de ejecución (promedio)*	4.2 ms

Tabla 6.8 Comparación del AGcEM en hardware sobre la versión en software para el problema max-one.

Problema Min-One

Hardware			Software	
Total de elementos lógicos	1,543/68,416	2%	*Versión del AGc programado en C++ y ejecutado en un ordenador con procesador AMD Athlon(tm) 64 X2 Dual Core 4000+ a 2.09 GHz con 1 GB de RAM con S.O. Ubuntu 8.04*	
- Total de funciones combinacionales	1,541/68,416	2%		
- Registros lógicos dedicados	791/68,416	1%		
Total de registros	791			
Multiplicadores de 9-bits embebidos	0/300	0%		
Iteraciones	252		*Iteraciones (promedio)*	475.85
Tiempo de ejecución (simulación)	30.33 μs		*Tiempo de ejecución (promedio)*	3.95 ms

Tabla 6.9 Comparación del AGcEM en hardware sobre la versión en software para el problema min-one.

Problema HammingInverso-1431655765

Hardware			Software	
Total de elementos lógicos	1,543/68,416	2%	*Versión del AGc programado en C++ y ejecutado en un ordenador con procesador AMD Athlon(tm) 64 X2 Dual Core 4000+ a 2.09 GHz con 1 GB de RAM con S.O. Ubuntu 8.04*	
- Total de funciones combinacionales	1,541/68,416	2%		
- Registros lógicos dedicados	791/68,416	1%		
Total de registros	791			
Multiplicadores de 9-bits embebidos	0/300	0%		
Iteraciones	386		*Iteraciones (promedio)*	481.4
Tiempo de ejecución (simulación)	46.4 μs		*Tiempo de ejecución (promedio)*	3.7 ms

Tabla 6.10 Comparación del AGcEM en hardware sobre la versión en software para el problema HammingInverso-1431655765.

Problema HammingInverso-858993459

Hardware			Software	
Total de elementos lógicos	1,543/68,416	2%	*Versión del AGc programado en C++ y ejecutado en un ordenador con procesador AMD Athlon(tm) 64 X2 Dual Core 4000+ a 2.09 GHz con 1 GB de RAM con S.O. Ubuntu 8.04*	
- Total de funciones combinacionales	1,541/68,416	2%		
- Registros lógicos dedicados	791/68,416	1%		
Total de registros	791			
Multiplicadores de 9-bits embebidos	0/300	0%		
Iteraciones	310		*Iteraciones (promedio)*	509.8
Tiempo de ejecución (simulación)	37.28 μs		*Tiempo de ejecución (promedio)*	4.1 ms

Tabla 6.11 Comparación del AGcEM en hardware sobre la versión en software para el problema HammingInverso-858993459.

Problema HammingInverso- 477218588

Hardware			Software	
Total de elementos lógicos	1,542/68,416	2%	*Versión del AGc programado en C++ y ejecutado en un ordenador con procesador AMD Athlon(tm) 64 X2 Dual Core 4000+ a 2.09 GHz con 1 GB de RAM con S.O. Ubuntu 8.04*	
- Total de funciones combinacionales	1,541/68,416	2%		
- Registros lógicos dedicados	791/68,416	1%		
Total de registros	791			
Multiplicadores de 9-bits embebidos	0/300	0%		
Iteraciones	377		*Iteraciones (promedio)*	502.1
Tiempo de ejecución (simulación)	45.32 μs		*Tiempo de ejecución (promedio)*	4.25 ms

Tabla 6.12 Comparación del AGcEM en hardware sobre la versión en software para el problema HammingInverso-477218588.

Es importante mencionar que para todos los casos, el AGc y el AGcEM encontraron el óptimo global correspondiente al problema.

Para que el AGc o el AGcEM ejecuten una iteración ó generación requieren de seis ciclos de reloj. El número de generaciones ejecutadas en hardware es calculado con la siguiente expresión:

$$Generaciones\ por\ segundo = \frac{frecuencia\ de\ operación\ (Hz)}{ciclos\ de\ reloj\ consumidos\ para\ una\ generación} \qquad (6.2)$$

Calculando las generaciones por segundo con la expresión 6.2 tenemos:

$$Generaciones\ por\ segundo = \frac{50*10^6\ Hz}{6} = 8.33\ millones\ de\ generaciones\ por\ segundo$$

Es importante notar que el AGc en hardware propuesto en [6] realiza **6.67 millones** de generaciones por segundo y nuestra propuesta realiza **8.33 millones** de generaciones por segundo, por tanto, nuestro diseño es superior en la cantidad de generaciones ejecutadas en un segundo. Este dato corresponde a nuestros algoritmos AGc y el AGcEM en hardware propuestos.

