



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

---

FACULTAD DE ESTUDIOS SUPERIORES  
ACATLÁN

CARDIAC: La evolución hacia  
un modelo concurrente y  
paralelo

TESIS

QUE PARA OBTENER EL TÍTULO DE  
MATEMÁTICAS APLICADAS Y COMPUTACIÓN

PRESENTA

**MARTÍN OSVALDO SANTOS SOTO**

ASESOR DE LA TESIS:

DR. JORGE VASCONCELOS SANTILLÁN

Santa Cruz Acatlán, Naucalpan de Juarez 13/05/2024

**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

# Introducción

Hoy en día tenemos multitud de aparatos electrónicos que suelen ser llamados computadoras; celulares, laptops, tabletas, relojes inteligentes, y un sinfín 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; en otras palabras, de *computar*. Esto, siguiendo la definición que recoge el diccionario de Oxford<sup>1</sup>, tiene sentido, pero si analizamos más detalladamente el término “computadora” notaremos que el origen de la palabra es anterior a las computadoras tal cual las conocemos hoy en día.

Por ejemplo, antes de 1950, la palabra “computadora” hacía referencia comúnmente a mujeres que trabajaban en la *N.A.S.A.* y realizaban cálculos manuales. El blog [1] describe esta situación, dedicado a Katherine Johnson (destacada “computadora humana”), nos muestra cómo aun con el machismo que se vivía en la época logró ser ampliamente reconocida, y vemos cómo esa misma palabra que la había descrito a ella en el pasado se empezó a usar para describir a las nuevas máquinas electrónicas de cálculo.

Si viajamos al pasado remoto, notaremos que desde épocas muy tempranas 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 y 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 pensados[2]. 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

---

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

las computadoras que conocemos hoy en día.

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, el cómo 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. 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 cómo explotar de mejor manera los recursos de las máquinas que estás utilizando en cualquier disciplina en la que te desempeñes. Aunado a esto, conocer la historia es otro punto fundamental, ya que te permitirá saber la razón por la cual ciertos aspectos de las computadoras no han cambiado en 80 años y por qué otros han cambiado o evolucionado de manera tan vertiginosa en los últimos, preparándote también para el futuro y dando el reconocimiento que se merecen 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 esta 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. Sin embargo, 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 cómo funciona, *grosso modo*, una computadora actual.

Para esto último me apoyaré en *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[3] con la intención, precisamente, de hacer más fácil de entender a los alumnos de aquel entonces cómo funcionaba una computadora. Por supuesto, no fue el único modelo que existió, pero sí uno muy interesante para abordar los temas

que he mencionado. Tanto este como otros modelos son mencionados en [4], 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* es que, dado que fue creado en 1968, la concurrencia, el paralelismo o la inclusión de un sistema operativo no pasaban por la mente de sus creadores, puesto 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 que un modelo da al abstraer un sistema complejo en los componentes principales que se quieren dar a entender. Utilizando el contexto histórico se darán las razones por las que cada paso en la evolución de las computadoras fue necesario, e incluso a veces imparable, manteniendo la atención principalmente en estos tres conceptos.

Es claro que en todo este 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 [5], 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 artículo [6] centrado en dicho lenguaje; por último, he de mencionar un desarrollo mostrado en el artículo [7] 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 cuatro 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 *CARDIAC*, el texto se verá acompañado, tal como la clásica *CARDIAC* distribuida por *Bell Labs*, con un “kit” que incluye un *software* que contendrá tres máquinas virtuales<sup>2</sup>. Será una para

---

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

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, así dando la posibilidad de que puedan experimentar en un ambiente controlado para poder ver cómo se van ejecutando los programas, lo cual en una versión paralela o concurrente sería un poco más difícil de ver con una computadora de papel.

# Índice general

<b>Introducción</b>	<b>IV</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: <i>The Big Iron Era</i> . . . . .	18
1.2. Más allá de los laboratorios . . . . .	20
1.2.1. Computadoras más compactas . . . . .	20
1.2.2. Lenguajes de programación . . . . .	23
1.2.3. Sistemas operativos y los cambios en el paradigma de programación .	27
1.2.4. Creación de los modelos didácticos de enseñanza . . . . .	32
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 . . . . .	45
2.2.1. Modelo de cómputo concurrente . . . . .	45
2.2.2. Modelo computó paralelo . . . . .	46

2.3. CARDIAC como modelo de computo . . . . .	49
2.3.1. Arquitectura de CARDIAC . . . . .	49
2.3.2. Funcionamiento y lenguaje en CARDIAC . . . . .	52
<b>3. Evolución del Modelo</b>	<b>58</b>
3.1. E-CARDIAC : Electronic CARDboard Illustrative Aid to Computation . . . . .	60
3.2. E-CARDIAC C : Electronic CARDboard Illustrative Aid to Concurrent Computing . . . . .	67
3.2.1. Necesidad de un sistema operativo . . . . .	67
3.2.2. Mejoras necesarias en el Hardware . . . . .	68
3.2.3. Cambios en el lenguaje . . . . .	71
3.2.4. Sistema Operativo Mínimo C: Aspectos Generales . . . . .	73
3.2.5. Sistema Operativo Mínimo C: Un nuevo proceso en la pila . . . . .	82
3.2.6. Sistema Operativo Mínimo C: Lanzamiento de procesos . . . . .	94
3.2.7. Sistema Operativo Mínimo C: Actualización y borrado . . . . .	107
3.2.8. SOMC: Guía rápida de uso . . . . .	119
3.3. E-CARDIAC PC: Electronic CARDboard Illustrative Aid to Concurrent Parallel Computation . . . . .	121
3.3.1. Arquitectura renovada para un modelo paralelo . . . . .	122
3.3.2. Un sistema operativo para dos procesadores . . . . .	126
3.3.3. Funcionamiento del sistema operativo mínimo . . . . .	128
3.3.4. Ejecutando procesos en E-CARDIAC PC . . . . .	142
3.3.5. Guía rápida de SOMPC . . . . .	147
<b>4. Conclusiones</b>	<b>149</b>
<b>Historia de la computación</b>	<b>156</b>
<b>Sistema Operativo Concurrente para E-CARDIAC</b>	<b>165</b>
<b>Sistema Operativo Concurrente y Paralelo para E-CARDIAC</b>	<b>170</b>
<b>Programas para E-CARDIAC</b>	<b>175</b>



# Índice de figuras

1.1.	Fuente: Sydney Padua (2015), The Analytical Engine. . . . .	6
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.	Paquete de CARDIAC abierto. . . . .	34
1.9.	Modelo de CARDIAC construido. . . . .	35
2.1.	Arquitectura CARDIAC, Jorge Vasconcelos(2018) . . . . .	38
2.2.	Obtenida de (Tanenbaum, 20202), a. Paralelismo On chip, b. CoProcesador, c. Multiprocesador, d. Memorias independientes, e. Múltiples computadoras.	47
2.3.	Arquitectura de CARDIAC, concepto original Bell Labs & Jorge Vasconcelos	51
3.1.	Inicio para selección de máquinas virtuales . . . . .	59
3.2.	Arquitectura de CARDIAC por Jorge Vasconcelos, 2018 . . . . .	61
3.3.	Pantalla de inicio de E-CARDIAC . . . . .	62
3.4.	Listas desplegables de E-CARDIAC . . . . .	62
3.5.	E-CARDIAC Muestra de Output . . . . .	64
3.6.	E-CARDIAC Carga masiva por tarjetas . . . . .	65
3.7.	E-CARDIAC Carga individual de instrucciones . . . . .	65
3.8.	E-CARDIAC Cola de instrucciones/datos . . . . .	65
3.9.	E-CARDIAC después de iniciar sus funciones . . . . .	66

3.10. Diagrama de Arquitectura de E-CARDIAC C . . . . .	69
3.11. Diagrama de flujo de SOMC . . . . .	75
3.12. SOMC:Preámbulo . . . . .	76
3.13. E-CARDIAC C Apagada . . . . .	78
3.14. E-CARDIAC C durante el arranque . . . . .	79
3.15. E-CARDIAC C Sistema operativo mínimo cargado . . . . .	80
3.16. Formas de entrar al sistema operativo mínimo C . . . . .	81
3.17. SOMC : Variables del sistema . . . . .	84
3.19. SOMC : Contenidos generales del núcleo . . . . .	85
3.18. Diagrama de segmento para añadir un proceso . . . . .	86
3.20. SOMC: Añadir un proceso, parte 1 . . . . .	87
3.21. SOMC: Añadir un proceso, parte 2 . . . . .	87
3.22. SOMC: Núcleo del sistema operativo . . . . .	89
3.23. E-CARDIAC C Programa en deck . . . . .	91
3.24. E-CARDIAC C Programa en cola . . . . .	92
3.25. E-CARDIAC C Instrucción para añadir proceso . . . . .	93
3.26. E-CARDIAC C Programa 1 cargado en memoria . . . . .	93
3.27. E-CARDIAC C Proceso 1 cargado . . . . .	94
3.28. E-CARDIAC C Tres procesos agregados . . . . .	94
3.29. Acercamiento al preámbulo en el diagrama . . . . .	96
3.30. E-CARDIAC Preámbulo después de una detención en P0 . . . . .	97
3.31. Acercamiento a la parte del bloqueo al proceso 0 . . . . .	98
3.32. SOMC segmento del proceso 0 . . . . .	98
3.33. E-CARDIAC C Borrado de proceso parte 1 . . . . .	99
3.34. SOMC Lanzamiento de procesos . . . . .	101
3.35. SOMC Lanzamiento de procesos parte 1 . . . . .	102
3.36. SOMC Lanzamiento de procesos parte 2 . . . . .	103
3.37. SOMC Lanzamiento de procesos parte 3 . . . . .	104
3.38. Estatus de organizadores antes de lanzar el proceso . . . . .	104
3.39. Valores del SOM antes de lanzar el proceso . . . . .	105

3.40. E- CARDIAC C . . . . .	106
3.41. Diagrama de actualización de proceso . . . . .	108
3.42. SOMC Actualizar proceso . . . . .	109
3.43. Actualizar proceso 1 . . . . .	109
3.44. Conexión entre actualización de procesos y lanzamiento . . . . .	110
3.45. Contexto del proceso 1 actualizado . . . . .	110
3.46. Proceso 1 antes de finalizar . . . . .	111
3.47. Preámbulo en la finalización del proceso 1 . . . . .	112
3.48. Preámbulo en la finalización del proceso 1 a punto de saltar . . . . .	112
3.49. Acercamiento a zona de borrado en diagrama . . . . .	113
3.50. Zona de procesos antes de que el proceso 1 sea borrado . . . . .	114
3.51. SOMC Borrar proceso parte 2 . . . . .	115
3.52. SOMC Borrar proceso parte 3 . . . . .	116
3.53. Zona de procesos después de borrar el proceso 1 . . . . .	116
3.54. Variables del sistema antes de borrar el proceso 1 . . . . .	116
3.55. Variables del sistema después de borrar el proceso 1 . . . . .	117
3.56. Salidas finales de los procesos . . . . .	117
3.57. Salidas finales de los procesos en texto plano . . . . .	118
3.58. Arquitectura de cómputo paralela. . . . .	123
3.59. Diagrama de Sistema Operativo Mínimo Paralelo . . . . .	127
3.60. Añadir proceso en SOMP . . . . .	129
3.61. Preámbulo sistema operativo mínimo paralelo . . . . .	130
3.62. Añadir proceso en un sistema operativo mínimo paralelo . . . . .	131
3.63. E-CARDIAC PC sin iniciar . . . . .	132
3.64. E-CARDIAC PC Booteo . . . . .	133
3.65. E-CARDIAC PC Iniciado . . . . .	134
3.66. Acercamiento al preámbulo del sistema operativo mínimo . . . . .	135
3.67. Acercamiento a segmento de <i>waiter</i> y control de procesos . . . . .	136
3.68. Segmento de control de procesos . . . . .	136
3.69. Acercamiento a fracción de lanzamiento de procesos . . . . .	137

3.70. Sistema operativo mínimo paralelo, fracción de lanzamiento . . . . .	137
3.71. Sistema operativo mínimo paralelo, fracción de lanzamiento, parte 2 . . . . .	138
3.73. Sistema operativo mínimo actualización de procesos . . . . .	138
3.72. Acercamiento al segmento de actualización de procesos . . . . .	139
3.74. Acercamiento a la fracción de borrado . . . . .	140
3.75. Sistema operativo mínimo borrar proceso . . . . .	140
3.76. Sistema operativo mínimo borrar proceso: parte 2 . . . . .	141
3.78. Proceso de Fibonacci cargado . . . . .	142
3.79. Subrutina de proceso de Fibonacci . . . . .	142
3.77. Añadiendo un nuevo proceso en E-CARDIAC PC . . . . .	143
3.80. Programa pintor en el <i>deck</i> . . . . .	144
3.81. Agregando cantidad de números para serie de Fibonacci . . . . .	144
3.82. Primeras salidas de “pintor” . . . . .	145
3.83. Ejecución finalizada . . . . .	146
3.84. Salida en texto de los procesos ejecutados . . . . .	146

# 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 cálculo. A medida que los cálculos se volvían más complejos, necesitábamos herramientas más complejas.

Las primeras herramientas para apoyarnos en esto fueron las representaciones de números de maneras 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 cual 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 antes de nuestra era [2].

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. Tal 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 [2].

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**. 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 [2].

Siguiendo la mencionada evolución, es importante destacar 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 XVII en Europa occidental, saliendo del Renacimiento europeo, en el cual había 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 John Napier, en 1614, fue uno de los grandes avances en la forma de resolver problemas aritméticos. Los logaritmos, *grosso modo*, permiten sustituir multiplicaciones por sumas, lo cual implica una simplificación considerable en el tiempo de resolución de estas operaciones [8, p. 24].

Pero aun así, los cálculos seguían sin ser tan rápidos, por lo que se desarrollaron tablas de logaritmos, ya calculados, que acortaban el tiempo aún más. Otro avance relacionado fueron los dispositivos mecánicos que permitían optimizar algunos cálculos aritméticos. *The Gunter Scale* (la regla de Gunter) y *The Slide Rule* (la regla de cálculo) fueron dos dispositivos que permitían al usuario conseguir tal optimización. La primera fue creada por Edmund Gunter y la segunda por William Oughtred como mejora a la primera. De hecho, *The Slide Rule* siguió evolucionando a lo largo de los años para añadir más funcionalidades, incluido el uso de logaritmos, lo que la mantuvo vigente hasta hace relativamente poco tiempo (especialmente en áreas de ingeniería) [8, 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 inventó la *Step Reckoner*, o simplemente “máquina de Leibniz”, en el año 1673, basándose en *The Pascaline* (Pascal, 1642) ; ambas permitían hacer cálculos aritméticos como la suma y la resta de manera mecánica [8, p.25].

Aunque desde muchos años antes se venía pensando en un dispositivo como la calculadora, la dificultad del proyecto había frenado su desarrollo. El mismo Leonardo da Vinci lo intentó,

pero solo dejó sus planos para construir un dispositivo con esas características, puesto que no lo concretó. Ahora sabemos que el modelo de Leonardo sí era viable, solo que con tecnología más avanzada, dado que ingenieros de la época moderna pudieron seguir sus instrucciones para crear aquel dispositivo [9].

Podemos notar que las inquietudes por automatizar operaciones estaban ya desde antes que la tecnología, de su época, les permitiera a los inventores llevar a cabo sus planes. A pesar de esto, las siguientes generaciones seguían adelante con esas ideas, a veces directamente y otras de forma más independiente; demostrando que el avance teórico es completamente necesario, aunque no se tengan las herramientas en ese momento del tiempo para su desarrollo práctico. Este interesante patrón, donde la teoría avanza más rápido que los desarrollos prácticos, lo veremos de nuevo en repetidas ocasiones en el futuro.

Aunque las máquinas de Leibniz y Pascal eran calculadoras en forma, su replicación no era sencilla y tampoco fueron realmente populares; pero si hay una época donde 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 primera calculadora en forma que se vendió masivamente. En los años sucesivos, continuó recibiendo mejoras y, al final, fue toda una inspiración para una multitud de inventores en todo el mundo [2, p. 127].

El antecesor directo de la computadora, y sucesor casi directo del ábaco, no solo 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 paso? ¿Hacerla más pequeña? Sí, 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 en un dispositivo que fuera capaz de alcanzar nuevos horizontes tecnológicos.

Esa persona era Charles Babbage, un reconocido matemático de su época, miembro fundador de la *Royal Astronomical Society* (1820), inventor y pionero en la investigación de operaciones; alguien muy distinguido en su época, sin duda. Para el año de 1821, estaba diseñando su *Differential Engine* (máquina diferencial), una máquina 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 porque Charles ya estaba pensando en algo incluso más avanzado; aun así, la máquina se logró construir en 1853 por un par de ingenieros suecos que se basaron en los planes de Charles [8, p.201].

En lo que Charles ya estaba pensando, y en lo que se centró a partir de 1833, cuando se dio por terminada la construcción de la *Differential Engine* (porque el maquinista que la construía renunció), era en una máquina llamada *Analytical Engine* (máquina analítica), capaz de realizar cualquier tarea que pudiera ser expresada en notación algebraica, y que disminuía la interacción humana para realizar los cálculos. Era una máquina mucho más compleja; tendría un almacenamiento y un procesador al que Charles llamaba *the mill* (el molino), porque tenía la forma de un molino y era la parte central 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 [8].

Charles pensó en usar estas tarjetas para representar números, que a su vez se pudieran utilizar para realizar las operaciones aritméticas de la máquina, y de esta forma, que su máquina fuese “programable”. Incluso consideraba dos tipos de tarjetas:

1. Tarjetas de operación
2. Tarjetas de datos

Con esta última idea, de hecho, se puede vislumbrar algo muy parecido a las computadoras con arquitectura von Neumann (que se analizarán más adelante): un procesador, almacenamiento interno y programas almacenados (en forma de las tarjetas mencionadas) [8, p.204]. Alguien que vio más una computadora, como la conocemos hoy en día, que una calculadora muy avanzada, como la percibía su propio creador, fue Lady Ada Lovelace, una

---

<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 vídeo: *Punch Card Programming*, <https://www.youtube.com/watch?v=KG2M4ttzBnY>

matemática que estaba entusiasmada con el trabajo de Charles y que era realmente brillante. En 1843, escribió un artículo que nombraría *Notes*, al final de una traducción que realizó del francés al inglés de un trabajo escrito por Luigi Menabrea sobre la *máquina analítica*. En este artículo, detalla el funcionamiento de la máquina, las cualidades que la hacen diferente a las calculadoras de la época, y ejemplos de programas para cálculos realmente complejos, como el cálculo de los números de Bernoulli. De hecho, este último se considera por muchos como el primer programa de la historia [10].

Por supuesto, tenía una cercanía muy alta con Charles, como se deja ver en su correspondencia, lo que le permitió llevar a cabo un trabajo muy completo. Esté artículo, junto con las notas de Ada, posteriormente sería publicado en la revista *Scientific Memoir* bajo el título *Sketch of the analytical engine invented by Charles Babbage*[10].

Entre los elementos más valiosos que nos dejó, se encuentran, por supuesto los programas y los detalles del funcionamiento de la máquina que realizó (junto a Menabrea), pero también su visión sobre el trabajo de Charles, que era un tanto diferente a lo que él mismo consideraba. Ella pensaba que la máquina podría actuar sobre algo más que números, si se creaban las relaciones correctas con los números, como representación de algo abstracto, podrían fungir como símbolos para resolver más tareas. Ada consideraba a la máquina analítica como algo muy lejano a las calculadoras de su época, y lo era en muchos sentidos, por lo que, 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 primera computadora(mecánica) de la historia[10]. Podemos ver una representación de esta en la figura 1.1, donde también vemos a Ada y Charles trabajando con ella.

En los años siguientes, el avance fue continuo con las máquinas calculadoras de propósito específico, por ejemplo, se desarrolló, por el fundador de la empresa hoy conocida como IBM, una máquina de censos que usaba el concepto de las tarjetas perforadas para contabilizar datos. También la tecnología en general evolucionó: la era de la electricidad y lo electromecánico llegó a finales del siglo XIX, cambiando por completo las cosas. Junto al invento del tubo de vacío y avances teóricos como el álgebra de Boole, entre otros, fueron parte fundamental

---

<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.

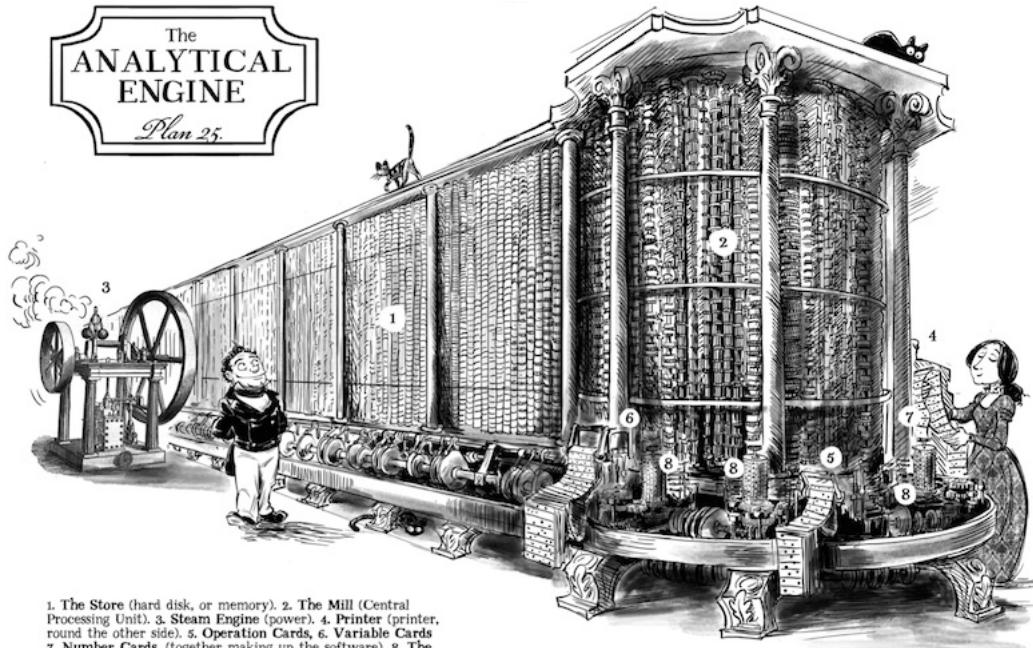


Figura 1.1: Fuente: Sydney Padua (2015), The Analytical Engine.

de las continuas invenciones del siglo XX [2, p. 127].

Me parece importante destacar que no es una sucesión lineal de hechos o descubrimientos lo que llevó 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 fue lo que condujo al descubrimiento de nuevas tecnologías a partir de la unión de todas estas áreas de estudio. 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 XX con la llegada de la electricidad y la electromecánica. Para el año 1913, el matemático español Leonardo Torres Quevedo desarrolló su *Aritmómetro electromecánico*, evitando las dificultades que había tenido Babbage gracias a la electromecánica, le fue posible culminar este desarrollo que resolvía diversos problemas aritméticos. Este invento, además, incluía una máquina de escribir como entrada [2], [11].

Este no sería el único aporte de Quevedo a la computación, 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 “máquina que imita la figura y los movimientos de un ser animado”, Quevedo desarrolló la idea centrada en las posibilidades de las máquinas como calculadoras, pero que pudieran reconocer más objetos, más situaciones y resolver problemas más difíciles, para así quitarles trabajos repetitivos a los humanos. Él, en su afán de demostrar que su teoría tenía sentido, desarrolló 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 sí podía terminarla [2], [11].

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 máquina para hacer algo más que lo que se lograba con una calculadora. Motivados por *los problemas de Hilbert*, una serie de problemas matemáticos que buscaban dar más rigurosidad a las matemáticas desde el principio del siglo XX, Alan Turing y Alonzo Church continuaron ese camino al comenzar a trabajar, por separado, en resolver un problema muy particular: el famoso *problema de la parada* [2].

Sin entrar en mucho detalle, trata sobre la posibilidad de determinar si todo problema matemático reproducible en un algoritmo puede ser resuelto; si una máquina que está ejecutando el algoritmo se detiene siempre con seguridad, significa que sí puede ser resuelto. Siendo un algoritmo una lista de instrucciones ordenadas y finitas que permite solucionar un problema. La resolución del *problema de la parada* es muy compleja, y no es la idea del texto entrar en tales detalles, pero sí dar a entender lo importante que fue para la computación su

solución [2].

Se desarrollaron modelos teóricos de máquinas que podían “computar”, en el sentido más clásico de la palabra, es decir hacer cálculos, para resolver este problema. Turing trató de crear una máquina lo más general posible, de forma que cualquier problema se pudiera computar en esa “máquina de propósito general”; a tal máquina la llamó *Máquina Universal de Turing* en su famoso artículo *On Computable Numbers, With an Application to the Entscheidungsproblem* (Sobre números computables, con aplicación al problema de la parada)<sup>4</sup>[2].

Un modelo totalmente teórico, pero que mostraba todo el potencial de una máquina de propósito general que podía resolver cualquier problema que pudiese ser expresado como un algoritmo. Sin embargo, no podía resolver todos los problemas aritméticos, porque 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 basándose en un algoritmo [2].

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 cómputo. Esta se subdivide en varias ramas, una particularmente interesante es la *teoría de la computabilidad*. Esta teoría estudia cuáles algoritmos puede computar una máquina para llegar a establecer si ciertos problemas son *no computables* y, por ende, no son solubles por medio de ninguna computadora [2, p. 272].

La forma de pensar sobre las máquinas ha evolucionado mucho a este punto en la historia; la visión ahora es sobre una máquina que dé 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 son dos nombres fundamentales en esta ciencia, que en ese momento aún no existía, de la teoría de la computación. En esos mismos años, la evolución de 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 según la definición general que se tiene de estas. Repasaremos lo sucedido entre 1930 y 1946, cuando por diversas causas, la madurez en el entendimiento de las máquinas de cálculo, de la electromecánica y de la electricidad, así como la necesidad 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 calculadoras, pero que se fueron transformando hasta convertirse en máquinas que resolvían problemas más allá de la aritmética.

Como en esta parte de la historia se centrada la discusión de cuál fue la primera computadora de la historia, es necesario tener una definición más clara de lo que entendemos por computadora, porque ciertas máquinas están en el limbo entre ser calculadoras o computadoras, mientras que otras ya son computadoras en un sentido más completo. Empecemos con la definición del diccionario de Oxford, que uso dado que es entre Gran Bretaña y Estados Unidos donde se da principalmente el desarrollo de las computadoras en sus inicios, por lo tanto, es su *lingua 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. 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. Profundizando en la idea de la segunda definición, en [12] 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 numéri-

cos.” A partir de estas definiciones podemos establecer cuatro características fundamentales en una computadora:

1. Es un dispositivo electrónico.
2. Es capaz de almacenar datos.
3. Es capaz de procesar datos.
4. Realiza operaciones a partir de un conjunto de instrucciones dadas por el usuario, entendiendo conjunto de instrucciones/programa como una forma de algoritmo.

Podemos notar fácilmente que un dispositivo de este estilo tiene más funciones que una calculadora, pero hay un punto importante para nuestro estudio, ¿que no sea un dispositivo electrónico es suficiente para que una máquina que tiene las otras tres características se deje de considerar como computadora? La mayoría de los autores lo manejan directamente como un dispositivo electrónico y no se involucran en una discusión más profunda del tema. Como veremos, realmente un dispositivo mecánico o electromecánico puede cumplir con el resto de especificaciones, pero dado que su uso se limitado a los inicios de la computación, y que quedaron rápidamente superados por la potencia de los dispositivos electrónicos, la definición tradicional de computadora se quedó únicamente con estos últimos.

### **Época previa a la segunda guerra mundial**

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>. Esta máquina podía resolver una mayor variedad de ecuaciones, lo que la convirtió en la primera calculadora análoga multipropósito [2, p.158].

---

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

Otra calculadora realmente llamativa de ese momento fue la *Complex Number Calculator*, creada por Samuel Williams y George Stibitz en los laboratorios Bell en 1939, una calculadora electromecánica capaz de manejar números complejos. Esta contribución no fue única por parte de Stibitz, ya que en Bell Labs desarrolló muchos conceptos relacionados con la comunicación y la computación, generando así una de las etapas más brillantes para Bell Labs [2, 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, que leía instrucciones de una tarjeta perforada. Para 1939, Konrad llevaría más lejos esta idea, eliminando la dependencia de las partes mecánicas, que eran muy complejas, y sustituyéndolas por relés para funcionar con circuitos eléctricos y puertas lógicas (*AND*, *OR*, *NOT*). Esto lo logró aprovechando las ideas de Claude Shannon, quien introdujo la idea de implementar el álgebra booleana mediante relés eléctricos para crear circuitos; y así lo hizo Konrad en su *Z2* [2, p.206].

Su siguiente gran trabajo no tardó en llegar, y es por el que es recordado principalmente: la **Z3** (se puede ver en la figura 1.2), máquina que fue terminada en 1941. Usaba 2,600 relés, realizaba aritmética de punto flotante, tenía una 'longitud de palabra'<sup>6</sup> de 22 bits, contaba 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, incluido [8], como la primera computadora de programas almacenados de la historia.

Lamentablemente, resultó destruida en 1943 por un bombardeo, y aunque fue posteriormente reconstruida para demostrar sus capacidades, en su momento fue un impedimento para afirmar que fue la primera computadora. De hecho, la Z3 es más parecida a las computadoras actuales que a otras de más renombre, como la *ENIAC*. Incluso, en 1998, Raúl Rojas probó que la Z3 es *Turing completa*, lo que demuestra que era una computadora con la capacidad de computar cualquier problema que pudiera ser expresado como un algoritmo [13].

---

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

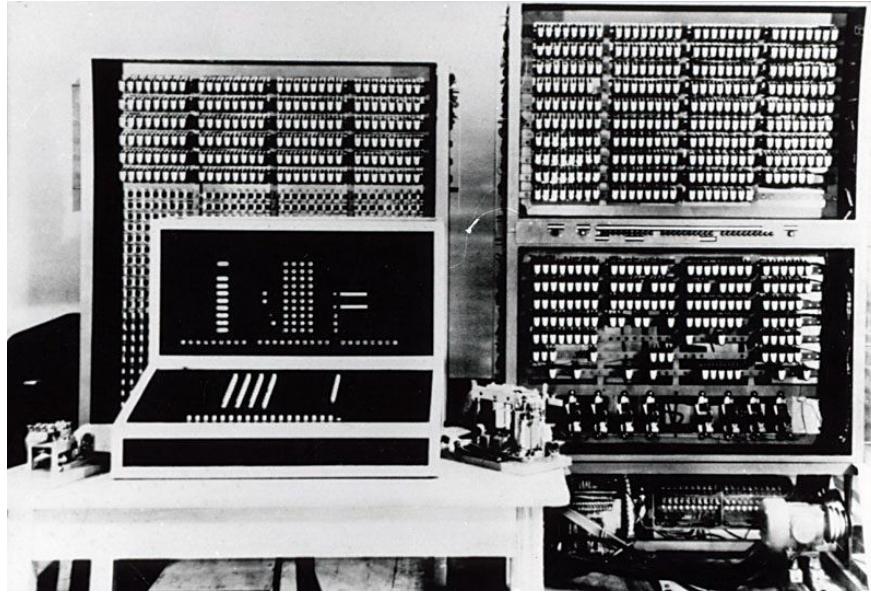


Figura 1.2: Fuente: Computer History Museum, The Zuse Z3 Computer.

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 sobre Konrad en el resto del mundo fue alto, en los últimos años, los museos y los autores le han dado el lugar que merece como uno de los padres de la computación [2, p.206].

### Época de la Segunda Guerra Mundial

La discusión sobre quién construyó la primera computadora fue un tema de controversia, tanto que se llevó a tribunales en Estados Unidos. Esto se debió a que los creadores de la computadora llamada *ABC* afirmaban que John Mauchly, co-creador de *ENIAC*, había visto previamente a la ABC. Finalmente, el dictamen estableció que el concepto de la computadora fue la concepción de diversas ideas por muchas personas, por lo que no era patentable. Una resolución que tiene mucho sentido dada la historia que hemos revisado sobre la creación de las computadoras, ya que muchas mentes han aportado en diversos aspectos a la concepción de lo que hoy en día conocemos como computadoras [14].

Justamente, veamos a uno de los implicados en la disputa, que siguió al desarrollo de Zuse, pero sin relación con él. La computadora fue nombrada como *The Atanasoff-Berry Computer (ABC)*, construida por el profesor John Vincent Atanasoff en el Colegio Estatal

de Iowa con la ayuda de su estudiante Clifford Berry entre 1939 y 1942. Tenía un sistema binario para la aritmética, memoria para guardar datos, circuitos electrónicos y separación entre datos y programas; era, en toda regla, una computadora, aunque no programable y de propósito específico [2, pp. 212-218].

Otro desarrollo en paralelo, pero esta vez en el M.I.T., fue la *Harvard Mark I*. Fue construida por Howard Aiken y un equipo con el apoyo de IBM, destinada a ayudar con cálculos balísticos en la Segunda Guerra Mundial; una máquina electromecánica que fue la primera en ser capaz de imprimir tablas matemáticas, algo que Babbage soñó casi un siglo atrás [2, pp. 212-218].

Algo interesante a destacar de esta máquina 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 es cada vez más relevante, pero que en su momento no fue la arquitectura central del desarrollo de las computadoras [15].

Después de revisar algunas computadoras un tanto desconocidas, nos quedan dos que son quizá las más famosas de la época. Empezando con la *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 máquina de descifrado llamada *Bombe* lograron desencriptar los mensajes de la famosa máquina *enigma*, pero con el avance de la guerra se encontraron con un problema aún mayor: la máquina *Lorenz SZ40/42*, que tenía una codificación de muy alta calidad y se usaba únicamente para los mensajes más importantes en la armada alemana [8].

Para descifrar estos códigos entra en escena Tommy Flowers, diseñando la *Colossus Mark 1*, una máquina semiprogramable 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; estuvo disponible a principios de 1944, y su segunda versión se lanzó 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

---

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

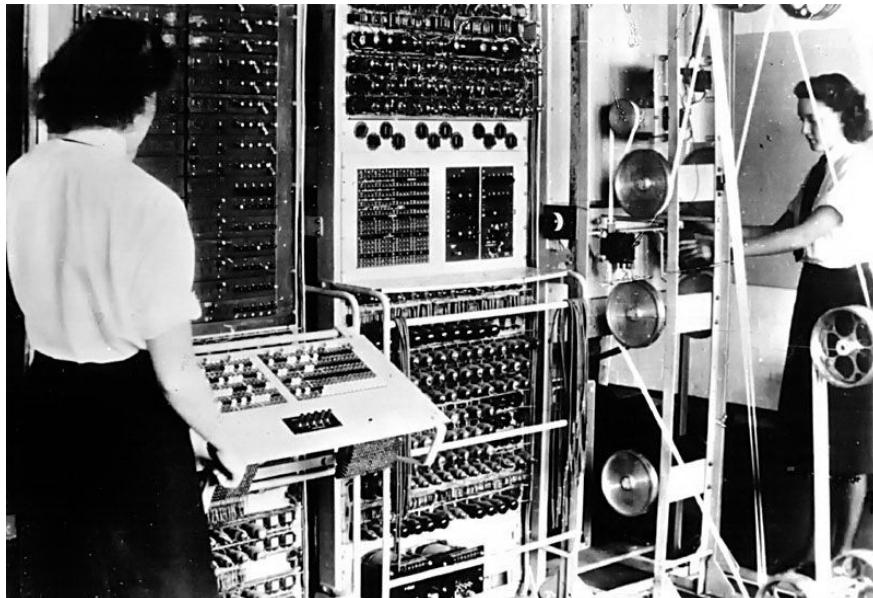


Figura 1.3: Fuente: Computer History Museum, Colossus Mark 1.

una réplica de la máquina en el museo de Bletchley Park [8, 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* (*Electronic Numerical Integrator and Computer*); computadora construida por John Mauchly, J. Presper Eckert, y un equipo en el que se encontraba como consejero externo John von Neumann, en *The Moore School of Electrical Engineering* de la Universidad de Pensilvania. Esta computadora, a pesar de ser de propósito específico, al ser programable, se le considera una de las primeras computadoras de propósito general [2].

Su programación era muy compleja, ya que requería de mover o reordenar interruptores manualmente e ingresar la información por medio de tarjetas perforadas, lo cual era realmente tedioso y tomaba demasiado tiempo. Además, si le sumamos que los tubos de vacío que usaba no eran muy confiables, puesto que estos explotaban fácilmente, tenemos una máquina que no era muy confiable y bastante lenta de programar. Aun así, fue un aliado valioso al final de la guerra, pues fue puesta en ejecución en diciembre de 1945 para uso militar [2].

## Época posterior a la Segunda Guerra Mundial

Fue precisamente en la Segunda Guerra Mundial donde se dieron algunos de los avances más importantes en la historia de la computación, donde se cambió la forma de concebir las computadoras, pasando de pensar en ellas como máquinas que solo resolvían cálculos aritméticos simples, a máquinas que resuelven problemas más avanzados, descifran códigos, calculan trayectorias para misiles, y demás tareas propias de una guerra. La necesidad del avance tecnológico llevó a estos desarrollos a su cúspide, pero con el final de la guerra los desarrollos no pararon, sino que se incrementaron, y la idea de 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 Goldstine y John von Neumann escriben un ensayo llamado *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument* (discusión preliminar de la lógica en el diseño de un instrumento electrónico de computación), para sintetizar las ideas que existían sobre estos dispositivos electrónicos, obtener una imagen de la situación en la que estaban y presentar cómo los problemas matemáticos podían ser ahora codificados en un lenguaje que la máquina entendiera [16]. Es en este documento donde se establece por primera vez la arquitectura de una computadora con *programas almacenados*, un concepto fundamental que, hasta hoy en día, la mayor parte de las computadoras utiliza.

Notaremos que en el título de su ensayo no usan en ningún momento la palabra 'computadora', y es que para la época lo que tenían era un dispositivo de cómputo electrónico. Como hemos venido leyendo, el uso generalizado de la palabra computadora se daría durante el mismo desarrollo de estos dispositivos.

Von Neuman es alguien que aportó mucho a la computación. Poco antes, ya había escrito un reporte llamado *First Draft of a Report on the EDVAC* (Primer borrador de un reporte sobre *EDVAC*), puesto 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, uno de ellos fue Maurice Wilkes, quien

desarrolló su propia computadora con programas almacenados, la *EDSAC* (*Electronic Delay Storage Automatic Computer*) [8].

La construcción de la *EDVAC* tenía la intención de mejorar en los aspectos donde la *ENIAC* fallaba. Un punto fundamental es que estaba construida con una arquitectura de programas almacenados, es decir, que no se necesitaba reconectar cables para poder ejecutar una tarea distinta, lo que a su vez ayudaba a reducir los errores y facilitaba su programación. Se empezó a construir en 1946 y comenzó a operar en 1951 [8, p. 45].

Otro de los desarrollos que continuó al final de la guerra, o para ser más preciso, comenzó al final de la guerra, es el desarrollo de una computadora de propósito general en Manchester. Fue en Inglaterra, con Manchester como uno de sus principales centros académicos, en donde Tom Kilburn y Frederic Williams desarrollaron la primera computadora completamente electrónica, digital y con programas almacenados. En 1948 se culminó la creación de la llamada *Manchester Small Scale Experimental Computer*, mejor conocida como *Baby*, porque era más pequeña de lo común en la época [8].

Esta computadora era más bien un prototipo que luego extenderían en la conocida *Manchester Mark I*. De hecho, ganaría tanta notoriedad por sus avances que una compañía británica llamada *Ferranti Ltd.* se asoció con Kilburn y Williams para comercializar una computadora basada en la que ellos habían construido. Esta llevó el nombre de **Ferranti Mark 1**, y fue la primera computadora electrónica de propósito general comercializada; fue lanzada en febrero de 1951, poco antes de la **UNIVAC**, que fue lanzada en marzo de 1951 [8, p. 36].

## Inicios de la comercialización

Para cerrar esta época, está quizá una de las computadoras más recordadas, creada también por Mauchly y Eckert: la muy conocida *UNIVAC* (*UNIversal Automatic Computer*) (figura 1.4), diseñada como evolución de su propio desarrollo anterior, la *BINAC* (*BINary Automatic Computer, 1949*), la cual era la primera computadora electrónica con programas almacenados en los Estados Unidos, que basaba su diseño, a su vez, en la *EDVAC* [8].

Ya era una computadora con programas almacenados, como lo eran la mayoría en esa época. Esta decisión no es casualidad, las ventajas al momento de escribir instrucciones para



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

estas máquinas son muy significativas y marcaron un punto de inflexión para el desarrollo de las computadoras. El aporte de los programas almacenados puede parecer algo simple, pero es tan importante que se mantiene vigente hasta hoy en día.

Incluía un teclado y una consola para escribir; era una computadora de negocios en su totalidad, que fue entregada a la Oficina de Censos en marzo de 1951 (en su versión 1) y se mantuvo en comercialización para buscar otros compradores. Sin embargo, no era fácil vender una máquina tan grande, que tenía usos muy específicos, y cuyo costo de más de un millón de dólares no alentaba a los compradores, que en su mayoría eran departamentos del gobierno de Estados Unidos, como la *U.S. Air Force*, *U.S. Steel* o la *U.S. Navy* [8, p.43].

