



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

---

FACULTAD DE ESTUDIOS SUPERIORES  
ACATLÁN

CARDIAC: La evolución hacia  
un modelo concurrente y  
paralelo

TESIS

PARA OBTENER EL TÍTULO DE

MATEMÁTICAS APLICADAS Y COMPUTACIÓN

PRESENTA

**MARTÍN OSVALDO SANTOS SOTO**

ASESOR DE LA TESIS:

DR. JORGE VASCONSELOS

Santa Cruz Acatlán, Naucalpan de Juarez 01/03/2023

F  
E  
S  
UNAM  
**ACATLÁN**

*“These are fast-moving times, and those who make no effort to understand computers may very well get left behind”*

- David W. Hagelbarger, 1968

# Agradecimientos

# Abstract

# Introducción

Hoy en día tenemos multitud de aparatos electrónicos que suelen ser llamados computadoras; celulares, laptops, tabletas, relojes inteligentes, y un sin fin más. Estos aparatos suelen ser llamados así dado que son capaces de resolver operaciones aritméticas y lógicas, guardar datos, procesarlos, y recibir instrucciones del usuario, esto siguiendo la definición que recoge el diccionario de Oxford<sup>1</sup> tiene sentido, pero si analizamos más detalladamente el termino “computadora” notaremos que el origen de la palabra es anterior a las computadoras tal cual las conocemos hoy día, por ejemplo antes de 1950 la palabra “computadora” era referida comúnmente a mujeres que trabajaban en la NASA y realizaban cálculos manuales, como lo describe el blog (NASA, 2020) de la NASA dedicado a Katherine Johnson, destacada computadora humana"que aún con el machismo que se vivía en la época logró ser ampliamente reconocida y ver como esa misma palabra que la había descrito a ella en el pasado era usada para describir a potentes máquinas electrónicas de cálculo.

Si viajamos al pasado remoto notaremos que desde muy temprana edad se buscaba disminuir las tareas repetitivas para los humanos, pasando por el ábaco en muchas culturas, hasta máquinas más complejas a partir del siglo XVI, o con técnicas como los logaritmos o las tablas de multiplicar. Esta búsqueda junto al desarrollo de otras tecnologías en paralelo permitió la creación de máquinas que podían ir más allá que automatizar cálculos, máquinas de propósito general, que pudieran cambiar su funcionamiento de acuerdo a la interacción con el usuario y resolver una multitud de problemas nunca antes pensada(Ifrah, 2001). En tal estudio del pasado nos daremos cuenta de la cantidad de sucesos y desarrollos que tuvieron que acontecer para llegar desde automatizaciones muy particulares de cálculos aritméticos hasta las computadoras que conocemos hoy en día.

---

<sup>1</sup>Usado como fuente dado que la lengua franca de la computación es el inglés.

Aunque tenemos una remota idea de lo compleja que fue la invención de la computadora no se suele conocer ni siquiera superficialmente la forma en que funciona una computadora, el como podemos enviar mensajes de texto mientras escuchamos una canción, si acaso hay algún programa que ejecuta a los demás programas que utilizamos, o incluso si hay un programa que da inicio a todos los procesos de una computadora. Por que al final son temas complejos que requieren su tiempo de estudio, lo que no puede suceder es que profesionales o estudiantes de carreras afines a la computación no conozcan estos detalles, y es que muchas veces se da por sentado que es conocimiento general y en especializaciones como desarrollo web o arquitectura de software(solo como ejemplo) no se le suele prestar mucha atención. Sin embargo el conocimiento, al menos básico, del funcionamiento interno de una computadora es fundamental para saber como explotar de mejor manera los recursos de las máquinas que estás utilizando en cualquier disciplina que te desempeñes. Aunado a esto conocer la historia es otro punto fundamental, te permitirá saber la razón por la cuál ciertos aspectos de las computadoras no han cambiado en 80 años y el por que otros han cambiado o evolucionado de manera tan vertiginosa en los últimos años, preparándote también para el futuro y dando el reconocimiento que se merecen a aquellas personas que con sus aportes contribuyeron a la revolución que trajo consigo la invención de la computadora.

Estos pensamientos surgieron a lo largo de mi carrera universitaria, pero fue al final de está que decidí abordarlos y empezar este proyecto de investigación con el fin de centrar en un mismo texto un repaso histórico de la computación acompañado de una descripción didáctica del funcionamiento de las computadoras y su evolución a lo largo de la historia. Pero como la intención tampoco es crear un manual completo de las computadoras y su historia el repaso se centrará principalmente en el origen de las computadoras y como funciona, groso modo, una computadora actual. Para esto último me apoyaré de *CARDIAC(CARDboard Illustrative Aid to Computation)*, un modelo de computación desarrollado por Bell Labs de la mano de David W. Hagelbarger y Saul Fingerman en 1968(Hagelbarger y Fingerman, 1968) con la intención, precisamente, de hacer más fácil de entender a los alumnos de aquel entonces el como funcionaba una computadora. No fue el único intento, otro que se debe mencionar es *Little Man Computer(LMC)* creada por Stuart Madnick y Jhon Donovan del MIT en los años 60, que con un reducido conjunto de instrucciones permitía a los estudiantes probar

la ejecución de programas como si estuvieran en una computadora real. Ambos modelos mencionados en (Mark Jones Lorenzo, 2017), que no puedo dejar de mencionar como una de las lecturas que más me inspiraron en la escritura de esta tesis.

El reto con *CARDIAC(CARDboard Illustrative Aid to Computation)* es que fue creado en 1968, por lo que la concurrencia, el paralelismo o la inclusión de un sistema operativo no pasaban por la mente de sus creadores dado que aún no eran tan relevantes en la industria estos términos. Así que para poder llegar a las computadoras actuales concurrentes, paralelas y con un sistema operativo decidí diseñar una “evolución” de CARDIAC que permita entender estos conceptos de una forma didáctica, aprovechando las ventajas un modelo que abstrae un sistema complejo en los componentes principales que se quieren dar a entender. Aprovechando el contexto histórico se darán las razones por las que cuales cada paso en la evolución de las computadoras fue necesario e incluso aveces imparable, manteniendo la atención principalmente en estos 3 conceptos sin desviar la atención en muchos otros avances, especialmente de hardware, que son fundamentales pero no el tema central de esta tesis.

Es claro que en todo esté tiempo surgieron otros modelos con la intención de ayudar a los estudiantes a entender el funcionamiento de una computadora, algunos de ellos que debo mencionar por su relación con CARDIAC son : *MARIE(Machine Architecture that is Really Intuitive and Easy)* un muy interesante desarrollo presentado en (Null, 2003) que se enfoca en los aspectos básicos del funcionamiento al igual que CARDIAC, otro es un desarrollo hecho en Argentina alrededor de los años 80 llamado *TIMBA(Terrible Imbecile Machine for Boring Algorithms)* que incluso tenía un lenguaje de programación como nos describe el articulo (Álvaro Frías, 2022) centrado en dicho lenguaje, por último he de mencionar un desarrollo mostrado en el articulo (Ajdari y Tabandeh, 2012) que presenta a *Abu-Reiah* un procesador de 8 bits simplificado que incluye un simulador gráfico para ayudar a los estudiantes de arquitectura de computación. Mi intención con las mejoras a CARDIAC no es suplir ninguno de los trabajos antes mencionados, sino más bien complementarlos con un desarrollo histórico que presente de forma clara, concisa y didáctica 4 aspectos fundamentales en la evolución de la computación, la concurrencia, el paralelismo, el uso del sistema operativo y la programación.

Más allá del recorrido en la construcción y diseño de modelos “evolucionados” de CAR-

DIAC el texto se verá acompañado, tal como la clásica CARDIAC distribuida por Bell Labs, con un “kit” que incluye un programa que contendrá tres máquinas virtuales<sup>2</sup>, uno para cada modelo de CARDIAC, con la diferencia de que estas máquinas virtuales ya no serán en papel, sino interfaces gráficas para uso en computadoras de escritorio con la idea de que sea fácil para el estudiante entender la teoría y practicarla, poder ver como se van ejecutando los programas que ya en una versión paralela o concurrente sería un poco más difícil de ver con una computadora de papel dada la cantidad de información que se tiene que almacenar.

---

<sup>2</sup>La emulación de una maquina física que virtualiza cada uno de sus componentes, incluido el hardware.

# Índice general

<b>Introducción</b>	<b>v</b>
<b>1. Historia de la computación</b>	<b>1</b>
1.1. Breve recorrido por el pasado . . . . .	1
1.1.1. Antecedentes . . . . .	1
1.1.2. Primeros autómatas . . . . .	7
1.1.3. Primeras computadoras . . . . .	9
1.1.4. Expansión de las computadoras: The Big Iron Era . . . . .	18
1.2. Más allá de los laboratorios . . . . .	21
1.2.1. Computadoras más compactas . . . . .	21
1.2.2. Lenguajes de programación . . . . .	24
1.2.3. Sistemas operativos y los cambios en el paradigma de programación .	28
1.2.4. Creación de los modelos didácticos de enseñanza . . . . .	33
1.2.5. Actualidad de las computadoras . . . . .	36
<b>2. Arquitectura básica de las computadoras</b>	<b>37</b>
2.1. Funcionamiento de las computadoras . . . . .	37
2.1.1. Arquitectura Von Neumann . . . . .	37
2.1.2. Sistema Operativo . . . . .	41
2.1.3. Iniciando la computadora . . . . .	43
2.2. Modelos de computación . . . . .	44
2.2.1. Modelo de computo concurrente . . . . .	44
2.2.2. Modelo computo paralelo . . . . .	45

2.3. CARDIAC como modelo de computo . . . . .	48
2.3.1. Arquitectura de CARDIAC . . . . .	48
2.3.2. Funcionamiento y lenguaje en CARDIAC . . . . .	51
<b>3. Evolución del Modelo</b>	<b>57</b>
3.1. E-CARDIAC : Electronic CARDboard Illustrative Aid to Computation . . . . .	59
3.2. E-CARDIAC C : Electronic CARDboard Illustrative Aid to Concurrent Computing . . . . .	66
3.2.1. Necesidad de un sistema operativo . . . . .	66
3.2.2. Mejoras necesarias en el Hardware . . . . .	67
3.2.3. Cambios en el lenguaje . . . . .	70
3.2.4. Sistema Operativo Mínimo C . . . . .	72
3.3. E-CARDIAC PC . . . . .	117
3.3.1. Arquitectura renovada para un modelo paralelo . . . . .	118
3.3.2. Un sistema operativo para dos procesadores . . . . .	121
3.3.3. Funcionamiento del sistema operativo mínimo . . . . .	123
3.3.4. Ejecutando procesos en E-CARDIAC PC . . . . .	137
3.3.5. Guía rápida de SOMPC . . . . .	142
<b>4. Conclusiones</b>	<b>144</b>
Referencias . . . . .	146

# Índice de figuras

1.1.	Fuente: Sydney Padua (2015), The Analytical Engine. . . . .	5
1.2.	Fuente: Computer History Museum, The Zuse Z3 Computer. . . . .	12
1.3.	Fuente: Computer History Museum, Colossus Mark 1. . . . .	14
1.4.	Fuente: Computer History Museum, UNIVAC 1. . . . .	17
1.5.	Fuente: ArnoldReinhold - Flickr, IBM 650 en Texas A&M University. . . .	19
1.6.	Fuente: Matthew Hutchinson - Flickr, Ordenador PDP-1. . . . .	20
1.7.	Fuente: Computer History Museum, Ordenador PDP-8 en un automóvil. . . .	22
1.8.	Kit de CARDIAC abierto . . . . .	34
1.9.	CARDIAC construida . . . . .	35
2.1.	Arquitectura CARDIAC, Jorge Vasconcelos(2018) . . . . .	38
2.2.	Obtenida de Tanenbaum, a. Paralelismo On chip, b. CoProcesador, c. Multi-procesador, d. Memorias independientes, e. Multiples computadoras . . . .	46
2.3.	Arquitectura de CARDIAC, concepto original Bell Labs & Jorge Vasconcelos	50
3.1.	Inicio para selección de maquinas virtuales . . . . .	58
3.2.	Arquitectura de CARDIAC por Jorge Vasconcelos, 2018 . . . . .	59
3.3.	Pantalla de inicio de E-CARDIAC . . . . .	61
3.4.	Listas desplegables de E-CARDIAC . . . . .	62
3.5.	E-CARDIAC Muestra de Output . . . . .	62
3.6.	E-CARDIAC Carga masiva por tarjetas . . . . .	63
3.7.	E-CARDIAC Carga individual de instrucciones . . . . .	63
3.8.	E-CARDIAC Lista de espera o cola de instrucciones/datos . . . . .	64
3.9.	E-CARDIAC después de iniciar sus funciones . . . . .	65

3.10. Diagrama de Arquitectura de E-CARDIAC C . . . . .	68
3.11. Diagrama de flujo de SOMC . . . . .	74
3.12. SOMC:Preámbulo . . . . .	75
3.13. E-CARDIAC C Apagada . . . . .	77
3.14. E-CARDIAC C durante el arranque . . . . .	78
3.15. E-CARDIAC C Sistema operativo mínimo cargado . . . . .	79
3.16. Formas de entrar al sistema operativo mínimo C . . . . .	80
3.17. SOMC : Variables del sistema . . . . .	83
3.18. Diagrama de segmento para añadir un proceso . . . . .	84
3.19. SOMC : Contenidos generales del núcleo . . . . .	85
3.20. SOMC : Añadir un proceso, parte 1 . . . . .	85
3.21. SOMC : Añadir un proceso, parte 2 . . . . .	85
3.22. SOMC : Nucleo del sistema operativo . . . . .	86
3.23. E-CARDIAC C Programa en deck . . . . .	89
3.24. E-CARDIAC C Programa en cola . . . . .	90
3.25. E-CARDIAC C Instrucción para añadir proceso . . . . .	91
3.26. E-CARDIAC C Programa 1 cargado en memoria . . . . .	91
3.27. E-CARDIAC C Proceso 1 cargado . . . . .	92
3.28. E-CARDIAC C Tres procesos agregados . . . . .	92
3.29. Acercamiento al preámbulo en el diagrama . . . . .	93
3.30. E-CARDIAC Preámbulo después de una detención en P0 . . . . .	93
3.31. E-CARDIAC C Borrado de proceso parte 1 . . . . .	95
3.32. SOMC segmento del proceso 0 . . . . .	95
3.33. SOMC Lanzamiento de procesos . . . . .	97
3.34. SOMC Lanzamiento de procesos parte 1 . . . . .	98
3.35. SOMC Lanzamiento de procesos parte 2 . . . . .	99
3.36. SOMC Lanzamiento de procesos parte 3 . . . . .	100
3.37. Estatus de organizadores antes de lanzar el proceso . . . . .	100
3.38. Valores del SOM antes de lanzar el proceso . . . . .	101
3.39. Proceso 1 en ejecución . . . . .	101

3.40. Parte final de la primera parte de la ejecución de P1 . . . . .	102
3.41. Diagrama de actualización de proceso . . . . .	104
3.42. SOMC Actualizar proceso . . . . .	105
3.43. Actualizar proceso 1 . . . . .	105
3.44. Conexión entre actualización de procesos y lanzamiento . . . . .	106
3.45. Contexto del proceso 1 actualizado . . . . .	106
3.46. Proceso 1 antes de finalizar . . . . .	107
3.47. Preámbulo en la finalización del proceso 1 . . . . .	108
3.48. Preámbulo en la finalización del proceso 1 a punto de saltar . . . . .	108
3.49. Acercamiento a zona de borrado en diagrama . . . . .	109
3.50. Zona de procesos antes de que el proceso 1 sea borrado . . . . .	110
3.51. SOMC Borrar proceso parte 2 . . . . .	111
3.52. SOMC Borrar proceso parte 3 . . . . .	112
3.53. Zona de procesos después de borrar el proceso 1 . . . . .	112
3.54. Variables del sistema antes de borrar el proceso 1 . . . . .	113
3.55. Variables del sistema después de borrar el proceso 1 . . . . .	113
3.56. Salidas finales de los procesos . . . . .	113
3.57. Salidas finales de los procesos en texto plano . . . . .	114
3.58. Arquitectura Paralela . . . . .	120
3.59. Diagrama de Sistema Operativo Mínimo Paralelo . . . . .	122
3.60. Añadir proceso en SOMP . . . . .	124
3.61. Preámbulo sistema operativo mínimo paralelo . . . . .	125
3.62. Añadir proceso sistema operativo mínimo paralelo . . . . .	126
3.63. E-CARDIAC PC sin iniciar . . . . .	127
3.64. E-CARDIAC PC Booteo . . . . .	128
3.65. E-CARDIAC PC Iniciado . . . . .	129
3.66. Acercamiento al preámbulo del sistema operativo mínimo . . . . .	130
3.67. Acercamiento a segmento de <i>waiter</i> y control de procesos . . . . .	131
3.68. Segmento de control de procesos . . . . .	131
3.69. Acercamiento a segmento de lanzamiento de procesos . . . . .	132

3.70. Sistema operativo mínimo paralelo, segmento de lanzamiento . . . . .	133
3.71. Sistema operativo mínimo paralelo, segmento de lanzamiento parte 2 . . . . .	133
3.72. Acercamiento al segmento de actualización de procesos . . . . .	134
3.73. Sistema operativo mínimo actualización de procesos . . . . .	134
3.74. Acercamiento al segmento de borrado . . . . .	135
3.75. Sistema operativo mínimo borrar proceso . . . . .	135
3.76. Sistema operativo mínimo borrar proceso : parte 2 . . . . .	136
3.77. Proceso de Fibonacci cargado . . . . .	137
3.78. Subrutina de proceso de Fibonacci . . . . .	137
3.79. Añadiendo un nuevo proceso en E-CARDIAC PC . . . . .	138
3.80. Programa pintor en el <i>deck</i> . . . . .	139
3.81. Agregando cantidad de números para serie de Fibonacci . . . . .	139
3.82. Primeras salidas de “pintor” . . . . .	140
3.83. Ejecución finalizada . . . . .	141
3.84. Salida en texto de los procesos ejecutados . . . . .	141

# Capítulo 1

## Historia de la computación

### 1.1. Breve recorrido por el pasado

#### 1.1.1. Antecedentes

Desde que los humanos empezamos a hacer cálculos en el sentido de sumar o restar cantidades representadas por números hemos necesitado de herramientas que nos apoyen en la resolución del calculo, y entre más complejo el calculo necesitamos herramientas más complejas. Las primeras herramientas para apoyarnos en esto fueron formas de representar los números de formas abstractas, una evolución continua en la abstracción de los números permitió a algunas civilizaciones crear artefactos con más potencia de cálculo, como el **ábaco**; el cuál se ha encontrado en diversas civilizaciones en diferentes épocas de la humanidad, el más antiguo del que se tiene registro es el inventado por la civilización Sumeria alrededor del año 2700 A.C. Otras civilizaciones siguieron sus desarrollos de manera independiente no solo para llegar a un artefacto similar al ábaco, sino para evolucionar su forma de hacer cálculos y la complejidad de estos, como es el caso de los griegos, una de las civilizaciones más importantes de nuestro mundo, que dieron un gran aporte al desarrollo de la matemática, tanto que muchos de sus descubrimientos siguen siendo enseñados hoy en día, y que por supuesto aumentaron la complejidad en los cálculos aritméticos. Esta evolución paralela entre la aritmética que teníamos como humanidad y las herramientas para solucionar esos cálculos fue construyendo un camino, o quizá varios, que en ciertos puntos confluyeron en la

creación del siguiente hito en la simplificación de la resolución de cálculos, la **calculadora**, y de la misma forma nos llevarían hasta la creación de la primera **computadora** digital, que posteriormente se convertiría en algo mucho más grande y completo de lo que quizá se imaginaba en su creación(Ifrah, 2001).

Siguiendo la mencionada evolución es importante mencionar la época en la que surgieron las primeras calculadoras mecánicas, así como ciertas herramientas para minimizar el esfuerzo en los cálculos realizados por los usuarios. Es el siglo 17 en Europa occidental, saliendo del renacimiento(europeo), en el cuál tienen necesidades más complejas en ciencias exactas y en problemas aplicados, los cálculos para la navegación y los astronómicos son ejemplos claros de ello. Por esa razón el descubrimiento de los logaritmos por parte de Jhon Napier en 1614 fue uno de los grandes avances en la forma de resolver problemas aritméticos, los logaritmos, grosso modo, te permiten sustituir multiplicaciones por sumas, lo cuál significa una simplificación enorme en la resolución de estas operaciones(O'Regan, 2012, p. 24). Pero aún así los cálculos seguían sin ser tan rápidos, por lo que se desarrollaron tablas de logaritmos ya calculados que hacían esté proceso más simple; otro avance importante en este tiempo fue el desarrollo de dispositivos mecánicos que permitían optimizar ciertos cálculos, *The Gunter Scale* y *The Slide Rule*, creada la primera por Edmund Gunter y la segunda por William Oughtred como mejora a la primera, fueron dos dispositivos que ayudaban en la solución de operaciones aritméticas, de hecho *The Slide Rule*(la regla de cálculo) siguió evolucionando a lo largo de los años para añadir más funcionalidades, como el uso de logaritmos, lo que la mantuvo vigente hasta hace relativamente poco tiempo(especialmente en áreas de ingeniería)(O'Regan, 2012, p.24 y p. 96). Un poco más lejos llegarían Blaise Pascal y Wilhelm Gottfried Leibniz con sus inventos, que ya serían calculadoras en forma, Leibniz inventaría la *Step Reackoner* o simplemente “maquina de Leibniz”, que permitía hacer cálculos aritméticos, en el año de 1673, basada en *The Pascaline* o *Arithmetic Machine* creada por Pascal en 1642(O'Regan, 2012, p.25). Ambas ya podían ser llamadas calculadoras funcionales, aunque ya desde muchos años antes se venía pensando en un dispositivo como la calculadora, de hecho Leonardo Da Vinci dejaría sus planos para construir un dispositivo de calculo(el cual no pudo construir), que ingenieros de la época moderna pudieron seguir para crearla(California State University, s.f.). Podemos notar que las inquietudes por auto-

matizar operaciones estaban ya desde antes que la tecnología de su época les permitiera a los inventores llevar a cabo sus planes, pero que a pesar de esto las siguientes generaciones siguieron adelante con esas ideas, a veces independientemente y otras usando de base lo que ya existía de antes, para continuar con la innovación; este interesante patrón, dónde la teoría avanza más rápido que los desarrollos prácticos, lo veremos de nuevo en el futuro.

Aunque las máquinas de Leibniz y Pascal eran calculadoras en forma, su replicación no era tan fácil y tampoco fueron muy populares, pero si hay una época dónde replicar máquinas para automatizar procesos va a estar en su auge esa es la **revolución industrial**. En 1820 un francés llamado Charles Xavier Thomas de Colmar construyó el *Arithmometer* basándose en la terminología de Leibniz, fue la primer calculadora en forma que se vendió masivamente, en los años sucesivos continuo recibiendo mejoras y al final fue toda una inspiración para una multitud de inventores alrededor del mundo(Ifrah, 2001, p. 127).

El antecesor directo de la computadora, y sucesor casi directo del ábaco no sólo había nacido, sino que ya había conseguido llegar a gran parte de la población, así que, si ya se había logrado el reto, ¿cuál era el siguiente? ¿hacerla más pequeña? si, más pequeña para que pudiera llegar a más personas, más veloz y que pudiera hacerse más fácil el uso para el usuario, al menos esa sería la respuesta general, ¿cuál sería la de un visionario? Alguien que piensa más allá de las convenciones quizás pensaría un dispositivo que fuera capaz de ir más lejos. Y ese era Charles Babbage, un reconocido matemático de su época, miembro fundador de la *Royal Astronomical Society* en 1820, inventor, pionero en la investigación de operaciones, alguien muy distinguido en su época sin duda. Para el año de 1821 estaba diseñando su *Differential Engine*, una maquina totalmente mecánica que en principio buscaba resolver el problema de precisión que tenían las calculadoras del momento, además de resolver funciones polinómicas de hasta grado 6. Lamentablemente no llegaría a ser producida por Charles, en parte por lo difícil(y costoso) que era en la época, y en parte por que Charles ya estaba pensando en algo incluso más innovador; aún así la maquina se logró construir en 1853 por un par de ingenieros suecos que se basaron en los planes de Charles (O'Regan, 2012, p.201).

Lo que Charles ya estaba pensando, y de hecho en 1833, cuando se dio por terminada la construcción de la *Differential Engine*(por que el maquinista que la construía renuncio), estaba decidido a construir era la *Analytical Engine*, una maquina capaz de realizar cualquier

tarea que pudiera ser expresada en notación algebraica y que disminuía la interacción humana para realizar los cálculos. Era una maquina mucho más compleja, tendría un almacenamiento, y un procesador que Charles llamaba *the mill* por que tenía la forma de un molino y era la parte que realizaba las operaciones; además contaría con elementos para la entrada y salida de datos usando la idea del *telar de Jacquard*<sup>1</sup>, que usaba tarjetas perforadas<sup>2</sup> para cambiar los patrones de diseños del telar, Charles pensó en usar estas tarjetas para representar números en ellas para realizar las operaciones de la maquina y de esta forma podía hacer que su maquina fuese “programable”, por que de hecho consideraba dos tipos de tarjetas:

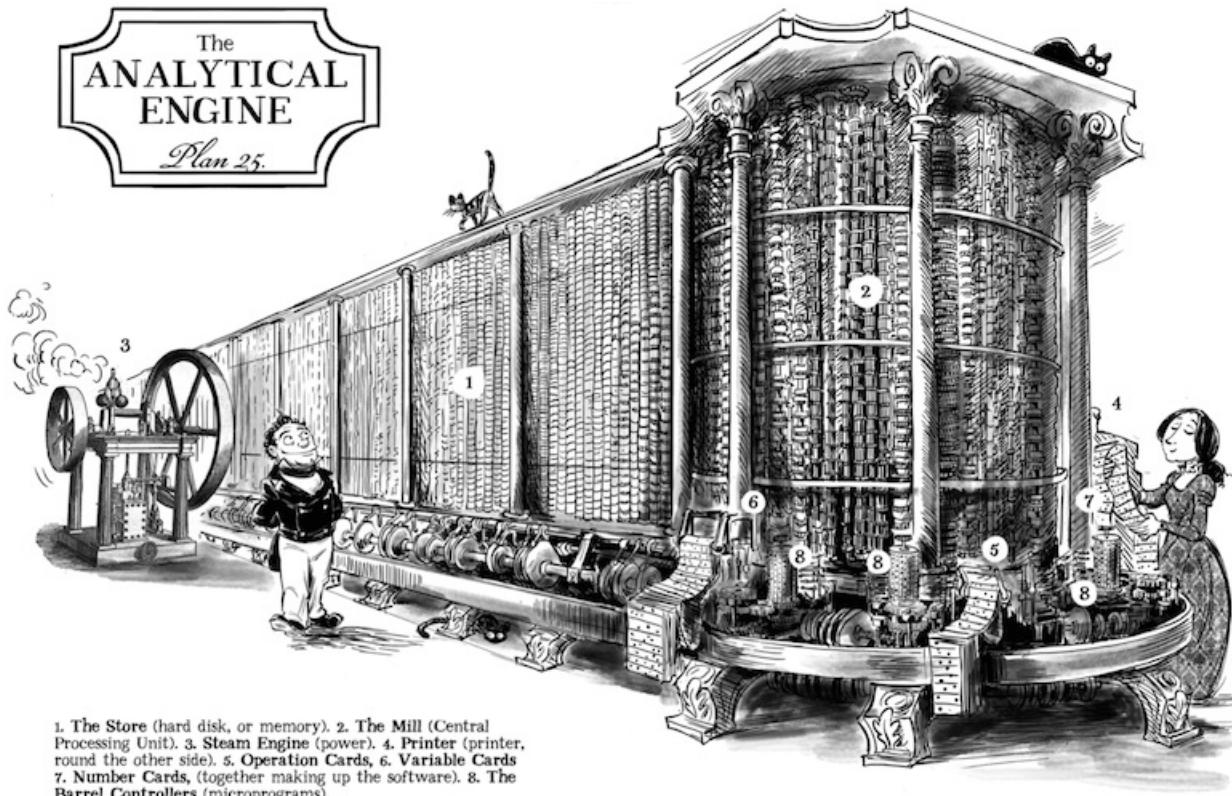
1. Tarjetas de operación
2. Tarjetas de datos

Con está última idea de hecho se puede vislumbrar algo muy parecido a las computadoras con arquitectura Von Neumann, un procesador, almacenamiento interno, programas almacenados(en forma de tarjetas de operación y de datos)(O'Regan, 2012, p.204). Es más en 1943 Lady Ada Lovelace, una matemática que estaba entusiasmada con el trabajo de Charles, realmente brillante, y que escribió un articulo al que nombraría *Notes* al final de una traducción que realizo de un trabajo del francés al inglés escrito por Luigi Manebra sobre la *analytical engine*, en las cuales detalla el funcionamiento de la maquina, las cualidades que la hacen diferente a las calculadoras de la época, y ejemplos de programas para cálculos realmente complejos como el calculo de los números de Bernoulli(de hecho esté se considera como el primer programa de la historia), por supuesto con una cercanía muy alta hacia Charles, como se deja ver en su correspondencia, lo que le permitió realizar un trabajo muy completo, esté articulo junto con las notas de Ada posteriormente sería publicado en al revista *Scientific Memoir* bajo el titulo *Sketch of the analytical engine invented by Charles Babbage*. De los elementos más valioso que nos dejó,fueron por supuesto los programas y los detalles del funcionamiento de la maquina(junto a Manebra) que realizó, pero también su

---

<sup>1</sup>Un telar inventado por el francés Joseph-Marie- Jacquard en 1804 para poder cambiar los patrones de los tejidos.

<sup>2</sup>Eran tarjetas de cartón(usualmente) que tenían orificios con los que se representaban patrones, se continuaron usando por mucho tiempo en las computadoras digitales como podemos ver en el video: *Punch Card Programming*, <https://www.youtube.com/watch?v=KG2M4ttzBnY>



1. The Store (hard disk, or memory). 2. The Mill (Central Processing Unit). 3. Steam Engine (power). 4. Printer (printer, round the other side). 5. Operation Cards, 6. Variable Cards 7. Number Cards, (together making up the software). 8. The Barrel Controllers (microprograms).

Figura 1.1: Fuente: Sydney Padua (2015), The Analytical Engine.

visión sobre el trabajo de Charles, que era un tanto diferente a lo que el mismo consideraba, dado que ella pensaba que la maquina podría actuar sobre otras algo más que números, si se creaban las relaciones correctas entre los números como representación de algo abstracto para fungir como símbolos se podrían realizar más tareas. Ada consideraba a la maquina analítica como algo muy lejano a las calculadoras de su época, y lo era en muchos sentidos, y aunque no fue construida en su tiempo al igual que su predecesora<sup>3</sup>, por sus aportes a la computación se le considera como la primer computadora(mecánica) de la historia(Eric Kim y Alexandra Toole, 1999).

En los años siguientes el avance fue continuo con estás máquinas calculadoras de propósito específico que fueron mejorando con el paso del tiempo, se desarrollo por ejemplo una maquina de censos muy avanzada para su época por el fundador de la empresa hoy conocida como IBM. También la tecnología en general evolucionó, la era de la electricidad y lo electro-

<sup>3</sup>Fue construida en 1991 por un equipo en el museo de ciencia de Londres usando los planos de Charles y es exhibida actualmente ahí, probando que Charles tenía razón.

mecánico llegó a finales del siglo IX, el invento del tubo de vacío, y también avances teóricos que sucedieron, como el álgebra de Boole. Me parece importante hacer notar como no es una sucesión lineal de hechos o descubrimientos lo que llevaron a la creación de la computadora digital o de sus predecesoras, sino que el avance en muchas ramas de la ciencia, de la mecánica, y otras disciplinas fueron interactuando y creando descubrimientos revolucionarios para su época como lo hemos visto en esté capítulo y como se verá en los siguientes. En la siguiente sección analizaremos principalmente ese avance teórico que dio lugar a la evolución de las “calculadoras” mecánicas.

### 1.1.2. Primeros autómatas

Los avances tecnológicos en las máquinas que automatizan cálculos no pararon, e incluso mejoraron en el siglo 20 con la llegada de la electricidad y la electromecánica. Para el año 1913 el matemático Español Leonardo Torres Quevedo desarrollo su *Artimómetro electromecánico*, evitando las dificultades que había tenido Babbage con ayuda de la electromecánica y con su gran ingenio pudo llevar a cabo este desarrollo que resolvía diversos problemas aritméticos y contaba con una maquina de escribir como entrada. Este no sería el único aporte a la computación de Quevedo, otro de los hechos por los que es conocido es por sentar las bases de la *automática* en su *Ensayo sobre automática*, tomando la definición de autómata como “maquina que imita la figura y los movimientos de un ser animado” Quevedo desarrollo la idea centrado en las posibilidades de las máquinas como las calculadoras, pero que pudieran reconocer más objetos, más situaciones y en resolver problemas más difíciles, para así quitarles trabajos repetitivos a los humanos. El en su afán de demostrar que su teoría tenía sentido desarrollo un autómata llamado *El Ajedrecista* que en su primera versión podía terminar un juego de ajedrez, es decir no podía jugar la partida completa, pero podía terminarla(Museo Torres Quevedo, s.f.; Ifrah, 2001).

Quevedo había lanzado preguntas muy interesantes en su ensayo acerca de los autómatas que se relacionan directamente con la computación, pues cuestiona las capacidades de una maquina para hacer algo más que lo que se lograba con una calculadora, aunque por las limitaciones tecnológicas estos cuestionamientos eran más teóricas que prácticas. Pero, de hecho es justo en la teoría dónde se da el siguiente gran avance en la computación; motivados por *los problemas de Hilbert*, una serie de problemas matemáticos que buscaban de dar más rigurosidad a las matemáticas desde el principio del siglo 20, Alan Turing y Alonzo Church comenzaron a trabajar por separado en resolver un problema muy particular, el famoso *problema de la parada*, que sin entrar en mucho detalle trata sobre saber si todo problema matemático reproducible en un algoritmo puede ser resuelto, si una maquina que está ejecutando el algoritmo se detiene con seguridad siempre significa que si puede ser resuelto, siendo un algoritmo una lista de instrucciones ordenadas y finitas que permite solucionar un problema. La resolución del problema es muy compleja y no es la idea del texto

entrar en esos detalles, pero si dar a entender lo importante que fue para la computación su resolución y el trabajo teórico que se hizo.

Se desarrollaron modelos teóricos de máquinas que podían “computar” en el sentido más clásico de la palabra, hacer cálculos. Turing trato de crear una maquina lo más general posible, de forma que cualquier problema que se pudiera computar en esa “maquina de propósito general”, sea una calculadora o una maquina que resuelve polinomios, a tal maquina la llamo *Maquina Universal de Turing* en su famoso articulo *On computable numbers, with an application to the Entscheidungsprobkm*(Sobre números computables, con aplicación a el problema de la parada)<sup>4</sup>, dando también una definición rigurosa del concepto de algoritmo. Un modelo totalmente teórico, pero que mostraba todo el potencial de una maquina de propósito general que podía resolver cualquier problema que pudiese ser expresado como un algoritmo, pero que no podía resolver todos los problemas aritméticos, por que no hay suficientes algoritmos para representar todos los problemas que existen, así que la respuesta al problema de la parada es no, no se pueden resolver todos los problemas matemáticos en base a un algoritmo. De esta forma se iba creando lo que posteriormente sería conocido como teoría de la computación, un estudio matemáticamente riguroso sobre las capacidades de las máquinas de computo que se subdivide en varias ramas, una que nos interesa es la *teoría de la computabilidad*, que estudia que algoritmos puede computar una maquina, para llegar a establecer que ciertos problemas son no computables y por ende no son solubles por medio de ninguna computadora(Ifrah, 2001, p. 272).

La forma de pensar sobre las máquinas ha evolucionado mucho a esté punto en la historia, la visión ahora es sobre una maquina que de soluciones a problemas que puedan ser descritos en forma de algoritmo, y ya no solo soluciones a problemas aritméticos o de funciones matemáticas. Turing y Church fueron dos nombres que dieron un salto gigante en está ciencia, que en ese momento aún no existía, de la teoría de la computación. En estos mismos años la evolución de en las máquinas continuaba, y en los siguientes años su expansión, causada también por el conflicto armado, no haría más que acelerarse.

---

<sup>4</sup>En 1936 se presenta la tesis de Alan Turing y Alonzo Church, con aproximaciones distintas a la solución del problema de la parada.

### 1.1.3. Primeras computadoras

Es el momento de hablar de computadoras en forma, máquinas que ya se pueden considerar uniformemente como computadoras para la definición general que se tiene de éstas. Repasaremos lo sucedido entre el año de 1930 y 1946 dónde por diversas causas, la madurez en el entendimiento de las máquinas de cálculo, de la electromecánica y la electricidad, así como necesidades de una mejora tecnológica para enfrentar una de las guerras más crueles que ha visto la humanidad, se dieron las condiciones para el desarrollo de la computación. En primera instancia como la evolución de las calculadoras, pero que se fue deformando hasta convertirse en más bien máquinas que resolvían problemas específicos.

Como en esta parte de la historia está centrada la discusión de cuál fue la primer computadora de la historia es necesario tener una definición más clara de lo que entendemos por computadora, y por qué ciertas máquinas están en el limbo entre calculadoras o computadoras y otras ya son computadoras en un sentido más completo. Empecemos con la definición del diccionario de Oxford, definición que uso dado que es entre Gran Bretaña y Estados Unidos que se da principalmente el desarrollo de las computadoras en sus inicios, y por tanto es su lengua franca:

*Definición 1:* Una persona que hace cálculos, especialmente con una máquina de calcular.

*Definición 2:* Un dispositivo electrónico para guardar y procesar datos, típicamente en forma binaria, de acuerdo a las instrucciones dadas en un programa (conjunto de instrucciones).

La primera definición, que aún perdura, hace referencia principalmente al significado que tenía antes de 1940. Porque posteriormente, entre 1940 y 1950 con la aparición de las máquinas electromecánicas o eléctricas se empezó a usar el término con la connotación que tenemos de él hoy en día. Por ende la segunda definición nos interesa más, y que es acorde a muchos libros de computación, como en (Goldstine, 1972) Goldstine nos dice lo siguiente acerca de la computadora : “Es un dispositivo electrónico que puede recibir un conjunto de instrucciones, o un programa, y entonces resolver este programa realizando

varias operaciones matemáticas en datos numéricos ”, a partir de estas definiciones podemos establecer 4 características fundamentales en una computadora:

1. Dispositivo electrónico
2. Guarda datos
3. Procesa datos
4. Realiza operaciones a partir de un conjunto de instrucciones dadas por el usuario(Entendiendo programa como una forma de algoritmo)

Todo lo que hace una calculadora lo hace, y con más capacidades, pero hay un punto importante para nuestro estudio, ¿que no sea un dispositivo electrónico es necesario para que una máquina que tiene las otras tres características no sea una computadora? Realmente la mayoría de los autores lo manejan directamente como un dispositivo electrónico y no se involucran en la discusión de que es una computadora. Como veremos realmente un dispositivo mecánico o electromecánico puede cumplir con el resto de especificaciones, pero dado que su uso fue más bien en el nacimiento de la computación y quedó rápidamente superado por la potencia de los dispositivos electrónicos no se encuentra mucho problema en la definición dada por Goldstine.

En esta época hubo una gran explosión de desarrollos en cuanto a máquinas que automatizaban cálculos, desde aquellas que eran calculadoras muy potentes, algunas que ya entran en la discusión de si son o no computadoras, hasta las que evidentemente lo son. Un ejemplo a resaltar es la *Differential Analiser* creada por Vannevar Bush en 1931 en el M.I.T., uno de los logros más representativos en la historia de las calculadoras análogas<sup>5</sup>, dado que desde su inicio se había tenido el problema que eran de un propósito muy específico, pero esta máquina podía resolver una variedad más alta de ecuaciones, lo que la convirtió en la primera calculadora análoga multi-propósito(Ifrah, 2001, p.158). Otra calculadora realmente llamativa de la época fue la *Complex Number Calculator* creada por Samuel Williams y George Stibitz en los laboratorios Bell, una calculadora electromecánica capaz de manejar número

---

<sup>5</sup>La mayoría de máquinas actuales son digitales por su operación sobre cantidades discretas, mientras que los equipos análogos operan sobre cantidades continuas que en versatilidad han quedado superadas por las máquinas digitales.

complejos, contribución que no fue única por parte de Stibitz, ya que en Bell Labs desarrolló muchos conceptos relacionados con la comunicación y computación, generando así una de las etapas más brillantes para Bell Labs sin duda (Ifrah, 2001, p.207).

Precisamente entre estos dos sucesos el Alemán Konrad Zuse estaba trabajando en la construcción de prototipos de máquinas que disminuyeran su trabajo como ingeniero; para 1938 había desarrollada la *Z1*, una calculadora binaria, mecánica, de accionamiento electromecánico y que leía instrucciones de una tarjeta perforada. Para 1939 Konrad llevaría más lejos esta idea al intentar eliminar la dependencia en las partes mecánicas que eran muy complejas y sustituirlas por relés para funcionar con circuitos eléctricos y puertas lógicas (AND, OR, NOT) aprovechando las ideas de George Boole y su álgebra, y de Claude Shannon que introdujo la idea de implementar el álgebra booleana mediante relés eléctricos para crear circuitos, esta sería la *Z2*. Su siguiente gran trabajo no tardó en llegar, y es por la que es recordado principalmente, la *Z3* (se puede ver en la figura 1.2), máquina que fue terminada en 1941; usaba 2600 relés, realizaba aritmética de punto flotante, tenía una “longitud de palabra”<sup>6</sup> de 22 bits, con un almacenamiento de 64 palabras y los cálculos eran realizados puramente en binario, dado que Zuse lo consideraba más eficiente. Era programable mediante tarjetas perforadas y completamente automática, hoy en día es considerada por ciertos autores, como (O'Regan, 2012), como la primera computadora de programas almacenados de la historia. Aunque dicha atribución podría parecer difícil de hacer dado que resultó destruida en 1943 por un bombardeo, fue posteriormente reconstruida y se demostró la capacidad que tenía. De hecho la *Z3* es más parecida a las computadoras actuales que computadoras de más renombre como la ENIAC, e incluso en 1998 Raúl Rojas probó que la *Z3* es Turing Completa, por lo que podemos considerarlo una prueba de que era una computadora con la capacidad de computar cualquier problema que pudiera ser expresado como un algoritmo. A pesar de los problemas en Alemania Zuse pudo construir una versión más avanzada a la cual llamó *Z4*, que fue la primera computadora comercial del mundo, introducida al mercado en 1950. Aunque el desconocimiento en el resto del mundo fue alto, en los últimos años los museos y los autores le han dado el lugar que merece a uno de los padres de la computación (Ifrah, 2001, p.206).

---

<sup>6</sup>En esos años se usaba esa expresión para referirse al tamaño de una variable.

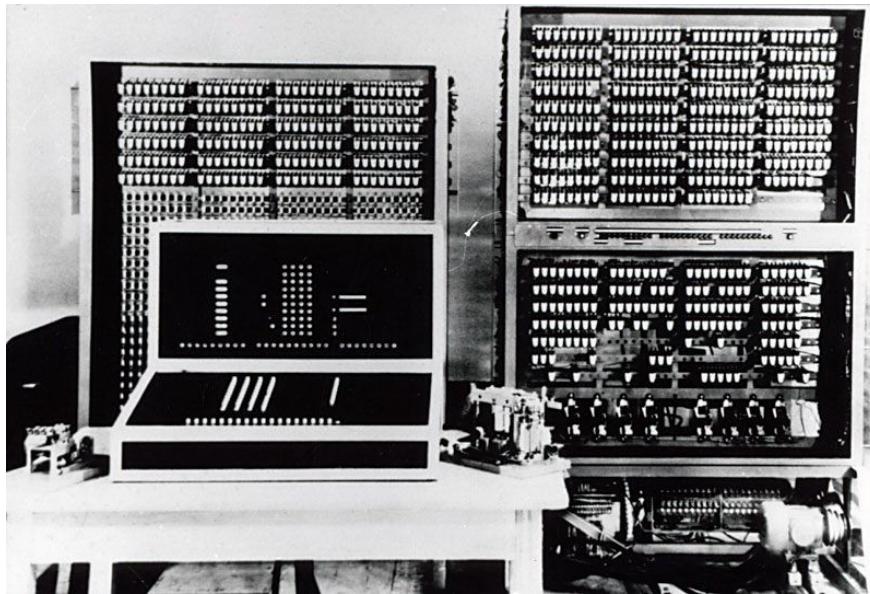


Figura 1.2: Fuente: Computer History Museum, The Zuse Z3 Computer.

La discusión de cuál fue la primer computadora fue un tema de controversia, tanto que se llevo a tribunales en Estados Unidos dado que los creadores de la *ABC* decían que Jhon Mauchly, co-creador de *ENIAC* había visto previamente a la ABC. Finalmente el dictamen fue que el concepto de la computadora fue la concepción de diversas ideas por muchas personas, por lo que no era patentable, resolución que tiene mucho sentido dada la historia que hemos revisado sobre la creación de las computadoras, muchas mentes han aportado en diversos aspectos a la concepción de lo que hoy en día conocemos como computadoras(Computer History Museum, s.f.).

Justamente veamos a uno de los implicados en la disputa, que siguió al desarrollo de Zuse, pero sin nada que ver con el. La computadora fue nombrada como *The Atanasoff-Berry Computer(ABC)*, fue construida por el profesor John Vincent Atanasoff en el colegio estatal de Iowa con ayuda de su estudiante Clifford Berry entre 1939 y 1942, tenía un sistema binario para la aritmética, memoria para guardad datos, circuitos electrónicos, separación entre datos y programas; era en toda regla una computadora, no programable y de propósito específico. Otro desarrollo en paralelo, pero está vez en el M.I.T. fue la *Harvard Mark I*, construida por Howard Aiken y un equipo con apoyo de IBM, destinada a ayudar con cálculos balísticos en la segunda guerra mundial, una maquina electromecánica que fue la primera en ser capaz de imprimir tablas matemáticas, algo que Babbage soñó casi un siglo atrás(Ifrah, 2001, p.

212-218). Algo interesante a destacar de esta maquina es que no tenía las instrucciones y los datos guardados en la misma “memoria”, a diferencia de las que hemos visto y lo que será el estándar en cuanto a arquitectura de computadoras; a este tipo de arquitectura se le llamará arquitectura Harvard con la llegada de los microprocesadores, una arquitectura que hoy en día empieza a tomar cada vez más relevancia, pero que en su momento no fue la arquitectura central del desarrollo de las computadoras(Pawson, 2022).

Después de revisar algunas un tanto desconocidas nos quedan dos que son quizá las famosas computadoras de la época, empezando con *Colossus Mark 1*(figura 1.3), uno de los grandes aportes que dejó *Bletchley Park*<sup>7</sup>, instalación especializada en el trabajo de descifrado de códigos durante la segunda guerra mundial. Con una maquina de descifrado llamada *Bombe* lograron desencriptar los mensajes de la famosa maquina enigma, pero con el avance de la guerra se encontraron con un problema aún mayor, la maquina *Lorenz SZ40/42* que tenía una codificación de muy alta calidad y se usaba únicamente para los mensajes más importantes en la armada alemana. Para descifrar estos códigos es que entra en escena Tommy Flowers diseñando una maquina semi programable que usaba tubos de vacío, lo que la hacía relativamente veloz para la época, logrando realizar una cantidad ingente de cálculos matemáticos con el fin de descifrar los mensajes que usaban la codificación Lorenz; estando disponible a principios de 1944, y su segunda versión unos meses después. Dado su uso militar su existencia se mantuvo en alto secreto por orden del gobierno hasta los años 70, hoy en día se conserva una replica de la maquina en el museo de Bletchley Park(O'Regan, 2012, p.39).

Prácticamente en paralelo se estaba desarrollando en estados unidos una computadora electrónica digital con propósitos militares, centrada en realizar cálculos de artillería para el gobierno, *ENIAC* (*Electronical Numerical Integrator and Computer*), computadora construida por John Mauchly, J. Presper Eckert y un equipo, en el que se encontraba como consejero externo Jhon von Neumann, en *the Moore School of Electrical Engineering of the University of Pennsylvania*, la cuál a pesar de ser de propósito específico al ser programable se le considera una de las primeras computadoras de propósito general. Su programación era muy compleja, requería de mover o reordenar interruptores manualmente e ingresar la información por medio de tarjetas perforadas, lo cuál era realmente tedioso y tardaba demasiado

---

<sup>7</sup>Es el nombre de una instalación militar localizada en Buckinghamshire, Inglaterra.

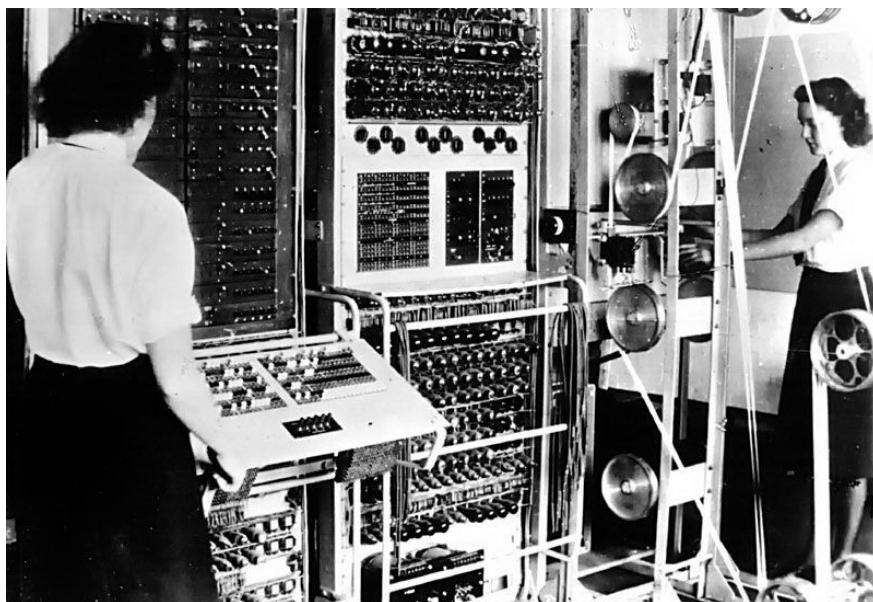


Figura 1.3: Fuente: Computer History Museum, Colossus Mark 1.

tiempo, si además le sumamos que los tubos de vacío que usaba no eran muy confiables, puesto que estos explotaban fácilmente, tenemos una maquina que no era muy confiable y era bastante lenta de programar. Aún así los ingenieros involucrados encontraron la forma de reducir las explosiones de los tubos al no seguido y no hay que perder de vista que lograba reducir el tiempo de procesamiento de horas a segundos en muchos cálculos, un salto enorme en la computación sin duda(Ifrah, 2001).

Fue precisamente en la segunda guerra mundial dónde se dieron los avances más importantes en la historia de la computación, dónde se cambio la forma de pensar en máquinas que resolvían cálculos aritméticos a resolver problemas más avanzados, desciframiento de códigos, calculo de trayectoria para misiles y demás tareas propias de una guerra. La necesidad del avance tecnológico llevo a estos desarrollos a su cúspide, pero con el final de la guerra los desarrollos no pararon, sino que se incrementaron, y el pensamiento sobre una computadora de propósito general se veía presente en las mentes de los científicos que habían impulsado y seguían impulsando los desarrollos de máquinas cada vez más potentes.

En 1946, con la segunda guerra mundial terminada Arthur Burks, Herman Goldstein y Jhon Von Neumann nos describen en el prefacio de (Burks, Goldstine, y Neumann, 1982), hacen un primer ensayo para sintetizar las ideas que hay sobre estos dispositivos electrónicos, tener una imagen de la situación en la que estaban y presentar como los problemas

matemáticos podían ser ahora codificados en un lenguaje que la maquina entendiera. Es en este documento dónde se establece por primera vez la arquitectura de una computadora con *programas almacenados*, concepto fundamental y que hasta hoy en día la mayor parte de las computadoras usa. Notaremos que en el titulo de su ensayo no usan en ningún momento la palabra computadora, y es que para la época lo que tenían era un dispositivo de computo electrónico, como hemos venido leyendo el uso generalizado de la palabra computadora se daría durante el mismo desarrollo de estás. De hecho von Neumann escribiría otro reporte un poco antes llamado *First Draft of a Report on the EDVAC* dado que se había unido a Eckert y Mauchly en el desarrollo de *EDVAC*(*Electronic Discrete Variable Automatic Calculator*), en tal reporte se detalla el diseño de un sistema de computación digital, automático y de alta velocidad, fue uno de los textos que marcaron la época y sirvió de inspiración a otros como Maurice Wilkes para el desarrollo de su propia computadora con programas almacenados, la *EDSAC*(*Electronic Display Storage Automatic Computer*), la cuál tuvo avances muy relevantes que se comentaran en capítulos posteriores. La construcción de *EDVAC* tenía la intención de mejorar dónde *ENIAC* fallaba, uno fundamental que estaba construida con una arquitectura de programas almacenados, es decir que no se necesitaba reconectar para poder ejecutar otra tarea, lo que a su vez ayudaba reducir los errores y facilitaba su programación; se empezó a construir en 1946 y comenzó a operar en 1951(O'Regan, 2012, p. 45).