En la Tabla 6.13 se muestra una tabla comparativa entre el AGc y el AGcEM con *CAPRNG*'s en el caso del problema max-one.

AGc			***AGcEM***		
Total de elementos lógicos	1,785/68,416	3%	*Total de elementos lógicos*	1,543/68,416	2%
- Total de funciones combinacionales	1,764/68,416	3%	*- Total de funciones combinacionales*	1,541/68,416	2%
- Registros lógicos dedicados	754/68,416	1%	*- Registros lógicos dedicados*	791/68,416	1%
Total de registros	754		*Total de registros*	791	
Multiplicadores de 9-bits embebidos	0/300	0%	*Multiplicadores de 9-bits embebidos*	0/300	0%
Iteraciones	*697*		*Iteraciones*	*358*	
Tiempo de ejecución (simulación)	*83.72 μs*		*Tiempo de ejecución (simulación)*	*43.04 μs*	

Tabla 6.13 Comparación del AGc y el AGcEM en hardware para el problema max-one de 32 bits.

De acuerdo a la Tabla 6.13 y a los resultados reportados anteriormente para ambas versiones, se puede apreciar que el AGcEM requiere menos recursos que el AGc. Otra característica importante es que la convergencia se alcanza en un menor número de iteraciones.

Capítulo 7

Conclusiones y trabajo futuro

7.1 Conclusiones

De acuerdo a los resultados obtenidos, podemos observar que un AGc o AGcEM implementado en un FPGA es muy rápido, del orden de microsegundos para el caso de los problemas abordados, ocupando a lo más un 3% de los recursos del FPGA (utilizando un Cyclone II EP2C70F896C6 de Altera con un reloj de 50 MHz) en el caso del AGc y máximo un 2% para el AGcEM.

Esta velocidad es relativa porque depende del tipo de función objetivo a optimizar, es decir, es posible que debido a la complejidad de la función se requieran de varios ciclos de reloj para ejecutarla, ocasionando que la ejecución total del algoritmo tome más tiempo. Una forma de compensar un poco esta desventaja sería diseñar dos bloques de la función objetivo para realizar la evaluación de cada individuo en paralelo. Otra alternativa sería analizar la función objetivo y tratar de paralelizarla de acuerdo a las operaciones que contenga, sin embargo esto depende de la habilidad del diseñador.

Una desventaja de nuestros diseños es que se tienen registros para las semillas de los generadores de números pseudo-aleatorios con valores fijos, esto provoca que cuando se reinicialice el algoritmo, se tengan las mismas secuencias y por ende el algoritmo arroje la misma salida. Una forma de solucionar esto sería incorporar una fuente externa de semillas, contar con un arreglo de semillas dentro del propio hardware, o generarlas en base a un reloj como en los ordenadores.

En el caso de tener una función objetivo dinámica, tal vez no sea necesario incorporar una fuente externa de semillas, debido a que este tipo de funciones reciben parámetros externos que de cierta forma introducen aleatoreidad, y por tanto los resultados variaran aunque se reinicialice el algoritmo.

Otro aspecto importante es que estas propuestas no están diseñadas para sustituir a los AG's, sino como una alternativa en aquellas aplicaciones donde existan restricciones de hardware, por ejemplo, en el consumo de recursos o en aquellas aplicaciones de tiempo real.

Debido a las velocidades que se pueden alcanzar con las propuestas, es posible su implementación en aplicaciones de tiempo real y además por su mínimo consumo de recursos, son adecuadas para formar parte de aplicaciones de mayor escala, por ejemplo, la incorporación de un módulo o componente de este tipo de algoritmos en alguna aplicación de hardware evolutivo, ya nuestro diseño es modular y está capacitado para formar parte de otro módulo en un nivel superior, ventaja que proporcionan los lenguajes de descripción de hardware como VHDL.

Los módulos diseñados que forman parte del algoritmo genético compacto en cualquiera de sus versiones pueden ser de utilidad en el diseño de otros tipos de algoritmos bio-inspirados, incrementando la cantidad de alternativas para implementar motores de búsqueda en hardware.