Como podemos notar en la línea de tiempo del apéndice A, el desarrollo de las computadoras experimentó un crecimiento muy acelerado en poco tiempo, en menos de 20 años se pasó de apenas tener la idea de una calculadora muy potente a la de máquinas completamente electrónicas que resolvían cualquier problema descrito en forma de algoritmo. El paso del desarrollo de las computadoras a manos de entidades privados, que aprovechaban los descubrimientos hechos en la época de guerras y añadían los suyos, permitió el gran despegue en términos comerciales, así como una mayor difusión del conocimiento sobre las

computadoras entre las masas.

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

De acuerdo a Tanenbaum, en [17], 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 cual 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* o unidades centrales. La tercera generación fue la del **circuito integrado**, que podía juntar cientos de transistores en un espacio realmente pequeño, permitiendo así 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

A finales de 1954, IBM estrenaba su primera computadora producida en masa, la *IBM 650*, vendiendo alrededor de 450 aparatos en ese año, y siendo una de las últimas grandes computadoras que usaban tubos de vacío. Porque desde principios de los años 50 se dio a conocer el desarrollo del **transistor** por parte de tres inventores en Bell Labs (Shockley, Bardeen y Brattan); este invento permitía diseñar circuitos eléctricos para las computadoras remplazando los (muy poco confiables) tubos de vacío por estos nuevos semiconductores, con los que podían representar los números binarios (0 y 1) a través de cargas eléctricas y crear circuitos lógicos más complejos al unirlos [17].

---

<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.

Dado que eran más confiables, duraderos y eficientes, los costos y tamaños de las computadoras se redujeron considerablemente. Empezó 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; solo grandes compañías, universidades o el gobierno podían pagar los millones que valían [8], [17].

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 porque unos estudiantes del MIT escribieron el primer videojuego 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 continua innovación [8].

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 máquina especializada para negocios. Y, por si fuera poco, en 1961 estrenarían el tope de su serie 7000, la llamada *IBM 7030 Stretch*,

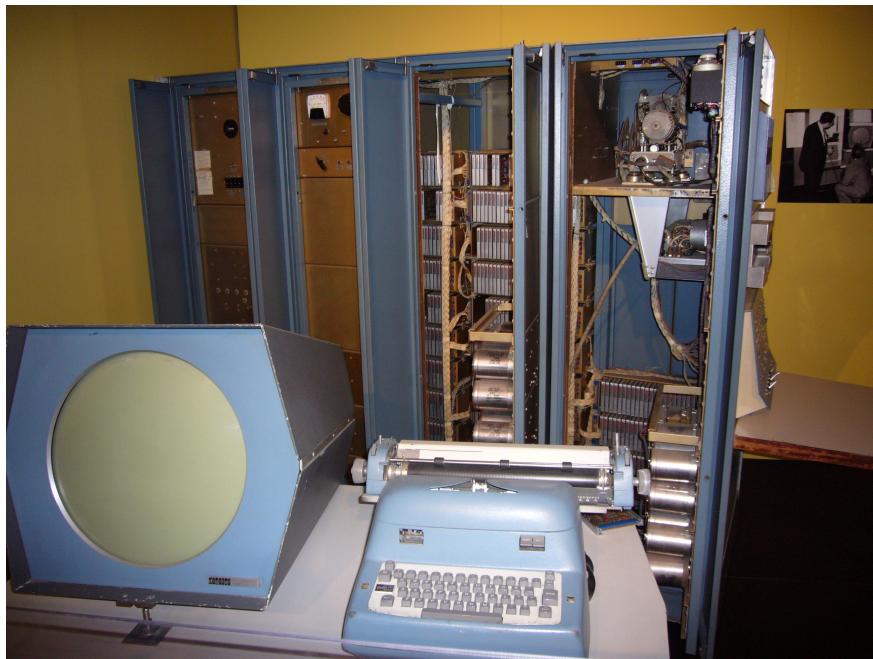


Figura 1.6: Fuente: Matthew Hutchinson - Flickr, Ordenador PDP-1.

realmente veloz y que representaba los avances más grandes que tenía IBM en ese momento [8], [14].

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 [8]. 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 solo estaban empezando.

## 1.2. Más allá de los laboratorios

### 1.2.1. Computadoras más compactas

El término *computadora* era cada vez más recurrente en universidades y empresas, pero solo aquellas con un alto presupuesto podían darse el lujo de tener una unidad central en sus oficinas. Con la intención de ampliar el mercado, algunas compañías estaban trabajando en

productos comerciales que estuvieran al alcance de más entidades. El avance más relevante en computación que se daría en la década de 1960, y que marcaría el inicio de la tercera generación de computadoras, logró precisamente eso.

El **circuito integrado** marcaría esa gran revolución en la computación. Inventado por Jack Kilby, tuvo su primera demostración práctica en 1958; un microchip de silicio (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, rápidas y 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 supuesto, marcó el inicio de una era en que las computadoras podían salir de los laboratorios [5].

La evolución de la *unidad central* (o *mainframe*) fue la *minicomputadora*, que por su tamaño podía caber en un auto (fig.1.7). El término podría parecer poco intuitivo para el lector del siglo XXI, dado que las computadoras actuales son varias veces más pequeñas, pero en ese momento se había pasado de ocupar todo un cuarto para una computadora a solo 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 debido a la potencia que ofrecía en comparación a su precio, que era incluso 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 [5].

Una de las computadoras más pequeñas del momento fue la *Apollo Guidance Computer (AGC)*, computadora diseñada específicamente para el viaje de la nave *Apollo 11* a la Luna, y en la que lograron reducir el tamaño de una computadora a solo 61 centímetros, cuando usualmente podían tener el tamaño de varios escritorios. La terminaron en 1968 y fue todo un hito en la ingeniería del momento. Este logro fue realizado por un equipo especializado del MIT que fue convocado por el gobierno de Estados Unidos para ayudar en la misión espacial [14].

La tercera generación también fue un punto de inflexión para los sistemas operativos y los lenguajes de programación, que en los inicios ni siquiera eran considerados temas realmente



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

relevantes. El hardware primaba sobre cualquier avance en software, pero en esta época la importancia del software comenzó a ser cada vez más fuerte [17].

El final de la tercera generación llegaría con la invención del **microprocesador**, un solo chip que podía contener miles de circuitos integrados. Pero que no se detenía ahí: podía albergar casi todos los componentes eléctricos de una computadora, permitiendo que pudieras tener la unidad central de procesamiento (CPU, por sus siglas en inglés) en la palma de tu mano [5].

La primera microcomputadora<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 [5].

El cambio era enorme: se pasaba de tener artefactos que solo 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)* cerrando por completo la tercera generación e iniciando a lo grande la cuarta [5].

---

<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 como intermediarios entre las personas y las máquinas. Sin embargo, en esta sección quiero ofrecer un breve repaso histórico sobre este aspecto fundamental cuando hablamos de computadoras, y es que tales lenguajes son la forma en que nos “comunicamos” con estos aparatos para resolver los problemas computacionales que enfrentamos.

Para ser más claros, deberíamos definir lo que es un algoritmo, que informalmente hemos descrito como una colección de instrucciones simples para resolver alguna tarea. Esta definición nos sirve, en lo suficiente, para entender el rol que tienen los algoritmos en la programación; una definición más formal es dada en la tesis de Church-Turing [18]. 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 máquina entienda.

La comunicación, entendiéndose como la forma de traducir instrucciones entre humanos y computadoras, estaba 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 máquina*, que son las instrucciones en números binarios o decimales, dependiendo de la arquitectura que tenga la máquina (las actuales son en su mayoría binarias). Por ejemplo, para programar las primeras máquinas de propósito específico, como *ENIAC* o *Colossus*, se usaban enchufes y cables<sup>10</sup> para poder definir patrones en lenguaje máquina que daban la instrucción a la computadora de realizar una tarea en específico [17, p. 8].

Pero a medida que las computadoras fueron aumentando su complejidad y podían resolver problemas más difíciles, el traducir estos problemas a lenguaje máquina se volvía una tarea muy ardua. Las tarjetas perforadas y los procesos almacenados permitieron desechar los enchufes y cables, las tarjetas te permitían crear los patrones para el lenguaje máquina, y

---

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

los procesos almacenados conseguían que esta información se guardara en la memoria de la computadora. Aun así, llevar un problema a lenguaje máquina era una tarea en la que se consumía mucho tiempo. De ahí que empezaron a surgir ideas para hacer este proceso más amigable con el usuario. Fue en ese tiempo cuando se empieza a gestar la idea del “lenguaje ensamblador”, que en la definición que da Solomon en [19] nos dice:

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

Es decir, se crea un lenguaje simbólico para sustituir las instrucciones del código máquina por otras más simples, por medio de un *ensamblador*. Aunque al principio se utilizaba solo el lenguaje simbólico para describir un programa de forma más entendible para los programadores, y luego era traducido manualmente por otras personas a lenguaje máquina, con el tiempo se fueron creando “traductores” automáticos a partir de estos lenguajes.

En el video [20], Maurice Wilkes nos muestra el proceso de programación que se seguía en la *EDSAC*, la cual tenía un lenguaje simbólico para describir los programas, pero se requería de una traducción manual para convertir el código simbólico del programa al lenguaje que la computadora entendía. A pesar de esa traducción manual, podemos observar una traducción automática en la misma *EDSAC*, ya que había un programa llamado *Initial Orders* que se cargaba al inicio en la máquina y funcionaba como un traductor, o lo que hoy llamaríamos un *ensamblador arcaico*, porque traducía algunos códigos muy específicos que facilitaban la programación [19], [21].

Lo siguiente para facilitar la programación fue la creación de ensambladores más desarrollados, para que el programador solo necesitara escribir el programa en lenguaje simbólico y el ensamblador pudiera traducirlo a lenguaje máquina. En 1953, la *IBM 650* ya incluía un ensamblador llamado *SOAP (Symbolic Optimizer and Assembly Program)*, que daba mucha más facilidad a los programadores para desarrollar sus programas en un lenguaje más amigable; solo ocho años después de la *ENIAC* y su programación con enchufes [19].

Es en esta época, mediados de los años 50 e inicios de los años 60, cuando toma fuerza la idea de ir más allá de los lenguajes de ensamblador, que si bien eran una mejora, aún se debía seguir prácticamente la misma lógica para programar. Pese a la facilidad de no tener

que programar únicamente con números, se requería una simplificación mayor en la escritura de programas debido al aumento en la complejidad de estos; se necesitaban **lenguajes de alto nivel** [8].

La idea de estos lenguajes era facilitar la lógica de programación, de manera que el programador ya no tuviera que preocuparse por direcciones de memoria, y pudiera centrarse en otros aspectos más sustanciales, como la optimización y calidad de su programa. A estos lenguajes se les conoce como la **tercera generación** de lenguajes de programación, antecedidos por el lenguaje máquina y el ensamblador como primera y segunda generación, respectivamente [8].

Se debe hacer mención especial a **Plankalkül**, que traducido significa algo así como “cálculo 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,<sup>W</sup> lenguajes que en el mundo “occidental” no llegarían hasta mediados de los años 50. Lamentablemente, este y sus demás descubrimientos quedaron sepultados<sup>11</sup> por mucho tiempo a causa de la Segunda Guerra Mundial [8].

No sería hasta mediados de los años 50, en la era de los *mainframes* y de las primeras minicomputadoras, cuando 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 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 [8].

El problema con estos lenguajes de alto nivel es que, al igual que con el lenguaje ensamblador, se requería un programa para traducirlos a código máquina, o incluso transformarlos a lenguaje ensamblador para que este actuara como intermediario. Se tenían dos opciones principales: un *intérprete* que fuese traduciendo línea a línea de código en tiempo de ejecución, o bien un *compilador* que transformara todo el código de alto nivel a lenguaje máquina

---

<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 este lenguaje y su creador.

y posteriormente permitiera ejecutar ese código las veces que se deseara. Ambos procesos eran costosos en tiempo de procesamiento, y esta fue una de las razones por las que tardó tanto la aceptación de los lenguajes de alto nivel [8].

En los años siguientes, el desarrollo de nuevos lenguajes de programación, junto a sus intérpretes o compiladores, continuó; desde Pascal, C y Basic, pasando por otros más modernos como Python y Java, junto a muchos más que se han seguido desarrollando con una idea en común: facilitar la programación. Que el programador no se torture buscando formas de transmitir sus ideas a la máquina y que se enfoque en diseñar los algoritmos correctos para resolver sus problemas.

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

Possiblemente, la frase **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*, o que nuestra computadora seguramente tiene un sistema llamado *Windows* o *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, las máquinas más potentes y los usuarios buscaron optimizar su tiempo. Aun así, un sistema operativo reconocible por alguien del siglo XXI aparecería hasta los años 80, por ejemplo, con el ahora legendario **MS-DOS** para las computadoras de IBM, y potencialmente para cualquier empresa que pudiera pagar por su licencia.

Pero, ¿qué es un sistema operativo? La respuesta no es simple. Sin embargo, para explorar su evolución podemos describir un sistema operativo como un programa especial dentro de la máquina que realiza dos tareas principales: ser una conexión entre los demás programas y el hardware del ordenador, y gestionar los recursos del hardware [17].

#### Primeras apariciones de sistemas operativos

En el libro [17], Tanenbaum nos presenta la evolución de los sistemas operativos siguiendo el esquema de las generaciones de computadoras, empezando en la **primera generación** (1945-1955) cuando los sistemas operativos eran prácticamente inexistentes. Un programa de esa generación que tenía funciones relacionadas al sistema operativo fue *initial orders*, un ejemplo muy temprano de lo que puede hacer un *loader* (iniciador), que es básicamente un programa que carga programas en la memoria. Esto no es una tarea sencilla, ya que debe llevar cada instrucción desde una memoria secundaria a su correspondiente lugar en la memoria principal [19].

La **segunda generación** (1955-1965), que se encuentra en la época de los *mainframes* y de los primeros lenguajes de alto nivel, nos dejó los sistemas **batch**, o sistemas por lote, como ancestros más directos de los sistemas operativos. En esta época ya había más usuarios

trabajando en computadoras, por lo que se requería de un nivel de servicio cada vez más alto. El que cada usuario fuese con una máquina, dejara a su programa, y esperara a que el operador le diera respuesta era poco eficiente; la solución que generalmente se adoptó para resolver esto fue la de los sistemas por lote [17].

Básicamente, es una forma de trabajo en la que se colecta un conjunto de lotes (o *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 que se hayan terminado de ejecutar. En este flujo, además de los programadores, también tienen relevancia los operadores que llevaban esos *lotes* y los cargaban en las máquinas; además, tenían que cargar los *iniciadores* cuando eran requeridos y los compiladores si se había trabajado en algún lenguaje de alto nivel. Entre los programas especiales, como los iniciadores, y los operadores, realizaban el trabajo que hoy se asocia a los sistemas operativos [17].

Para este punto, ya existía la necesidad de un programa que administre los recursos de la máquina, y hacia la **tercera generación** de computadoras (1965-1980) se vislumbraría 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 [17].

La estrategia que adoptó IBM para eliminar esa división fue crear una familia de computadoras que compartieran ciertas características en el ámbito de hardware, y un sistema operativo común llamado **OS/360**. Este sistema operativo se libero en 1964 y tenía la intención de funcionar en todas las computadoras de esta familia siendo el enlace para los demás programas, de forma que no se tuviera que programar de manera distinta entre una máquina y otra [17].

El problema con este sistema operativo es que, para funcionar tanto en computadoras orientadas a hacer pronósticos del clima u otros cálculos complejos, así como ejecutar operaciones para ambientes más comerciales, tales como la impresión, se convirtió un programa realmente complejo. Construido con millones de líneas en lenguaje ensamblador, era realmente difícil de mantener [17].

Sin embargo, es inevitable pensar en un programa de millones de líneas de código si se quieren cumplir todos los requerimientos de un sistema operativo . Los libros [17] y [22],

representan en sus portadas, con un toque de sátira, la complejidad de un sistema operativo. Silberschatz lo muestra con la clásica portada de dinosaurios, haciendo referencia al libro escrito por Fred Brooks, uno de los diseñadores de OS/360, que critica la complejidad del programa usando dinosaurios. Tanenbaum, por su parte, lo representa con un circo que tiene una gran cantidad de involucrados, simbolizando a los “participantes” del sistema operativo.

## Sistemas operativos y concurrencia

Ahora que se tenía un sistema operativo que agilizaba las tareas humanas, surgieron otras formas de ejecutar los programas para aumentar la eficiencia; ahora se podía conseguir la **concurrencia**. La concurrencia no es más que la ejecución de varios procesos en “simultáneo”, es decir, se ejecuta parte de un proceso A y luego parte de un proceso B, buscando la eficiencia y que ambos usuarios obtengan sus resultados sin esperar a que el proceso del otro termine [17].

Entendemos **proceso**, cuando hablamos de computación, como un conjunto de variables y de código que se estará ejecutando en la máquina. Se parte de un programa, un código que contiene un algoritmo para realizar una tarea, este programa se convertirá en “proceso” cuando entre en la pila o lista de ejecución, ya que se le asignará un identificador único, y se resguardarán ciertas variables asociadas a tal programa. Esto es lo que permite que, en la multiprogramación, aunque suspendas un proceso en algún punto, cuando se le devuelvan los recursos de ejecución pueda continuar como si nunca se hubiera detenido, manteniendo todas sus variables asociadas intactas [17].

Particularmente, en ese tiempo cobró mucha notoriedad una técnica llamada *multiprogramming/multitasking* o multiprogramación. Esta técnica buscaba explotar al máximo los “tiempos muertos”, generalmente causados cuando el ordenador esperaba alguna instrucción por parte del usuario. En las máquinas comerciales, alrededor del 80 % o 90 % del tiempo se utilizaba en esto; en la multiprogramación, se le asignaba ese tiempo de procesamiento a otro programa que estuviera en espera [17].

Esta forma de computación es la que actualmente se utiliza en las computadoras que tenemos, dando la sensación de que todos los programas se están ejecutando al mismo tiempo. Aunque en aquellos tiempos el procesamiento no era lo suficientemente rápido para pensar

esto, la comodidad para los usuarios al usar computadoras mejoró notablemente [17].

La posibilidad de usar concurrencia en las máquinas dio paso al **timesharing** o tiempo compartido, que es básicamente el uso de los recursos de una misma máquina por diferentes usuarios. Esto significaba tener una unidad central a la que varios usuarios, con diferentes interfaces, podían acceder y ejecutar programas sobre ella, mientras que internamente la unidad central repartía tiempos de ejecución, intentando no dejar tiempo muerto en ningún momento. El objetivo del tiempo compartido era proveer a los usuarios una respuesta más inmediata y de optimizar el uso de los recursos [17].

De hecho, su invención vino 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 ni con el hardware ni con la seguridad necesarios para su uso masivo por parte de los usuarios, los sistemas operativos que permitían concurrencia ayudaron a solventar estos problemas [17].

Por tal razón, la popularización del tiempo compartido en la tercera generación de computadoras fue alta, y llegó tan lejos que el M.I.T., Bell Labs y *General Electric (GE)* intentaron construir un sistema operativo llamado **MULTICS**, que funcionaría sobre máquinas conectadas por la red eléctrica para soportar cientos de usuarios. Por supuesto, la tarea era demasiado ambiciosa para su tiempo, y aunque al final GE y Bell Labs salieron del proyecto, el sistema operativo terminó funcionando de la mano del MIT, aunque con diferencias respecto al plan original [17].

Este sistema tuvo una gran influencia en el desarrollo de **UNIX**, que a su vez es una gran influencia para sistemas como *GNU/Linux*, *macOS* y *FreeBSD*. Quizá el concepto de tiempo compartido, tal como se conocía en esa época, no llegó a nuestros días, pero el de **Cloud Computing** o computación en la nube es una clara evolución, con máquinas miles de veces más potentes para un uso masivo de miles de usuarios [17].

## Sistemas operativos y paralelismo

En 1969, el sistema operativo *MULTICS* ya era capaz de trabajar con múltiples unidades de procesamiento en **paralelo**, al mismo tiempo, un concepto que no puede faltar en las computadoras actuales [22, p. 899]. Esto fue un avance importante en la construcción de

computadoras cada vez más potentes. Ya no se trataba de la ilusión que generaba la concurrencia al aparentar que se ejecutaban dos procesos al mismo tiempo, con el paralelismo, eso se volvía una realidad. Sin embargo, es necesario contar con un sistema que pueda manejar estos procesadores y aprovecharlos de manera eficiente.

Cabe aclarar que no es la única forma de paralelismo; puede haber paralelismo en los datos, en los procesos, e incluso a nivel instrucción, entre muchos otros [23].

Pero sus inicios no fueron fáciles; requerían de hardware muy especializado para ser realmente útiles, por lo que no hay muchos casos documentados de computadoras o sistemas paralelos en la tercera generación de computadoras. La *ILLIAC IV* es un ejemplo de una 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. Culminó su desarrollo a mediados de los años 70 a pesar de sus problemas de construcción, principalmente relacionados con el hardware de la época [24].

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 ni una arquitectura enfocada al paralelismo. No fue hasta la década de 1990 y principios de los 2000, que el ascenso de las computadoras con múltiples procesadores comenzó [17].

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

Estamos a mediados de la década de 1960, época del estreno de *Star Trek*, que comenzaba a maravillar a las personas con su ciencia ficción y sus computadoras capaces de resolver cualquier problema. Era poco antes del primer viaje a la Luna realizado en el Apolo 11, y sobre todo, en un tiempo en que el desarrollo tecnológico no hacía más que crecer.

Como ya leímos en secciones anteriores, es la época donde las computadoras empiezan a ser más pequeñas. Curiosamente, a computadoras como la *PDP-1* y la serie de computadoras que detonó, se les llamaba “minicomputadoras”, dado que la reducción de tamaño comparada con otras, como la *ENIAC*, era enorme. Una época en la que los sistemas operativos aún no llegaban al uso masivo, y los primeros lenguajes de alto nivel estaban apenas apareciendo entre los usuarios.

Uno de los problemas en ese 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. Así que se buscaron alternativas a la pura teoría que pudieran hacer de este un mejor proceso; fue entonces cuando varias empresas y universidades, principalmente de Estados Unidos, empezaron a desarrollar modelos didácticos de enseñanza de las computadoras que no requerieran de una computadora real. Y es que, aunque había universidades con bastante dinero, como el MIT, que tenían algunos modelos de computadoras para la investigación, el acceso para los alumnos en general era difícil, por no decir imposible [25, p. 71].

Una de esas compañías era **Bell Telephone Laboratories**, mejor conocido como *Bell Labs*. En aquellos tiempos, era un centro de trabajo e investigación muy prestigioso, y parte de la poderosa *American Telephone and Telegraph*. Esta, a pesar de que su negocio principal eran las telecomunicaciones, tenía un área de investigación dedicada al desarrollo de nuevas tecnologías [26].

Pero *Bell Labs* no solo se dedicaba a la investigación, también tenía una sección dedicada a la enseñanza, en la cual se asociaba con universidades para distribuir materiales y paquetes de aprendizaje de diversos temas. Por ejemplo, posterior a la invención del transistor, sacaron un documental junto con un paquete electrónico que incluía un pequeño transistor para que

los interesados 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 YouTube *AT&T Tech Channel* (canal de la empresa) recopila muchos de estos documentales, un ejemplo es el documental del transistor en [27].

Estas acciones no eran altruismo para *Bell Labs*, pero a los estudiantes de las escuelas donde 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 de algún material de apoyo, como el que compartía *Bell Labs*. Cabe destacar que, incluso en la época actual, hay muchas zonas del mundo que dependen de materiales de apoyo y libros, en la medida en que estos estén disponibles, para continuar sus estudios.

Así fue como, en 1968, los laboratorios Bell lanzaron un *kit* acompañado con un video llamado **Thinking Machines**, el cual se puede consultar en [28]. En él, narran a través de la pregunta “¿las computadoras piensan?”, la lógica que siguen las computadoras para resolver las tareas que se les asignan y los funcionamientos internos que tienen para solucionar los problemas que se les plantean.

El paquete que acompaña el video era un modelo de cómputo llamado **CARDIAC**. La Real Academia de la Lengua Española define “modelo” como:

*Esquema teórico, generalmente en forma matemática, de un sistema o de una realidad compleja, como la evolución económica de un país, que se elabora para facilitar su comprensión y el estudio de su comportamiento.*

Precisamente eso lo que crea *Bell Labs* con este paquete: un modelo, que no es una computadora real, sino un esquema que sirve para facilitar la comprensión de una computadora real y compleja, aislando únicamente los elementos que se quieren explicar.

En la caja de cartón venía el manual de instrucciones, que se puede consultar en [3], y unas hojas de papel como se muestra en la figura 1.8. Con estos elementos, el estudiante podía comenzar la construcción de su propia “computadora de papel”; en la figura 1.9 se puede ver la “computadora” construida [29].

A la derecha, en la imagen 1.9, tenemos la memoria principal, y en particular podemos

ver un espacio 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 donde llegan los datos desde la memoria y se realizan las operaciones aritméticas y lógicas, que se depositan en el acumulador, ubicado en la parte más izquierda de la imagen.



Figura 1.8: Paquete de CARDIAC abierto.

En esta computadora, es la mente del usuario la que realiza las operaciones siguiendo el flujo que tiene establecido el modelo, de manera que el usuario puede observar cómo se van moviendo los datos a través de las diferentes unidades, cómo se van activando determinadas partes de la unidad central de procesamiento, y cómo esos datos pueden terminar de vuelta en la memoria principal. El proceso completo de un dato sería salir de la memoria principal, pasar a la unidad central de procesamiento y ser descifrado por los registros correspondientes para evaluar si es un dato simple o una instrucción, para finalmente entrar en la zona de la evaluación aritmética y lógica, y producir un resultado que se guardará en el acumulador; esos datos del acumulador pueden regresar a la memoria principal para ser utilizados nuevamente o para ser “impresos” en la salida [3].

Este proceso es, de forma muy simplificado, lo que realizan nuestras computadoras hoy en día, aunque nosotros solo vemos los resultados. Es por esta razón que al estudiante de la actualidad, que ya cuenta con una computadora, le puede ser de mucha utilidad CARDIAC

para razonar y analizar los procesos que se ejecutan en una computadora y entender su funcionamiento.

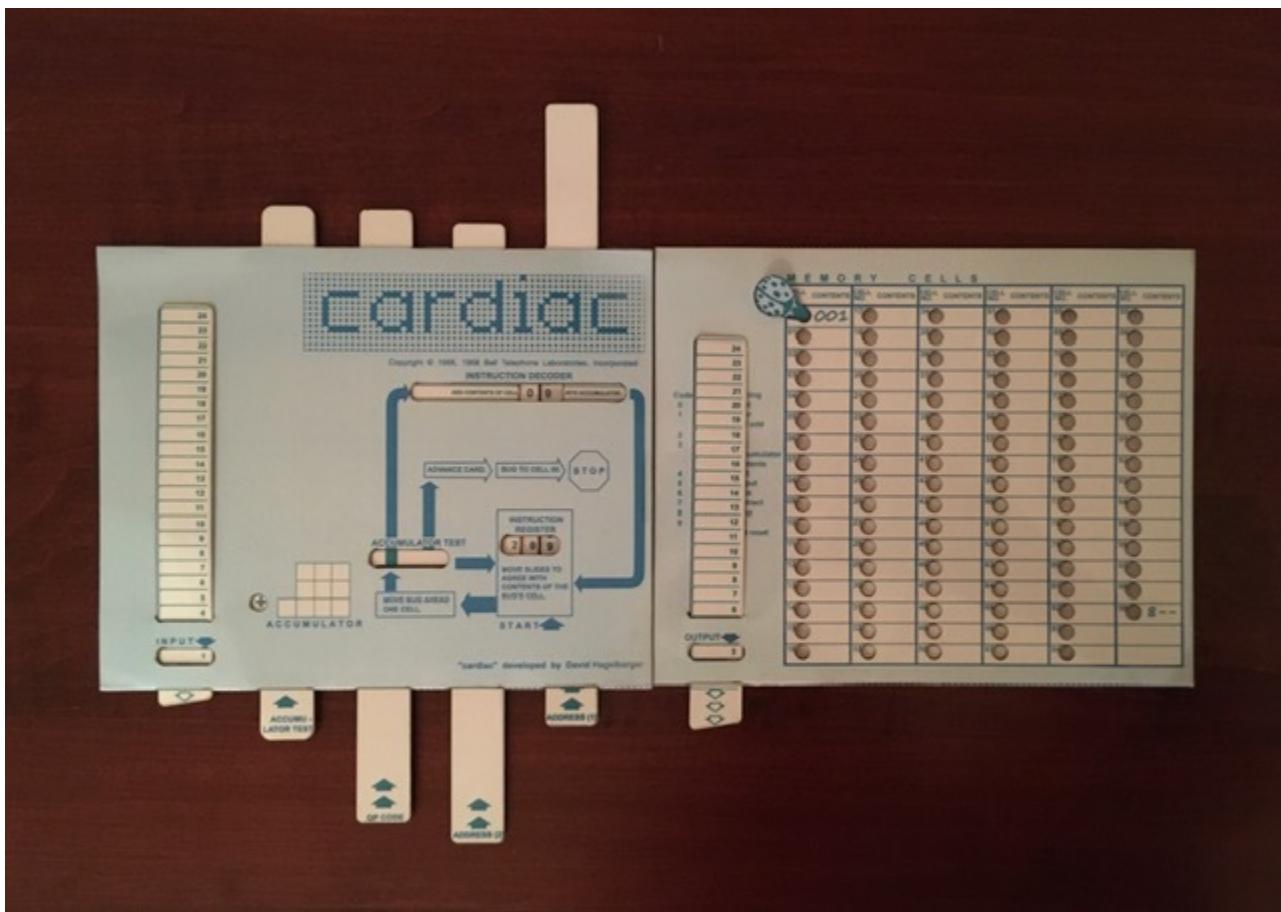


Figura 1.9: Modelo de CARDIAC construido.

### 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 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, y otros que incluso solo han mejorado. Incluso la arquitectura que se ha vuelto muy popular con los teléfonos móviles, es una versión moderna de la ya lejana arquitectura Harvard que viene en los procesadores ARM [30, p. 109].

También hay cambios bastante notables, como el cómputo en la nube o computo distribuido, que toma la herencia del ahora innecesario tiempo compartido(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 lo incluían.

Quizá las dos más grandes novedades de la computación moderna, los procesadores ARM y el cómputo 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 siempre continua.

# Capítulo 2

## Arquitectura básica de las computadoras

En este capítulo revisaremos la composición de una computadora, es decir, los elementos que contiene una computadora y su organización . En la terminología usual de la computación [31] 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 usaré un enfoque a través de modelos para mostrar solo algunas partes relevantes de la computadora, dejando fuera muchos objetos importantes en ella, pero logrando así una simplificación necesaria para lograr un acercamiento claro a la estructura de una computadora.

### 2.1. Funcionamiento de las computadoras

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

#### 2.1.1. Arquitectura Von Neumann

En [32], 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 concentró en ese escrito, siendo *EDSAC* una de las primeras máquinas que seguían de lleno estas ideas. Siendo la idea más representativa la de los *programas almacenados* (en memoria).

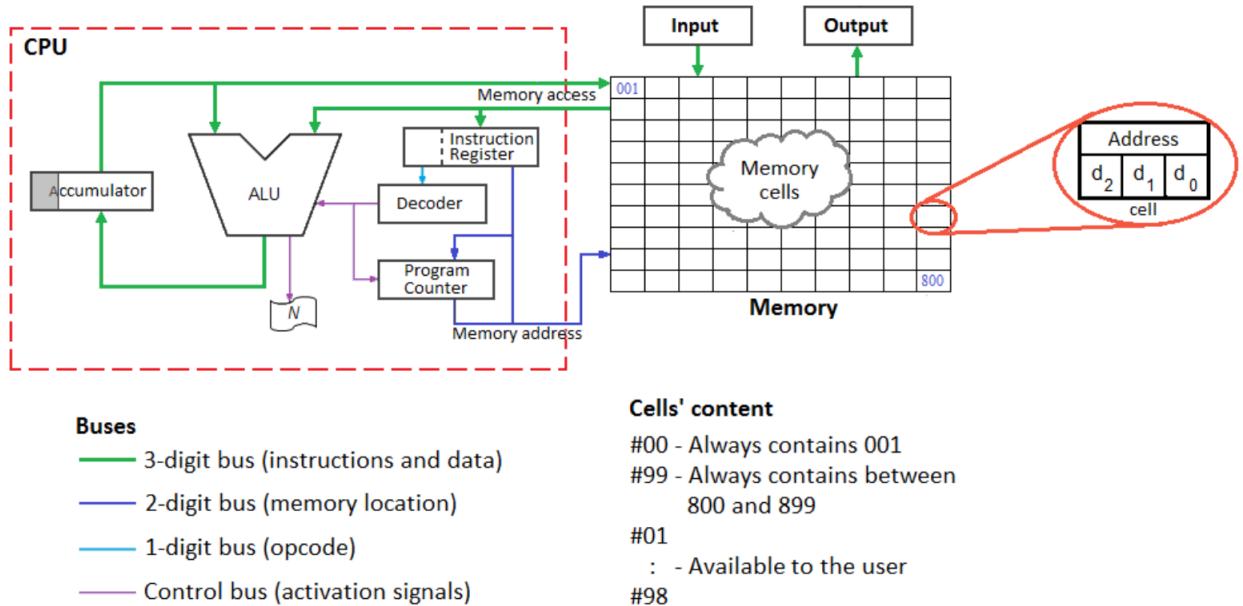


Figura 2.1: Arquitectura CARDIAC, Jorge Vasconcelos(2018)

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 cual 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 esta fue una de las ideas claves para llevar las máquinas de cómputo 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 [31].

Esto nos lleva a su segundo punto, si tanto datos como instrucciones van a estar en la misma memoria, ¿cómo 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 por él **Instruction Register**, o registro de instrucciones, como instrucciones o puramente datos, y en caso de ser instrucciones ser decodificadas por el **Decoder**/Decodificador para pasar a la siguiente unidad, la unidad que hace los cálculos aritméticos, la unidad aritmético lógica(**ALU** por sus siglas en inglés)[32].

La *ALU* 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**, es una unidad separada de la memoria principal en el que se puede almacenar poca información, pero de forma muy rápida, no se usará muy a menudo ese nombre por qué todos los objetos dentro de la unidad de control son registros[32].

Se necesita además de una ejecución secuencial, de forma que una vez se lea una instrucción/dato de la memoria principal continúe 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**, o contador de programa, logra esto en CARDIAC al incrementar de uno en uno su valor y apuntar así a la siguiente dirección de memoria, 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 a otra dirección[32].

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 él para comunicar los diferentes registros dentro de la máquina[32].

Por supuesto, se necesita también una forma de comunicación con el usuario, de modo que se puedan transmitir las instrucciones a la máquina y que esta las ejecute de forma automática. Para ello se necesita de dispositivos de entrada y salida(**Imput/Ouput**), para que el usuario sea capaz tanto de dar instrucciones a la máquina como de recibir resultados de esta[32].

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 artículo describió con más profundidad su modelo, que acompaña con detalles técnicos a nivel arquitectura y a nivel de implementación, dónde también incluyó una *jerarquía de memorias*, que no aparece en el modelo de *CARDIAC*, en la cual además de la memoria principal añade una memoria *cache* y una *memoria secundaria*[32].

Siendo cada una más lenta que la anterior, pero con mucha más capacidad, puesto que el problema, desde aquellos tiempos, hasta la actualidad es la competencia entre velocidad contra capacidad de almacenamiento. La memoria secundaria es lo que hoy conocemos como disco duro, y la memoria caché es un intermedio entre la memoria principal(hoy RAM) y la memoria secundaria, permitiendo aumentar la velocidad en las ejecuciones de procesos[32]. Una vez que se tiene la arquitectura se define el lenguaje de la máquina, aunque ciertamente hay una correspondencia bilateral en estos dos conceptos, dado que tanto el lenguaje define a la máquina como la máquina al lenguaje [31]. El lenguaje de una máquina 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[32].

Con todas estas unidades, registros, y un lenguaje, tenemos un modelo bien definido que cumple con la arquitectura de von Neumann. 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 máquina con tal de tener más potencia y efectividad de cálculos[31].

Aun 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 distintas mejoras.

### 2.1.2. Sistema Operativo

En el capítulo primero vimos cómo se fue dando la necesidad de tener un *software* que pudiera ser el intermediario entre la máquina y otros programas, así como administrar los recursos de la misma. En esta sección exploraremos las características principales de un sistema operativo para lograr tales objetivos.

Como comenta Tanenbaum en [17], depende de a quién leas, será la aproximación que te dé sobre los sistemas operativos, ya sea más orientado a ser una extensión de la máquina 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 con arquitectura von Neumann. Una de esas funciones es ser una extensión de la máquina, en el sentido de presentar a los programas(del usuario) una abstracción de la misma. Por ejemplo, un programa de visualización de imágenes no tiene que ser programado para entender cómo funciona internamente un disco duro, en cuál de los discos que lo conforman debe buscar la imagen, o cómo se debe comunicar con este dispositivo. Es el SO quien crea una capa de abstracción en la que se encarga de comunicarse con el disco duro y de 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 [17].

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 máquina. Para esto, debe tener los permisos de acceso más altos que cualquier otro programa, pues puede requerir ejecutar instrucciones que para un programa de usuario pueden ser peligrosas [17].

En la otra rama, está su actividad como administrador de recursos, que es una consecuencia inmediata del control que tiene sobre la máquina. 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 máquina de 2GB de RAM es prácticamente inservible para el usuario, mientras que, en esa misma máquina, un sistema operativo como GNU/Linux Ubuntu 20.0 puede funcionar de manera óptima para el usuario por su mejor gestión de recursos del sistema. Por ende, la administración de recursos es vital para un buen sistema operativo, la cual va desde los dispositivos de entrada y salida, hasta los tiempos de ejecución de cada **proceso** en un modelo concurrente o paralelo. Dichos modelos son precisamente a los que está orientado de manera fundamental un sistema operativo, pues en un modelo simple de cómputo no es tan necesario [17].

Un proceso es básicamente un programa en ejecución, por lo que necesita más información que solo las instrucciones del programa; necesita tener acceso al espacio de direcciones para escribir o leer de este, la información del contador de programa (para saber su ubicación en todo momento), una lista de archivos abiertos, su relación con otros procesos, y conocer las variables globales. Con todo esto se forma un paquete al que llamamos **proceso** [17].

La información de los procesos debe estar almacenada en la memoria principal, pues se necesita un acceso rápido y solo estar disponible mientras la máquina esté encendida. Esta información se guarda en una especie de **tabla de procesos** en la que cada proceso tiene un identificador único, la cual es administrada por el sistema operativo, quien decide los tiempos de ejecución de cada proceso [17].

Tanto el espacio de memoria donde se encuentra la tabla de procesos, como el espacio

de memoria donde está el sistema operativo porque, en efecto, el sistema operativo es un proceso en sí mismo que se encuentra en ejecución dentro de la memoria principal, deben ser protegidos. Por lo tanto, estas y otras áreas son espacios cuyo acceso es restringido por el mismo sistema operativo, restricción que aplica para todos los programas de usuario. Esta protección puede estar desde el mismo *hardware*. Como se mencionó al principio, el sistema operativo también se encarga del sistema de archivos, el cual requiere de la restricción de accesos para su protección, tarea que también lleva a cabo el SO [17].

### 2.1.3. Iniciando la computadora

Como continuación de la sección anterior es necesario hablar del inicio de operaciones de la computadora, del **Booting** o arranque<sup>1</sup>, la acción de cargar un programa inicial que permita al usuario interactuar con la computadora. Muchas computadoras en el pasado no necesitaban de esto, especialmente cuando tenían que mover cables para cambiar rutinas de ejecución. Pero desde 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[17].

La idea general es tener almacenado un programa en una memoria no volátil, y un sistema que haga cargar esas instrucciones en la máquina. Así, cuando encienda la máquina se van a cargar estas instrucciones, iniciando la máquina con un programa ya cargado. Es el programa a través del cual se van a cargar el resto de programas a la memoria principal para su ejecución. El no tener un sistema de arranque implica que el usuario debe programarlo para interactuar con la máquina, o bien realizar un proceso más tedioso modificando directamente las direcciones de memoria[17].

Hoy en día cada computadora tiene un sistema de arranque en la memoria **ROM Read-Only-Memory**, memoria de la cual no hemos hablado, pero que en algunos modelos se coloca en el mismo espacio que la memoria principal. Es una memoria de solo 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[17].

Este pequeño sistema de arranque funciona para preparar las operaciones básicas de la

---

<sup>1</sup>Forma en la que se le dice al inicio de operaciones de una computadora.

computadora y en el caso de tener un sistema operativo de iniciarla. 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[17].

## 2.2. Modelos de computación

### 2.2.1. Modelo de cómputo concurrente

Ya se mencionó que en la tercera generación de computadoras cobró mucha notoriedad la “multiprogramación”. La cual 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”[17].

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[17].

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[33].

El campo de estudio de la concurrencia es bastante amplio, se dice que comenzó en 1965, con la presentación de Edsger Dijkstra 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 de cada uno de ellos a los recursos[34]. 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 máximo estos

problemas de sincronización.