Otro de los desarrollos que continuo al final de la guerra, o para ser más preciso, empezó al final de la guerra es el desarrollo de una computadora de propósito general en Manchester. Es en Inglaterra con Manchester como uno de sus principales centros académicos, en donde Tom Kilburn y Frederick Williams desarrollaron la primer computadora completamente electrónica, digital, y con programas almacenados; en 1948 se culmino la creación de la llamada *Manchester Small Scale Experimental Computer*, mejor conocida como *Baby* porque era más pequeña de lo común en la época. Esta computadora era más bien un prototipo que después extenderían en la conocida *Manchester Mark I*; de hecho ganaría tanta notoriedad por sus avances que una compañía británica llamada “*Ferranti Ltd.*” se asocio con Kilburn y Williams para comercializar una computadora basada en la que ellos habían construido, está llevó el nombre de **Ferranti Mark 1** y fue la primer computadora electrónica de propósito general comercializada, la cuál fue liberada en Febrero de 1951, poco antes de la **UNIVAC**,

que fue liberada en marzo de 1951(O'Regan, 2012, p. 36).

Precisamente para cerrar esta época está quizá una de las más recordadas, creada también por Mauchly y Eckert tenemos a la muy conocida *UNIVAC UNIversal Automatic Computer*(figura 1.4), diseñada como evolución de su propio desarrollo anterior *BINAC (BINary AutomaticComputer(1949)*, la cual era la primer computadora electrónica con programas almacenados(en EEUU), que basaba su diseño a su vez en *EDVAC*. Era ya una computadora con programas almacenados como lo venían siendo la mayoría de la época, la decisión no es casualidad, las ventajas al momento de escribir instrucciones para estas son muy altas y marcaron un punto de inflexión para el desarrollo de las computadoras, el aporte de los programas almacenados puede ser algo simple pero tan importante y revolucionario que se mantiene vigente aún hoy en día. Incluía un teclado y una consola para escribir, era una computadora de negocios realmente, que fue entregada la oficina de censos en marzo de 1951(en su versión 1) y se mantendría en comercialización para buscar otros vendedores, aunque no era tan fácil vender una maquina tan grande tenía usos muy específicos y quienes podían pagar más de un millón de dólares por ella eran muy pocos; sus compradores en su mayoría eran departamentos del gobierno de Estados Unidos como el *U.S. Air Force*, *U.S. Steel* o *U.S. Navy* que son algunos de los departamentos que compraron la UNIVAC para su uso además del departamento de censos(O'Regan, 2012, p.43).

Como podemos notar también en la línea de tiempo el desarrollo de computadoras sufrió un crecimiento muy alto, en menos de 20 años se paso de apenas tener la idea de una calculadora muy potente a la idea de máquinas completamente electrónicas que resolvían cualquier problema que pudiera ser descrito como un algoritmo. Pero aunque el desarrollo tanto en hardware como en software crecía de manera impresionante en estos años, para la gente normal las computadoras seguían siendo artefactos muy lejanos, lo cuál es entendible, puesto que muy pocas personas en el mundo tenían acceso a una computadora por mucho que sus desarrollos hubieran aumentado. Pero con el tiempo las computadoras dejarían ese aspecto de equipos muy especializados sólo para el uso de grandes corporaciones y oficinas del gobierno, una situación fundamental para que esto ocurriera fue que los desarrollos dejaron de ser gubernamentales(en EEUU) y empezaron a ser privados con el fin de la segunda guerra mundial. De esta forma los diseñadores de las computadoras del gobierno empeza-



Figura 1.4: Fuente: Computer History Museum, UNIVAC 1.

ron a crear sus propias compañías, como Mauchly y Eckert con *Eckert-Mauchly Computer Corporation (EMCC)* o asociarse con otras empresas o bien seguir trabajando de lleno con una compañía de tecnología como hizo Aiken en IBM pero para desarrollos comerciales. Esto trajo como resultado la búsqueda de comercializar estos aparatos cada vez más, por lo que tenían que ampliar el mercado y no sólo se podían quedar con las máquinas hechas para oficinas gubernamentales, debían llegar a las personas.

#### 1.1.4. Expansión de las computadoras: The Big Iron Era

De acuerdo a (Tanenbaum, 2002) tenemos 4 generaciones de computadoras, la primera y que revisamos en la sección anterior se caracteriza por usar tubos de vacío como conductores eléctricos, lo cuál nos da computadoras gigantes con una alta tendencia al error. La siguiente es caracterizada por la invención del **transistor**<sup>8</sup> lo que permitió computadoras más confiables y más pequeñas, los conocidos *mainframes*. La tercera generación fue la del **circuito integrado**, que podía juntar cientos de transistores en un espacio realmente pequeño, lo que permitió la creación de las famosas **minicomputadoras**, computadoras del tamaño de un refrigerador pero que en comparación a sus contemporáneas eran realmente pequeñas. Por último tenemos la cuarta generación, la generación del **microprocesador**, que básicamente llevaba toda la parte operativa de la computadora a un pequeño dispositivo electrónico que tenía más potencia que varias minicomputadoras juntas.

- Primera generación (1945-1955) Tubos de vacío
- Segunda generación (1955-1965) Transistores
- Tercera generación (1965-1980) Circuitos integrados
- Cuarta generación (1980-Presente) Microprocesadores

En la sección anterior nos ubicamos prácticamente en su totalidad en la primera generación y unas décadas antes, para esta sección repasaremos algunos de los avances fundamentales en la computación, pero principalmente algunas computadoras que aparecieron y su impacto. En la primer y segunda generación es dónde se da realmente el avance en cuanto a lenguajes de programación, sistemas operativos, los diversos paradigmas de programación y la forma de usar las computadoras, por lo que esos temas se dejarán para secciones independientes más adelante para darles un espacio propio.

A finales de 1954 IBM estrenaba su primer computadora producida en masa, la *IBM 650*, vendiendo alrededor de 450 en ese año y siendo una de las últimas grandes computadoras que usaban tubos de vacío. Por que desde principios de los años 50 se dio a conocer el desarrollo

---

<sup>8</sup>Un semiconductor que mejoraba en confiabilidad, rapidez y potencia lo que hacían los tubos de vacío y los relés



Figura 1.5: Fuente: ArnoldReinhold - Flickr, IBM 650 en Texas A&M University.

del **transistor** por parte de tres inventores en Bell Labs(Shockley, Bardeen and Brattan), esté invento permitía que diseñar circuitos eléctricos para las computadoras remplazando los (muy poco confiables) tubos de vacío por estos nuevos semi-conductores con los que podían representar los números binarios 0 y 1 a través de cargas eléctricas y crear circuitos lógicos más complejos al unirlos. Dado que eran más confiables, duraderos y eficientes, los costos y tamaños de las computadoras se redujeron considerablemente, está la era de los mainframes, computadoras del tamaño de una habitación con más potencia que las de la primera generación y con un equipo especializado para usarlas; sólo grandes compañías, universidades o el gobierno podían pagar los millones que valían(O'Regan, 2012; Tanenbaum, 2002).

Una de las primeras computadoras en usar transistores fue la *PDP-1 (Programable Data Processor)*(1959) desarrollada por *Digital Corporation*, usaba 2,700 transistores y es muy recordada por que unos estudiantes del MIT escribieron el primer video juego para computadora justo para esta misma, el famoso *Space War!*. Fue todo un éxito para la empresa que en los siguientes años continuaría teniendo relevancia en el mundo de la computación por su

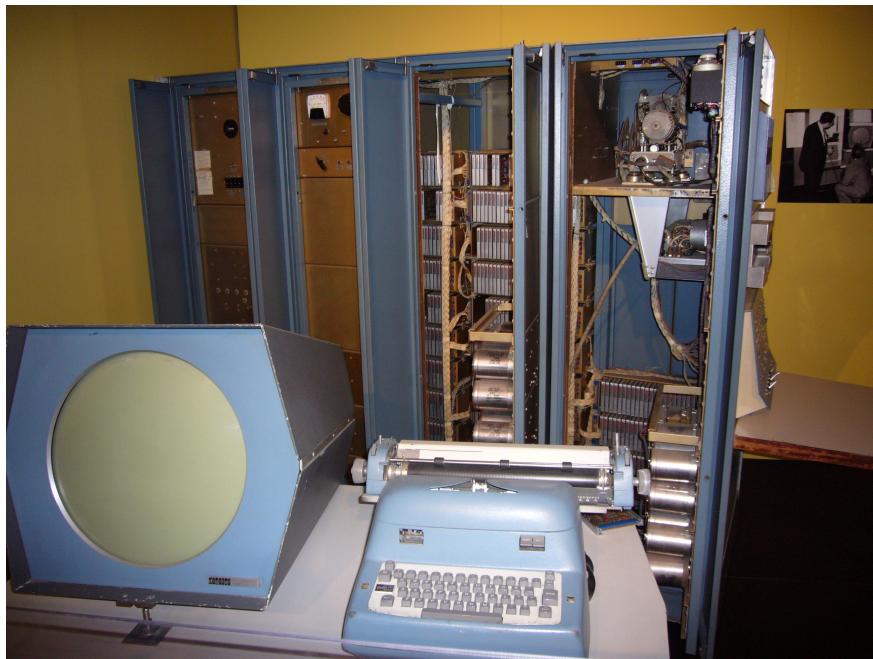


Figura 1.6: Fuente: Matthew Hutchinson - Flickr, Ordenador PDP-1.

continua innovación. Pero IBM siempre va por delante, al menos en esa época, unos años antes, en 1957 introduce la *IBM 608* su primer ordenador que usaba transistores en lugar de tubos de vacío, un año después siguió la *7090* que era una maquina especializada para negocios. Y por si fuera poco en 1961 estrenarían el tope de su serie 7000, la llamada *IBM 7030 Stretch*, realmente veloz y que representaba los avances más grandes que tenía IBM en ese momento(O'Regan, 2012; Computer History Museum, s.f.).

Aún estaba lejos de terminar la década de 1960 y los circuitos integrados comenzaban a aparecer a la par de que desarrollos como la familia de computadoras *System/360* de IBM empezaban a ocupar aún más velocidad y reducir su espacio para poder alcanzar un mercado más grande que aún era muy reducido en esta generación(O'Regan, 2012). La era de las grandes computadoras que ocupaban toda una habitación se empezaba a ver desplazada apenas 10 años después de que empezara, y los continuos desarrollos de nuevas computadoras y de nuevas compañías sólo estaban empezando.

## 1.2. Más allá de los laboratorios

### 1.2.1. Computadoras más compactas

El termino *computadora* era cada vez más recurrente en universidades y empresas, pero sólo aquellas con un alto presupuesto se podían dar el lujo de tener un mainframe en su oficina. Con la intención de aumentar el mercado algunas compañías estaban trabajando en productos comerciales que estuvieran al alcance de más entidades, el paso más relevante que se daría en la década de 1960 y marcaría el inicio de la tercera generación de computadoras lograba precisamente eso. El **circuito integrado** marcaría la siguiente gran revolución en la computación, inventado por Jack Kilby tuvo su primera demostración práctica en 1958; un microchip de silicona(germanio en su invención) que permitía integrar en un mismo y diminuto chip docenas de transistores(u otros componentes eléctricos). Esto permitió tener computadoras más baratas, pequeñas y rápidas con una potencia de procesamiento más alta. La historia detrás de este invento es fascinante, fue tan importante que le permitió a su creador ganar el premio nobel de física en el año 2000, y por su puesto marco el inicio de una era de computadoras que podían salir de los laboratorios(Null, 2003).

El siguiente paso a los *mainframes* fueron las *minicomputadoras*, que por su espacio podían caber en un auto(fig.1.7), el termino podría parecer un poco antintuitivo para el lector del siglo 21 dado que las computadoras de la época son varias veces más pequeñas, pero para ese momento habían pasado de ocupar todo un cuarto para una computadora a sólo utilizar un escritorio. Una de las primeras, y la primera exitosa comercialmente, fue la **PDP-8(Programmed Data Processor -8)** creada en 1965 por “*Digital Equipment Corporation*”, con un precio de 18,500 dólares fue todo un éxito en el mercado por la potencia que tenía contra su precio, que incluso era mucho más bajo que los precios de la serie *System/360* de IBM, una línea de computadoras desarrollada con la intención de que todas fueran compatibles y que ya contaban con circuitos integrados(Null, 2003). En ese tiempo “*HP*” empieza a sonar cada vez más fuerte en el mercado, y en 1966 lanza su primer computadora, la “*HP 2116A*” que ya contaba con circuitos integrados para realizar las operaciones, era el momento dónde todas las empresas estaban sustituyendo sus transistores pro circuitos integrados, con esto cada vez más empresas de otros giros podían tener acceso a una computadora que



Figura 1.7: Fuente:Computer History Museum, Ordenador PDP-8 en un automóvil.

ayudará en sus operaciones diarias(Computer History Museum, s.f.).

Una de las computadoras más pequeñas del momento fue la *Apolo Guidance Computer(AGC)*, computadora diseñada específicamente para el viaje de la nave Apolo 11 a la luna y en la que lograron reducir el tamaño de “7 refrigeradores” a una computadora compacta que pesaba solo 70 libras. La terminaron en 1968 y fue todo un hito en la ingeniería del momento, logro realizado por un equipo especializado del MIT que fue buscado por el gobierno de Estados Unidos para completar una de las tareas más importantes para el despegue del Apolo(Computer History Museum, s.f.).

La tercera generación también fue un punto de inflexión para los sistemas operativos y los lenguajes de programación, que en los principios ni siquiera eran considerados como temas realmente relevantes, el hardware primaba sobre cualquier avance en software, pero en esta época los cambios y la importancia del software empiezan a ser cada vez más fuertes(Tanenbaum, 2002).

El final de la tercera generación llegaría con la invención del **microporcesador**, un sólo chip que podía contener miles de circuitos integrados, pero que no se detenía ahí, podía contener todos los componentes eléctricos de las computadoras, podías tener la unidad central de procesamiento(CPU por sus siglas en inglés) en la palma de tu mano. La primer micro-

computadora<sup>9</sup> exitosa y funcional fue la *Altair 8800*, desarrollada por la empresa MITS, venía en un kit para que los aficionados la pudieran armar; Contaba con el microprocesador *Intel 8080*, era del tamaño de una caja, contaba con 256 bytes de memoria y costaba menos de 400 dólares, el cambio era gigante, pasaban de ser artefactos que sólo las grandes corporaciones podían adquirir a algo que una persona común con recursos suficientes podía comprar, no estaba al alcance de todos, pero ya estaba al alcance de muchas personas. Le seguirían las famosas *Apple I* y *Apple II* poco tiempo después, y en 1981 IBM introduciría su *IBM PC(Personal Computer)*(Null, 2003).

---

<sup>9</sup>Se puede usar como sinónimo de computadora personal y hace alusión a que usaban microprocesadores.

### 1.2.2. Lenguajes de programación

Hablar de los lenguajes de programación requeriría un libro completo para abarcar su historia y evolución, desde el uso de los algoritmos para resolver problemas, hasta la creación de “lenguajes formales” que funcionan de intermediario entre las personas y las máquinas. Pero en esta sección quiero abordar un breve repaso histórico sobre esté aspecto fundamental cuando hablamos de computadoras, y es que tales lenguajes son la forma en que nos “comunicamos” con estos aparatos para dar resolución a los problemas computacionales que tenemos.

Para ser más claros debemos definir lo que es un algoritmo, que informalmente lo podemos definir como una colección de instrucciones simples para resolver alguna tarea, está definición nos sirve lo suficiente para entender el rol que tienen en la programación, una definición más formal es dada en la tesis de Church-Turing, que es la antesala de la teoría de la computación(Sipser, 2013). Siguiendo la definición informal podemos darnos cuenta de que esa “colección” es justo lo que se necesita para indicarle a una computadora lo que tiene que realizar, el único detalle está en que hay que traducirla de lenguaje humano a uno que la maquina entienda.

La comunicación, entendiéndose como la forma de traducir instrucciones, entre humanos y computadoras está completamente ligada al desarrollo del hardware computacional, a medida que las computadoras se volvían más potentes se requería una forma más sencilla de traducir los problemas de lenguaje humano a lo que se conoce como *lenguaje maquina*, que son las instrucciones en números binarios o decimales dependiendo la arquitectura que sea la maquina(las actuales son en su mayoría binaria). Por ejemplo para programar las primeras máquinas de propósito específico como ENIAC o Colossus se usaban enchufes y cables para poder definir patrones en lenguaje maquina que daban la instrucción a la computadora de hacer una tarea en específico(Tanenbaum, 2002, p. 8) <sup>10</sup>.

Pero a medida que las computadoras fueron aumentando su complejidad y podían resolver problemas más complejos el traducir estos problemas más complejos a lenguaje maquina se volvía una tarea muy ardua, incluso con la mejora de la inclusión de las tarjetas perforadas

---

<sup>10</sup>Un ejemplo de como funcionaban las computadoras se puede ver en el siguiente vídeo: *Colossus & Other Early Computers*, [Vídeo], <https://www.youtube.com/watch?v=KkSxC9pFGZs>

para programar a principios de los años 50 llevar un problema a lenguaje maquina era una tarea en la que se perdía mucho tiempo. De ahí que empiezan a surgir ideas para hacer esté proceso más amigable con el usuario, ahí se empieza a gestar la idea del “lenguaje ensamblador”, que en la definición que da (Salomon, 1992) nos dice:

*Un ensamblador es un traductor que traduce instrucciones de origen(en lenguaje simbólico) a instrucciones de destino(en lenguaje maquina), uno a uno.*

Es decir que se crea un lenguaje simbólico para poder sustituir las instrucciones en bruto en código maquina, este lenguaje por supuesto necesitará de una traducción para esas instrucciones. Aunque en las versiones más arcaicas del uso de esté tipo de lenguaje para programar más bien se usaba para describir un problema de forma más entendible para los humanos que después manualmente se traducía por otras personas a lenguaje maquina. En el vídeo (Maurice Vincent Wilkes, 1976) creado por el mismo Maurice Wilkes, creador de la EDSAC, nos muestra el proceso de programación que se seguía en la EDSAC, el cuál tenía un lenguaje simbólico para definir los problemas, pero aún así se requería de una traducción manual para llevarlo a ejecución en la EDSAC. El siguiente paso en el uso del lenguaje simbólico lo podemos ver en la misma EDSAC por que a pesar de que mencione que se tenía que hacer una traducción manual había un programa llamado *Initial Orders* que se cargaba de inicio en la maquina, que funcionaba como un traductor o lo que hoy llamaríamos *ensamblador arcaico* por que traducía algunos códigos muy específicos que facilitaban la programación (Salomon, 1992; Richards, s.f.).

Los siguientes pasos en facilitar la programación fueron la creación de ensambladores más en forma para que el programador sólo necesitara escribir el programa en lenguaje simbólico y el ensamblador pudiera traducirlo a lenguaje maquina. Ya en 1953 la IBM 650 incluía un ensamblador llamado *SOAP(Symbolic Optimizer and Assembly Program)*, sólo 7 años después de la UNIVAC, esté programa daba mucha más facilidad a los programadores para desarrollar sus problemas en un lenguaje más amigable(Salomon, 1992).

Quizá todos estos avances para el programador actual parezcan muy simples, pero fueron verdaderas innovaciones que buscaban que el programador destinara más tiempo a resolver problemas y menos a traducir códigos. Es en esta época, mediados de los años 50 e inicios

de los años 60 la idea era ir más allá de los lenguajes de ensamblador, que si bien eran una mejora aún se tenía que seguir prácticamente la misma lógica para programar. con la facilidad de no tener que hacerlo únicamente con números claro está, así que el siguiente avance lógico serían los *lenguajes de alto nivel*. La idea de estos lenguajes era facilitar la lógica de programación, que el programador ya no se tuviera que preocupar por direcciones de memoria y se centrara en otros aspectos más sustanciales, como la optimización y calidad de su programa. A estos lenguajes se les puede llamar la **tercera generación** de lenguajes de programación, antecedidos por el lenguaje maquina y el ensamblador como primera y segunda generación respectivamente(O'Regan, 2012).

Se debe hacer mención especial a **Plankalkül**, que traducido significa algo así como “calculo de programas”, el primer lenguaje de programación de alto nivel, desarrollado por Konrad Zuse en 1946 para su serie de computadoras *Z*, incluía estructuras de datos, álgebra booleana y condicionales, entre otros aspectos que lo asemejan demasiado a lenguajes más modernos, a lenguajes que en el mundo “occidental” no llegarían hasta mediados de los años 50. Lamentablemente esté y sus demás descubrimientos quedaron sepultados por mucho tiempo a causa de la segunda guerra mundial(O'Regan, 2012)<sup>11</sup>.

No sería hasta mediados de los años 50, en la era de los mainframes y de las primeras minicomputadoras, que los primeros lenguajes de programación de alto nivel comenzarían a surgir. Dos gigantes que aún existen hoy en día nacieron en esa época, **FORTRAN**(Formula Translating System) desarrollado por IBM y **COBOL**(Common Business-Oriented Language) desarrollado por un el comité CODASYL; El primero orientado principalmente al sector científico que usaba las computadoras y el segundo más enfocado en los negocios y las empresas que necesitaban formas más amigables y óptimas de programar(O'Regan, 2012).

En los años siguientes el desarrollo de nuevos lenguajes de programación con diferentes enfoques continuo, desde Pascal, C y Basic, pasando por otros más modernos como Python y Java, junto a muchos más que se han continuado desarrollando con una idea en común, facilitar la programación, que el programador no se torture buscando formas de transmitir sus ideas a la maquina y que se enfoque en diseñar los algoritmos correctos para resolver sus

---

<sup>11</sup>En el video: *Computer History: Dr. Konrad Zuse, Computer Pioneer and the Z Computers (Z3)*, <https://www.youtube.com/watch?v=6GSZQ9g-jiY> se puede conocer un poco más de esté lenguaje y su creador.

problemas.

### 1.2.3. Sistemas operativos y los cambios en el paradigma de programación

Possiblemente la palabra **Sistema Operativo** le resulte muy común al lector contemporáneo, prácticamente todos los que tenemos contacto con la tecnología sabemos que el celular que usamos usa un sistema operativo llamado “Android” o “IOS”, que nuestra computadora seguramente tiene un sistema llamado “Windows” o si somos muy aficionados a los sistemas operativos quizá tengamos “GNU/Linux”. Tal como los lenguajes de alto nivel los sistemas operativos no estaban presentes cuando se crearon las primeras computadoras, estos aparecieron años más tarde cuando las tareas se volvieron más complejas, con máquinas más potentes y con usuarios que querían un trabajo menos estresante. Y aún así un sistema operativo reconocible por alguien del siglo 21 aparecería hasta los años 80 por ejemplo con, el ahora legendario, **MS-DOS** para las computadoras de IBM y potencialmente para cualquier empresa que pudiese pagar por su licencia.

Pero ¿que es un sistema operativo?, la respuesta no es simple, pero para explorar su evolución podemos entender a un sistema operativo como un programa especial dentro de la maquina que realiza dos tareas principales; ser una conexión entre los demás programas y el hardware del ordenador, y gestionar los recursos del hardware(Tanenbaum, 2002).

En el libro (Tanenbaum, 2002) nos presenta la evolución de los sistemas operativos siguiendo el esquema de las generaciones de computadoras, empezando en la **primera generación**(1945-1955) con la primera de las computadoras y la prácticamente inexistencia de los sistemas operativos o algo cercano en ellas, dado que como vimos en la sección anterior la programación era básicamente en lenguaje maquina y quizás en algo cercano aun ensamblador como lo fue “Initial Orders”. Y aunque no fue muy relevante esa época para los sistemas operativos si nos dejó, justamente en “initial orders”, un ejemplo muy temprano de lo que puede hacer un *loader*, que es básicamente un programa que carga programas en la memoria, lo cuál no es una tarea sencilla, puesto que debe llevar cada instrucción a su correspondiente lugar en memoria, por ende, entre más compleja es la maquina más compleja será la tarea del “loader”(Salomon, 1992).

La **segunda generación**(1955-1965) que se encuentra en la época de los mainframes

y de los primeros lenguajes de alto nivel nos deja los sistemas **batch** como ancestros más directos de los sistemas operativos. En esta época ya se tenían a más usuarios trabajando en más máquinas, por lo que se requería de un nivel de servicio cada vez más alto, el que cada usuario fuese con una máquina, dejará su programa y esperara a que le diera respuesta el operador era poco eficiente, la idea que generalmente se adoptó para resolver esto fueron los sistemas batch. Básicamente es una forma de trabajo en la que se colecta un conjunto de *jobs* (un conjunto de programas usualmente de un usuario) para que una o varias máquinas los procesen, sin intervención humana, y se entreguen los resultados a los usuarios una vez se hayan terminado de ejecutar. En este flujo tienen relevancia, además de los programadores, los operadores que llevaban esos *jobs* y cargaban en las máquinas, que además necesitaban cargar los “loaders” cuando eran requeridos y los compiladores si se había trabajado en algún lenguaje de alto nivel; entre los programas especiales como los “loaders” y los operadores hacían el trabajo que hoy se asocia a los sistemas operativos.

Para este punto ya tenemos una necesidad, un programa que administre los recursos de la máquina, y hacia la **tercera generación** de computadoras (1965-1980) se vislumbraría la otra. Había dos ramas en la construcción de las computadoras, las máquinas con un enorme poder computacional (para su época) enfocadas en cálculos científicos, y las computadoras comerciales que tenían un mercado en las empresas. La aproximación que dio IBM para solucionar esto fue crear una familia de computadores que tuvieran ciertas características en común a nivel de hardware, con un sistema operativo común llamado **OS/360**, el cual tenía la intención de funcionar en todas las computadoras de esta familia y funcionando como enlace para los demás programas y el software de forma que no se tuviera que programar de una forma totalmente distinta entre una máquina y otra. El problema con este sistema operativo, es que para funcionar tanto en computadoras orientadas a hacer pronósticos del clima y otros complicados cálculos, así como ejecutar operaciones como imprimir información para ambientes más comerciales hacían de este sistema operativo un programa realmente complejo, construido con millones de líneas en lenguaje ensamblador era realmente difícil darle mantenimiento. Pero es que para cumplir con todos los requerimientos de un sistema operativo es inevitable pensar en un programa de millones de líneas de código, los libros de (Tanenbaum, 2002) y (Silberschatz, Galvin, y Gagne, 2009) muestran en su portada,

con un toque de sátira, dos formas diferentes que un sistema operativo es un programa extremadamente complejo y que controla muchas actividades de la maquina; Silberschatz lo muestra con la clásica portada de dinosaurios haciendo referencia al libro que escribió Fred Brooks, uno de los diseñadores de OS/360 y Tanenbaum con un circo que tiene una cantidad muy grande de participantes que son parte del sistema operativo.

Ahora que se tenía un sistema operativo que agilizaba bastante las tareas, a pesar los problemas que tenía, surgieron otras formas de procesar los programas para aumentar la eficiencia, ahora se podía conseguir la **concurrencia**. La concurrencia no es más que la ejecución de varios procesos en “simultaneo”, es decir que se ejecuta parte de un proceso A y luego parte de un proceso B, buscando la eficiencia y que ambos usuarios obtengan sus resultados de forma eficiente sin retrasar al otro. Entendiendo **proceso**, cuando hablamos de computación, como un conjunto de variables y código que se estará ejecutando en la maquina; el primer paso es tener un programa, un código que contenga un algoritmo para realizar una tarea en algún lenguaje particular para una maquina, esté programa se convertirá en “proceso” cuando entre en la pila de ejecución, pues se le asignará un identificador único, y se resguardaran ciertas variables asociadas a tal programa. Esto en la multiprogramación es lo que permite que aunque suspendas un proceso en algún punto, cuando se regresen a el los recursos de ejecución pueda continuar como si nunca se hubiera detenido, manteniendo todas sus variables asociadas.

Particularmente, en ese tiempo cobro mucha notoriedad una técnica llamada “multiprogramming”, “multitasking”, o “multiprogramación”, lo que buscaba es explotar al máximo los “tiempos muertos” generalmente causados cuando el ordenador esperaba alguna instrucción por parte del usuario, en máquinas comerciales alrededor del 80% o 90% del tiempo era usado en esto, por lo que con la multiprogramación se le da tiempo de procesamiento a otro programa que este en espera. Esta forma de computación es la que actualmente se utiliza en las computadoras que tenemos, dando la sensación de que todos los programas que se ejecutan están ejecutándose al mismo tiempo, aunque en aquellos tiempos el procesamiento no era tan rápido para pensar esto, la comodidad para los usuarios al momento de usar computadoras mejoró mucho. Cabe destacar que las computadoras actuales tienen más que solo computo concurrente, como por ejemplo computo paralelo, pero eso será revisado más

adelante.

En este contexto hay otra variante de la multiprogramación que es importante mencionar, el **timesharing**, aunque dejo de tener relevancia cuando las computadoras se volvieron tan potentes como para tener una por persona, pero que dejo muchas enseñanzas que se siguen aplicando. El concepto busca la concurrencia al igual que la multiprogramación, pero la diferencia es que la segunda en esos momentos estaba principalmente implantada para sistemas batch, lo que buscaba el timesharing es proveer a los usuarios una respuesta más inmediata, que por ejemplo les ayudara en la tarea de debuggear<sup>12</sup> sus programas. De hecho su invención vino desde antes del uso de la multiprogramación, en 1961 fue presentado el **CTSS(Compatible Time-Sharing System)**, desarrollado en el M.I.T., el problema que tuvo es que no se contaba con el hardware necesario para uso en masa, y seguridad de protección suficiente para sus usuarios, en el sentido de aislar a los diferentes usuarios involucrados por ejemplo. Pero la popularización del time sharing en la tercera generación de computadoras se dio, y llegó tan lejos que el MIT, Bell Labs y GE intentaron construir un sistema operativo llamado **MULTICS** que funcionara sobre máquinas conectadas por la red eléctrica para soportar cientos de usuarios, por supuesto la tarea era demasiado ambiciosa para su tiempo, pero aunque al final GE y Bell Labs salieron del proyecto el sistema operativo terminó funcionando de la mano del MIT, de hecho tuvo una gran influencia para el desarrollo de **UNIX** que a su vez es una gran influencia para sistemas como GNU/Linux, macOS, y FreeBSD. Quizá el concepto de time-sharing como se conocía en esa época no llegó a nuestros días, pero el **Cloud Computing** es una clara evolución, con máquinas miles de veces más potentes para un uso masivo de miles de usuarios(Tanenbaum, 2002).

Precisamente esté sistema operativo, MULTICS, en 1969 era capaz de trabajar con múltiples procesadores en **paralelo**, un concepto que no puede faltar en las computadoras actuales(Silberschatz y cols., 2009, p. 899). Esto fue un avance importante en la construcción de computadoras cada vez más potentes, por que al añadir más procesadores puedes conseguir computadoras más potentes, pero necesitas de un sistema que pueda manejar estos procesadores y aprovecharlos de forma eficiente. Cabe aclarar que no es la única forma de

---

<sup>12</sup>Es una palabra proveniente del idioma inglés que se usa para expresar que se hacen pruebas para detectar errores en el algoritmo

paralelismo, puede haber paralelismo en los datos, o bien en las instrucciones, incluso paralelismo a nivel instrucción, entre muchos otros(Null y Lobur, 2003); haciendo énfasis en que esta tesis se centrará en el paralelismo de múltiples procesadores.

Pero sus inicios no fueron fáciles, requerían de hardware muy especializado para realmente ser útiles, realmente no hay muchos casos documentados de computadoras o sistemas paralelos en la tercera generación de computadoras, la ILLIAC IV es un ejemplo de computadora con una arquitectura y un sistema que podían explotar los beneficios del paralelismo, de hecho por la potencia que logró se dice que es la primera supercomputadora. Aunque culminó su desarrollo a mediados de los años 70, tuvo muchos problemas principalmente relacionados al hardware de la época, pero sus logros no faltaron tampoco y por algo es recordada aún hoy en día como una de las pioneras en el mundo del paralelismo(Hord, 1982).

Para las computadoras personales de la **cuarta generación** tampoco fue una adaptación inmediata, por ejemplo las primeras computadoras de IBM y Apple no tenían múltiples procesadores o alguna arquitectura enfocada al paralelismo, no fue hasta los años de 1990 y principios de los 2000 que el ascenso de las computadoras con múltiples procesadores llegaría. A pesar de que las ideas de arquitecturas paralelas como los multiprocesadores estaban ahí desde décadas antes, e incluso había varias implementaciones, especialmente en supercomputadoras, aún faltaba que las computadoras evolucionaran tanto a nivel hardware como software para volverse una opción realista para las computadoras, puesto que no solo necesitas múltiples procesadores(en el caso de dicha arquitectura) sino un sistema operativo que pueda organizar los recursos de manera eficiente para lograr que sea realmente útil la arquitectura(Computer History Museum, s.f.).

#### 1.2.4. Creación de los modelos didácticos de enseñanza

Estamos a mediados de los años 60, época del estreno de *Star Trek* que empezaba a maravillar a las personas con su ciencia ficción y sus computadoras que podían resolver cualquier problema, poco antes del primer viaje a la luna(realizado en el Apollo 11) y sobretodo en un tiempo en que el desarrollo tecnológico sólo iba creciendo. Como ya leímos en secciones anteriores es la época dónde las computadoras empiezan a ser más pequeñas, curiosamente a computadoras como la *PDP-1* y la serie de computadoras que detono les llamaban “minicomputadoras” dado que la reducción de tamaño comparada con otras como la legendaria *ENIAC* era enorme, una época dónde los sistemas operativos aún no llegaban al uso masivo y los primeros lenguajes de alto nivel estaban apareciendo entre los usuarios. Uno de los problemas para esté tiempo era claramente el tener que explicar a los usuarios el funcionamiento de las computadoras cuando estos no habían visto una computadora en su vida, y la primera vez que la veían era para usarla, es algo bastante complejo, así que se buscaron alternativas a la pura teoría que pudieran hacer de este un mejor proceso, así fue que varias empresas y universidades(de estados unidos principalmente) empezaron a desarrollar modelos didácticos de enseñanza de las computadoras que no requiriesen de una computadora real, y es que aunque había universidades con bastante prestigio como el MIT que tenían algunos modelos de computadoras para la investigación, era difícil el acceso para los alumnos, por no decir imposible(Ceruzzi, 2012, p. 71). Una de esas compañías era **Bell Telephone Laboratories** mejor conocido como *Bell Labs*, en aquellos tiempos era un centro de trabajo y de investigación muy prestigioso, y parte de la poderosa *American Telephone and Telegraph* que a pesar de que su negocio principal eran los teléfonos, tenía un área de investigación dedicada al desarrollo de nuevas tecnologías; fue en los laboratorios bell dónde se inventó el transistor, que dio un cambio total a la concepción de las computadoras dada la cantidad de espacio que ahorrar sin perder la potencia de computo, e incluso mejorarla comparada a sus antecesoras las bombas de vacío(Isaacson, 2014, p. 161). Pero *Bell Labs* no sólo se dedicaba a la investigación, también tenía una sección dedicada a la enseñanza, en la cuál se asociaba con universidades para repartir materiales entre ellas y kits de aprendizaje de diversos temas relacionados a las investigaciones de los laboratorios, por ejemplo posterior a la invención del

transistor sacaron un documental junto con un kit electrónico que incluía un pequeño transistor para que los estudiantes pudieran estudiar con aparatos tecnológicos reales y aprender de los “expertos” su funcionamiento, los documentales además estaban dirigidos a público no experto en la nueva área de las computadoras y por ende eran bastante claros, hoy en día el canal de You Tube *AT&T Tech Channel*(canal de la empresa) recopila muchos de estos documentales, un ejemplo es el documental del transistor (AT&T Tech Channel, 2015).

Estas acciones no eran altruismo para Bell Labs, pero aún así a los estudiantes de las escuelas dónde llegaban estos *kits* les era de gran ayuda, hoy en día prácticamente todo lo podemos investigar en internet, pero en aquel tiempo se dependía de las bibliotecas y algún apoyo como el de Bell Labs.

Así fue como en 1968 laboratorios Bell lanza un kit(una caja de cartón), acompañado con un vídeo llamado **Thinking Machines**, el cuál se puede encontrar en youtube (AT&T Tech Channel, 2012), en el cuál narran a través de la pregunta “¿las computadoras piensan?” la lógica que siguen las computadoras para resolver las tareas que se les daban y los funcionamientos internos que tienen para solucionar los problemas que les asignamos. En la caja de cartón venía el manual de instrucciones (Hegelbarger y Fingerman, 1968) y unas hojas de papel como se ve en la figura 1.8, con estos elementos el estudiante podía empezar la construcción de su “propia computadora de papel”, el resultado se puede ver en la figura 1.9 (megardi, s.f.). A la derecha de la imagen tenemos el espacio de memoria con un 001 al inicio y un “8–” al final como apartados de memoria reservada, a la izquierda está la unidad de procesamiento central, el lugar dónde llegan los datos desde la memoria y se realizan las operaciones aritméticas y lógicas que se depositan en el acumulador.



Figura 1.8: Kit de CARDIAC abierto

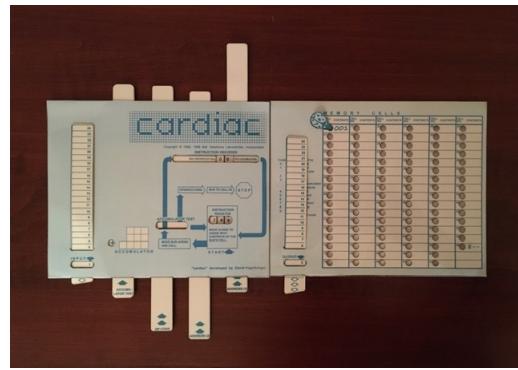


Figura 1.9: CARDIAC construida

Con esta estructura de papel es posible realizar las operaciones básicas que hace una computadora a la vez que puedes ir viendo el proceso que llevan los datos desde la entrada(teclado o una tarjeta perforada) hacia la memoria principal y el flujo que sigue la “maquina” para ejecutar las instrucciones que están en la memoria y diferenciarlas de los datos, así como la interpretación de esas instrucciones, su procesamiento lógico y los cálculos aritméticos, hasta que llegan al acumulador y de ahí según lo que dicten las instrucciones se puede seguir trabajando con esos datos en el acumulador o guardarlos en memoria para su posterior “impresión” fuera de la memoria. Este proceso es, muy a grandes rasgos y muy simplificado lo que realizan nuestras computadoras hoy en día(con varias mejoras), pero que nosotros no vemos, sólo vemos los resultados, es por esta situación que a nosotros como estudiantes del siglo 21 que ya contamos con una computadora a nuestro alcance, en mayor medida que en los años 60, nos puede ser de utilidad CARDIAC para razonar y analizar los procesos que se ejecutan en una computadora y entender su funcionamiento; quizá hoy en día un CARDIAC de papel no nos sea tan práctico, pero el concepto sí que lo es, y llevarlo a un entorno electrónico que nos de más potencia sin perder la esencia del modelo didáctico que es CARDIAC puede ser una gran ayuda para comprender cómo opera una computadora.

### **1.2.5. Actualidad de las computadoras**

La evolución desde los años 80 hasta la actualidad ha sido sorprendente, es muy interesante leer o ver los comentarios de las personas que han interactuado con las computadoras en este lapso de tiempo y han sentido el cambio directamente en su trabajo diario, pasar de usar esos enormes mainframes llamados minicomputadoras, a esas pequeñas computadoras que hoy nos parecen más máquinas de escribir con una pantalla, y por supuesto a las computadoras que tenemos en la palma de nuestra mano en forma de celulares o tabletas. Evidentemente si hacemos una comparación directa las diferencias son bastante claras, pero si nos vamos a los detalles, como hemos visto, hay muchos aspectos que conservan de sus antepasados, otros que incluso sólo han “mejorado”, pero que el concepto sigue siendo el mismo. Como lo es el procesador que utiliza una maquina, o la misma arquitectura, que en su mayoría sigue la arquitectura de Von Neumann o bien la que se ha vuelto muy popular con los teléfonos móviles, una versión moderna de la ya lejana arquitectura Harvard que viene en los procesadores ARM (Valvano y Valvano, 2017, p. 109). Y también hay cambios bastante notables, como el computo en la nube o computo distribuido, que toma la herencia del ahora innecesario time-sharing(por la potencia de las computadoras actuales), pero que toma el concepto base y lo lleva mucho más lejos de lo que siquiera llegaron a pensar los creadores de los primeros modelos computacionales que incluían el time-sharing.

Quizá las dos más grandes novedades de la computación moderna, los procesadores ARM y el computo en la nube, entre otros avances a nivel de hardware como los son las GPU o TPU, procesadores especializados para realizar operaciones matemáticas muy concretas, o a nivel de software como la inteligencia artificial y el *blockchain* nos recuerdan que en el mundo de la computación nada es estático, y la evolución es continua.

# Capítulo 2

## Arquitectura básica de las computadoras

En esta sección revisaremos la composición de una computadora, es decir, los elementos que contiene una computadora y su organización . En la terminología usual de la computación (Tanenbaum, Austin, y Chandavarkar, 2013) nos dice que la *organización o arquitectura de computadoras* no incluye los aspectos de implementación, o el tipo de tecnología usada en los diferentes componentes, para poder centrarse en los elementos que dan forma a la computadora; más aún en este texto usare un enfoque a través de modelos para mostrar sólo algunas partes relevantes de la computadora, dejando fuera muchos objetos importantes en ella, pero logrando así un simplificación necesaria para lograr un acercamiento claro a la estructura de una computadora.

### 2.1. Funcionamiento de las computadoras

En esta sección se tocaran varios conceptos teóricos detrás de la construcción de una computadora, que elementos hacen posible su funcionamiento, y cuales son aquellos que son necesarios a medida que las computadoras evolucionan y los usuarios tienen más necesidades.

#### 2.1.1. Arquitectura Von Neumann

En (Aspray, 1987) von Neumann nos describe como debería ser una computadora de propósito general completamente digital(sin usar el termino computadora), estableciendo

que para el momento se podía tener alguna aproximación a una máquina de cálculo de propósito general o bien a una digital de propósito específico, pero no ambas, fue a partir del final de la segunda guerra mundial que esto empezó a cambiar. De hecho con el tiempo se fueron adoptando las ideas que von Neumann concientemente en ese escrito, siendo *EDSAC* una de las primeras máquinas que seguían de lleno estas ideas. Siendo la más representativa la idea de los *programas almacenados* (en memoria).

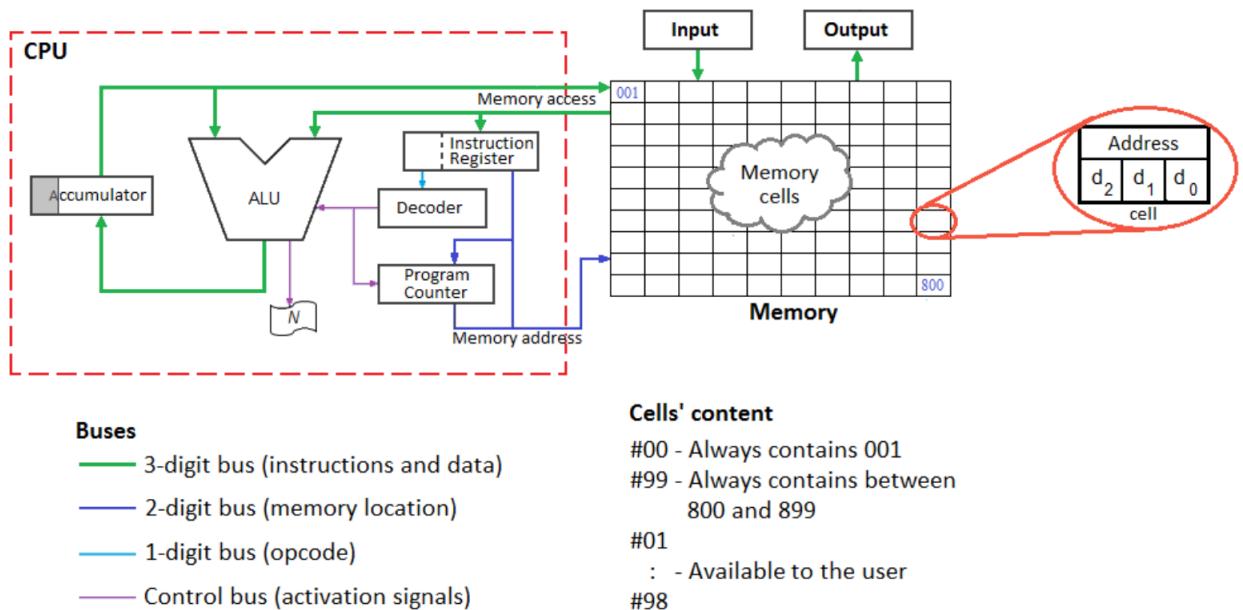


Figura 2.1: Arquitectura CARDIAC, Jorge Vasconcelos(2018)

La figura 2.1 nos ayudará a seguir las ideas de la arquitectura von Neumann. Que como se mencionó al principio tiene como idea representativa la de los programas almacenados, el cuál se refiere a que en la **memoria principal** se almacenen los datos de los programas, pero también las instrucciones. Siendo esta quizás una de las ideas más polémicas que tiene el modelo debido a que la seguridad e integridad de los programas es un problema, instrucciones de un programa en ejecución pueden ser sustituidas por datos y hacer que el programa falle si la implementación no es adecuada. Pero al mismo tiempo está fue una de las ideas clave para llevar las máquinas de computo más lejos de lo que se podía imaginar en los años 40, puesto que esta idea permitía ahorrar muchos costes al almacenar todo en la misma memoria y hacer la programación más eficiente al tener todo en la misma memoria. Memoria que es

muy veloz a costa de ser volátil, es decir en el momento que no tenga corriente eléctrica perderá toda la información que tiene (Tanenbaum y cols., 2013).

Esto nos lleva a su segundo punto, si tanto datos como instrucciones van a estar en la misma memoria ¿como se les va a diferenciar?, es el turno de la unidad llamada **Control**. En esta unidad se deben recibir los datos de la memoria principal y ser clasificados (**Instruction Register**) como instrucciones o puramente datos, y en caso de ser instrucciones ser decodificadas(**Decoder**) para pasar a la siguiente unidad, la unidad que hace los cálculos aritméticos, la unidad aritmético lógica(**ALU** por sus siglas en inglés). Unidad que necesita resguardar sus resultados en un espacio particular llamado **Acumulator** o acumulador, un espacio de almacenamiento especial en el que terminan todos los cálculos de la *ALU*, llamado también **registro** a este espacio es una unidad separada de la memoria principal en el que se puede de almacenar poca información pero de forma muy rápida, todos los objetos dentro de la unidad de control son registros . Además se necesita una ejecución secuencial, de forma que una vez se lea una instrucción/dato de la memoria principal continue a la siguiente, teniendo la memoria ordenada de forma ascendente con números que serán las direcciones de cada espacio de memoria basta con tener un contador que vaya incrementando para así conseguir la ejecución secuencial, el **program counter** logra esto en CARDIAC al incrementar de uno en uno a menos que la unidad de control a través de sus buses que pueden llevar información del decodificador indiquen que se debe hacer un salto diferente. Los buses son el sistema de comunicación interno de la computadora, cada uno con una especificación de acuerdo al tipo de datos que pueden pasar por el para comunicar las diferentes unidades dentro de la maquina(Aspray, 1987).

Por supuesto se necesita también una forma de comunicación con el usuario, de modo que se puedan transmitir las instrucciones a la maquina y que está las ejecute de forma automática. Para ello se necesita de dispositivos de entrada y salida(**Input/Ouput devices**) para que el usuario sea capaz tanto de dar instrucciones a la maquina como de recibir resultados de ésta(Aspray, 1987).

Con estos puntos básicos tenemos un modelo de computación que representa de forma bastante cercana a las computadoras actuales y que es muy cercano a las computadoras de los años 60. von Neuman en su escrito ciertamente describió con más profundidad su modelo,

que acompaña con detalles técnicos a nivel arquitectura y a nivel de implementación, pero también incluye una *jerarquía de memorias* que no aparece en el modelo de *CARDIAC* en la cuál además de la memoria principal añade una memoria *cache* y una *memoria secundaria*, siendo cada una más lenta que la anterior, pero con mucha más capacidad, puesto que el problema desde aquellos tiempos hasta la actualidad en este sentido es la competencia entre velocidad contra capacidad de almacenamiento. La memoria secundaria es lo que hoy conocemos como disco duro, y la memoria cache es un intermedio entre la memoria principal(hoy RAM) y la memoria secundaria, permitiendo aumentar la velocidad en las ejecuciones de procesos(Aspray, 1987).

Una vez que se tiene la arquitectura se define el lenguaje de la maquina, aunque ciertamente hay una correspondencia bilateral en estos dos conceptos, dado que tanto el lenguaje define a la maquina como la maquina al lenguaje (Tanenbaum y cols., 2013). El lenguaje de una maquina con esta arquitectura debe tener las operaciones básicas, sumar, restar, multiplicar y dividir, pero si tratamos de ser eficientes podemos quitar las operaciones de multiplicar y dividir dado que son extensiones de la suma y la resta. Otro aspecto que debe considerarse es el de las “condicionales”, instrucciones que permitan continuar por una rama del código u otra dependiendo del resultado de alguna operación, de la misma forma puede haber subrutinas que varios programas quieran utilizar por lo que una instrucción que permita “saltar” a dichas rutinas directamente también será necesaria. Por último la comunicación con los dispositivos de entrada y salida nos hacen requerir instrucciones para que el usuario pueda ingresar datos a una dirección particular de memoria y también para poder obtener los resultados desde la memoria principal o desde el acumulador en el dispositivo de salida(Aspray, 1987).

Con todas estas unidades, y un lenguaje definido tenemos un modelo bien definido que cumple con la arquitectura de von Neumann, y en este corto texto se puede apreciar otra de las características que hizo a este modelo tan ampliamente aceptado hasta el día de hoy, su simplicidad. Sin necesidad de más de unas cuantas páginas pudimos describir los puntos básicos para tener una computadora que siga esta arquitectura, ciertamente más allá del modelo hay una profunda complejidad al momento de construir una computadora, desde los aspectos más técnicos del Hardware que requieren un avanzado estudio en ingeniería y

física, hasta aquellos elementos del Software que separan en muchas más capas las unidades que mencionamos y llenan de muchas más instrucciones a la maquina con tal de tener más potencia y efectividad de cálculos(Tanenbaum y cols., 2013). Pero que aún así nos permite tener una vista clara y concisa de una computadora clásica, y que de hecho es la arquitectura que siguen la mayoría de computadoras con procesadores Intel o AMD, con ligeros cambios, evolución y muchas mejoras, tales como las capacidades de concurrencia y paralelismo para aumentar el poder de computo.

### 2.1.2. Sistema Operativo

En el capítulo primero vimos como se fue dando la necesidad de tener un software que pudiera ser el intermediario entre la maquina y otros programas, así como administrar los recursos de la misma. En esta sección exploraremos las características principales de un sistema operativo que nos permitirán entender de forma más completa su necesidad dentro de la maquina.

Como comenta Tanenbaum en (Tanenbaum, 2002) depende de a quien leas será la aproximación que te de sobre los sistemas operativos, ya sea más orientado a ser una extensión de la maquina o bien a ser un administrador de recursos. Para los usos que se le darán en el desarrollo de los modelos concurrentes y paralelos adoptaremos un enfoque más centrado en la administración de recursos, pero sin dejar de lado la otra parte de lo que es un sistema operativo, a partir de ahora abreviado como *SO*.

Un *SO* cumple con las dos funciones mencionadas anteriormente para una computadora, usualmente con arquitectura von Neumann. Una de esas funciones es ser una extensión de la maquina en el sentido de presentar a los programas(del usuario) una abstracción de la maquina, de esta forma por ejemplo un programa de visualización de imágenes no tiene que ser programado para entender como funciona internamente un disco duro, en cual de los discos que lo conforman debe buscar la imagen o como se debe comunicar con este dispositivo, en cambio el SO crea una capa de abstracción en la que se encarga de comunicarse con el disco duro y presentarle al visualizador un sistema de administración de archivos más simple, general, con el que la comunicación sea más sencilla para buscar la imagen en cuestión. De esta forma podemos tener más eficiencia y facilidad en la programación, dado que el sistema

operativo se encargará de toda la tarea de comunicación directa con la maquina, para esto debe tener los permisos de acceso más altos que cualquier otro programa, la posibilidad de ejecutar instrucciones que podrían ser peligrosas para programas de los usuarios(Tanenbaum, 2002).