Para finalizar, en el presente trabajo planteamos el diseño en hardware del algoritmo genético compacto y su versión con elitismo y mutación, siendo esta última una opción muy competitiva para realizar optimización en tiempo real. Los resultados demuestran que el AGcEM converge más rápidamente que el AGc, y pruebas realizadas en [8] demuestran que este algoritmo proporciona soluciones de mejor calidad que el AGc.

7.2 Trabajo futuro

Existe mucho trabajo por hacer en lo que respecta a algoritmos bio-inpirados en hardware. Como trabajo futuro se propone la aplicación de las propuestas en algún área de la ingeniería para probar su efectividad en problemas reales.

También se propone realizar el diseño del algoritmo genético compacto extendido, el cual es una versión un poco más compleja que las mencionadas en este trabajo, sin embargo, pueden utilizarse algunos módulos diseñados en nuestras propuestas para la realización del diseño de este algoritmo.

Otra propuesta interesante, es realizar el diseño de un conjunto de módulos que nos permitan implementar un conjunto de funciones objetivo que requieran de cálculos en punto flotante. Esto con el fin de que podamos implementar la propuesta más rápidamente sin preocuparnos del tipo de problema, ya que la función objetivo puede variar demasiado, complicándose un poco su diseño.

Otro aporte interesante sería implementar algoritmos genéticos compactos con diferente cardinalidad a la utilizada en nuestras propuestas. Esto podría tener efectos positivos para ciertos tipos de problemas, es decir, al cambiar la cardinalidad del algoritmo, este podría tener mejor desempeño en la búsqueda la solución.

Convendría realizar un análisis del impacto del tipo de generadores de números pseudo-aleatorios utilizados en la propuesta, ya que tienen características particulares que los hacen muy diferentes a los que tenemos disponibles para la programación en software. Por ejemplo, los generadores de números pseudos-aleatorios empleados tienen un periodo de secuencia algo limitado, es decir, si el CAPRNG es de 6 bits, el periodo de la secuencia será de 2^6-1, lo cual es diferente en la programación en software.

Referencias

[1] J. I. Hidalgo Pérez y C. Cervigón Rückauer. *Una revisión de los algoritmos evolutivos y sus aplicaciones*. Madrid, España: [s. n.], 2002.

[2] A. E. Eiben y J. E. Smith. *Introduction to Evolutionary Computing*. Natural Computing Series, Springer, 2003.

[3] Carlos A. Coello Coello. *Introducción a la Computación Evolutiva (Notas de Curso)*. México, D.F., 2006.

[4] G. Harik. *Linkage Learning via Probabilistic Modeling in the ECGA*. IlliGAL Technical Report 99010, Illinois University, 1999.

[5] F. Cupertino, E. Mininno, E. Lino y D. Naso. *Optimization of Position Control of Induction Motors using Compact Genetic Algorithms*, IEEE Industrial Electronics, IECON 2006-32nd Annual Conference on, París, Francia, 2006, pp. 55-60.

[6] C. Aporntewan and P. Chongstitvatana. *A Hardware Implementation of the Compact Genetic Algorithm*. Proc. 2001 IEEE Congress Evolutionary Computation, Seoul, Korea, 2001, pp. 624–629.

[7] G. Harik, F. G. Lobo, and D. E. Goldberg. *The compact genetic algorithm*. IEEE Trans. Evolutionary Computation, Vol. 3, No. 4, pp. 287-297, Nov. 1999.

[8] J. C. Gallagher, S. Vigraham y G. Kramer. *A family of compact genetic algorithms for intrinsic evolvable hardware*. IEEE Trans. Evolutionary Computation, Vol. 8, No. 2, 2004.

[9] F. Cupertino, E. Mininno y D. Naso. *Elitist Compact Genetic Algorithms for Induction Motor Self-tuning Control*. Evolutionary Computation, CEC 2006, IEEE Congress on, Vancouver, BC, 2006, pp. 3057-3063.

[10] M. Pelikan, D. Goldberg y F. Lobo. *A Survey of Optimization by Building and Using Probabilistic Models*. IlliGAL Technical Report 99018, Illinois University, 1999.

[11] R. Rastegar y A. Hariri. *A Step Forward in Studying the Compact Genetic Algorithm.* Evolutionary Computation, MIT Press, USA, Vol. 14, No. 3, 2006, pp. 277-289.