### 2.2.2. Modelo computó 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 múltiples *CPUs*, que pueden ejecutar múltiples instrucciones al mismo tiempo, otro ejemplo es el de procesar múltiples cadenas de datos en un solo procesador[5].

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. 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[5].

El **pipelining** es uno de ellos, una forma de **paralelismo a nivel de instrucción**, también conocida como “paralelismo de bajo nivel”, debido a que no logra aumentar tanto las capacidades de cómputo como otras formas si lo hacen. Se caracteriza por ser una “línea de producción”, es decir que cada etapa en el proceso de cómputo es tratada como una etapa en una línea de producción de forma que cuando una instrucción está en la etapa de cálculo 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[5, p.421].

Un 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 la línea de producción se consigue una eficiencia muy alta al no desperdiciar tiempos, pero el manejo de los recursos es realmente complicado para el sistema operativo por los recursos compartidos que se tienen, y que se están modificando[5, 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

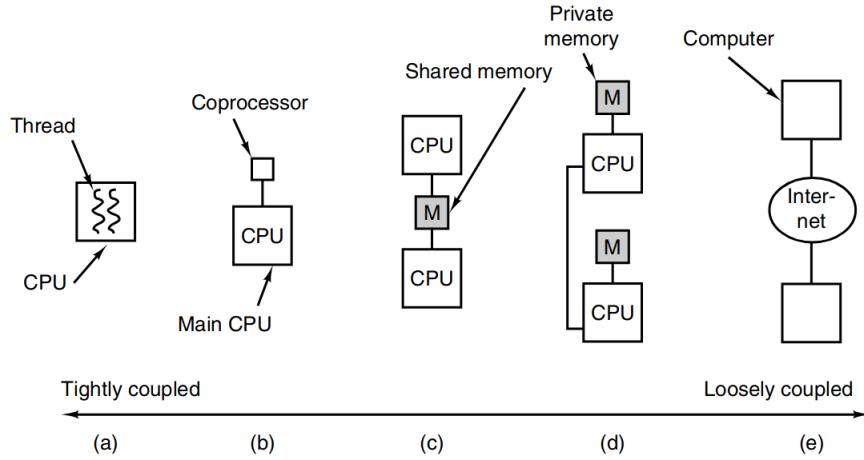


Figura 2.2: Obtenida de (Tanenbaum, 20202), a. Paralelismo On chip, b. CoProcesador, c. Multiprocesador, d. Memorias independientes, e. Múltiples computadoras.

individual para cada procesador, o podría ser que esta también fuese compartida. También está la alternativa de que cada procesador tenga su propia memoria (parte d de la figura 2.2), permitiendo más independencia a cada procesador. En todos los casos el problema de la sincronización es clave para lograr que el cómputo en paralelo sea realmente funcional y no termine siendo más costoso, riesgoso y lento[5].

Las otras formas de paralelismo que podemos ver en la figura 2.2, son, en la parte a, el paralelismo a nivel de hilos, usando un solo procesador, o varios, tratando de explotar al máximo el tiempo de ejecución, usando generalmente la línea de producción para conseguirlo. 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 las entradas y salidas, que suelen ser unos de los procesos más complejos que tiene una máquina[31].

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 síncrona con otra. un cómputo distribuido [31].

Las computadoras actuales generalmente usan todos estos recursos, por qué si la concurrencia y el *pipelining* son técnicas para eficientar al máximo un solo 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 solo juntar diversas técnicas y formas de paralelismo, cada procesador o computadora tienen características que permiten ciertas formas de paralelismo/concurrencia diferentes, y su implementación depende del diseño que estas tengan[31].

## 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 cómo 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* [3].

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 [3].

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[3].

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* [3].

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 parecido pasa con los ciclos en *CARDIAC*. Un ciclo empieza cuando el *Program Counter* o

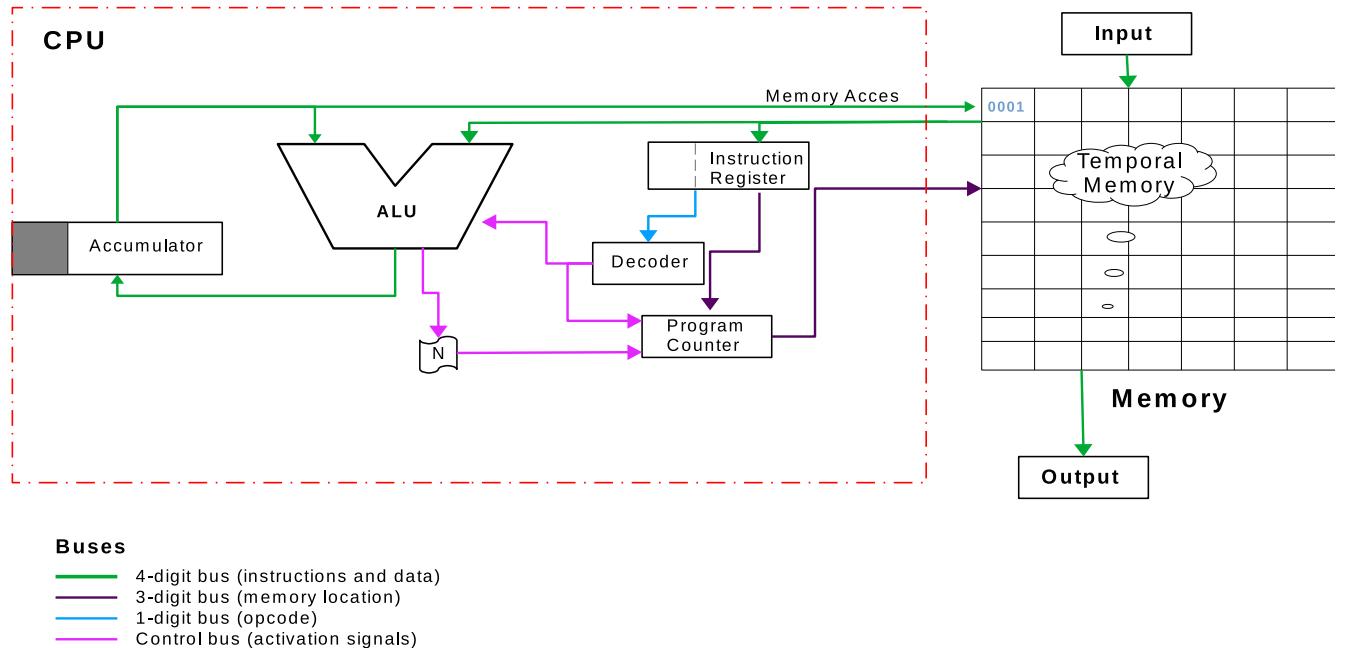


Figura 2.3: Arquitectura de CARDIAC, concepto original Bell Labs & Jorge Vasconcelos

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 registro de instrucciones, que se encarga de separar el dígito a extrema izquierda  $d_2$  para enviarlo al decodificador por medio de un bus azul, y que este decodifique que código de operación es para mandarle esa indicación a la unidad aritmético lógica[3].

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 envió una señal por medio del bus rosa que se lo indicó [3].

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 *Negativo(N)*, que es una especie de booleano, el cual indica si un número que ha sido depositado en el

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 solo cuatro dígitos(uno más que la memoria principal), y es dónde se almacenarán todos los cálculos que realice el *ALU* [3].

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 debe recuperarla. Solamente si el *ALU* obtiene el contenido de una dirección de esta forma es 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 puede causar inconsistencias si no se es lo suficientemente cuidadoso al escribir [3].

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* termine sus cálculos y entonces el contador de programa puede continuar a la siguiente dirección de forma incremental sin algún cambio [3].

Los periféricos también juegan un papel importante, aunque solo se conectan directamente a la memoria principal. 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 [3].

### 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 [3] para entender la razón de su existencia.

Para empezar necesitamos establecer una conexión entre el usuario y la máquina, que

Código de operación	Mnemotecnia	Definición
0	INP	(INPUT) Guardar dato 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 contador de programa 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*.

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. 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, dónde podremos visualizar los códigos, su equivalente en ensamblador y como afectan al acumulador. Esté 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, dónde 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*. 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 #25 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. Posteriormente 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 origen 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 *JMP*, 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*, la operación que se realiza en esa dirección es una suma para dejar 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 *020*. 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 este 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 desplazemos en el sentido contrario, si este 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 persistir los datos en #31 tenemos la instrucción para guardar la información del acumulador(*STO*) en la dirección de memoria número #13. 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 en los capítulos anteriores, CARDIAC es sumamente útil para explicar aspectos importantes de la computación y el cómo 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 [3] es necesaria. Pero ello implica diferentes retos, retos que se abordarán en este capítulo.

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 **máquina 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 dónde se puede seleccionar la máquina virtual que queremos probar. Como se podrá notar, he agregado la *E*(de electrónico) que vemos tanto últimamente, como sufijo de las tres máquinas virtuales para resaltar su aspecto

electrónico distante del que tenía en los años 60.



Figura 3.1: Inicio para selección de máquinas 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 máquina 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 máquinas 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 estén 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 la CPU a la izquierda, la memoria a la derecha con una descripción de cómo 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.

#### Configuración de la máquina virtual

En la figura 3.3 podemos ver el esqueleto de lo que será nuestra máquina, que aún se encuentra apagada, porque al entrar lo que nos encontramos es una máquina 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 página de inicio si así lo deseamos.

Continuando en el centro tenemos tres botones, el que ya vimos para encenderla, otro para pausarla, algo poco común en una máquina de verdad, pero bastante usual en los software de máquinas virtuales. El uso de este botón de pausa nos permite detener el funcionamiento completo sin afectar los procesos internos, analizar su estado y después continuar como si nada hubiera pasado; por último, el tercer botón es para reiniciarla.

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

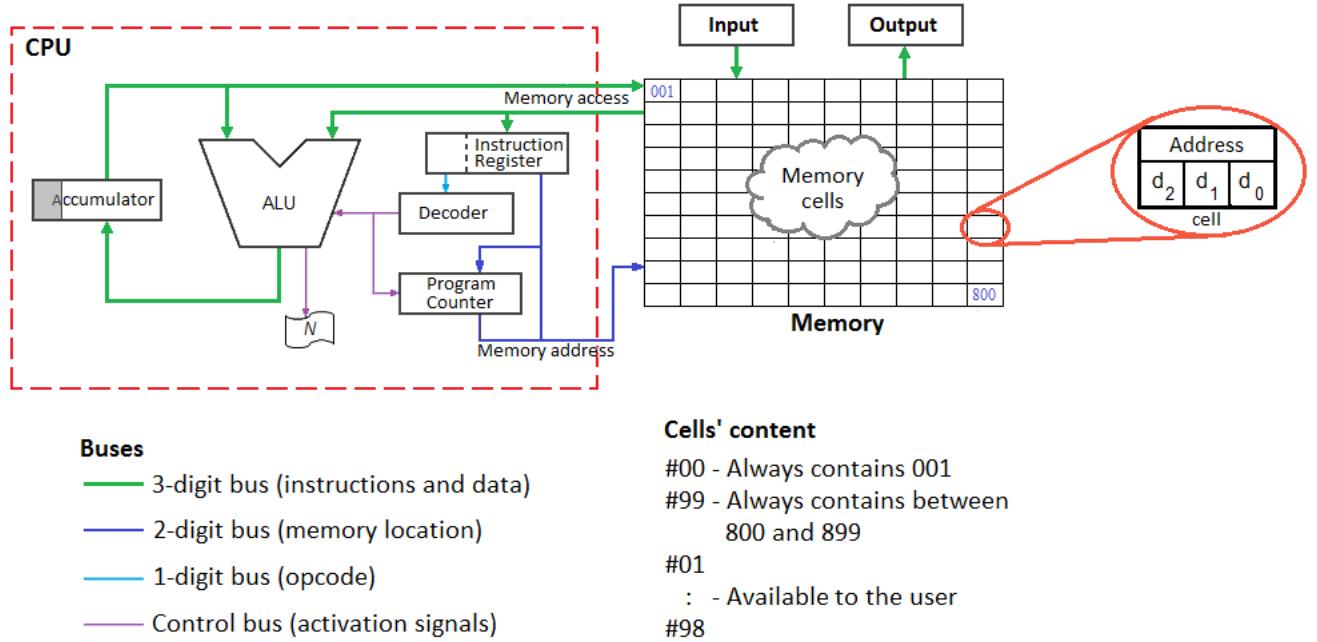


Figura 3.2: Arquitectura de CARDIAC por Jorge Vasconcelos, 2018

cantidad de celdas con las que la máquina funcionara, es decir, la memoria disponible, y la segunda es para elegir la velocidad de la máquina. De esa forma podemos decidir cuanto va a tardar cada ciclo en ser completado con el fin de observar su comportamiento.

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 está *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 máquina, 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 solo sería la adaptación, mismo caso que para 10,000 celdas. Al elegir la cantidad de memoria y luego encender la máquina, esta se configura en su arquitectura para trabajar con esa cantidad de memoria y recibir instrucciones con direcciones más grandes.

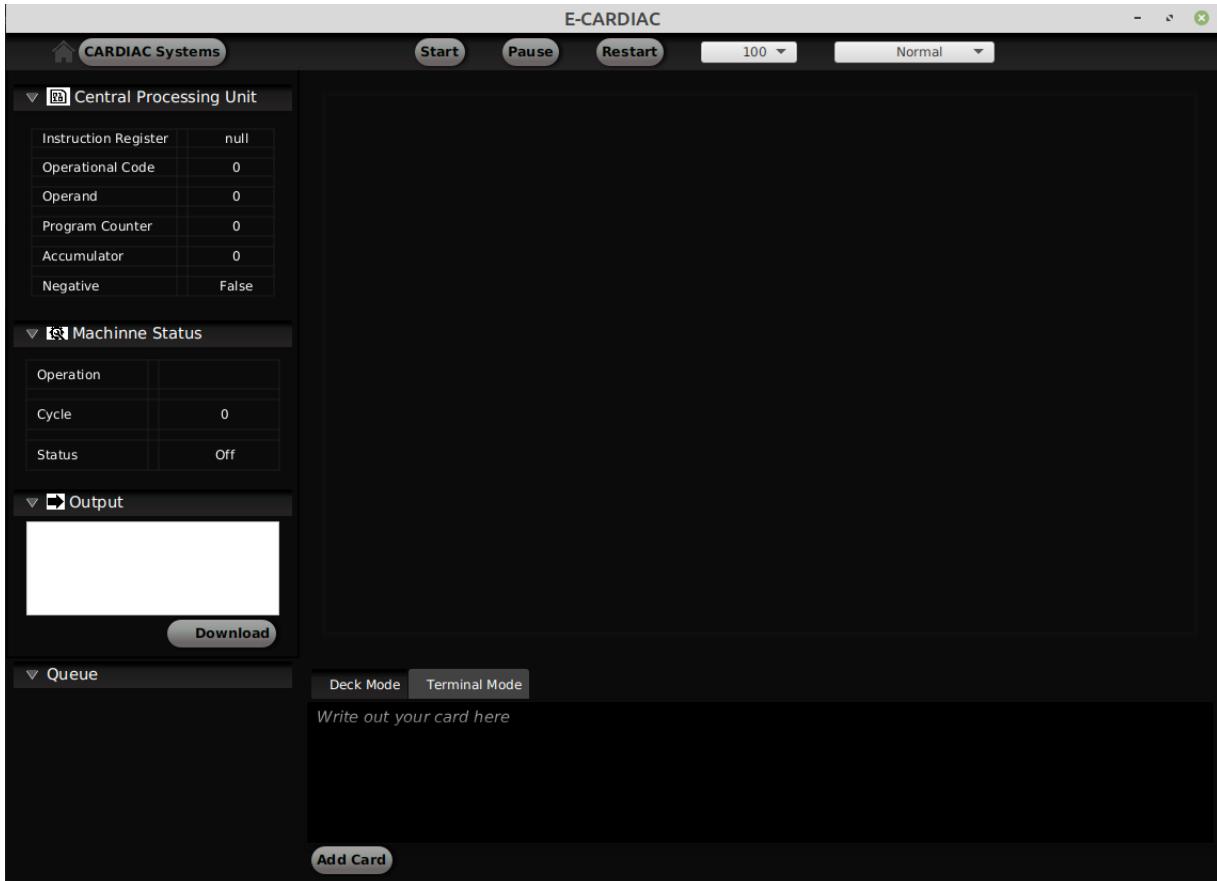


Figura 3.3: Pantalla de inicio de E-CARDIAC



Figura 3.4: Listas desplegables de E-CARDIAC

## Unidad central de procesamiento

Continuando con los elementos que tiene la máquina virtual, podemos ver en la parte izquierda de la figura 3.3 cuatro secciones; los componentes principales de la máquina en la *Central Processing Unit(CPU)*, el estado de la máquina en la parte del *machinne status*(*estado de la máquina*), un apartado para las salidas en el *output*, y un apartado en blanco(*queue*), que servirá como cola de espera para la ejecución de las instrucciones.

En la primer sección, que es la *CPU*, podemos notar que está el registro de instruc-

ciones(*instruction register*), que contendrá la instrucción que viaja a través del bus verde desde la memoria a la *CPU*. Después el código de operación(*Operational Code*) y el operando(*Operand*), que son los resultados que presenta el decodificador, en la figura 3.2), después de que una instrucción pasa por él.

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 máquina, es decir que el contenido de tal dirección que es “apuntada” por el contador será transmitido a la *CPU*.

Más abajo se encuentra el acumulador(*Acumulator*) 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*.

## Estado de la máquina y dispositivo de salida

En la sección *Machinne Status*, tenemos elementos externos a la arquitectura de la computadora. Estos nos definen el estado de la máquina en cada momento, en cada ciclo que pasa nos indica directamente si la máquina 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 CARDIAC realice operaciones indebidas, como pasar un negativo como instrucción, para lo cual no tiene respuesta y lo único que hará es detenerse.

También en esta sección está un apartado que indica la operación que estará ejecutando la *CPU*, *Operation*, y el ciclo en el que se encuentra con *Cycle*.

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. 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.

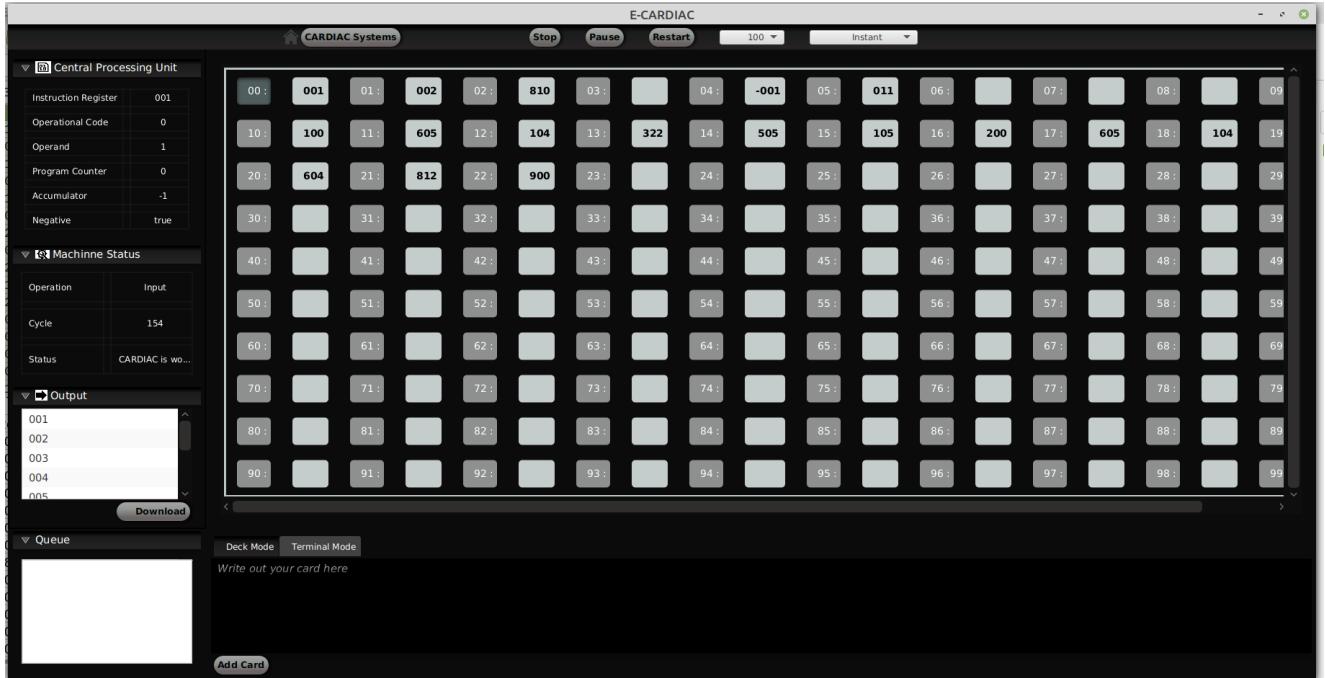


Figura 3.5: E-CARDIAC Muestra de Output

### Entrada de datos e instrucciones

Toda la parte 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. Estas instrucciones, el usuario podrá añadirlas a la cola desde la pestaña *Deck Mode*(modo tarjeta), 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*(modo terminal), del cual 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 se espera una entrada en una celda, en caso contrario no se permitirá agregar datos de esa manera. Debido a que este modo transfiere directo los datos o instrucciones a la memoria en la posición solicitada, a diferencia del modo de tarjetas, que siempre primero las manda a la cola de espera, que las hará pasar posteriormente, cuando se solicite, a la memoria principal.

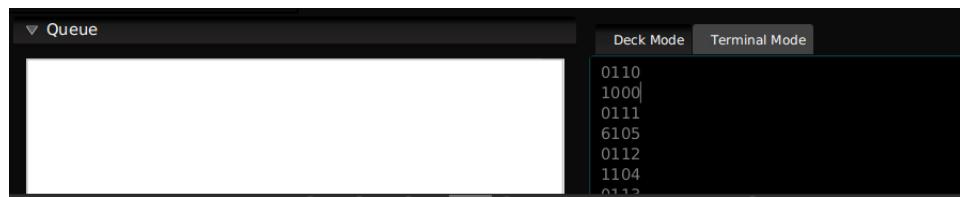


Figura 3.6: E-CARDIAC Carga masiva por tarjetas

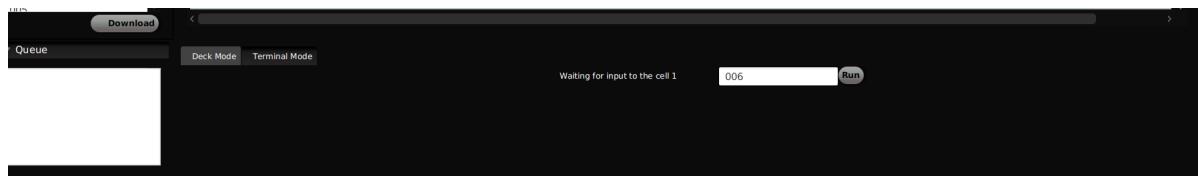


Figura 3.7: E-CARDIAC Carga individual de instrucciones

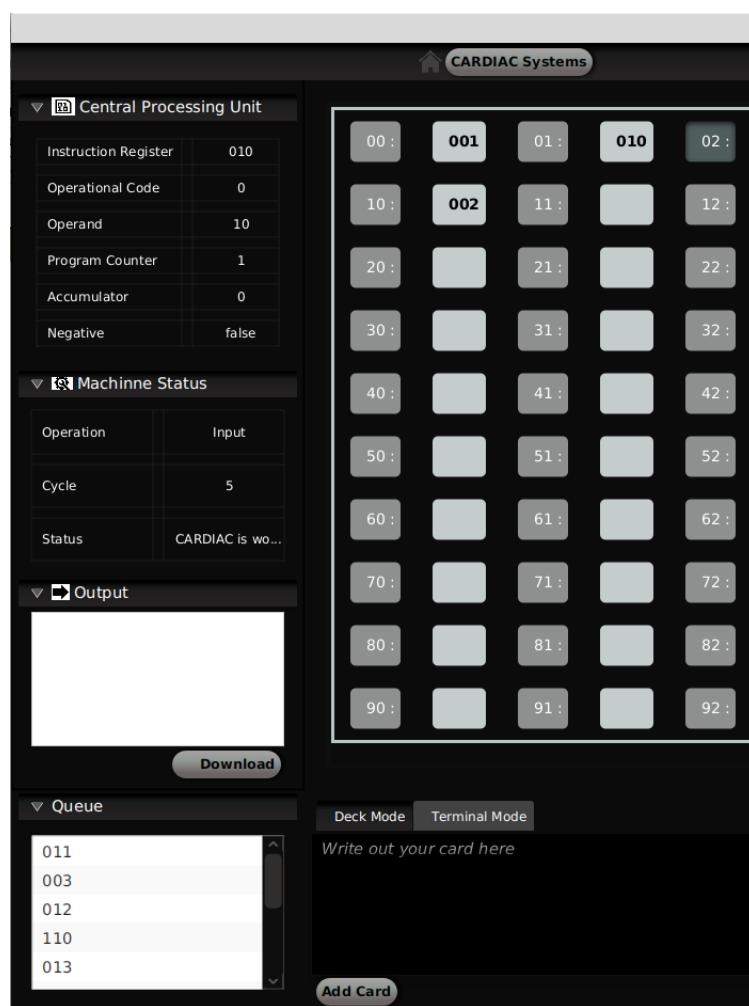


Figura 3.8: E-CARDIAC Cola de instrucciones/datos

## Encendiendo E-CARDIAC

En varias de las imágenes mencionadas en párrafos anteriores se mostró cómo se ve la memoria principal, pero veamos ahora la figura 3.9 para ver la máquina 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. Para diferenciar la dirección de la celda de la cual el contenido está siendo transmitido a la *CPU* se tiene el color azul marino, en el caso del ejemplo nos indica que se está leyendo la instrucción en la dirección #00.

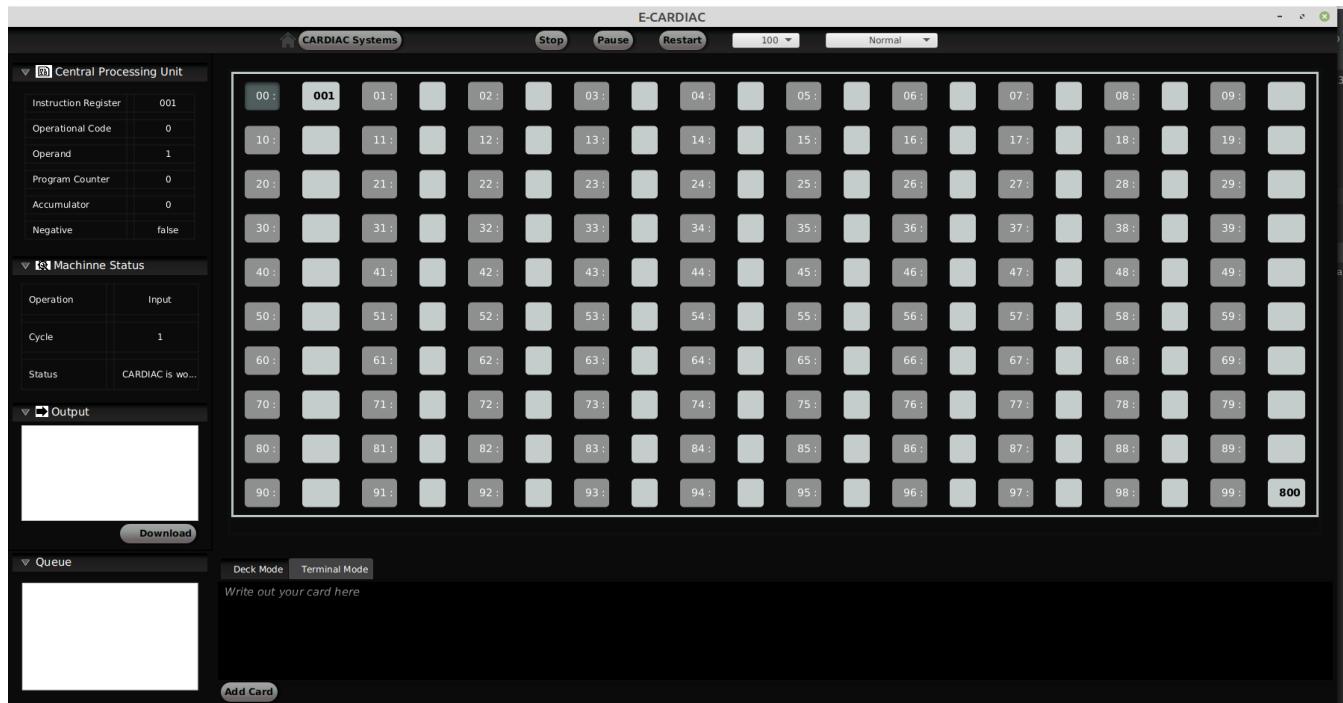


Figura 3.9: E-CARDIAC después de iniciar sus funciones

De esta forma terminamos el recorrido por la máquina virtual de **E-CARDIAC**, que nos permitirá experimentar con más soltura la programación en lenguaje ensamblador. La siguiente evolución del modelo requerirá de un aumento en la memoria de CARDIAC, que es muy fácil de aplicar sobre esta máquina virtual como ya lo hemos visto, y que podremos estudiar con más profundidad en la siguiente sección.

## 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 cómo 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 máquina, 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. En [3, p. 42] nos presentan un sistema de carga de programas para el modelo original, un cargador(*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, pero esto aún nos deja con la tarea de decidir qué 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 qué la otra característica que define a un sistema operativo es ser una capa de abstracción entre la máquina y otros programas. En este caso, el sistema operativo que se diseñará no tiene la intención de ser una capa de abstracción tan marcada, tendrá algunos aspectos que pueden ser considerados una capa de abstracción entre los programas y la máquina, 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 es 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 tratara. 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 máquina 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, así un número completo de cuatro dígitos puede representar tanto el número en sí mismo, como un código de operación acompañado de tres operandos.

Por lo tanto, 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 solo las direcciones, mientras que los que están en color azul solo 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. La misma situación la tiene el valor del final de la memoria, que solo

podrá contener valores que inicien 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 tomó una instrucción de salto que fue ejecutada.

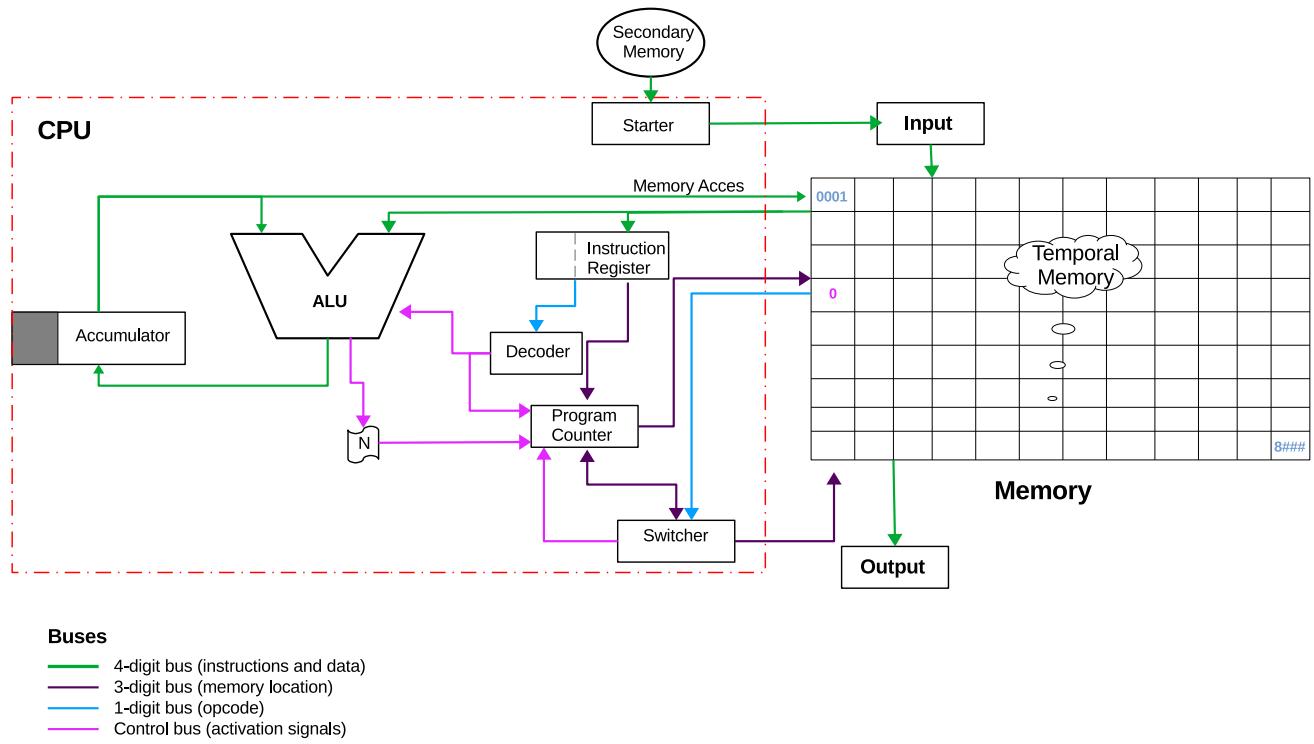


Figura 3.10: Diagrama de Arquitectura de E-CARDIAC C

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. Analicemos primero al que está conectado con el dispositivo de entrada(*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 qué solo sería de un almacenamiento muy puntual

para el sistema operativo. Siendo una especie de tarjeta con un programa específico, que está conectada a través del iniciador para que apenas se inicie la máquina, lo primero que haga es empezar a agregar las instrucciones guardadas en la memoria secundaria a la pila de ejecución a través del dispositivo de entrada.

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 como si fuera parte de la memoria ROM.

El otro integrante de este nuevo modelo es el que tiene por nombre **switcher**(comutador), 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, y poder recuperarlo en un momento particular.

Como se puede observar en la imagen, tiene 4 diferentes conexiones, siendo uno de los que más tiene. Su función radica en que cada N ciclos 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 inicial de la misma arquitectura. 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 cual puede disponer de ellos sin límite al tener todos los privilegios posibles. Esta conexión entre el comutador y la memoria principal se hace con un bus que azul, pues solo requiere de un dígito.

Otra conexión que tiene es una bidireccional al contador de programa, que le permite contar los ciclos que han sucedido desde que el SOM cedió el control e indicarle al contador de programa cuando debe regresar los recursos al SOM, y a qué dirección tiene que saltar cuando se trata de regresar el control de los recursos al SOM. El contador del *comutador* sigue al contador de programa, pero se reinicia cada que la bandera de saltos cambia de valor. Para todo esto se requiere un bus de color morado, para transmitir las direcciones y datos, y uno color rosa para mandar la señal al contador de programa de que debe interrumpir su

proceso natural.

Una razón adicional para esta conexión bidireccional es que el *conmutador* recibe del contador programa la última dirección apuntada por el proceso del usuario, antes de que fuera ejecutada. Esto sucede debido a que para ese momento si se llegaron a los N ciclos se da la instrucción de regresar los recursos al SOM, y en ese momento la dirección es transmitida al *conmutador*, para que este, por medio de su conexión a la memoria principal(por el bus morado), la coloque en una dirección especial para que el SOM pueda recuperarla. Para que después, cuando regrese los recursos a ese proceso, pueda continuar justo en la instrucción que ya no pudo ejecutar.

El sistema operativo mínimo en su mismo proceso lanza los procesos del usuario, y por ende realiza los saltos necesarios para ceder los recursos. Pero antes de lanzarlos, modifica el valor de la dirección #003 a 1, para que el *conmutador* pueda regresar el control al SOM después de N ciclos aproximadamente.<Aclarando la parte de “N ciclos aproximadamente”, es por qué 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 antes de regresar los recursos.

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 máquina muchas veces es consecuencia de las posibilidades que el hardware le provee. Pero también el lenguaje define ciertas necesidades del hardware, por lo que en la construcción de una computadora se necesita pensar en todos los elementos que van a interactuar juntos.

Para E-CARDIAC C 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*. Esta está diseñada ahora para funcionar con un sistema operativo,

pues por sí sola no tendrá las capacidades completas, que sí tendrá con un SOM.

En la configuración inicial de la computadora se definirá la dirección establecida como *área de borrado* para el SOM, área que ocupa la instrucción *halt* para finalizar procesos. Con esto podemos considerar que nuestro SOM es, en cierto punto, una capa de abstracción entre el código máquina puro, y las instrucciones que usa el usuario en sus procesos.

Código de operación	Mnemotecnia	Definición
0	INP	Guardar datos 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 d1 veces a la izquierda el número y d0 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*.

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* en la derecha, tenemos que el dígito *d3* tendrá el código de operación, como en todas las demás instrucciones. Pero para esta solo importarán los dígitos *d1* y *d0*, puesto que el *d2* no representará nada en la instrucción, siendo *d1* la cantidad de lugares que se desplaza el número a la izquierda, y *d0* la cantidad de lugares que se desplaza a la derecha.

El último cambio a destacar es que el valor del contador de programa, que se guardaba en #99 cuando se hacía un salto con la instrucción de salto, ahora será guardado en la

dirección #999, que es la última en esta arquitectura. Pero para el resto de instrucciones, el comportamiento será el mismo, solo que los tres dígitos de la derecha representan la dirección, lo cual es bastante lógico dado que las direcciones ya no ocupan dos dígitos, sino tres.

Con esto tenemos el diseño completo de la máquina, 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 computadora. Aunque por supuesto, y como se dijo, ya se había pensado en que características necesitaba la computadora para poder implementar un SOM.

### 3.2.4. Sistema Operativo Mínimo C: Aspectos Generales

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 lo tanto, detrás del desarrollo de este sistema operativo mínimo, llamado *SOMC*, está una lista de tareas que debe ser capaz de realizar para poder ejecutar concurrentemente diferentes procesos, que es la finalidad principal de su construcción.

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 está 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 cual ese nombre, otro es la **zona de procesos**, para almacenar el contexto de los procesos que se estén ejecutando en la máquina. Por separado está una **zona de variables**, que serán variables

o constantes de uso recurrente que el sistema operativo mínimo pueda guardar o consultar. Por último, un segmento llamado **preámbulo**, que contiene 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 se pueden cambiar las direcciones de inicio de cada segmento una vez que ya se tenga todo escrito sin afectar la estructura del código.

Adicionalmente, el núcleo del SOM se fraccionó dependiendo de las tareas que realiza cada parte del SOM, separándolas en: añadir proceso, actualizar proceso, borrar proceso, ejecutar proceso, y ejecución del cargador(*bootloader*). Para poder seguir todos estos conceptos con más claridad, diseñé un diagrama de flujo, que se puede ver en la imagen 3.11 , en el cual cada fracción del núcleo 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 entender las conexiones entre cada zona, esta 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, donde 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, es una descripción corta acerca de esas direcciones, y solo se coloca si es necesaria. Por ejemplo, en las instrucciones coloreadas de color rosa hay un nombre clave que indica que es una bandera. A su derecha tenemos la dirección de memoria, después la instrucción en lenguaje máquina, 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 fracción 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 cómo se usan, si nos fijamos en la fila dónde está la dirección #e10 veremos la instrucción *l(e(0))*, que indica que se cargue el contenido de la dirección #e0 en el acumulador. En la implementación la variable de

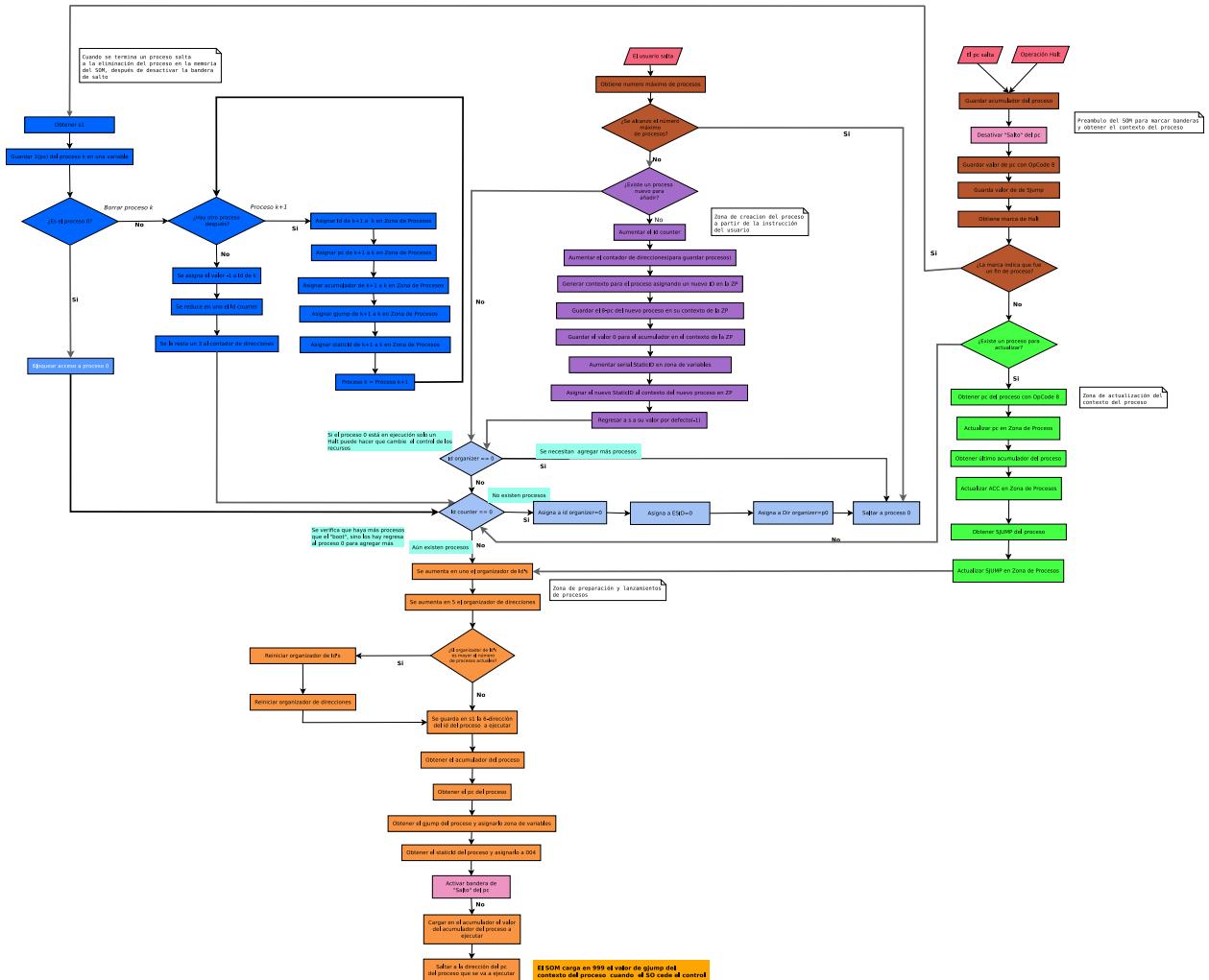


Figura 3.11: Diagrama de flujo de SOMC

dirección sería sustituida por una dirección real de la máquina.