En la otra rama está su actividad como administrador de recursos, que es una consecuencia inmediata del control que tiene sobre la maquina. Cuando las máquinas aumentaron su complejidad la necesidad del sistema operativo fue absoluta, con una cantidad finita de recursos el correcto uso de ellos puede hacer la diferencia entre un buen funcionamiento para el usuario y uno deplorable. Un ejemplo claro de esto lo podemos ver con Windows en computadoras que no tienen especificaciones muy altas, Windows 8 en una maquina de 2GB de RAM es prácticamente inservible para el usuario, mientras que en esa misma maquina un sistema operativo como GNU/Linux Ubuntu 20.0 puede funcionar de manera optima para el usuario por su mejor gestión de recursos del sistema. Por ende la administración de recursos es vital para un buen sistema operativo, esta gestión va desde los dispositivos de entrada y salida hasta los tiempos de ejecución de cada **proceso** en un modelo concurrente o paralelo, que es precisamente a los que está orientado principalmente un sistema operativo, en un modelo simple de computo no es tan necesario(Tanenbaum, 2002).

Un proceso es básicamente un programa en ejecución, por lo que necesita más información que sólo las instrucciones del programa, necesita el espacio de direcciones para escribir o leer del programa, un conjunto de registros entre los que se encuentran el program counter(para saber su ubicación en todo momento), una lista de archivos abiertos, procesos relacionados, y toda la información que se necesite para ejecutar el programa, con todo esto se forma un paquete al que llamamos **proceso**. La información de los procesos debe estar almacenada en la memoria principal, pues se necesita un acceso rápido y sólo está disponible mientras la maquina esté encendida, esta información se guarda en una especie de **tabla de procesos** en la que cada proceso tiene un identificador único, esta tabla es administrada por el sistema operativo que decide sus tiempos de ejecución(Tanenbaum, 2002).

Tanto el espacio de memoria dónde se encuentra la tabla de procesos, como el espacio de memoria dónde está el sistema operativo, por que en efecto, el sistema operativo es un proceso en si mismo que se encuentra en ejecución dentro de la memoria principal, deben ser

protegidos, por lo que estás y otras áreas son espacios restringidos de memoria que el mismo sistema operativo protege de su acceso a los usuarios, o incluso la protección puede estar desde el mismo hardware. Y como se menciono al principio también el sistema operativo se encarga del sistema de archivos, al cuál además de proveer una capa de abstracción para los demás programas también gestiona el acceso a diversos archivos, así como realizar la protección de los mismos dependiendo de las necesidades del usuario(Tanenbaum, 2002).

### 2.1.3. Iniciando la computadora

Como continuación de la sección anterior es necesario hablar del inicio de operaciones de la computadora, del **Booting** forma en la que se le dice al inicio de operaciones de una computadora, la acción de cargar un programa inicial que permita al usuario usar la computadora. Muchas computadoras en el pasado no necesitaban de esto, especialmente cuando tenían que mover cables para cambiar rutinas de ejecución, pero con la programación por tarjetas perforadas y la idea de tener procesos más automáticos la necesidad de que la computadora iniciara algún programa al principio para que respondiera de forma automática fue inevitable. Hoy en día cada computadora tiene un sistema de arranque aparte en la memoria **ROM Read-Only-Memory**, memoria de la cuál no hemos hablado pero que en algunos modelos se coloca en el mismo espacio que la memoria principal, dado que es una memoria de sólo lectura se puede considerar que las direcciones #00 y #99 en el modelo de CARDIAC de la figura 2.1 son instrucciones en la memoria ROM puesto que no se pueden editar. Este pequeño sistema de arranque funciona para preparar las operaciones básicas de la computadora y en el caso de tener un sistema operativo de iniciararlo. El sistema operativo que debe estar almacenado en un espacio físico también es cargado por el sistema de arranque en la memoria principal y así inicia su ejecución, en ese momento toma el control de la administración de recursos en la computadora(Tanenbaum, 2002).

## 2.2. Modelos de computación

### 2.2.1. Modelo de computo concurrente

Ya se menciono que en la tercera generación de computadoras cobro mucha notoriedad la “multiprogramación”, un claro ejemplo de concurrencia. La cuál podemos definir de forma muy general como la acción de ejecutar varios procesos por partes en un mismo *CPU* de forma que todos tengan tiempo de ejecución. de esta forma si tengo dos procesos A y B con 15 etapas cada uno se pueden ir ejecutando intercaladamente estas etapas para que se ejecuten los dos “casi al mismo tiempo”. Las computadoras actuales consiguen esto e incluso por su velocidad nos hacen creer que de verdad se ejecutan al mismo tiempo varios procesos, cabe aclarar que esta disciplina ha evolucionado y no son solo los procesos los que se ejecutan concurrentemente, sino los **hilos**(partes de un proceso) se ejecutan de forma concurrente para conseguir aún más eficiencia en la ejecución de programas(Tanenbaum, 2002).

La concurrencia ofrece grandes ventajas para la ejecución de procesos, pero de la misma forma exige lidiar con otros problemas, puesto que hay que gestionar los recursos para repartir entre N procesos, asegurar que esos procesos no invadan el espacio de memoria de los demás y que en cada momento que un proceso sea sacado de la ejecución toda su información, incluidas las variables globales que esté pueda tener sean resguardadas en un espacio de memoria seguro para que cuando vuelva a tener tiempo de ejecución sus datos estén dónde el proceso espera que estén y esa “pausa” no afecte el comportamiento del proceso. Asegurar esto es una tarea difícil cuando los procesos comparten recursos, y es tarea del administrador, es decir del sistema operativo lograr que todo esto funcione como debe(Leslie Lamport, 2015).

El campo de estudio de la concurrencia es bastante amplio, se dice que comenzó en 1965 con la presentación de Edsger Dijikstra de uno de los problemas más relevantes en este campo, conocido como “problema de exclusión mutua”, un problema de recursos y sincronización que consta de varios procesos compitiendo por estos y dónde se tiene que asegurar el acceso exclusivo a cada uno de ellos a los recursos(Gadi Taubenfeld, s.f.). Problemas como este, y como los mencionados anteriormente, son temas comunes en la concurrencia, gran parte del esfuerzo que se tiene al momento de un desarrollo con procesos concurrentes es evitar al máximos estos problemas de sincronización.

### 2.2.2. Modelo computo paralelo

Paralelismo es una palabra que hoy es usada en muchas situaciones, la noción general la tenemos medianamente clara, dos o más procesos ejecutándose al mismo tiempo. Pero no hay una sola forma de paralelismo, quizá la forma más común en que pensamos en paralelismo es la de multiples *CPUs* que pueden ejecutar múltiples instrucciones al mismo tiempo, otro ejemplo es el de procesar múltiples cadenas de datos en un solo procesador. Dependiendo de la arquitectura se pueden lograr diferentes formas de paralelismo, con la evolución de la computación la distancia entre diversas arquitecturas se fue haciendo más pequeña ya que cada una fue tomando elementos de otras para volverse más eficiente, por lo que hay formas de paralelismo que podrían parecer exclusivas de las arquitecturas Harvard que se pueden ver en procesadores con arquitectura von Neumann. El **pipelining** es uno de ellos, una forma de **paralelismo a nivel de instrucción** también conocida como “parelismo de bajo nivel”, debido a que no logra aumentar tanto las capacidades de computo como otras formas si lo hacen. Se caracteriza por ser una “línea de producción”, es decir que cada etapa en el proceso de computo es tratada como una etapa en una línea de producción de forma que cuando una instrucción está en la etapa de calculo aritmético, en ese momento debe haber otra instrucción en la etapa de decodificación, y así sucesivamente, de forma que cada etapa de la línea de producción esté ocupada en cada ciclo de reloj. Este ciclo se conforma desde que una instrucción es tomada de la memoria hasta que el contador de programa salta a la siguiente dirección de memoria, con el pipelining se consigue una eficiencia muy alta al no desperdiciar tiempos, se consigue paralelismo, pero el manejo de los recursos es realmente complicado para el sistema operativo por los recursos compartidos que se tienen(Null, 2003, p.421).

Ahora, si pensamos en la alternativa de múltiples procesadores aún puede haber variantes, tenemos el concepto de memoria compartida en múltiples procesadores que data desde la década del 70. En este varios procesadores comparten una misma memoria con diferentes buses que los conectan a ella(parte c de la figura 2.2), posiblemente con una memoria cache individual para cada procesador, o podría ser que está también fuese compartida. También está la alternativa de que cada procesador tenga su propia memoria(parte d de la figura

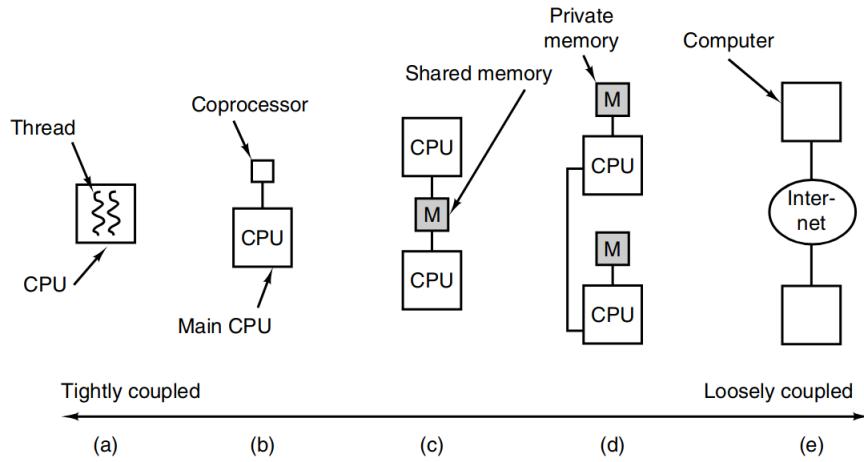


Figura 2.2: Obtenida de Tanenbaum, a. Paralelismo On chip, b. CoProcesador, c. Multiprocesador, d. Memorias independientes, e. Multiples computadoras

2.2), permitiendo más independencia a cada procesador. En todos los casos el problema de la sincronización es clave para lograr que el computo en paralelo sea realmente funcional y no termine siendo más costoso, riesgoso y lento(Null, 2003).

Las otras formas de paralelismo que podemos ver en la figura 2.2, son por una parte en la parte a, el paralelismo a nivel de hilos, usando un sólo procesador o varios tratando de explotar al máximo el tiempo de ejecución, usando generalmente el *pipelining* para conseguir más eficiencia. Después tenemos la parte b, que es más bien tener un procesador principal y uno auxiliar para ciertas ejecuciones, esté es un concepto realmente interesante que ya mostraban los mainframes de la serie *IBM 360* con procesadores independientes para los *I/O* que suelen ser unos de los procesos más complejos que tiene una maquina. Finalmente la parte e nos presenta el paralelismo con diferentes computadoras conectadas a través de internet, el cual es el que otorga un nivel más alto de paralelismo al tener completamente más de una computadora realizando cálculos de forma sincrona con otra (Tanenbaum y cols., 2013).

Las computadoras actuales generalmente usan todos estos recursos, por que si la concurrencia y el *pipelining* son técnicas para eficientar al máximo un sólo CPU, cuando se tienen varios *CPUs* no se van a usar a un nivel de eficiencia medio por no usar las técnicas antes mencionadas, se tendrá la intención de maximizar la eficiencia de cada uno y luego maximizar la eficiencia en su sincronización. Claramente no es sólo juntar diversas técnicas

y formas de paralelismo, cada procesador o computadora tienen sus características que permiten ciertas formas de paralelismo/concurrencia y su implementación depende del diseño que estas tengan(Tanenbaum y cols., 2013).

## 2.3. CARDIAC como modelo de computo

Este es un magnífico ejemplo de lo que un modelo debería ser, la formulación de un objeto complejo en aspectos puntuales que explican características de tal objeto. Si quisiéramos estudiar la física detrás de una computadora el modelo tendría que estar centrado en el diseño de los chips de silicio que hacen posible la creación de miles de transistores en unos cuantos centímetros, en cambio, si quisiéramos analizar las conexiones que hacen posible que la electricidad pase a través de semiconductores y nos dé la oportunidad de crear puertas lógicas, el modelo se debería centrar en la electricidad y los materiales conductores que permiten su uso. Si queremos analizar la organización de una computadora, los elementos que hacen posible el cómputo de instrucciones escritas en forma de algoritmo, la programación en un bajo nivel, y como interactúa todo esto para que los usuarios obtengan resultados de forma automática, así como tener un ejemplo gráfico y conciso de como opera una computadora de propósito general sin tener una, el modelo indicado es **CARDIAC**.

### 2.3.1. Arquitectura de CARDIAC

Vamos a analizar la arquitectura de este modelo siguiendo la figura 2.1 de izquierda a derecha. En la parte derecha tenemos una cuadrícula con 100 recuadros que representan los espacios de memoria (la memoria principal y volátil), cada espacio de memoria cuenta con una dirección numerada del #00 al #99 y puede almacenar tres dígitos, como *CARDIAC* utiliza numeración decimal por facilidad, cada dígito puede ir del 0 al 9. Dado que sigue una arquitectura von Neumann, los datos y las instrucciones se almacenan en la misma memoria, en la memoria no se pueden distinguir estos, solo se hace distinción en el *CPU* con las unidades de control (Hegelbarger y Fingerman, 1968).

En el caso de que el contenido sea un dato se utilizan los tres dígitos para representar el dato completo, con ceros a la izquierda si el número no llega a tres dígitos representativos, en cambio, si el contenido es una instrucción significa que el dígito a extrema izquierda ( $d_2$ ) es el código de operación, y los demás ( $d_1$  y  $d_0$ ) son información para el código de operación. Para la primera dirección de memoria tenemos una especie de memoria *ROM*, puesto que por defecto viene el número 001, y para la última dirección una especie *EEPROM* (*Electrically*

*Erasable Programmable ROM*) por que siempre va a tener un número entre el #800 y el #899 debido a que el dígito  $d_2$  se mantiene fijo, pero cambian los últimos dos. Dejando disponible para el usuario los espacios de memoria desde la dirección #01 hasta la #98 (Hegelbarger y Fingerman, 1968).

Con el resto de espacio disponible para el usuario en la memoria principal, necesitamos darle conexiones a la memoria principal para que el usuario pueda hacer uso de ese espacio. Para eso contamos con los **buses**, el sistema de comunicación que suelen usar las computadoras para transferir información entre diferentes unidades. En el caso de *CARDIAC* tenemos 4 tipos de buses identificados con colores diferentes; los de color verde transfieren datos e instrucciones, puesto que son los que tienen mayor capacidad de transferencia(tres dígitos), después están los de color azul que transfieren direcciones de memoria(ocupan dos dígitos), para los códigos de operación se usan los de color azul que solo ocupan un dígito, y por último están los buses de control en color morado que transfieren señales de activación. Para la entrada y salida de información se ocupan los buses de tres dígitos, mismos que se ocupan para transferir datos e instrucciones al *CPU* (Hegelbarger y Fingerman, 1968).

En la imagen 2.3 podemos ver un diagrama más centrado en los buses y las conexiones de estos; en la parte inferior izquierda los vemos con su etiqueta, y sobre ellos está la CPU que se conecta a la memoria principal usando los buses verdes(de tres dígitos) para mover tanto datos como instrucciones, y con buses de color azul(dos dígitos) para enviar direcciones de memoria. A su vez, la memoria se conecta a los periféricos de entrada(*input*) y salida(*output*) con los buses de tres dígitos, los más amplios de los que dispone. Internamente, la CPU utiliza buses azules para mandar únicamente un dígito, que será el código de operación, y los buses lilas para enviar señales de activación a los componentes que utiliza el mismo CPU.

El *CPU* es la unidad dónde se concentran todos los elementos que hacen posible el cómputo de las instrucciones y datos que se almacenan en la memoria principal, está conformado por lo que está dentro de la línea punteada en rojo en la imagen 2.1, por la importancia de cada elemento dentro de la unidad de procesamiento central(*Central Processing Unit*) se hará un repaso individual a cada uno.

Revisemos cada elemento del CPU analizando sus funciones, para ello es preciso tener claro el concepto de **ciclo**, un ciclo completo es cuando regresas a la posición inicial, y algo

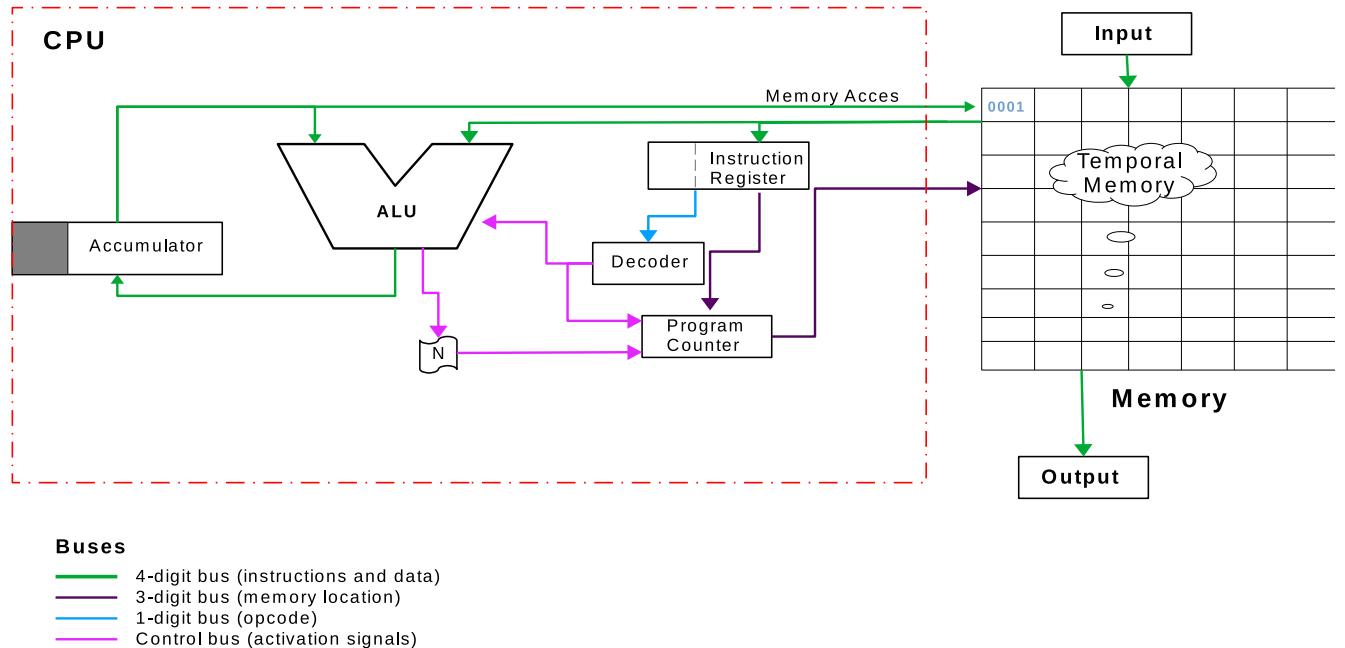


Figura 2.3: Arquitectura de CARDIAC, concepto original Bell Labs & Jorge Vasconcelos

parecido pasa con los ciclos en *CARDIAC*. Un ciclo empieza cuando el *Program Counter* o contador de programa apunta hacia una dirección, la primera sería la #00, continua cuando la CPU toma el contenido de esa dirección y por medio del bus verde la transporta hacia el *Instruction Register* o registro de instrucciones que se encarga de separar el dígito a extrema izquierda  $d_2$  para enviarlo al *decoder*(decodificador) por medio de un bus azul y que esté decodifique que código de operación es para mandarle esa indicación a la *Aritmetic Logic Unit (ALU)* o unidad aritmético lógica. Por otra parte, el mismo registro de instrucciones transfiere al contador de programa los dígitos  $d_1$  y  $d_2$  por si esté los ocupa para saltar a una dirección específica, si los ocupa el decodificador le enviará una señal por medio del bus rosa para indicarle que los debe usar (Hegelbarger y Fingerman, 1968).

Continuando con el *ALU*, esté realiza la operación que le indica el código de operación, si

el resultado que va a guardar en el acumulador es negativo manda una señal a *Negative(N)* que es una especie de booleano, el cual indica si un número que ha sido depositado en el *Accumulator* o acumulador es o no negativo. Lo último que realiza el *ALU* es almacenar el resultado de la operación en el acumulador, un único registro que tiene la capacidad de almacenar sólo cuatro dígitos(uno más que la memoria principal), y es dónde se almacenarán todos los cómputos que realice el *ALU* (Hegelbarger y Fingerman, 1968).

En caso de que la unidad aritmética requiera información de la memoria para ejecutar una instrucción por medio de los buses verdes la puede recuperar directamente, dado que si requiere información de la memoria los dígitos *d1* y *d0* de la instrucción recogida por el registro tienen la dirección de dónde puede recuperarla, y es solamente si el *ALU* obtiene el contenido de una dirección de esta forma que este será tratado como dato y no como instrucción. Si el contador de programa apunta a cualquier lugar de la memoria, el contenido de ese espacio será tratado como instrucción, a pesar de que no lo sea, lo que pudo causar inconsistencias, por lo que hay que ser muy cuidadosos con lo que escribimos (Hegelbarger y Fingerman, 1968).

De esta forma terminamos un ciclo cuando el contador de programa salta, que puede ser por qué una instrucción se lo indique, o bien porque el *ALU* terminó sus cómputos y entonces el contador de programa puede continuar a la siguiente dirección de forma incremental sin algún cambio (Hegelbarger y Fingerman, 1968).

Los periféricos también juegan un papel importante, aunque solo se conectan directamente a la memoria principal, tanto el *input* o entrada, como el *output*(salida). Si se requiere imprimir algún resultado que esté en el acumulador, este debe ser enviado primero a la memoria principal para que se pueda realizar posteriormente la operación de impresión de un espacio de la memoria principal (Hegelbarger y Fingerman, 1968).

### 2.3.2. Funcionamiento y lenguaje en CARDIAC

Ahora conocemos los elementos de este modelo y como interactúan, pero lo visitamos de cierta forma “a ciegas”, no conocemos el lenguaje que utiliza esta “máquina”, y como mencionamos antes, tanto la máquina hace al lenguaje como el lenguaje a la máquina. En la tabla 2.1 podemos ver el lenguaje que se usa en código, máquina(Código de operación) y en

ensamblador(Mnemotecnia) para CARDIAC que es mucho más fácil de entender, analizaremos cada una de las instrucciones presentadas originalmente en (Hegelbarger y Fingerman, 1968) para entender la razón de su existencia.

Código de operación	Mnemotecnia	Definición
0	INP	(INPUT) Guardar tarjeta en memoria.
1	LDA	(LOAD) Cargar en el acumulador información de la memoria.
2	ADD	(ADD) Sumar al contenido del acumulador contenido en la memoria.
3	BLZ	(Branch if Less than Zero) Saltar si la información en el acumulador es menor que cero.
4	SHF	(SHIFT) Mover de izquierda y/o derecha el contenido del acumulador.
5	OUT	(OUTPUT) Escribir en la salida el contenido de la memoria.
6	STO	(STORE) Guardar información del acumulador en la memoria.
7	SUB	(SUBSTRACT) Restar información de la memoria al acumulador.
8	JMP	(JUMP) Saltar y guardar el valor del program counter en la dirección #99.
9	HLT	(HALT) Detener la ejecución del programa y reiniciar el contador del programa.

Tabla 2.1: Lenguaje de programación de *CARDIAC*.

Para empezar necesitamos establecer una conexión entre el usuario y la máquina, que el usuario a través del dispositivo de entrada pueda darle información, con el código *INP* podemos indicar el ingreso de información a la memoria principal, por ende con la instrucción *012* estamos indicando que se guarde el dato que ingresa el usuario a través del dispositivo de entrada en la dirección de memoria 12, notaremos que en 9 de las 10 instrucciones los últimos dos dígitos hacen referencia a la dirección. Posteriormente, necesitamos llevar información de la memoria al acumulador para realizar operaciones sobre ella, esto lo realizamos con el código *LDA*, por lo tanto, con la instrucción *112* indicamos que se cargue en el acumulador la información almacenada en la dirección de memoria 12. Supongamos que el valor que ingresamos, y que fue guardado en *#12* es *005*, entonces *005* será cargado en el acumulador. Ahora, si lo que queremos es incrementar el valor del acumulador en una unidad podemos

usar el contenido de la dirección #00, dónde se encuentra el dato *001*, mismo que podemos utilizar en la instrucción *ADD* de la siguiente forma: *200*, que indica que hay que sumar el contenido de la dirección #00 a lo que tiene el acumulador, obteniendo un *006* en el acumulador.

Dirección #	Código máquina	Ensamblador	Estatus acumulador
20	012	INP 12	000
21	112	LDA 12	005
22	200	ADD 00	006
23	712	SUB 12	001
24	712	SUB 12	-004
25	200	ADD 00	-003
26	325	BLZ 25	000
27	200	ADD 00	001
28	880	JMP 80	001
29	212	ADD 12	012
30	432	SHF 21	020
31	613	STO 13	020
32	513	OUT 13	020
33	900	HLT 00	020

Tabla 2.2: Programa principal (In-Util).

Para seguir analizando los códigos de operación disponibles tenemos un código en la tabla 2.2 donde podremos visualizar los códigos, su equivalente en ensamblador y como afectan al acumulador. Este programa empieza en la dirección #20, si nos damos cuenta las primeras tres instrucciones son las que revisamos en el párrafo anterior, y el estatus del acumulador expresa precisamente el valor que se mencionaba, el *006*.

Estamos ubicados en la dirección #22, donde nos moveremos a la #23 en la que ocuparemos la instrucción de *SUB* para restar, restaremos el número que cargamos en la dirección #12 a lo que tenemos en el acumulador, de hecho lo haremos dos veces seguidas para conseguir un número negativo, él *-004*, si lo viésemos en operaciones matemáticas la operación sería la siguiente:  $6 - 5 - 5 = -4$ . Los números negativos en este modelo son especiales, como se habrá observado coloque el signo justo a la extrema izquierda, esto es porque para simplicidad del modelo los números negativos o positivos ocupan el mismo espacio, el signo siempre estará más a la izquierda que cualquier otro dígito, como las direcciones no pueden ser negativas no nos podemos encontrar con un caso del estilo *1-42*, solo con casos como los

del ejemplo, de modo parecido el modelo se toma la libertad de tomar al cero como positivo, situación que será importante con las siguientes instrucciones.

Nos movemos a la dirección #25 en la que se suma un uno al acumulador y en la dirección #26 ocupamos el código *BLZ* en la instrucción 325 para obtener un salto condicional, es decir, si el contenido del acumulador es menor a cero salta a la dirección #25, por lo que se formara un bucle en el que el acumulador cambiara de valor de uno en uno hasta llegar a cero después de cuatro vueltas, momento en el que el condicional dejara avanzar al contador de programa a la siguiente instrucción y no forzando el salto a #25, pues, el valor del acumulador ya es positivo(para los fines del modelo).

Lo que sigue es ver el funcionamiento de la operación *JMP*, para esto continuamos con la ejecución, vemos que en la dirección #27 se le suma un uno al acumulador para tener en el acumulador un 001 y posteriormente ya está la operación de salto, que realiza uno a la dirección #80, dónde se encuentra una **subrutina**, un programa “pequeño” generalmente usado por otros para reproducir un resultado. En este caso lo que hace es sumar un seis a lo que tenga el acumulador cuando el programa origén salte. Como nosotros tenemos un uno en el acumulador y queremos sumarle seis, la forma más fácil de hacerlo es saltando a esa subrutina que regresara a la dirección siguiente cuando termine, la #29.

Esté funcionamiento, lo podemos lograr fácilmente debido a que la instrucción *JMP* al momento de saltar guarda la dirección de memoria de la cual salta en la dirección #99, con el sufijo 8. Si miramos la tabla 2.3 dónde se encuentra está subrutina notaremos que lo primero que hace es cargar en el acumulador el contenido de la dirección #99, que en esté caso es un 828, pues salto desde la dirección #28, posterior se le suma un uno para así tener la dirección #29, que es a la que tiene que regresar la subrutina cuando termine, para lograrlo en la dirección #82 indica que se guarde el resultado del acumulador en la última dirección de la subrutina, así cuando terminen todas las adiciones saltará a la dirección #29.

Como apunte importante, si leyeron el manual de *CARDIAC* se habrán dado cuenta de que hay una ligera diferencia con la instrucción *Jump*, pues en el manual dice que guarda la dirección de la que salto más uno, es decir si salto de la dirección #28 lo que se guardará en #99 será la dirección #29, un 829. Lo que en nuestro ejemplo particular ahorraría código, pero por los usos que se le dará en los siguientes modelos decidí que sería más eficiente como

la he definido en esta sección, y esto es porque se tiene más precisión al tener la información más pura de dónde salto el programa. Y por supuesto, ambas definiciones pueden llegar a los mismos resultados con algunos cambios en la codificación.

Dirección #	Código máquina	Ensamblador	Estatus acumulador
80	199	LDA 99	828
81	200	ADD 00	829
82	690	STO 90	829
83	100	LDA 00	001
84	200	ADD 00	002
85	200	ADD 00	003
86	200	ADD 00	004
87	200	ADD 00	005
88	200	ADD 00	006
89	200	ADD 00	007
90	829	JMP 29	007

Tabla 2.3: Subrutina para sumar varios unos.

Continuando con el programa principal de la tabla 2.2 y habiendo regresado de la subrutina en la dirección #29 con un valor en el acumulador de *006* tenemos una suma que deja el valor del acumulador en *012*, valor que será muy interesante para analizar el siguiente código de operación, *shift/SHF*. Deje esté código para casi el final por qué es la menos intuitiva, no es una de las operaciones básicas que usualmente tenemos en mente; sin embargo, cumple un papel fundamental, en las computadoras binarias cumple un papel aún más crucial por las operaciones entre bits que se pueden realizar, pero en una decimal también aporta mucha flexibilidad y eficiencia en el uso de memoria para una gran variedad de operaciones.

En la dirección #30 vemos que el efecto de aplicar esta operación con la instrucción *432* cambia el número *012* por el *20*. Esto sucede por qué el dígito *d1* indica cuantos lugares a la izquierda se desplazará el valor del acumulador, dejando en 0 los espacios vacíos, mientras que el dígito *d0* indica cuantos lugares a la derecha se desplazará. Aquí será muy importante la consideración especial de espacio que tiene el acumulador respecto a las celdas de la memoria principal, pues para evitar el desbordamiento accidental el acumulador siempre tendrá **un dígito más** del que las celdas de la memoria principal tengan, en este caso el acumulador puede almacenar hasta 4 dígitos.

Si consideramos lo anterior y vemos la instrucción *432* lo que nos indica primero es mover

tres lugares a la izquierda el valor del acumulador, 0012, dando como resultado 2000, pues el 1 queda fuera del espacio del acumulador, por lo que esté número se pierde para siempre y no se podrá recuperar. Si le aplicamos el desplazamiento de dos lugares a la derecha que se nos indica, terminaremos con 0020 en el acumulador, como podemos ver el 1 se perdió, y aunque desplacemos en el sentido contrario, si esté ya se quedó fuera del espacio del acumulador no podrá regresar. Para simplificar y como prácticamente no se usa el cuarto dígito dejamos indicado el resultado solo como 020 en la tabla 2.2.

Para finalizar el programa tenemos en #31 la instrucción para guardar la información del acumulador(*STO*) en la dirección de memoria número #13 y, posteriormente con el código *OUT* se establece la conexión con el dispositivo de salida para exportar los resultados que se guardaron en #13, debido a que la instrucción 513 indica con sus últimos dos dígitos la dirección de memoria de la cual se tomará la información. Finalmente, el programa se termina con el código *HLT*, marcando el final del programa y se reinicia el contador del programa a la dirección #00, pues es lo que se indica con los dígitos *d1* y *d0* de la instrucción 900. Después de este repaso podemos constatar que en casi todos los códigos de operación los dígitos *d1* y *d0* hacen referencia a una dirección de memoria, salvo por *SHF* en la cual tienen un funcionamiento especial.

Con este lenguaje, simple, pero eficaz, podemos construir cualquier programa que queramos, puesto al tener ciclos, condicionales y la posibilidad de realizar las operaciones aritméticas básicas (suma y resta), podemos decir que es Turing completo, por lo que nuestra única limitante es la memoria.

# Capítulo 3

## Evolución del Modelo

Como pudimos ver con anterioridad CARDIAC es sumamente útil para explicar aspectos importantes de la computación y el como se organizan sus componentes, sin embargo no podemos negar que a día de hoy es un tanto insuficiente para explicar las computadoras más modernas que tienen una concurrencia y un paralelismo sin los cuales no entenderíamos a las computadoras. No podemos imaginar una computadora en la cual no podamos ejecutar más de un proceso a la vez, por ello una evolución del modelo creado en los años 60 por (Hegelbarger y Fingerman, 1968) es necesaria. Pero ello implica diferentes retos, retos que se abordarán en las siguientes secciones.

Pero no solo en el apartado de diseño y organización del modelo se tendrá una actualización, sino también en la forma de presentarlo. Aprovechando las facilidades de algunos lenguajes de programación he realizado una simulación en Java para mostrar las mejoras realizadas al modelo original y que sea más sencillo mostrar la interacción de los distintos componentes de una computadora. La idea detrás de esta simulación no es solo emular los comportamientos de CARDIAC en una computadora actual para ejecutar algunos programas, sino realmente realizar una **maquina virtual**, es decir un software que represente el modelo tanto a nivel Hardware como a nivel Software de manera virtual. En la imagen 3.1 tenemos la pantalla de inicio donde se puede seleccionar la maquina virtual que queremos probar, como se podrá notar he agregado la iónica *E*(de electrónico), que se puede apreciar en muchos productos a día de hoy, como sufijo de las tres maquinas virtuales para resaltar su aspecto electrónico distante del que tenía en los años 60. En la siguiente sección veremos al-

gunos ejemplos de programas de CARDIAC ejecutados en la maquina virtual *E-CARDIAC*, para posteriormente adentrarnos en las evoluciones del modelo.



Figura 3.1: Inicio para selección de maquinas virtuales

### 3.1. E-CARDIAC : Electronic CARDboard Illustrative Aid to Computation

Después de entrar al software de simulación y ver la pantalla de bienvenida como se ve en la figura 3.1 lo que sigue es elegir la maquina que queremos probar, en este caso comenzaremos con la versión que es prácticamente el modelo original llevado a un software de simulación. En términos generales las tres maquinas tendrán un esqueleto similar, por lo que una vez que conozcamos el funcionamiento de esta primera será muy sencillo entender el funcionamiento de las otras dos aunque estás tengan más componentes. Para empezar será bueno visualizar el diagrama de la arquitectura de CARDIAC, diagrama que se puede apreciar en la figura 3.2 y que se irá comentando de acuerdo a los componentes a los que se haga referencia, pero como observación general se puede apreciar el CPU a la izquierda, la memoria a la derecha con una descripción de como son sus dígitos, y en la parte inferior las diferencias entre los buses que la componen así como el contenido de las celdas con información predefinida.

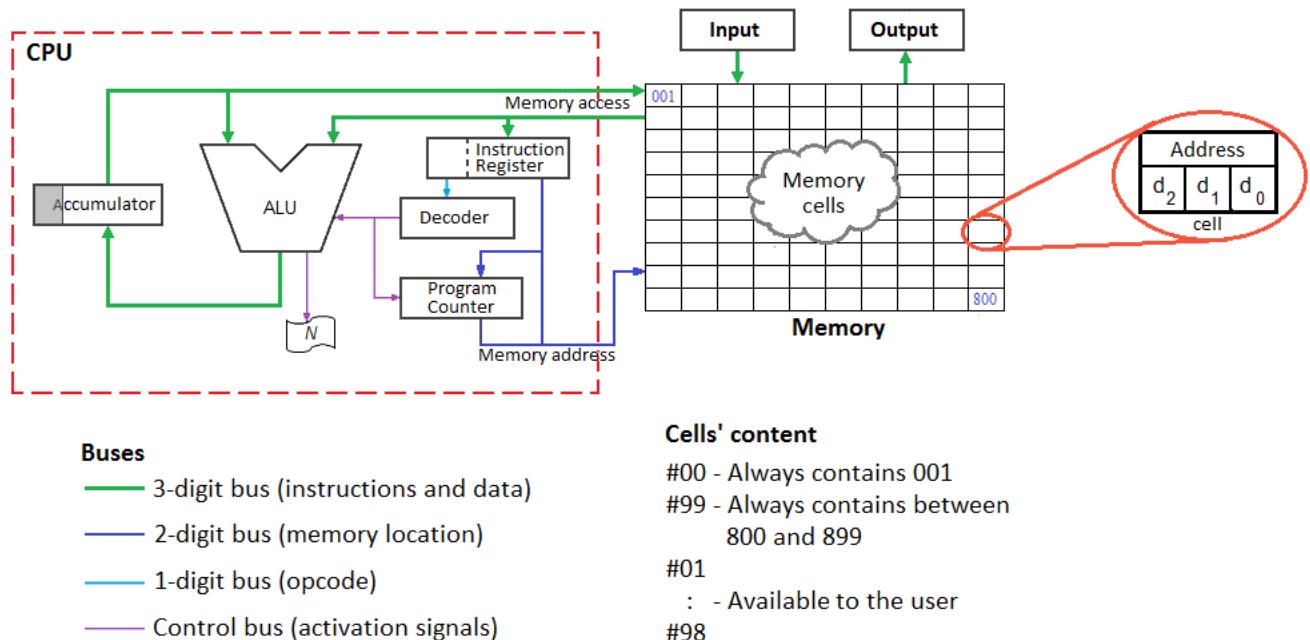


Figura 3.2: Arquitectura de CARDIAC por Jorge Vasconcelos, 2018

En la figura 3.3 podemos ver el esqueleto de lo que será nuestra maquina que aún se encuentra apagada, por que al entrar lo que nos encontramos es una maquina apagada, que

podemos encender al dar clic sobre el botón *start*. Pero antes de dar clic ahí podemos ver que en esa barra principal tenemos varias opciones, a extrema izquierda está un símbolo de casa y un botón que dice *CARDIAC Systems*, que nos permitirá regresar a la pagina de inicio si así lo deseamos. Continuando en el centro tenemos tres botones, el que ya vimos para encender la maquina, otro para pausar, algo poco común en una maquina de verdad, pero bastante usual en los software de maquinas virtuales, puesto que nos permite detener el funcionamiento completo sin afectar los procesos internos, analizar el estado de la maquina y después continuar como si nada hubiera pasado; el tercer botón es para reiniciar la maquina, un clásico en las computadoras.

En la parte derecha de la misma imagen podemos ver dos casillas, una con un 100 y la otra con la palabra “Normal”, estas son dos listas desplegables, la primera permite elegir la cantidad de celdas con las que la maquina funcionara, es decir la memoria disponible, y la segunda es para elegir la velocidad de la maquina, de esa forma podemos decidir cuanto va a tardar cada ciclo en ser completado con el fin de observar el comportamiento de la maquina. En la figura 3.4 podemos ver las listas desplegadas con las opciones que tienen disponibles, como podemos ver hay distintas velocidades para ver con más detalle el proceso en *slow* o si queremos el resultado de inmediato está la opción *instant*. En el caso de la memoria el funcionamiento es más interesante, pues no es solo una configuración externa a la maquina, sino que afecta directamente a la arquitectura y al lenguaje, puesto que con 1000 celdas el lenguaje debe cambiar para recibir direcciones de 3 dígitos, aunque el cambio en el lenguaje sería solo ese, por lo demás sólo sería la adaptación, mismo caso que para 10,000 celdas. Al elegir la cantidad de memoria y luego encender la maquina está se configura en su arquitectura para trabajar con esa cantidad de memoria y recibir instrucciones con direcciones más grandes.

Continuando con los elementos que tiene la maquina virtual podemos ver que en la parte izquierda de la figura 3.3 cuatro secciones, los componentes principales de la maquina en la *Central Processing Unit*, el estado de la maquina en la parte del *machinne status*, un apartado para las salidas en el *output*, y un apartado en blanco que servirá como cola de espera para la ejecución de las instrucciones *queue*. En la primer sección, que es el *CPU* podemos notar que está el registro de instrucciones(*instruction register*) que contendrá la

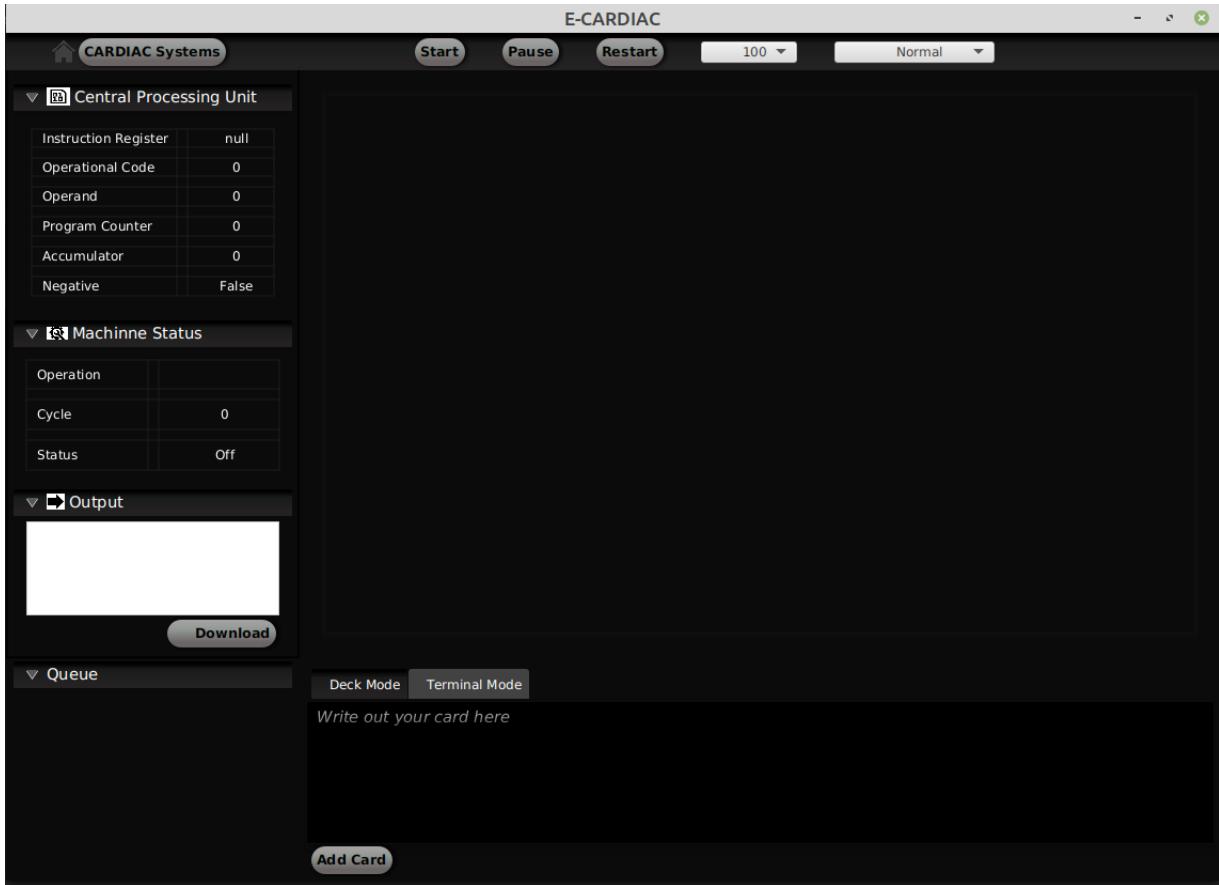


Figura 3.3: Pantalla de inicio de E-CARDIAC

instrucción que viajo a través del bus verde desde la memoria al *CPU*, después el código de operación(*Operational Code*) y el operando(*Operand*) que son los resultados que presenta el decodificador(*decoder* en la figura 3.2) después de que una instrucción pasa por el. Posterior se encuentra el contador de programa (*Program Counter*),el componente que se encarga de hacer avanzar la lectura de instrucciones sobre la memoria, el número que contenga el contador de programa es la dirección en la que se encuentra el “puntero” de la maquina, es decir que el contenido de tal dirección que es “apuntada” por el contador será transmitido al *CPU*. Más abajo se encuentra el acumulador(*Accumulator*) y la bandera para saber si un número es negativo(*Negative*), el acumulador contendrá los resultados que arroje la unidad aritmético-lógica(*ALU* en diagrama de CARDIAC) o bien algún valor que sea cargado directamente desde memoria por alguna instrucción como *LDA*.

En la siguiente sección, *Machinne Status* tenemos elementos externos a la arquitectura de la computadora, y que más bien nos definen el estado de la maquina en cada momento, en



Figura 3.4: Listas desplegables de E-CARDIAC

cada ciclo que pasa nos indica directamente si la maquina está funcionando correctamente (*CARDIAC is working*), pausada(*CARDIAC is paused*) o si por el contrario dejó de funcionar(*CARDIAC is dead*). Esté último puede ser causado por diversas razones que hagan que la maquina realice operaciones indebidas, como pasar un negativo como instrucción, para lo cuál la maquina no tiene respuesta y lo único que hará es detenerse. También en esta sección está un apartado que indica la operación que estará ejecutando el *CPU*.

La tercera sección muestra la salida en forma de lista, como podemos ver en la figura 3.5, emulando lo que eran las salidas de las primeras computadoras que podían imprimir resultados en cintas perforadas, permitiendo así al usuario ver el resultado de su programa; en esta misma sección está un apartado para “descargar” la cinta y guardarla como un archivo de texto, que por defecto se guarda en la carpeta de descargas.

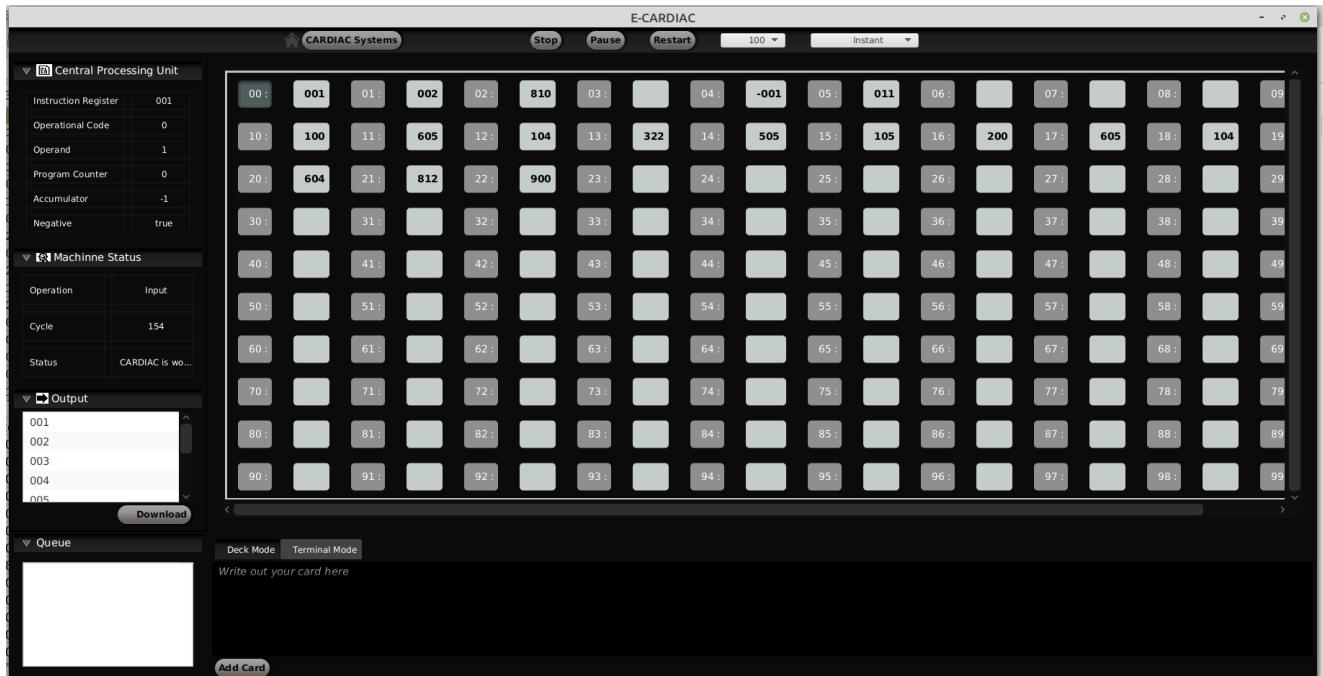


Figura 3.5: E-CARDIAC Muestra de Output

Toda la sección inferior está relacionada puesto que en la izquierda está la lista de instrucciones en cola por ser leídas(en *Queue*), como en la figura 3.8 se muestra, estás instrucciones el usuario podrá añadirlas a la cola por parte del *Deck Mode*, se podrán cargar desde ahí con solo escribirlas en forma de lista y dar clic en agregar tarjeta(*Add Card*), como se ve en la figura 3.6 dónde podemos ver una lista de instrucciones previo a ser añadidas a la cola. Por otra parte si se desea añadir instrucciones/datos de forma individual, uno a uno, se cuenta con la otra pestaña que dice *Terminal Mode* y del cuál podemos ver un ejemplo de la carga en la figura 3.7, pero para hacer uso de esta pestaña es necesario que exista una instrucción que indique, como en el ejemplo, que dice que se espera una entrada en una celda, en caso contrario no se permitirá agregar datos de esa manera, puesto que esté modo transfiere directo los datos o instrucciones a la memoria en la posición solicitada, a diferencia del modo de tarjetas(*Deck Mode*) que siempre primero las manda a la cola de espera, que las hará pasar posteriormente, cuando se solicite, a la memoria principal.



Figura 3.6: E-CARDIAC Carga masiva por tarjetas

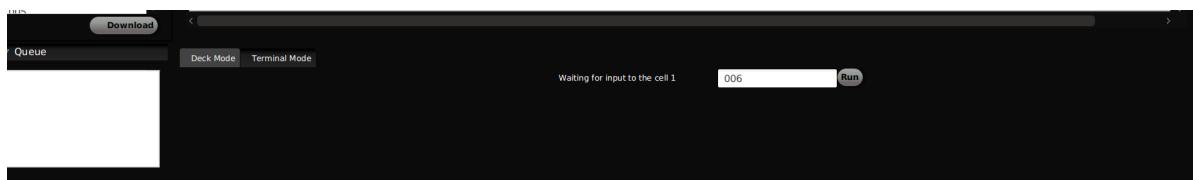


Figura 3.7: E-CARDIAC Carga individual de instrucciones

Y ya en varias de las imágenes mencionadas en párrafos anteriores se mostró como se ve la memoria principal, pero vemos ahora la figura 3.9 para ver la maquina cuando acaba de encender y centrémonos en la memoria principal, compuesta de 100 celdas numeradas del 00 al 99 con los datos por defecto que tienen en esas especies de memorias ROM. En color blanco con fondo gris están todas las direcciones de memoria, y con un fondo blanco pero contenido en negro los datos que contiene cada dirección, y para diferenciar la dirección de

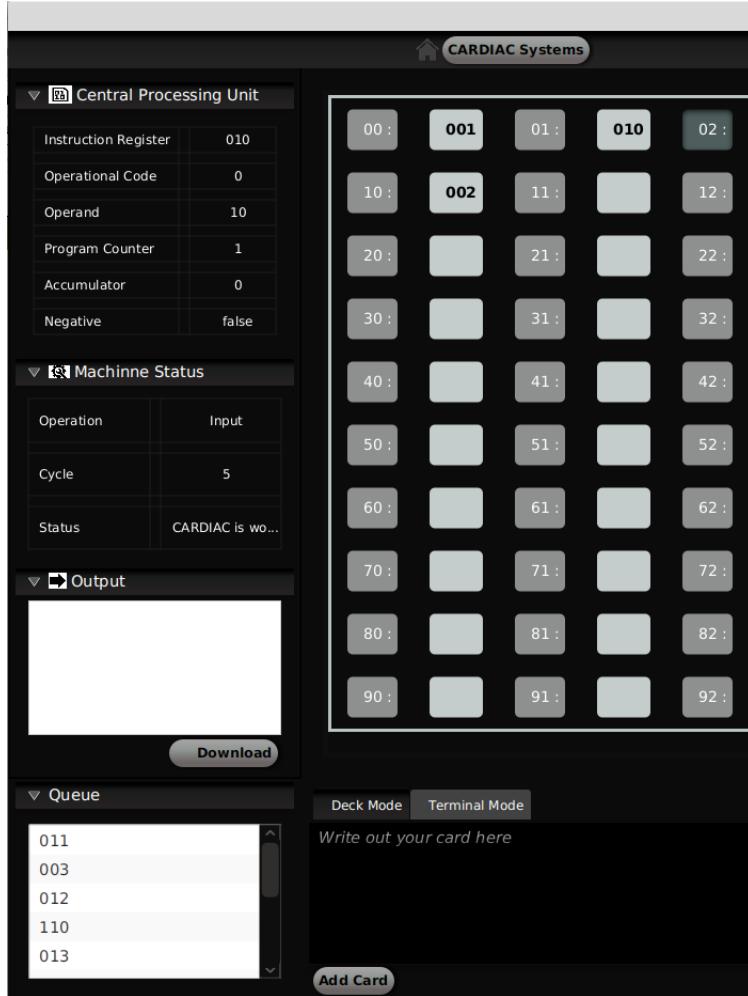


Figura 3.8: E-CARDIAC Lista de espera o cola de instrucciones/datos

la celda de la cuál el contenido está siendo transmitido al *CPU* de la maquina se tiene el color azul marino, en el caso del ejemplo nos indica que se está leyendo la instrucción en la dirección #00.