[12] H. G. Beyer y H. P. Schwefel. *Evolution strategies. A comprehensive introduction.* Natural Computing: an international journal, Kluwer Academic Publishers, vol. 1, núm. 1, 2002, pp. 3-52.

[13] M. Chen. *Second Generation Particle Swarm Optimization.* Evolutionary Computation, CEC 2008 (IEEE World Congress on Computational Intelligence), Hong Kong, 2008, pp. 90-96.

[14] J. M. García Nieto. *Algoritmos basados en cúmulos de partículas para la resolución de problemas complejos.* Tesis de Licenciatura de la Universidad de Málaga, España, Septiembre 2006.

[15] J. Kennedy, R. Eberhart y Y. Shi. *Swarm Intelligence.* Morgan Kaufmann Publishers, EUA, 2001.

[16] E. K. Burke y G. Kendall. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques.* Springer, 2005.

[17] S. Jagannath Nanda. *Artificial immune systems: principle, algorithms and applications.* Tesis de Maestría del Instituto Nacional de Tecnología de Rourkela, India, 2008.

[18] L. N. de Castro y J. Timmis. *Artificial Immune Systems: A Novel Paradigm to Pattern Recognition.* In Artificial Neural Networks in Pattern Recognition, SOCO 2002, University of Paisley, UK, 2002, pp. 67-84.

[19] M. Dorigo y L. M. Gambardella. *Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem.* IEEE Transactions on Evolutionary Computation, Vol. 1, No. 1, 1996, pp. 55-36.

[20] T. Stützle y H. H. Hoos. *MAX-MIN Ant System.* ELSEVIER Science, 1999.

[21] E. Mezura, J. Velázquez y C. A. Coello. *A comparative study of differential evolution variants for global optimization.* In Proc. of the 8th annual conference on GECCO, USA, 2006 pp. 485-492.

[22] C. Maxfield. *The Design Warrior's Guide to FPGAs. Devices, Tools and Flows.* ELSEVIER, E.U.A., 2004.

[23] Altera Corporation. *Cyclone II Device Handbook Vol. 1*. 2007

[24] R. J. Tocci y N. S. Widmer. *Sistemas digitales, principios y aplicaciones.* Pearson Education, Octava edición, 2003, México.

[25] F. Pardo y J. A. Boluda. *VHDL, lenguaje para síntesis y modelado de circuitos.* Alfaomega, segunda edición, 2004, México.

[26] T. Higuchi, Y. Liu, M. Iwata y X. Yao. *Introduction to evolvable hardware.* Evolvable hardware. Genetic and evolutionary computation, Series editors: D. E. Goldberg and J. R. Koza. Springer 1999.

[27] B. Shackleford et al. *A high-performance, pipelined, FPGA-based Genetic Algorithm machine.* Genetic Programming and Evolvable Machines, Kluwer Academic Publishers. Vol. 2, Núm. 1, 2001, pp. 33-60.

[28] J. J. Kim y D. J. Chung. *Implementation of Genetic Algorithm Based on Hardware Optimization*. TENCON 99, Proceedings of the IEEE Region 10 Conference, South Korea, Vol. 2, 1999, pp. 1490-1493.

[29] S. D. Scott, S. Seth y A. Samal. *A Synthesizable VHDL Coding of a Genetic Algorithm*. Reporte técnico UNL-CSE-97-009 Universidad de Nebraska-Lincoln, 1997.

[30] I. Kajitani, M. Iwata y T. Higuchi. *A GA hardware engine and its applications.* . Evolvable hardware. Genetic and evolutionary computation, Series editors: D. E. Goldberg and J. R. Koza. Springer 1999.

[31] F. Cupertino et al. *Optimization of Position Control of Induction Motors using Compact Genetic Algorithms*. IEEE Industrial Electronics, IECON 2006 - 32nd Annual Conference on, Paris, 2006, pp. 55-60.

[32] W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1992.

[33] P. D. Hortensius, R. D. McLeod and H. C. Card, *Parallel Random Number Generation for VLSI Systems Using Cellular Automata*, IEEE Transactions on Computers, Vol. 38, Num. 10, Octubre 1989, pp. 1466-1473.

[34] L. N. de Castro, *Fundamentals of natural computing: an overview*, Science Direct, ELSEVIER 2006.