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 qué 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* que se encuentra en el interior de la instrucción. Examinándola nos dice que se va a sumar al contenido del acumulador el contenido de la dirección #c13, y tanto la descripción como el nombre clave nos dan más información para entender mejor la instrucción.

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

Figura 3.12: SOMC:Preámbulo

## Inicio de operaciones para la máquina

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. Para esto, las instrucciones que el *iniciador* mande a la cola de ejecución deben estar en forma de tarjeta, es decir, una instrucción o dato seguida de otra en forma de lista. Si usamos el método de arranque(*booteo*) que nos provee [3] la tarjeta debe empezar siempre con las siguientes instrucciones:

0002

0008

Con estas instrucciones de inicio basta para que las siguientes solo tengan que ser pares de instrucciones, dónde la primera indica el destino de la segunda. Abajo vemos cuáles serían las siguientes instrucciones:

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 estático o *id* *estático* del proceso que se está ejecutando en ese mismo momento, y no se ha escogido el 0 solo por ser un número por defecto, es porque justamente se está ejecutando el **proceso 0**.

Debido a que para el SOM, este sistema de arranque, una vez que ha cargado las primeras instrucciones del SOM, 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 del sistema operativo mínimo, la tarjeta usada para cargar el sistema se encuentra completa en el apéndice E para su consulta, pero básicamente son pares de instrucciones para colocar cada segmento del sistema en su lugar.

## Encendiendo la máquina

En la figura 3.13 podemos observar como será la máquina virtual para E-CARDIAC C, 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 máquina,

sino que se encuentra el contenido de la memoria secundaria, 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. Si no que al ser está una memoria no volátil se queda ahí estática la información, no cuenta como memoria ROM porque 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 máquina.

Como se notará en las imágenes, otra parte que cambio fue la parte del estatus de la máquina, que ahora tiene un campo llamado *Starter*, que en la máquina apagada tiene un valor de “Waiting”(esperando), y en la encendida de “Booted”(arrancada). Esto es porque al encender la computadora, el iniciador(*starter*) en automático coloca la tarjeta de la memoria secundaria en la cola, solo es cuestión de dejar seguir la ejecución para que la carga se complete.

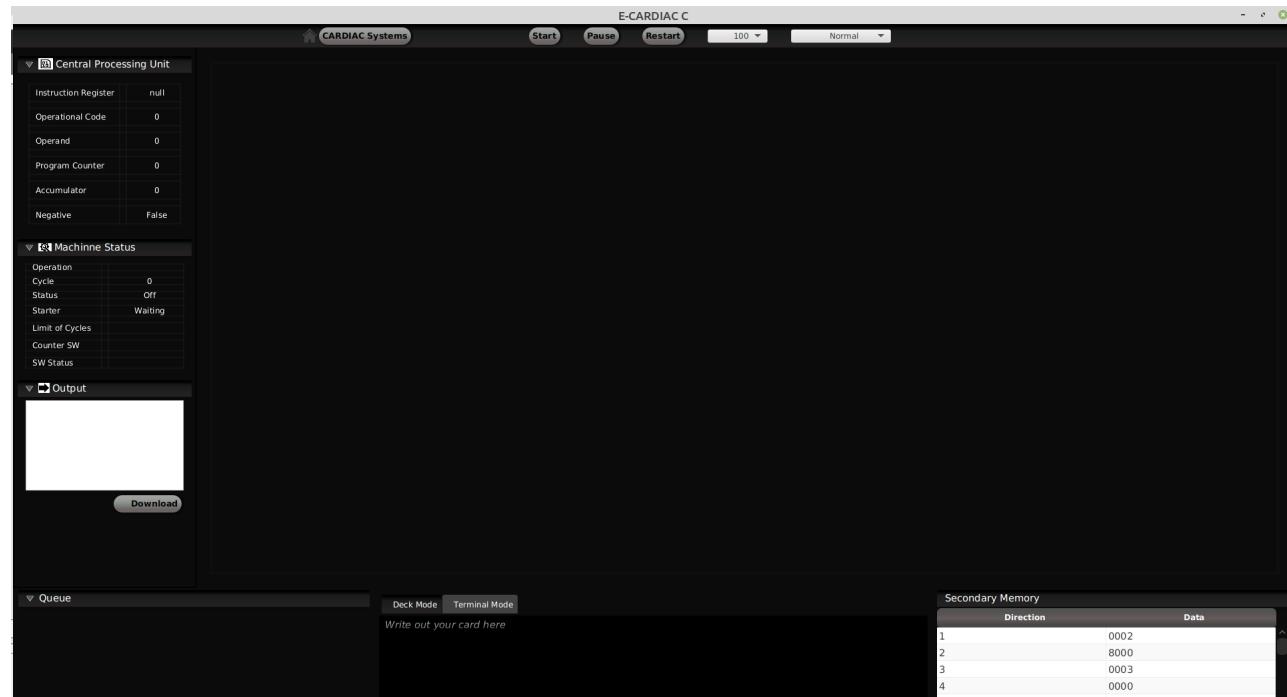


Figura 3.13: E-CARDIAC C Apagada

Aparte del *Starter*, tenemos otros tres campos que nos ayudaran a ver quién tiene el control de los recursos y por cuantos ciclos. El campo *Limit of Cycles*(límite de ciclos) contiene

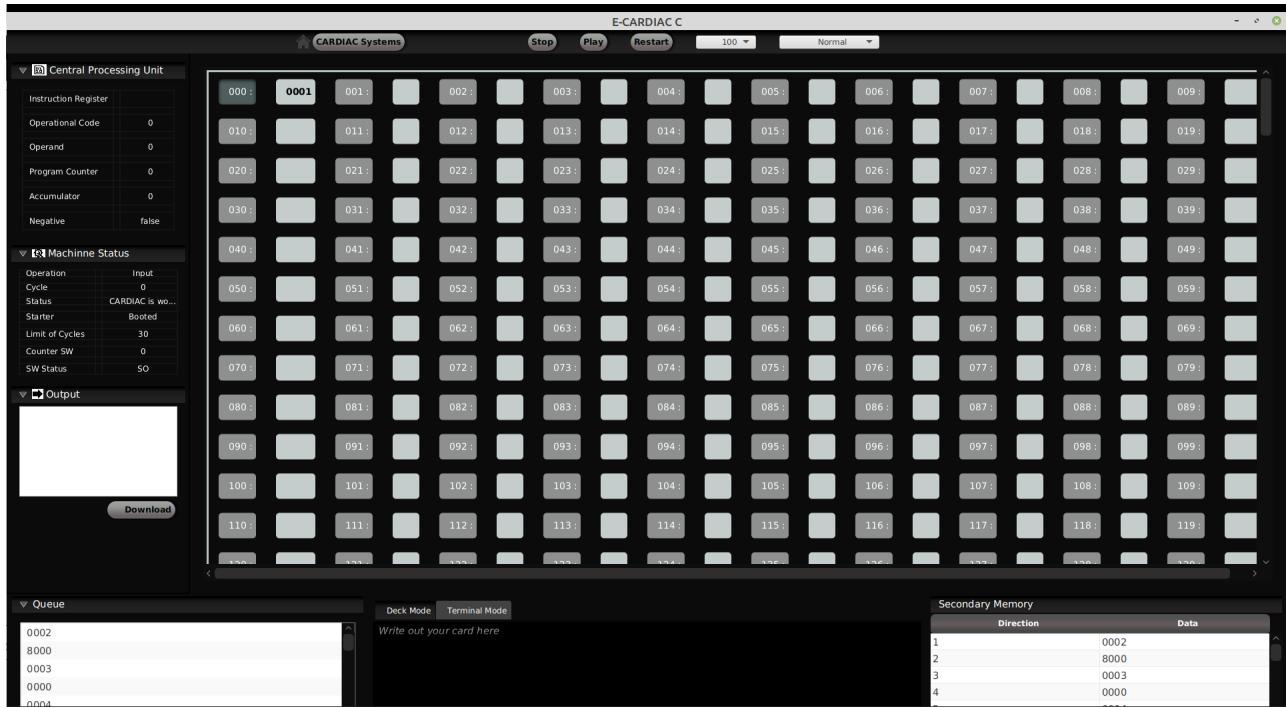


Figura 3.14: E-CARDIAC C durante el arranque

el valor de cuantos ciclos máximos tendrá un proceso del usuario antes que obligadamente tenga que ceder los recursos de nuevo al SOM, la implementación que se muestra tiene un máximo de 30 ciclos para cada proceso del usuario.

El *Counter SW* es el contador del conmutador, tiene un contador de ciclos, pero a diferencia de *Cycle*, este se reinicia cada que el control de los recursos cambia de propietario. Si está ejecutándose un proceso del usuario, cuando este contador alcance el límite de ciclos, el *conmutador* saltará de inmediato al preámbulo del SOM para que este tome control de los recursos. Si es al revés el contador solo 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 conmutador no podrá hacer ningún salto.

Para saber quién tiene el control también podemos ver el último campo, *SW Status*, que es el estatus del conmutador y nos indicará quién tiene el control de los recursos, en las imágenes lo tiene el SOM y por eso 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, porque en esas primeras instrucciones se cambia la bandera de valor, pero para el contador del conmutador 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, y que ahora es “basura”.

También podemos ver que las banderas están cargadas correctamente, y algo que nos puede llamar la atención es que el estatus del conmutador marca que el control de los recursos lo tiene el sistema operativo, y esto no es un error, porque 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.

Esto sucede porque 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 solo 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.

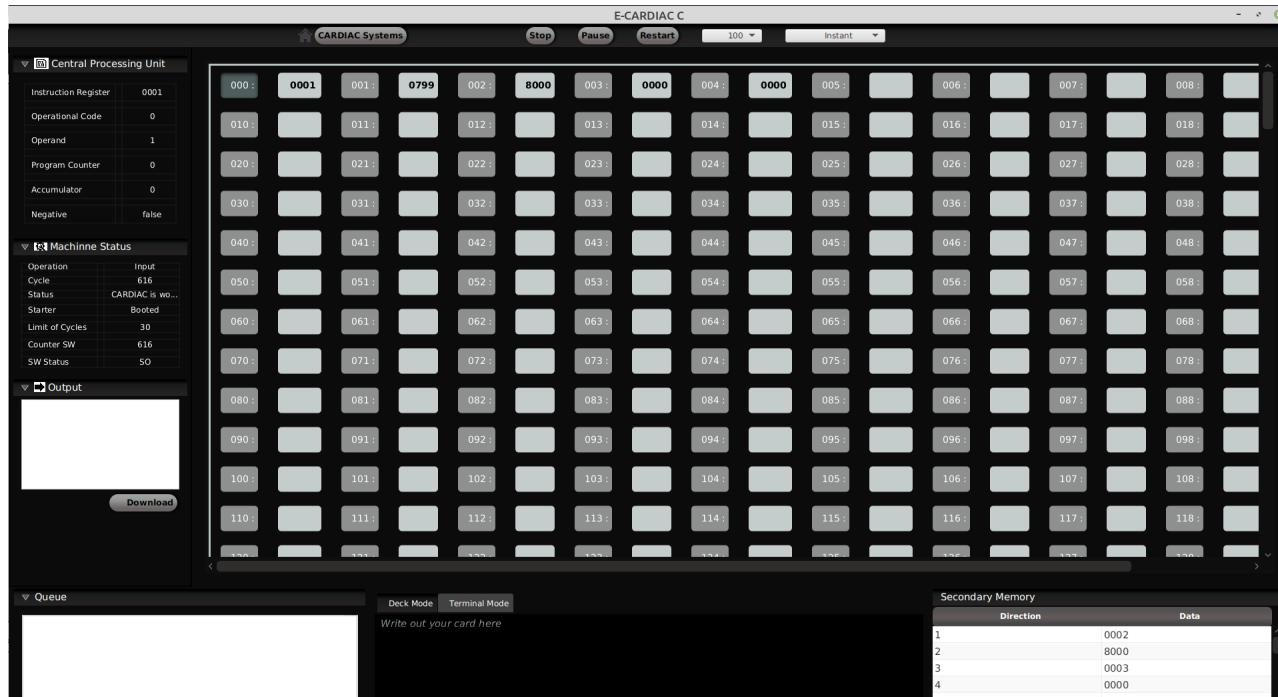


Figura 3.15: E-CARDIAC C Sistema operativo mínimo cargado

## ¿Cómo toma control el sistema operativo mínimo?

Como pudimos ver en las imágenes anteriores, a pesar de que para el *comutador* el control de los recursos este actualmente del lado del SOM, la realidad es el usuario es quien los está usando y controlando limitadamente, hasta que los ceda al mandar a ejecutar los procesos que cargo.

En la figura 3.11 se nos presenta un diagrama de flujo(muy general) en el que podemos alcanzar a ver tres recuadros rosas, que tienen una forma más inclinada que la de un rectángulo normal, esto es porque son entradas de datos por parte del usuario, y que en el diagrama usé para exemplificar las tres formas en que el SOM toma de nuevo el control de los recursos. La primera es cuando se va a añadir un nuevo proceso(el usuario salta), la segunda cuando el contador de programa por medio del comutador salta en automático, y la tercera si se da la indicación de borrado de un proceso(operación *Halt*), es decir, si se lee la instrucción #9000.

En la figura 3.16 apreciamos estas 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 porque siempre, antes de llegar al núcleo del SOMC se debe pasar por el preámbulo, que se encarga de actividades previas, para que puedan llevarse a cabo con normalidad y seguridad las actividades del núcleo.

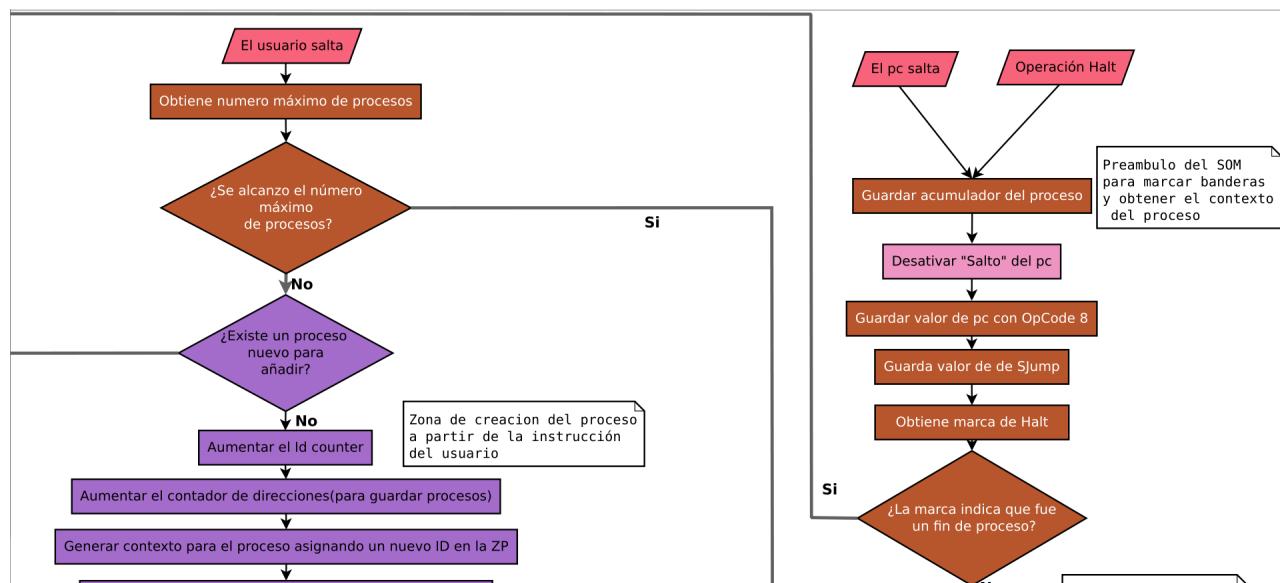


Figura 3.16: Formas de entrar al sistema operativo mínimo C

### 3.2.5. Sistema Operativo Mínimo C: Un nuevo proceso en la pila

Veamos en orden como el SOM interactuara con los programas que añadamos y queramos ejecutar en E-CARDIAC C. 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.

#### Añadir proceso: tareas del usuario

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 otra que inicia después de la dirección #300. Veamos la tarjeta de la versión 1, que usa direcciones después de la #100, en la tabla 3.2. En ella observamos un fragmento de la tarjeta que el usuario colocará en la máquina virtual para añadir esté programa a la zona de procesos, vemos que la primer instrucción de la tarjeta es para indicar que se cargue en la dirección #110 la primer instrucción de *pintador v1*. Siguiendo el mismo concepto de pares de instrucciones en toda la tarjeta, dónde siempre va primero la instrucción que indica dónde se cargara la segunda.

Pero si notamos al final hay una sola instrucción que se sale de esta regla, la instrucción *JMP 985*, que 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 la información del programa, que ya cargo en memoria, sea añadida a la zona de procesos para que se convierta, como tal, en un proceso. Además de ser la única dirección, que es parte del sistema operativo mínimo, a la que el usuario puede saltar.

Para ver la tarjeta completa podemos observar la tabla 3.3, 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 máquina	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 para imprimir números del 1 al 10

### Añadir proceso: tareas del Preámbulo

Veamos ahora lo que el sistema tiene que hacer, y lo primero es saber a qué 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 se asocia a esta y a las siguientes direcciones a la fracción *añadir un proceso*. Es en este punto que ya vemos a la aparición de las *variables del sistema*.

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 este 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, para conseguirlo en *c16* se coloca el número máximo que queremos menos uno.

Volviendo al preámbulo entenderemos por qué 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 saltará directamente a la fracción

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

añadir proceso. 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.

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

Figura 3.17: SOMC : Variables del sistema

## Fracción Añadir proceso: Validaciones previas

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. El salto lo hace a la dirección *s66*, y que como recordaremos la letra “s” en las variables es para representar direcciones del núcleo del sistema operativo.

Pero veamos primero el 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 v1*. Este par contiene la instrucción *0800* que significa cargar en la primer posición del núcleo del sistema, que es **la única dirección del SOM 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 programa que queremos cargar en la zona de procesos.

En la imagen 3.19 podemos ver que *s0*, que en nuestra implementación sería *800*, tiene un valor por defecto de *-0001*. Debido a que con este valor se indica que no hay un proceso a cargar, y cada que se termina de cargar un nuevo proceso, el contenido de esta dirección regresa a su valor por defecto.

En caso de que el valor de *s0* fuese negativo, el flujo sería más corto (figura 3.18), pues irá a verificar el valor del **Id organizer** u organizador de identificadores, que tiene el identificador de procesos que se está ejecutando. Si el valor es 0 es porque la orden de añadir un nuevo proceso fue lanzada desde el proceso 0, y nunca se cambió el valor de *s0*, entonces regresará al proceso 0 con el recorrido más corto.

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

Figura 3.19: SOMC : Contenidos generales del núcleo

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

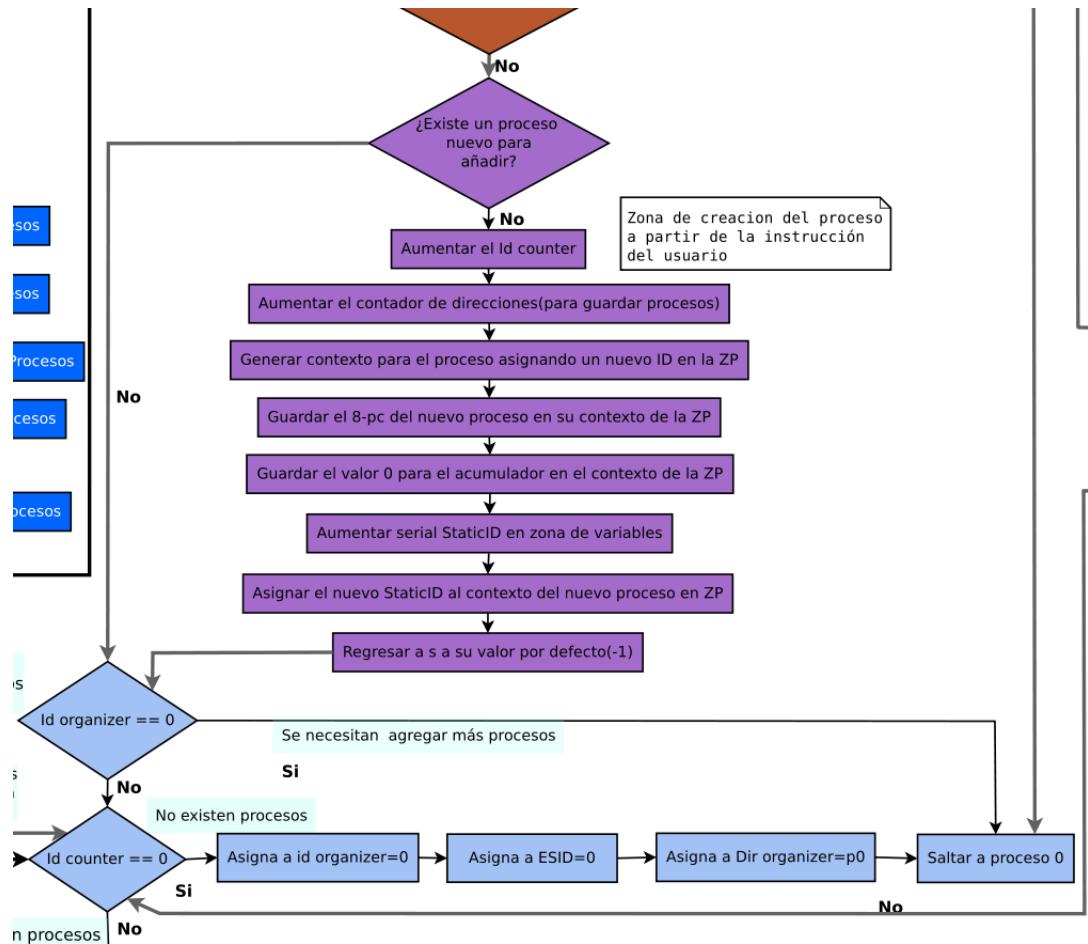


Figura 3.18: Diagrama de segmento para añadir un proceso

por revisar si la orden fue lanzada desde el proceso 0. Esta verificación es importante, en el resultado más sencillo es que sí se haya lanzado desde ese proceso, pero en el caso de que no sea así, la entrada a la fracción azul da un resultado es muy diferente. En el caso de que no se haya lanzado la orden desde el proceso 0 significa que un proceso del usuario dio la orden de agregar un proceso, por lo tanto, si fue un proceso el que cedió los recursos, el sistema operativo tiene que elegir al siguiente proceso a ejecutar. Por lo tanto, si el organizador tiene un valor diferente de 0 el flujo se enlazará con otra fracción del núcleo del sistema, que es el lanzamiento de procesos, a través de la fracción especial de gestión del proceso 0(fracción azul clara). Esta fracción la veremos más adelante, su función es “elegir” que proceso será el siguiente en ejecutarse.

## Añadir proceso: Contexto del proceso

Regresando a la fracción 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, la tarea 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. Por otra parte,  $c_5$ , el “contador de direcciones”, esté contador, lo que tiene como valor por defecto es  $p_0$ , como recordaremos toda variable que inicie con  $p$  hace referencia a la zona de procesos. Precisamente  $p_0$  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
<i>Aumenta el Id counter M[c4]++</i>	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
	s68	1(c4)	LDA c4	Se obtiene el valor del Id counter
	s69	2000	ADD 000	
	s70	6(c4)	STO c4	
<i>Aumenta el Dir counter M[c5]+=3</i>	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	
<i>Previa de ID</i>	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)
<i>Previa de pc</i>	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
<i>Previa de acc</i>	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
<i>Guardar Nuevo Static ID para el proceso</i>	s80	2(c12)	ADD C12	Añade 2 para obtener 6(px)+4(Static ID)
	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
<i>Id del nuevo proceso</i>	s85	6(px)+4	STO (px)+4	Guardar el nuevo serial Static ID para el 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
<i>pc nuevo</i>	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
<i>acc del nuevo proceso</i>	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
<i>Poner valor default en s</i>	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 cómo está compuesta la zona de procesos, llamaremos “contexto del proceso” a todas las variables asociadas al proceso y que se almacenarán en

la zona de procesos bajo un mismo identificador. Para nuestros contextos requeriremos de cinco variables, si vemos  $p5$  y  $p10$  tienen valores de  $-0001$ , esto es porque son los inicios de cada contexto. Un 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. Por esa razón 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, porque ahora el último proceso es el que se encontrará en el contexto de  $p5$ .

Para el resto de variables que están en el contexto del proceso tenemos primero a las dos más evidentes: su último contador de programa, guardado en  $gpc$ , y su último acumulador, guardado en  $gacc$ . Adicionalmente, tenemos otras variables que son muy importantes para el contexto, pero que quizás no son tan evidentes, como el  $gjump$ . Esta variable guarda el último valor que tuvo la última dirección de la máquina (#999 en la implementación actual), debido a 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. Una posible falla sería tener un proceso A que dejó un  $8819$  en la última dirección, luego el proceso B dejó un  $8514$ , y cuando vuelva el control al proceso A intente usar el valor que hay en #999 para ir a la dirección #819 y termine en la #514.

Por último, para identificar a los procesos, tenemos al ID principal que tienen y con el que inicia el contexto, el cual tiene por nombre completo *ID counter*/*ID contador* o simplemente ID principal, pues va en orden ascendente y sirve también para contar cuantos procesos existen. Pero además, tenemos el identificador estático, guardado en *Static Id*, que mantiene un identificador inamovible para cada proceso, más adelante en la sección de borrado veremos la importancia de este identificador estático. Pero por lo pronto y desde este punto, siempre haremos la distinción entre cada identificador, pues sus funciones son distintas.

### **Fracción Añadir proceso: Actualizando información del proceso**

Regresando a las figuras 3.18, 3.20, y 3.21 podemos tener más claro como suceden estas actualizaciones, primero actualiza las variables del sistema para que este conozca cuantos procesos hay, dónde están, y cuál es el último contexto de la zona de procesos activo, es decir, con un ID contador no negativo. Ahora, desde la dirección  $s74$  hasta la  $s92$  lo que

Zona de Procesos				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>Id Counter</i>	<i>p0</i>	<i>0000</i>		Id del primer programa
<i>gpc</i>	<i>p1</i>	<i>8000</i>		Es el pc del proceso 0 con op-code 8
<i>gacc</i>	<i>p2</i>	<i>0000</i>		Es el acumulador del proceso
<i>gjump</i>	<i>p3</i>	<i>8000</i>		Contiene el valor del que se tenía en la última dirección de memoria antes de saltar
<i>Static ID</i>	<i>p4</i>	<i>0000</i>		Es el ID Static del proceso que no va a cambiar
<i>Id Counter</i>	<i>p5</i>	-001		Es el id del proceso correspondiente a está sección
<i>gpc</i>	<i>p6</i>			Puede estar lleno de basura si no hay proceso en este contexto
<i>gacc</i>	<i>p7</i>			Puede estar lleno de basura si no hay proceso en este contexto
<i>gjump</i>	<i>p8</i>	8000		Por defecto tiene el 8000
<i>Static ID</i>	<i>p9</i>			
<i>Id Counter</i>	<i>p10</i>	-001		
<i>gpc</i>	<i>p11</i>			
<i>gacc</i>	<i>p12</i>			
<i>gjump</i>	<i>p13</i>	8000		
	<i>p14</i>			
	<i>p15</i>			

Figura 3.22: SOMC: Núcleo del sistema operativo

hace es llenar el contexto del proceso. Ya se definió que *p5* es la última dirección de inicio de un contexto de proceso activo, por lo que lo primero es asignar el número 1 al contenido de *p5*, pues el *ID contador* será un contador de identificadores que tendrá valores ascendentes. Si agregó 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 del resto de 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 están así porque 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*) en el acumulador. Resultado que con la instrucción de #s75 guardamos en la dirección #s87, que como vemos en la imagen, tiene un contenido con un código de operación que indica la operación *Store/guardar*, acompañado de un *px*, que en nuestro caso sería el *p5*. Y como en

`#s86` se cargó 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 contador del proceso, un `px+1` el del *gpc* del proceso, y así sucesivamente. Debido a 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 será el que tomará el acumulador cuando el proceso arranque, ese valor por defecto es `0`. La variable que no actualiza es *gjump/p8*, por qué todas las direcciones que contendrán a este tipo de variable contendrán como valor por defecto, el `8000`.

Lo que sigue en el flujo es modificar el 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 contador 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 máquina esté encendida. Para lograr esto, lo primero que se hace es aumentar el valor del contenido de `c15` en una unidad, y después ese valor guardarlo en `#p9`, que es la dirección donde se guarda el ID estático 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. Continúa su flujo a la fracción azul claro para validar si el proceso fue lanzado desde el proceso `0`, y en su caso regresar al proceso `0`, o si hay que tomar el camino largo. Como notarán, en ningún momento se hace cambio de bandera aquí, y es porque desde que el usuario está agregando el programa, hasta que el SOM lo añade a la zona de procesos,

para el *comutador* nunca cambia el dueño de los recursos, es decir, sigue siendo el SOM.

### Práctica: añadiendo un proceso nuevo

Veamos en la práctica utilizando la máquina virtual como es añadir un proceso nuevo en E-CARDIAC C. Estamos en la situación de la máquina iniciada, con el sistema operativo cargado, 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, dónde 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.

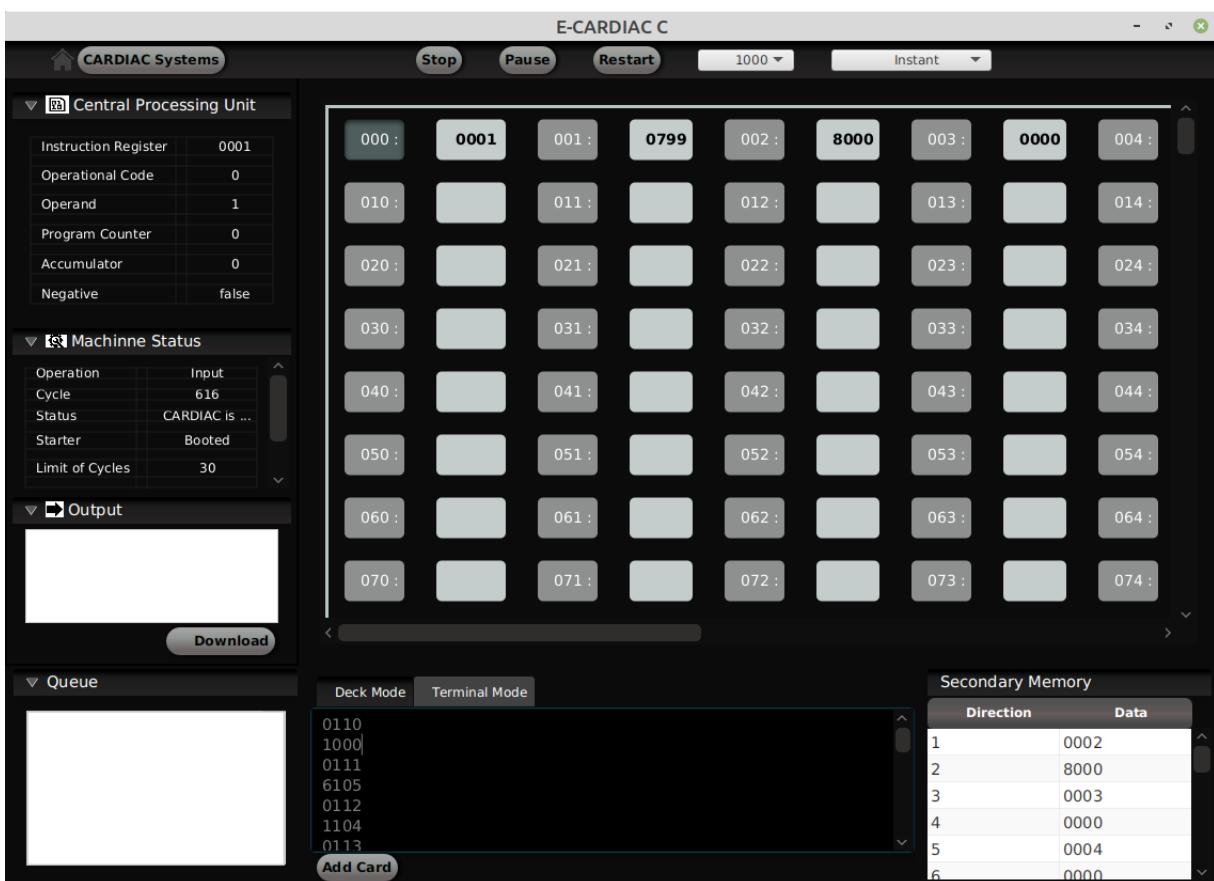


Figura 3.23: E-CARDIAC C Programa en deck

Como es evidente(en 3.15), la máquina no tiene 100 celdas, y es que cuando se inicia la máquina virtual hace una validación sobre si las celdas que indica la celda lista desplegable son suficientes para el tipo de procesos que llevará a cabo. **E-CARDIAC C** solo 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 máquina, y ahora si tiene el valor correcto de 1000 celdas en la lista desplegable. Además, en la imagen 3.24 podemos ver una velocidad diferente en la parte superior derecha, tiene el valor de *Normal*, una velocidad en la que podemos apreciar los cambios, y como va ejecutando cada instrucción. Los cambios de velocidad se pueden aplicar a mitad del proceso dependiendo de las necesidades del usuario, el cambio en la cantidad de celdas solo puede efectuarse al iniciarla.

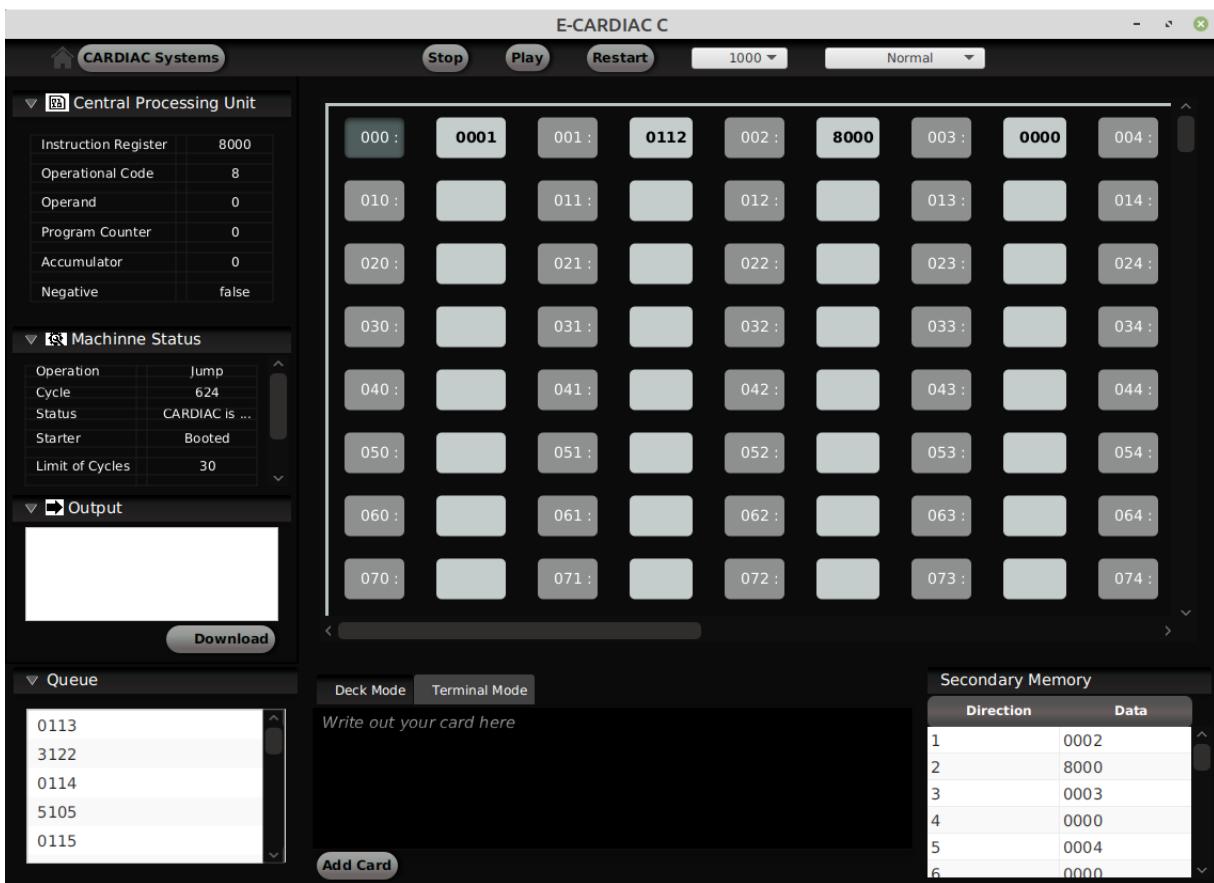


Figura 3.24: E-CARDIAC C Programa en cola

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 sí 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 cuidar el SOM para

que cada que el proceso recupere los recursos no tenga fallas lógicas.

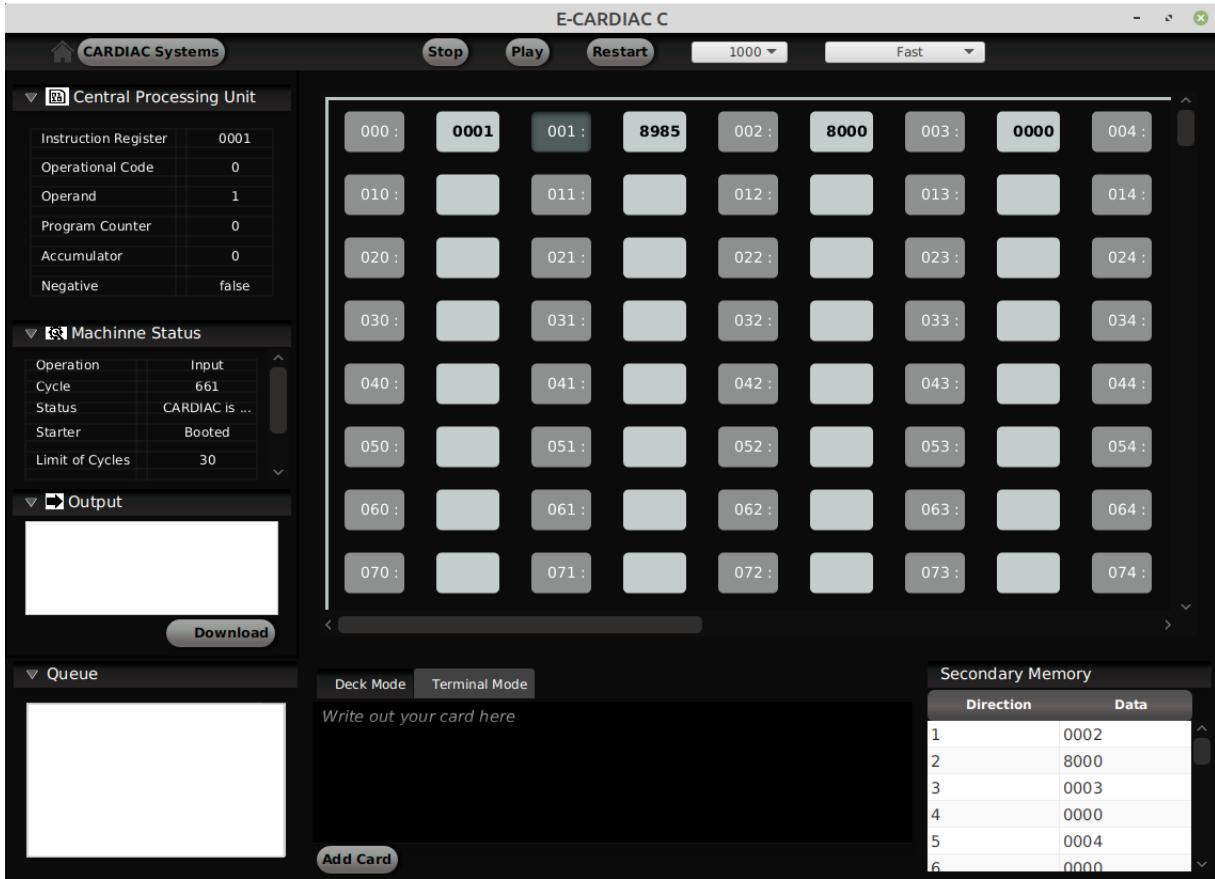


Figura 3.25: E-CARDIAC C Instrucción para añadir proceso

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

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 contador con su correspondiente valor de 1, un *gpc* que indica 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 máquina para la dirección #999, y con un 1 para el ID estático, 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.