De esta forma terminamos el recorrido por la maquina virtual de **E-CARDIAC** que nos permitirá experimentar con más soltura la programación en lenguaje ensamblador, el lenguaje ensamblador particular de esta maquina, además de que tenemos la posibilidad de hacer crecer su memoria principal aumentando así las posibilidades que nos da esta maquina. Y precisamente la siguiente evolución del modelo requerirá de un aumento en la memoria de CARDIAC que es muy fácil de aplicar sobre la maquina virtual puesto que desde el principio se considero esa posibilidad, y que podremos estudiar con más profundidad en la siguiente sección.

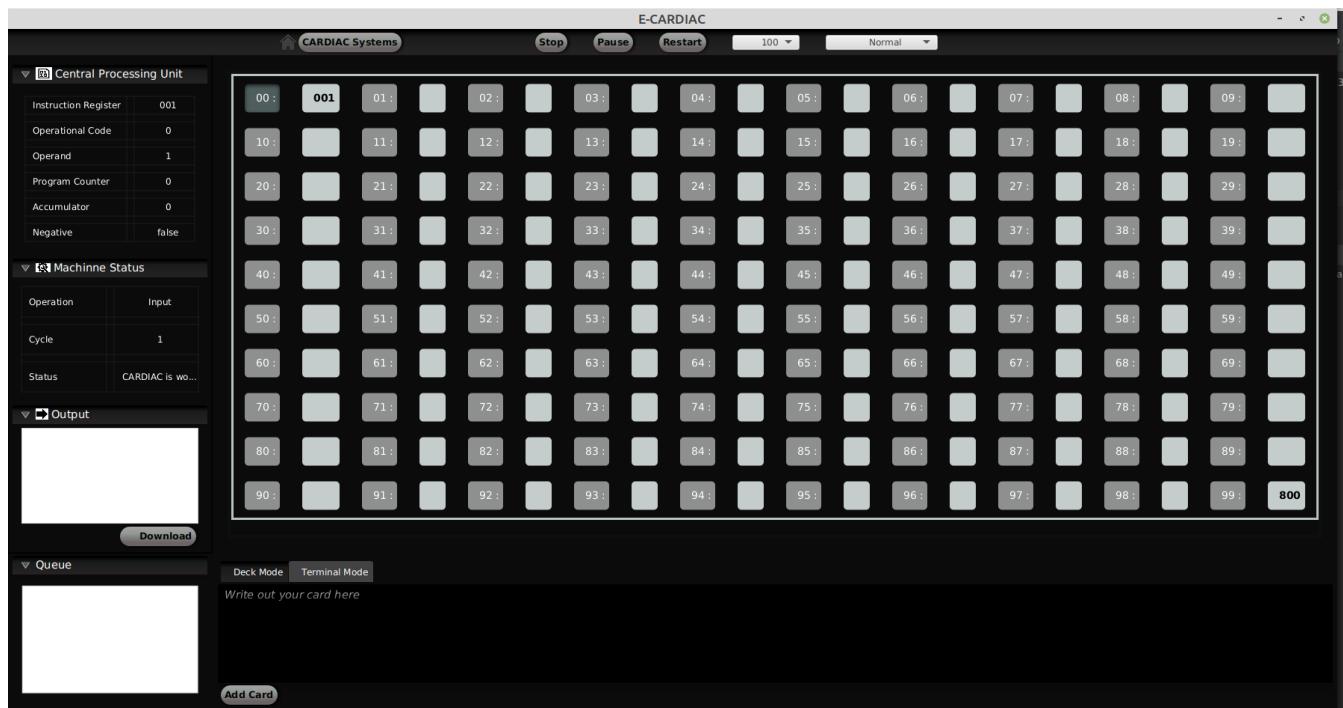


Figura 3.9: E-CARDIAC después de iniciar sus funciones

## **3.2. E-CARDIAC C : Electronic CARDboard Illustrative Aid to Concurrent Computing**

El nombre es un homenaje al lenguaje de programación **C**, agregando esa “C” al final del nombre que tenía el modelo anterior para expresar que la diferencia entre estos dos es la concurrencia. *E-CARIDAC C*, ayuda ilustrativa de cartón electrónico para la computación concurrente, por sus siglas en inglés, tiene la intención de ser un modelo que represente la organización de una computadora que puede tener operaciones concurrentes y poder ver como interactúan las distintas partes de la computadora para lograr la concurrencia. Buscamos la concurrencia a nivel procesos en este caso, dado que generalmente es utilizada en las computadoras comunes. Esto nos lleva a tener que hacer varios cambios en el diseño original del modelo, agregar mecanismos en el Hardware, en el Software y diseñar una especie de sistema operativo para poder lograr la concurrencia de procesos.

### **3.2.1. Necesidad de un sistema operativo**

El primer paso es entender la necesidad de tener un sistema que administre los recursos de la maquina, sin ello podemos construir muchos programas, pero la ejecución deberá ser individual a nuestra disposición, de cierta manera seríamos nosotros mismos ese sistema de administración. Si bien en (Hegelbarger y Fingerman, 1968, p. 42) nos presentan un sistema de carga de programas para el modelo original, un bootloader diseñado para poder escribir en “tarjetas” un programa y que se vaya añadiendo a la memoria de CARDIAC sin la necesidad de literalmente colocar en los espacios de memoria de CARDIAC las instrucciones, esto aún nos deja con la tarea de decidir que programa va primero. Así que partiendo de este punto podemos concluir que el programa que realice esa tarea será un sistema operativo mínimo, y necesita el sufijo de mínimo por que la otra característica que define a un sistema operativo es ser una capa de abstracción entre la maquina y otros programas, como puede ser un teclado o un sistema de audio que estén programados para funcionar sobre un sistema operativo y no directamente sobre la maquina, por ende la maquina podría cambiar y el programa seguir funcionando. En este caso este sistema operativo que se diseñará no tiene la intención de ser

una capa de abstracción tan marcada, tendrá algunos aspectos que pueden ser considerados un capa de abstracción entre los programas y la maquina, pero no lo será completamente. Por tal razón lo nombraremos como sistema operativo mínimo o SOM en el resto del texto.

Entendiendo un programa que realice tal tarea parece evidente que el espacio de memoria del modelo original será insuficiente, por lo que una de las necesidades derivadas del SOM será la ampliación de la memoria, lo que llevará a un cambio también en el lenguaje que esa diseñado específicamente para direcciones de dos dígitos. Y seguramente la implementación de un sistema operativo requerirá de algún sistema de almacenamiento secundario para el programa, de forma que no esté cargado directamente en la memoria principal como si de una memoria ROM se tratará, esto derivaría en la necesidad de conectar ese almacenamiento secundario con la memoria principal y por ende aumentar la cantidad de buses. Además al ser un programa que requiera unos privilegios de control de la maquina más elevados quizás se necesiten piezas de hardware que en su versión original CARDIAC no necesitaba.

### **3.2.2. Mejoras necesarias en el Hardware**

Para solucionar las necesidades mencionadas más arriba decidí aumentar las celdas de memoria de 100 a 1000, de forma que cada dirección posible tendrá tres dígitos, de forma que un número completo de cuatro dígitos puede representar tanto el número en sí mismo, como un código de operación acompañado de una dirección. Por ende los buses deberán ser capaces de transmitir más dígitos que en el modelo original, en la figura 3.10 podemos observar en la parte inferior izquierda como serán los buses; los de color verde, que son los de mayor capacidad, transmitirán instrucciones y datos, los que están en color morado sólo las direcciones, mientras que los que están en color azul sólo los códigos de operación. Adicionalmente hay unos buses en color rosa que transmitirán señales de activación. En la misma imagen se puede apreciar que al principio y al final de la memoria las celdas no han cambiado mucho, al principio está un valor que será inmutable que tiene el mismo significado que en el modelo original salvo que con un cero extra por el crecimiento en el tamaño de la memoria, y la misma situación la tiene el valor del final de la memoria, que solo podrá contener un valor el cuál inicie con un 8, y que por defecto será 8000, el sufijo que sigue al código de operación 8 será la dirección de memoria de dónde se tomo una instrucción de

salto que fue ejecutada.

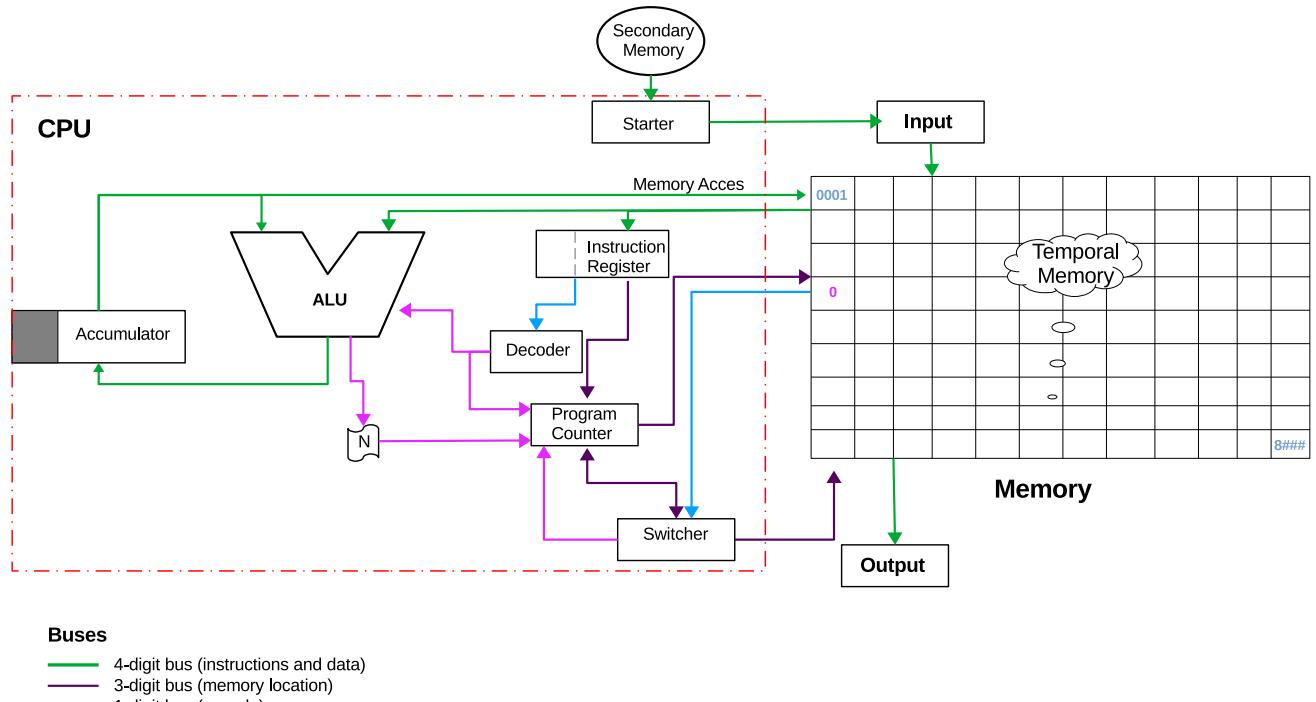


Figura 3.10: Diagrama de Arquitectura de E-CARDIAC C

Esto permite mantener varios elementos del funcionamiento del modelo original sin cambios, pero podemos notar tres de elementos nuevos, que a su vez incrementan el número de buses presentes en el modelo. Detengámonos primero en el que está conectado al *input* en la parte superior del centro, el que tiene por nombre **starter**(iniciador) y que a su vez está conectado con otro elemento nuevo llamado **secondary memory**(memoria secundaria). La memoria secundaria es la representación de lo que vendría siendo el disco duro en las computadoras, pero más limitada por que sólo sería de un almacenamiento muy puntual para el sistema operativo, que tendría que hacerse por fuera de la computadora, y siendo más bien como una tarjeta con un programa específico que está conectada a través del **starter** para que apenas se inicie la maquina lo primero que haga es empezar a agregar las instrucciones

guardadas en la memoria secundaria a la pila de ejecución que recibe el *input*. De esta manera podemos mantener simple el modelo, pero establecer una conexión con otra fuente de información diferente a la memoria principal, y poder colocar el sistema operativo mínimo de una forma más natural sin que esté ahí desde siempre como si fuera parte de la memoria ROM.

El otro integrante de este nuevo modelo es el que tiene por nombre **switcher**(cambiador) que está en la parte inferior derecha del área del CPU, este elemento es la solución al problema que tiene la inclusión de un programa que necesita tener los privilegios de poder tomar el control de las ejecuciones, cederlo a otros programas, pero también recuperarlo en un momento particular. Como se puede observar en la imagen tiene 4 diferentes conexiones, siendo uno de los que más tiene, y su función radica en que cada **N** ciclos dar manda una señal(con el bus rosa) al contador de programa para que salte a la dirección de inicio del sistema operativo, dirección que tiene que ser establecida en la configuración de la misma maquina. Esto lo hace siempre y cuando en la dirección #003 el valor que esté ahí sea 1, que indica que es tiempo de ejecución del usuario y por tanto cada N ciclos los recursos regresan al SOM, si el valor fuese un 0, como es por defecto, no haría esos saltos, puesto que significa que los recursos los tiene el SOM, el cuál puede disponer de ellos sin límite al tener todos los privilegios posibles. Esta conexión se hace con un bus que es azul pues solo requiere de un dígito, además contiene una conexión bidireccional al contador de programa que le permite contar los ciclos que han sucedido e indicarle al contador de programa a que dirección tiene que saltar cuando se trata de regresar el control de los recursos al SOM, aquí se requiere un bus de color morado puesto que requiere de una capacidad de 3 dígitos. La otra razón de esta conexión bidireccional es que el **switcher** recibe del contador programa la última dirección apuntada por el proceso del usuario, antes de que fuera ejecutada puesto que para ese momento si se llegaron a los N ciclos se da la instrucción de regresar los recursos al SOM y al mismo tiempo la instrucción es transmitida al **switcher** para que este por medio de su conexión a la memoria principal la coloque en una dirección especial de la cual el SOM podrá recuperarla para después cuando regrese los recursos a es proceso pueda continuar justo en el punto que había dejado.

El sistema operativo mínimo en su mismo proceso ejecuta otros procesos y por ende

realiza los saltos necesarios para ceder los recursos, pero antes de ejecutarlos modifica el valor de la dirección #003 a 1 para que el *switcher* pueda regresar el control al SOM después de N ciclos aproximadamente, para lograr la consistencia en la ejecución de procesos en el momento que la bandera es cambiada el contador del *switcher* se reinicia. Aclarando la parte de “N ciclos aproximadamente” es por que una vez que es cambiado el valor a 1 por el SOM en la dirección que funge como **bandera de saltos** aún ejecutará un par de instrucciones propias del SOM para lanzar otros procesos, por lo que en realidad el proceso del usuario tendrá  $N - 2$  ciclos disponibles.

Por supuesto con todos estos cambios hechos para poder instalar un sistema que administre la ejecución de procesos concurrentemente el lenguaje también requiere modificaciones, en su mayoría mínimas, pero otras que van orientadas totalmente a funcionar con un sistema operativo.

### 3.2.3. Cambios en el lenguaje

Como ya leímos en secciones pasadas el lenguaje de una maquina muchas veces es consecuencia de las posibilidades que el hardware proveé a la maquina, pero también el lenguaje define ciertas necesidades del hardware, por lo que en la construcción de una maquina se necesita pensar en todos los elementos que van a interactuar juntos. Para este caso agregamos un nivel extra, la consideración del sistema operativo mínimo, pues como podemos notar en la tabla 3.1 uno de los únicos dos cambios que hubo es en la instrucción *halt*, está esta diseñada ahora para funcionar con un sistema operativo, pues por si sola no tendrá las capacidades completas que si tendrá con un SOM. En la configuración inicial de la maquina se definirá la dirección que está definida como “área de borrado” para el SOM, y es con este detalle que podemos considerar que nuestro SOM es en cierto punto una capa de abstracción entre el código maquina puro y lo que harán los procesos, que será comunicarse con el SOM y el se comunicara con la maquina, en este caso para detener el proceso.

El otro cambio a destacar es en la instrucción *shift*, considerando el dígito a extrema izquierda como *d3* y el resto de manera descendente hasta llegar a *d0* a extrema derecha, tenemos que el dígito *d3* tendrá el *opcode* como en todas las demás instrucciones. Pero para está solo importarán los dígitos *d1* y *d0*, puesto que el *d2* no representará nada en la

Código de operación	Mnemotecnia	Definición
0	INP	Guardar tarjeta en memoria.
1	LDA	Cargar en el acumulador información de la memoria.
2	ADD	Sumar al contenido del acumulador contenido en la memoria.
3	BLZ	Saltar si la información en el acumulador es menor que cero.
4	SHF	Mover d2 veces a la izquierda el número y d1 veces a la derecha el número.
5	OUT	Escribir en la salida el contenido de la memoria.
6	STO	Guardar información del acumulador en la memoria.
7	SUB	Restar información de la memoria al acumulador.
8	JMP	Saltar y guardar el valor del program counter en la dirección #999.
9	HLT	Detener la ejecución del programa y saltar al área de borrado de procesos del SO, no importa el valor que los dígitos d0,d1, y d2 tengan

Tabla 3.1: Lenguaje de programación de *E-CARDIAC C*.

instrucción, de esta forma se mantiene el funcionamiento del modelo original de mover el número de izquierda a derecha una cantidad determinada de veces.

El último cambio a destacar es que el valor del *program counter* que se guardaba en #99 cuando se hacía un salto con la instrucción *jump* ahora será guardado en la dirección #999, que es la última en esta arquitectura. Pero para todas las instrucción, a excepción de *halt* y *jump*, el comportamiento será el mismo sólo que los tres dígitos de la derecha representan la dirección, lo cuál es bastante lógico dado que las direcciones ya no ocupan dos dígitos, sino tres.

Con esto tenemos el diseño completo de la maquina, su arquitectura, y su lenguaje. Ahora podemos dar el paso a escribir el sistema operativo mínimo que pueda realizar las tareas que esperamos en esta maquina, aunque por supuesto y como se dijo ya se había pensado en que características necesitaba la maquina para poder implementar un SOM y que tipo necesitaría.

### 3.2.4. Sistema Operativo Mínimo C

Conocemos ahora los aspectos esenciales que debe tener un proceso para ser un sistema operativo y las necesidades particulares que tiene el que necesitamos implementar en nuestro modelo, por ende detrás de el desarrollo de esté sistema operativo mínimo, llamado *SOMC*, está una lista de tareas que debe ser capaz de realizar para poder ejecutar concurrentemente diferentes procesos, en las siguientes secciones iremos viendo cada una de esas tareas.

El diseño de este programa lo hice pensando en una arquitectura de mil celdas, pero que pudiese ser extensible a más, para ello era necesario que las direcciones no fuesen fijas en el diseño del programa, por lo que use variables para las direcciones, de forma que en el diseño puedo tener una dirección como `#s1`, pero en la implementación esa dirección se transforma en `#801`, así si tenía una instrucción de la forma `1(s1)` su transformación será `1801`. De esta manera tenemos mucha flexibilidad al momento de escribir el código del programa, cuando comencé pensé que con menos de 100 celdas de memoria podría escribir todo el programa, por ende que iniciara en la celda `#900` parecía razonable, pero a medida que avance descubrí que no, y esta flexibilidad en cuanto a las direcciones me permitió seguir escribiendo para más de 100 direcciones sin tener que reescribir lo que ya tenía, lo único que tenía que hacer es cambiar la dirección de inicio de mi programa, si antes `#s0` era `#900` ahora sería `#800` y lo mismo pasaría para sus consecutivos.

Otra parte importante para mantener esta flexibilidad fue dividir el programa en diversos segmentos, el principal es el núcleo del sistema operativo mínimo y que lleva tal cuál ese nombre, otro es la **zona de procesos** para almacenar el contexto de los procesos que se estén ejecutando en la maquina, y además por separado está una **zona de variables** que serán variables o constantes de uso recurrente que el sistema operativo mínimo pueda guardar o consultar, y por último un segmento llamado **preámbulo** que serían las primeras instrucciones que se ejecutan cuando el SOM toma control de los recursos y prepara así ciertas variables para que cuando el núcleo del SOM esté en ejecución todas las variables estén en su correcta posición. Las ventajas de segmentar el programa es que puedo cambiar las direcciones de inicio de cada segmento una vez que ya tenga todo escrito sin afectar la estructura del código.

Adicionalmente en el núcleo del SOM también agregué una especie de segmentación en cuanto a las tareas que realiza cada sección del SOM, separandolas en: añadir proceso, actualizar proceso, borrar proceso, ejecutar proceso, y ejecución de bootloader. Para poder seguir todos estos conceptos con más claridad también diseñe un diagrama de flujo, que se puede ver en la imagen 3.11 , en el cuál cada sección está con un color diferente. Por el tamaño del diagrama es difícil apreciar los que dicen las letras, pero en cada sección que ocupemos de una parte del diagrama se le hará un acercamiento a esa zona, pero para darse una idea de las conexiones entre cada zona está imagen es de mucha ayuda.

## Nomenclatura del diseño del código

Para explicar el código del sistema utilizaré además del ya mencionado diagrama imágenes como la de la figura 3.12, dónde se puede ver el código, así como una descripción de cada instrucción. A la izquierda está el “nombre clave” de cada dirección o grupo de direcciones si es que lo necesitan, es una descripción corta acerca de esas direcciones, por ejemplo en las instrucciones coloreadas de color rosa hay un nombre clave que indica que es una bandera; en la parte del medio tenemos la dirección de memoria, después la instrucción en lenguaje maquina, y al lado la instrucción en lenguaje ensamblador, para finalizar en la parte derecha con la descripción completa de la instrucción si está fuera necesaria. Cada sección del SOM tendrá variables de direcciones diferentes, así para el preámbulo las variables que indiquen las direcciones empezaran por una *e* y continuarán de forma serial. En las columnas dónde están las instrucciones podemos ver como se usaran, si nos fijamos en la fila dónde está la dirección #e10 veremos la instrucción *1(e(0))*, que indica que se cargue el contenido de la dirección #e0 en el acumulador, en la implementación la variable de dirección sería sustituida por una dirección real de la maquina.

Así como para el preámbulo las variables de direcciones empiezan por *e*, para la zona de procesos empezaran por *p*, en la zona de variables del sistema por *c* y las del núcleo del sistema operativo empezarán por *s*. De esta forma también será rápido en el código identificar a que zona está haciendo referencia la instrucción, por ejemplo en la imagen del preámbulo en la fila de la dirección #e6 se encuentra una instrucción de la forma *2(c13)*, lo que ya nos indica que va a trabajar con un valor de la zona de variables del sistema por la letra *c*

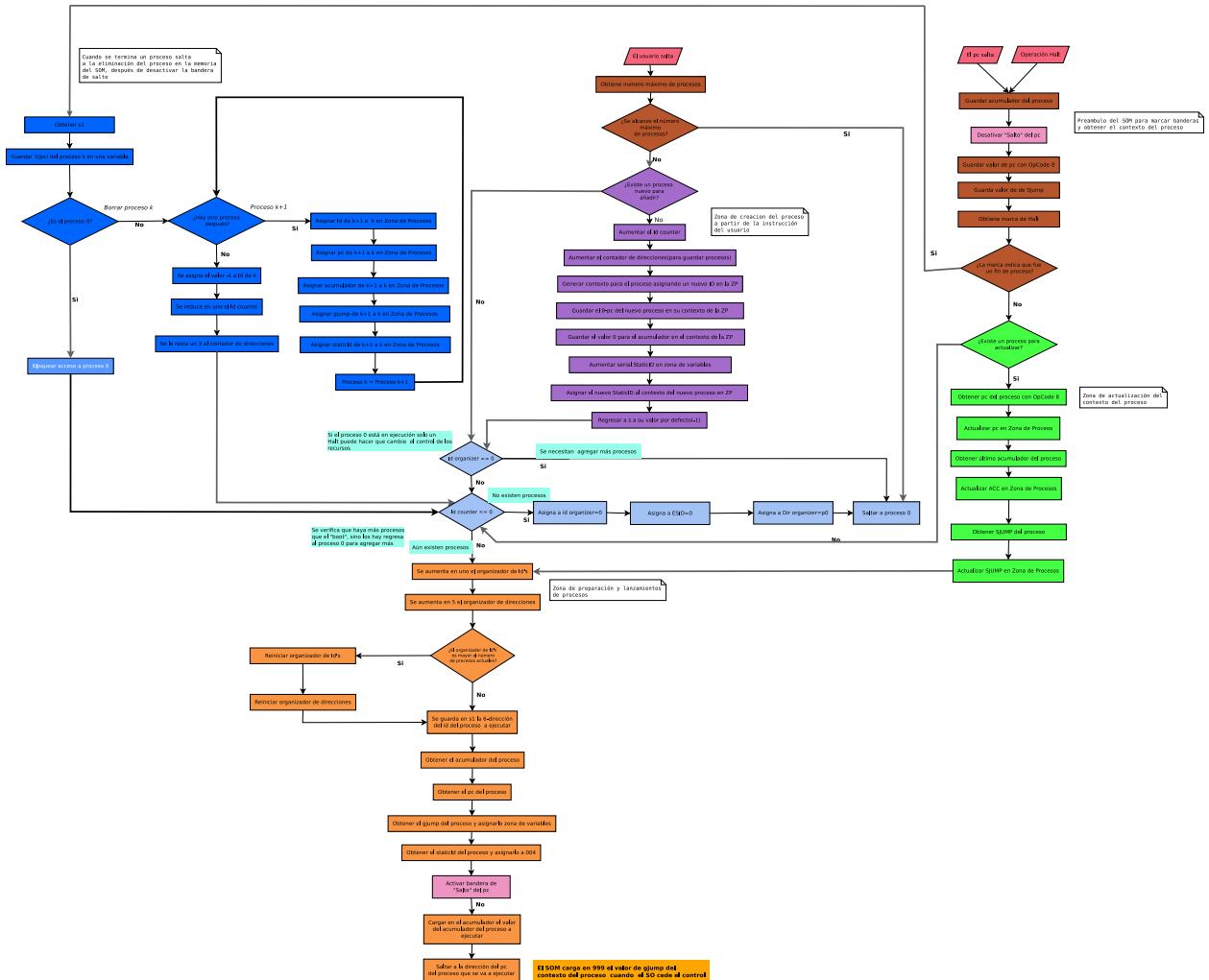


Figura 3.11: Diagrama de flujo de SOMC

que se encuentra en el interior de la instrucción, examinándola nos dice que se va sumar al contenido del acumulador el contenido de la dirección #c13, y tanto la descripción como el nombre clave nos proveerán de el contexto de la instrucción para entender su significado.

Preámbulo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>Recibe el pc</i>	e0			Le coloca la maquina -1 si la instrucción fue 9(px), si salto el sw coloca el pc
<i>Manda acc a C</i>	e1	6(c2)	STO c2	Manda a c2 el último acc del proceso antes de saltar
<i>Cambiar bandera a "No" permitir</i>	e2	1(000)	LDA 000	Con bandera==0 no se permiten saltos
	e3	7(000)	SUB 000	No se permiten saltos
	e4	6(003)	STO 003	Saltamos a actualizar
	e5	1(e0)	LDA e0	Carga en el acumulador el último pc del proceso antes de saltar
<i>Mandar pc a C</i>	e6	2(c13)	ADD c13	Le coloca al pc el op-code 8
	e7	6(c1)	STO c1	En c1 se guarda el pc con op-code 8
<i>Obtener Saver Jump</i>	e8	1(999)	LDA 999	Cargar el último valor de 999 del proceso que salio
	e9	6(c14)	STO c14	Guardar en c14 el nuevo valor
<i>Verificar marca</i>	e10	1(e0)	LDA e0	Se obtiene la marca
	e11	3(e13)	BLZ e11	Si la marca es menor a 1 se va al área de borrado
<i>Salto</i>	e12	8(s2)	JMP s2	Si la marca no es menor a 1 se va al área de actualización
<i>Salto</i>	e13	8(s19)	JMP s19	Salta al área de borrado
<i>Preámbulo para añadir procesos</i>	A	1(c16)	LDA c16	Obtiene el numero máximo de procesos
	e15	7(c4)	SUB c4	Le resta la cantidad de procesos que hay
	e16	3(000)	BLZ 000	Si acc<0 se ha alcanzado el número máximo de procesos
	e17	8(s66)	JMP s66	Si no se ha alcanzado el máximo de procesos añadir otro
	e18			

Figura 3.12: SOMC:Preámbulo

## Inicio de operaciones para la maquina

El sistema operativo mínimo estará en la memoria secundaria y para poder proceder con su carga automática y que para el usuario sea transparente se requiere de un sistema de arranque o sistema de *booteo*, para esto las instrucciones que el *starter* mande a la cola y se empiecen a ejecutar para cargar el sistema en la memoria principal deben estar en forma de tarjeta, es decir una instrucción o dato seguida de otra en forma de lista, deben empezar siempre por las siguientes:

0002

0008

con estas instrucciones de inicio basta para que las siguientes sólo tengan que ser pares de instrucciones dónde la primera indica el destino de la segunda para que se pueda cargar por si misma el resto de la tarjeta, abajo vemos cuales serían las siguientes instrucciones a ejecutar:

0003

0000

0004

0000

Son pares de instrucciones, el primer par es para cargar la bandera que indica si es un proceso del usuario o uno del SOM el que está en ejecución, como notamos la primer instrucción del par indica que la segunda sea cargada en la dirección #003 y que el valor cargado ahí es un 0000, puesto ahí por defecto para indicar que los recursos son del SOM. El siguiente par lo que nos indicará es el identificador o ID del proceso que se está ejecutando en ese mismo momento, y no se escogido el 0 sólo por ser un número por defecto, es porque justamente se está ejecutando el **proceso 0**, puesto que para el SOM este sistema de *booteo*, una vez que ha arrancado la maquina, al ejecutar las primeras dos instrucciones se transforma en un proceso que es parte del sistema operativo y que será gestionado como tal, pero con algunas características especiales que son causantes a su vez que se tenga un color especial en el diagrama y en el código cuando el SOM tiene que hacer operaciones referentes al proceso 0.

Lo siguiente que vendrá en la tarjeta es todo el contenido de cada segmento del sistema operativo, la tarjeta usada para cargar el sistema se encuentra completa en el anexo 2 para su consulta, pero básicamente son pares de instrucciones que para colocar cada segmento del sistema en su lugar.

## Encendiendo la maquina

En la figura 3.13 podemos observar como será la maquina virtual para este modelo y nos ayudará como guía para comprender muchos de los aspectos del modelo. Podemos ver que ahora en la parte inferior derecha ya no hay un espacio vacío como en la primer maquina, sino que se encuentra el contenido de la memoria secundaria(*Secondary Memory*), que tiene una dirección de memoria(en la secundaria) y el contenido, si oprimimos *Start* el contenido de la memoria secundaria se mueve a la cola de ejecución(*Queue*), pero no desaparece de la memoria secundaria como podemos ver en la figura 3.14, sino que al ser está una memoria no volátil o permanente se queda ahí estática, no cuenta como memoria ROM por que es posible reescribirla, pero de forma externa, lo podemos pensar como que es una tarjeta que se puede cargar, una cinta que se le puede colocar a la maquina para que está a través del *Starter* la coloque en la cola. Como se notará en las imágenes otra parte que cambio fue la parte del estatus de la maquina, que ahora tiene un campo llamado *Starter* que en la

maquina apagada tiene un valor de “Waiting” y en la encendida de “Booted”, esto es por que al encender el starter en automático ha colocado la tarjeta de la memoria secundaria en la cola, solo es cuestión de dejar seguir la ejecución para que la carga se complete.

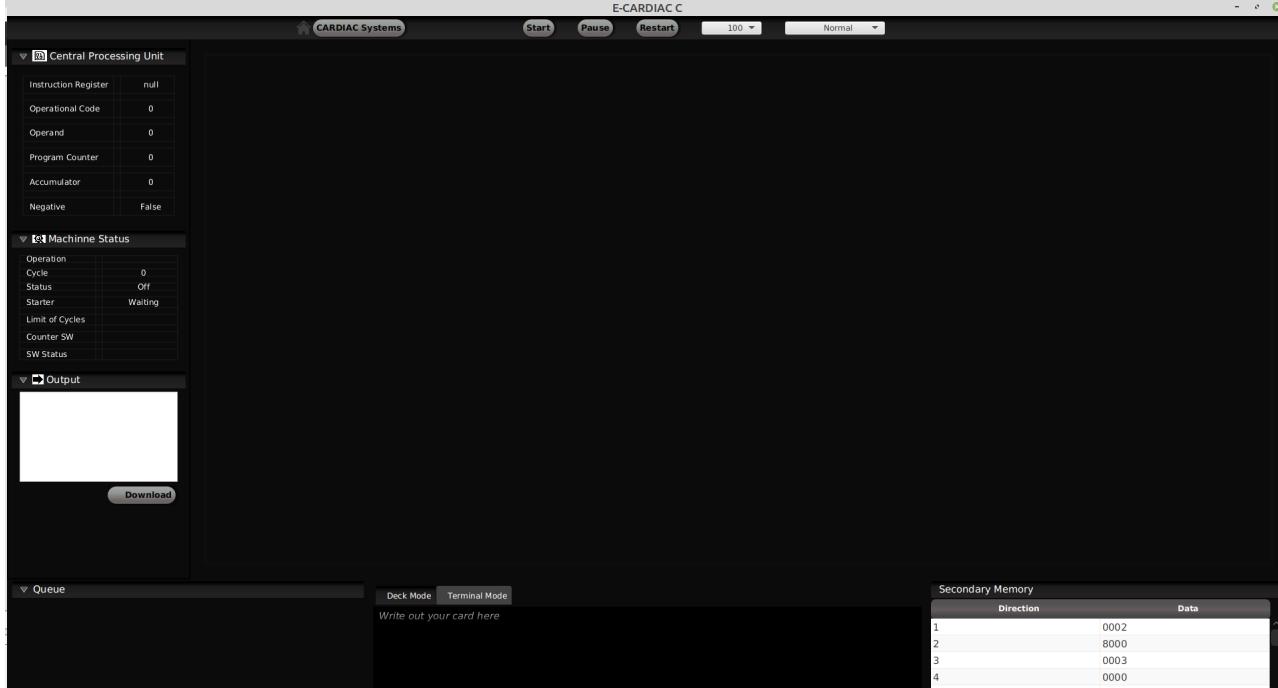


Figura 3.13: E-CARDIAC C Apagada

A parte del *Starter* tenemos otros tres campos que nos ayudaran a ver quien tiene el control de los recursos y por cuantos ciclos, el campo *Limit of Cycles* contiene el valor de cuantos ciclos máximos tendrá un proceso del usuario antes que obligadamente tenga que ceder los recurso de nuevo al SOM, la implementación que se muestra tiene un máximo de 30 ciclos para cada proceso del usuario. El *Counter SW* tiene un contador de ciclos pero a diferencia de *Cycle* esté se reinicia cada que el control de los recursos cambia de propietario, si está ejecutándose un proceso del usuario cuando esté contador alcance el mismo número que tiene *Limit of Cycles* el *Switcher* saltará de inmediato al preámbulo del SOM para que esté tome control de los recursos, si es al revés el contador sólo nos servirá de indicador de cuantos ciclos lleva el SOM desde que le cedieron los recursos, pero como la bandera se encuentra en 0 si el control lo tiene el SOM el switcher no podrá hacer ningún salto. Para saber quien tiene el control también podemos ver el último campo, el *SW Status* que nos indicará quien tiene el control de los recursos, en las imágenes lo tiene el SOM y por eso

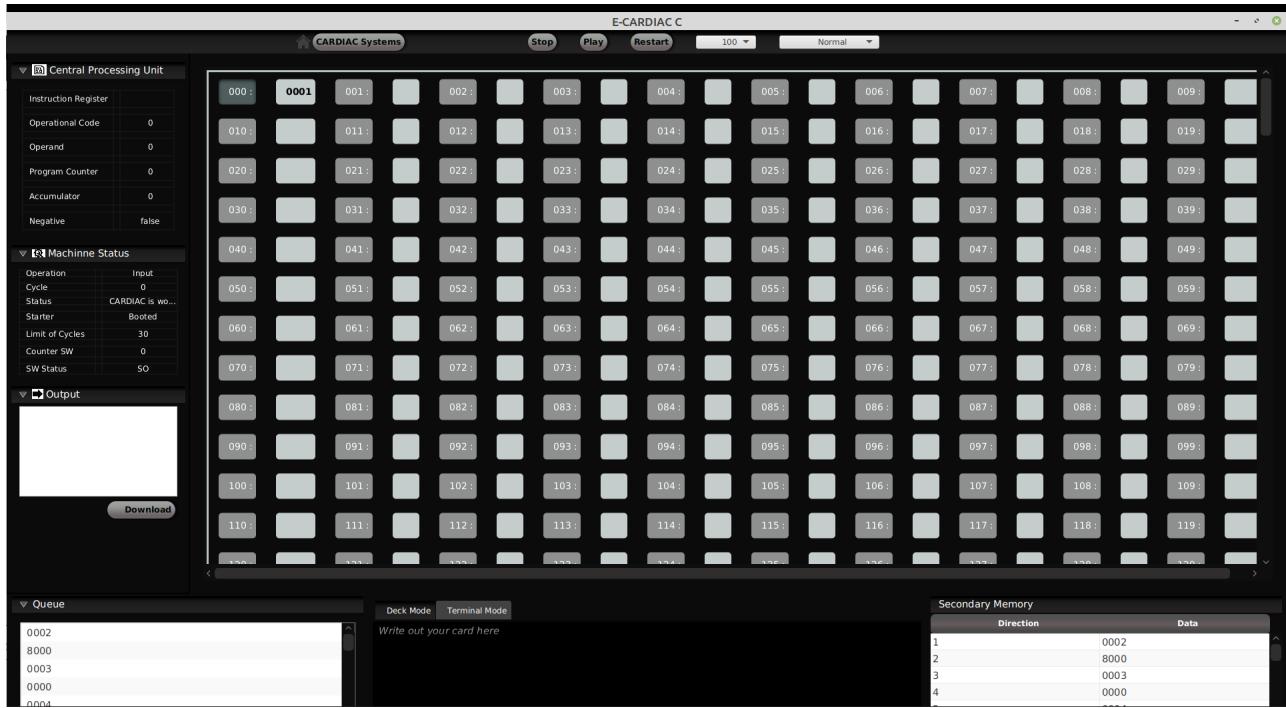


Figura 3.14: E-CARDIAC C durante el arranque

tiene un valor de  $SO$  si fuera un proceso del usuario tendría *User*. Es importante mencionar que las primeras instrucciones del preámbulo son consideradas aún como parte del proceso del usuario para el estatus por que en esas primeras instrucciones se cambia la bandera de valor, aunque para el contador del *switcher* esto ya se contempla como parte del proceso del SOM.

Lo que sucede una vez que el sistema ha sido cargado en la memoria lo podemos ver en la figura 3.15, dónde podemos ver que el puntero está en la dirección #000, es decir está esperando un dato para cargarlo en la dirección #001, que ya tiene un dato que fue cargado en algún momento durante la carga del SOM en memoria. También podemos ver que las banderas están cargadas correctamente y algo que nos puede llamar la atención es que el *SW Status* marca que el control de los recursos lo tiene el sistema operativo, y esto no es un error, por que parte de los permisos especiales que tiene el proceso 0 es que no tiene un límite para ceder los recursos, es una extensión del mismo sistema operativo, por que es el proceso que va a permitir al usuario cargar programas para que sean ejecutados, y no sería nada óptimo que cada N ciclos tuvieras que detener la carga de programa para que otro se ejecute, la carga de programas por el usuario tiene el privilegio más alto entre los procesos,

y sólo el usuario cuando haya cargado los procesos que quiera cederá el control al SOM para que administre los recursos y empiece la ejecución de los procesos previamente cargados.

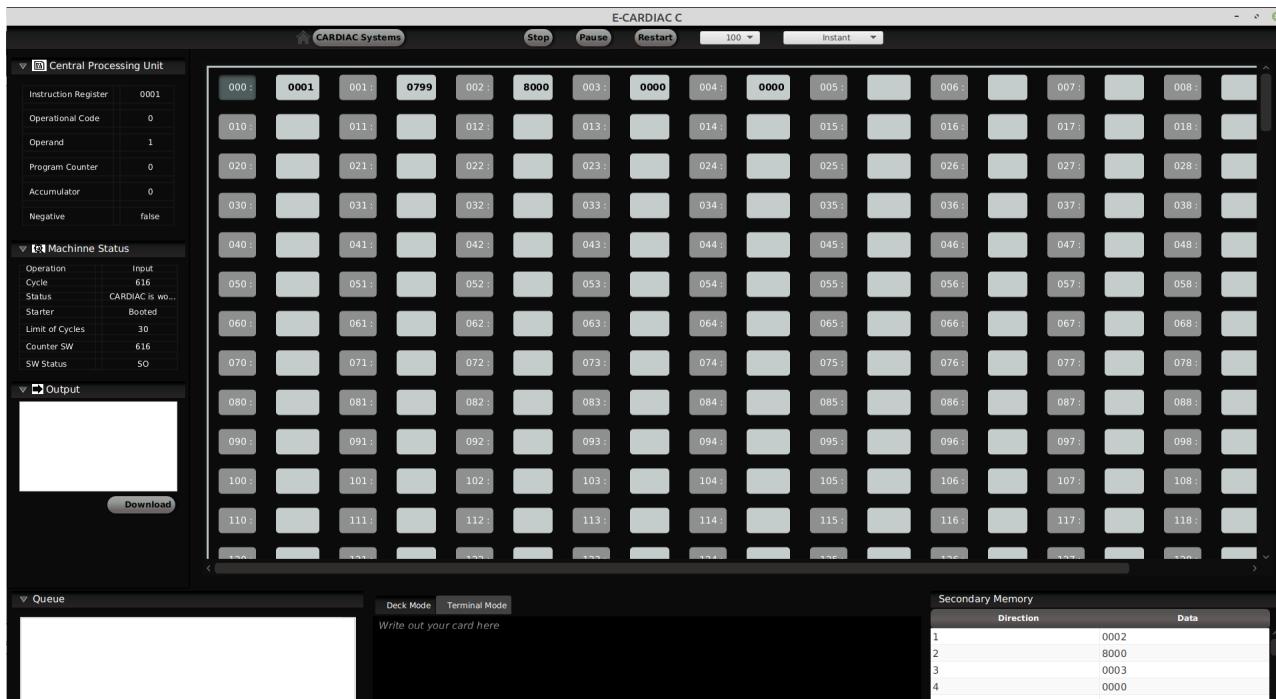


Figura 3.15: E-CARDIAC C Sistema operativo mínimo cargado

### ¿Como toma control el sistema operativo mínimo?

Como pudimos ver en las imágenes anteriores, a pesar de que para el *switcher* el control de los recursos este actualmente del lado del SOM, la realidad es que no controla los recursos y más bien es el usuario, aunque no uno de los procesos del usuario, quien controla los recursos. En la figura 3.11 que nos presenta un diagrama de flujo muy general podemos alcanzara ver tres recuadros rosas que tienen una forma más inclinada que la de un rectángulo normal, esto es por que serían entradas de datos por parte del usuario, y que en el diagrama use para ejemplificar las tres formas en que el SOM toma de nuevo el control de los recursos; la primera es cuando el va a añadir un nuevo proceso, la segunda cuando el contador de programa por medio del *switcher* salta en automático, y la tercera si se da la indicación de borrado de un proceso, es decir si se lee la instrucción #9000.

En la figura 3.16 apreciamos estás tres formas en que el sistema entra en acción como tal, y podemos notar que todas van a instrucciones de color café siempre, esto es por que

siempre antes de llegar al núcleo del SOMC se debe pasar por el preámbulo que se encarga de actividades previas a las ya mencionadas para que éstas puedan llevarse a cabo con normalidad y seguridad.

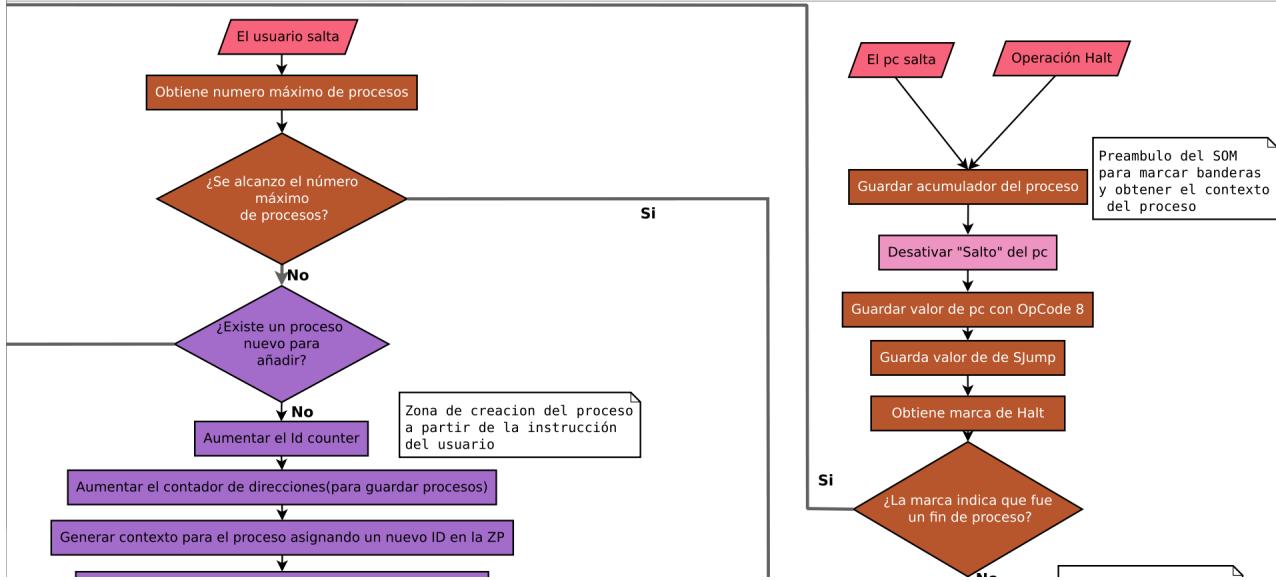


Figura 3.16: Formas de entrar al sistema operativo mínimo C

## Añadir un proceso

Veamos en orden como el SOM interactuara con los programas que añadimos y queramos ejecutar en la maquina virtual, para ello partiremos de un ejemplo práctico, lo primero que haremos es añadir un par de procesos, después empezaremos la ejecución, estos tendrán que actualizar su contexto en la zona de procesos, y finalmente veremos el borrado de cada uno.

EL programa que añadiremos es para imprimir números del 1 al 10 al que llamaremos “pintador”, pero lo haremos en dos versiones, uno que inicia después de la dirección #100 y la otro que inicia después de la dirección #300. Veamos las tarjetas del que usará direcciones después de la #100, en la tabla 3.2 observamos tres columnas en las que vemos un fragmento de la tarjeta que el usuario colocará en la maquina virtual para añadir este programa a la zona de procesos y que después lo pueda ejecutar, vemos que la primer instrucción de la tarjeta que el usuario está ingresando es para indicar que se cargue en la dirección #110 la primer instrucción de “pintador”, y seguimos el mismo concepto de pares de instrucciones en estas tarjetas, pues siempre va primero la instrucción que indica dónde se cargara la segunda,

podemos ver así el segundo par, el antepenúltimo y el penúltimo. Pero si notamos al final hay una sola instrucción que se sale de está regla, esa instrucción es para saltar directamente a la dirección dónde el sistema operativo empezará a añadir el programa a la zona de procesos, en ese momento el usuario está cediendo por completo el control al SOM para que el programa que ya cargo en memoria sea añadido a la zona de procesos y se convierta en uno de los procesos que ejecutara el sistema.

Para ver la tarjeta completa podemos observar la tabla 3.3 dónde está la tarjeta completa, se lee de arriba hacia abajo y de izquierda a derecha, puesto que ponerla en modo vertical sería dejar mucho espacio en blanco. En esta podemos observar que todas las instrucciones que están en la mitad, las que fueron omitidas en la tabla 3.2 son pares de instrucciones para ir cargando el programa en la memoria principal.

Código maquina	Ensamblador	Comentarios
0110	LDA 110	Cargará la primer instrucción en 110
1000	LDA 000	Primera instrucción del programa
0111	LDA 111	
6105	STO 105	Segunda instrucción del programa
...	...	...
0122	LDA 122	Última dirección del programa
9000	HLT 000	Dato
0104	LDA 104	Asignar constante n
0009	LDA 009	Dato
0800	LDA 800	Cargar en 800 la dirección de inicio del nuevo proceso
8110	JMP 110	Dato
8985	JMP 985	Saltar al segmento de añadir proceso del SO

Tabla 3.2: Fragmento de código, con explicación, para imprimir números del 1 al 10

Veamos ahora lo que el sistema tiene que hacer, y lo primero es saber a que parte del sistema se salta con esa instrucción del final, el salto es al preámbulo del sistema como ya podíamos suponer por lo revisado y más precisamente a la dirección que tiene como variable **e14**, es así que podemos decir que en nuestra implementación el valor de **e14** es #985, y si vemos en la imagen 3.12 en la parte de abajo dónde empieza **e14** desde el nombre clave que se asocia está y las siguientes direcciones hace referencia a añadir un nuevo proceso, y en esta parte empiezan ya las interacciones con ciertas variables predefinidas que se encuentran

Pintor		
0110	0116	0122
1000	2000	9000
0111	0117	0104
6105	6105	0009
0112	0118	0800
1104	1104	8110
0113	0119	8985
3122	7000	
0114	0120	
5105	6104	
0115	0121	
1105	8112	

Tabla 3.3: Tarjeta para cargar proceso para imprimir números del 1 al 10

en la zona de variables.

Podemos ver en la imagen 3.17 que tenemos 19 variables o constantes que el sistema estará usando a lo largo de su ejecución, particularmente en este momento nos interesan la *c4* y la *c16*, la primera es un contador de los procesos del usuario, que por defecto tiene 0 y en esté caso debe tener 0; la otra contiene el número máximo de procesos que se pueden cargar en el sistema operativo, el número máximo para la implementación es 5 pero para un funcionamiento más optimo se coloca el número máximo menos uno. Volviendo al preámbulo entenderemos por que se hace ese ajuste, lo primero que hace es tomar el número máximo de procesos permitidos(menos uno), a ese número le resta la cantidad de procesos que el usuario ha agregado y toma una decisión, si el resultado es menor que cero regresa al proceso 0, en caso contrario saltara directamente a añadir un proceso nuevo. En el caso de que el usuario tenga 4 procesos agregados y se vaya a añadir otro(el quinto), *c4* tendrá un valor de 4 y *c16* de 4 también, por lo que el resultado será 0, por lo que permitirá añadir el quinto proceso, si en cambio el usuario ya tuviera 5 procesos el resultado sería negativo y se regresaría al proceso 0.