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

Para que sea más útil este ejemplo práctico agregaremos otros dos procesos, que hacen lo mismo(son *pintores*), pero que iniciarán 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 mismos valores de arranque en su contexto, salvo las direcciones de inicio y sus identificadores, tanto estáticos como contadores.

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

### 3.2.6. Sistema Operativo Mínimo C: Lanzamiento 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 es para añadir un proceso, otra cuando el contador de programa salta en automático, y la tercera cuando se borra un proceso. Entonces, ¿cómo empezamos la ejecución de los procesos que tenemos cargados?, la respuesta está en la última entrada mencionada, y en características

especiales del proceso 0. 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 proceso, para detener al proceso 0. Lo que hace después es saltar al preámbulo, por lo que hay que ver la figura 3.12 y la 3.29, dónde podemos ver en más detalle las instrucciones que ejecuta.

### Acciones del usuario y del preámbulo para lanzar un proceso

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 *conmutador*. 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. Este valor lo consigue guardar porque el conmutador, en automático cuando salta, coloca el último valor del *pc* que tuvo el proceso en #e0, mientras que si fue la instrucción *halt* la que salto, coloca un negativo en #e0.

Lo que sigue es salvar el valor del *gjump* tomando directamente el valor actual del contenido de #999, porque los saltos hacia #e1, por parte del conmutador y la instrucción *halt*, no dejan esa marca en #999, entonces para cuando llega la ejecución a #e8 perfectamente puede tomar el valor de #999, y guardarlo en la zona de variables del sistema, más precisamente en #c14.

Por último lee el valor de #e0, y si es negativo salta a la fracción de borrado, en caso contrario salta a la dirección de actualización, en nuestro caso #e0 será negativo, por lo que saltará a la fracción 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 saltará al área de borrado. Si vemos el diagrama(en fig.3.29 y fig.3.11), es en esta parte dónde la flecha sale de la fracción del preámbulo hasta el extremo izquierdo dónde está la fracción de borrado.

Como vemos en el diagrama(fig. 3.31), una vez que llega al área de borrado, lo primero que hace es preguntarse si es el proceso 0 desde dónde se ejecutó esta instrucción. Si no lo es, continua con la ejecución normal, pero en caso contrario salta a la fracción azul claro,

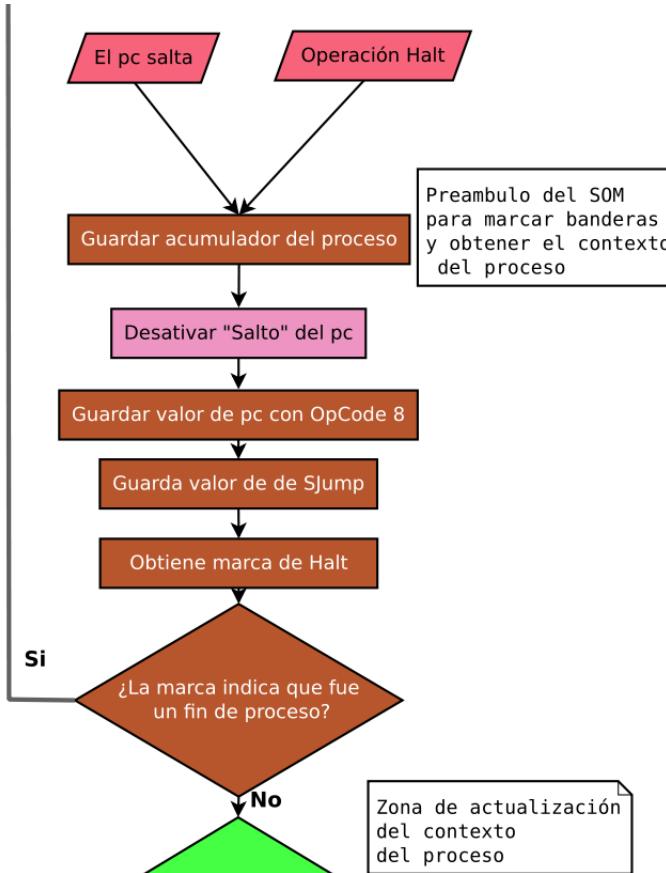


Figura 3.29: Acercamiento al preámbulo en el diagrama

que realiza operaciones especiales en el proceso 0, la fracción de gestión del proceso 0 dentro del núcleo del sistema. Por lo tanto, el proceso no es borrado, se llega a una zona de gestión que termina conectando con la fracción del lanzamiento de procesos. Es por eso que para lanzar los procesos que se han cargado el usuario requiere de escribir la instrucción *halt* en el proceso 0.

Como el bloqueo del proceso 0 pasa necesariamente por la parte que recupera el valor de #s1(#801 en la implementación), es buen momento para entender qué función tiene esta variable. 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 contador del proceso que se está ejecutando en ese momento, con un código de operación 6. Para acortarlo usaremos la nomenclatura *6-dir*, usando como prefijo el código de operación que tiene la dirección a la que hacemos referencia.

Podemos ver la imagen 3.33, que muestra el inicio de la zona de borrado, y lo primero

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

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

que hace es obtener el valor de `#s1`, lo convierte en una *1-dir*, o dirección con un código de operación 1, para guardarla en una variable del sistema que más adelante será útil en caso de que el proceso sea distinto del proceso 0. Por ahora lo que sigue es obtener el ID contador del proceso que se está ejecutando, y verificar si es el proceso 0, en caso de que lo sea salta a la fracción de gestión del proceso 0. Como se tiene la dirección en la forma *1-dir*, esta se guarda en la dirección `#s29`, para que cuando ejecute esa instrucción obtenga el contenido de la dirección de la zona de procesos a la que hace referencia, que será el ID contador.

En nuestro caso, como si es el proceso 0, salta a la fracción azul claro en la dirección `#s98` para 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 éste proceso este en ejecución.

Si vemos la imagen 3.32, podremos notar que en `#s98`, dirección a la que salta el proceso de borrado, verifica primero si hay más procesos. Esto se logra consultando a la variable del sistema `#c4`, que contiene un **contador de identificadores general**, que guarda el valor del ID contador más alto 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.

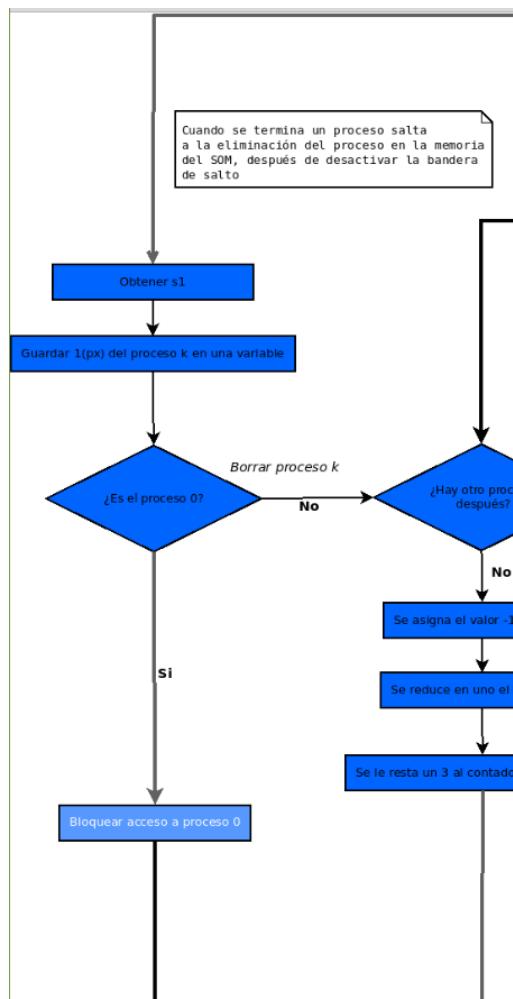


Figura 3.31: Acercamiento a la parte del bloqueo al proceso 0

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	
Asigna id organizer=0	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
	s145	1(c3)	LDA c3	Obtiene p4
Asigna a dir organizer=p0	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

En la misma imagen(3.32) está la condicional de salto a #s141, el salto que hace cuanto no hay más procesos, y ahí mismo hay una continuación directa de la dirección #s100 a la #s141, puesto que en medio está la zona de lanzamiento de procesos, a la que el flujo continuará si hay procesos para lanzar. Lo que hace el flujo cuando continúa en #s141 es reiniciar los valores del *id organizer/organizador de ID*, y del *dir organizer/organizador de direcciones*. También coloca en la dirección #004 el valor de 0, puesto que esta dirección está reservada para el SOM, ahí se encuentra el valor del **identificador estático** del proceso que se está ejecutando, en este caso el 0, y posteriormente salta de regreso al proceso 0. En caso contrario, en el que sí haya más procesos para ejecutar, cuando el ID contador general de la zona de variables del sistema tenga un valor mayor a 0 continúa en la fracción lanzamiento de los procesos.

Sistema operativo			
	s19	1(s1)	LDA s1
<i>Guardar 1(px)</i>	s20	4011	SHT 11
	s21	2(c8)	ADD c8
	s22	6(c0)	STO c0
	s23	1c7	LDA c7
			Se obtiene el id del proceso que se estaba ejecutando
<i>Verificar si es el proceso 0</i>	s24	7(000)	SUB 000
	s25	3(s98)	BLZ s
<i>Si no lo es</i>	s26	1(c0)	LDA c0
	s27	2(c11)	ADD c11
	s28	6(s29)	STO s29
	s29	1(py)	LDA py
<i>¿Hay otro?</i>	s30	3(s52)	BLZ s52
<i>Guardar 1(py) en c17</i>	s31	1(s29)	LDA s29
	s32	6(c17)	STO c17
<i>Iniciarizar contador de secciones</i>	s33	1(c18)	LDA 004
	s34	6(c0)	STO C0
			se Carga 4 para funcionar como contador
			Se guarda el contador de secciones del proceso

Figura 3.33: E-CARDIAC C Borrado de proceso parte 1

## Zona de lanzamiento de procesos

En nuestro caso, la ejecución continuará en la fracción naranja del diagrama, las direcciones que están entre #s100 y #s141, debido a que si tenemos más procesos para ejecutar. Como vemos en la imagen 3.34, continuamos, después de verificar que el ID contador general(*id counter*) no es cero, en la fracción naranja. 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 los procesos de forma correcta, 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`, porque 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 contador general(*id counter*), que contiene el valor máximo de los contadores que están en la zona de procesos, su valor por defecto es 0, porque ese es el máximo al inicio. Después de agregar tres procesos en nuestra implementación tiene un valor de 3; `#c5` es su símil, pero en lugar de tener el identificador mayor de la zona de procesos, tiene la dirección de ese ID contador, es decir, tiene la dirección del último ID contador de la zona de procesos que no es negativo.

De esta forma podemos conocer cuál es el último proceso, pero para conocer la información del proceso a ejecutar necesitamos de los **organizadores**, de `#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, respectivamente.

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*, 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 otro después. En ese punto se reinician, empezando por el proceso 1 de nuevo hasta que terminen de todos los procesos su ejecución. 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, solo irán de 1 a n, 1 a 5 en esta implementación.

En la imagen 3.35 podemos seguir los pasos descritos anteriormente, el aumento de los organizadores para acceder al siguiente proceso en cola, hasta la condicional en la que puede saltar a `#s134`(lo hace en `#s109`) en caso de que haya que reiniciar los organizadores, es decir, que ya se ejecutó el último proceso de la cola, pero que aún quedan más. Para ver cómo se realiza este reinicio podemos observar como en la imagen 3.37 se usan las constantes

de la zona de variables para obtener el contexto del primer proceso del usuario disponible, y después como regresa a #s110 a lanzarlo. En caso de que ya no hubiera procesos, ni siquiera se llegaría a la fracción naranja.

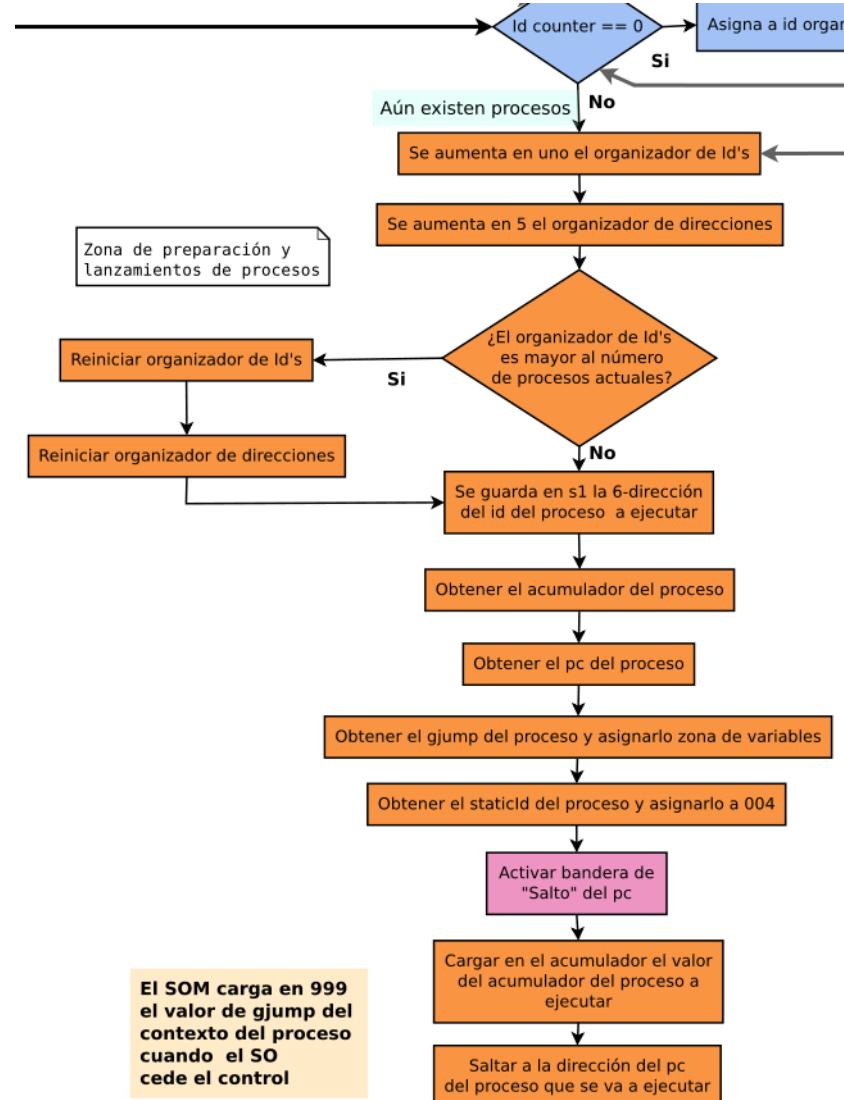


Figura 3.34: SOMC Lanzamiento de procesos

Para detectar que no hay otro proceso después en la cola se le resta al ID contador general el valor del organizador de identificadores, si este resultado es negativo significa que el proceso que el organizador va a lanzar no existe, si el organizador es mayor al contador de identificadores general es porque hay que reiniciar y volver a ejecutar el primer proceso.

Continuando en #s110, lo que sigue después del rombo naranja en el diagrama(fig.3.34), es la colocación de la *6-dir* de dónde se encuentra el identificador del proceso a ejecutar en

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 3.35: SOMC Lanzamiento de procesos parte 1

#s1, para su uso posterior en el borrado y la actualización. Las instrucciones siguientes, que se ven en la parte final de la figura 3.35, toda la 3.36, y la parte inicial de la 3.37 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, el organizador de identificadores, pues con solo ir sumando un uno a esa dirección 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, para que esos valores sean cargados por el acumulador en el momento indicado.

Por ejemplo, el caso del contador de programa, la dirección de este se obtiene en #s115 al sumarle un 1 al contenido del acumulador, que en ese momento era un *1(p5)/1774*, 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 este 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 máquina lo que contiene la dirección #776, que es el acumulador que se encuentra en el contexto del proceso.

Por otra parte, en #s121 lo que sucede es que se carga en el acumulador *1p7/1776* para

sumarle otro 1 y obtener 1777, que es guardada en #s128 para poder obtener el *gJump*, y guardarla en la variable del sistema #c14. Esto es de suma importancia por qué es la máquina quien coloca en #999 está variable. La acción se realiza justo después de ejecutar la instrucción que se encuentra en #s133, puesto que si no lo que haría la máquina por defecto es guardar en #999 el valor de  $8(s133)$ , que es de dónde salta, 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, porque 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. Esta dirección es una reservada para el sistema que es usada por la máquina para identificar que proceso se está ejecutando, usando como referencia al identificador estático, porque 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, porque 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 *conmutador*, 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
<i>Preparación gpc y gcc</i>	<b>s117</b>	<b>2000</b>	<b>ADD 000</b>	Se obtiene la 1-dir del gacc del proceso a ejecutar
	<b>s118</b>	<b>6(s132)</b>	<b>STO s131</b>	En 112 se guarda la 1-dir del gacc del proceso a ejecutar
<i>Preparación gjump</i>	<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>Guardar Static ID en 004</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
<i>Guardar en 004</i>	<b>s123</b>	<b>6(s128)</b>	<b>STO s127</b>	Guarda la 1 dir del gjump en 113
	<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
<i>Guardar en 004</i>	<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 3.36: SOMC Lanzamiento de procesos parte 2

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 3.37: SOMC Lanzamiento de procesos parte 3

### Lanzamiento de procesos en la máquina virtual

Observamos en la figura 3.38, que en las variables del sistema que representan a los “organizadores”, #971(#c6) y #972(#c7), están la dirección del ID contador del proceso que se va a lanzar y el contenido del mismo, respectivamente.

Para observar la situación del sistema antes de lanzar el proceso observemos la imagen 3.39, 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 estatus del conmutador(*SW Status*) que indica que ya se está ejecutando un proceso del usuario. Pero con un contador del conmutador(*Counter SW*) 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 verifico la bandera y se reinició quedándose en 0, antes de ejecutar #933 aún no ha sumado nada a ese ciclo, cuando lo termine le sumará un uno, por lo que el proceso del usuario iniciará con un ciclo ya usado.



Figura 3.38: Estatus de organizadores antes de lanzar el proceso

Para la imagen 3.40a ya vemos al contador de programa en #110 y al contador del conmutador en 1, por lo que le quedan 29 ciclos antes de ceder los recursos nuevamente, ahora ya es turno de que el proceso *pintor v1* empiece a imprimir los números del 1 al 10. En

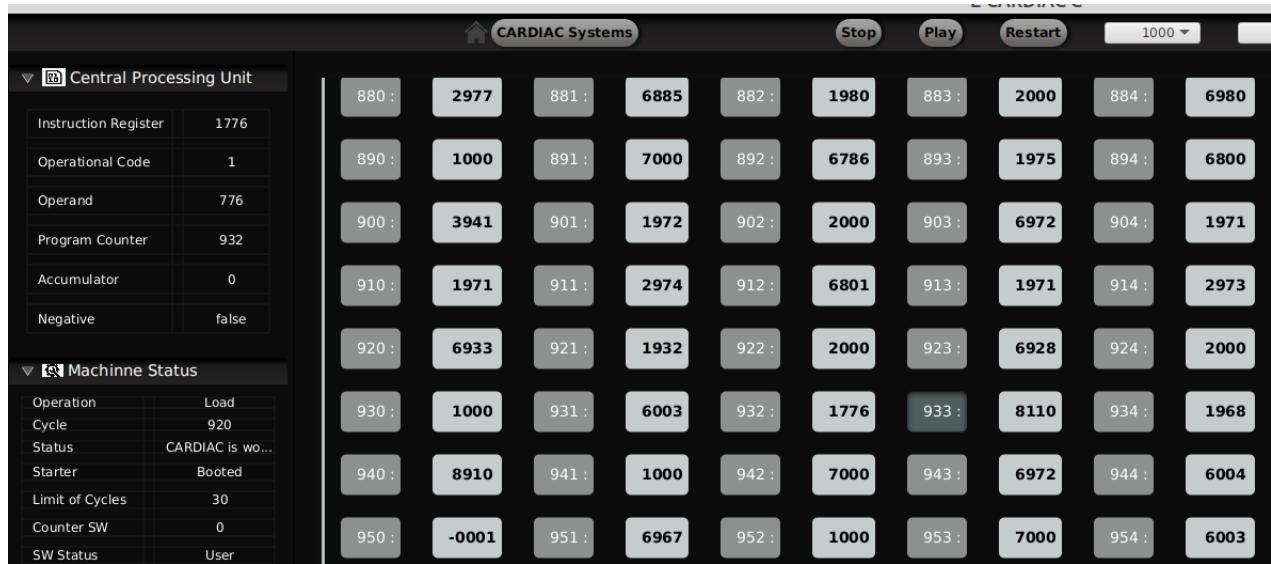


Figura 3.39: Valores del SOM antes de lanzar el proceso

la imagen 3.40b vemos al proceso justo antes de ceder los recursos al SOM de nuevo, como el contador del conmutador ya tiene el valor de 30 la instrucción que se encuentra en #119 ya no será ejecutada, y saltará directamente al preámbulo.

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 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. Cada que nos refiramos a un proceso no nos referiremos a él 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 solo falta imprimirlo, como la impresión sucede antes de que la condición de parada se efectúe la lógica del proceso es correcta.

▼ Central Processing Unit	
Instruction Register	8110
Operational Code	8
Operand	110
Program Counter	110
Accumulator	0
Negative	false
▼ \$1 Machine Status	
Operation Cycle	Jump
Status Starter	CARDIAC is w... Booted
Limit of Cycles	30
Counter SW	1
SW Status	User

(a) Proceso 1 en ejecución

▼ Central Processing Unit	
Instruction Register	1104
Operational Code	1
Operand	104
Program Counter	118
Accumulator	7
Negative	false
▼ \$1 Machine Status	
Operation Cycle	Load
Status Starter	CARDIAC is w... Booted
Limit of Cycles	30
Counter SW	User
SW Status	
▼ \$2 Output	
ID:0000 ----	
ID:0001 0001	
ID:0001 0002	
ID:0001 0003	
Download	
Queue	
Secondary Memory	
Deck Mode Terminal Mode	

(b) Final del primer ciclo de *pintor v1*

Figura 3.40: E- CARDIAC C

### 3.2.7. Sistema Operativo Mínimo C: Actualización y borrado

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 conmutador es directo a  $\#951(\#e1)$ , el preámbulo, que guarda el valor del contador de programa, el acumulador, y 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 el conmutador. 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  $\#802(\#s2)$ , a la zona de actualización.

#### Fracción Actualizar proceso

Podemos observar el flujo en la fracción 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. Pero primero hace una validación para verificar que la fracción de lanzamientos haya hecho lo correcto y asignado la *6-dir* a  $\#s1$ , pues será un valor pivote para conocer en qué dirección está el contexto del proceso que estaba en ejecución.

Lo que hace en  $\#s2$  es obtener la *6-dir* del ID contador del proceso(que está en  $\#s1$ ), 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, contador que fue resguardado en  $\#c1$  por el preámbulo. Este 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 estratégicos, y el subproceso de actualizar coloca

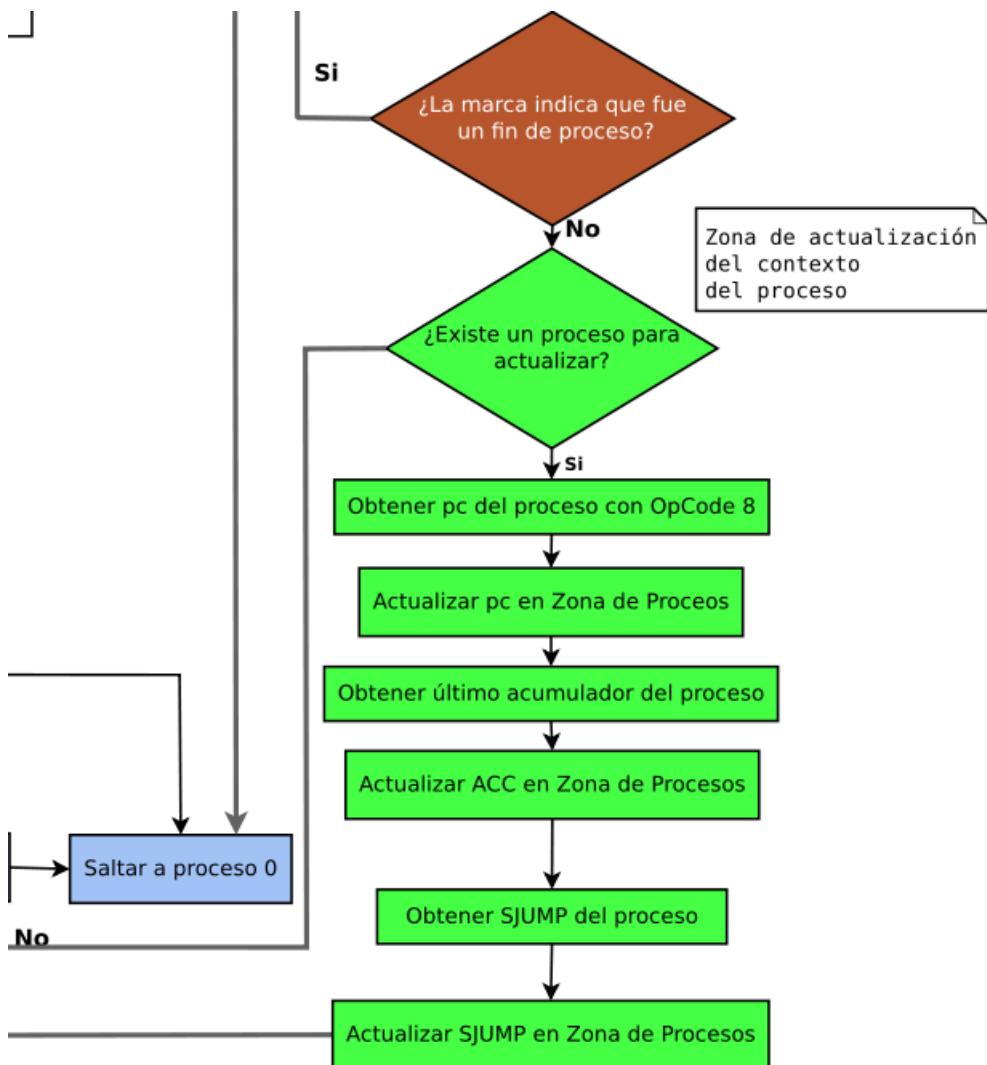


Figura 3.41: Diagrama de actualización de proceso

esos valores en el lugar correspondiente para cada variable del contexto del proceso. Una vez terminado esto, el flujo sigue hacia la zona de lanzamiento de procesos.

Pero antes de movernos hacía allá regresemos a la validación y veamos que sucede si no la 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 subprocesso que verifica si es que ya no hay más procesos para ejecutar, si es que es necesario lanzar de nuevo el proceso 0 y reiniciar los organizadores. En caso de que sí haya más procesos para ejecutar se irá a la zona de lanzamiento de procesos. En caso de que el contenido de `#s1` no sea negativo, el flujo actualiza el contexto del proceso

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
	<b>s6</b>	<b>1(c1)</b>	<b>LDA C1</b>	Se obtiene el último pc del proceso con forma 8(pc)
<i>Actualizar gacc</i>	<b>s7</b>	<b>6(p5)</b>	<b>STO p5</b>	En p5 se actualiza gpc
	<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>Actualizar gjump</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
	<b>s12</b>	<b>6(p6)</b>	<b>STO p6</b>	Se actualiza el valor de gacc
<i>Saltar</i>	<b>s13</b>	<b>1(s12)</b>	<b>LDA s12</b>	Obtiene la 6(p6) del proceso que se está actualizando
	<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

y cuando termina salta directo al lanzamiento de procesos, porque 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 en la cual 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 máquina durante el proceso 1.

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

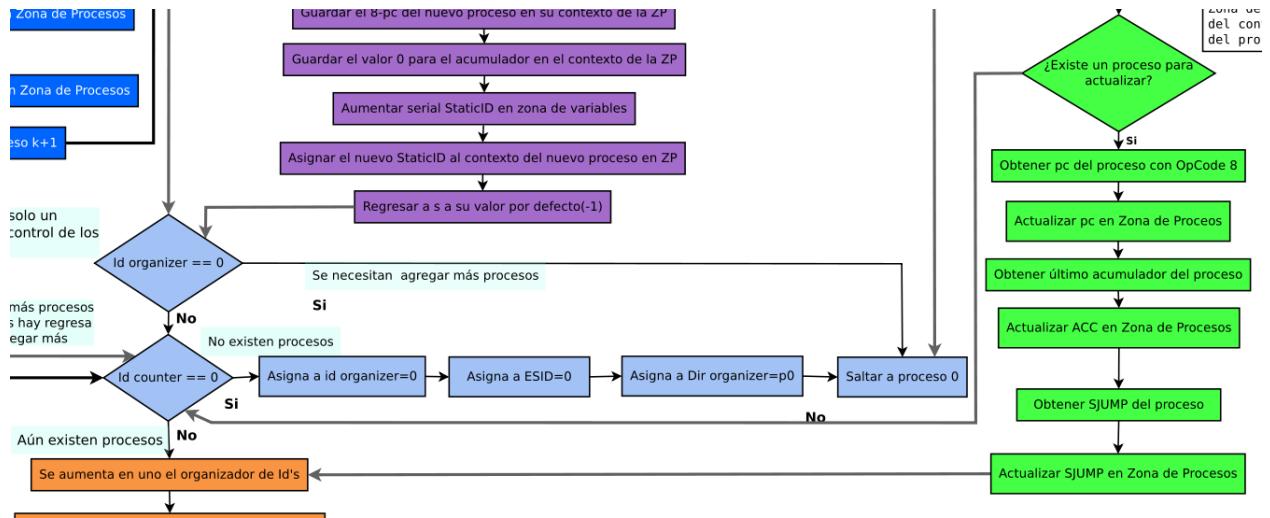


Figura 3.44: Conexión entre actualización de procesos y lanzamiento



Figura 3.45: Contexto del proceso 1 actualizado

### 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 del 2. Actualmente los recursos los tiene el proceso 1 que está a punto de saltar a #122, porque acaba de guardar lo que hay en #104 en el acumulador, y es un negativo. Como la instrucción que está en #122 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

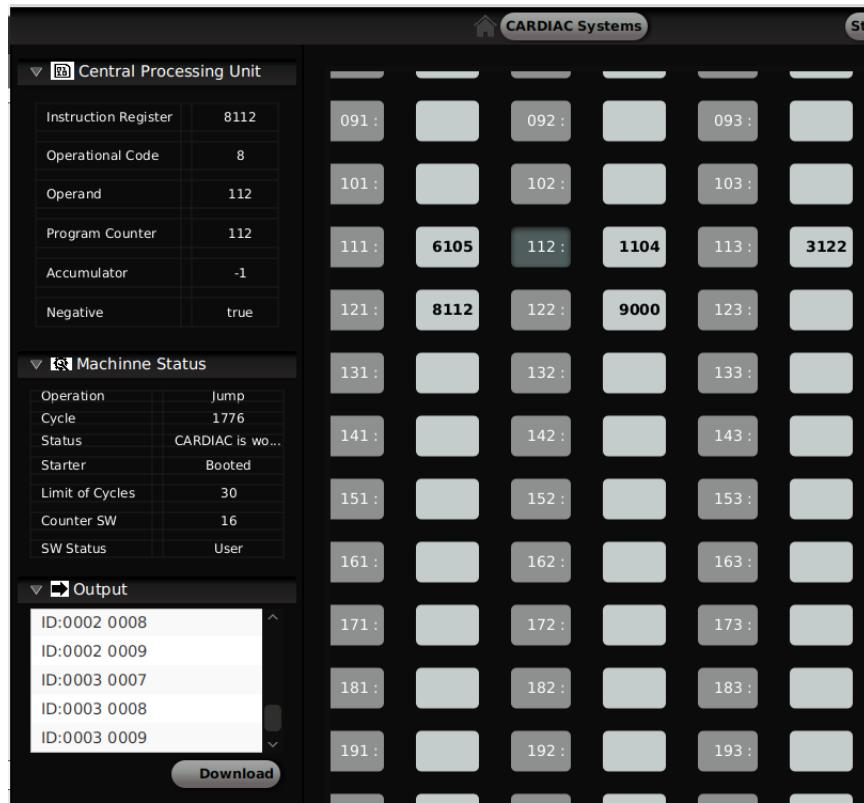


Figura 3.46: Proceso 1 antes de finalizar

proceso 0. Para seguir el código podemos ver las imágenes 3.33, 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 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, esto es porque 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, si no está ligado a la zona de procesos con su correspondiente contexto, para el sistema no existe.

Entonces lo que hace el subproceso para eliminar un proceso del usuario es volver negativo el ID contador de dicho proceso, si es que es el último en la lista, y disminuir el ID contador

Limit of Cycles	30								
Counter SW	19								
SW Status	50								
▼ Output									
ID:0003 0007 ID:0003 0008 ID:0003 0009 ID:0001 0010 ID:0001 ----									
<b>Download</b>									
940 :	<b>8910</b>	941 :	<b>1000</b>	942 :	<b>7000</b>	943 :	<b>6972</b>	944 :	<b>6004</b>
950 :	<b>-0001</b>	951 :	<b>6967</b>	952 :	<b>1000</b>	953 :	<b>7000</b>	954 :	<b>6003</b>
960 :	<b>1950</b>	961 :	<b>3963</b>	962 :	<b>8802</b>	963 :	<b>8819</b>	964 :	
970 :	<b>0784</b>	971 :	<b>0774</b>	972 :	<b>0001</b>	973 :	<b>1000</b>	974 :	<b>6000</b>
980 :	<b>0003</b>	981 :	<b>0004</b>	982 :	<b>0998</b>	983 :	<b>0004</b>	984 :	

Figura 3.47: Preámbulo en la finalización del proceso 1

950 :	<b>-0001</b>	951 :	<b>6967</b>	952 :	<b>1000</b>	953 :	<b>7000</b>	954 :	<b>6003</b>
960 :	<b>1950</b>	961 :	<b>3963</b>	962 :	<b>8802</b>	963 :	<b>8819</b>	964 :	
970 :	<b>0784</b>	971 :	<b>0774</b>	972 :	<b>0001</b>	973 :	<b>1000</b>	974 :	<b>6000</b>
980 :	<b>0003</b>	981 :	<b>0004</b>	982 :	<b>0998</b>	983 :	<b>0004</b>	984 :	
990 :		991 :		992 :		993 :		994 :	

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

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 la fracción azul claro(como en la imagen 3.49). No hay necesidad de borrar el contenido del resto del contexto, se sobrescribirá cuando haya otro proceso, lo importante es el ID contador, porque 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 no se puede realizar la acción antes descrita. Por qué el sistema espera que los procesos sean consecutivos, con un ID contador que sea ascendente, y que con el primer contenido negativo que encuentre en una dirección dónde debe estar un ID contador 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 contador, el cual no se mueve, porque es un valor ascendente, y lo que se hace para borrar el proceso es convertir el ID contador con el valor más alto, que actualmente tiene un valor de 3, en -1. Por ende el proceso 2 ahora tendrá

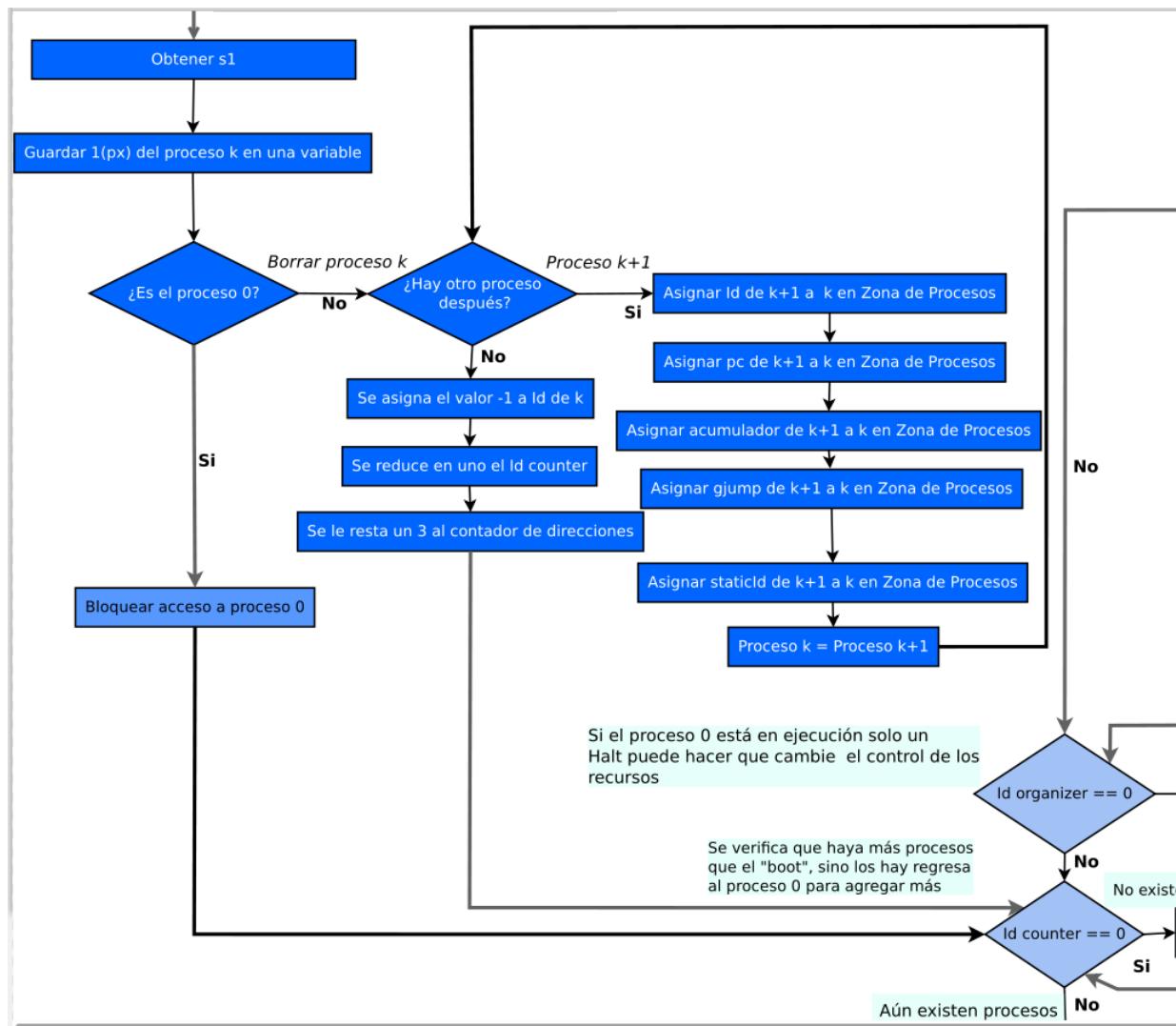


Figura 3.49: Acercamiento a zona de borrado en diagrama

un ID contador con un valor de 1, pero guarda su identidad por el identificador estático, así tanto el sistema como nosotros podemos identificar siempre al proceso 2 aunque su ID contador sea un 1 o un 2.

En la imagen 3.50 vemos la zona de procesos antes de que el proceso 1 sea borrado, aun todos los ID contador e ID estáticos son iguales, mientras que en la imagen 3.53 vemos que ahora el estático que se encuentra en #778 es el 2, diferente al ID contador del proceso que ocupa ese contexto, porque se han recorrido los procesos. Mismo caso para el proceso 3, que está en el contexto que inicia con el ID contador 2, además podemos ver que dónde antes estaba el ID contador 3, en #784, ahora hay un -1 lo que indica el fin de la lista de procesos,

pero 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. Esto es porque se quedan ahí como basura, se sobrescribirá cuando haya un nuevo proceso, solo se necesita que el ID contador sea negativo para que se marque el final de la lista.

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

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

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

Para analizar el borrado desde el código vamos a la imagen 3.33, a partir de la dirección `#s27`, después de verificar que no se trata del proceso 0. Llamemos  $k$  al proceso que va a ser borrado, entonces como lo que necesitamos es verificar si hay más procesos con ID contadores mayores, en `#s27` se obtiene la *1-dir* del siguiente proceso, al que llamaremos  $k+1$ . Se carga el contenido en el acumulador para verificar 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 contador del proceso  $k+1$ . Si a esa dirección se le suma uno se obtendrá la dirección que tiene al *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 cual está siendo borrado o recorrido).