Ahora sabemos que desde el preámbulo va a saltar al segmento del núcleo del SOM para añadir un nuevo proceso, puesto que hay 0 procesos del usuario y que como recordaremos la letra “s” en las variables que uso para representar direcciones del sistema operativo indica que es una dirección del núcleo, más precisamente la dirección *s66*. Pero veamos primero el

Variables del sistema				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
	c0			Espacio para el SOM
<i>8-pc</i>	c1			Espacio para el SOM/Preámbulo, recibe un valor en cada iteración
<i>acc</i>	c2			Espacio para el SOM/Preámbulo, recibe un valor en cada iteración
<i>Inicial</i>	c3	p5		Es la dir de inicio de la zona de procesos(usuario)
<i>Id counter</i>	c4	0		El contador de procesos de usuario
<i>Dir counter</i>	c5	p0		El contador de direcciones
<i>Dir organizer</i>	c6	p0		Contiene la dirección del proceso que se ejecutará
<i>Id organizer</i>	c7	0		Contiene el id del proceso que se ejecutará
	c8	1000		Valor para convertir op-code en LOAD
	c9	6000		Valor para convertir op-code en STORE
	c10	-1		Valor de uso recurrente
	c11	5		Valor de uso recurrente, para el salto de los procesos
	c12	2		Valor de uso recurrente
	c13	8000		Valor para convertir op-code en JUMP
<i>Saver Jump</i>	c14			Guardará el valor de 999 por parte de la arquitectura
<i>Serial Id Proces</i>	c15	0000		Contendrá un valor serial para los id de los procesos
<i>Máximo numero</i>	c16	0004		Máximo numero de procesos disponibles (Menos 1 para la resta)
	c17	998		Espacio para el SOM/Área de borrado
	c18	0004		Espacio para el SOM/Área de borrado

Figura 3.17: SOMC : Variables del sistema

diagrama con un acercamiento en la parte de añadir un nuevo proceso (figura 3.18), después de pasar por la parte café del preámbulo con un resultado de “No”, es decir no se ha alcanzado el máximo número de procesos llega a otro condicional que pregunta si en realidad existe un proceso nuevo a añadir. Para este punto es que se utiliza ese par de instrucciones último antes de saltar del proceso 0, en la tarjeta para cargar el programa “pintor” esté par contiene la instrucción *0800* que significa cargar en la primer posición del núcleo del sistema, que es la única dirección en la que el usuario puede directamente escribir sin restricciones, y la instrucción que se carga ahí es *8110* que contiene un código de operación para saltar y la dirección de inicio del proceso que queremos cargar. En la imagen 3.19 podemos ver que *s0*, que en nuestra implementación sería *800*, tiene un valor por defecto de *-0001*, puesto que con este valor se indica que no hay un proceso a cargar y cada que se carga un nuevo proceso el contenido de esta dirección regresa a su valor por defecto.

En caso de que el valor fuese negativo el flujo sería más corto, pues irá a verificar que el *Id organizer* u organizador de identificadores sea igual a cero, lo que indica que se encuentra en el proceso 0, si la orden de añadir un nuevo proceso fue lanzada desde el proceso 0 y nunca se cambio el valor de *s0* entonces regresará al proceso 0 con el recorrido más corto. Pero el último recuadro morado continua su flujo también en esta condicional, así en ambos resultados posibles del condicional “¿ Hay un nuevo proceso a añadir?” se termina por revisar

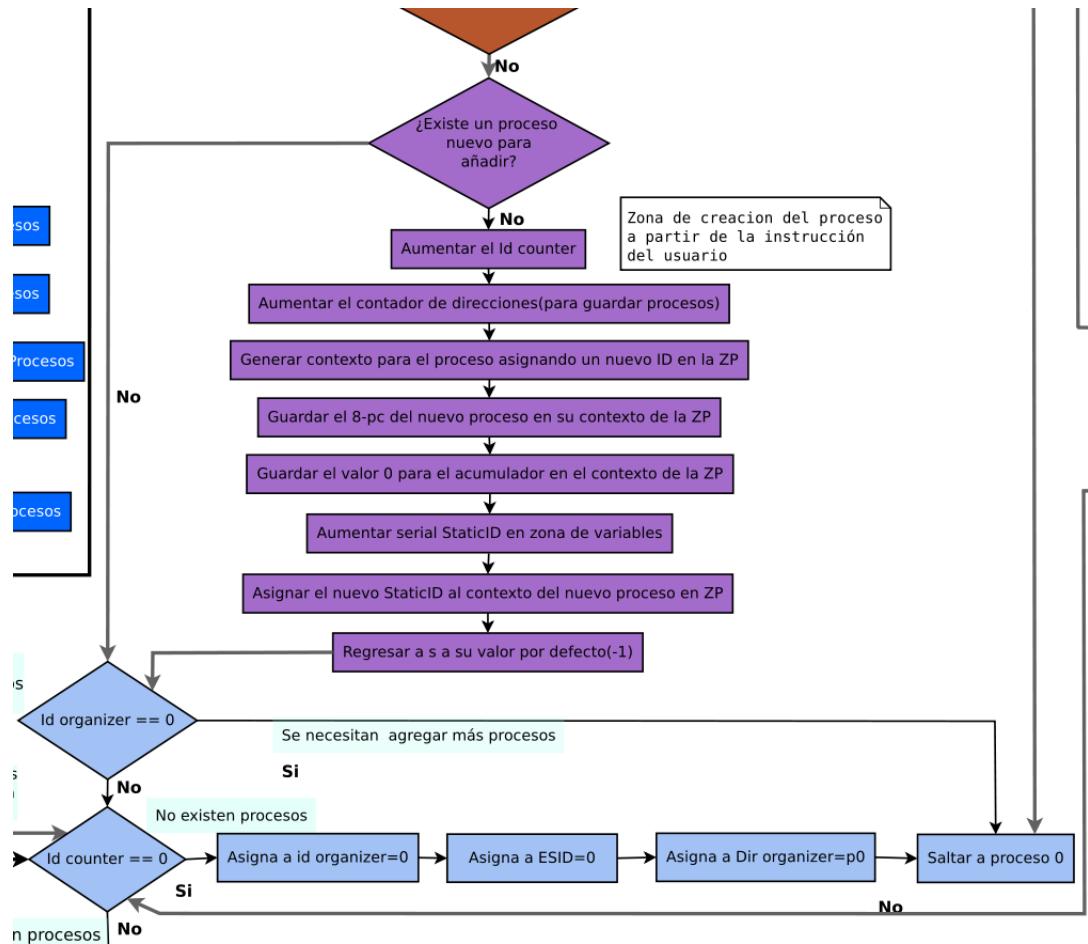


Figura 3.18: Diagrama de segmento para añadir un proceso

si la orden fue lanzada desde el proceso 0, el resultado más sencillo es que si hayan lanzadas desde ese proceso, pero en el caso de que no sea así la entrada a esa sección azul es muy relevante puesto que si vemos el diagrama completo en esta sección confluyen los tres caminos posibles en los que el SOM vuelve a administrar los recursos. Y en el caso de que no se haya lanzado la orden desde el proceso 0 la ejecución se enlazará con el otro segmento del núcleo del sistema que es el lanzamiento de procesos, segmento que veremos más adelante y que se encarga de “elegir” que proceso será el siguiente en ejecutarse.

Regresando a la parte morada del diagrama, después de verificar que si hay un nuevo proceso a añadir, y apoyándonos también en las imágenes 3.20 y 3.21 es actualizar valores tanto en la zona de variables como en la zona de procesos. En cuanto a las variables del sistema vemos que actualiza los valores de  $c_4$ , el contador de procesos que ya vimos y que aumenta en uno naturalmente; y por otra parte  $c_5$ , el “contador de direcciones”, esté contador

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
Nuevo	s0	-1		-1 Es su valor por defecto, el usuario siempre agregará una 8-dir
Actual	s1	6000		Contiene la 6-dir del id del proceso actual, controlado por el lanzador, 6(px)

Figura 3.19: SOMC : Contenidos generales del núcleo

lo que tiene como valor por defecto es  $p0$ , como recordaremos toda variable que inicie con  $p$  hace referencia a la zona de procesos, precisamente  $p0$  es la dirección de inicio de la zona de procesos y contiene el identificador del proceso 0.

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
	s66	1(s)	LDA s	Se carga en el acumulador la 8-dir de inicio del proceso
	s67	3(s95)	BLZ s94	si acc<0 no hay nuevo proceso
Aumenta el Id counter M[c4]++	s68	1(c4)	LDA c4	Se obtiene el valor del Id counter
	s69	2000	ADD 000	
	s70	6(c4)	STO c4	
Aumenta el Dir counter M[c5]=+3	s71	1(c5)	LDA c5	Se obtiene el valor del Dir counter
	s72	2(c11)	ADD c11	Se le suma 5 para tener la siguiente dirección
	s73	6(c5)	STO c5	
Previa de ID	s74	2(c9)	ADD c9	Convierte la nueva dirección en dir counter en 6-dir
	s75	6(s87)	STO s86	En s70 se guarda 6(px)
Previa de pc	s76	2000	ADD 000	Se obtiene la 6-dir del gpc del nuevo proceso
	s77	6(s89)	STO s78	En s72 se guarda 6(px)+1, la 6-dir del gpc del nuevo proceso
Previa de acc	s78	2000	ADD 000	Se obtiene la 6-dir del gacc del nuevo proceso
	s79	6(s92)	STO s81	En s75 se guarda 6(px)+2

Figura 3.20: SOMC : Añadir un proceso, parte 1

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
	s80	2(c12)	ADD C12	Añade 2 para obtener 6(px)+4(Static ID)
Guardar Nuevo Static ID para el proceso	s81	6(s85)	STO s85	Guardar en s85
	s82	1(c15)	LDA c15	Obtener Serial de Static ID
	s83	2000	ADD 000	Añadir una unidad
	s84	6(c15)	STO C15	Guardar en c15 el folio actualizado
	s85	6(px)+4	STO (px)+4	Guarda el nuevo serial Static ID para el proceso
Id del nuevo proceso	s86	1(c4)	LDA c4	Se obtiene el Id para el nuevo proceso
	s87	6(px)	STO px	Se guarda el Id en la zona de proceso
pc nuevo	s88	1(s)	LDA s	Se obtiene el 8-pc del proceso
	s89	6(px)+1	STO px+1	Se guarda el 8-pc en el gpc del proceso
acc del nuevo proceso	s90	1000	LDA 000	
	s91	7000	SUB 000	Se obtiene el 0
	s92	6(px)+2	STO px+2	El gacc del proceso se inicializa en 0
Poner valor default en s	s93	1(c10)	LDA c10	
	s94	6(s)	STO s	En s se coloca el valor -1

Figura 3.21: SOMC : Añadir un proceso, parte 2

En la figura 3.22 podemos ver como está compuesta la zona de procesos, llamaremos “contexto del procesos” a todas las variables asociadas al proceso y que se almacenaran en la zona de procesos bajo un mismo identificador. Para nuestros contextos requeriremos de cinco variables, si vemos  $p5$  y  $p10$  tienen valores de -0001 y esto es por que son los inicios

de cada contexto, cada contexto inicia cada 5 direcciones y un contenido negativo en su primera dirección que es el identificador significa que no hay proceso en ese contexto. Es por ello que al contador de direcciones se le añade un 5, puesto que si está en  $p0$  con ese 5 puede cambiar al contexto de  $p5$ , para nuestro caso es justo lo que hace, cambia a  $p5$  el contador de direcciones por que ahora el último proceso es el que está asociado a ese contexto. El resto de variables que consideramos para el contexto de cada proceso son su último contador de programa guardado en  $gpc$ , su último acumulador guardado en  $gacc$ , el último valor que tuvo la última dirección de la maquina (#999 en la implementación actual) dado que este valor cambia de acuerdo a las instrucciones de cada proceso y es vital guardarlo en el contexto de cada proceso para que no haya fallas lógicas por que el proceso A dejo un  $819$ , luego el proceso B dejo un  $514$  y cuando vuelva el control al proceso intente usar el valor que hay en #999 para ir a alguna parte de su proceso y termine en el proceso B, esté valor es guardado en  $gjump$ . Por último para identificar a los procesos además del id principal que tienen y con el que inicia el contexto, el cuál tiene por nombre completo *id counter*, pues va en orden ascendente y sirve también para contar cuantos procesos existen, tenemos el identificador estático guardado en *Static Id*, más adelante en la sección de borrado veremos la importancia de esté identificador estático, pero por ahora concentrémonos en que ya sabemos cuales son las variables de contexto para los procesos y por ende las que tenemos que llenar al agregar uno nuevo.

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
Nuevo	s0	-1		-1 Es su valor por defecto, el usuario siempre agregará una 8-dir
Actual	s1	6000		Contiene la 6-dir del id del proceso actual, controlado por el lanzador, 6(px)

Figura 3.22: SOMC : Nucleo del sistema operativo

Regresando a las figuras 3.18, 3.20, y 3.21 podemos tener más claro que es lo que sucede, primero afecta a las variables del sistema para que estén actualizadas la cantidad de procesos agregados y cuál es el último contexto de la zona de procesos activo, es decir con un *Id counter* no negativo. Ahora desde la dirección  $s74$  hasta la  $s92$  lo que hace es llenar el contexto del proceso, ya se definió que  $p5$  es la última dirección de inicio de un contexto de proceso, por lo que lo primero es asignar el número 1 al contenido de  $p5$ , pues el *id coutner* será un contador de identificadores literalmente que tendrá valores ascendentes, si agregará otro

proceso el contenido de  $p10$  sería un 2, y cuando borre ese proceso su contenido regresará a ser negativo, hasta que agregue otro proceso y su valor vuelva a ser dos. Esta estabilidad en el identificador principal nos permite que con solo ese valor conozcamos la cantidad de procesos que hay y saber el lugar que ocupa cada proceso en la lista de procesos que se van a ejecutar.

Para hacer la carga de estos tres valores del contexto del proceso en su correspondiente lugar hacemos que sea el mismo sistema operativo quien coloque las instrucciones para hacer la carga, las instrucciones resaltadas en rojo en las imágenes del código para añadir un nuevo proceso están así por que fue el mismo sistema quien las colocó. Veamos el caso del contador de identificadores, en la dirección  $\#s73$  el valor del acumulador es  $p5$ , es decir la dirección de inicio del contexto del nuevo proceso, como es una dirección sin más tiene un código de operación de 0, por lo que si en  $\#s74$  le sumamos el valor que contiene  $\#c9$ , que es un  $6000$  lo que haremos será cambiar el código de operación que acompaña a la dirección  $p5$  de un 0 a un 6, dando por resultado un  $6(p5)$ , resultado que con la instrucción de  $\#s75$  guardamos en la dirección  $\#s87$ , que como vemos tiene un contenido con un código de operación que indica la operación *Store* acompañado de un  $px$ , que en nuestro caso sería el  $p5$ . Y como en  $\#s86$  se cargo en el acumulador el valor de  $\#c4$  que tiene el número de procesos que se han agregado, cuando llega a  $\#s87$  y la instrucción indica que el valor del acumulador se guarde en  $p5$  se guardará el número 1, pues es el número de procesos que se han agregado hasta ese momento.

En las imágenes para simplificar el código un  $px$  siempre representará la dirección del *id counter del proceso*, un  $px+1$  el del *gpc* del proceso, y así sucesivamente dado que como se puede ver en el código no van en orden la carga de los valores y eso puede llegar a ser confuso, con esta regla podemos notar más fácilmente que en  $\#s92$  se está cargando el contenido que deberá tener el *gacc* del contexto del proceso.

Después se requiere colocar en  $p6$ , que representa a la variable *gpc*(el contador de programa guardado), la dirección de inicio del proceso antecedida por un código de operación 8 para que cuando se quiera lanzar el proceso sea sencillo. En *gacc* que se encuentra en  $p7$  el valor seria el valor que tomará el acumulador cuando el proceso arranque, ese valor por defecto es 0. El proceso se salta al *gjump* por que todas las direcciones que contendrán la

variable de contexto del proceso que hace referencia al valor de `#999` en su última instrucción ejecutada tendrán como valor por defecto el `8000`, pues se estaría reiniciando en cada inicio de un nuevo proceso. Lo que sigue en el flujo entonces es modificar la *static id* o identificador estático, que se debe modificar tanto en la zona de variables del sistema como en el contexto del proceso que estamos agregando, y es que mientras el *id counter* no guarda ninguna relación con el proceso una vez que esté termina, pues se reutiliza para otros una y otra vez, el identificador estático depende de un valor en la zona de variables del sistema que es un serial, es decir empieza en `0` y se va aumentando a medida que agregamos procesos, así el proceso que tenga asignado el número `1` será el único que tenga esa asignación mientras la maquina este encendida. Para lograr esto lo primero que se hace es aumentar el valor del contenido de la `c15` en una unidad y después ese valor guardarlo en `#p9`, que es la dirección dónde seguirá el *static id* del proceso nuevo que se está añadiendo.

Para finalizar reinicia el valor de `#s0` a un negativo indicando que ya no hay un nuevo proceso a añadir y sigue su flujo al segmento especial para validar si el proceso fue lanzado desde el proceso `0` y en su caso regresar al proceso `0` o si bien hay que tomar el camino largo. Como notarán en ningún momento se hace cambio de bandera aquí, y es por que desde que el usuario está agregando el programa hasta que el SOM lo añade a la zona de procesos para el *switcher* nunca cambia el dueño de los recursos, es decir, sigue siendo el SOM.

### Practica añadiendo un proceso nuevo

Veamos en la práctica utilizando la máquina virtual como es añadir un proceso nuevo a la máquina, recordando, estamos en la situación de la máquina iniciada y con el sistema operativo cargando y en la dirección `#000` esperando a que un nuevo proceso sea cargado. Podemos notar un par de cosas que diferencian al inicio en la figura 3.15 donde en las últimas dos casillas que son listas desplegables(en la parte superior) tenemos un `100` que hace referencia al número de celdas e *Instant* que hace referencia a la velocidad en que cada ciclo es completado, como pudimos notar la máquina no tiene `100` celdas, y es que cuando se inicia la máquina virtual hace una validación de si las celdas que indica la celda lista desplegable son suficientes para el tipo de procesos que llevará a cabo, **E-CARDIAC C** sólo puede funcionar con mil celdas o más, por eso si la lista desplegable tiene un valor

menor será sustituido en automático por el mínimo permitido. En la figura 3.23 he cargado de nuevo la maquina y ahora si tiene el valor correcto de 1000 celdas en la lista desplegable, además en la imagen 3.24 podemos ver que podemos cambiar el que tarda en terminar un ciclo a mitad de la ejecución pues ahí se ve que tiene el valor de *Normal*, una velocidad en la que podemos apreciar los cambios y como va ejecutando cada instrucción.

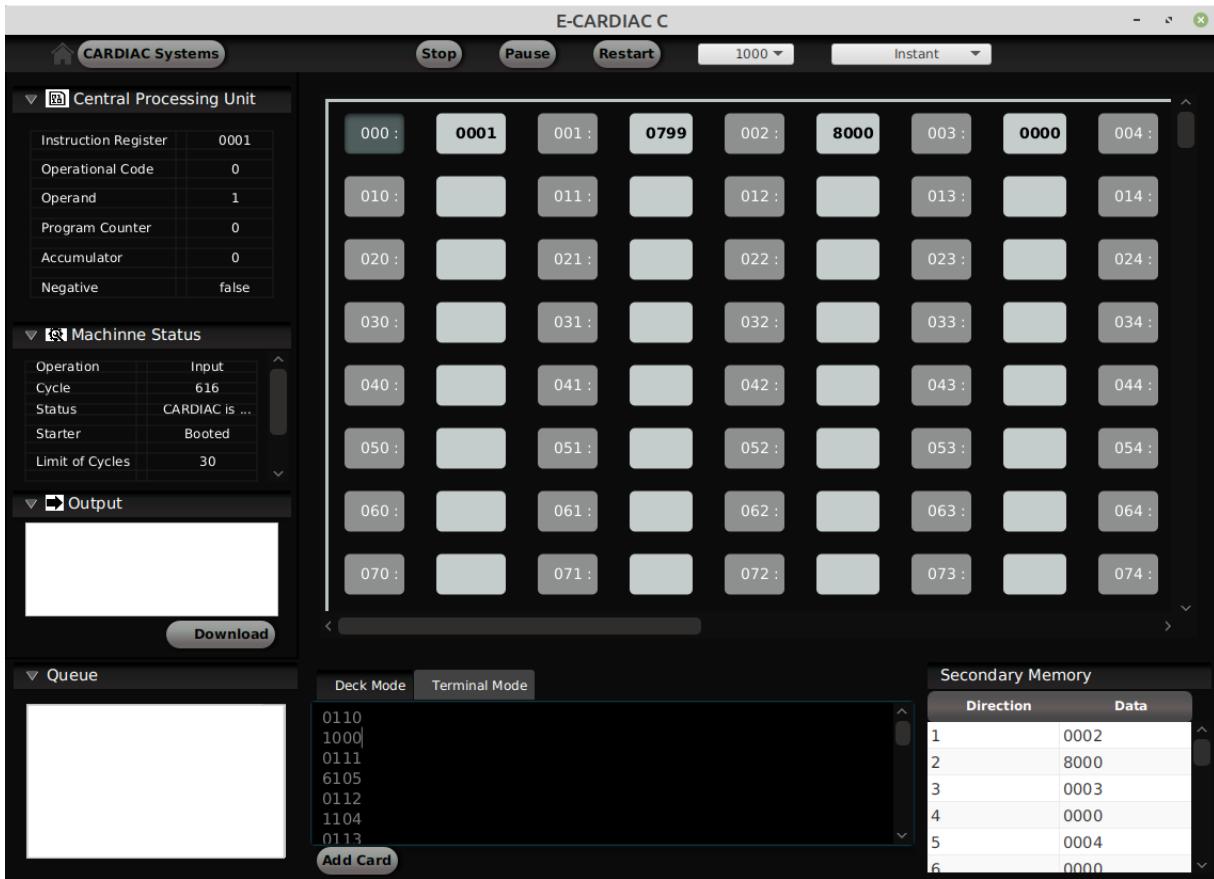


Figura 3.23: E-CARDIAC C Programa en deck

Ahora en la imagen 3.25 la instrucción que va a ejecutar es para saltar al preámbulo y añadir un nuevo proceso a la lista, y en la figura 3.26 vemos al programa como tal cargado en memoria, cuando el sistema termina de agregar los datos del programa a la zona de procesos esté se convierte en si en un proceso, ya no son solo instrucciones de código, el contexto del proceso ha quedado definido y con ello todas las variables que necesita mantener con los mismos valores cuando ceda y luego recupere los recursos. En la figura 3.27 vemos al contexto del proceso 0, que inicia en la dirección #769, y también el del proceso 1, que inicia en la dirección #774 con un *id counter* con su correspondiente valor de 1, un *gpc* que indica

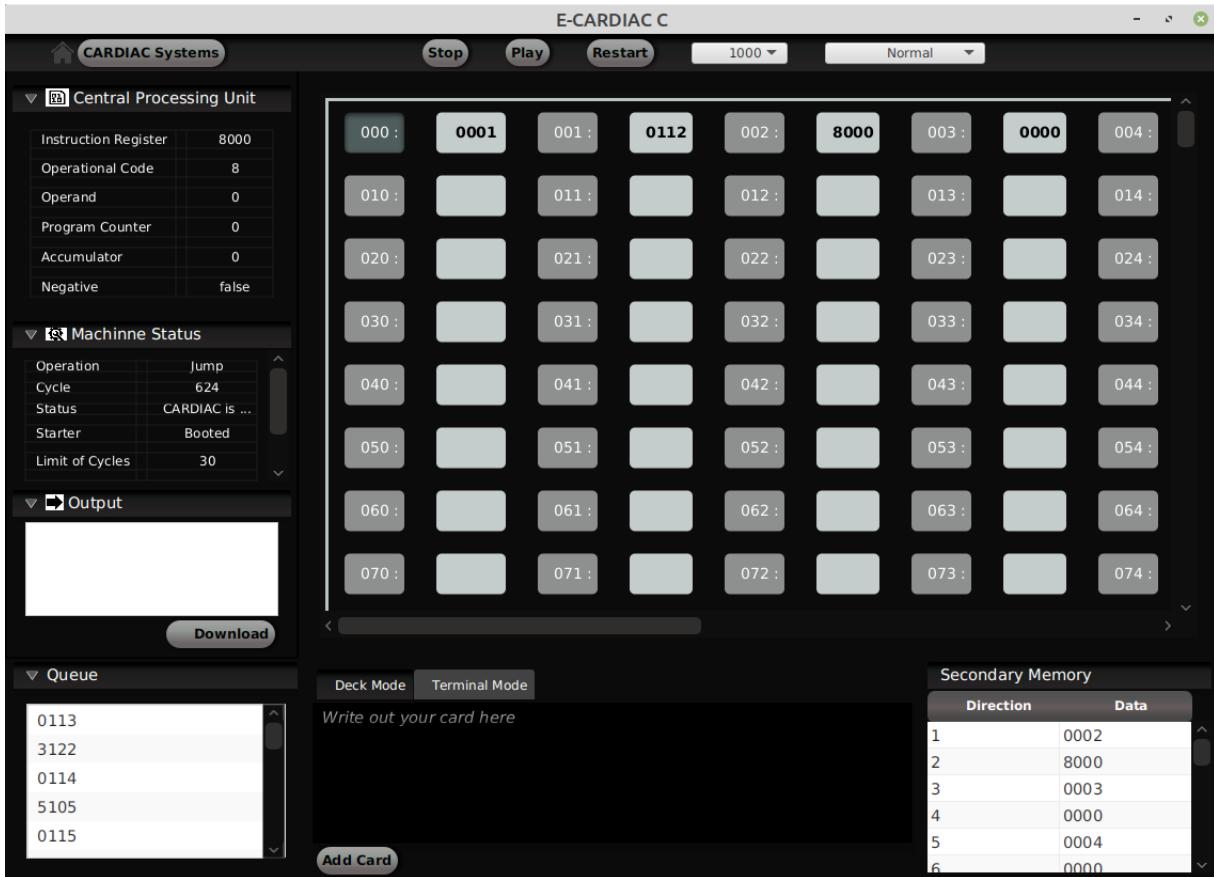


Figura 3.24: E-CARDIAC C Programa en cola

la dirección de inicio del proceso, un *gacc* que con ceros en su valor inicial, el *gjump* con el valor por defecto que tiene la maquina para la dirección #999 y con un 1 para el *static id* puesto que es el primer proceso que se está agregando. Notamos también que la dirección #779 tiene un valor de -1 puesto que no hay otro proceso más hasta este momento.

Por lo que para que sea más útil este ejemplo practico agregaremos otros dos procesos, que hacen lo mismo(son “pintores”) pero que iniciaran en #310 y #510 respectivamente, como una vez que se termina de agregar un proceso se regresa al proceso 0, podemos sin ningún problema repetir el proceso hasta alcanzar el número máximo de procesos, en la imagen 3.28 vemos que ahora #799 tiene un 2 como identificador. y #784 tiene uno con un valor de 3, los tres procesos que agregamos tienen casi los mismo valores de arranque en su contexto, salvo las direcciones de inicio y sus identificadores, tanto estáticos como contadores.

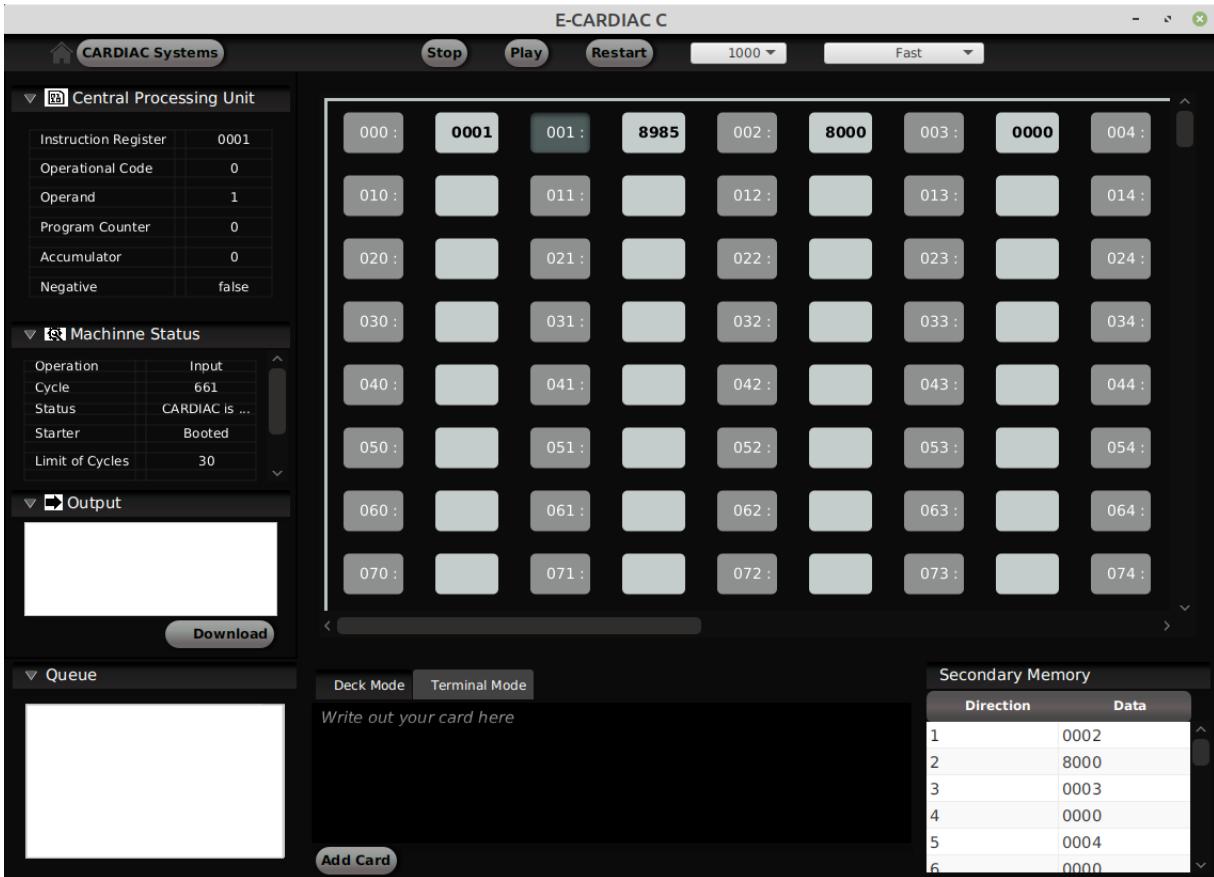


Figura 3.25: E-CARDIAC C Instrucción para añadir proceso

100:		101:		102:		103:		104:	0009	105:		106:		107:		108:		109:	
110:	1000	111:	6105	112:	1104	113:	3122	114:	5105	115:	1105	116:	2000	117:	6105	118:	1104	119:	7000
120:	6104	121:	8112	122:	9000	123:		124:		125:		126:		127:		128:		129:	
130:		131:		132:		133:		134:		135:		136:		137:		138:		139:	

Figura 3.26: E-CARDIAC C Programa 1 cargado en memoria

## Inicio de ejecución de procesos

Si nos fijamos atentamente en el diagrama de flujo principal notaremos que no hay una entrada en la que diga algo como “ejecutar proceso”, una de las entradas al SOM la vimos para añadir un proceso, otra cuando el contador de programa salta en automático, y la tercera cuando se borra un proceso, entonces ¿como empezamos la ejecución de los procesos que tenemos cargados?. La respuesta está en la última entrada mencionada, y en características especiales del proceso 0. Puesto que lo que el usuario tiene que hacer para iniciar la ejecución de los procesos que ha cargado es colocar en #001 el valor *9000*, la instrucción de detener un

760 :		761 :		762 :		763 :		764 :	
765 :		766 :		767 :		768 :		769 :	<b>0000</b>
770 :	<b>8000</b>	771 :	<b>0000</b>	772 :	<b>8000</b>	773 :	<b>0000</b>	774 :	<b>0001</b>
775 :	<b>8110</b>	776 :	<b>0000</b>	777 :	<b>8000</b>	778 :	<b>0001</b>	779 :	<b>-0001</b>

Figura 3.27: E-CARDIAC C Proceso 1 cargado

760 :		761 :		762 :		763 :		764 :	
765 :		766 :		767 :		768 :		769 :	<b>0000</b>
770 :	<b>8000</b>	771 :	<b>0000</b>	772 :	<b>8000</b>	773 :	<b>0000</b>	774 :	<b>0001</b>
775 :	<b>8110</b>	776 :	<b>0000</b>	777 :	<b>8000</b>	778 :	<b>0001</b>	779 :	<b>0002</b>
780 :	<b>8310</b>	781 :	<b>0000</b>	782 :	<b>8000</b>	783 :	<b>0002</b>	784 :	<b>0003</b>
785 :	<b>8510</b>	786 :	<b>0000</b>	787 :	<b>8000</b>	788 :	<b>0003</b>	789 :	<b>-0001</b>

Figura 3.28: E-CARDIAC C Tres procesos agregados

proceso, para detener al proceso y ¿borrarlo?. Lo que hace entonces es saltar al preámbulo, y ahora podemos ver en más detalle lo que hacen las primeras instrucciones de esté, por lo que hay que ver la figura 3.12 y la 3.29 dónde podemos ver en más detalle las instrucciones que ejecuta.

Después de un Halt el salto es directo a la dirección #e1, y es exactamente el mismo flujo para cuando ha saltado el contador de programa por medio del *switcher*, lo primero que hace es salvar el último valor del acumulador(del proceso) antes de saltar en una variable del sistema, después cambia la bandera para que ya no permita saltos, y a continuación guarda el último contador de programa del proceso del que acaba de salir en otra variable del sistema, esté valor lo consigue guardar por que el switcher en automático cuando salta coloca el último valor del *pc* que tuvo el proceso, mientras que si fue la instrucción *halt* la que salto coloca un negativo en su lugar. Continua salvando el valor del *gjump* tomando directamente el valor actual del contenido de #999 por que los saltos hacia #e1 son por fuera de las instrucciones internas de la maquina, son saltos realizados por comandos de funcionamiento de la maquina, entonces para cuando llega la ejecución a #e8 perfectamente puede tomar

el valor de #999 y guardarlo en la zona de variables del sistema. Por último vuelve a leer el valor de #e0 y si es negativo salta a la sección de borrado en caso contrario salta a la dirección de actualización, en nuestro caso #e0 será negativo por lo que saltará a la sección de borrado. En la figura 3.30 podemos ver que en la dirección #950, el equivalente a #e0, está un valor negativo, por lo que cuando llega al condicional en #961 salta la dirección que llevará a ejecutar otro salto, pero está vez para el área de borrado, si vemos el diagrama en esta parte la flecha sale hasta el extremo izquierdo dónde está el área de borrado.

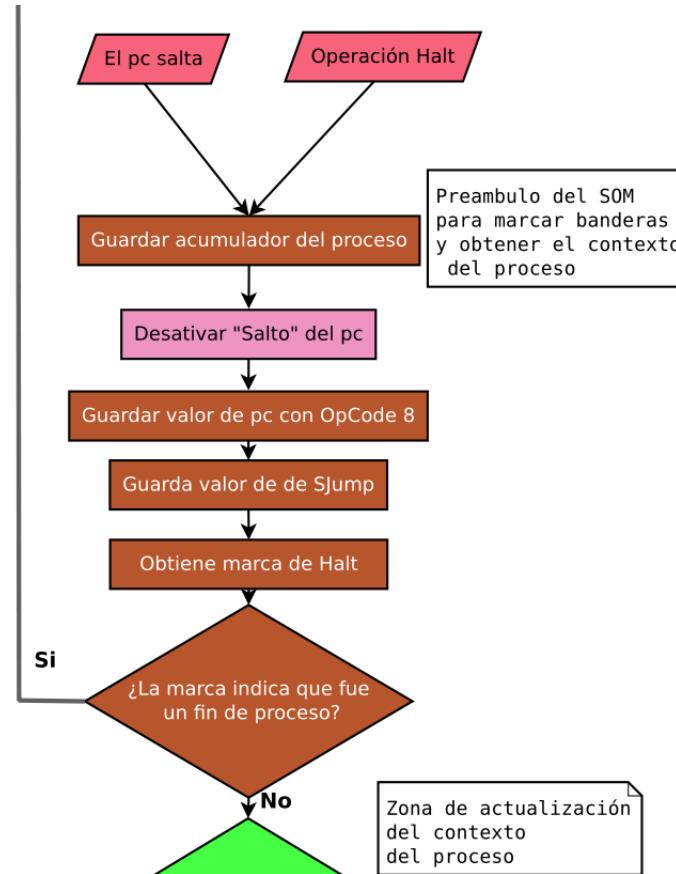


Figura 3.29: Acercamiento al preámbulo en el diagrama

950 :	-0001	951 :	6967	952 :	1000	953 :	7000	954 :	6003
955 :	1950	956 :	2978	957 :	6966	958 :	1999	959 :	6979
960 :	1950	961 :	3963	962 :	8802	963 :	8819	964 :	
965 :	0998	966 :	7999	967 :	-0001	968 :	0774	969 :	0003

Figura 3.30: E-CARDIAC Preámbulo después de una detención en P0

Y como vemos en el diagrama una vez que llega al área de borrado lo primero que hace es preguntarse si es el proceso 0 desde dónde se ejecuto está instrucción, si no lo es continua con la ejecución normal, pero en caso contrario salta a un segmento que es más bien para realizar ciertas operaciones especiales en el proceso 0, el segmento del proceso 0 dentro del núcleo del sistema. Como la ejecución pasa necesariamente por la parte que recupera el valor de `#s1(#801` en la implementación) es buen momento para entender que función tiene, si vemos la figura 3.19 podemos ver que en el comentario ya nos da una pequeña explicación, y es que en efecto el valor que guarda esa dirección siempre será la dirección del *id counter* del proceso que se está ejecutando en ese momento con un código de operación *6*, a lo que para acortarlo podemos llamarle una *6-dir*, usando como prefijo el código de operación que tiene la dirección a la que hacemos referencia. Por lo tanto podemos ver la imagen 3.31, que muestra el inicio de la zona de borrado y vemos que lo primero que hace es obtener el valor de `#s1`, lo convierte en una *1-dir* o dirección con un op-code *1* para guardarlo en una variable del sistema por que más adelante será útil en caso de que el proceso sea distinto del proceso 0, por que lo que sigue es obtener el *id counter* del proceso que se está ejecutando y verificar si es el proceso 0, en caso de que lo sea salta al segmento del proceso 0. En nuestro caso como es el proceso 0 salta a la zona del proceso 0 a bloquear el proceso, y no a borrarlo, al bloquearlo impide su ejecución infinita y deja que otros procesos se ejecuten hasta que se requiera nuevamente que esté proceso esté en ejecución.

Lo que hace el sistema para bloquearlo es sólo una verificación, y si es el proceso 0 continua por un flujo que lleva a la ejecución de otros procesos, si vemos la imagen 3.32 podremos notar que en `#s98` dirección a la que salta el proceso de borrado verifica primero hay más proceso, esto se logra consultando a la variable del sistema `#c4` que contiene un *id counter* general, o más bien el más alto que no es negativo de los que se encuentran en la zona de procesos, si esté tiene un valor de 0 quiere decir que no hay más, y lo que haría es regresar al proceso 0. Si notamos en la imagen indica que saltaría a `#s141`, y en la imagen hay una continuación directa de la dirección `#s100` a la `#s141` puesto que en medio está la zona de lanzamiento de procesos y es más cómodo tener en la imagen a todo el segmento del proceso 0 junto. Lo que hace el flujo que continua en `#s141` es reiniciar los valores del *id organizer*, *dir organizer* y también coloca en la dirección `#004` el valor de 0 puesto que indica

que el proceso actual ejecutándose es el 0, posteriormente salta de regreso al proceso 0. En caso contrario, en el que si haya más procesos para ejecutar y el *id counter* de la zona de variables del sistema tenga un valor mayor a 0 se procederá al lanzamiento de los procesos.

				Sistema operativo
Guardar 1(px)	s19	1(s1)	LDA s1	Llamemos k al proceso asociado a la dir px
	s20	4011	SHT 11	Se convierte 6(px) en 0(px)
	s21	2(c8)	ADD c8	Se convierte 0(px) en 1(px)
	s22	6(c0)	STO c0	Guarda en c0 1(px)
Verificar si es el proceso 0	s23	1c7	LDA c7	Se obtiene el id del proceso que se estaba ejecutando
	s24	7(000)	SUB 000	Se le resta 1 al id, acc=-1
	s25	3(s98)	BLZ s	Si acc<0 es el proceso 0 y se bloquea, no se borra
	s26	1(c0)	LDA c0	Se obtiene 1(px)
Si no lo es	s27	2(c11)	ADD c11	Se obtiene 1(px)+5, la 1-dir del id del proceso siguiente (k+1)
	s28	6(s29)	STO s29	En s24 se guarda 1(px)+5, al que llamamos 1(py)
	s29	1(py)	LDA py	Se obtiene el id del proceso (k+1)
	s30	3(s52)	BLZ s52	Si acc<0 no hay otro proceso adelante
¿Hay otro?	s31	1(s29)	LDA s29	Se obtiene 1(py) en el acumulador
	s32	6(c17)	STO c17	Se guarda en c17 para usarla más tarde
Inicializar contador de secciones	s33	1(c18)	LDA 004	se Carga 4 para funcionar como contador
	s34	6(c0)	STO C0	Se guarda el contador de secciones del proceso

Figura 3.31: E-CARDIAC C Borrado de proceso parte 1

				Sistema operativo
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
¿está en el proceso 0?	s95	1(c7)	LDA c7	Se obtiene el id organizer
	s96	7(000)	SUB 000	Se le resta un 1, si es menor a 1 significa que es el p0
	s97	3(000)	BLZ 000	Si acc<0 salta al proceso 0
	s98	1(c4)	LDA c4	Se obtiene el id counter, si es 1 hay que saltar a lanzar el proceso
¿Se acabaron los programas?	s99	7(000)	SUB 000	Se le resta un 1 , acc=-1
	s100	3(s141)	BLZ s140	Si acc<0 salta al proceso 0
Reiniciar id organizer a 0	s141	1(000)	LDA 000	
	s142	7000	SUB 000	Carga el id=0 del proceso 0
	s143	6(c7)	STO c7	Asignar a id organizer=0
	s144	6(004)	STO 004	Guardar en 0004 el static Id 0 para el proceso 0
Asigna a dir organizer=p0	s145	1(c3)	LDA c3	Obtiene p4
	s146	7(c11)	SUB c11	Obtiene p1 al restarle 5
	s147	6(c6)	STO c6	Asignar a dir organizer=p0
	s148	8(000)	JMP 000	Saltar a proceso 0

Figura 3.32: SOMC segmento del proceso 0

## Zona de lanzamiento de procesos

En nuestro caso la ejecución continuará en la parte naranja del diagrama, puesto que si tenemos más procesos para ejecutar, y como vemos en la imagen 3.33 continuamos el flujo después de verificar que el *id counter* no es cero, ya en la sección naranja, y aquí las variables del sistema que serán muy importantes son de la #c3 a la #c7, dado que con estas

el sistema puede controlar el orden de la ejecución de los procesos y conocer las direcciones necesarias para lanzar el procesos de forma debida, es decir las direcciones dónde encontrar el contexto del proceso. Siguiendo la imagen 3.17 vemos que en #c3 se guarda siempre la dirección de inicio de la zona de procesos propios del usuario, por eso tiene #p5 y no #p0, por que ese es un proceso del SOM, este valor es una constante que se decide al momento de la implementación del sistema, en la carga misma del sistema. Después en #c4, como ya habíamos visto, tenemos el *id counter* que contiene el valor máximo de los contadores que están en la zona de procesos, su valor por defecto es 0 por que ese es el máximo al inicio, para esté momento en nuestra implementación tiene un valor de 3, pues ya se tienen 3 procesos cargados; #c5 es su símil, pero en lugar de tener el contenido del *id counter* de la zona de procesos tiene la dirección de ese *id counter*, es decir tiene la dirección del último *id counter* de la zona de procesos que no es negativo. De esta forma podemos conocer cuál es el último proceso, y como continuación tenemos a #c6 y #c7 que en lugar de ser “contadores” de *ids* y direcciones, contienen la dirección del *id counter* del proceso que se va a ejecutar y el valor de ese identificador.

Con todo esto podemos lograr una concurrencia en la que tomemos el primer proceso de usuario disponible, se coloquen los valores correspondientes en los **organizadores**(#c6 y #c7) y con ello se pueda tener acceso al contexto completo del proceso y ejecutarlo, posterior a eso se regresa a la zona de lanzamiento y se aumenta el valor de los “organizadores” para ejecutar el siguiente proceso en cola, teniendo un sistema *FIFO First Input First Output* de ejecución, es decir el primero que entra es el primero que se ejecuta, y así sucesivamente hasta llegar al último y detectar que ya no hay más para ejecutar. En ese momento se reinician los organizadores para ir al proceso 0, puesto que mientras se estén ejecutando procesos del usuario los organizadores no regresaran a lanzar el proceso 0, sólo irán de 1 a n, 1 a 5 en esta implementación por que el máximo número de procesos permitidos es 5.

En la imagen 3.34 podemos seguir los pasos descritos anteriormente, hasta la parte en la que salta a #s134 en caso de que haya que reiniciar los organizadores, es decir en caso de que ya no haya más procesos disponibles para ejecutar, para continuar podemos ver en la imagen 3.36 como se realiza esté reinicio, usando la constante que se encuentra en #c3 y que contiene la dirección #p0 para de ahí obtener el contenido de esa dirección.

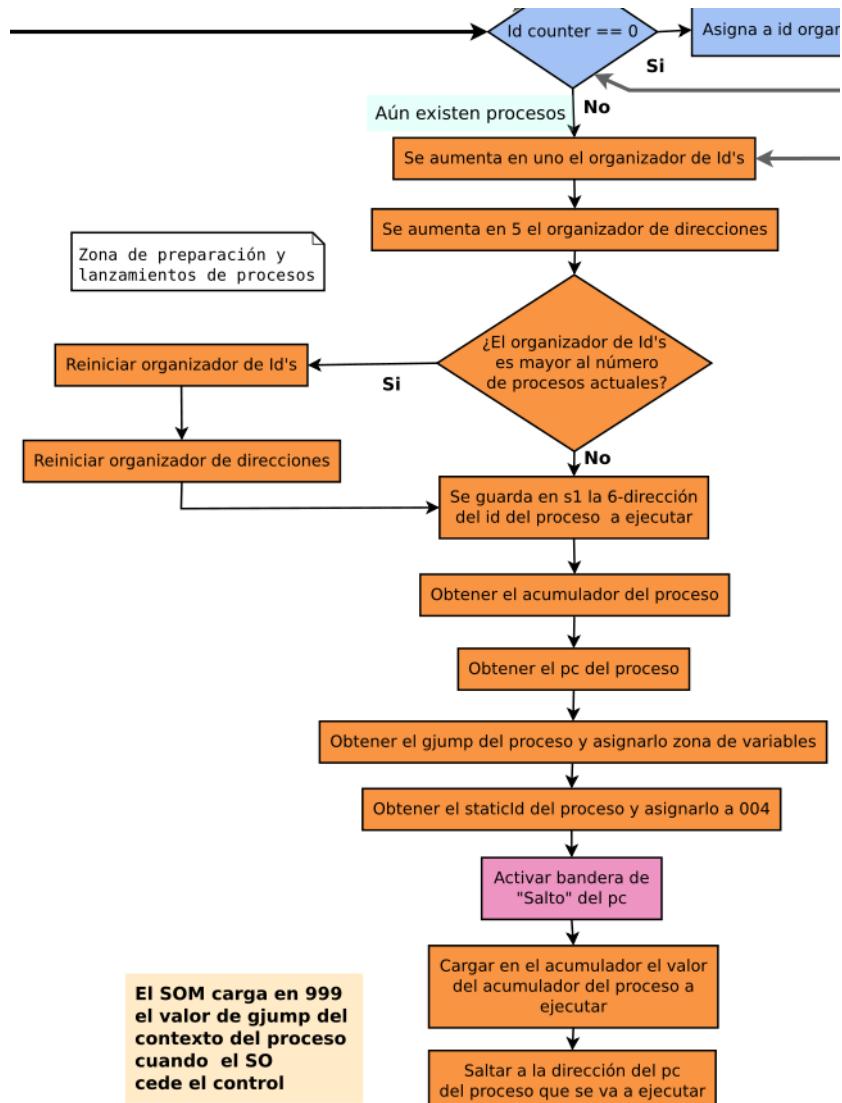


Figura 3.33: SOMC Lanzamiento de procesos

Continuando en #s110, lo que sigue después del rombo en el diagrama, es la colocación de la *6-dir* de dónde se encuentra el identificador del proceso a ejecutar en #s1, para su uso posterior en el borrado y la actualización. Las instrucciones siguientes, que se ven en la parte final de la figura 3.34, toda la 3.35, y la parte inicial de la 3.36 son para obtener el contador de programa, acumulador, salto guardado(*gJump*), y el identificador estático del proceso, así como ir asignándolos a sus respectivos lugares. Para lograr esto se usa de pivote la dirección que se encuentra en #c6 pues con solo ir sumando un uno a ese contenido se puede obtener la dirección de cada uno de los elementos del contexto del proceso, a estas direcciones se les coloca un código de operación de 1 y se van guardando en lugares estratégicos de la memoria,

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>Aumentar Id organizer M[c7]++</i>	s101	1(c7)	LDA c7	
	s102	2000	ADD 000	
	s103	6(c7)	STO c7	Se aumenta el id organizer
<i>Aumentar Dir organizer M[c6]+=5</i>	s104	1(c6)	LDA c6	
	s105	2(c11)	ADD c11	
	s106	6(c6)	STO c6	Se aumenta el dir organizer
<i>Verifica si hay que reiniciar</i>	s107	1(c4)	LDA c4	Se obtiene id counter
	s108	7(c7)	SUB c7	Al id counter se le resta el id organizer, M[c4]-M[c7]
	s109	3(s134)	BLZ s133	si acc<0 tenemos que reiniciar, por que llegamos al ultimo
<i>Sino-s108</i>	s110	1(c6)	LDA c6	Se obtiene la dir del id del proceso a ejecutar
<i>Actualizar s1 para salir al proceso con su dc v acc</i>	s111	2(c9)	ADD c9	Se crea la 6-dir del id del proceso a ejecutar
	s112	6(s1)	STO s1	Se guarda en s1, será el proceso "actual"
	s113	1(c6)	LDA c6	
	s114	2(c8)	ADD c8	Se convierte en la 1-dir del id del proceso a ejecutar

Figura 3.34: SOMC Lanzamiento de procesos parte 1

para que esos valores sean cargados por el acumulador en el momento indicado. Por ejemplo, el caso del contador de programa, la dirección de esté se obtiene en #s115 al sumarle un 1 al contenido del acumulador, que en ese momento era un *1-p5* o un *1774* en la implementación, y ese resultado se guarda en #s119, que a su vez cuando es leído lo que hace es obtener el contenido de la dirección #775, que es el contador de programa del proceso con un código de operación de 8, y esté es guardado en #s133, la última dirección del sistema, dado que es con ese código que se realiza el salto desde la zona del sistema operativo al proceso del usuario.

Previamente, si nos fijamos en la #s117 se suma uno al contenido del acumulador, que en ese momento era de *1775*, con lo que se obtiene *1776* es decir la *1-dir* del acumulador del proceso, y ese resultado se guarda en #s132, la penúltima dirección del proceso, lo que significa que justo antes de saltar carga en el acumulador de la maquina lo que contiene la dirección #776, que es el acumulador que se encuentra en el contexto del proceso, si es su primera ejecución será 0, y en caso contrario será el último valor que tuvo el acumulador antes de que el proceso cediera los recursos. Por otra parte en #s121 lo que sucede es que se carga en el acumulador *1p7(1776* en la implementación) para sumarle otro 1 y obtener *1777*, que es guardada en #s128, y así poder obtener el *gJump* y guardarla en la variable del sistema #c14. Esto es de suma importancia por que es la maquina quien coloca en #999 está variable, que es parte del contexto del proceso, la acción se realiza justo después de ejecutar la instrucción que se encuentra en #s133, puesto que sino lo que haría la maquina por defecto es guardar en #999 el valor de *8(s133)*, pero lo que queremos que guarde es el

último valor que #999 tuvo durante el proceso que se está lanzando, y en caso de que sea la primera ejecución que tenga el valor por defecto.

Lo único que falta es el id estático, su dirección es obtenida en #s124 al sumar otro 1 al acumulador por que en ese momento el valor que tenía era 1777 y así se consigue el 1778 que es guardado en #s126 para obtener el valor del identificador estático y poderlo guardar en #004, que es una dirección reservada del sistema que es usada por la maquina para identificar que proceso se está ejecutando haciendo referencia al identificador estático, por que esté nunca cambia ni se repite, y así en los *outputs* poder colocar que proceso está escribiendo. De ahí la necesidad de un identificador estático para que el usuario conozca que proceso escribió cada salida, por que como se van a ir intercalando los procesos será fácil perderse en a quien corresponde cada salida. Por último, antes de cargar el acumulador y saltar al proceso, lo que se hace en #s130 y #s131 es cambiar el valor de la bandera para permitir saltos, por lo que las dos últimas instrucciones del sistema antes de saltar son, para el *switcher*, instrucciones del usuario y hay que tenerlo en consideración al contar los ciclos que cada proceso tiene.

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>Continuación</i>	<b>s115</b>	<b>2000</b>	<b>ADD 000</b>	Se obtiene la 1-dir del gpc del proceso a ejecutar
	<b>s116</b>	<b>6(s119)</b>	<b>STO s118</b>	En s101 se guarda la 1-dir del gpc del proceso a ejecutar
	<b>s117</b>	<b>2000</b>	<b>ADD 000</b>	Se obtiene la 1-dir del gacc del proceso a ejecutar
<i>Preparación gpc y gcc</i>	<b>s118</b>	<b>6(s132)</b>	<b>STO s131</b>	En 112 se guarda la 1-dir del gacc del proceso a ejecutar
	<b>s119</b>	<b>1(px)+1</b>	<b>LDA px+1</b>	Obtiene el gpc del proceso a ejecutar
	<b>s120</b>	<b>6(s133)</b>	<b>STO s132</b>	Guarda el gpc del proceso a ejecutar en s106
<i>Preparación gjump</i>	<b>s121</b>	<b>1(s132)</b>	<b>LDA s131</b>	Obtiene la 1 dir del gacc
	<b>s122</b>	<b>2000</b>	<b>ADD 000</b>	Obtiene la 1 dir del gjump
	<b>s123</b>	<b>6(s128)</b>	<b>STO s127</b>	Guarda la 1 dir del gjump en 113
<i>Salvar Static ID en 004</i>	<b>s124</b>	<b>2000</b>	<b>ADD 000</b>	Añadir un 1 para obtener el 1(px)+4, SID
	<b>s125</b>	<b>6(s126)</b>	<b>STO s125</b>	Guardar en la siguiente celda
	<b>s126</b>	<b>1(px)+4</b>	<b>LDA px+4</b>	Obtener Static ID del proceso
	<b>s127</b>	<b>6004</b>	<b>STO 004</b>	Guardar en 0004 para usarlo como identificador el proceso que se está ejecutando

Figura 3.35: SOMC Lanzamiento de procesos parte 2

### Lanzamiento de procesos en la maquina virtual

Si dejamos avanzar la ejecución del sistema cuando haya pasado de #s107 y sabiendo que no hay que reiniciar los organizadores observamos en la figura 3.37 que en las variables del sistema que representan a los “organizadores”, #971(#c6) y #972(#c7) están la dirección del *id counter* del proceso que se va a lanzar y el contenido del mismo, respectivamente.

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
Salvar gjump	s128	1(px)+3	LDA px+3	Carga el valor de la gjump
	s129	6(c14)	STO c14	Guarda el gjump en c14 para que la arquitectura lo intercambie
bandera	s130	1(000)	LDA 000	Obtiene el número 1
Bandera	s131	6003	STO 003	Se permiten saltos con bandera==1
	s132	1(px)+2	LDA px+2	Se obtiene acc= acc del proceso a ejecutar
LastDirectionSO	s133	8(xx)	JMP xx	La dir xx representa el valor donde el proceso se ejecutará
Reiniciar dir organizer	s134	1(c3)	LDA c3	Se obtiene la primera dir de la zona de procesos, p3
	s135	6(c6)	STO c6	Se reinicia el dir organizer
	s136	2(c8)	ADD c8	Se crea la 1-dir de inicio en el acumulador
Reiniciar id organizer	s137	6(s138)	STO s137	En s138 se guarda px
	s138	1(px)	LDA px	Se obtiene el id de inicio
	s139	6(c7)	STO c7	Se reinicia el id organizer
Rearresar	s140	8(s110)	JMP s109	Rearresar para lanzar el proceso

Figura 3.36: SOMC Lanzamiento de procesos parte 3

Para observar la situación del sistema antes de lanzar el proceso observemos la imagen 3.38, dónde vemos que en #933(#s133), la instrucción que se va a ejecutar es un 8110, es decir ya va a saltar al proceso, con un acumulador que es igual a 0, y con un *switcher status* que indica que ya se está ejecutando un proceso del usuario. Pero con un *switcher counter* que es igual a 0 por que está desfasado un ciclo debido a que verifica el estado de la bandera justo antes de ejecutar una instrucción, antes de ejecutar lo que hay en #932 verifco la bandera y se reinicio quedándose en 0, antes de ejecutar #933 aún o ha sumado nada a ese ciclo y cuando lo termine le sumará un uno, por lo que el proceso del usuario iniciara con un ciclo ya usado.



Figura 3.37: Estatus de organizadores antes de lanzar el proceso

Para la imagen 3.39 ya vemos al *contador de programa* en #110 y un *counter switcher* en 1, por lo que le quedan 29 ciclos antes de ceder los recursos nuevamente. Ahora ya es turno de que el proceso “pintor” empiece a imprimir los números del 1 al 10, en la imagen 3.40 vemos al proceso justo antes de ceder los recursos, como el *switcher counter* ya tiene el valor de 30 la instrucción que se encuentra en #119 ya no será ejecutada y saltará directamente a la zona de actualización. Pero observemos primero el estatus del proceso, vemos que en el *output* ya ha impreso cuatro valores, el primero lo imprimió cuando “finalizó” el proceso 0, por eso pone como identificador ese número seguido de guiones que indican que ese proceso

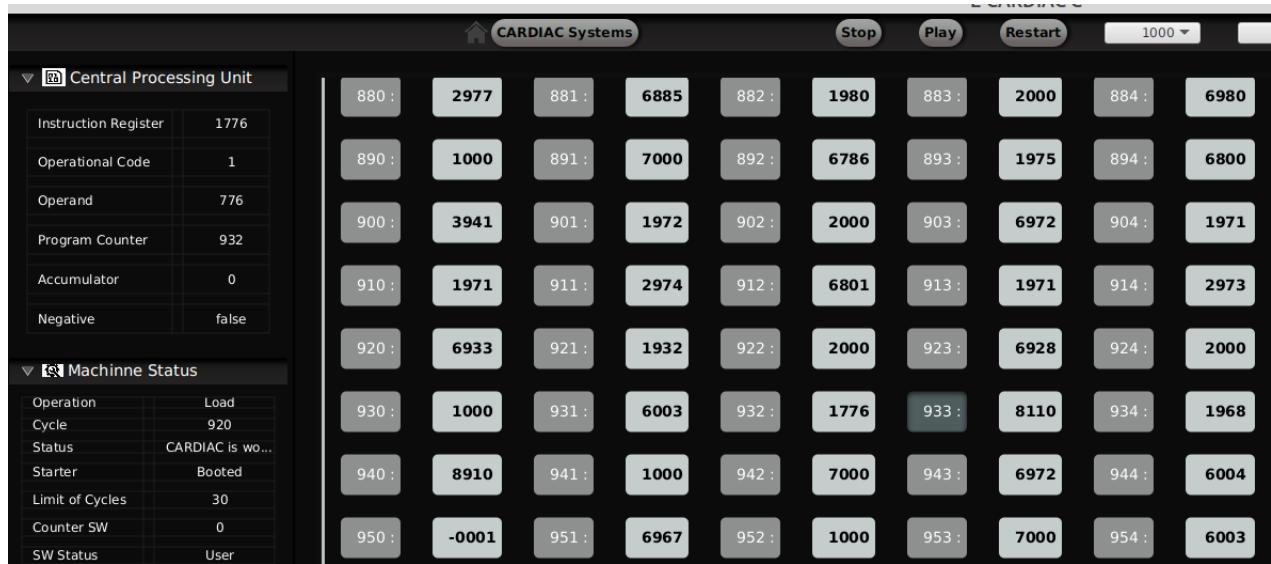


Figura 3.38: Valores del SOM antes de lanzar el proceso

ha finalizado, después ya hay escritos que corresponden al proceso 1, por ello el *id* 0001, que hace referencia al identificador estático y no al contador, puesto que el segundo puede ir cambiando, y cada que nos refiramos a un proceso no nos referiremos a el por su lugar en la lista de procesos, sino por su identificador estático. Por lo que vemos ya escribió tres números y el acumulador tiene el valor de 7, un indicador de los números que faltan por escribir, pero si notamos la instrucción que iba a ejecutar veremos que iba a restarle un 1, puesto que en #105 ya está el número 4 cargado y sólo falta imprimirlo, como la impresión sucede antes de que la condición de parada se efectué la lógica del proceso es correcta.

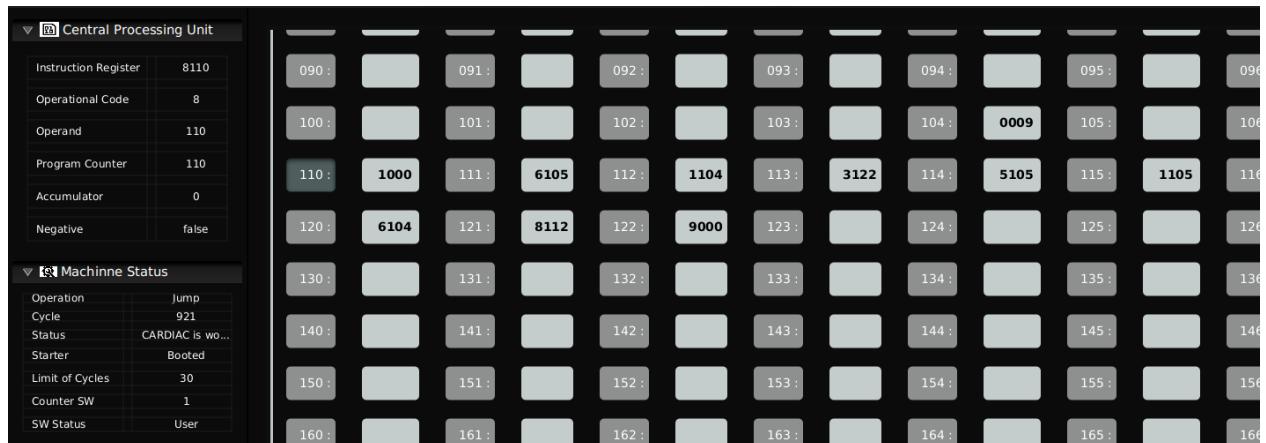


Figura 3.39: Proceso 1 en ejecución

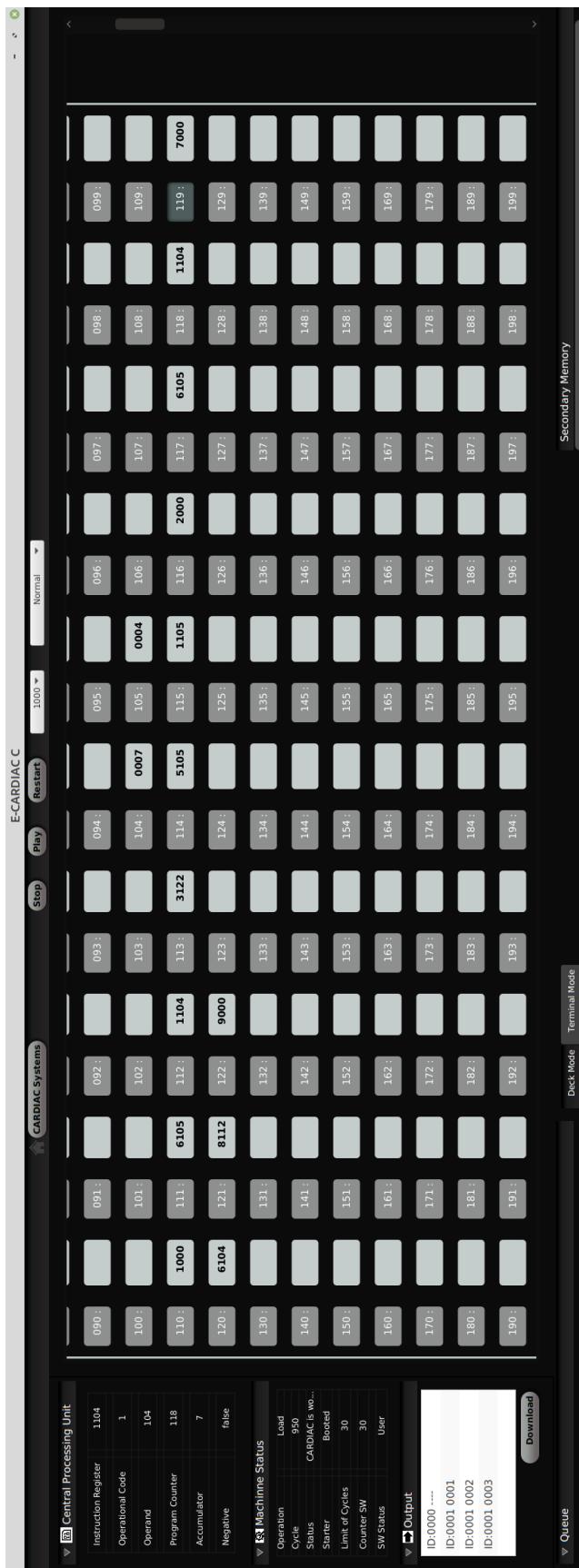


Figura 3.40: Parte final de la primera parte de la ejecución de P1  
102

## Actualizar un proceso

Tenemos un proceso que acaba de ceder sus recursos de nuevo al sistema operativo, por lo que lo primero que hace el sistema es guardar los valores asociados a él. El salto que hace el *switcher* es directo a `#e1`(`#951` en la implementación), el preámbulo, que como vimos lo que hace es guardar el valor del contador de programa, el acumulador, el valor que hay en `#999`, que no se ve afectado por este salto porque no se hace con la instrucción de *jump*, sino por la máquina virtual en sí. También cambia el valor de la bandera, y como en este caso `#e0` tiene un valor diferente de un negativo puesto que no se llegó al preámbulo por una instrucción *halt* el salto que se haga desde el preámbulo será a `#s2`(`#802` en la implementación), a la zona de actualización. Podemos observar el flujo en la zona verde del diagrama general y en la figura 3.41 dónde se hace un acercamiento a esa zona a la cual se llega posterior a haber pasado por el preámbulo.

Para analizar lo que hace la actualización podemos ver también la imagen 3.42 que contiene el código para la actualización. Lo que hace en términos generales es actualizar el contexto del proceso, es decir actualiza los valores del contador de programa, acumulador y del salto guardado. Hace una validación para verificar que la zona de lanzamientos haya echo lo correcto y asignando la *6-dir* a `#s1`, pues será un valor pivote para conocer en qué dirección está el contexto del proceso que estaba en ejecución, como lo que hace en `#s2` es obtener la *6-dir* del *id counter* del proceso, y suponiendo que pasa la validación, lo que sucede es que se le suma un 1, con eso se obtiene la *6-dir* del *gpc* del proceso, es decir que se obtiene la dirección del contador de programa guardado del proceso pero con un código de operación 6, en la imagen 3.43 podemos ver que en `#807`(`#s7`) se encuentra `6775`, y la instrucción que ejecuta justo antes de esa es la que recupera el último contador de programa que tuvo el proceso antes de saltar y que fue resguardado en `#c1` por el preámbulo. Esté mismo sistema se sigue para las otras dos variables, en las que el preámbulo guarda los valores en la zona de variables del sistema en lugares genéricos, y el subproceso de actualizar el proceso coloca esos valores en el lugar correspondiente para cada proceso, y una vez terminado esto el flujo sigue hacia la zona de lanzamiento de procesos.

Pero antes de movernos hacia allá regresemos a la validación y veamos qué sucede si no la

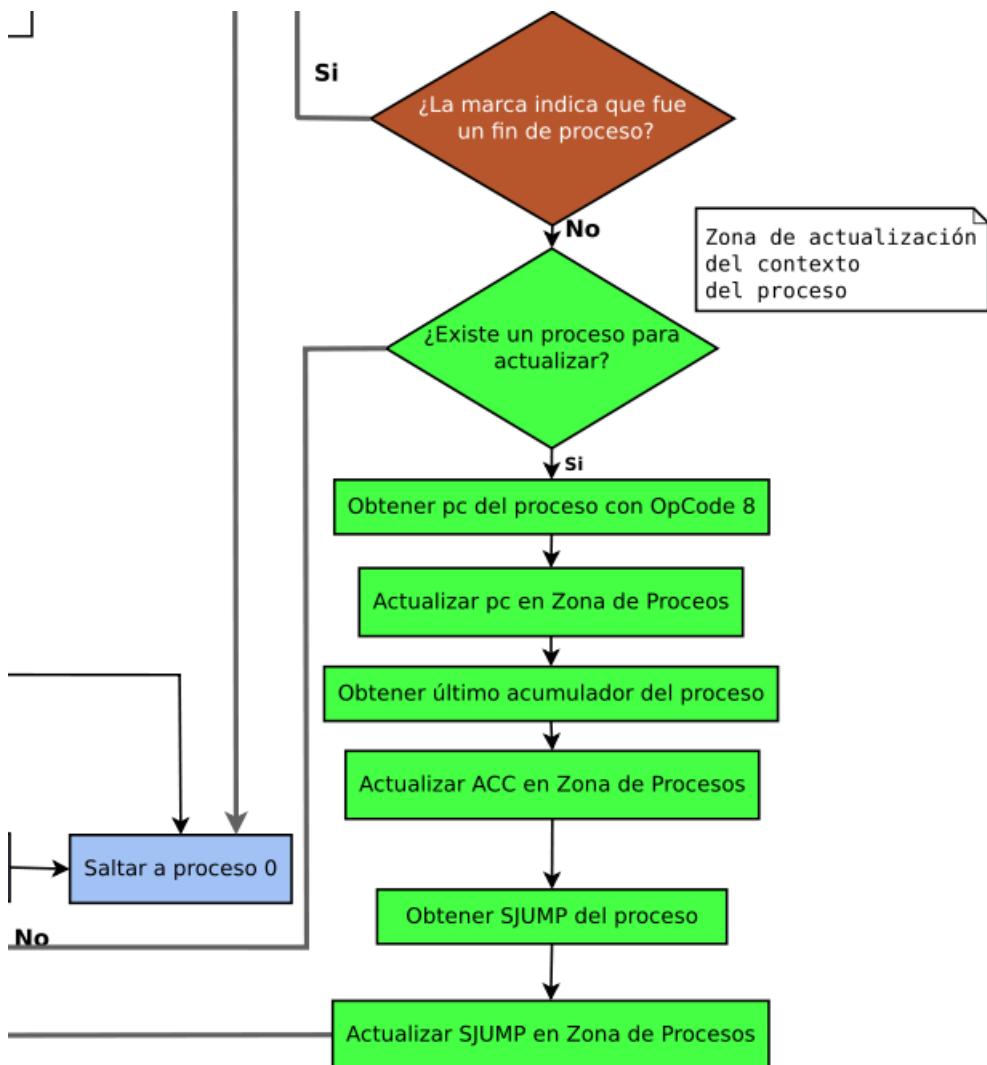


Figura 3.41: Diagrama de actualización de proceso

pasa, en caso de que `#s1` tenga un valor negativo significa que no hay un proceso a actualizar, por lo que se podría generar un error, por ende como vemos también en la imagen 3.44 el flujo sigue hacia el subproceso que verifica si es que ya no hay más procesos para ejecutar y si es es necesario lanzar de nuevo el proceso 0 y reiniciar los organizadores. En caso de que si haya más procesos para ejecutar se irá a la zona de lanzamiento de procesos, misma a la que llega el flujo por la otra rama, en caso de que el contenido de `#s1` no sea negativo, esta rama se mueve directo al lanzamiento de procesos por que como acaba de actualizar uno necesariamente hay al menos un proceso para lanzar.

En la imagen 3.45 vemos el contexto del proceso actualizado, en `#775` está la dirección

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>Validación</i>	<b>s2</b>	<b>1(s1)</b>	<b>LDA s1</b>	El acumulador toma un valor de la forma 6(px)
	<b>s3</b>	<b>3(s98)</b>	<b>BLZ s97</b>	Si acc<0 no hay proceso para actualizar
<i>Actualizar gpc</i>	<b>s4</b>	<b>2000</b>	<b>ADD 000</b>	Se obtiene la 6-dir del gpc del proceso
	<b>s5</b>	<b>6(s7)</b>	<b>STO s7</b>	Se guarda la instrucción 6(px)+1
<i>Actualizar gacc</i>	<b>s6</b>	<b>1(c1)</b>	<b>LDA C1</b>	Se obtiene el último pc del proceso con forma 8(pc)
	<b>s7</b>	<b>6(p5)</b>	<b>STO p5</b>	En p5 se actualiza gpc
<i>Actualizar gjump</i>	<b>s8</b>	<b>1(s7)</b>	<b>LDA s7</b>	Se obtiene la instrucción 6(px)+1
	<b>s9</b>	<b>2000</b>	<b>ADD 000</b>	Para acceder a la 6-dir del gacc del proceso
<i>Saltar</i>	<b>s10</b>	<b>6(s12)</b>	<b>STO s12</b>	En s12 se guarda 6(px)+2
	<b>s11</b>	<b>1(c2)</b>	<b>LDA c2</b>	Se obtiene el último acc del proceso
<i>Actualizar gjump</i>	<b>s12</b>	<b>6(p6)</b>	<b>STO p6</b>	Se actualiza el valor de gacc
	<b>s13</b>	<b>1(s12)</b>	<b>LDA s12</b>	Obtiene la 6(p6) del proceso que se está actualizando
<i>Saltar</i>	<b>s14</b>	<b>2000</b>	<b>ADD 000</b>	Obtiene la 6(p7) del proceso que se está ejecutando
	<b>s15</b>	<b>6(s17)</b>	<b>STO s17</b>	Guarda en e18 el código para guardar en la zona de procesos c14
<i>Saltar</i>	<b>s16</b>	<b>1(c14)</b>	<b>LDA c14</b>	Obtiene c14
	<b>s17</b>	<b>6(p7)</b>	<b>STO p7</b>	Guarda en la zona de procesos correspondiente al proceso el valor saber jump
<i>Saltar</i>	<b>s18</b>	<b>8(s101)</b>	<b>JMP S100</b>	Saltamos a cambiar de proceso

Figura 3.42: SOMC Actualizar proceso

800 :	<b>-0001</b>	801 :	<b>6774</b>	802 :	<b>1801</b>	803 :	<b>3898</b>	804 :	<b>2000</b>
805 :	<b>6807</b>	806 :	<b>1966</b>	807 :	<b>6775</b>	808 :	<b>1807</b>	809 :	<b>2000</b>
810 :	<b>6812</b>	811 :	<b>1967</b>	812 :	<b>0998</b>	813 :	<b>1812</b>	814 :	<b>2000</b>
815 :	<b>6817</b>	816 :	<b>1979</b>	817 :	<b>0998</b>	818 :	<b>8901</b>	819 :	<b>1801</b>

Figura 3.43: Actualizar proceso 1

en la cuál continuará el proceso cuando recupere los recursos, en #776 el valor que tenía el acumulador antes de saltar, y en #777 está el último valor que tuvo la dirección #999 de la maquina durante el proceso 1.

Lo que sigue es que en la zona de procesos, los “organizadores” dicten el siguiente proceso a ser lanzado, que para esté caso será el proceso 2, luego el proceso 3, y entonces vuelve a tocarle el turno al proceso 1 y a seguir la misma lista de ejecuciones hasta que alguno finalice.

### Funcionamiento del borrado de un proceso

Después de varias iteraciones tenemos el siguiente estatus en el sistema, mostrado en la imagen 3.46, en la cual vemos que en el *output* ya se ha escrito hasta el número 9 del proceso 3 y hasta el número 10 del proceso 1. Actualmente los recursos los tiene el proceso 1 que está a punto de saltar a #122, por que lo que hay en el acumulador es negativo. Como la

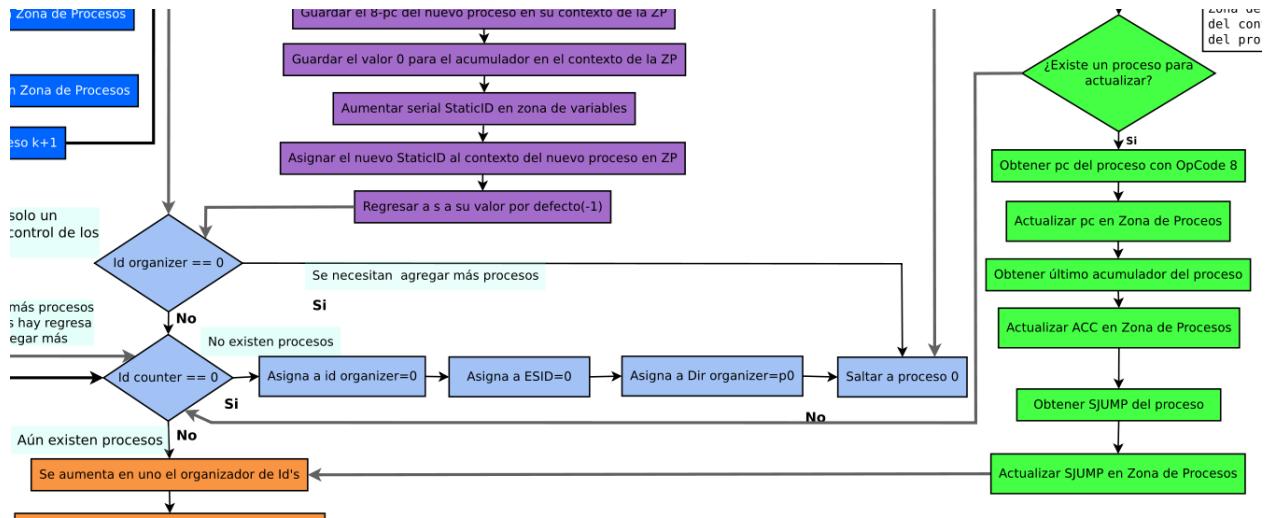


Figura 3.44: Conexión entre actualización de procesos y lanzamiento

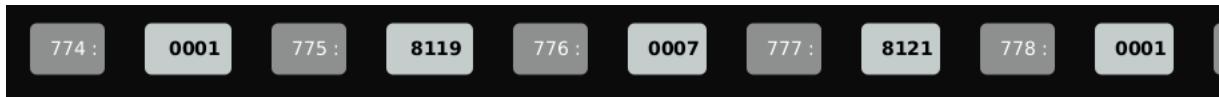


Figura 3.45: Contexto del proceso 1 actualizado

instrucción que está ahí es un *halt* lo que sucede es que se saltará al preámbulo, pero dejando una marca en `#e0` para indicar que se llegó ahí por una finalización de proceso, en la figura 3.47 vemos que en `#950(#e0)` está la marca de un número negativo y que en el *output* lo último que hay es una salida correspondiente al proceso 1, pero con unas líneas que indican que el proceso terminó satisfactoriamente.

Más adelante podemos ver en la figura 3.48 que como el valor que contiene la dirección `#950` es negativo el salto para salir del preámbulo es hacia la dirección `#819`, la zona de borrado de procesos. En el diagrama general y en la imagen 3.49 podemos ver el flujo que sigue un proceso para ser borrado, y como ya vimos anteriormente los primeros pasos son convertir el valor que está en `#s1` de una *6-dir* a una *1-dir* y verificar que no se trate del proceso 0. Para seguir el código podemos ver las imágenes 3.31, que se usó en la explicación de la zona de borrado cuando se trata del proceso 0, además de la 3.51, y 3.52 que continúan con la descripción de las instrucciones que se usan para borrar el proceso.

La mecánica de borrado es simple y explica la necesidad de dos identificadores, si vemos el diagrama notaremos que indica que si no es el proceso 0 hay que borrar el proceso, al que llamamos k y en nuestro caso será el proceso 1. Borrar un proceso lo que quiere decir es

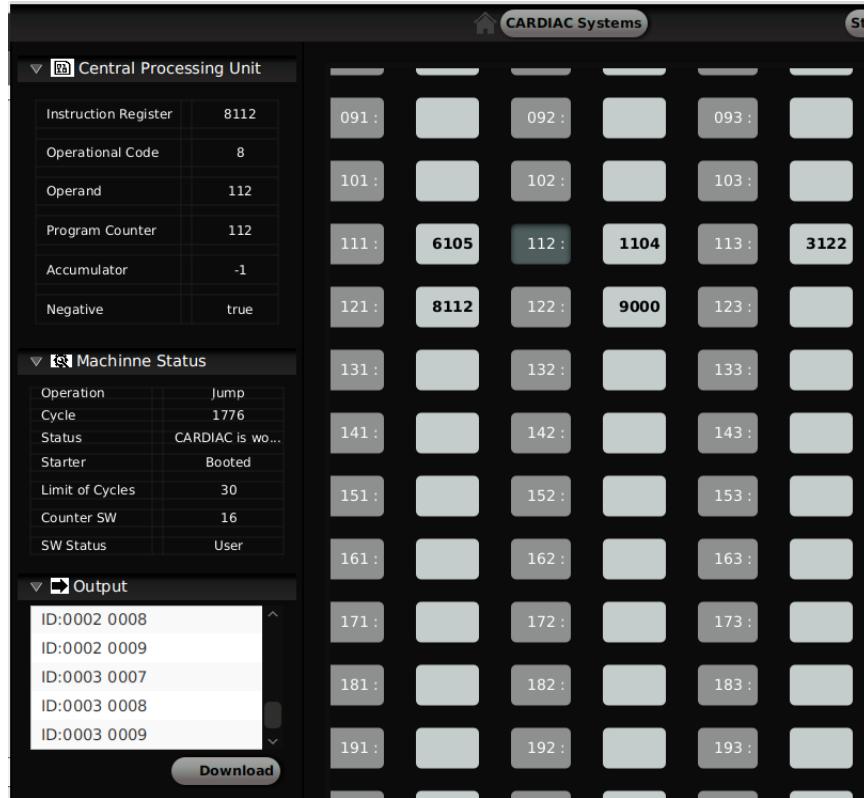


Figura 3.46: Proceso 1 antes de finalizar

borrar su contenido en la zona de procesos, para que cuando los “organizadores” deban elegir un proceso a ejecutar el proceso 1 ya no esté entre ellos. Notarán que no he mencionado el programa como tal que está en la dirección #110 y esto es por que no importa lo que haya en esa dirección, para el sistema lo que importa es lo que hay en la zona de procesos, no importa que ahí aún haya un programa, sino está ligado a la zona de procesos con su correspondiente contexto, para el sistema no existe.

Entonces lo que hace el subprocesso para eliminar un proceso del usuario es volver negativo el *id counter* de dicho proceso si es que es el último en la lista y disminuir el *id counter* general que está en la zona de variables(#c4). Si el proceso a borrar fuese el 3 en este ejercicio ese es el flujo que seguiría y después continuaría hacia el subprocesso azul claro(como en la imagen 3.49) del proceso 0, no hay necesidad de borrar el contenido del resto del contexto, se sobrescribirá cuando haya otro proceso, lo importante es el *id counter*, por que así es como el sistema nota si un proceso está en esa zona.

Pero en nuestro caso el proceso a borrar no es el último en la lista, es el primero, por lo que

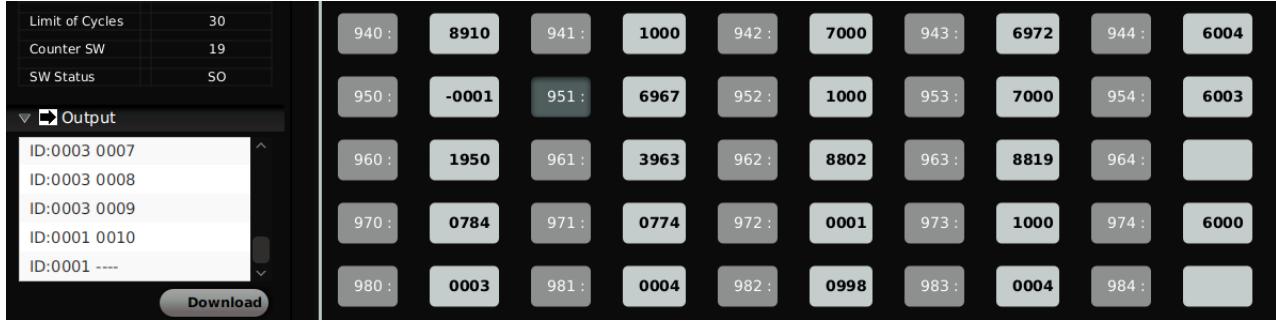


Figura 3.47: Preámbulo en la finalización del proceso 1

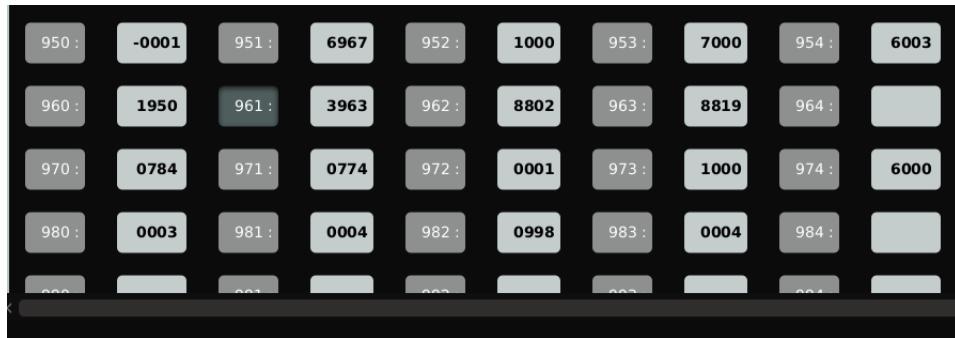


Figura 3.48: Preámbulo en la finalización del proceso 1 a punto de saltar

no se puede realizar la acción antes descrita. Puesto el sistema espera que los procesos sean consecutivos, con un *id counter* que sea ascendente y que con el primer contenido negativo que encuentre en una dirección dónde debe estar un *id counter* se detenga. Por lo tanto lo que hace el sistema en estos casos es recorrer los procesos, el contexto del proceso 2 lo pasa a las direcciones que ocupa el contexto del proceso 1, y el contexto del proceso 3 es transferido a las direcciones que ocupaba el proceso 2; todas las variables del contexto del proceso son movidas, excepto el *id counter*, el cuál no se mueve, por que es un valor ascendente y lo que se hace para borrar el proceso es convertir el *id counter* con el valor más alto, que actualmente tiene un valor de 3, en -1. Por ende el proceso 2 ahora tendrá un *id counter* con un valor de 1, pero guarda su identidad por el identificador estático, así tanto el sistema como nosotros podemos identificar siempre al proceso 3 aunque su *id counter* sea un 1 o un 2.

En la imagen 3.50 vemos la zona de procesos antes de que el proceso 1 sea borrado, aún todos los *id counter* y *static id* son iguales, mientras que en la imagen 3.53 vemos que ahora el *static id* que se encuentra en #778 es el 2, diferente al *id counter* del proceso que ocupa ese contexto, por que se han recorrido los procesos. Mismo caso para el proceso 3 que está

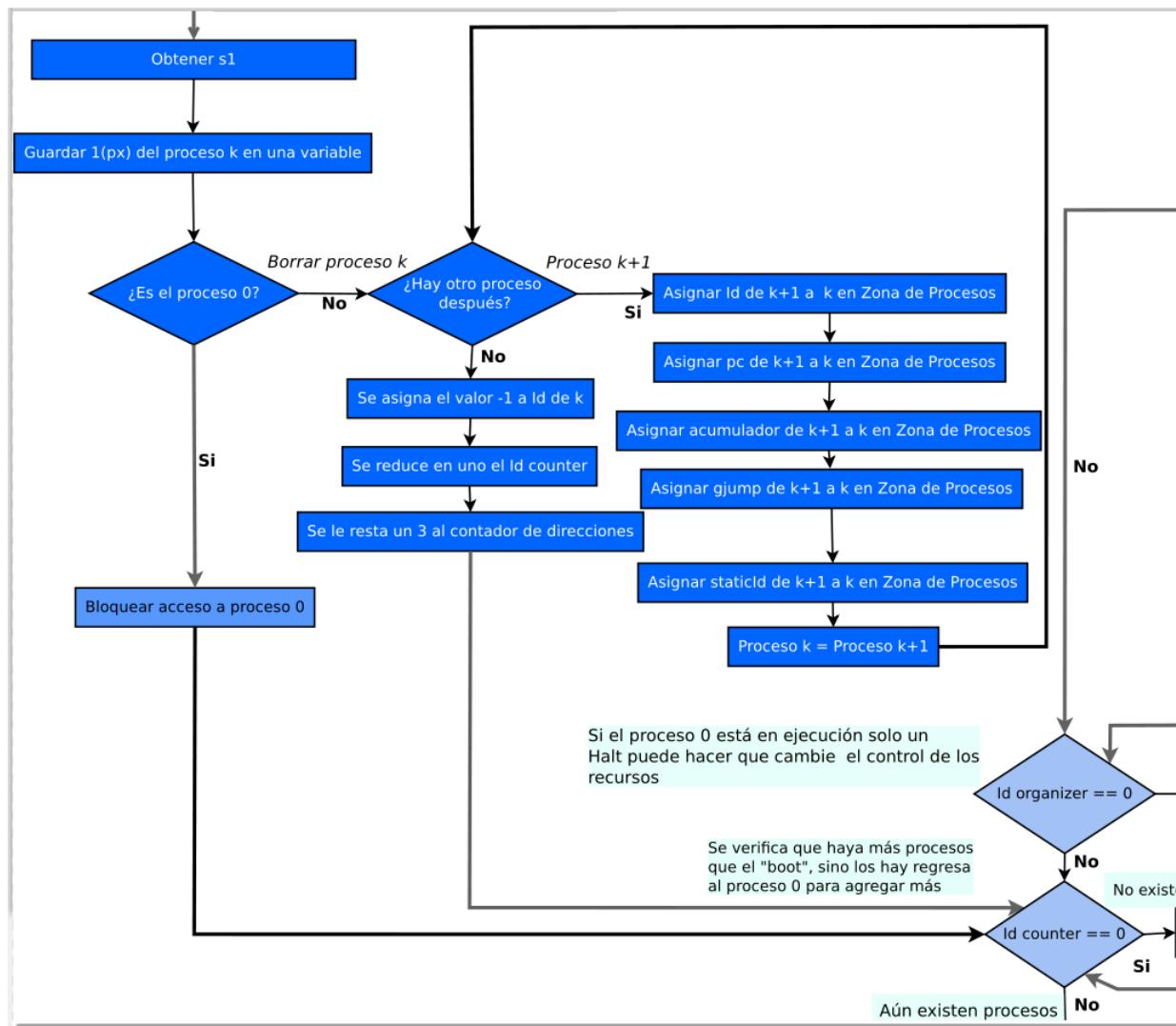


Figura 3.49: Acercamiento a zona de borrado en diagrama

en el contexto que inicia con el *id coutner* 2, y además podemos ver que dónde antes estaba el *id counter* 3, en #784, ahora hay un -1 lo que indica el fin de la lista de procesos, pero que el resto de las variables de contexto aún están ahí y de hecho son los mismos valores que tiene el proceso 3 en sus nuevas direcciones. Y es por que se quedan ahí como basura, se sobrescribirá cuando haya un nuevo proceso, sólo basta que lo que tendría que ser un *id counter* sea negativo para que se marque el final de la lista.

765 :		766 :		767 :		768 :		769 :	<b>0000</b>
770 :	<b>8000</b>	771 :	<b>0000</b>	772 :	<b>8000</b>	773 :	<b>0000</b>	774 :	<b>0001</b>
775 :	<b>8117</b>	776 :	<b>0010</b>	777 :	<b>8121</b>	778 :	<b>0001</b>	779 :	<b>0002</b>
780 :	<b>8317</b>	781 :	<b>0010</b>	782 :	<b>8321</b>	783 :	<b>0002</b>	784 :	<b>0003</b>
785 :	<b>8517</b>	786 :	<b>0010</b>	787 :	<b>8521</b>	788 :	<b>0003</b>	789 :	<b>-0001</b>

Figura 3.50: Zona de procesos antes de que el proceso 1 sea borrado

### ¿Como borrar un proceso en E-CARDIAC C?

Para analizar el borrado desde el código necesario para lograrlo vamos a la imagen 3.31, a partir de la dirección #s27, después de verificar que no se trata del proceso 0. Llaremos  $k$  al proceso que va a ser borrado, entonces como lo que necesitamos es verificar si hay más procesos con *id counter* mayores en #s27 se obtiene la *1-dir* del siguiente proceso, al que llamaremos  $k+1$ , y después cargar el contenido en el acumulador se verifica si es negativo, en caso de que no lo sea quiere decir que hay otro proceso y se tienen que recorrer.

Para recorrer cada una de las variables de contexto del proceso se usará una subrutina que inicia en #s35 y termina en #s49(visible en la figura 3.51 y la 3.52), usando como pivote la dirección que fue obtenida en #s27 y que es guardada en #c17(para el fácil acceso de la subrutina), pues es la dirección del *id counter* del proceso  $k+1$ . Si a esa dirección se le suma uno se obtendrá la dirección que tiene el *gpc* del proceso a recorrer, y si a esa dirección se le resta 5 se obtendrá la dirección dónde está el *gpc* del proceso que está siendo borrado. Es decir, que con solo sumar 1 y restar 5 podemos acceder a todo el contexto del proceso  $k+1$  y copiar cada variable en el lugar correspondiente del proceso anterior,  $k$ (el cuál está siendo borrado o recorrido).

Después de completar el copiado de contexto del proceso  $k+1$  a la zona del proceso  $k$  viene otra iteración, pues la subrutina de copiado esta anidada en la iteración para ir recorriendo los procesos siguientes al que será borrado. En #s50 después de haber movido todas las variables de contexto del proceso  $k+1$  lo que hace el sistema operativo es cargar en el acumulador la *1-dir* del proceso que acaba de mover y salta a #s27. Lo que causa que ahora el proceso  $k+1$  se convierta en el proceso  $k$ , por ende la dirección que ese proceso

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>Verificar si faltan secciones</i>	s35	7000	SUB C11	Se verifica si ya termino con las secciones
	s36	3(s50)	BLZ s50	Si ya se pasaron las 4 secciones salta a ver si hay mas procesos
	s37	1(c17)	LDA c17	Se obtiene 1(py) en el acumulador
	s38	2(c0)	ADD Cc0	Obtiene la dirección del gpc/gacc/gjump/staticid del proceso py
<i>Recorrer las secciones gpc,gacc,gjmp,staticid de py a px</i>	s39	6(s44)	STO s44	Se guarda en s44 para tener obtener el valor de la sección del proceso
	s40	7c11	SUB C11	Obtiene la dirección correspondiente del gpc(u otra sección) del proceso px
	s41	4011	SHT 11	Se convierte 1(px)+u en 0(px)+u
	s42	2(c9)	ADD c9	Se convierte 0(px)+u en 6(px)+u
	s43	6(s45)	STO s45	Se guarda en s45 para que sea cargada la sección en px
	s44	1(py)+u	LDA py+u	
	s45	6(px)+u	STO px+u	
<i>Aumentar contador de secciones</i>	s46	1(c0)	LDA c0	Obtener contador de contextos
	s47	7000	ADD 000	Se le resta uno al contador de contextos
	s48	6(c0)	STO c0	Lo guarda en c0 de nuevo
<i>Regresar a recorrer</i>	s49	8(s35)	JMP s35	

Figura 3.51: SOMC Borrar proceso parte 2

ocupaba ahora tiene que ser limpiada, ya sea recorriendo otro proceso(si existe) o dejando el *id counter* de ese proceso con un valor de -1.

Se llega al final cuando el contenido que se carga en el acumulador después de ejecutar la instrucción que está en #s29 es negativo, es decir cuando el contenido del *id counter* del proceso  $k+1$  es negativo, o mejor dicho cuando ya no existe otro proceso después. Entonces salta a #s52, dónde se obtiene la dirección del *id counter* del proceso  $k$  y se actualiza con un valor de -1, este paso salto sin tocar las iteraciones anidadas se da cuando el proceso a borrar es el último de la lista y no se tienen que recorrer procesos.

Lo que sigue es actualizar el valor de los contadores de identificadores y de direcciones que marcan el último proceso al cuál los “organizadores” pueden acceder en la zona de procesos, esto se aprecia en la figura 3.52, y posteriormente salta hacia la el subproceso que verifica si hay más procesos para ejecutar o si hay que lanzar el proceso 0 y reiniciar los organizadores.

En la figura 3.54 vemos los contadores de #c4(#969) y #c5(#970) antes de que el proceso 1 fuese borrado. Se marcaba un 3 en el contador de identificadores indicando que había 3 procesos activos, y un 784 en el contador de direcciones indicando la dirección del último *id counter* disponible. A diferencia de la figura 3.55, dónde se ve como se encuentran las variables después de haber borrado el proceso 1, indicando que el máximo de procesos que hay es 2 y que el último *id counter* que se puede encontrar está en 779.

Algo curioso pasa con los organizadores, y es que como el los organizadores lanzan los

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
¿Hay proceso después de (k+1)?	s50	1(s29)	LDA S29	Se obtiene 1(py) en el acumulador
"Si" de s30	s51	8(s27)	JMP s27	Va a verificar si hay otro proceso adelante
Borrado	s52	1(s29)	LDA s29	Se carga la 1-dir del id del proceso k+1
	s53	7(c11)	SUB c11	Se obtiene la 1-dir del id del proceso k, acc=-5
	s54	4011	SHT 11	Convierte 1(px) en 0(px)
	s55	2(c9)	ADD c9	Convierte 0(px) en 6(px)
	s56	6(s58)	STO s58	Guardar en s58 6(px)
Borrado	s57	1(c10)	LDA c10	Se cambia el valor del acumulador, acc=-1
	s58	6(px)	STO px	El proceso con id en la dir px ya no es accesible
M[c4]-	s59	1(c4)	LDA c4	Carga el id counter
	s60	7(000)	SUB 000	Se le resta 1 al id counter
	s61	6(c4)	STO c4	Guardar en c4 el nuevo valor
M[c5]=M[c5]-5	s62	1(c5)	LDA c5	Cargar el dir counter en el acumulador
	s63	7(c11)	SUB c11	Se le resta 4 al dir counter para estar a la par con el id counter
	s64	6(c5)	STO c5	Se guarda en c5
	s65	8(s98)	JMP S97	Salta a ver si el id counter es el ultimo

Figura 3.52: SOMC Borrar proceso parte 3

765 :		766 :		767 :		768 :		769 :	0000
770 :	8000	771 :	0000	772 :	8000	773 :	0000	774 :	0001
775 :	8317	776 :	0010	777 :	8321	778 :	0002	779 :	0002
780 :	8517	781 :	0010	782 :	8521	783 :	0003	784 :	-0001
785 :	8517	786 :	0010	787 :	8521	788 :	0003	789 :	-0001

Figura 3.53: Zona de procesos después de borrar el proceso 1

procesos de acuerdo a su posición en la lista, es decir de acuerdo a su *id counter*, como se acaba de lanzar el proceso con *id counter* 1 el que sigue es el proceso con *id counter* 2, por lo que se estaría saltando para está ejecución al proceso estático 2 que ahora tiene *id counter* igual 1 y ejecutaría el proceso 3 con *id counter* 2. Es por ello que si nos fijamos en la imagen 3.56 que muestra la foto final de la memoria cuando los tres procesos han terminado, y que también muestra las salidas de estos, vemos que primero termina el proceso 3 y posteriormente termina el proceso 2.

Como la salida es bastante grande la podemos descargar en un archivo de texto plano, como se muestra en la imagen 3.57, dónde vemos la salida completa que produjo la ejecución de los tres “pintores” concurrentemente, como tenemos el identificador estático asociado a cada salida es fácil identificar a quien corresponde cada parte del texto.

965 :	<b>1000</b>	966 :	<b>7999</b>	967 :	<b>-0001</b>	968 :	<b>0774</b>	969 :	<b>0003</b>
970 :	<b>0784</b>	971 :	<b>0774</b>	972 :	<b>0001</b>	973 :	<b>1000</b>	974 :	<b>6000</b>
975 :	<b>-0001</b>	976 :	<b>0005</b>	977 :	<b>0002</b>	978 :	<b>8000</b>	979 :	<b>8121</b>

Figura 3.54: Variables del sistema antes de borrar el proceso 1

965 :	<b>0000</b>	966 :	<b>7999</b>	967 :	<b>-0001</b>	968 :	<b>0774</b>	969 :	<b>0002</b>
970 :	<b>0779</b>	971 :	<b>0779</b>	972 :	<b>0002</b>	973 :	<b>1000</b>	974 :	<b>6000</b>
975 :	<b>-0001</b>	976 :	<b>0005</b>	977 :	<b>0002</b>	978 :	<b>8000</b>	979 :	<b>8121</b>

Figura 3.55: Variables del sistema después de borrar el proceso 1

Machine Status

Operation	Input
Cycle	2164
Status	CARDIAC is ...
Starter	Booted
Limit of Cycles	30
Counter SW	49
SW Status	SO

Output

- ID:0001 ----
- ID:0003 0010
- ID:0003 ----
- ID:0002 0010
- ID:0002 ----

Download

770 :	<b>8000</b>	771 :	<b>0000</b>	772 :	<b>8000</b>	773 :	<b>0000</b>	774 :	<b>-0001</b>
780 :	<b>8517</b>	781 :	<b>0010</b>	782 :	<b>8521</b>	783 :	<b>0003</b>	784 :	<b>-0001</b>
790 :		791 :		792 :	<b>8000</b>	793 :		794 :	<b>-0001</b>
800 :	<b>-0001</b>	801 :	<b>6774</b>	802 :	<b>1801</b>	803 :	<b>3898</b>	804 :	<b>2000</b>
810 :	<b>6812</b>	811 :	<b>1967</b>	812 :	<b>6786</b>	813 :	<b>1812</b>	814 :	<b>2000</b>
820 :	<b>4011</b>	821 :	<b>2973</b>	822 :	<b>6965</b>	823 :	<b>1972</b>	824 :	<b>7000</b>
775 :	<b>8317</b>	776 :	<b>0010</b>	777 :	<b>8321</b>	778 :	<b>0002</b>	779 :	<b>-0001</b>
785 :	<b>8517</b>	786 :	<b>0010</b>	787 :	<b>8521</b>	788 :	<b>0003</b>	789 :	<b>-0001</b>
795 :		796 :		797 :	<b>8000</b>	798 :		799 :	<b>-0001</b>
805 :	<b>6807</b>	806 :	<b>1966</b>	807 :	<b>6785</b>	808 :	<b>1807</b>	809 :	<b>2000</b>
815 :	<b>6817</b>	816 :	<b>1979</b>	817 :	<b>6787</b>	818 :	<b>8901</b>	819 :	<b>1801</b>
825 :	<b>3898</b>	826 :	<b>1965</b>	827 :	<b>2976</b>	828 :	<b>6829</b>	829 :	<b>1779</b>

Figura 3.56: Salidas finales de los procesos

```
CARDIAC_Output_2024...-06 21:01:05.433.txt ×
1 ID:0000 ----
2 ID:0001 0001
3 ID:0001 0002
4 ID:0001 0003
5 ID:0002 0001
6 ID:0002 0002
7 ID:0002 0003
8 ID:0003 0001
9 ID:0003 0002
10 ID:0003 0003
11 ID:0001 0004
12 ID:0001 0005
13 ID:0001 0006
14 ID:0002 0004
15 ID:0002 0005
16 ID:0002 0006
17 ID:0003 0004
18 ID:0003 0005
19 ID:0003 0006
20 ID:0001 0007
21 ID:0001 0008
22 ID:0001 0009
23 ID:0002 0007
24 ID:0002 0008
25 ID:0002 0009
26 ID:0003 0007
27 ID:0003 0008
28 ID:0003 0009
29 ID:0001 0010
30 ID:0001 ----
31 ID:0003 0010
32 ID:0003 ----
33 ID:0002 0010
34 ID:0002 ----
```

Figura 3.57: Salidas finales de los procesos en texto plano

## Guía rápida de uso

Con el repaso hecho pudimos ver el ciclo de vida de un proceso, desde su nacimiento como programa hasta que entrega resultados al usuario, durante ese recorrido conocimos el funcionamiento interno tanto de **E-CARDIAC C** como del sistema operativo mínimo **SOMC** que está adaptado a la maquina para generar una ejecución concurrente de procesos de forma que al usuario le sea practica la ejecución, veamos en la siguiente lista los pasos a seguir del usuario para ejecutar programas en la maquina virtual.

1. Diseñar un programa de acuerdo a la arquitectura de la maquina con el lenguaje establecido para **E-CARDIAC C** en la versión elegida(de 1000 celdas o más).
2. Agregar a ese programa instrucciones que indiquen en que lugar de la memoria cargar cada instrucción, de forma que puedas tener una “tarjeta” con instrucciones pareadas dónde la primera indique la dirección de memoria que ocupará la segunda.
3. El último par de instrucciones de este estilo que debe tener la tarjeta debe contener como segunda instrucción la operación *Halt* que indica el final del programa.
4. Posterior a está instrucción final agregar otro par dónde la primera será *0800* y la segunda la dirección de inicio del programa con un código de operación 8. Dónde *0800* indica que se cargue la segunda en la primer dirección del SOM en la implementación de 1000 celdas hecha.
5. Para finalizar agregar la instrucción preestablecida, *8985* en la implementación, que indica el salto a la dirección de inicio del preámbulo y la única a la que el usuario tiene permitido saltar directamente, con eso se iniciará la carga del programa para convertirlo en un proceso del SOM.
6. Esperar a que se cargue el programa y que el proceso 0 tenga de nuevo el control para que el usuario siga añadiendo procesos, con un máximo de 5 en está implementación.
7. Si el usuario ha cargado los programas que necesitaba cuando este en la dirección #000 esperando para cargar una instrucción que será leída en el siguiente ciclo, colocar *9000*,

que indica el final del proceso 0, y será utilizada no para borrar el proceso sino para pausarlo y empezar la ejecución de los procesos que el usuario ha cargado.