Después de completar el copiado de contexto del proceso  $k+1$  a la zona del 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	
	s46	1(c0)	LDA c0	Obtener contador de contextos
<i>Aumentar contador de secciones</i>	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

$k$  viene otra iteración, pues la subrutina de copiado está 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 ocupaba ahora tiene que ser limpiada, ya sea recorriendo otro proceso(si existe) o dejando el ID contador 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 contador 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 contador 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 cual los “organizadores” pueden acceder en la zona de procesos. Esto se aprecia en la figura 3.52, y posteriormente salta hacia la fracción de gestión del proceso 0.

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

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
	s51	8(s27)	JMP s27	Va a verificar si hay otro proceso adelante
Si de s30	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
Borrado	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

había 3 procesos activos, y un 784 en el contador de direcciones, indicando la dirección del último ID contador disponible. A diferencia de la figura 3.55, donde se ve cómo 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 contador que se puede encontrar está en 779.

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

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

Algo curioso pasa con los organizadores, y es que como los organizadores lanzan los procesos de acuerdo a su posición en la lista, es decir de acuerdo a su ID contador, como se acaba de lanzar el proceso con ID contador 1 el que sigue es el proceso con ID contador

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

2, por lo que se estaría saltando para esta ejecución al proceso estático 2 que ahora tiene ID contador igual 1 y ejecutaría el proceso 3 con ID contador 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.

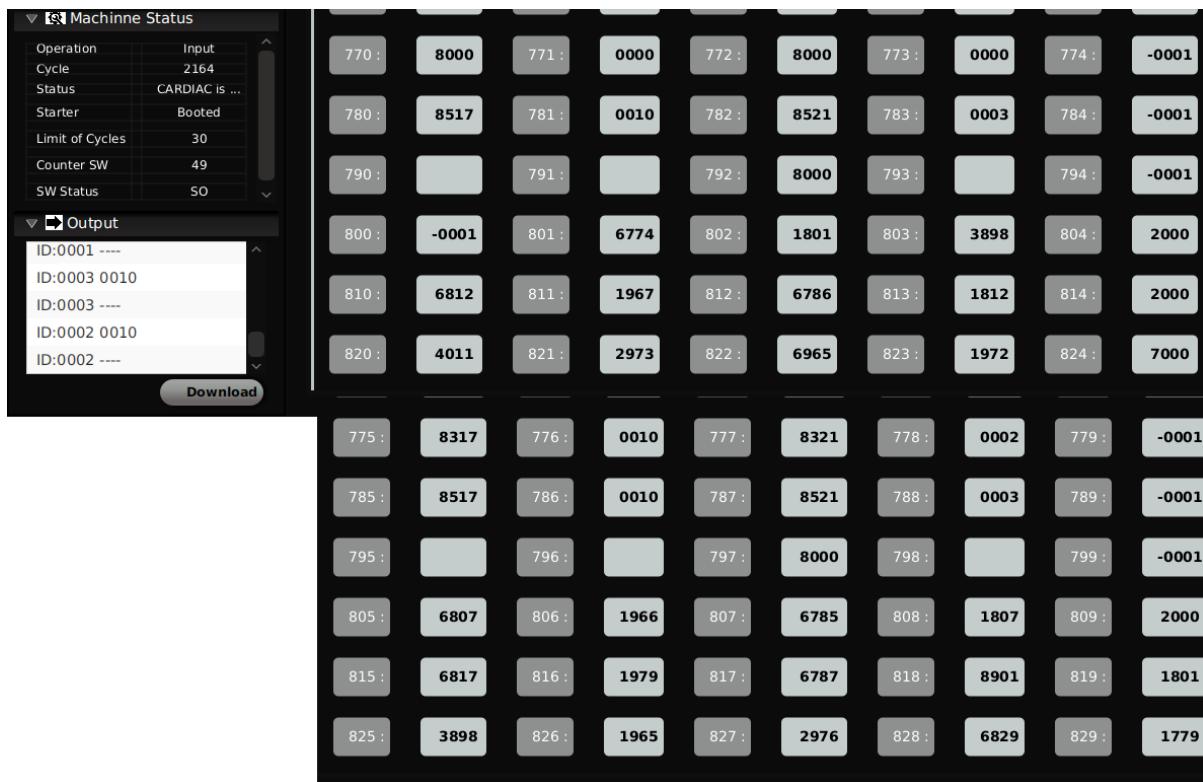


Figura 3.56: Salidas finales de los procesos

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.

```
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

### 3.2.8. SOMC: 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 máquina para generar una ejecución concurrente de procesos de forma que al usuario le sea práctica la ejecución. Veamos en la siguiente lista los pasos a seguir del usuario para ejecutar programas en la máquina virtual.

1. Diseñar un programa de acuerdo a la arquitectura de la máquina 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 esta instrucción final agregar otro par donde 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.
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 esta implementación.
7. Si el usuario ha cargado los programas que necesitaba, cuando esté 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 máquina virtual resulte exitosa y poder apreciar de la ejecución concurrente a nivel de procesos.

### **3.3. E-CARDIAC PC: Electronic CARDboard Illustrative Aid to Concurrent Parallel Computation**

Pensar en paralelismo no es una tarea sencilla para nuestras mentes, aunque a veces así lo pareciera. Basta pensar en la cantidad de actividades que puedes realizar al mismo tiempo, generalmente es solo 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 calentando 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 de 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 mientras el otro u otros están de fondo con su ejecución, están afectando a sentidos (como el auditivo). O en otro caso esperamos el resultado de cálculos en paralelo que convergen hacia un solo resultado, no prestamos atención al cómo se ejecutan los dos o más procesos al mismo tiempo.

Para hacer más intuitivo el mostrar procesos en paralelo decidí añadir un coprocesador al modelo de cómputo que tenemos, de esta forma el procesador principal se encargará de unas tareas y el coprocesador 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 creó la siguiente evolución del modelo, **Electronic CARDboard 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 coprocesador, aunque será difícil diferenciar al procesador principal del secundario por qué ambos realizarán tareas muy importantes para el modelo de cómputo. En la figura 3.58 se encuentra el diagrama para esta 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, las dos CPUs, uno en azul llamado *CPU Loader*(el cargador) y otro en rojo *CPU Executor*(el ejecutor). Cada uno tendrá tareas particulares para el modelo, el principal es el cargador puesto que este es el que está conectado a la memoria secundaria y al *input/entrada* principal, la entrada que se conecta a la memoria secundaria y al usuario directamente por medio del modo de entrada de tarjetas, la entrada masiva de información. Mientras que el ejecutor es el coprocesador porque solo realizará un número determinado de tareas que el usuario haya solicitado desde la CPU principal.

Mencionaba que sería difícil de diferenciar al principal del que no lo es porque el segundo, el ejecutor, será el encargado de ejecutar como tal los procesos, el cargador solo se encarga de cargar el sistema operativo y de permitir al usuario la interacción con la máquina. Aunque el ejecutor también tiene una conexión a un método de entrada, el método que en la máquina 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.

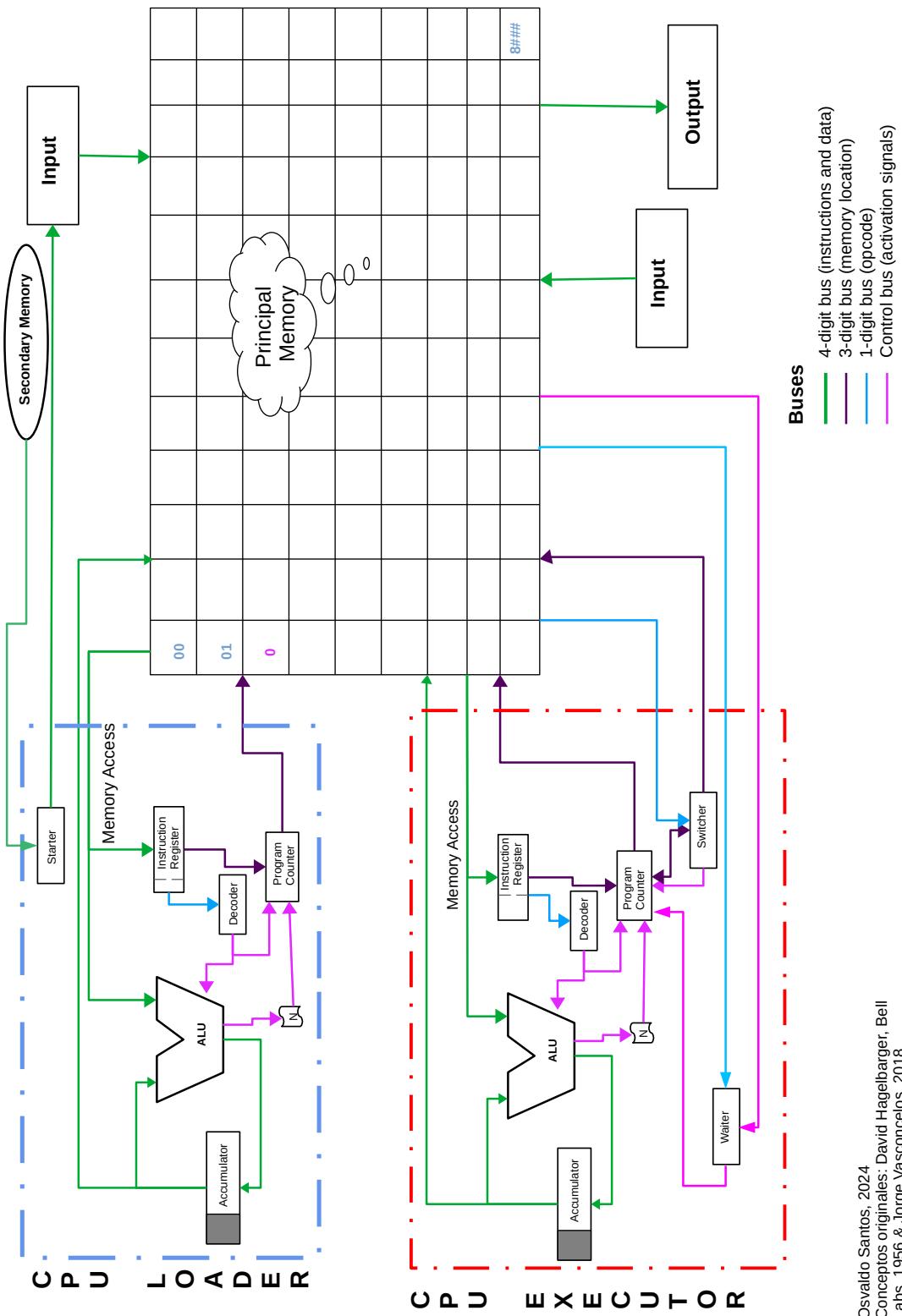


Figura 3.58: Arquitectura de cómputo paralela.

Podemos también observar que la CPU principal tiene menos componentes que la CPU de E-CARDIAC C, puesto que solo mantiene el *starter/iniciador*, para poder cargar el sistema operativo mínimo, ya que es el que tiene la conexión a esa entrada, pero el *switcher/conmutador*, como no lo necesita, se quedó únicamente en el ejecutor. No necesita al conmutador porque una vez que se cargue el sistema operativo mínimo, el usuario solo usara la CPU principal para estar cargando programas 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 la CPU secundaria. Al terminar de añadir el proceso, los recursos del cargador regresarán de inmediato al proceso 0 para que el usuario pueda seguir añadiendo programas.

Mientras que el ejecutor sí necesita de un conmutador, por qué estará cambiando el control de los recursos entre el sistema operativo mínimo, y los procesos que esté ejecutando. Adicionalmente, requiere de un elemento extra en el hardware, un elemento que he llamado *waiter* o vigilante, 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.

Para lograrlo utiliza el bus azul, que lo conecta directamente a la memoria, y por el cual recupera el valor que está guardado en #c4, que es el contador de identificadores general<sup>1</sup>, si tiene un cero significa que no hay nada para ejecutar. Pero no siempre está activo, se activa cuando el contador de programa apunta a la dirección #s97 del sistema operativo, dirección que solo sirve como parada, y que se define al inicio de la implementación como la dirección de activación del vigilante. Para realizar esto se necesita del bus de control que conecta al vigilante con la memoria, y transfiere la información de que se está apuntando a la dirección #s97. Para pausar al contador de programa, el vigilante tiene otro bus de control conectado a este que le sirve para indicarle que debe hacer una pausa, o en su caso, si ya hay programas para ejecutar, seguir con la ejecución en la dirección #s98.

Como podrán haber notado se intentó duplicar lo menos posible, 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 ambos procesadores, puesto que algunas serían innecesarias. De esta forma

---

<sup>1</sup>Como solo tiene valores menores a 9 no necesita más de un dígito, por eso el bus azul.

se logra economizar en la arquitectura al añadir la menor cantidad de hardware posible, lo que nos permite tener un sistema operativo mínimo que no aumenta mucho en complejidad en comparación con el modelo anterior.

### 3.3.2. Un sistema operativo para dos procesadores

El reto ahora es pensar en un sistema operativo que pueda manejar estas dos CPU, para ello se buscó 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 la segunda CPU cambie variables que la primera usa y produzca errores de sincronización, 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, y compleja, con procesos en paralelo. Para disminuir estos problemas sería necesario añadir mecanismos de sincronización de procesos para mantener la integridad de la memoria compartida. Pero 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, en la figura 3.59 podemos ver el diagrama para el nuevo sistema operativo *SOMPC*, dónde podremos analizar como será este nuevo sistema de forma muy general. En este diagrama podemos ver una clara separación en dos grupos, a la izquierda las fracciones de borrado, actualización y lanzamiento de procesos; y a la derecha, el de añadir un nuevo proceso, con una parte de la fracción del *preámbulo*, que está partido en dos porque es parte de ambos grupos, pero que es independiente en cada grupo. Es decir que no hay posibilidad que las instrucciones del grupo uno, que ejecutara el ejecutor, tengan un conflicto directo con las del grupo dos, que ejecutará el cargador. Tienen entradas y salidas diferentes, y no hay saltos que puedan llevar de alguna fracción del grupo uno a alguna 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

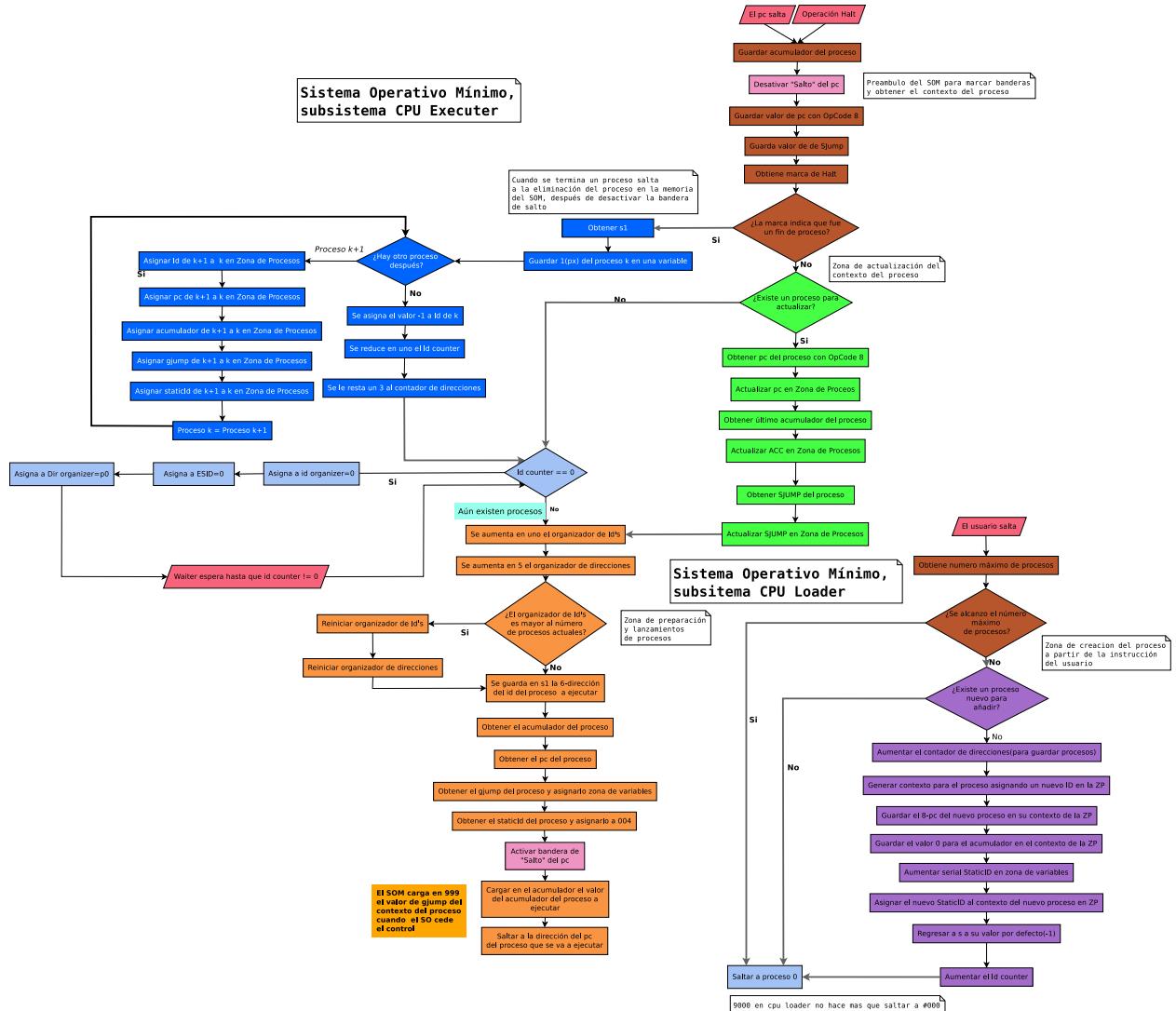


Figura 3.59: Diagrama de Sistema Operativo Mínimo Paralelo

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 sincronización al mantener al margen cada CPU con sus tareas, y hacer más intuitivo su funcionamiento. Diferente de un modelo de múltiples procesadores que puedan realizar exactamente las mismas tareas, y 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 las dos CPU tienen un funcionamiento homogéneo para darle al usuario una 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 este 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 cómo funciona esté sistema operativo mínimo mejorado.

#### Sistema operativo para CPU Loader

Empecemos con el *CPU Loader/Cargador*, porque es el que empezará las operaciones de la máquina, y además es el más corto en su funcionamiento. En la imagen 3.60 vemos un acercamiento a todo lo que realizará esta 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, primero el preámbulo verifica que no se haya alcanzado el número máximo de procesos, en caso de que sí, salta al proceso 0, pero ahora está instrucción de saltar al proceso 0 no se conecta con una fracción de gestión del proceso 0, solo salta. 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. Tal como pasaba en el SOM de C, la única forma de ejecutar esas instrucciones es si el usuario salta a añadir un proceso. No hay una interferencia con el resto del preámbulo porque si no se llega al preámbulo por medio de la instrucción de añadir un proceso, el flujo iniciará en #e1, e inevitablemente terminará en un salto a #s2 o #s24.

Ahora, un detalle importante al momento de añadir un proceso que cambia a diferencia

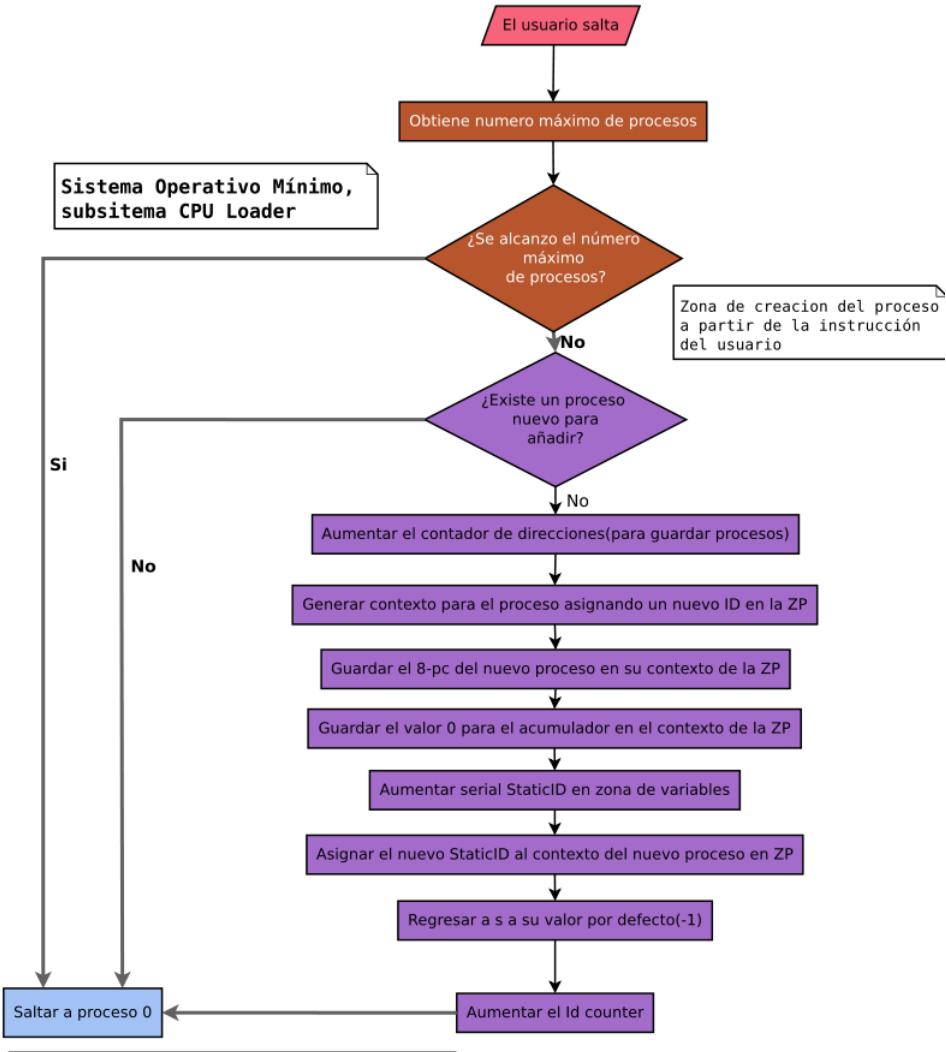


Figura 3.60: Añadir proceso en SOMP

de la versión para C, es el momento en el que se aumenta el ID contador general en `#c4`, que ahora se aumenta hasta el final, justo cuando se han preparado todas las variables y solo queda saltar de regreso al proceso 0, esto lo podemos ver también en la figura 3.62. Esto es sumamente importante porque en todo momento la otra CPU está activa, y en estado de espera a que `#c4` cambie su valor a uno distinto de 0. Cuando el *vigilante* detecta que cambió, permite que la ejecución en el ejecutor continúe y haga todos los procesos necesarios para lanzar un nuevo proceso. Si el valor de `#c4` cambiara al principio, como sucede en C, el vigilante permitiría la que el ejecutor continúe, y aún no estarían todas las variables que espera.

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

Aquí termina toda la participación en el sistema operativo para el cargador, el resto son ejecuciones del usuario y del *iniciador/starter*. Debido a que el resto de operaciones que realiza esta CPU son solo la carga de instrucciones que vienen por medio del usuario desde el modo de tarjetas/*deck*, el modo masivo, y el *arranque/booteo* por medio del iniciador, 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 de tarjetas, esto es, porque como el mismo diagrama de la arquitectura indica, 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 una CPU en específico, para no causar conflictos porque ambos requieran hacer uso del periférico al mismo tiempo. Así, el método de tarjetas queda reservado para el cargador, y el método de la terminal para el ejecutor.

Las salidas también tendrán conexiones diferentes para economizar, el ALU del cargador no reconocerá la operación *output*, y simplemente no ejecutará nada, puesto que el único que necesita hacer impresiones es el ejecutor. Otra instrucción que tendrá un comportamiento limitado en el cargador será la de *halt*, porque ahora si es ejecutada solo saltará a #000 sin importar qué operadores le acompañen.

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>Aumenta el Dir counter M[c5]++</i>	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
	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	
	s71	2(c9)	ADD c9	Convierte la nueva dirección en dir counter en 6-dir
<i>Previa de ID</i>	s72	6(s92)	STO s86	En s70 se guarda 6(px)
<i>Previa de pc</i>	s73	2000	ADD 000	Se obtiene la 6-dir del gpc del nuevo proceso
<i>Previa de acc</i>	s74	6(s84)	STO s78	En s72 se guarda 6(px)+1, la 6-dir del gpc del nuevo proceso
	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
<i>Guardar Nuevo Static ID para el proceso</i>	s77	2(c12)	ADD C12	Añade 2 para obtener 6(px)+4(Static ID)
	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
<i>pc nuevo</i>	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
<i>acc del nuevo proceso</i>	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
<i>Poner valor default en s</i>	s88	1(c10)	LDA c10	
	s89	6(s)	STO s	En s se coloca el valor -1
<i>Aumenta el Id counter M[c4]++</i>	s90	1(c4)	LDA c4	Obtener ID counter
	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
<i>Return</i>	s94	8(000)	JMP 000	Salta al proceso 0

Figura 3.62: Añadir proceso en un sistema operativo mínimo paralelo

## Máquina 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 una CPU diferente, y por supuesto que ahora tenemos dos CPU, uno en cada lateral. Al tener dos CPU fue necesario también reordenar el contenido del estado de la máquina(*machine status*), pues varios de sus elementos eran propios de la 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

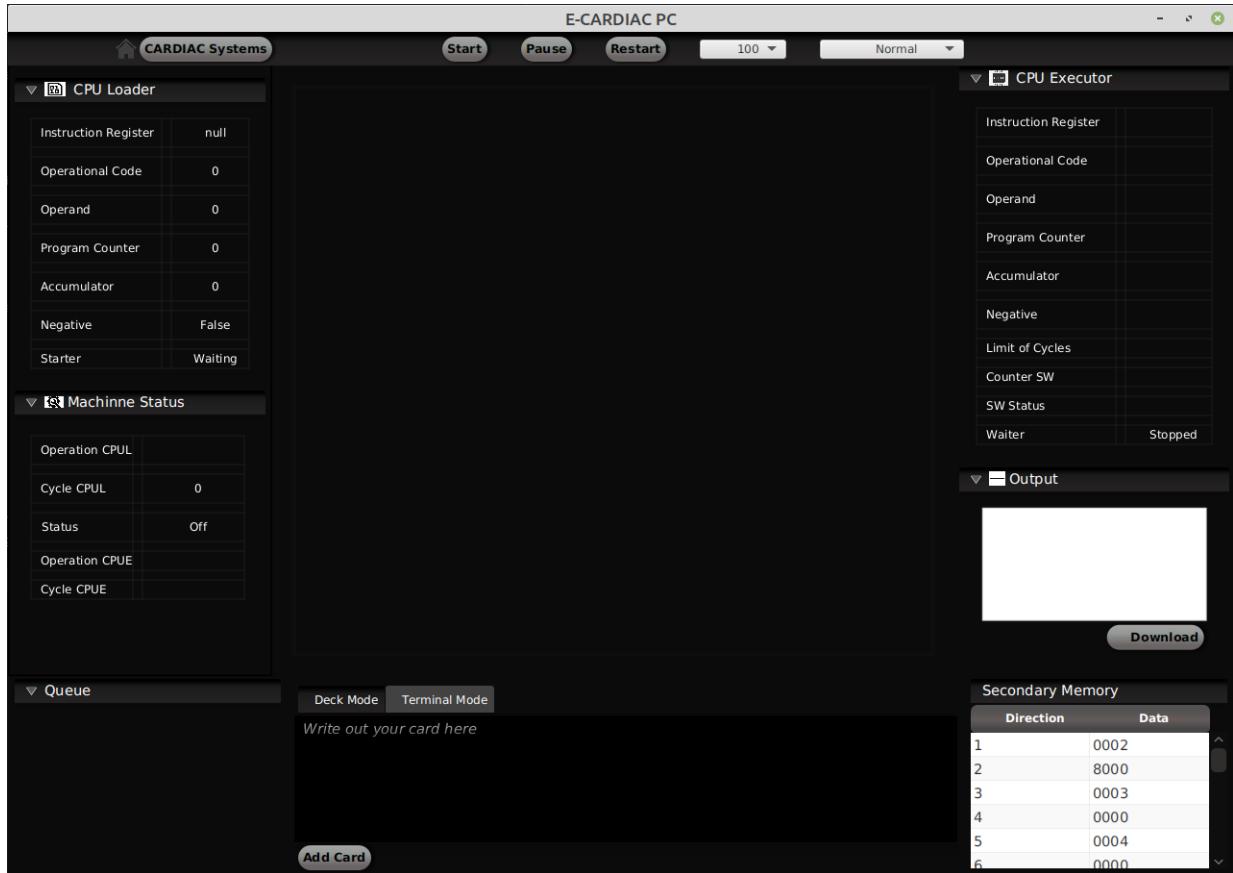


Figura 3.63: E-CARDIAC PC sin iniciar

starter, hacia la memoria principal.

En la imagen 3.64 podemos ver lo que sucede después de iniciar la carga(*booteo*), básicamente lo mismo que en C, con la diferencia que ahora podemos notar que está funcionando la CPU principal con *Cycle CPUL* y *Operation CPUL*, que indican el ciclo en el que va y la instrucción que está ejecutando, respectivamente. Mientras que los ciclos correspondientes al segundo CPU están vacíos(*Cycle CPUE* y *Operation CPUE*), al igual que la mayoría de las variables propias de la segunda CPU. Esto se debe a que una vez que inicia la máquina se inician las dos CPU, la primera inicia la carga a través del iniciador y continúa ejecutando instrucciones, pero la segunda inicia en la dirección #s97, dirección que activa al *vigilante/-waiter*, y que impide que el contador de programa avance hasta que haya un proceso cargado por el usuario.

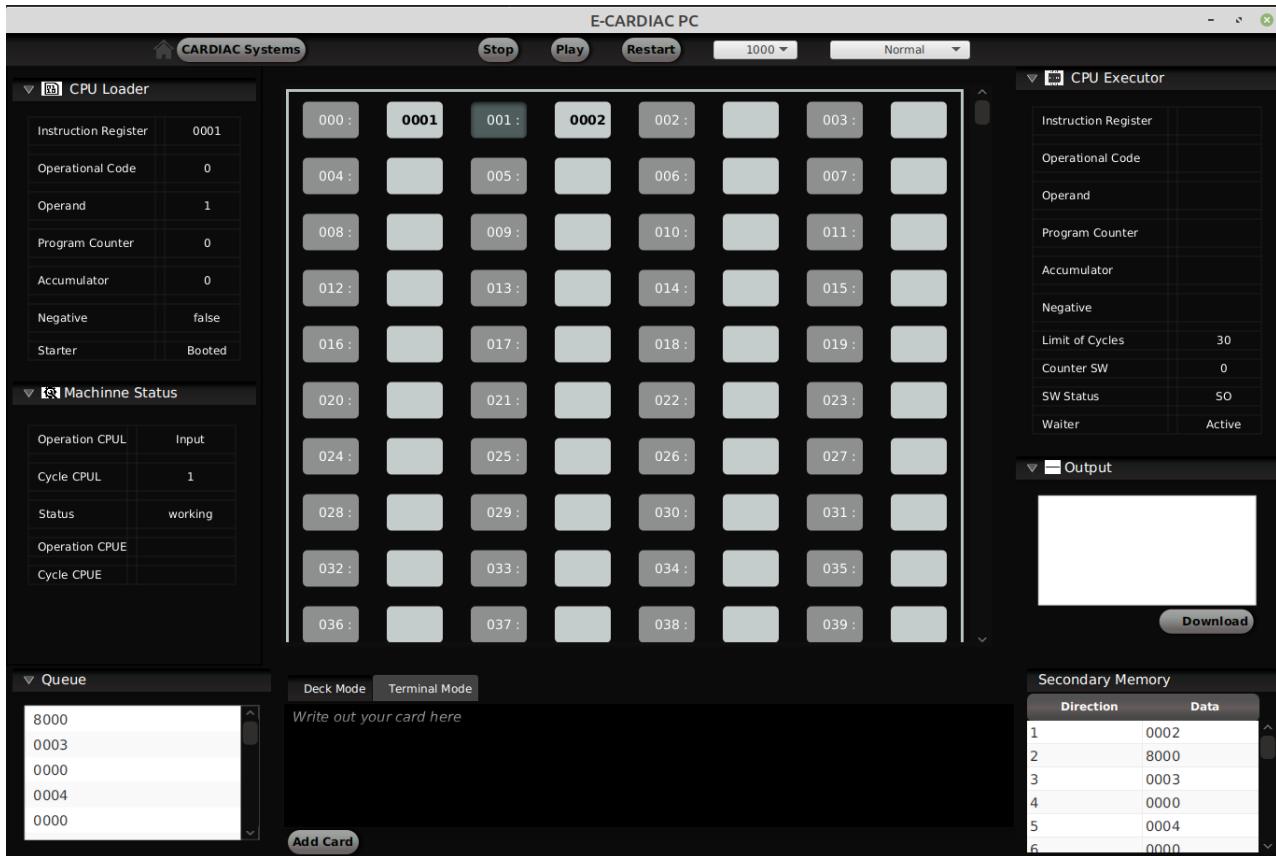


Figura 3.64: E-CARDIAC PC Booteo

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 cargó 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.

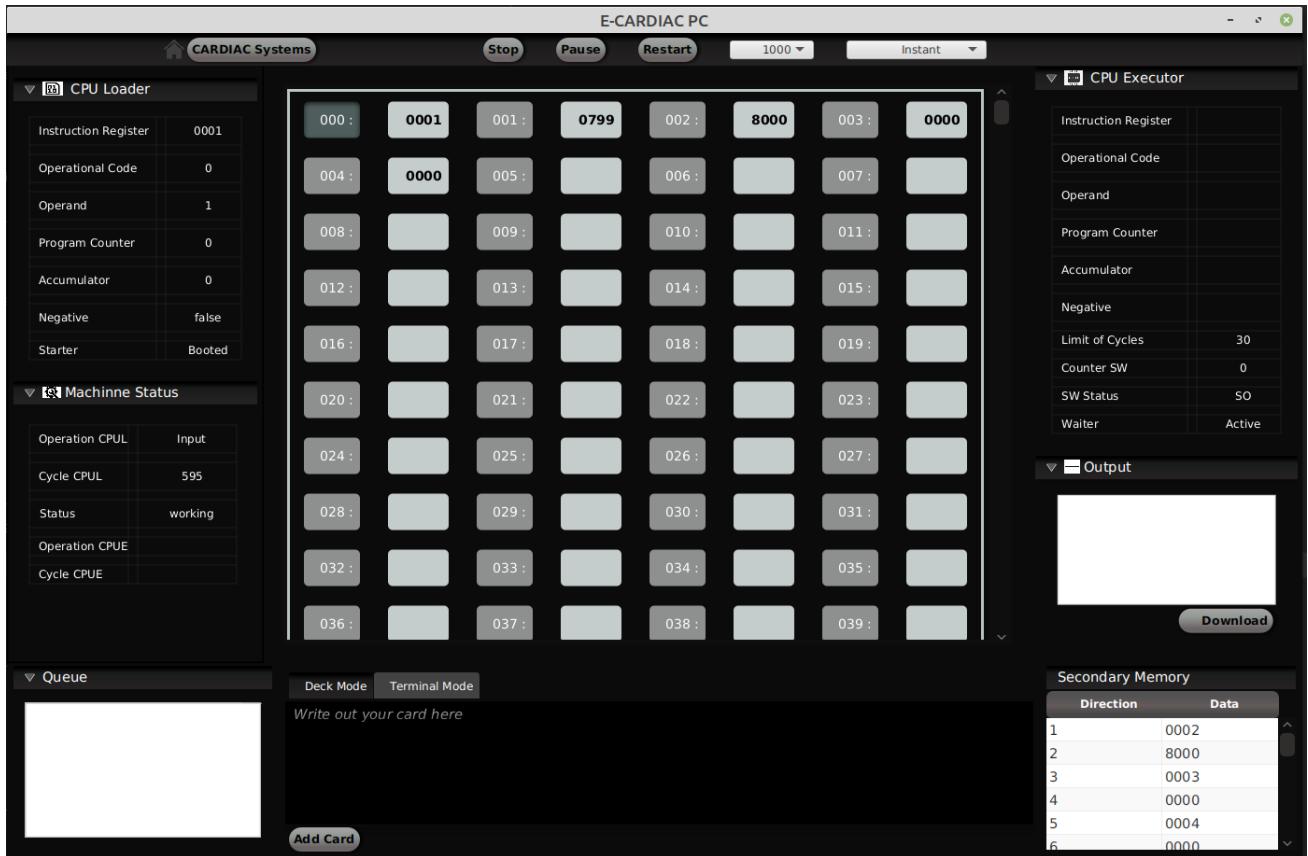


Figura 3.65: E-CARDIAC PC Iniciado

## Sistema operativo para CPU Ejecutor

Antes de cargar el primer programa en la nueva máquina será necesario conocer como funciona el sistema operativo para la segunda CPU. Si vemos el diagrama general del sistema operativo(fig. 3.59) notaremos que toda la parte izquierda es ejecutada por la segunda CPU. Si vemos de más cerca al preámbulo(fig. 3.66) podemos notar que las dos formas de entrada que tiene son si el contador del programa salta en automático, por medio del *conmutador*, 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, porque como fueron cambiadas algunas instrucciones de la fracción de borrado cambiaron

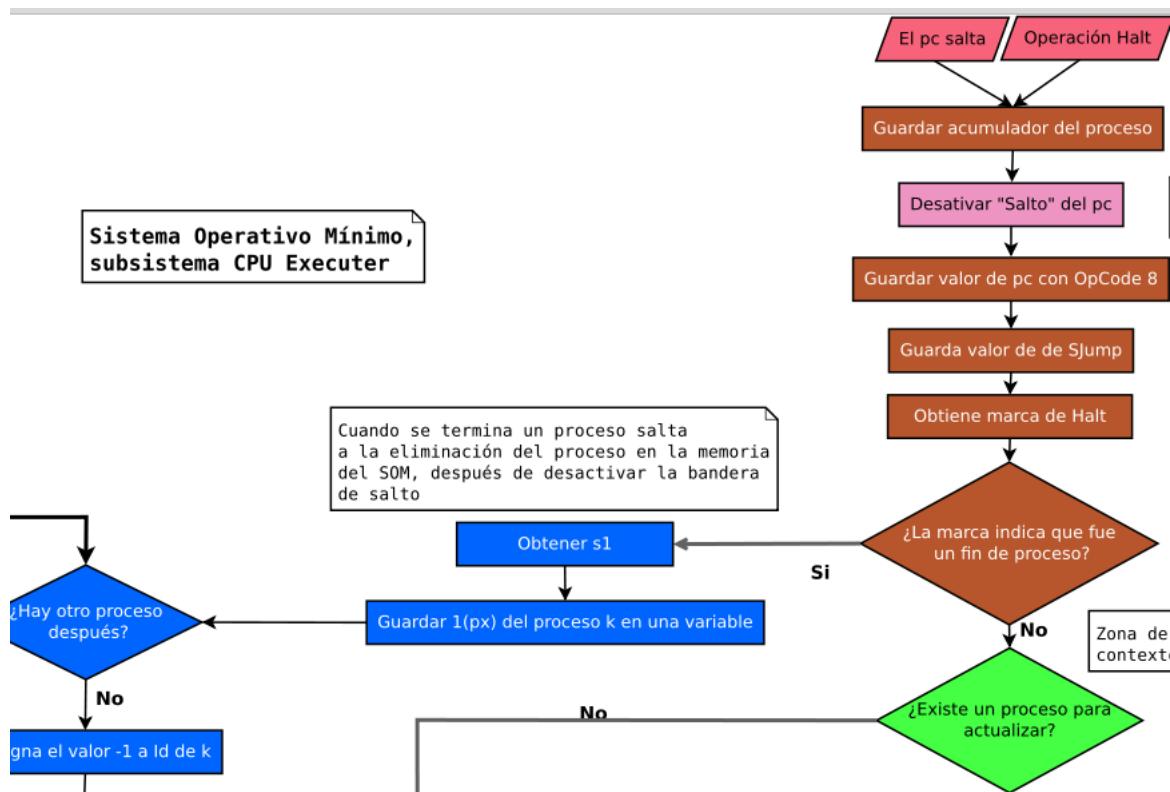


Figura 3.66: Acercamiento al preámbulo del sistema operativo mínimo

las posiciones para ir ahí, que justamente es la fracción que más redujo su tamaño respecto a la que estaba para la versión anterior.

Ahora, si recordamos, el sistema operativo no avanza nada hasta que el *vigilante* 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), dónde está resaltado en rosa un recuadro indicando que continuará el flujo hasta que el ID contador general sea distinto de cero, el rosa es para indicar que es una especie de entrada al sistema, por qué es un punto en el que el sistema puede continuar su ejecución.

Como podemos ver el flujo que continúa 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 fracción de “control de procesos”. En esta fracción lo que hace es que si ya no hay procesos,  $ID\ contador\ general == 0$ , reinicia los organizadores y el identificador estático que se usa para las impresiones antes de activar al *vigilante*. 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.

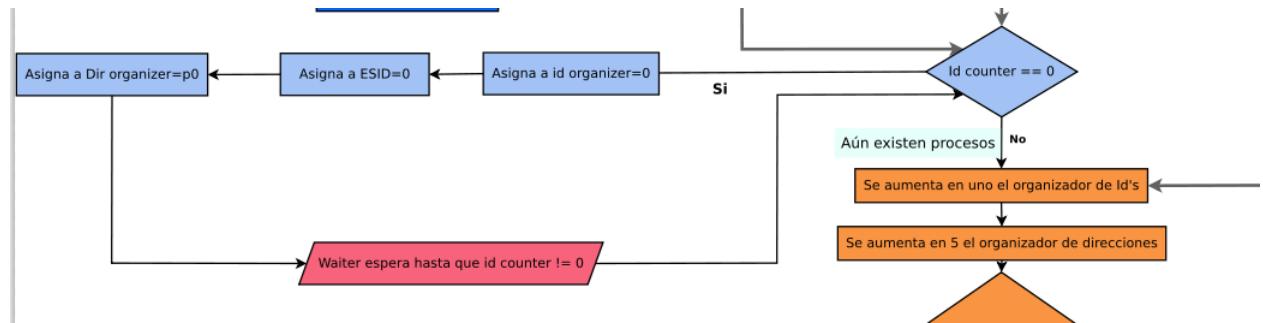


Figura 3.67: Acercamiento a segmento de *waiter* y control de procesos

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 waiter

Figura 3.68: Segmento de control de procesos

La fracción 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 el contexto del proceso, la coloca en dónde el proceso la pueda tomar, activa el salto del *conmutador*, y salta al proceso del usuario. 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.

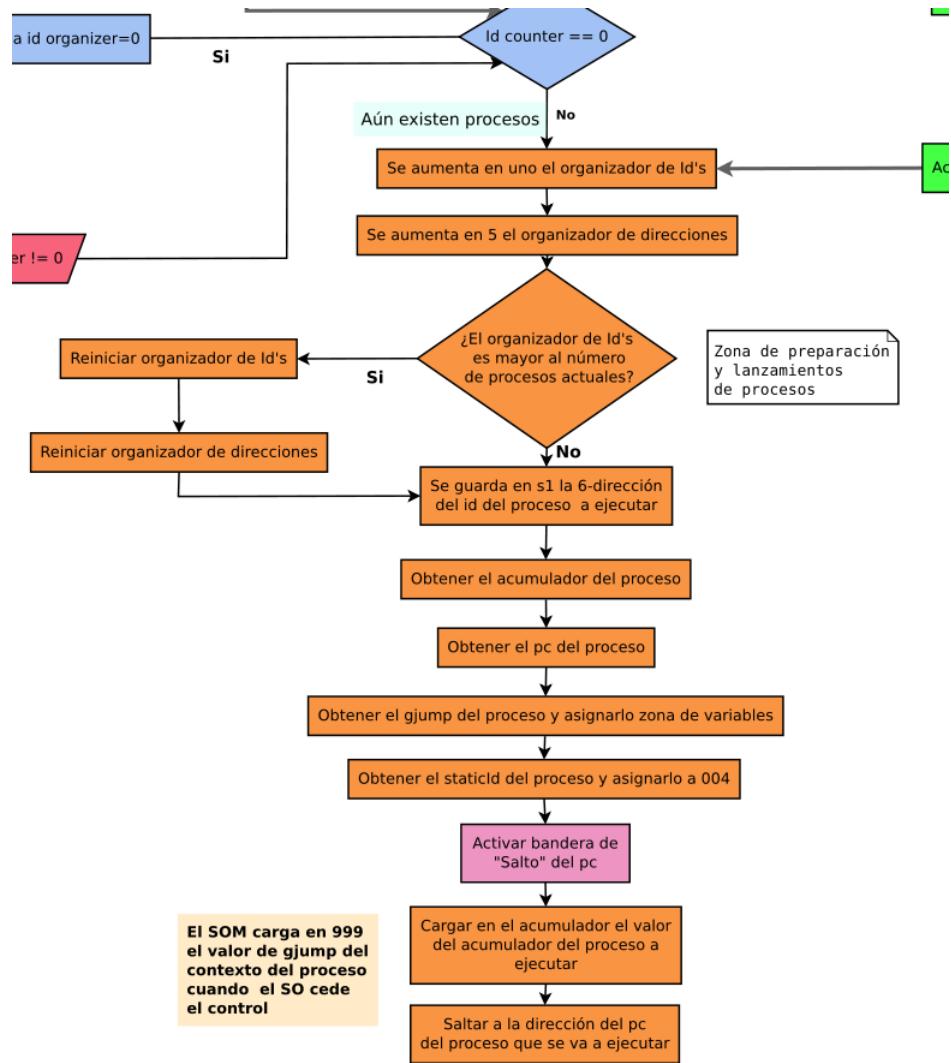


Figura 3.69: Acercamiento a fracción de lanzamiento de procesos

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
Aumentar Id organizer M[c7]++	s101	1(c7)	LDA c7	
	s102	2000	ADD 000	
	s103	6(c7)	STO c7	Se aumenta el id organizer
Aumentar Dir organizer M[c6]++=5	s104	1(c6)	LDA c6	
	s105	2(c11)	ADD c11	
	s106	6(c6)	STO c6	Se aumenta el dir organizer
Verifica si hay que reiniciar	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
Sino-s109	s110	1(c6)	LDA c6	Se obtiene la dir del id del proceso a ejecutar
	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"
Actualizar s1 para salir al proceso con su prioridad	s113	1(c6)	LDA c6	
	s114	2(c8)	ADD c8	Se convierte en la 1-dir del id del proceso a ejecutar
	s115	2000	ADD 000	Se obtiene la 1-dir del gpc del proceso a ejecutar
Continuación	s116	6(s119)	STO s118	En s101 se guarda la 1-dir del gpc del proceso a ejecutar
	s117	2000	ADD 000	Se obtiene la 1-dir del gacc del proceso a ejecutar

Figura 3.70: Sistema operativo mínimo paralelo, fracción de lanzamiento

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
Preparación gpc y gcc	s118	6(s132)	STO s131	En 112 se guarda la 1-dir del gacc del proceso a ejecutar
	s119	1(px)+1	LDA px+1	Obtiene el gpc del proceso a ejecutar
	s120	6(s133)	STO s132	Guarda el gpc del proceso a ejecutar en s106
Preparación gjump	s121	1(s132)	LDA s131	Obtiene la 1 dir del gacc
	s122	2000	ADD 000	Obtiene la 1 dir del gjump
	s123	6(s128)	STO s127	Guarda la 1 dir del gjump en s126
Salvar Static ID en 004	s124	2000	ADD 000	Añadir un 1 para obtener el 1(px)+4, SID
	s125	6(s126)	STO s125	Guardar en la siguiente celda
	s126	1(px)+4	LDA px+4	Obtener Static ID del proceso
	s127	6004	STO 004	Guardar en 0004 para usarlo como identificador el proceso que se está ejecutando
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 Bandera	s130	1(000)	LDA 000	Obtiene el número 1
	s131	6003	STO 003	Se permiten saltos con bandera==1
LastDirectionS*	s132	1(px)+2	LDA px+2	Se obtiene acc= acc del proceso a ejecutar
	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
Reiniciar id organizer	s136	2(c8)	ADD c8	Se crea la 1-dir de inicio en el acumulador
	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
Regresar	s140	8(s110)	JMP s109	Regresar para lanzar el proceso

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

En cuanto a la **actualización de procesos**, que podemos ver en la parte derecha 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.

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

Figura 3.73: Sistema operativo mínimo actualización de procesos

Lo que sí tuvo un cambio importante es la **fracción de borrado**(fig. 3.74), que no cambia la forma de borrar, esas subrutinas se quedan sin cambios, elimina los valores del proceso a

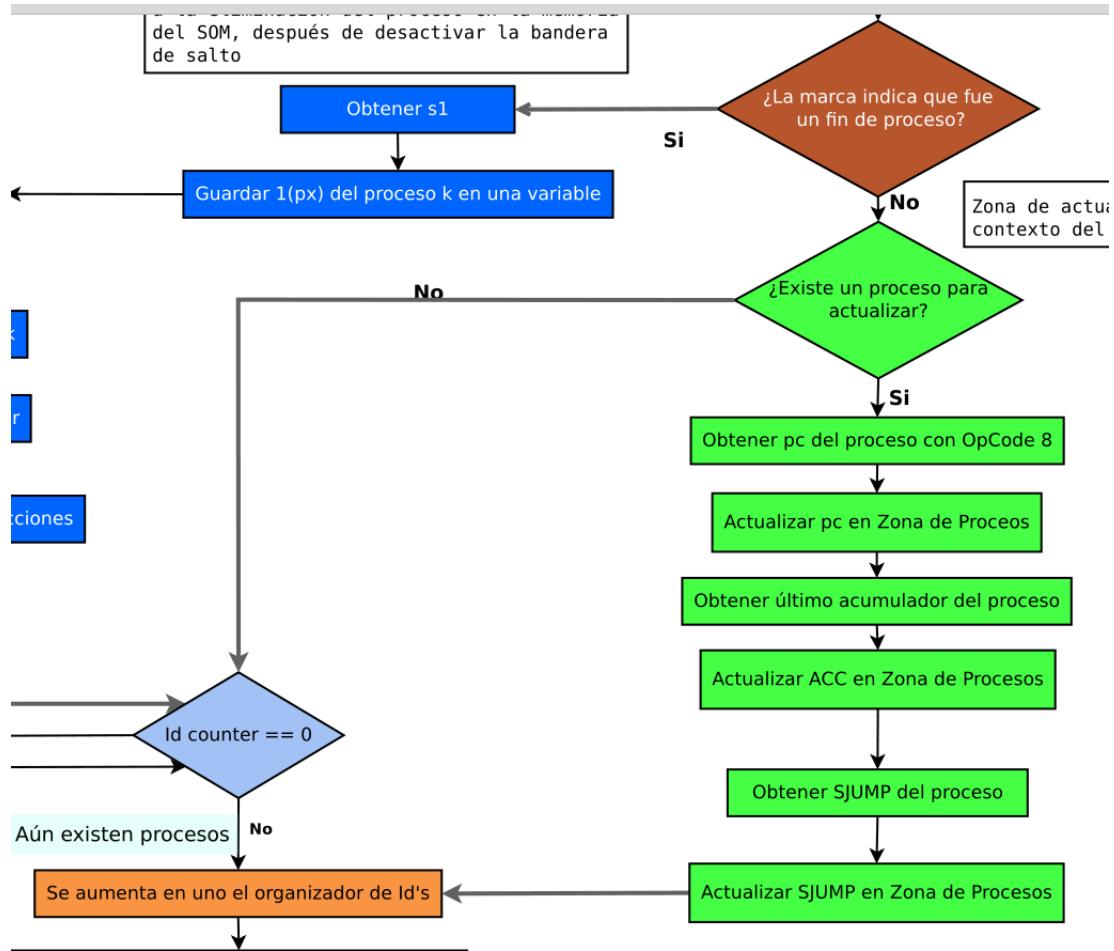


Figura 3.72: Acercamiento al segmento de actualización de procesos

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 última imagen, es que ya no se tiene un tratamiento especial para el proceso 0, como la CPU principal no tiene definida la instrucción *halt*, el proceso 0 no puede ser borrado, no hay forma de que sea enviado a esta fracción que opera la segunda CPU, y como ahora solo recibe procesos de usuario para ser borrados, solo requiere obtener su información y continuar con la ejecución. Razón por la que se dejaron vacías cinco celdas que ya no se ocupan. Se decidió no recorrer el programa para que no se tuvieran que editar el resto de fracciones por esta razón, así que el borrado es incluso más eficiente para la versión paralela del sistema operativo.

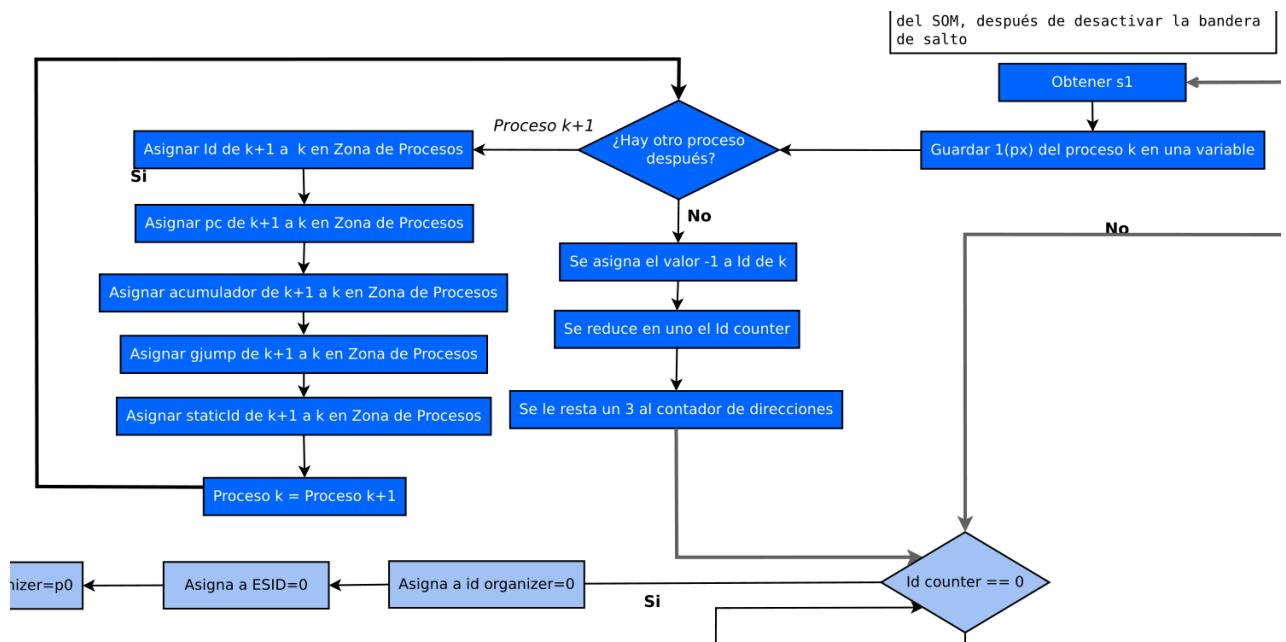


Figura 3.74: Acercamiento a la fracción de borrado

Sistema operativo				
Nombre clave	Dirección	Instrucción	Código	Descripción
	s19			
	s20			
	s21			
	s22			
	s23			
<i>Guardar 1(px)</i>	s24	1(s1)	SUB 000	Llamamos 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 termino 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
	s44	1(py)+u	LDA py+u	
	s45	6(px)+u	STO px+u	
	s46	1(c0)	LDA c0	Obtener contador de contextos
<i>Aumentar contador de secciones</i>	s47	7000	ADD 000	Se le resta uno al contador de contextos
	s48	6(c0)	STO c0	Lo guarda en c0 de nuevo
	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

Por lo que el flujo en general sería el siguiente, el usuario añade un proceso con ayuda de la primer CPU, una vez esté este cargado el *vigilante* permitirá avanzar al contador de programa de la segunda CPU, del ejecutor, para que añada el programa a la lista de procesos a ejecutar, y tal cual pasaba en C, si ya hay un proceso agregado a la zona de procesos el lanzador de procesos puede empezar a ejecutarlos. Una vez que empieza este ciclo, continúa 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 *vigilante* una vez que se hayan terminado los procesos y la segunda CPU regrese a un estado de espera.

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

Añadiremos dos procesos para ver el funcionamiento de la máquina, 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 cual se hacía en C, en la imagen 3.77 vemos que están en la cola de carga unas instrucciones y que la CPU principal está realizando operaciones, mientras que el segundo está en espera. En la figura 3.78 vemos el programa ya cargado de la dirección #204 hasta la #239, y una subrutina que utiliza el programa la podemos ver en la imagen 3.79. Además, vemos que el segundo CPU ya está en funcionamiento(en fig. 3.78), y de hecho está en la zona de lanzamiento de procesos, puesto que se encuentra en la dirección #907.

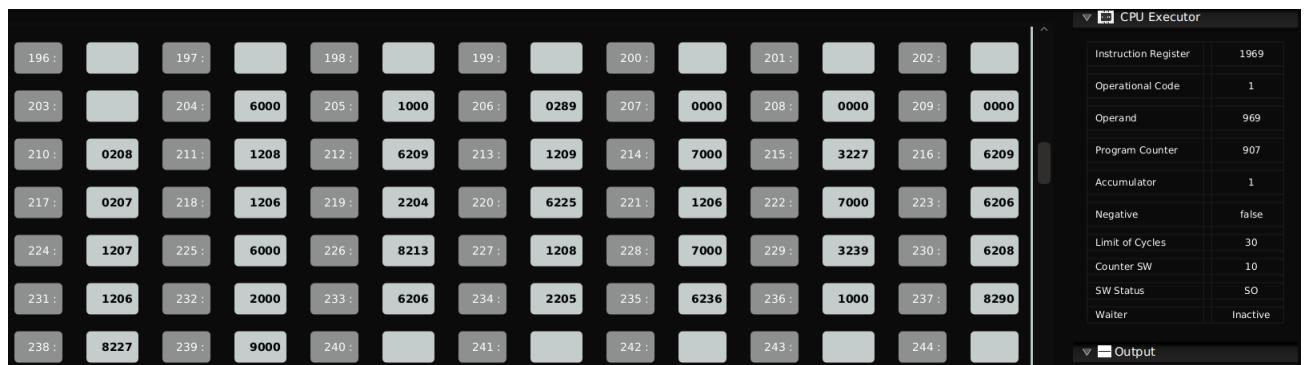


Figura 3.78: Proceso de Fibonacci cargado

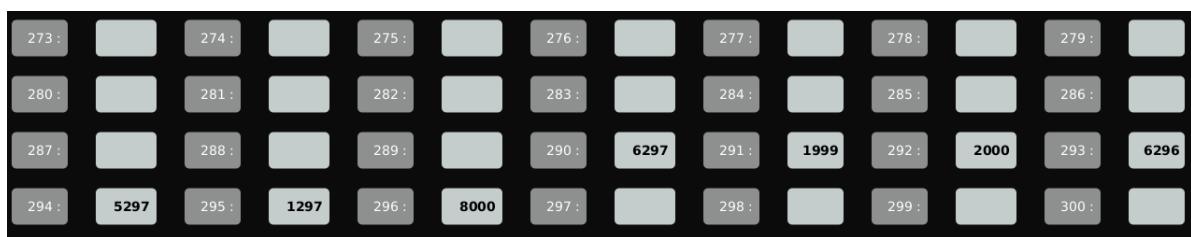


Figura 3.79: Subrutina de proceso de Fibonacci

Lo que sigue es añadir el siguiente proceso, en la imagen 3.80 vemos en el *deck* la tarjeta para añadirla a la cola. También vemos ahí que la segunda CPU ya ha lanzado el primer

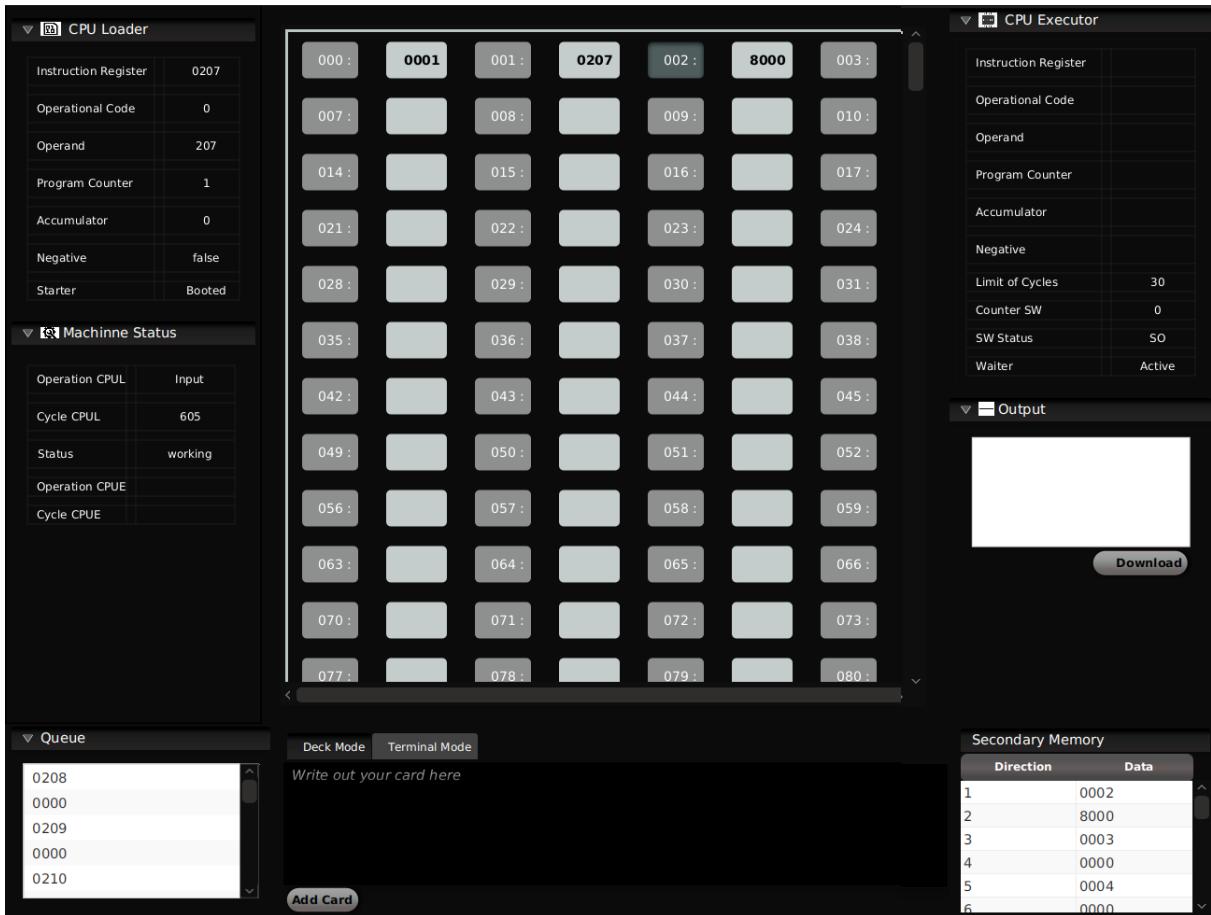


Figura 3.77: Añadiendo un nuevo proceso en E-CARDIAC PC

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 ejecutor no es verde, sino naranja/rojiza, para diferenciar fácilmente que CPU está usando determinada zona de la 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.

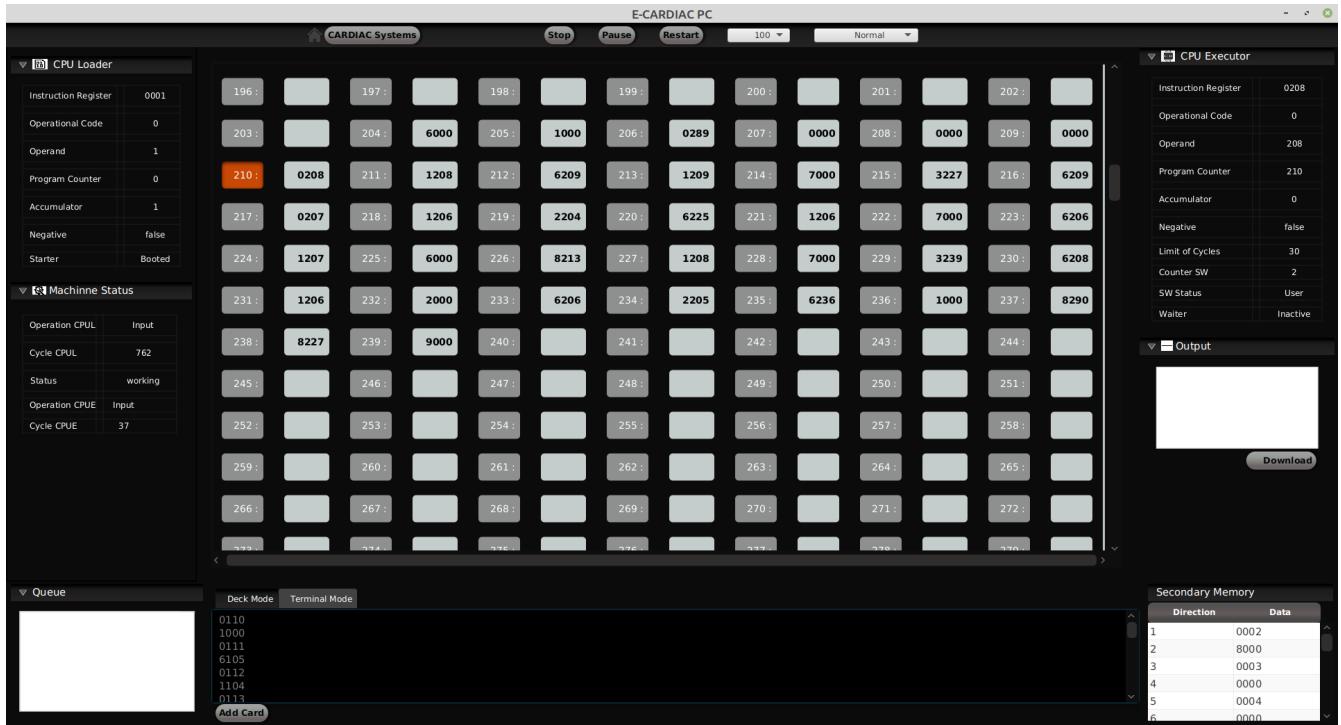


Figura 3.80: Programa pintor en el *deck*

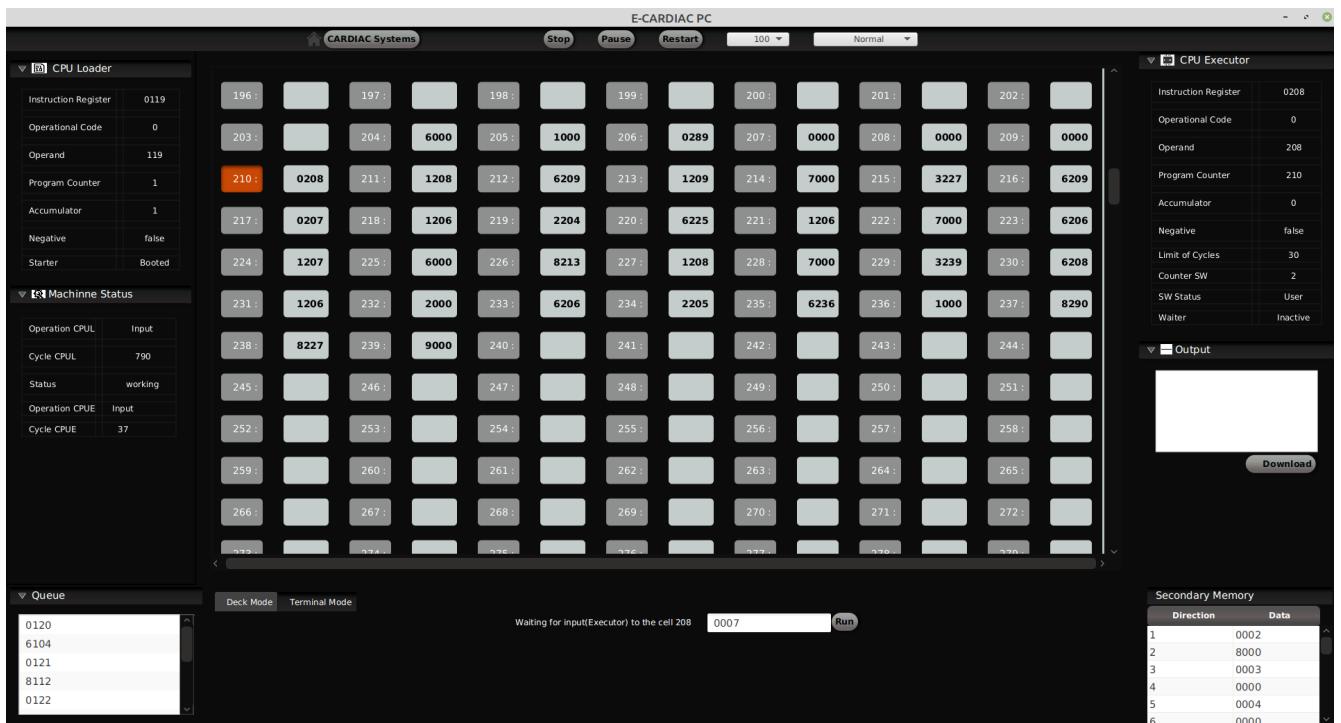


Figura 3.81: Agregando cantidad de números para serie de Fibonacci

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

*pintor*, y que el ejecutor está en el sistema operativo actualizando uno de los procesos. Ya en la imagen 3.83 vemos que la ejecución termino, terminando al final el proceso 1, y con el *vigilante/waiter* activado porque el ejecutor al no haber más procesos se encuentra en estado de espera, mientras que el principal está listo para continuar agregando procesos.

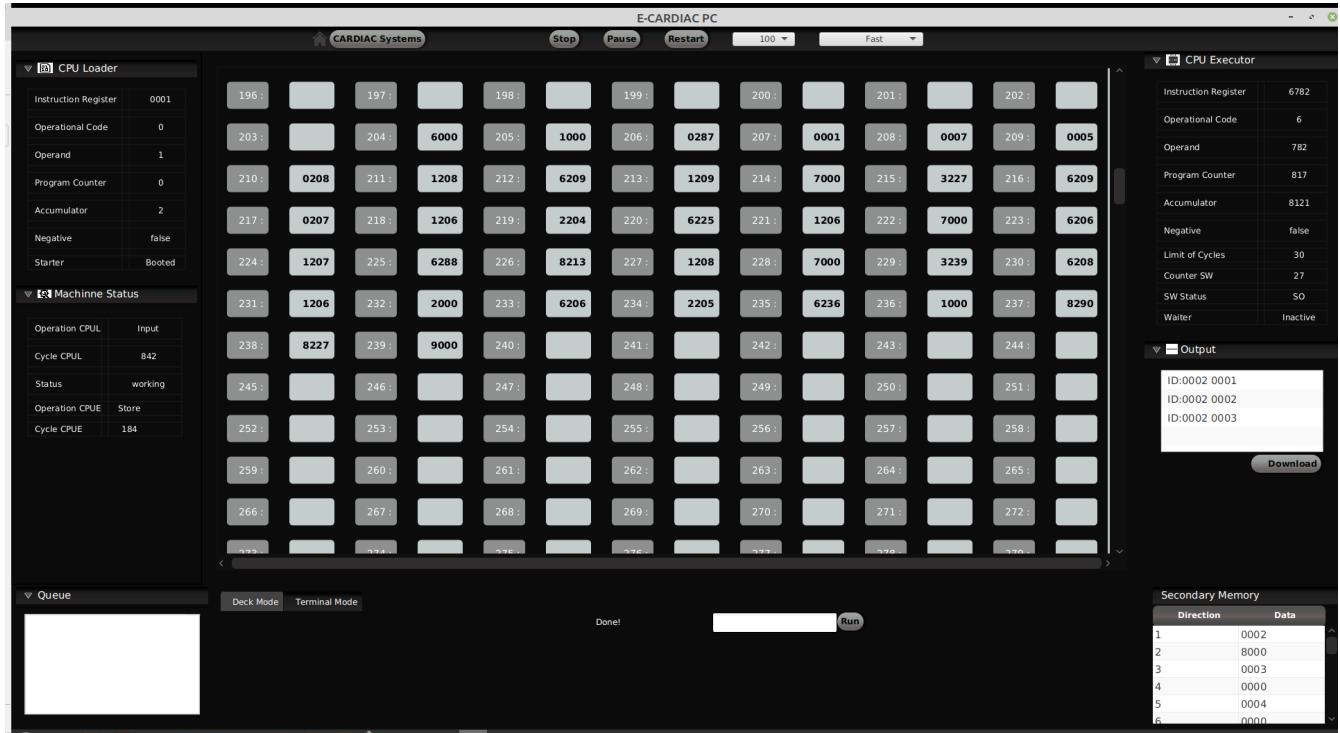


Figura 3.82: Primeras salidas de “pintor”

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*.

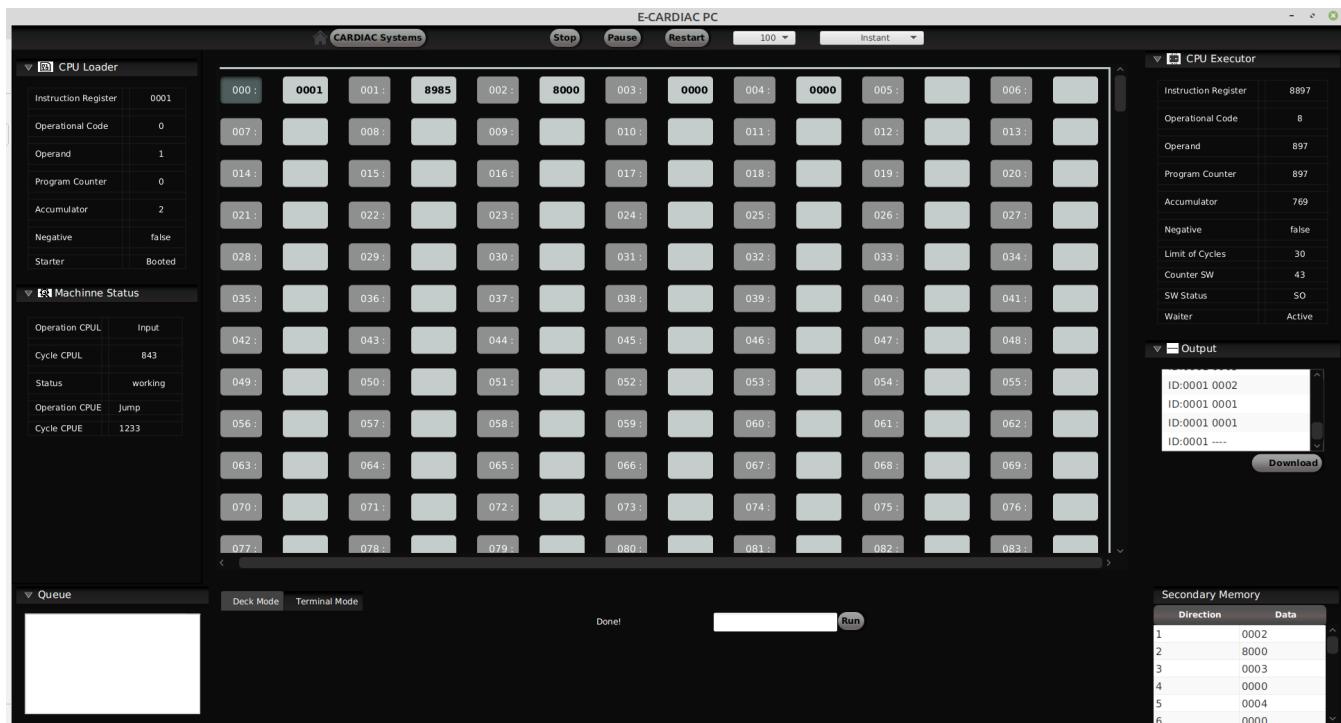


Figura 3.83: Ejecución finalizada