8. Esperar la ejecución de los procesos y cuando haya terminado podrá descargar los resultados del área de *output* dónde cada salida indica a qué proceso pertenece(según su identificador estático) y si un proceso finaliza se imprimirá el identificador seguido de varios guiones medios.
9. Después de finalizar todos los procesos en ejecución el sistema vuelve a lanzar el proceso 0 para que el usuario pueda seguir añadiendo procesos.

Con esta guía rápida el usuario puede crear tarjetas que sigan las pautas necesarias para que la ejecución de los procesos en la maquina virtual resulte exitosa y poder apreciar de la ejecución concurrente a nivel de procesos.

### 3.3. E-CARDIAC PC

Pensar en paralelismo no es una tarea sencilla para nuestras mentes, aunque a veces así lo pareciera. Basta pensar en las cantidad de actividades que puedes realizar al mismo tiempo, generalmente es sólo una y si son varias más bien son actividades que realizas “concurrentemente” y dan la apariencia de ser al mismo tiempo. Un ejemplo sería en la cocina, mientras cortas vegetales el aceite se puede estar colocando en el sartén, o quizá cuando pláticas por chat y con alguien en persona “al mismo tiempo” y vas poniendo más atención en determinada conversación. Pero también hay actividades que podemos realizar realmente al mismo tiempo, aunque para eso son importantes los diversos sentidos que tenemos y como los orientamos, por ejemplo cuando leemos un libro y escuchamos música al mismo tiempo, es difícil poner atención a ambos al mismo tiempo, pero si dejas de escuchar la música de inmediato te das cuenta que ya no está, de cierta manera estamos ejecutando dos acciones en paralelo sin prestar total atención a ambas.

Es por eso que pensar en paralelismo, aunque parezca intuitivo, es muchas veces antinatural, las acciones concurrentes nos son muy naturales, pero pensar en acciones que se ejecuten en paralelo nos lleva muchas veces a acciones que más bien se ejecutan concurrentemente. Fue esté problema el más importante de resolver al momento de diseñar un modelo paralelo de CARDIAC, pues esté al ser un modelo pensado para que el usuario sea una especie de “cpu” que va resolviendo los cálculos y cambiando los datos se tiene que repensar para que el usuario que no puede hacer dos acciones al mismo tiempo las pueda comprender. Y es que analicemos la situación, cuando usamos una computadora que ejecuta procesos en paralelo usualmente estamos prestando atención a uno de ellos mientra el otro u otros están de fondo con su ejecución, están afectando a sentidos (como el auditivo) o bien esperamos el resultado de cálculos en paralelo que convergen hacia un solo resultado, no prestamos atención al como se ejecutan los dos o más procesos al mismo tiempo.

Para hacer más intuitivo el mostrar procesos en paralelo decidí añadir un co-procesador al modelo de computo que tenemos, de esta forma el procesador principal se encargará de unas tareas y el co-procesador de otras, haciendo un símil a lo que hace nuestra mente con los diversos sentidos, cada procesador estará centrado en ciertas actividades mientras ejecutan

procesos al mismo tiempo.

Para mostrar tal paralelismo se creo la siguiente evolución del modelo, *Electronic CARD-board Illustrative Aid to Concurrent Parallel Computation*(E-CARDIAC PC), o ayuda ilustrativa de cartón electrónico para la computación paralela y concurrente, por sus siglas en inglés. Al igual que el modelo anterior su nombre hace referencia también a un hecho histórico en la computación, la computadora personal.

### 3.3.1. Arquitectura renovada para un modelo paralelo

Para entender esté nuevo modelo lo primero que hay que hacer es ver el diagrama de su arquitectura, que ha recibido bastantes modificaciones para añadir un co-procesador, aunque será difícil diferenciar al procesador principal del secundario por que ambos realizarán tareas muy importantes para el modelo de computo. En la figura 3.58 se encuentra el diagrama para está arquitectura, que mantiene la misma memoria principal que su antecesor, así como los buses, la memoria secundaria y el output; todo lo demás se ha modificado o en algunos casos duplicado.

Empecemos por lo más llamativo, los dos cpu's, uno en azul llamado *cpu loader* y otro en rojo *cpu executor*. Cada uno tendrá tareas particulares para el modelo, el principal es el *loader* o cargador puesto que esté es el que está conectado a la memoria secundaria y al *input* principal, la entrada que se conecta a la memoria secundaria y al usuario directamente por medio del tipo de entrada de tarjetas, la entrada masiva de información. Mientras que el *executor* o ejecutor es el co-procesador por que sólo realizará un número determinado de tareas que el usuario hay solicitado desde el cpu principal. Mencionaba que sería difícil de diferenciar al principal del que no lo es por que el segundo, el ejecutor, será el encargado de ejecutar como tal los procesos, el cargador sólo se encarga de cargar el sistema operativo y de permitir al usuario la interacción con la maquina. Aunque el ejecutor también tiene una conexión a un método de entrada, el método que en la maquina está en la pestaña de *terminal mode*, pues habrá procesos que requieran de información del usuario y por tanto el “ejecutor” también requerirá de una forma de que el usuario pueda interactuar.

Podemos también observar que el cpu principal tiene menos componentes que el cpu de E-CARDIAC C, puesto que solo mantiene el *starter*, para poder cargar el sistema operativo

mínimo ya que es el que tiene la conexión a esa entrada, pero el *switcher* como no lo necesita se quedo únicamente en el “cpu ejecutor”. No necesita al *switcher* porque una vez que se cargue el sistema operativo mínimo sólo el usuario usara el cpu principal para estar cargando programa por medio del proceso 0 y es el usuario mismo el que da la instrucción de ceder el control al SOM para que esté añada el proceso que será ejecutado por el cpu secundario y regresará de inmediato al proceso 0 para que el usuario pueda seguir añadiendo programas.

Mientras que el “cpu ejecutor” si necesita de un *switcher* porque estará cambiando el control de los recursos entre el sistema operativo mínimo y los procesos que este ejecutando. Adicionalmente requiere de un elemento extra en el hardware, un elemento que he llamado “*waiter*” porque se encarga de vigilar si hay procesos en la zona de procesos para continuar con la ejecución, en caso de que no haya mantiene al ejecutor en una especie de pausa hasta que detecta que hay algún proceso a ejecutar, esto lo hace por medio del bus azul que lo conecta directamente a la memoria y por el cuál recupera el valor que está guardado en #c4, que es el contador de identificadores general y que como sólo tiene valores menores a 9 no necesita más de un dígito; y tiene otro bus que sale de él hacia el contador de programa, un bus de control, que le indica al contador que puede seguir adelante. Y realiza esa acción por que la forma de activar al *waiter* es cuando el contador de programa apunta a la dirección #s97 del sistema operativo, en ese momento por el bus de control que conecta la memoria principal al *waiter* manda una señal para que esté se active y no deje avanzar al contador de programa hasta que haya un valor distinto de 0 en #c4.

Como podrán haber notado se intento duplicar lo menos posibles, haciendo que no sea una arquitectura muy costosa con dos procesadores exactamente iguales, sino dos procesadores que se complementan, de hecho la *ALU* está pensada para que no funcionen todas las operaciones en los dos procesadores, puesto que algunas serían innecesarias. De esta forma se busca economizar en la arquitectura para añadir la menor cantidad de hardware, y tener una arquitectura que permita tener un sistema operativo mínimo que no aumente en complejidad en comparación con el modelo anterior.

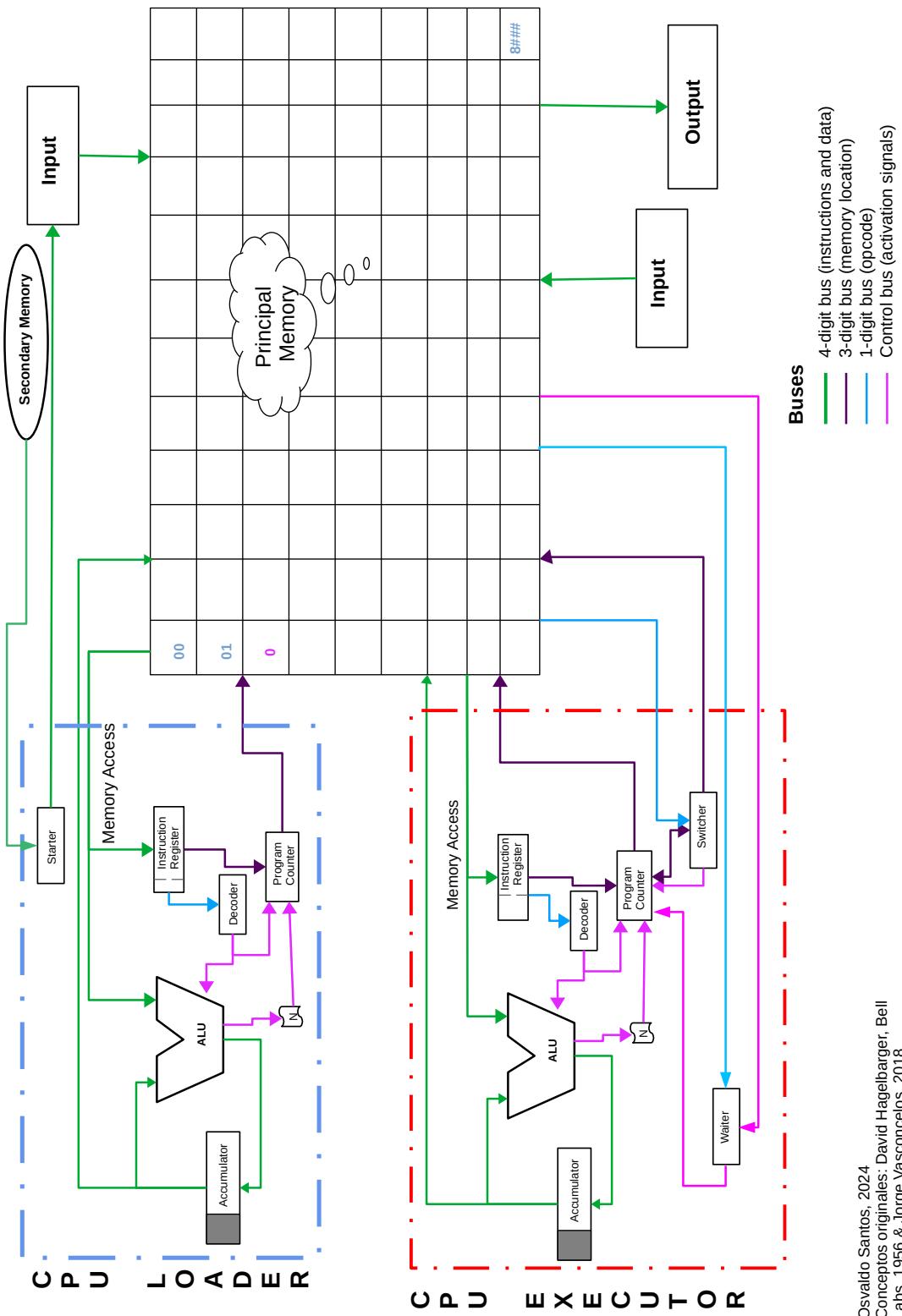


Figura 3.58: Arquitectura Paralela

### 3.3.2. Un sistema operativo para dos procesadores

El reto ahora es pensar en un sistema operativo que pueda manejar estos cpu, para ello se busco reducir los costos y aumentar la eficiencia, para lograr esto fue fundamental disminuir al máximo las interacciones de ambos en la memoria, pues al ser un modelo que comparte memoria se podrían dar situaciones en las que el segundo cpu cambie variables que el primero usa y produzca errores, o que el segundo espere valores del primero y el primero nunca los entregue lo que terminará causando que el segundo no pueda avanzar. E incluso peor, el primer proceso podría estar esperando el resultado del segundo para avanzar, y el segundo el del primero, generando así una muerte por inanición o starvation. Esta situación también es común en procesos concurrentes que comparten recursos, pero puede ser más clara con procesos en paralelo, para disminuir estos problemas sería necesario añadir mecanismos de sincronización de procesos para mantener la integridad de la memoria compartida, como nuestros recursos son muy limitados lo mejor será evitar al máximo estas situaciones, prevenirlas.

La principal manera de prevenirlo será limitando el acceso a la memoria de cada cpu desde el sistema operativo, desde la figura 3.59 podemos ver el diagrama para el nuevo sistema operativo *SOMPC*. En esté diagrama podemos ver una clara separación en dos grupos, a la izquierda los segmentos de borrado, actualización y lanzamiento de procesos; y a la derecha el de añadir un nuevo proceso, e incluso el segmento del *preámbulo* está particionado(no se conecta el del grupo uno con el del dos). Es decir que no hay posibilidad que las instrucciones del grupo uno, que ejecutara el *cpu executor* tengan un conflicto directo con las del grupo dos, que ejecutara el *cpu loader*, puesto que tienen entradas y salidas diferentes y no hay saltos que puedan llevar de algún segmento del grupo uno al segmento del dos, son independientes. De esta forma cada cpu tiene restringidos espacios de memoria desde el mismo diseño sistema operativo.

Evidentemente habrá situaciones que se escapen a esta medida, como sucede también con las computadoras convencionales, es muy difícil, por no decir imposible evadir todos los posibles errores o problemas de sincronización que se presentan cuando trabajas con más de un procesador. Pero esta medida es clave para reducir el impacto de los problemas de

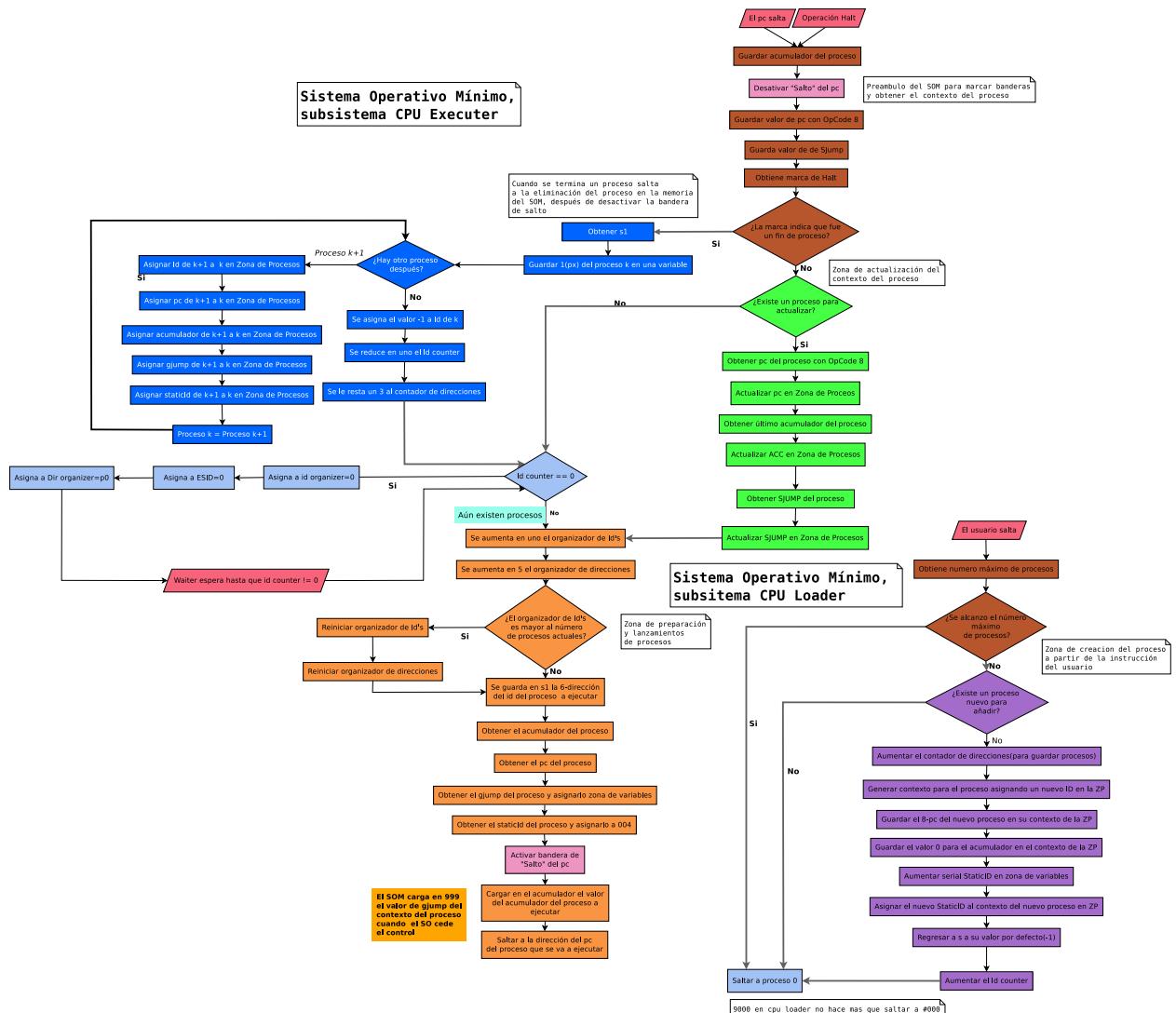


Figura 3.59: Diagrama de Sistema Operativo Mínimo Paralelo

sincronización al mantener al margen cada cpu con sus tareas, y hacer más intuitivo su funcionamiento, al contrario de un modelo de múltiples procesadores que pudieran realizar exactamente las mismas tareas, que en principio es más poderoso, pero también mucho más costoso en su funcionamiento y en la comprensión.

### 3.3.3. Funcionamiento del sistema operativo mínimo

Para explorar el sistema operativo revisaremos por separado lo que ejecuta cada CPU, pues al final son procesos del sistema operativo independientes entre sí generalmente, para terminar en su funcionamiento conjunto y ser capaces de entender como sin interacciones directas, a través del sistema operativo los dos cpu's tienen un funcionamiento homogéneo para darle al usuario un experiencia de ejecuciones en paralelo.

Será importante además, considerar en las siguientes explicaciones que la base es el sistema operativo mínimo de E-CARDIAC C o simplemente C en esté texto, por lo que fácilmente el 90 % del código y funcionamiento del sistema serán idénticos al de C, por lo que será más fácil de entender como funciona esté sistema operativo mínimo mejorado.

#### Sistema operativo para CPU Loader

Empecemos con el CPU Loader por que es el que arrancará las operaciones de la maquina y además es el más corto en su funcionamiento. En la imagen 3.60 vemos un acercamiento a todo lo que realizará esté cpu para el sistema operativo, tiene una pequeña sección de preámbulo que hace lo mismo que en C(E-CARDIAC C), si el usuario salta desde el proceso 0 para añadir un nuevo proceso (proceso del usuario ahora), primero el preámbulo verifica que no se haya alcanzado el número máximo de procesos, en caso de que si salta al proceso 0, pero ahora está instrucción de saltar al proceso 0 no se conecta con nadamás que el segmento de añadir un proceso nuevo. Así mismo si no hay un proceso nuevo para añadir se regresa al proceso 0, y si se agrega con éxito el proceso del usuario se regresa al proceso 0, todo termina en el proceso 0 para que el usuario de nueva cuenta tenga posibilidades de añadir más procesos. Desde la imagen 3.61 podemos ver que la parte que vemos en el diagrama son las instrucciones desde #e14 hasta #e17 únicamente, y tal como pasaba en el SOM de C la única forma de ejecutar esas instrucciones es si el usuario salta a añadir un proceso pues salta a esa dirección, el código previo del preámbulo inevitablemente saltará antes de llegar a esa sección.

Ahora, un detalle importante al momento de añadir un proceso que cambia a diferencia de la versión para C es el momento en el que se aumenta el *id counter* general en #c4, que

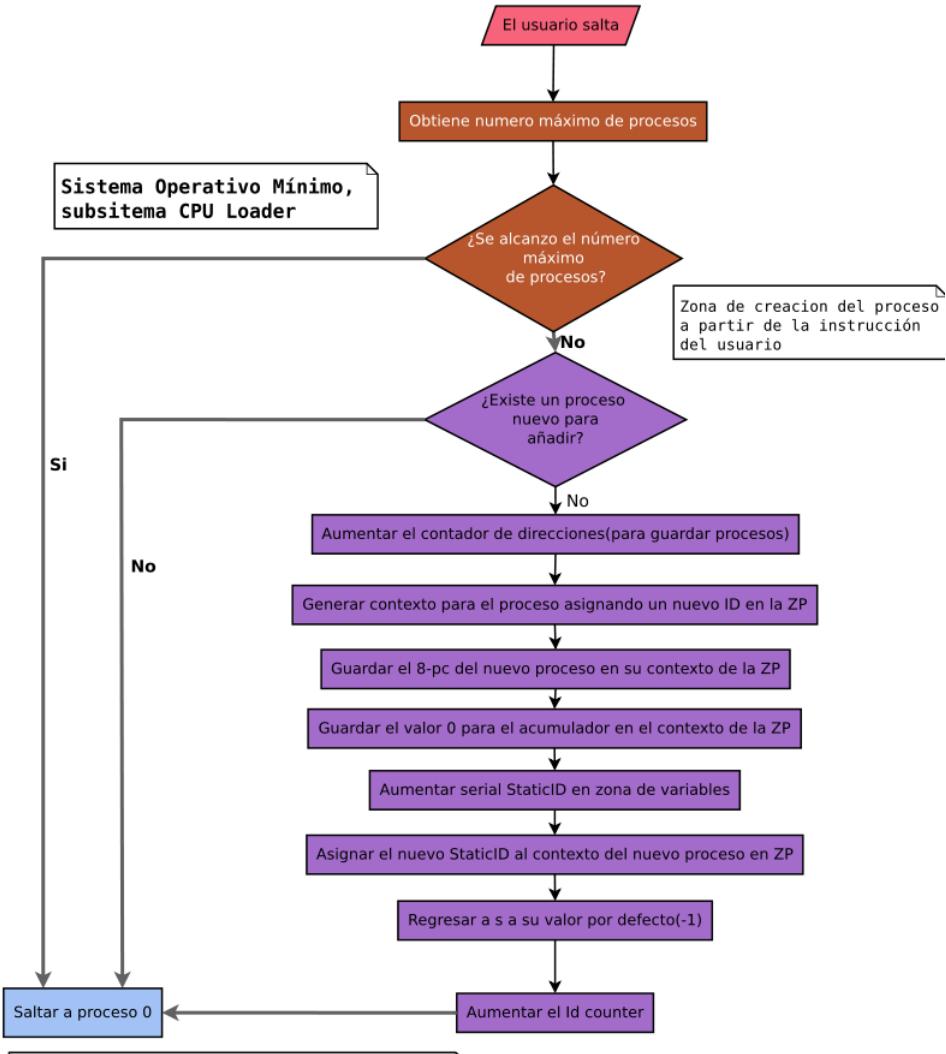


Figura 3.60: Añadir proceso en SOMP

ahora se aumenta hasta el final, justo cuando se han preparado todas las variables y sólo queda saltar de regreso al proceso 0, esto lo podemos ver también en la figura 3.62. Esto es sumamente importante por que en todo momento el otro cpu está activo y en estado de espera a que #c4 cambie su valor a uno distinto de 0, cuando el *waiter* detecta que cambio permite que la ejecución en el cpu ejecutor continúe y haga todos los procesos necesarios para lanzar un nuevo proceso.

Aquí termina toda la participación del sistema operativo para el cpu cargador, el resto son ejecuciones del usuario y del *starter*. Puesto que el resto de operaciones que realiza este cpu son solo la carga de instrucciones que vienen por medio del usuario desde el modo *deck*, el

Preámbulo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
Recibe el pc	e0			Le coloca la maquina -1 si la instrucción fue 9(px), si salto el sw coloca el pc
Manda acc a C	e1	6(c2)	STO c2	Manda a c2 el último acc del proceso antes de saltar
Cambiar bandera a "No" permitir	e2	1(000)	LDA 000	Con bandera==0 no se permiten saltos
	e3	7(000)	SUB 000	No se permiten saltos
	e4	6(003)	STO 003	Saltamos a actualizar
	e5	1(e0)	LDA e0	Carga en el acumulador el último pc del proceso antes de saltar
Mandar pc a C	e6	2(c13)	ADD c13	Le coloca al pc el op-code 8
	e7	6(c1)	STO c1	En c1 se guarda el pc con op-code 8
Obtener Saver Jump	e8	1(999)	LDA 999	Cargar el último valor de 999 del proceso que salio
	e9	6(c14)	STO c14	Guardar en c14 el nuevo valor
Verificar marca	e10	1(e0)	LDA e0	Se obtiene la marca
	e11	3(e13)	BLZ e11	Si la marca es menor a 1 se va al área de borrado
Salto	e12	8(s2)	JMP s2	Si la marca no es menor a 1 se va al área de actualización
Salto	e13	8(s24)	JMP s19	Salta al área de borrado
Preámbulo para añadir procesos	e14	1(c16)	LDA c16	Obtiene el numero máximo de procesos
	e15	7(c4)	SUB c4	Le resta la cantidad de procesos que hay
	e16	3(000)	BLZ 000	Si acc<0 se ha alcanzado el número máximo de procesos
	e17	8(s66)	JMP s66	Si no se ha alcanzado el máximo de procesos añadir otro
	e18			

Figura 3.61: Preámbulo sistema operativo mínimo paralelo

modo masivo, y el *booteo* que se inicializa por medio del *starter* en conexión con el *deck* para cargar al mismo proceso 0 y al sistema operativo en la memoria principal. Notarán que he sido incisivo con que la carga del usuario es por medio del modo *deck* y esto es por que como el mismo diagrama de la arquitectura habrá dos medios de entrada a diferencia de lo que sucedía en C, que era un mismo método de entrada en dos presentaciones diferentes, ahora usaremos esas dos presentaciones para que sean métodos de entrada independientes entre sí y cada uno sirva a un cpu en específico para no causar conflictos por que ambos requieran hacer uso del periférico al mismo tiempo. Por esta misma razón el ALU del *cpu loader* no reconocerá la operación *output* y simplemente no ejecutará nada, el único que puede hacer impresiones es el *cpu executor*.

Otra instrucción que tendrá un comportamiento limitado será la de *halt*, por que ahora si es ejecutada sólo saltará a #000 sin importar que operadores le acompañen, pues en el cpu cargador no tiene sentido el detener un proceso.

## Maquina virtual de E-CARDIAC PC

La pantalla principal de *E-CARDIAC PC* es la que vemos en la imagen 3.63, notando que tiene los dos métodos de entrada que ahora servirán cada uno a un cpu diferente, y por supuesto que ahora tenemos dos cpu, uno en cada lateral. Al tener dos cpu fue necesario

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
	s66	1(s)	LDA s	Se carga en el acumulador la 8-dir de inicio del proceso
	s67	3(s94)	BLZ s94	si acc<0 no hay nuevo proceso
Aumenta el Dir counter M[c5]++	s68	1(c5)	LDA c5	Se obtiene el valor del Dir counter
	s69	2(c11)	ADD c11	Se le suma 5 para tener la siguiente dirección
	s70	6(c5)	STO c5	
Previa de ID	s71	2(c9)	ADD c9	Convierte la nueva dirección en dir counter en 6-dir
	s72	6(s92)	STO s86	En s70 se guarda 6(px)
Previa de pc	s73	2000	ADD 000	Se obtiene la 6-dir del gpc del nuevo proceso
	s74	6(s84)	STO s78	En s72 se guarda 6(px)+1, la 6-dir del gpc del nuevo proceso
Previa de acc	s75	2000	ADD 000	Se obtiene la 6-dir del gacc del nuevo proceso
	s76	6(s87)	STO s81	En s75 se guarda 6(px)+2
	s77	2(c12)	ADD C12	Añade 2 para obtener 6(px)+4(Static ID)
Guardar Nuevo Static ID para el proceso	s78	6(s82)	STO s85	
	s79	1(c15)	LDA c15	Obtener Serial de Static ID
	s80	2000	ADD 000	Añadir una unidad
	s81	6(c15)	STO C15	Guardar en c15 el folio actualizado
	s82	6(px)+4	STO (px)+4	Guarda el nuevo serial Static ID para el proceso
pc nuevo	s83	1(s)	LDA s	Se obtiene el 8-pc del proceso
	s84	6(px)+1	STO px+1	Se guarda el 8-pc en el gpc del proceso
acc del nuevo proceso	s85	1000	LDA 000	
	s86	7000	SUB 000	Se obtiene el 0
	s87	6(px)+2	STO px+2	El gacc del proceso se inicializa en 0
Poner valor default en s	s88	1(c10)	LDA c10	
	s89	6(s)	STO s	En s se coloca el valor -1
	s90	1(c4)	LDA c4	Obtener ID counter
Aumenta el Id counter M[c4]++	s91	2000	ADD 000	Aumentar Id Counter
	s92	6(px)	STO px	Actualizar zona de procesos
	s93	6(c4)	STO c4	Actualizar zona de variables
Return	s94	8(000)	JMP 000	Salta al proceso 0

Figura 3.62: Añadir proceso sistema operativo mínimo paralelo

también reordenar el contenido de *machine status* pues varios de sus elementos eran propios del cpu y ahora que hay dos es necesario diferenciar a que cpu le pertenece cada uno. Además de que en ese mismo apartado se debe añadir otro contador de ciclos y otro visualizador de la operación que se está ejecutando para tener uno por cada cpu.

También fue necesario reordenar algunos elementos de la pantalla inicial para distribuir de mejor manera todos los elementos debido al aumento de estos mismos. Como podemos notar el esquema sigue siendo muy semejante a C, y el inicio es igual, después de dar clic en *start* se iniciará la carga de las instrucciones de la memoria secundaria, por medio del *starter*, hacia la memoria principal. En la imagen 3.64 podemos ver lo que sucede después de iniciar el booteo, básicamente lo mismo que en C, con la diferencia que ahora podemos notar que si bien el *cpu loader* está funcionando en su totalidad, con *Cycle CPUL* y *Operation CPUL* indicando el ciclo en el que va y la instrucción que está ejecutando respectivamente, los ciclos correspondientes al segundo cpu están vacíos, al igual que la mayoría de las variables propias del segundo cpu. Esto se debe a que una vez que inicia la maquina se inician los dos

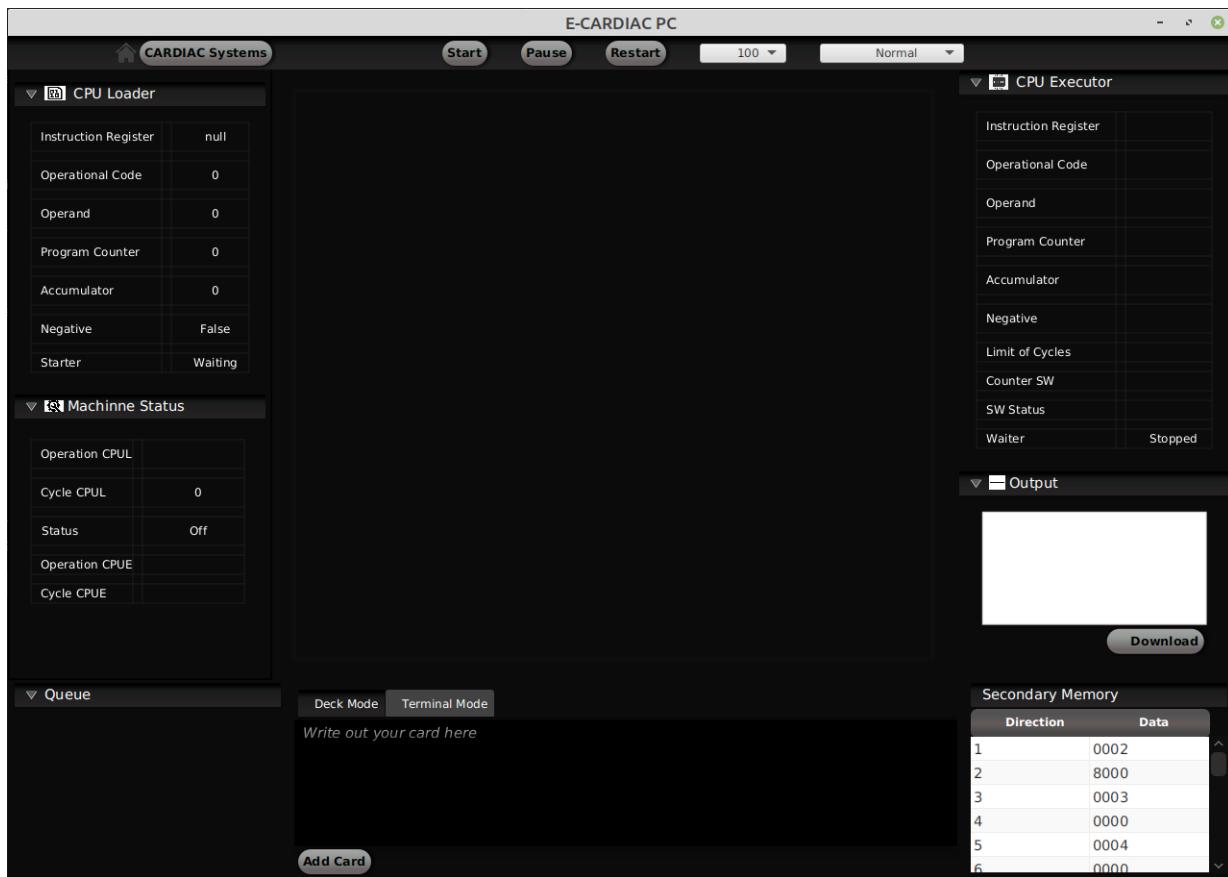


Figura 3.63: E-CARDIAC PC sin iniciar

cpu, el primero inicia la carga a través del loader y continua ejecutando instrucciones, pero el segundo inicia en la dirección #s97, dirección que activa al *waiter* y que impide que el contador de programa avance hasta que haya un proceso cargado por el usuario.

Cuando se termina la carga del sistema operativo mínimo lo que vemos es lo que se aprecia en la imagen 3.65, dónde vemos que en apenas 595 ciclos se cargo por completo el sistema y ahora está esperando en la dirección #000 a que el usuario empiece a cargar programas en memoria para añadirlos como procesos.

### Sistema operativo para CPU Executor

Antes de cargar el primer programa en la nueva maquina será necesario conocer como funciona el sistema operativo para el segundo cpu. Si vemos el diagrama general del sistema operativo(figura 3.59) notaremos que toda la parte izquierda es ejecutada por el segundo cpu, si vemos de más cerca al preámbulo(fig. 3.66) podemos notar que las dos formas de entrada

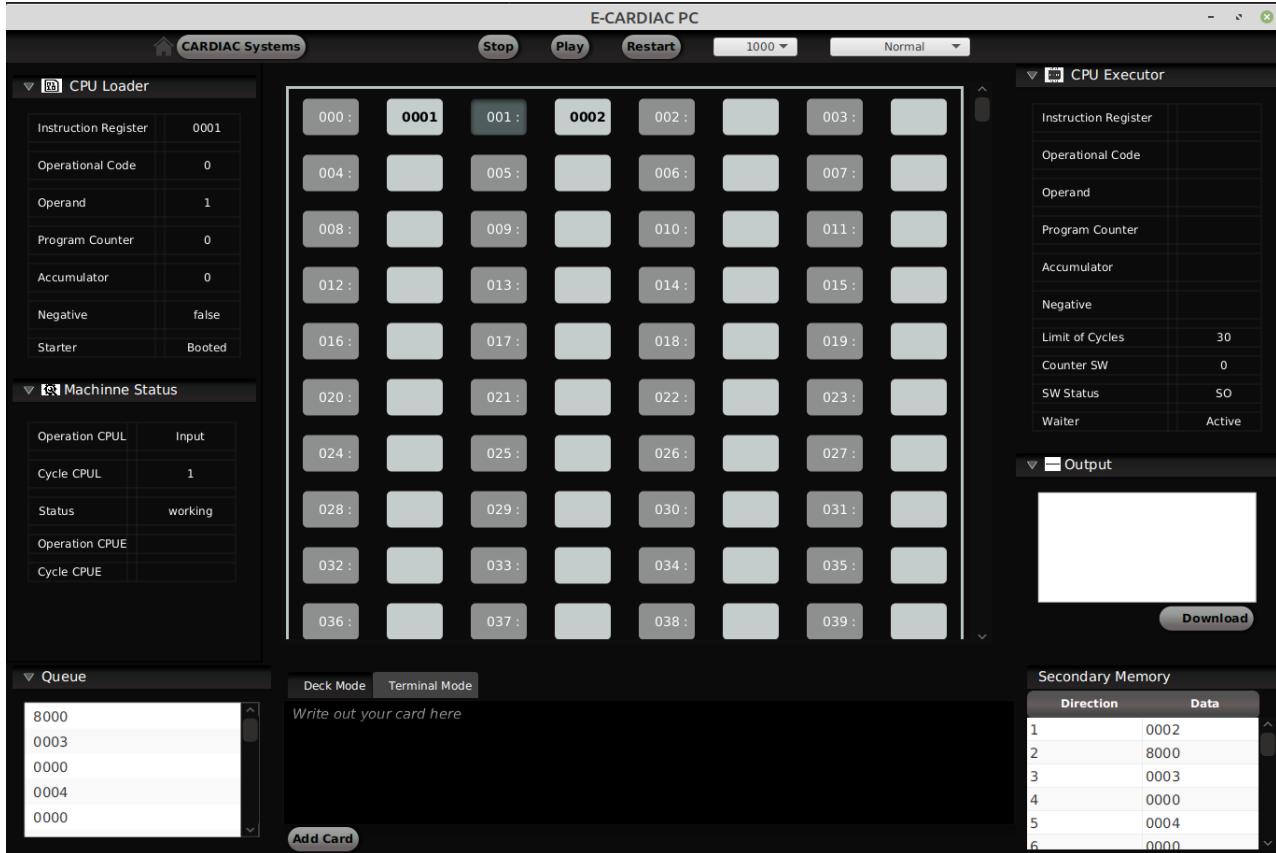


Figura 3.64: E-CARDIAC PC Booteo

que tiene son si el contador del programa salta en automático, por medio del *switcher* o si un proceso es terminado con la operación *halt*.

Después de eso realiza las operaciones que ya realizaba en para C el preámbulo, guarda las variables del sistema, desactiva el salto del contador de programa, y obtiene la marca para saber si un proceso ha sido terminado por la instrucción de finalizar. De hecho si vemos en la imagen 3.61 notaremos que lo único que cambia son las direcciones a dónde salta, por que como fueron cambiadas algunas instrucciones cambiaron las posiciones para ir a la sección de borrado, que es la sección que más redujo su tamaño respecto al sistema que estaba para la versión únicamente concurrente.

Ahora, si recordamos el sistema operativo no avanza nada hasta que el *waiter* se lo permita, y esto sucederá hasta que exista al menos un proceso cargado. Esto lo podemos ver en la parte inferior izquierda del diagrama(fig. 3.67) donde está resaltado en rosa un recuadro indicando que continuará el flujo hasta que el *id counter* sea distinto de cero, el

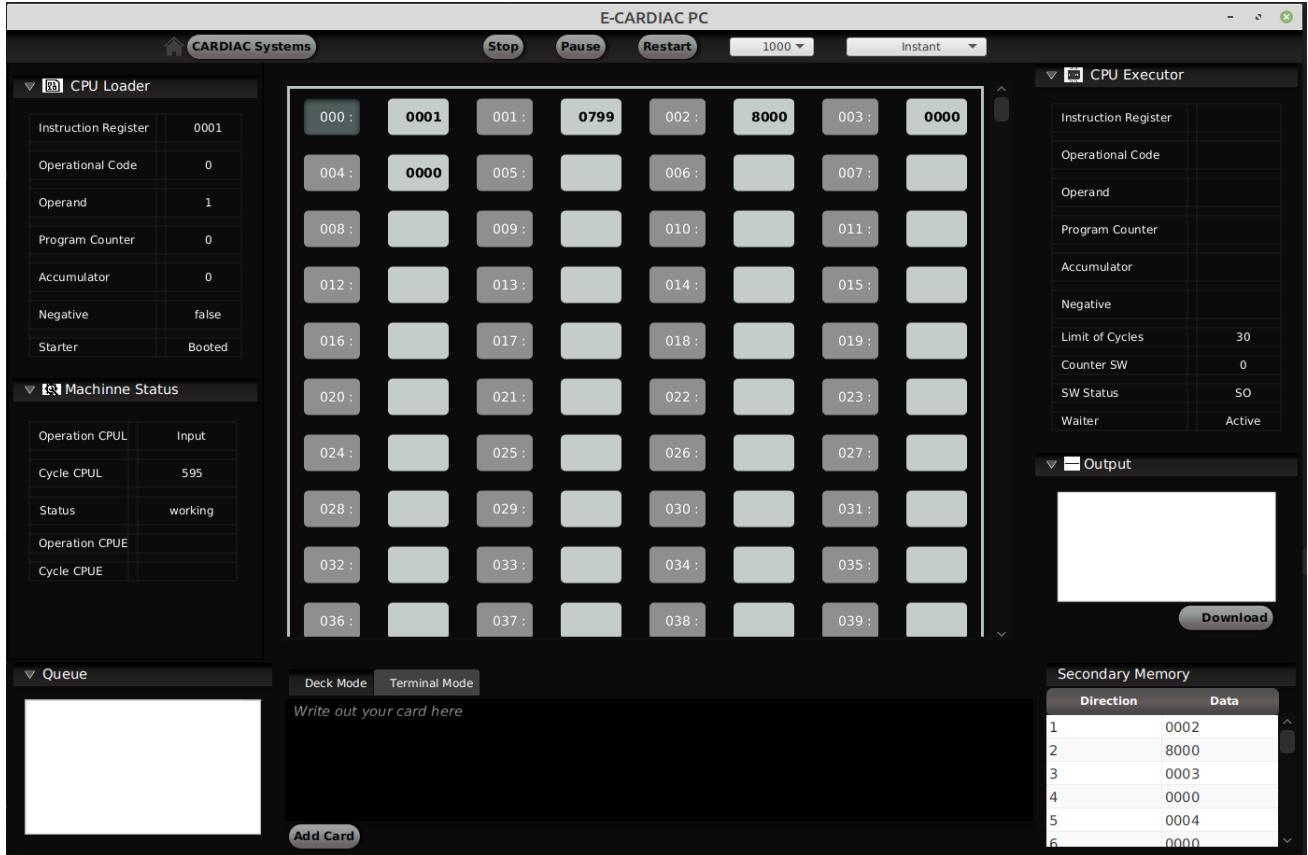


Figura 3.65: E-CARDIAC PC Iniciado

rosa es para indicar que es una especie de entrada al sistema, por que es un punto en el que el sistema puede continuar su ejecución. Como podemos ver el flujo que continua en azul claro, que en C era para el control del proceso 0, aquí tiene casi las mismas funciones de reinicio de operaciones, pero sin saltar al proceso 0, por lo que le llamaremos segmento de “control de procesos”. En este segmento lo que hace es que si ya no hay procesos,  $id_{counter} == 0$ , reinicia los organizadores y el identificador que se usa para las impresiones antes de activar el *watier*, para que cuando esté deje continuar al contador de programa y ya haya procesos los organizadores estén localizados en la posición correcta, en la imagen 3.68 se puede ver en código.

El segmento de lanzamiento de procesos, en el diagrama en la parte inferior en naranja(fig. 3.69) no cambio en nada, salvo direcciones, si hay procesos se llega a este segmento que cambia los valores de los organizadores, obtiene de la zona de procesos la información necesaria del proceso, la coloca en dónde el proceso la pueda tomar, activa el salto del *switcher*, y salta al

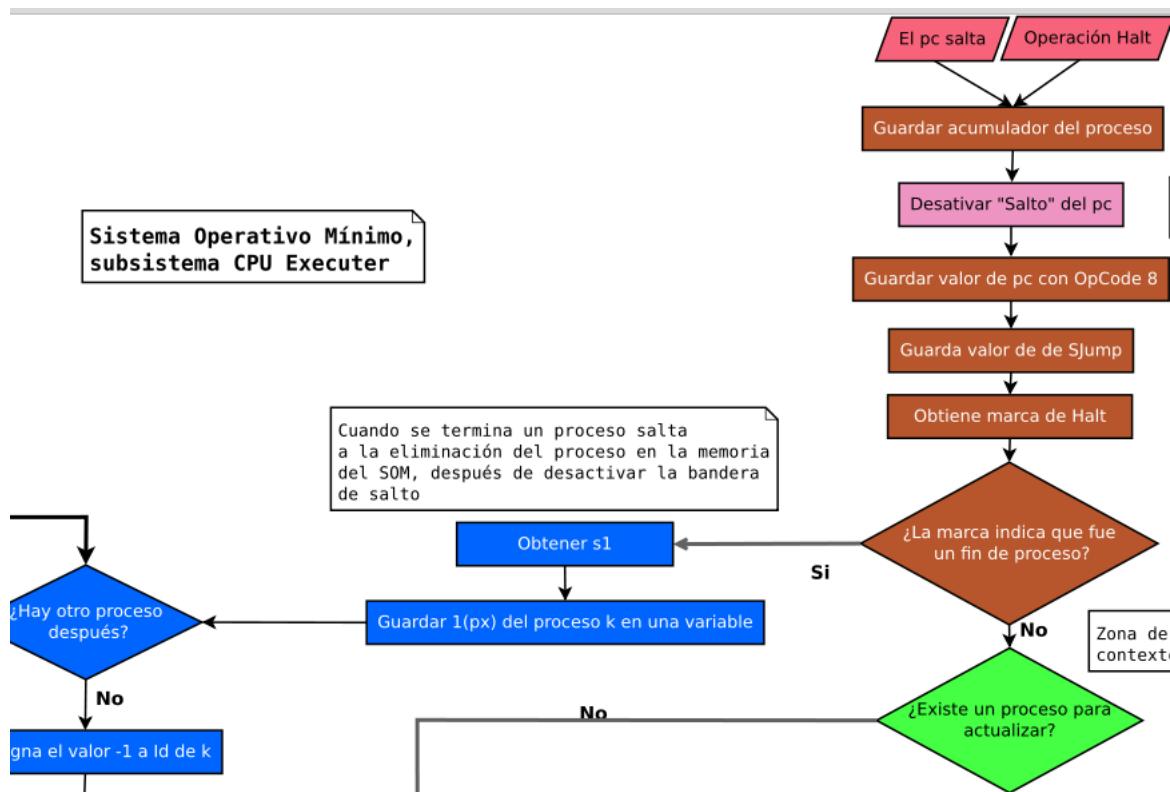


Figura 3.66: Acercamiento al preámbulo del sistema operativo mínimo

proceso. Para tener más cerca las imágenes dónde están los códigos para el lanzamiento de procesos las podemos ver en las figuras 3.70 y 3.71, que sirve para ver que es básicamente el mismo código, con el mismo funcionamiento.

En cuanto a la actualización de procesos, que podemos ver en la parte central del diagrama en color verde(fig. 3.72), tampoco sufrió modificaciones, salvo el cambio de direcciones que podemos ver en la imagen 3.73. Como la zona de procesos se mantiene sin cambios es natural que el actualizarlos no requiera un cambio, pues son las mismas variables.

Lo que si tuvo un cambio importante es la zona de borrado(fig. 3.74), que no cambia la forma de borrar, esas subrutinas se quedan sin cambios, elimina los valores del proceso a eliminar, y si hay más procesos con un *id* mayor los va desplazando para reducir el número de procesos, esto lo vemos en el diagrama en el ciclo que está a la izquierda y en el código en la figura 3.76 y en la parte final de la figura 3.75. Lo que cambia como podemos notar en esta ultima imagen es que ya no se tiene un tratamiento especial para el proceso 0, como el cpu principal no tiene definida la instrucción *halt*, el proceso 0 no puede ser borrado, no hay

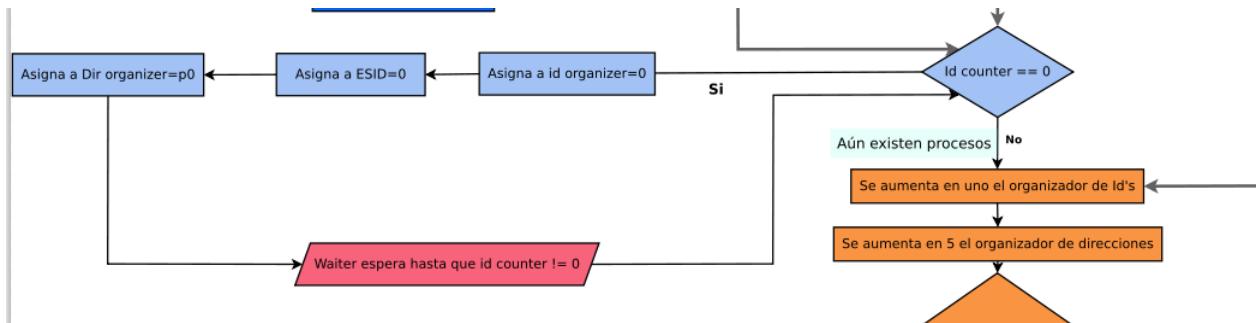


Figura 3.67: Acercamiento a segmento de *waiter* y control de procesos

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>Waiter</i>	s97	0(998)	INP 998	Indica el inicio de la espera
¿Se acabaron los programas?	s98	1(c4)	LDA c4	Se obtiene el id counter, si es 1 hay que saltar a lanzar el proceso
	s99	7(000)	SUB 000	Se le resta un 1 , acc=-1
	s100	3(s141)	BLZ s141	Si acc<0 salta al proceso 0
Reiniciar id organizer a 0	s141	1(000)	LDA 000	
	s142	7000	SUB 000	Carga el id=0 del proceso 0
	s143	6(c7)	STO c7	Asignar a id organizer=0
Asigna a 004 el 0	s144	6(004)	STO 004	Guardar en 0004 el static Id 0 para el proceso 0
Asigna a dir organizer=p0	s145	1(c3)	LDA c3	Obtiene p4
	s146	7(c11)	SUB c11	Obtiene p1 al restarle 5
	s147	6(c6)	STO c6	Asignar a dir organizer=p0
	s148	8(s97)	JMP s97	Saltar al <i>waiter</i>

Figura 3.68: Segmento de control de procesos

forma de que sea enviado a este segmento que opera el segundo cpu, y como ahora sólo recibe procesos de usuario para ser borrados sólo requiere obtener su información y continuar con la ejecución. Razón por la que se dejaron vacías cinco celdas que ya no se ocupan, y que se decidió no recorrer el programa para que no se tuvieran que editar el resto de segmentos por esta razón, así que el borrado es incluso más eficiente para la versión paralela del sistema operativo.

Por lo que el flujo en general sería el siguiente, el usuario añade un proceso con ayuda del primer cpu, una vez esté este cargado el *waiter* permitirá avanzar al contador de programa del segundo cpu, del ejecutor, para que añada el programa a la lista de procesos a ejecutar y tal cuál pasaba en C, si ya hay un proceso agregado a la zona de procesos el lanzador de procesos puede empezar a ejecutarlos. Y una vez que empieza este ciclo continua con las actualizaciones a la zona de procesos, si el proceso del usuario tiene que ceder los recursos y pausar su ejecución, y al borrado del proceso cuando esté haya terminado su ejecución. Volviendo al *waiter* una vez que se hayan terminado los procesos y el segundo cpu regrese a

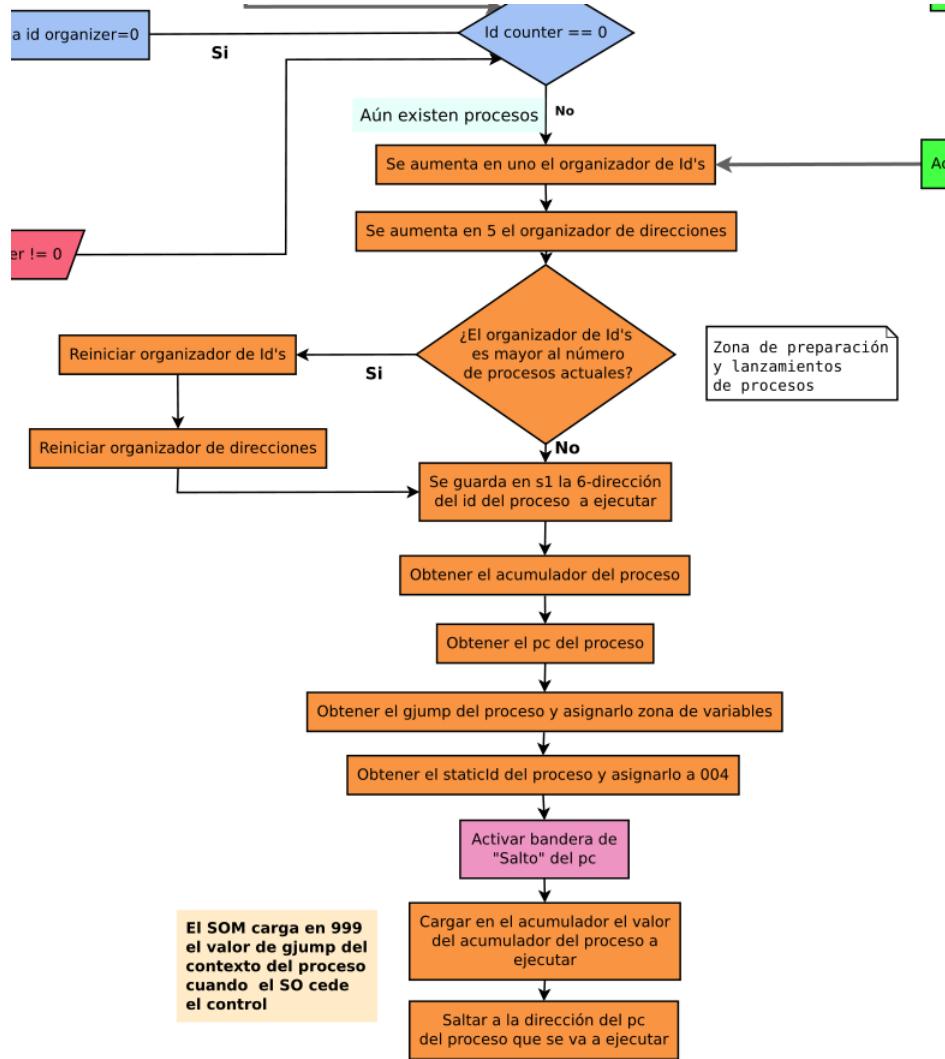


Figura 3.69: Acercamiento a segmento de lanzamiento de procesos

un estado de espera.

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>Aumentar Id organizer M[c7]++</i>	<b>s101</b>	<b>1(c7)</b>	LDA c7	
	<b>s102</b>	<b>2000</b>	ADD 000	
	<b>s103</b>	<b>6(c7)</b>	STO c7	Se aumenta el id organizer
<i>Aumentar Dir organizer M[c6]+=5</i>	<b>s104</b>	<b>1(c6)</b>	LDA c6	
	<b>s105</b>	<b>2(c11)</b>	ADD c11	
	<b>s106</b>	<b>6(c6)</b>	STO c6	Se aumenta el dir organizer
<i>Verifica si hay que reiniciar</i>	<b>s107</b>	<b>1(c4)</b>	LDA c4	Se obtiene id counter
	<b>s108</b>	<b>7(c7)</b>	SUB c7	Al id counter se le resta el id organizer, M[c4]-M[c7]
	<b>s109</b>	<b>3(s134)</b>	BLZ s133	si acc<0 tenemos que reiniciar, por que llegamos al ultimo
<i>Sino-s109</i>	<b>s110</b>	<b>1(c6)</b>	LDA c6	Se obtiene la dir del id del proceso a ejecutar
	<b>s111</b>	<b>2(c9)</b>	ADD c9	Se crea la 6-dir del id del proceso a ejecutar
	<b>s112</b>	<b>6(s1)</b>	STO s1	Se guarda en s1, será el proceso "actual"
<i>Actualizar s1 para salir al proceso con su dirección</i>	<b>s113</b>	<b>1(c6)</b>	LDA c6	
	<b>s114</b>	<b>2(c8)</b>	ADD c8	Se convierte en la 1-dir del id del proceso a ejecutar
	<b>s115</b>	<b>2000</b>	ADD 000	Se obtiene la 1-dir del gpc del proceso a ejecutar
<i>Continuación</i>	<b>s116</b>	<b>6(s119)</b>	STO s118	En s101 se guarda la 1-dir del gpc del proceso a ejecutar
	<b>s117</b>	<b>2000</b>	ADD 000	Se obtiene la 1-dir del gacc del proceso a ejecutar

Figura 3.70: Sistema operativo mínimo paralelo, segmento de lanzamiento

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>Preparación gpc y gcc</i>	<b>s118</b>	<b>6(s132)</b>	STO s131	En 112 se guarda la 1-dir del gacc del proceso a ejecutar
	<b>s119</b>	<b>1(px)+1</b>	LDA px+1	Obtiene el gpc del proceso a ejecutar
	<b>s120</b>	<b>6(s133)</b>	STO s132	Guarda el gpc del proceso a ejecutar en s106
<i>Preparación gjump</i>	<b>s121</b>	<b>1(s132)</b>	LDA s131	Obtiene la 1 dir del gacc
	<b>s122</b>	<b>2000</b>	ADD 000	Obtiene la 1 dir del gjump
	<b>s123</b>	<b>6(s128)</b>	STO s127	Guarda la 1 dir del gjump en s126
<i>Salvar Static ID en 004</i>	<b>s124</b>	<b>2000</b>	ADD 000	Añadir un 1 para obtener el 1(px)+4, SID
	<b>s125</b>	<b>6(s126)</b>	STO s125	Guardar en la siguiente celda
	<b>s126</b>	<b>1(px)+4</b>	LDA px+4	Obtener Static ID del proceso
<i>Salvar gjump</i>	<b>s127</b>	<b>6004</b>	STO 004	Guardar en 0004 para usarlo como identificador el proceso que se está ejecutando
	<b>s128</b>	<b>1(px)+3</b>	LDA px+3	Carga el valor de la gjump
	<b>s129</b>	<b>6(c14)</b>	STO c14	Guarda el gjump en c14 para que la arquitectura lo intercambie
<i>bandera</i>	<b>s130</b>	<b>1(000)</b>	LDA 000	Obtiene el número 1
	<b>s131</b>	<b>6003</b>	STO 003	Se permiten saltos con bandera==1
	<b>s132</b>	<b>1(px)+2</b>	LDA px+2	Se obtiene acc= acc del proceso a ejecutar
<i>LastDirectionS*</i>	<b>s133</b>	<b>8(xx)</b>	JMP xx	La dir xx representa el valor donde el proceso se ejecutará
<i>Reiniciar dir organizer</i>	<b>s134</b>	<b>1(c3)</b>	LDA c3	Se obtiene la primera dir de la zona de procesos, p3
	<b>s135</b>	<b>6(c6)</b>	STO c6	Se reinicia el dir organizer
	<b>s136</b>	<b>2(c8)</b>	ADD c8	Se crea la 1-dir de inicio en el acumulador
<i>Reiniciar id organizer</i>	<b>s137</b>	<b>6(s138)</b>	STO s137	En s138 se guarda px
	<b>s138</b>	<b>1(px)</b>	LDA px	Se obtiene el id de inicio
	<b>s139</b>	<b>6(c7)</b>	STO c7	Se reinicia el id organizer
<i>Regresar</i>	<b>s140</b>	<b>8(s110)</b>	JMP s109	Regresar para lanzar el proceso

Figura 3.71: Sistema operativo mínimo paralelo, segmento de lanzamiento parte 2

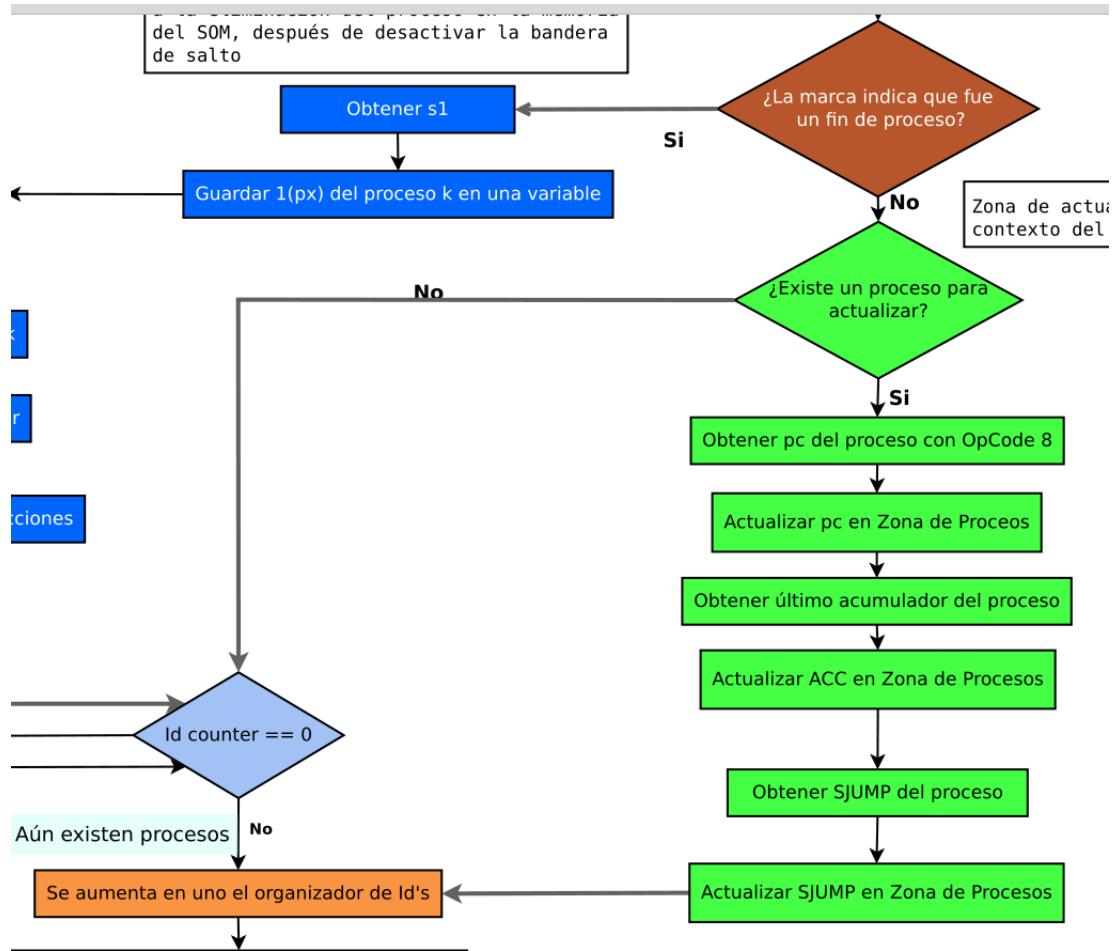


Figura 3.72: Acercamiento al segmento de actualización de procesos

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>Nuevo</i>	<i>s0</i>	-1		-1 Es su valor por defecto, el usuario siempre agregará una 8-dir
<i>Actual</i>	<i>s1</i>	6000		Contiene la 6-dir del id del proceso actual, controlado por el lanzador, 6(px)
<i>Validación</i>	<i>s2</i> <i>1(s1)</i>	<i>LDA s1</i>		El acumulador toma un valor de la forma 6(px)
	<i>s3</i> <i>3(s98)</i>	<i>BLZ s98</i>		Si acc<0 no hay proceso para actualizar
	<i>s4</i> 2000	<i>ADD 000</i>		Se obtiene la 6-dir del gpc del proceso
<i>Actualizar gpc</i>	<i>s5</i> <i>6(s7)</i>	<i>STO s7</i>		Se guarda la instrucción 6(px)+1
	<i>s6</i> <i>1(c1)</i>	<i>LDA C1</i>		Se obtiene el último pc del proceso con forma 8(pc)
	<i>s7</i> <i>6(p5)</i>	<i>STO p5</i>		En p5 se actualiza gpc
	<i>s8</i> <i>1(s7)</i>	<i>LDA s7</i>		Se obtiene la instrucción 6(px)+1
	<i>s9</i> 2000	<i>ADD 000</i>		Para acceder a la 6-dir del gacc del proceso
<i>Actualizar gacc</i>	<i>s10</i> <i>6(s12)</i>	<i>STO s12</i>		En s12 se guarda 6(px)+2
	<i>s11</i> <i>1(c2)</i>	<i>LDA c2</i>		Se obtiene el último acc del proceso
	<i>s12</i> <i>6(p6)</i>	<i>STO p6</i>		Se actualiza el valor de gacc
<i>Actualizar gjump</i>	<i>s13</i> <i>1(s12)</i>	<i>LDA s12</i>		Obtiene la 6(p6) del proceso que se está actualizando
	<i>s14</i> 2000	<i>ADD 000</i>		Obtiene la 6(p7) del proceso que se está ejecutando
	<i>s15</i> <i>6(s17)</i>	<i>STO s17</i>		Guarda en e18 el código para guardar en la zona de procesos c14
	<i>s16</i> <i>1(c14)</i>	<i>LDA c14</i>		Obtiene c14
	<i>s17</i> <i>6(p7)</i>	<i>STO p7</i>		Guarda en la zona de procesos correspondiente al proceso el valor saber jump
<i>Saltar</i>	<i>s18</i> <i>8(s101)</i>	<i>JMP S100</i>		Saltamos a cambiar de proceso

Figura 3.73: Sistema operativo mínimo actualización de procesos

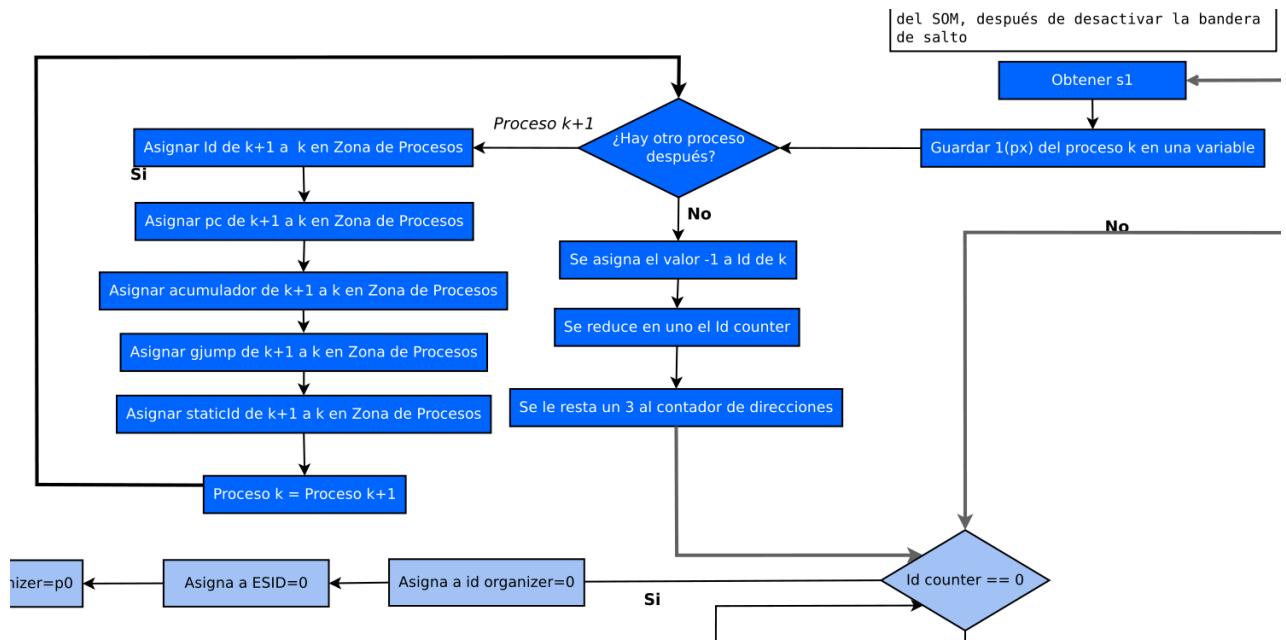


Figura 3.74: Acercamiento al segmento de borrado

Sistema operativo				
Nombre clave	Dirección	Instrucción	Código	Descripción
	s19			
	s20			
	s21			
	s22			
	s23			
<i>Guardar 1(px)</i>	s24	1(s1)	SUB 000	Llamemos k al proceso asociado a la dir px
	s25	4022	SHT 11	Se convierte 6(px) en 0(px)
	s26	2(c8)	ADD c8	Se convierte 0(px) en 1(px)
	s27	2(c11)	ADD c11	Se obtiene 1(px)+5, la 1-dir del id del proceso siguiente (k+1)
	s28	6(s29)	STO s29	En s29 se guarda 1(px)+5, al que llamamos 1(py)
<i>¿Hay otro?</i>	s29	1(py)	LDA py	Se obtiene el id del proceso (k+1)
	s30	3(s52)	BLZ s52	Si acc<0 no hay otro proceso adelante
	s31	1(s29)	LDA s29	Se obtiene 1(py) en el acumulador
<i>Guardar 1(py) en c17</i>	s32	6(c17)	STO c17	Se guarda en c17 para usarla más tarde
<i>Inicializar contador de secciones</i>	s33	1(c18)	LDA 004	se Carga 4 para funcionar como contador
	s34	6(c0)	STO C0	Se guarda el contador de secciones del proceso
	s35	7000	SUB C11	Se verifica si ya terminó con las secciones
	s36	3(s50)	BLZ s50	Si ya se pasaron las 4 secciones salta a ver si hay mas procesos

Figura 3.75: Sistema operativo mínimo borrar proceso

Sistema operativo				
Nombre clave	Dirección	Instrucción	Código	Descripción
<i>Recorrer las secciones gpc,gacc,gjump,staticid de py a px</i>	s37	1(c17)	LDA c17	Se obtiene 1(py) en el acumulador
	s38	2(c0)	ADD Cc0	Obtiene la dirección del gpc/gacc/gjump/staticid del proceso py
	s39	6(s44)	STO s44	Se guarda en s44 para tener obtener el valor de la sección del proceso
	s40	7c11	SUB C11	Obtiene la dirección correspondiente del gpc(u otra sección) del proceso px
	s41	4022	SHT 11	Se convierte 1(px)+u en 0(px)+u
	s42	2(c9)	ADD c9	Se convierte 0(px)+u en 6(px)+u
	s43	6(s45)	STO s45	Se guarda en s45 para que sea cargada la sección en px
<i>Aumentar contador de secciones</i>	s44	1(py)+u	LDA py+u	
	s45	6(px)+u	STO px+u	
	s46	1(c0)	LDA c0	Obtener contador de contextos
	s47	7000	ADD 000	Se le resta uno al contador de contextos
	s48	6(c0)	STO c0	Lo guarda en c0 de nuevo
<i>Regresar a recorrer</i>	s49	8(s35)	JMP s35	
<i>¿Hay proceso después de (k+1)?</i>	s50	1(s29)	LDA S29	Se obtiene 1(py) en el acumulador
	s51	8(s27)	JMP s27	Va a verificar si hay otro proceso adelante
	s52	1(s29)	LDA s29	Se carga la 1-dir del id del proceso k+1
<i>"Si" de s30</i>	s53	7(c11)	SUB c11	Se obtiene la 1-dir del id del proceso k, acc=-5
	s54	4022	SHT 11	Convierte 1(px) en 0(px)
	s55	2(c9)	ADD c9	Convierte 0(px) en 6(px)
<i>Borrado</i>	s56	6(s58)	STO s58	Guardar en s58 6(px)
	s57	1(c10)	LDA c10	Se cambia el valor del acumulador, acc=-1
	s58	6(px)	STO px	El proceso con id en la dir px ya no es accesible
<i>Borrado</i>	s59	1(c4)	LDA c4	Carga el id counter
	s60	7(000)	SUB 000	Se le resta 1 al id counter
	s61	6(c4)	STO c4	Guardar en c4 el nuevo valor
<i>M[c4]--</i>	s62	1(c5)	LDA c5	Cargar el dir counter en el acumulador
	s63	7(c11)	SUB c11	Se le resta 4 al dir counter para estar a la par con el id counter
	s64	6(c5)	STO c5	Se guarda en c5
	s65	8(s98)	JMP S98	Salta a ver si el id counter es el ultimo

Figura 3.76: Sistema operativo mínimo borrar proceso : parte 2

### 3.3.4. Ejecutando procesos en E-CARDIAC PC

Añadiremos dos procesos para ver el funcionamiento de la maquina, primero será un proceso para imprimir en reversa los primeros siete números de la serie de Fibonacci, los números serán proveídos por el usuario. Y el segundo será el que ya conocemos, el “pintor” para imprimir los primeros 10 dígitos. De esta forma podremos ver como mientras se ejecuta un programa estamos añadiendo otro.

Para empezar añadimos un programa tal cuál se hacía en C, en la imagen 3.79 vemos que están en la cola de carga unas instrucciones y que el cpu principal está realizando operaciones, mientras que el segundo está en espera. En la figura 3.77 vemos el programa ya cargado de la dirección #204 hasta la #239, y un subrutina que utiliza el programa en la imagen 3.78 vemos una subrutina que utiliza el programa, además vemos que el segundo cpu ya está en funcionamiento y de hecho está en la zona de lanzamiento de procesos, puesto que se encuentra en la dirección #907.

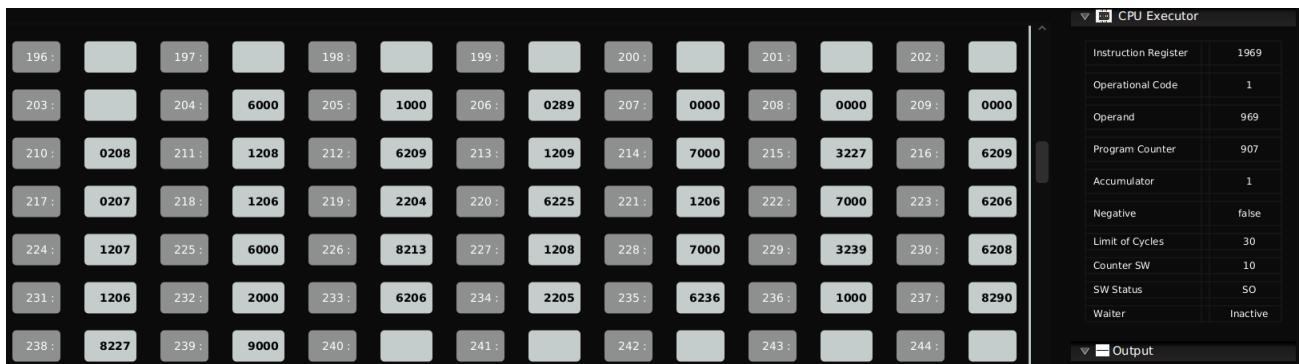


Figura 3.77: Proceso de Fibonacci cargado

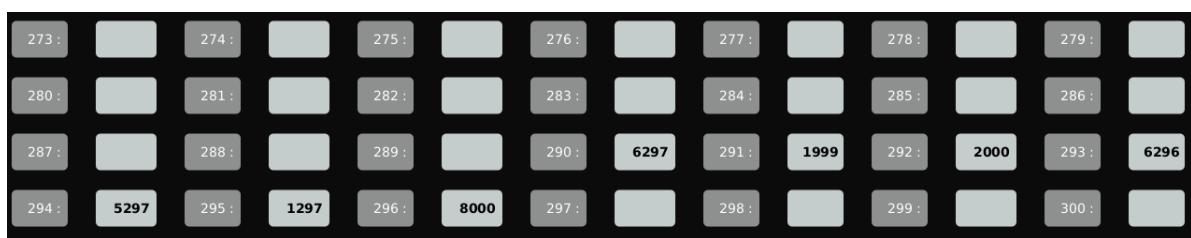


Figura 3.78: Subrutina de proceso de Fibonacci

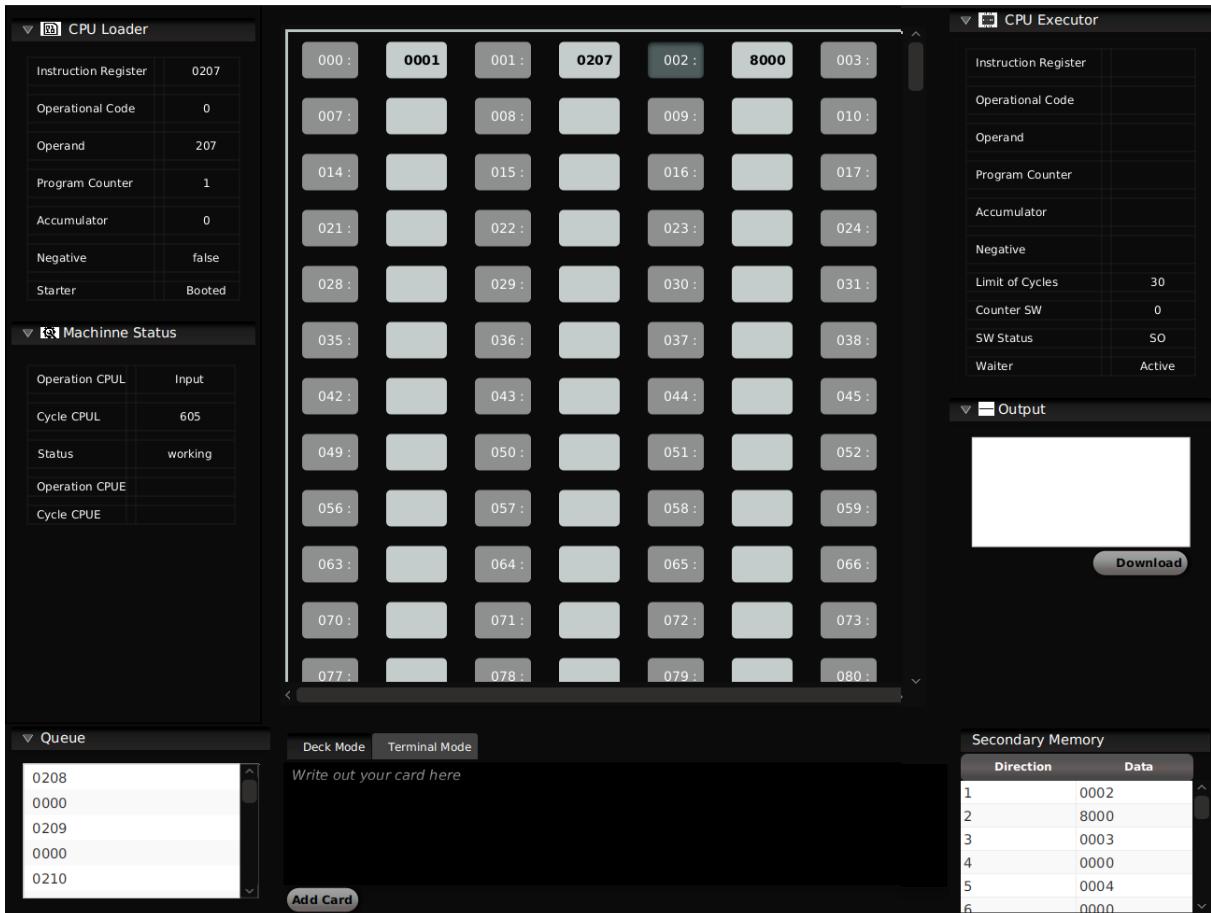


Figura 3.79: Añadiendo un nuevo proceso en E-CARDIAC PC

Lo que sigue es añadir el siguiente proceso, en la imagen 3.80 vemos en el *deck* la tarjeta para añadirla a la cola y vemos también que el segundo cpu ya ha lanzado el primer proceso que agregamos y está en la dirección #210 esperando datos del usuario. Y como notaran el color que resalta la dirección en la que se encuentra el contador de programa del *cpu executor* no es verde, sino naranja/rojiza, para diferenciar fácilmente que *cpu* está usando determinada zona de la memoria memoria.

En la siguiente imagen 3.81, vemos que ya está en la cola de carga el segundo programa y que desde el modo terminal estamos añadiendo la cantidad de números que vamos a querer que el primer proceso, el proceso de Fibonacci, imprima. De esta forma ya estamos avanzando en la ejecución del proceso de Fibonacci y al mismo tiempo ya se está añadiendo un proceso nuevo, ahorrando tiempo al combinar la programación concurrente y el uso de dos procesadores.

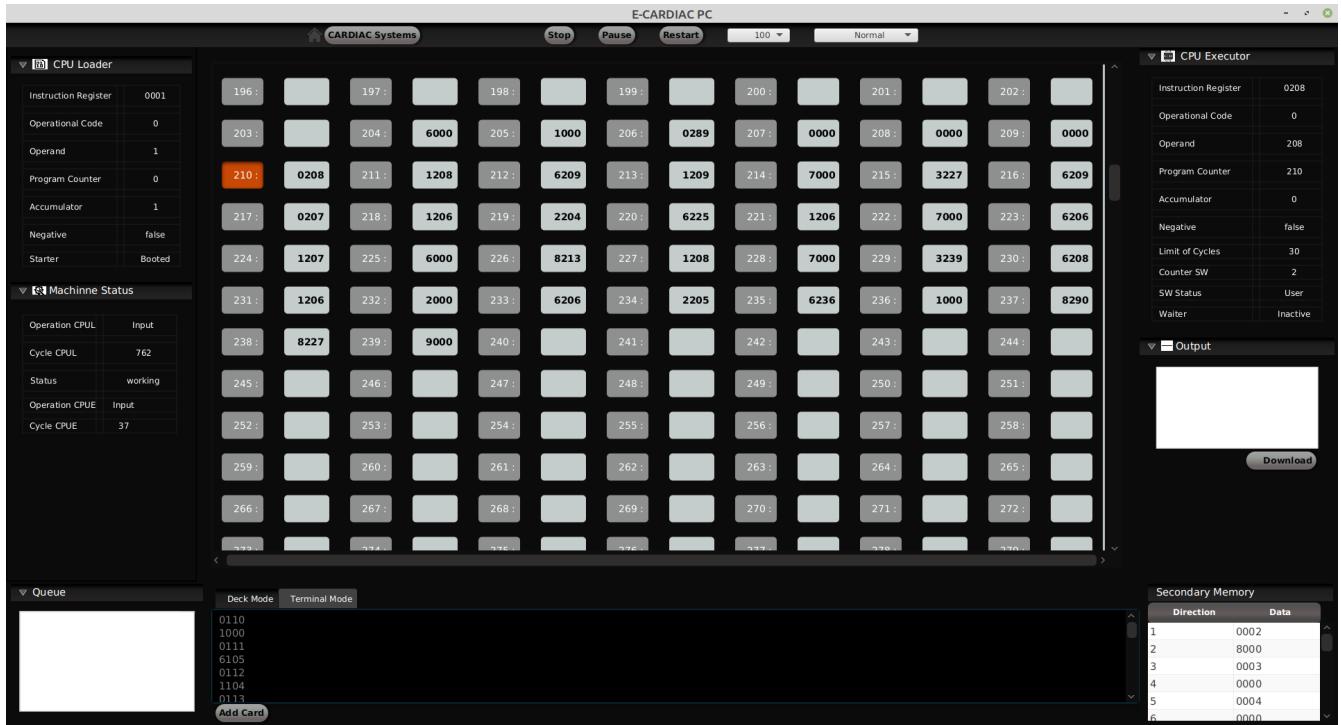


Figura 3.80: Programa pintor en el *deck*

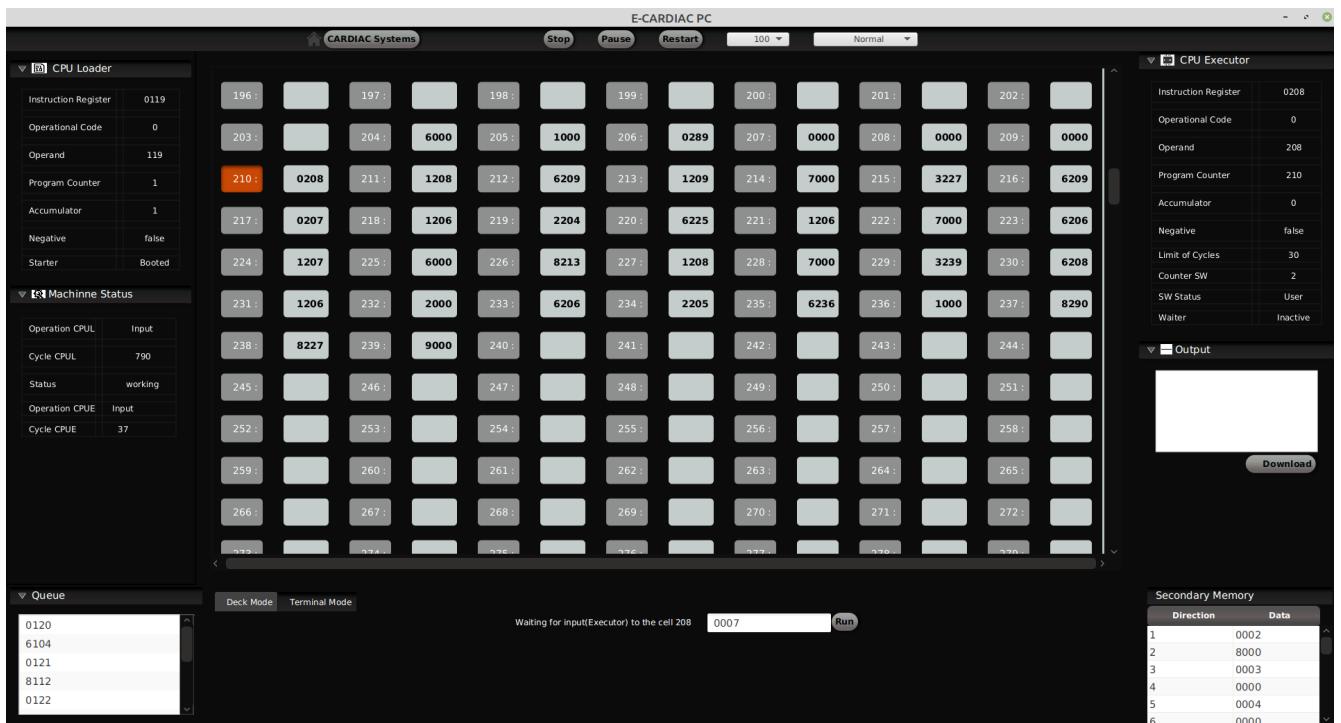


Figura 3.81: Agregando cantidad de números para serie de Fibonacci

Para la imagen 3.82 vemos que ya se está imprimiendo la salida del segundo proceso, del

pintor, y que el cpu *executor* está en el sistema operativo actualizando uno de los procesos. Y ya en la imagen 3.83 vemos que la ejecución termino, terminando al final el proceso 1, y con el *waiter* activado por que el segundo cpu al no haber más procesos se encuentra en estado de espera, mientras que el principal está listo para continuar agregando procesos.

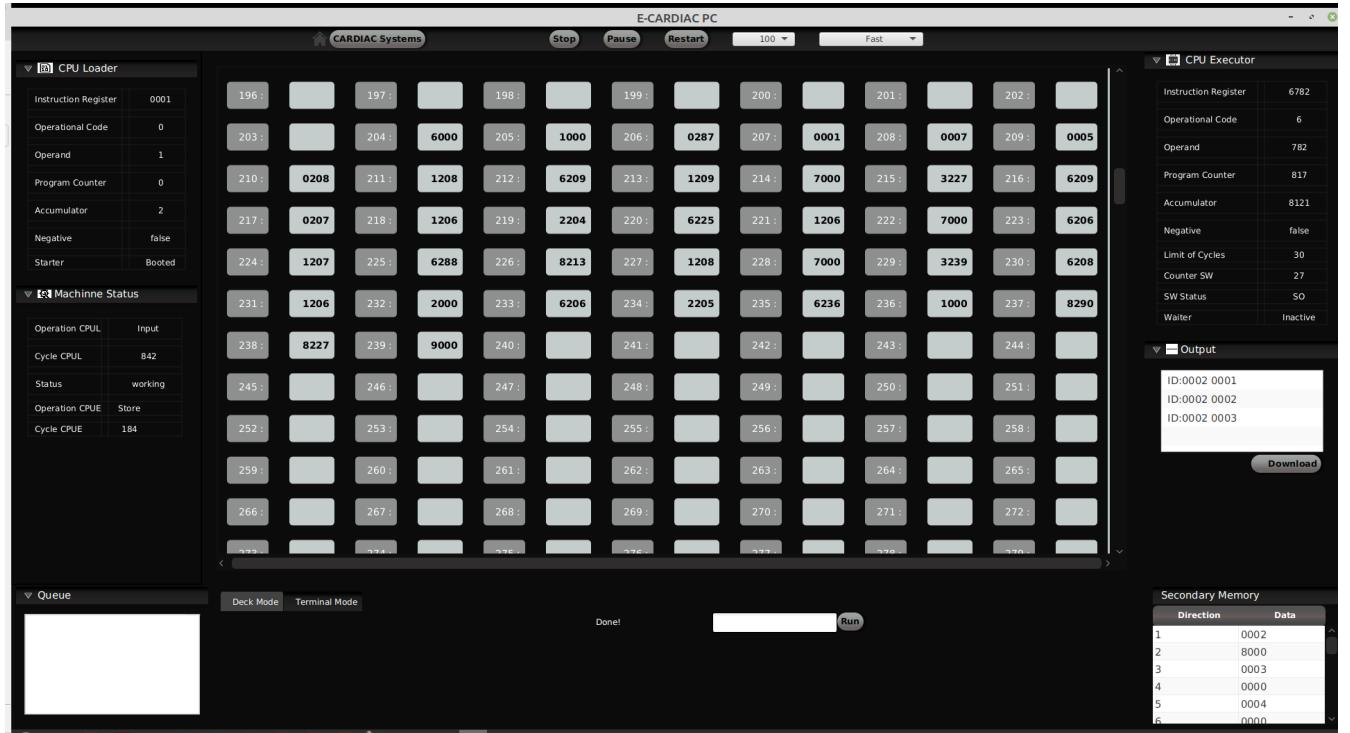


Figura 3.82: Primeras salidas de “pintor”

Para poder ver la salida de mejor forma podemos ir a la carpeta de descargas dónde se encuentra la salida en formato de texto como se puede ver en la imagen 3.84, dónde podemos ver que primero imprimió todo el proceso 2 antes de empezar con el proceso 1, pero que completo ambos sin errores y con una eficiencia notable en comparación con su antecesor *E-CARDIAC C*.

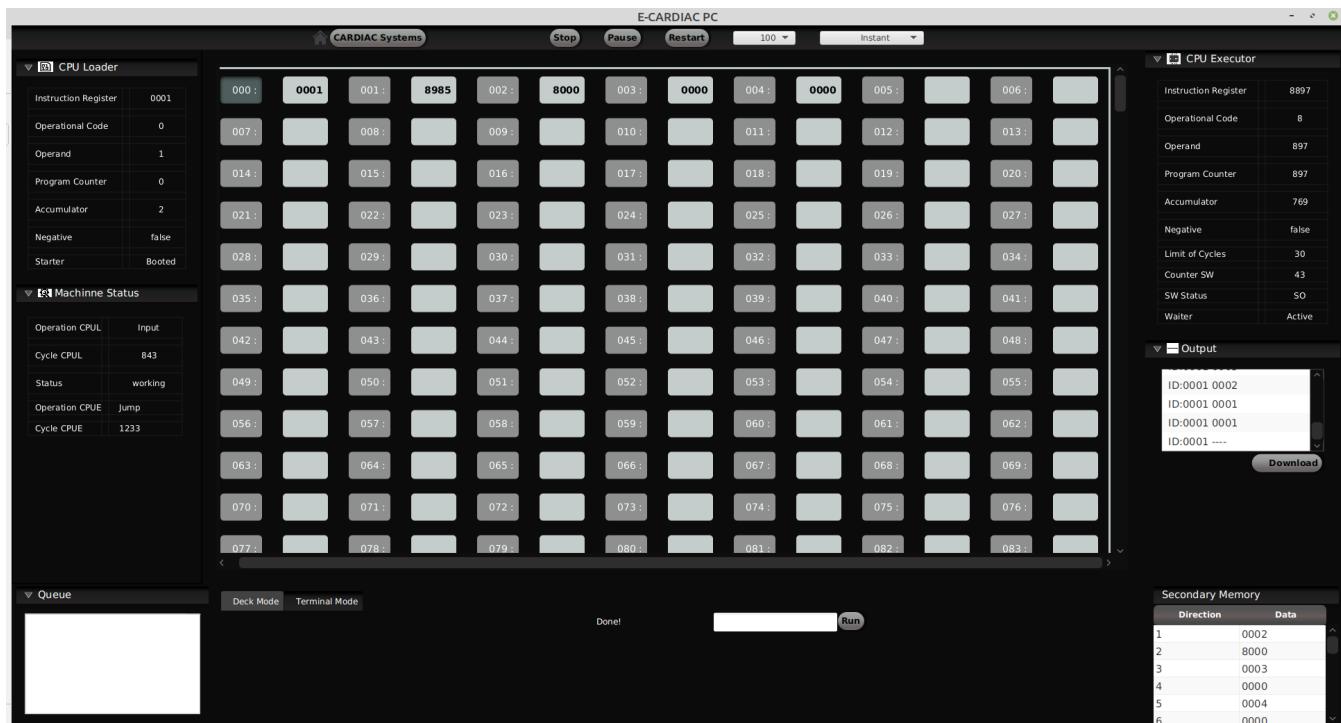


Figura 3.83: Ejecución finalizada

```
CARDIAC_Output_2024...-14 13:36:38.949.txt x
1 ID:0002 0001
2 ID:0002 0002
3 ID:0002 0003
4 ID:0002 0004
5 ID:0002 0005
6 ID:0002 0006
7 ID:0002 0007
8 ID:0002 0008
9 ID:0002 0009
10 ID:0002 0010
11 ID:0002 ----
12 ID:0001 0013
13 ID:0001 0008
14 ID:0001 0005
15 ID:0001 0003
16 ID:0001 0002
17 ID:0001 0001
18 ID:0001 0001
19 ID:0001 ----
```

Figura 3.84: Salida en texto de los procesos ejecutados

### 3.3.5. Guía rápida de SOMPC

Como habrán notado su uso es prácticamente igual al de C, con algunas diferencias, en la siguiente lista repasaremos de manera general los pasos para agregar un proceso a la lista de procesos a ejecutar profundizando menos en los pasos que son iguales a su versión anterior.

1. Diseñar un programa de acuerdo a la arquitectura de la maquina con el lenguaje establecido para **E-CARDIAC PC** en la versión elegida(de 1000 celdas o más).
2. Agregar a ese programa instrucciones que indiquen en que lugar de la memoria cargar cada instrucción, de forma que puedas tener una “tarjeta” con instrucciones pareadas dónde la primera indique la dirección de memoria que ocupará la segunda.
3. El último par de instrucciones de este estilo que debe tener la tarjeta debe contener como segunda instrucción la operación *Halt* que indica el final del programa.
4. Posterior a está instrucción final agregar otro par dónde la primera será *0800* y la segunda será la dirección de inicio del programa, con un código de operación 8. Dónde *0800* indica que se cargue la segunda en la primer dirección del SOMP, en la implementación de 1000 celdas.
5. Para finalizar agregar la instrucción preestablecida, *8985* en la implementación, la dirección de inicio del preámbulo y la única a la que el usuario tiene permitido saltar directamente, con eso se iniciará la carga del programa para convertirlo en un proceso del SOMP.
6. Esperar a que se cargue el programa y que el proceso 0 tenga de nuevo el control para que el usuario siga añadiendo procesos, con un máximo de 5 en esta implementación.
7. Una vez que al menos un proceso haya sido cargado el *cpu executor* empezará la ejecución de el o los procesos mientras el usuario puede seguir añadiendo procesos.
8. Si uno de los procesos del usuario requiere interacción con el usuario deberá usar el modo terminal para añadir los datos que el proceso requiera, recordando que el modo *deck* sólo es entrada de información para el *cpu principal*.

9. Esperar la ejecución de los procesos y cuando haya terminado podrá descargar los resultados del área de *output* dónde cada salida indica a que proceso pertenece(según su identificador estático) y si un proceso finaliza se imprimirá el identificador seguido de varios guiones medios.
10. Una vez terminada la ejecución de todos los procesos el *waiter* se vuelve a activar para dejar al segundo cpu en espera.
11. El cpu principal en paralelo puede seguir añadiendo procesos.

Con la guía podemos notar que salvo unos puntos, lo demás sigue la misma estructura, por que finalmente es una evolución del modelo concurrente que utiliza dos procesadores para hacer que el añadir procesos sea más rápido para el usuario sin perder las ventajas de la concurrencia.

# Capítulo 4

## Conclusiones

Después del recorrido por los distintos modelos de **CARDIAC** hemos aprendido no solo un funcionamiento básico de la computadora viendo como interactúan sus componentes, sino adentrarnos en otros aspectos relevantes de las computadoras modernas como los son el sistema operativo, la concurrencia y el paralelismo.

A pesar de que este modelo nació en 1968, pudimos demostrar que puede seguir siendo muy útil para representar funciones y operaciones de una máquina que es muy parecida a las que usamos cotidianamente. El traslado del papel al software fue de vital importancia para lograr esto, por qué cuando se aumenta el número de celdas de memoria de 100 a 1000 es muy difícil de seguir solo con papel, que la máquina virtual haga los cálculos y movimientos del apuntador por ti hace que te puedas centrar más en las interacciones que hay en los diversos componentes de la máquina y como van reaccionando a cada instrucción que es leída.

La dificultad de presentar una modelo concurrente y uno paralelo también fue bien subsanada por *CARDIAC*, buscando ser los más minimalistas en los modelos construidos para que fuesen claros para el estudiante, pero sin dejar de lado la intención de que con la ayuda de estos modelos el estudiante pueda explorar los conceptos de concurrencia y paralelismo en un ambiente más controlado, sin demasiados conceptos y elementos con los que lidiar. Además de que al analizar estos modelos nace la necesidad del sistema operativo orgánicamente, por lo que con este par de modelos el estudiante puede ver por qué es tan importante el sistema operativo y como sin el construir un modelo de cómputo con capacidades concurrentes y/o paralelas sería una tarea demasiado complicada y el resultado quizá demasiado complejo.

Por terminar, un aspecto realmente importante que me gustaría destacar es que con lo presentado en este trabajo el estudiante tiene herramientas para enfrentar textos más complejos, por supuesto, pero las máquinas virtuales diseñadas le pueden servir para explorar más situaciones o problemas que en este texto ya no planteo, tienen a su disposición un modelo concurrente y uno paralelo y concurrente a los cuales le pueden sacar mucho provecho practicando con ellos y escribiendo más programas en un lenguaje ensamblador muy amigable para un programador.

## Referencias

- Ajdari, M., y Tabandeh, M. (2012, septiembre). Design and construction of an 8-bit computer, along with the design of its graphical simulator for pedagogical purposes. En *2012 15th International Conference on Interactive Collaborative Learning (ICL)* (pp. 1–5). doi: 10.1109/ICL.2012.6402055
- Aspray, W. (Ed.). (1987). *Papers of John von Neumann on computing and computer theory* (n.º v. 12). Cambridge, Mass. : Los Angeles: MIT Press ; Tomash Publishers.
- AT&T Tech Channel. (2012, marzo). *AT&T Archives: The Thinking Machines (Bonus Edition)*. Descargado 2023-08-17, de <https://www.youtube.com/watch?v=clud9I18DXU>
- AT&T Tech Channel. (2015). *The Transistor: a 1953 documentary, anticipating its coming impact on technology*. Descargado 2023-08-18, de <https://www.youtube.com/watch?v=V9xUQWo4vN0>
- Burks, A. W., Goldstine, H. H., y Neumann, J. (1982). Preliminary Discussion of the Logical Design of an Electronic Computing Instrument. En B. Randell (Ed.), *The Origins of Digital Computers* (pp. 399–413). Berlin, Heidelberg: Springer Berlin Heidelberg. Descargado 2023-10-19, de [http://link.springer.com/10.1007/978-3-642-61812-3\\_32](http://link.springer.com/10.1007/978-3-642-61812-3_32) doi: 10.1007/978-3-642-61812-3\_32
- California State University. (s.f.). *The Historical Development of Computing*. Descargado 2023-10-11, de <https://home.csusb.edu/~cwallis/labs/computability/index.html#Final>
- Ceruzzi, P. E. (2012). *Computing: a concise history*. Cambridge, Mass: MIT Press. (OCLC: ocn758392163)
- Computer History Museum. (s.f.). *Computers / Timeline of Computer History*. Descargado 2023-09-30, de <https://www.computerhistory.org/timeline/computers/#169ebbe2ad45559efbc6eb3572083fb7>
- Eric Kim, E., y Alexandra Toole, B. (1999, mayo). Ada and the First Computer. *Scientific American*, 280(5), 76–81. Descargado 2023-10-12, de <https://www.scientificamerican.com/article/ada-and-the-first-computer> doi: 10.1038/scientificamerican0599-76

Gadi Taubenfeld. (s.f.). *Concurrent Programming, Mutual Exclusion (1965; Dijkstra)*. The Interdisciplinary Center, Herzliya, Israel.

Goldstine, H. H. (1972). *The computer from Pascal to von Neumann*. Princeton, N.J.: Princeton University Press.

Hegelbarger, D., y Fingerman, S. (1968, abril). *An instruction manual for cardiac*. Bell Telephone Laboratories. Descargado de [https://www.cs.drexel.edu/~bls96/museum/CARDIAC\\_manual.pdf](https://www.cs.drexel.edu/~bls96/museum/CARDIAC_manual.pdf)

Hord, R. M. (1982). *The Illiac IV: the First Supercomputer*. Berlin, Heidelberg: Springer Berlin Heidelberg. (OCLC: 851370561)

Ifrah, G. (2001). *The universal history of computing: from the abacus to the quantum computer*. New York: John Wiley.

Isaacson, W. (2014). *The innovators: how a group of hackers, geniuses, and geeks created the digital revolution* (First Simon & Schuster hardcover edition ed.). New York: Simon & Schuster.

Leslie Lamport. (2015, junio). The Computer Science of Concurrency: The Early Years. *Communications of the ACM*, 71–76. Descargado de <https://cacm.acm.org/magazines/2015/6/187316-turing-lecture-the-computer-science-of-concurrency/abstract>

Mark Jones Lorenzo. (2017). *The Paper Computer Unfolded: A Twenty-First Century Guide to the Bell Labs CARDIAC (CARDboard Illustrative Aid to Computation), the LMC (Little Man Computer), and the IPC (Instructo Paper Computer)*. Createspace Independent Publishing Platform.

Maurice Vincent Wilkes. (1976). *EDSAC 1951*. Descargado 2023-09-14, de <https://www.youtube.com/watch?v=6v4Juzn10gM>

megardi. (s.f.). *CARDIAC (CARDboard Illustrative Aid to Computation) Replica*. Descargado 2023-08-17, de <https://www.instructables.com/CARDIAC-CARDboard-Illustrative-Aid-to-Computation-/>

Museo Torres Quevedo. (s.f.). *El ajedrecista, el primer juego de ordenador de la historia*. Descargado 2023-07-30, de <https://artsandculture.google.com/story/el-ajedrecista-el-primer-juego-de-ordenador-de-la-historia/>

- NASA. (2020, febrero). *Who Was Katherine Johnson? (Grades K-4)* - NASA. Descargado 2023-10-26, de <https://www.nasa.gov/learning-resources/for-kids-and-students/who-was-katherine-johnson-grades-k-4/> (Section: For Kids and Students)
- Null, L. (2003). *The Essentials of Computer Organization and Architecture*. Jones & Bartlett Learning. (Google-Books-ID: c2K1EAAAQBAJ)
- Null, L., y Lobur, J. (2003, junio). MarieSim: The MARIE computer simulator. *Journal on Educational Resources in Computing*, 3(2), 1-es. Descargado 2023-05-09, de <https://doi.org/10.1145/982753.982754> doi: 10.1145/982753.982754
- O'Regan, G. (2012). *A brief history of computing* (2. ed ed.). London Heidelberg: Springer.
- Pawson, R. (2022, julio). The Myth of the Harvard Architecture. *IEEE Annals of the History of Computing*, 44(3), 59–69. Descargado 2023-10-18, de <https://ieeexplore.ieee.org/document/9779481/metrics#metrics> (Conference Name: IEEE Annals of the History of Computing) doi: 10.1109/MAHC.2022.3175612
- Richards, M. (s.f.). EDSAC Initial Orders and Squares Program. *University of Cambridge*. Descargado de <https://www.cl.cam.ac.uk/~mr10/Edsac/edsacposter.pdf>
- Salomon, D. (1992). *Assemblers and loaders*. New York: Ellis Horwood.
- Silberschatz, A., Galvin, P. B., y Gagne, G. (2009). *Operating system concepts* (8th ed ed.). Hoboken, NJ: J. Wiley & Sons.
- Sipser, M. (2013). *Introduction to the theory of computation* (Third edition, international edition ed.). Australia Brazil Japan Korea Mexiko Singapore Spain United Kingdom United States: Cengage Learning.
- Tanenbaum, A. S. (2002). *Modern operating systems*. (OCLC: 981051666)
- Tanenbaum, A. S., Austin, T., y Chandavarkar, B. R. (2013). *Structured computer organization* (Sixth edition, international edition ed.). Boston Columbus Indianapolis New York San Francisco Upper saddle River Amsterdam Cape Town Dubai London Madrid Milan Munich: Pearson.
- Valvano, J. W., y Valvano, J. W. (2017). *Introduction to the Arm® Cortex(TM)-M Microcontrollers* (Fifth Edition ed.) (n.º 1). s.l.: Eigenverl. d. Verf.

Álvaro Frías. (2022, junio). Retruco: un intérprete para TIMBA. *Electronic Journal of SADIO*, vol. 21, no. 1. Descargado 2023-05-09, de <http://sedici.unlp.edu.ar/handle/10915/142866>