```
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 máquina 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 esta instrucción final agregar otro par donde 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 esta 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, la *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* solo es entrada de información para la 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 qué proceso pertenece(según su identificador estático). Si un proceso finaliza se imprimirá el identificador seguido de varios guiones medios.
10. Una vez terminada la ejecución de todos los procesos, el *vigilaten/waiter* se vuelve a activar para dejar a la segunda CPU en espera.
11. La CPU principal en paralelo puede seguir añadiendo procesos.

Con la guía podemos notar que salvo unos puntos, lo demás sigue la misma estructura, porque 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 cómo 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 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.

# Bibliografía

- [1] NASA, *Who Was Katherine Johnson? (Grades K-4)* - NASA, en-US, Section: For Kids and Students, feb. de 2020. dirección: <https://www.nasa.gov/learning-resources/for-kids-and-students/who-was-katherine-johnson-grades-k-4/> (visitado 26-10-2023).
- [2] G. Ifrah, *The universal history of computing: from the abacus to the quantum computer*, eng. New York: John Wiley, 2001, ISBN: 978-0-471-39671-0.
- [3] S. Fingerman y D. Hagelbarger, *An instruction manual for cardiac*, English, abr. de 1968. dirección: [https://www.cs.drexel.edu/~bls96/museum/CARDIAC\\_manual.pdf](https://www.cs.drexel.edu/~bls96/museum/CARDIAC_manual.pdf).
- [4] Mark Jones Lorenzo, *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)*, English. Createspace Independent Publishing Platform, ago. de 2017, ISBN: 978-1-5374-2113-1.
- [5] L. Null, *The Essentials of Computer Organization and Architecture*, en. Jones & Bartlett Learning, 2003, Google-Books-ID: c2K1EAAAQBAJ, ISBN: 978-1-284-28463-8.
- [6] Álvaro Frías, “Retruco: un intérprete para TIMBA,” es, *Electronic Journal of SADIO*, vol. vol. 21, no. 1, jun. de 2022, ISSN: 1514-6774. dirección: <http://sedici.unlp.edu.ar/handle/10915/142866> (visitado 09-05-2023).
- [7] M. Ajdari y M. Tabandeh, “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)*, sep. de 2012, págs. 1-5. DOI: 10.1109/ICL.2012.6402055.

- [8] G. O'Regan, *A brief history of computing*, eng, 2. ed. London Heidelberg: Springer, 2012, ISBN: 978-1-4471-2358-3.
- [9] California State University, *The Historical Development of Computing*. dirección: <https://home.csulb.edu/~cwallis/labs/computability/index.html#Final> (visitado 11-10-2023).
- [10] E. Eric Kim y B. Alexandra Toole, “Ada and the First Computer,” en, *Scientific American*, vol. 280, n.º 5, págs. 76-81, mayo de 1999, ISSN: 0036-8733. DOI: 10.1038/scientificamerican0599-76. dirección: <https://www.scientificamerican.com/article/ada-and-the-first-computer> (visitado 12-10-2023).
- [11] Museo Torres Quevedo, *El ajedrecista, el primer juego de ordenador de la historia*, es. dirección: <https://artsandculture.google.com/story/el-ajedrecista-el-primer-juego-de-ordenador-de-la-historia/CQURBJHKlccLIA> (visitado 30-07-2023).
- [12] H. H. Goldstine, *The computer from Pascal to von Neumann*. Princeton, N.J.: Princeton University Press, 1972, ISBN: 978-0-691-08104-5.
- [13] A. Amt, *La generación Z: Konrad Zuse, pionero alemán de la computación*, es. dirección: <https://alemaniaparati.diplo.de/mxdz-es/aktuelles/konradzuse/1087764> (visitado 02-05-2024).
- [14] Computer History Museum, *Computers / Timeline of Computer History*. dirección: <https://www.computerhistory.org/timeline/computers/#169ebbe2ad45559efbc6eb3572083fb> (visitado 30-09-2023).
- [15] R. Pawson, “The Myth of the Harvard Architecture,” *IEEE Annals of the History of Computing*, vol. 44, n.º 3, págs. 59-69, jul. de 2022, Conference Name: IEEE Annals of the History of Computing, ISSN: 1934-1547. DOI: 10.1109/MAHC.2022.3175612. dirección: <https://ieeexplore.ieee.org/document/9779481/metrics#metrics> (visitado 18-10-2023).

- [16] A. W. Burks, H. H. Goldstine y J. Neumann, “Preliminary Discussion of the Logical Design of an Electronic Computing Instrument,” en, en *The Origins of Digital Computers*, B. Randell, ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, págs. 399-413, ISBN: 978-3-642-61814-7 978-3-642-61812-3. DOI: 10.1007/978-3-642-61812-3\_32. dirección: [http://link.springer.com/10.1007/978-3-642-61812-3\\_32](http://link.springer.com/10.1007/978-3-642-61812-3_32) (visitado 19-10-2023).
- [17] A. S. Tanenbaum, *Modern operating systems*, English. 2002, OCLC: 981051666, ISBN: 978-0-13-031358-4 978-7-111-09156-1.
- [18] M. Sipser, *Introduction to the theory of computation*, eng, Third edition, international edition. Australia Brazil Japan Korea Mexiko Singapore Spain United Kingdom United States: Cengage Learning, 2013, ISBN: 978-1-133-18781-3 978-1-133-18779-0 978-0-357-67058-3.
- [19] D. Salomon, *Assemblers and loaders*, ép. Ellis Horwood series in computers and their applications. New York: Ellis Horwood, 1992, ISBN: 978-0-13-052564-2.
- [20] Maurice Vincent Wilkes, *EDSAC 1951*, 1976. dirección: <https://www.youtube.com/watch?v=6v4Juzn10gM> (visitado 14-09-2023).
- [21] M. Richards, “EDSAC Initial Orders and Squares Program,” en, *University of Cambridge*, dirección: <https://www.cl.cam.ac.uk/~mr10/Edsac/edsacposter.pdf>.
- [22] A. Silberschatz, P. B. Galvin y G. Gagne, *Operating system concepts*, 8th ed. Hoboken, NJ: J. Wiley & Sons, 2009, ISBN: 978-0-470-27993-9 978-0-470-12872-5.
- [23] L. Null y J. Lobur, “MarieSim: The MARIE computer simulator,” *Journal on Educational Resources in Computing*, vol. 3, n.º 2, 1-es, jun. de 2003, ISSN: 1531-4278. DOI: 10.1145/982753.982754. dirección: <https://doi.org/10.1145/982753.982754> (visitado 09-05-2023).
- [24] R. M. Hord, *The Illiac IV: the First Supercomputer*, eng. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, OCLC: 851370561, ISBN: 978-3-662-10345-6.

- [25] P. E. Ceruzzi, *Computing: a concise history*, ép. The MIT Press essential knowledge series. Cambridge, Mass: MIT Press, 2012, OCLC: ocn758392163, ISBN: 978-0-262-51767-6.
- [26] W. Isaacson, *The innovators: how a group of hackers, geniuses, and geeks created the digital revolution*, First Simon & Schuster hardcover edition. New York: Simon & Schuster, 2014, ISBN: 978-1-4767-0869-0 978-1-4767-0870-6.
- [27] AT&T Tech Channel, *The Transistor: a 1953 documentary, anticipating its coming impact on technology*, 2015. dirección: <https://www.youtube.com/watch?v=V9xUQWo4vN0> (visitado 18-08-2023).
- [28] ——, *AT&T Archives: The Thinking Machines (Bonus Edition)*, mar. de 2012. dirección: <https://www.youtube.com/watch?v=clud9I18DXU> (visitado 17-08-2023).
- [29] megardi, *CARDIAC (CARDboard Illustrative Aid to Computation) Replica*, en. dirección: <https://www.instructables.com/CARDIAC-CARDboard-Illustrative-Aid-to-Computation-/> (visitado 17-08-2023).
- [30] J. W. Valvano y J. W. Valvano, *Introduction to the Arm® Cortex(TM)-M Microcontrollers*, eng, Fifth Edition, ép. Embedded systems / Jonathan W. Valvano 1. s.l.: Eigenverl. d. Verf, 2017, ISBN: 978-1-4775-0899-2.
- [31] A. S. Tanenbaum, T. Austin y B. R. Chandavarkar, *Structured computer organization*, eng, Sixth edition, international edition, ép. Always learning. Boston Columbus Indianapolis New York San Francisco Upper saddle River Amsterdam Cape Town Dubai London Madrid Milan Munich: Pearson, 2013, ISBN: 978-0-273-76924-8.
- [32] W. Aspray, ed., *Papers of John von Neumann on computing and computer theory*, ép. Charles Babbage Institute reprint series for the history of computing v. 12. Cambridge, Mass. : Los Angeles: MIT Press ; Tomash Publishers, 1987, ISBN: 978-0-262-22030-9.
- [33] Leslie Lamport, “The Computer Science of Concurrency: The Early Years,” *Communications of the ACM*, págs. 71-76, jun. de 2015. dirección: <https://cacm.acm.org>.

org/magazines/2015/6/187316-turing-lecture-the-computer-science-of-concurrency/abstract.

- [34] Gadi Taubenfeld, *Concurrent Programming, Mutual Exclusion* (1965; Dijkstra), English. dirección: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=30e83735eb72af97e7ab3ec7f0823b9a9ae5493c>.

# Historia de la computación

# Historia de la computación

2700 BCE

## ◀ Creación del ábaco



Se crea una especie de **ábaco** en Sumeria de acuerdo a la numeración que tenían.

1614



## ◀ Logaritmos

Jhon Napier descubre los **logaritmos** y al poco tiempo crea tablas para recordarlos más eficientemente.

1620



## ◀ Gunter's Scale

Edmund Gunter inventa la "escala de Gunter", un dispositivo de medición avanzado para su época.

1623



## ◀ Slide Rule

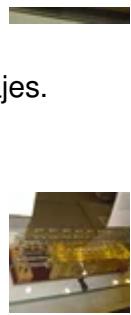
William Oughtred mejora el invento de Gunter y crea la "regla de cálculo", que es incluso usada en la ingeniería actual para ciertos cálculos.

1642



## Pascaline

Blaise Pascal inventa la **primer** maquina de calculo aritmético a base de ruedas y engranajes.



1673

## Stepped Reckoner

Gottfried Leibniz construye una calculadora mecánica llamada "Maquina de Leibniz" que podía realizar las 4 operaciones básicas de manera matemática.

1801

## Telar de Jacquard

Joseph Marie Jacquard inventa en 1804 un telar "programable" por tarjetas perforadas que podían definir patrones.



1820

## Arithmometre

Charles Xavier Thomas de Colmar inventa una maquina calculadora a parir de la maquina construida por Gottfried Leibniz, el **Aritmómetro**.



1830

## Difference Engine

Charle Babbage presenta un prototipo de lo que sería su maquina diferencial.



1841

## Analytical Engine

Ada Lovelace presenta sus "Notas" dónde describe el funcionamiento la nueva maquina de Charles Babbage, que sería programable y automática.



1890

## Tabulating Machine

Herman Hollerith desarrolla una maquina con tarjetas perforadas para procesar la información de los censos en EEUU.



1913

◀ **aritmómetro electromecánico**

Leonardo Torres de Quevedo crea al precursor de la calculadora digital moderna.



1914

◀ **El Ajedrecista**

Leonardo Torres Quevedo inventa la primera versión del "ajedrecista" y establece las bases de la automática.



1931

◀ **Differential Analyser**

Fue construida por Harlod Locke Hazen y Vannevar Bush en el MIT, capaz de resolver ecuaciones diferenciales, usado especialmente en problemas de ingeniería y física



1936

◀ **Formalización del computo**

Alan Turing y Alonzo Church presentan sus tesis formalizando la teoría del computo, iniciando la teoría de la computación.



1939

◀ **Complex Number Calculator**

Samuel Williams y George Stibitz desarrollan en los laboratorios Bell una calculadora que puede trabajar con números complejos y que es electromecánica.



1941

◀ **Z3 Computer**

Konrad Zuse termina la Z3, primer computadora, completamente automática y programable mediante tarjetas perforadas



1942



## Atanasoff-Berry Computer (ABC)

El profesor John Vincent Atanasoff y su estudiante Clifford Berry construyeron en el colegio de Iowa la primer computadora creada en EEUU.



1944

### Harvard Mark I

Concebida por el profesor de Harvard Howard Aiken y construida por IBM era una calculadora basada en relés con mucha exactitud.



1944

### Colossus

La maquina creada por Tommy flowers es puesta en operación para descifrar el código Lorenz. Usaba tubos de vacío y era programable.



1945

### ENIAC

Computadora construida por John Mauchly y J. Presper Eckert para e realizar cálculos de artillería. Era programable mediante el cambio de interruptores, aunque era muy lento hacerlo.



1948

### Manchester Mark 1

Computadora creada a partir de la "Baby" de Manchester por Tom Kilburn y Frederick Williams. Era completamente electrónica, digital y con programas almacenados



1949

### EDSAC

Creada por Maurice Wilkes en la universidad de Cambridge, era una computadora con programas almacenados y seguía los principios de la arquitectura Von Neumann.



1949

### BINAC

Computadora que dejaba los decimales por los números binarios creada por Eckert y

Mauchly.

1949

### CSIRAC The Australian Computer



La CSIRAC fue la primera computadora digital creada en Australia, que además fue la primera en el mundo capaz de reproducir música.

1951

### EDVAC



Construida por John Von Neumann, J. Eckert y Mauchly tenía varias mejoras respecto a ENIAC, como el hecho de que era totalmente eléctrica.

1951

### Ferranti Mark 1



Evolución de la Manchester Mark 1 para ser comercializada por Ferranti Ltd.

1951

### Whirlwind



Computadora desarrollada por los laboratorios del MIT para la U.S. Navy. Es recordada por ser la primera computadora de respuesta en "tiempo real" para los usuarios.

1951

### UNIVAC



Primer computadora comercial creada en EEUU, diseñada por J. Presper Eckert y John William Mauchly como evolución de BINAC

1952

### IAS Machine



Fue la primera computadora electrónica construida por el instituto de estudios avanzados de Princeton, y fue supervisada por el mismo John Von Neumann.



1954



## IBM 650

Computadora digital desarrollada por IBM, fue la computadora más popular de los años 50.

1959



## PDP-1

Un computador desarrollado por Digital Equipment Corporation, y es famoso por ser la computadora con la que inicio la cultura "hacker" en el MIT.

1961



## IBM 7030 Stretch

Fue el primer ordenador que usaba transistores de IBM. Originalmente tenía un precio de 13,5 millones de dólares.

1964



## IBM System/360

Es una familia de computadores que se empezaron a liberar en 1964. Que tenían como característica principal el compartir un mismo sistema operativo.



1965

## GE 645

AT&T y General Electric desarrollan una computadora para probar el poder del "time sharing" o "tiempo compartido"

1968



## Apollo Guidance Computer (AGC)

Computadora creada para la nave Apollo por el equipo del MIT. La misión era hacerla lo más pequeña posible para ahorrar espacio en la nave.



1968

## PDP-8

Minicomputador creado por Digital Equipment Corporation, fue la primer minicomputadora comercialmente exitosa y tenía un costo inicial de 18,000, dólares.

1972

### ILLIAC IV



Se pone en marcha la "primer" supercomputadora, contaba con un procesamiento paralelo y estuvo largos años en desarrollo.

1973

### IBM SCAMP



Desarrollada bajo la dirección de Paul Friedl, sería el punto de partida para la familia 5100, que buscaban ser más "personales".

1975

### Altair 8080



Primer computadora que puede llamarse "personal" por lo pequeña que es que es exitosa en el mercado. Diseñada por MITS fue el inicio de la revolución de las computadoras personales.

1977

### Apple II



Quizá la computadora más famosa de la historia, creada por Steve Wozniak y Steve Jobs para dar otro avance gigante en las computadoras personales

1981

### IBM PC



La primer computadora personal desarrollada por IBM, formalmente llamada IBM 5150, parte de la familia de computadoras 5100 de IBM. Revolucionó las computadoras de negocios y fue ampliamente "clonada".

2022

### HP Pavilion



Una de las computadoras más recientes de HP, con características medianas de acuerdo al mercado. 16 GB de RAM, 512 GB de almacenamiento interno y con el sistema operativo Windows 11 incluido.

|

# Sistema Operativo Concurrente para E-CARDIAC

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>Nuevo</i>	s0	-1		-1 Es su valor por defecto, el usuario siempre agregará una 8-dir
	s1	6000		Contiene la 6-dir del id del proceso actual, controlado por el lanzador, 6(px)
	s2	1(s1)	LDA s1	El acumulador toma un valor de la forma 6(px)
	s3	3(s98)	BLZ s97	Si acc<0 no hay proceso para actualizar
	s4	2000	ADD 000	Se obtiene la 6-dir del gpc del proceso
	s5	6(s7)	STO s7	Se guarda la instrucción 6(px)+1
	s6	1(c1)	LDA C1	Se obtiene el último pc del proceso con forma 8(pc)
	s7	6(p5)	STO p5	En p5 se actualiza gpc
	s8	1(s7)	LDA s7	Se obtiene la instrucción 6(px)+1
	s9	2000	ADD 000	Para acceder a la 6-dir del gacc del proceso
<i>Actualizar gpc</i>	s10	6(s12)	STO s12	En s12 se guarda 6(px)+2
	s11	1(c2)	LDA c2	Se obtiene el último acc del proceso
	s12	6(p6)	STO p6	Se actualiza el valor de gacc
	s13	1(s12)	LDA s12	Obtiene la 6(p6) del proceso que se está actualizando
	s14	2000	ADD 000	Obtiene la 6(p7) del proceso que se está ejecutando
	s15	6(s17)	STO s17	Guarda en e18 el código para guardar en la zona de procesos c14
	s16	1(c14)	LDA c14	Obtiene c14
	s17	6(p7)	STO p7	Guarda en la zona de procesos correspondiente al proceso el valor saber jump
	s18	8(s101)	JMP S100	Saltamos a cambiar de proceso
	s19	1(s1)	LDA s1	Llamemos k al proceso asociado a la dir px
<i>Guardar l(px)</i>	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 l(px)
	s23	1e7	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)
	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 l(py)
	s29	1(py)	LDA py	Se obtiene el id del proceso (k+1)
<i>Verificar si es el proceso 0</i>	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
	s32	6(c17)	STO c17	Se guarda en c17 para usarla más tarde
	s33	1(c18)	LDA 004	Se carga 0004 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
	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
<i>Recorrer las secciones gpc,gacc,gjump,staticid de py a px</i>	s40	7e11	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	
	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
	s49	8(s35)	JMP s35	
<i>Aumentar contador de secciones</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
	"Si" de s30	s52	1(s29)	LDA s29

# Sistema Operativo Concurrente

Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
Borrado	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
	s59	1(c4)	LDA c4	Carga el id counter
	s60	7(000)	SUB 000	Se le resta 1 al id counter
M[c4]--	s61	6(c4)	STO c4	Guardar en c4 el nuevo valor
	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
M[c5]=M[c5]-5	s65	8(s98)	JMP S97	Salta a ver si el id counter es el ultimo
	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
	s68	1(c4)	LDA c4	Se obtiene el valor del Id counter
Aumenta el Id counter M[c4]++	s69	2000	ADD 000	
	s70	6(c4)	STO c4	
	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
Aumenta el Dir counter M[c5]+=3	s73	6(c5)	STO c5	
	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)
	s76	2000	ADD 000	Se obtiene la 6-dir del gpc del nuevo proceso
Previa de ID	s77	6(s89)	STO s78	En s72 se guarda 6(px)+1, la 6-dir del gpc del nuevo proceso
	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
	s80	2(c12)	ADD C12	Añade 2 para obtener 6(px)+4(Static ID)
Previa de pc	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
Guardar Nuevo Static ID para el proceso	s85	6(px)+4	STO (px)+4	Guarda el nuevo serial Static ID para el 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
	s88	1(s)	LDA s	Se obtiene el 8-pc del proceso
pc nuevo	s89	6(px)+1	STO px+1	Se guarda el 8-pc en el gpc del 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
	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
¿está en el proceso 0?	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
	s99	7(000)	SUB 000	Se le resta un 1 , acc=-1
	s100	3(s141)	BLZ s140	Si acc<0 salta al proceso 0
¿Se acabaron los programas?	s101	1(c7)	LDA c7	
	s102	2000	ADD 000	
	s103	6(c7)	STO c7	Se aumenta el id organizer
	s104	1(c6)	LDA c6	
Aumentar Id organizer M[c7]++	s105	2(c11)	ADD c11	
	s106	6(c6)	STO c6	Se aumenta el dir organizer

# Sistema Operativo Concurrente

Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
Verifica si hay que reiniciar	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
Sino-s109	s110	1(c6)	LDA c6	Se obtiene la dir del id del proceso a ejecutar
	s111	2(c9)	ADD c9	Se crea la 6-dir del id del proceso a ejecutar
Actualizar s1	s112	6(s1)	STO s1	Se guarda en s1, será el proceso "actual"
	s113	1(c6)	LDA c6	
Preparación para saltar al proceso con su pc y acc correctos	s114	2(c8)	ADD c8	Se convierte en la 1-dir del id del proceso a ejecutar
	s115	2000	ADD 000	Se obtiene la 1-dir del gpc del proceso a ejecutar
	s116	6(s119)	STO s118	En s101 se guarda la 1-dir del gpc del proceso a ejecutar
Continuación	s117	2000	ADD 000	Se obtiene la 1-dir del gacc del proceso a ejecutar
	s118	6(s132)	STO s131	En s112 se guarda la 1-dir del gacc del proceso a ejecutar
Preparación gpc y gcc	s119	1(px)+1	LDA px+1	Obtiene el gpc del proceso a ejecutar
	s120	6(s133)	STO s132	Guarda el gpc del proceso a ejecutar en s106
Preparación gjump	s121	1(s132)	LDA s131	Obtiene la 1 dir del gacc
	s122	2000	ADD 000	Obtiene la 1 dir del gJump
Salvar Static ID en 004	s123	6(s128)	STO s127	Guarda la 1 dir del gJump en s126
	s124	2000	ADD 000	Añadir un 1 para obtener el 1(px)+4, SID
Salvar gjump	s125	6(s126)	STO s125	Guardar en la siguiente celda
	s126	1(px)+4	LDA px+4	Obtener Static ID del proceso
bandera	s127	6004	STO 004	Guardar en 0004 para usarlo como identificador el proceso que se está ejecutando
	s128	1(px)+3	LDA px+3	Carga el valor de la gJump
Bandera	s129	6(c14)	STO c14	Guarda el gJump en c14 para que la arquitectura lo intercambie
	s130	1(000)	LDA 000	Obtiene el número 1
LastDirectionSO	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
Reiniciar dir organizer	s133	8(xx)	JMP xx	La dir xx representa el valor donde el proceso se ejecutará
	s134	1(c3)	LDA c3	Se obtiene la primera dir de la zona de procesos, p3
Reinic平ar id organizer	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
Asigna a 004 el 0	s137	6(s138)	STO s137	En s138 se guarda px
	s138	1(px)	LDA px	Se obtiene el id de inicio
Asigna a 004 el 0	s139	6(c7)	STO c7	Se reinicia el id organizer
	s140	8(s110)	JMP s110	Regresar para lanzar el proceso
Reinic平ar id organizer a 0	s141	1(000)	LDA 000	
	s142	7000	SUB 000	Carga el id=0 del proceso 0
Asigna a 004 el 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
Asigna a dir organizer=p0	s147	6(c6)	STO c6	Asignar a dir organizer=p0
	s148	8(000)	JMP 000	Saltar a proceso 0

## 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
Mandar pc a C	e5	1(e0)	LDA e0	Carga en el acumulador el último pc del proceso antes de saltar
	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

# Sistema Operativo Concurrente

Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<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
	e12	8(s2)	JMP s2	Si la marca no es menor a 1 se va al área de actualización
	e13	8(s19)	JMP s19	Salta al área de borrado
	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
<i>Preámbulo para añadir procesos</i>	e18			
<b>Zona de Procesos</b>				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>Id Counter</i>	p0	0		Id del primer programa
<i>gpc</i>	p1	8000		Es el pc del proceso 0 con op-code 8
<i>gacc</i>	p2	0		Es el acumulador del proceso
<i>giump</i>	p3	8000		Contiene el valor del que se tenía en la última dirección de memoria antes de saltar
<i>Static ID</i>	p4	0		Es el ID Static del proceso que no va a cambiar
<i>Id Counter</i>	p5	-1		Es el id del proceso correspondiente a está sección
<i>gpc</i>	p6			Puede estar lleno de basura si no hay proceso en este contexto
<i>gacc</i>	p7			Puede estar lleno de basura si no hay proceso en este contexto
<i>giump</i>	p8	8000		Por defecto tiene el 8000
<i>Static ID</i>	p9			
<i>Id Counter</i>	p10	-1		
<i>gpc</i>	p11			
<i>gacc</i>	p12			
<i>giump</i>	p13	8000		
	p14			
	p15			
<b>Variables del sistema</b>				
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>Initial</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 a 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	0		Contendrá un valor serial para los id de los procesos
<i>Máximo numero</i>	c16	4		Máximo numero de procesos disponibles (Menos 1 para la resta)
	c17	998		Espacio para el SOM/Área de borrado
	c18	4		Espacio para el SOM/Área de borrado

# Sistema Operativo Concurrente y Paralelo para E-CARDIAC

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>Nuevo</i>	s0	-1		-1 Es su valor por defecto, el usuario siempre agregará una 8-dir
	s1	6000		Contiene la 6-dir del id del proceso actual, controlado por el lanzador, 6(px)
	s2	1(s1)	LDA s1	El acumulador toma un valor de la forma 6(px)
	s3	3(s98)	BLZ s98	Si acc<0 no hay proceso para actualizar
	s4	2000	ADD 000	Se obtiene la 6-dir del gpc del proceso
	s5	6(s7)	STO s7	Se guarda la instrucción 6(px)+1
	s6	1(c1)	LDA C1	Se obtiene el último pc del proceso con forma 8(pc)
	s7	6(p5)	STO p5	En p5 se actualiza gpc
	s8	1(s7)	LDA s7	Se obtiene la instrucción 6(px)+1
	s9	2000	ADD 000	Para acceder a la 6-dir del gacc del proceso
<i>Actualizar gpc</i>	s10	6(s12)	STO s12	En s12 se guarda 6(px)+2
	s11	1(c2)	LDA c2	Se obtiene el último acc del proceso
	s12	6(p6)	STO p6	Se actualiza el valor de gacc
	s13	1(s12)	LDA s12	Obtiene la 6(p6) del proceso que se está actualizando
	s14	2000	ADD 000	Obtiene la 6(p7) del proceso que se está ejecutando
	s15	6(s17)	STO s17	Guarda en e18 el código para guardar en la zona de procesos c14
	s16	1(c14)	LDA c14	Obtiene c14
	s17	6(p7)	STO p7	Guarda en la zona de procesos correspondiente al proceso el valor saber jump
	s18	8(s101)	JMP S100	Saltamos a cambiar de proceso
	s19			
<i>Actualizar gjump</i>	s20			
	s21			
	s22			
	s23			
	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)
	s29	1(py)	LDA py	Se obtiene el id del proceso (k+1)
<i>¿Hay otro?</i>	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
	s32	6(c17)	STO c17	Se guarda en c17 para usarla más tarde
	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
	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
<i>Recorrer las secciones gpc,gacc,gjump,staticid de py a px</i>	s40	7e11	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
	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
	s49	8(s35)	JMP s35	
<i>Regresar a recorrer</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
	"Si" de s30	s52	1(s29)	LDA s29

# Sistema Operativo Concurrente y Paralelo

Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
Borrado	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)
	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
	s59	1(c4)	LDA c4	Carga el id counter
	s60	7(000)	SUB 000	Se le resta 1 al id counter
M[c4]--	s61	6(c4)	STO c4	Guardar en c4 el nuevo valor
	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
M[c5]=M[c5]-5	s65	8(s98)	JMP S98	Salta a ver si el id counter es el ultimo
	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
	s68	1(c5)	LDA c5	Se obtiene el valor del Dir counter
Aumenta el Dir counter M[c5]+=5	s69	2(c11)	ADD c11	Se le suma 5 para tener la siguiente dirección
	s70	6(c5)	STO c5	
	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 ID	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
	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
Previa de pc	s77	2(c12)	ADD C12	Añade 2 para obtener 6(px)+4(Static ID)
	s78	6(s82)	STO s85	
	s79	1(c15)	LDA c15	Obtener Serial de Static ID
	s80	2000	ADD 000	Añadir una unidad
Guardar Nuevo Static ID para el proceso	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
	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
pc nuevo	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
	s88	1(c10)	LDA c10	
Poner valor default en s	s89	6(s)	STO s	En s se coloca el valor -1
	s90	1(c4)	LDA c4	Obtener ID counter
	s91	2000	ADD 000	Aumentar Id Counter
	s92	6(px)	STO px	Actualizar zona de procesos
Aumenta el Id counter M[c4]++	s93	6(c4)	STO c4	Actualizar zona de variables
	s94	8(000)	JMP 000	Salta al proceso 0
	s95			
	s96			
Waiter	s97	0(998)	INP 998	Indica el inicio de la espera
	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 s140	Si acc<0 salta al proceso 0
¿Se acabaron los programas?	s101	1(c7)	LDA c7	
	s102	2000	ADD 000	
	s103	6(c7)	STO c7	Se aumenta el id organizer
	s104	1(c6)	LDA c6	
Aumentar Id organizer M[c7]++	s105	2(c11)	ADD c11	
	s106	6(c6)	STO c6	Se aumenta el dir organizer

# Sistema Operativo Concurrente y Paralelo

Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
Verifica si hay que reiniciar	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
Sino-s109	s110	1(c6)	LDA c6	Se obtiene la dir del id del proceso a ejecutar
Actualizar s1	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"
Preparación para saltar al proceso con su pc y acc correctos	s113	1(c6)	LDA c6	
	s114	2(c8)	ADD c8	Se convierte en la 1-dir del id del proceso a ejecutar
	s115	2000	ADD 000	Se obtiene la 1-dir del gpc del proceso a ejecutar
Continuación	s116	6(s119)	STO s118	En s101 se guarda la 1-dir del gpc del proceso a ejecutar
	s117	2000	ADD 000	Se obtiene la 1-dir del gacc del proceso a ejecutar
	s118	6(s132)	STO s131	En s112 se guarda la 1-dir del gacc del proceso a ejecutar
Preparación gpc y gcc	s119	1(px)+1	LDA px+1	Obtiene el gpc del proceso a ejecutar
	s120	6(s133)	STO s132	Guarda el gpc del proceso a ejecutar en s106
Preparación gjump	s121	1(s132)	LDA s131	Obtiene la 1 dir del gacc
	s122	2000	ADD 000	Obtiene la 1 dir del gJump
	s123	6(s128)	STO s127	Guarda la 1 dir del gJump en s126
Salvar Static ID en 004	s124	2000	ADD 000	Añadir un 1 para obtener el 1(px)+4, SID
	s125	6(s126)	STO s125	Guardar en la siguiente celda
	s126	1(px)+4	LDA px+4	Obtener Static ID del proceso
Salvar gjump	s127	6004	STO 004	Guardar en 0004 para usarlo como identificador el proceso que se está ejecutando
	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
	s131	6003	STO 003	Se permiten saltos con bandera==1
Bandera	s132	1(px)+2	LDA px+2	Se obtiene acc= acc del proceso a ejecutar
	s133	8(xx)	JMP xx	La dir xx representa el valor donde el proceso se ejecutará
	s134	1(c3)	LDA c3	Se obtiene la primera dir de la zona de procesos, p3
Reiniciar dir organizer	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
Reinicidar 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
Regresar	s140	8(s110)	JMP s109	Regresar para lanzar el proceso
	s141	1(000)	LDA 000	
Reinicidar id organizer a 0	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
	s145	1(c3)	LDA c3	Obtiene p4
Asigna a dir organizer=p0	s146	7(c11)	SUB c11	Obtiene p1 al restarle 5
	s147	6(c6)	STO c6	Asignar a dir organizer=p0
	s148	8(s97)	JMP 000	Saltar a proceso 0

## 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
Mandar pc a C	e5	1(e0)	LDA e0	Carga en el acumulador el último pc del proceso antes de saltar
	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

# Sistema Operativo Concurrente y Paralelo

Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<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
	e12	8(s2)	JMP s2	Si la marca no es menor a 1 se va al área de actualización
	e13	8(s24)	JMP s19	Salta al área de borrado
	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
<i>Salto</i>	e17	8(s66)	JMP s66	Si no se ha alcanzado el máximo de procesos añadir otro
	e18			
<b>Zona de Procesos</b>				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>Id Counter</i>	p0	0		Id del primer programa
<i>gpc</i>	p1	8000		Es el pc del proceso 0 con op-code 8
<i>gacc</i>	p2	0		Es el acumulador del proceso
<i>giump</i>	p3	8000		Contiene el valor del que se tenía en la última dirección de memoria antes de saltar
<i>Static ID</i>	p4	0		Es el ID Static del proceso que no va a cambiar
<i>Id Counter</i>	p5	-1		Es el id del proceso correspondiente a está sección
<i>gpc</i>	p6			Puede estar lleno de basura si no hay proceso en este contexto
<i>gacc</i>	p7			Puede estar lleno de basura si no hay proceso en este contexto
<i>giump</i>	p8	8000		Por defecto tiene el 8000
<i>Static ID</i>	p9			
<i>Id Counter</i>	p10	-1		
<i>gpc</i>	p11			
<i>gacc</i>	p12			
<i>giump</i>	p13	8000		
	p14			
	p15			
<b>Variables del sistema</b>				
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>Initial</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 a 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	0		Contendrá un valor serial para los id de los procesos
<i>Máximo numero</i>	c16	4		Máximo numero de procesos disponibles (Menos 1 para la resta)
	c17	998		Espacio para el SOM/Área de borrado
	c18	4		Espacio para el SOM/Área de borrado

# Programas para E-CARDIAC

Imprimir Números del 1 al 10	
Versión CARDIAC	Versión CARDIAC C y PC
002	0110
800	1000
010	0111
100	6105
011	0112
605	1104
012	0113
104	3122
013	0114
322	5105
014	0115
505	1105
015	0116
105	2000
016	0117
200	6105
017	0118
605	1104
018	0119
104	7000
019	0120
700	6104
020	0121
604	8112
021	0122
812	9000
022	0104
900	0009
004	0800
009	8110
002	8985
810	

Lista reversa de números de Fibonacci	
Versión CARDIAC	Versión CARDIAC C y PC
002	0204
800	6000
004	0205
600	1000
005	0206
100	0289
006	0207
089	0000
007	0208

## Programas para CARDIAC

Versión CARDIAC	Versión CARDIAC C y PC
000	0000
008	0209
000	0000
009	0210
000	0208
010	0211
008	1208
011	0212
108	6209
012	0213
609	1209
013	0214
109	7000
014	0215
700	3227
015	0216
327	6209
016	0217
609	0207
017	0218
007	1206
018	0219
106	2204
019	0220
204	6225
020	0221
625	1206
021	0222
106	7000
022	0223
700	6206
023	0224
606	1207
024	0225
107	6000
025	0226
600	8213
026	0227
813	1208
027	0228
108	7000
028	0229
700	3239
029	0230
339	6208

## Programas para CARDIAC

Versión CARDIAC	Versión CARDIAC C y PC
030	0231
608	1206
031	0232
106	2000
032	0233
200	6206
033	0234
606	2205
034	0235
205	6236
035	0236
636	1000
036	0237
100	8290
037	0238
890	8227
038	0239
827	9000
039	0290
900	6297
090	0291
697	1999
091	0292
199	2000
092	0293
200	6296
093	0294
696	5297
094	0295
597	1297
095	0296
197	8000
096	0800
800	8210
002	8985
810	Entrada
007	0007
001	0001
001	0001
002	0002
002	0003
003	0005
005	0008
008	0013
013	

<b>Potencias de 2</b>	
Versión CARDIAC	Versión CARDIAC C y PC
002	0405
800	0009
005	0410
009	1000
010	0411
100	8480
011	0412
880	6404
012	0413
604	1405
013	0414
105	7000
014	0415
700	3421
015	0416
321	6405
016	0417
605	1404
017	0418
104	8490
018	0419
890	8480
019	0420
880	8412
020	0421
812	9000
021	0480
900	6487
080	0481
687	1999
081	0482
199	2000
082	0483
200	6486
083	0484
686	5487
084	0485
587	1487
085	0490
187	6497
090	0491
697	1999
091	0492

## Programas para CARDIAC

Versión CARDIAC	Versión CARDIAC C y PC
199	2000
092	0493
200	6496
093	0494
696	1497
094	0495
197	2497
095	0800
297	-----
002	-----
810	8985

# Tarjetas para cargar el Sistema Operativo

## Tarjetas del sistema Operativo

Tarjeta Sistema Operativo C		Tarjeta Sistema Operativo PC	
1	0002	1	0002
2	8000	2	8000
3	0003	3	0003
4	0000	4	0000
5	0004	5	0004
6	0000	6	0000
7	<b>0800</b>	7	<b>0800</b>
8	-0001	8	-0001
9	<b>0801</b>	9	<b>0801</b>
10	6000	10	6000
11	<b>0802</b>	11	<b>0802</b>
12	1801	12	1801
13	<b>0803</b>	13	<b>0803</b>
14	3898	14	3898
15	<b>0804</b>	15	<b>0804</b>
16	2000	16	2000
17	<b>0805</b>	17	<b>0805</b>
18	6807	18	6807
19	<b>0806</b>	19	<b>0806</b>
20	1966	20	1966
21	<b>0807</b>	21	<b>0807</b>
22	0998	22	0998
23	<b>0808</b>	23	<b>0808</b>
24	1807	24	1807
25	<b>0809</b>	25	<b>0809</b>
26	2000	26	2000
27	<b>0810</b>	27	<b>0810</b>
28	6812	28	6812
29	<b>0811</b>	29	<b>0811</b>
30	1967	30	1967
31	<b>0812</b>	31	<b>0812</b>
32	0998	32	0998
33	<b>0813</b>	33	<b>0813</b>
34	1812	34	1812
35	<b>0814</b>	35	<b>0814</b>
36	2000	36	2000
37	<b>0815</b>	37	<b>0815</b>
38	6817	38	6817
39	<b>0816</b>	39	<b>0816</b>
40	1979	40	1979
41	<b>0817</b>	41	<b>0817</b>
42	0998	42	0998
43	<b>0818</b>	43	<b>0818</b>
44	8901	44	8901
45	<b>0819</b>	45	<b>0824</b>

## Tarjetas del sistema Operativo

46	1801	46	1801
47	<b>0820</b>	47	<b>0825</b>
48	4011	48	4011
49	<b>0821</b>	49	<b>0826</b>
50	2973	50	2973
51	<b>0822</b>	51	<b>0827</b>
52	6965	52	2976
53	<b>0823</b>	53	<b>0828</b>
54	1972	54	6829
55	<b>0824</b>	55	<b>0829</b>
56	7000	56	0998
57	<b>0825</b>	57	<b>0830</b>
58	3898	58	3852
59	<b>0826</b>	59	<b>0831</b>
60	1965	60	1829
61	<b>0827</b>	61	<b>0832</b>
62	2976	62	6982
63	<b>0828</b>	63	<b>0833</b>
64	6829	64	1983
65	<b>0829</b>	65	<b>0834</b>
66	0998	66	6965
67	<b>0830</b>	67	<b>0835</b>
68	3852	68	7000
69	<b>0831</b>	69	<b>0836</b>
70	1829	70	3850
71	<b>0832</b>	71	<b>0837</b>
72	6982	72	1982
73	<b>0833</b>	73	<b>0838</b>
74	1983	74	2965
75	<b>0834</b>	75	<b>0839</b>
76	6965	76	6844
77	<b>0835</b>	77	<b>0840</b>
78	7000	78	7976
79	<b>0836</b>	79	<b>0841</b>
80	3850	80	4011
81	<b>0837</b>	81	<b>0842</b>
82	1982	82	2974
83	<b>0838</b>	83	<b>0843</b>
84	2965	84	6845
85	<b>0839</b>	85	<b>0844</b>
86	6844	86	0998
87	<b>0840</b>	87	<b>0845</b>
88	7976	88	0998
89	<b>0841</b>	89	<b>0846</b>
90	4011	90	1965
91	<b>0842</b>	91	<b>0847</b>

## Tarjetas del sistema Operativo

92	2974	92	7000
93	<b>0843</b>	93	<b>0848</b>
94	6845	94	6965
95	<b>0844</b>	95	<b>0849</b>
96	0998	96	8835
97	<b>0845</b>	97	<b>0850</b>
98	0998	98	1829
99	<b>0846</b>	99	<b>0851</b>
100	1965	100	8827
101	<b>0847</b>	101	<b>0852</b>
102	7000	102	1829
103	<b>0848</b>	103	<b>0853</b>
104	6965	104	7976
105	<b>0849</b>	105	<b>0854</b>
106	8835	106	4011
107	<b>0850</b>	107	<b>0855</b>
108	1829	108	2974
109	<b>0851</b>	109	<b>0856</b>
110	8827	110	6858
111	<b>0852</b>	111	<b>0857</b>
112	1829	112	1975
113	<b>0853</b>	113	<b>0858</b>
114	7976	114	0998
115	<b>0854</b>	115	<b>0859</b>
116	4011	116	1969
117	<b>0855</b>	117	<b>0860</b>
118	2974	118	7000
119	<b>0856</b>	119	<b>0861</b>
120	6858	120	6969
121	<b>0857</b>	121	<b>0862</b>
122	1975	122	1970
123	<b>0858</b>	123	<b>0863</b>
124	0998	124	7976
125	<b>0859</b>	125	<b>0864</b>
126	1969	126	6970
127	<b>0860</b>	127	<b>0865</b>
128	7000	128	8898
129	<b>0861</b>	129	<b>0866</b>
130	6969	130	1800
131	<b>0862</b>	131	<b>0867</b>
132	1970	132	3894
133	<b>0863</b>	133	<b>0868</b>
134	7976	134	1970
135	<b>0864</b>	135	<b>0869</b>
136	6970	136	2976
137	<b>0865</b>	137	<b>0870</b>

## Tarjetas del sistema Operativo

138	8898	138	6970
139	<b>0866</b>	139	<b>0871</b>
140	1800	140	2974
141	<b>0867</b>	141	<b>0872</b>
142	3895	142	6892
143	<b>0868</b>	143	<b>0873</b>
144	1969	144	2000
145	<b>0869</b>	145	<b>0874</b>
146	2000	146	6884
147	<b>0870</b>	147	<b>0875</b>
148	6969	148	2000
149	<b>0871</b>	149	<b>0876</b>
150	1970	150	6887
151	<b>0872</b>	151	<b>0877</b>
152	2976	152	2977
153	<b>0873</b>	153	<b>0878</b>
154	6970	154	6882
155	<b>0874</b>	155	<b>0879</b>
156	2974	156	1980
157	<b>0875</b>	157	<b>0880</b>
158	6887	158	2000
159	<b>0876</b>	159	<b>0881</b>
160	2000	160	6980
161	<b>0877</b>	161	<b>0882</b>
162	6889	162	0998
163	<b>0878</b>	163	<b>0883</b>
164	2000	164	1800
165	<b>0879</b>	165	<b>0884</b>
166	6892	166	0998
167	<b>0880</b>	167	<b>0885</b>
168	2977	168	1000
169	<b>0881</b>	169	<b>0886</b>
170	6885	170	7000
171	<b>0882</b>	171	<b>0887</b>
172	1980	172	0998
173	<b>0883</b>	173	<b>0888</b>
174	2000	174	1975
175	<b>0884</b>	175	<b>0889</b>
176	6980	176	6800
177	<b>0885</b>	177	<b>0890</b>
178	0998	178	1969
179	<b>0886</b>	179	<b>0891</b>
180	1969	180	2000
181	<b>0887</b>	181	<b>0892</b>
182	0998	182	0998
183	<b>0888</b>	183	<b>0893</b>

## Tarjetas del sistema Operativo

184	1800	184	6969
185	<b>0889</b>	185	<b>0894</b>
186	0998	186	8000
187	<b>0890</b>	187	<b>0897</b>
188	1000	188	0998
189	<b>0891</b>	189	<b>0898</b>
190	7000	190	1969
191	<b>0892</b>	191	<b>0899</b>
192	0998	192	7000
193	<b>0893</b>	193	<b>0900</b>
194	1975	194	3941
195	<b>0894</b>	195	<b>0901</b>
196	6800	196	1972
197	<b>0895</b>	197	<b>0902</b>
198	1972	198	2000
199	<b>0896</b>	199	<b>0903</b>
200	7000	200	6972
201	<b>0897</b>	201	<b>0904</b>
202	3000	202	1971
203	<b>0898</b>	203	<b>0905</b>
204	1969	204	2976
205	<b>0899</b>	205	<b>0906</b>
206	7000	206	6971
207	<b>0900</b>	207	<b>0907</b>
208	3941	208	1969
209	<b>0901</b>	209	<b>0908</b>
210	1972	210	7972
211	<b>0902</b>	211	<b>0909</b>
212	2000	212	3934
213	<b>0903</b>	213	<b>0910</b>
214	6972	214	1971
215	<b>0904</b>	215	<b>0911</b>
216	1971	216	2974
217	<b>0905</b>	217	<b>0912</b>
218	2976	218	6801
219	<b>0906</b>	219	<b>0913</b>
220	6971	220	1971
221	<b>0907</b>	221	<b>0914</b>
222	1969	222	2973
223	<b>0908</b>	223	<b>0915</b>
224	7972	224	2000
225	<b>0909</b>	225	<b>0916</b>
226	3934	226	6919
227	<b>0910</b>	227	<b>0917</b>
228	1971	228	2000
229	<b>0911</b>	229	<b>0918</b>

## Tarjetas del sistema Operativo

230	2974	230	6932
231	<b>0912</b>	231	<b>0919</b>
232	6801	232	0998
233	<b>0913</b>	233	<b>0920</b>
234	1971	234	6933
235	<b>0914</b>	235	<b>0921</b>
236	2973	236	1932
237	<b>0915</b>	237	<b>0922</b>
238	2000	238	2000
239	<b>0916</b>	239	<b>0923</b>
240	6919	240	6928
241	<b>0917</b>	241	<b>0924</b>
242	2000	242	2000
243	<b>0918</b>	243	<b>0925</b>
244	6932	244	6926
245	<b>0919</b>	245	<b>0926</b>
246	0998	246	0998
247	<b>0920</b>	247	<b>0927</b>
248	6933	248	6004
249	<b>0921</b>	249	<b>0928</b>
250	1932	250	0998
251	<b>0922</b>	251	<b>0929</b>
252	2000	252	6979
253	<b>0923</b>	253	<b>0930</b>
254	6928	254	1000
255	<b>0924</b>	255	<b>0931</b>
256	2000	256	6003
257	<b>0925</b>	257	<b>0932</b>
258	6926	258	0998
259	<b>0926</b>	259	<b>0933</b>
260	0998	260	0998
261	<b>0927</b>	261	<b>0934</b>
262	6004	262	1968
263	<b>0928</b>	263	<b>0935</b>
264	0998	264	6971
265	<b>0929</b>	265	<b>0936</b>
266	6979	266	2973
267	<b>0930</b>	267	<b>0937</b>
268	1000	268	6938
269	<b>0931</b>	269	<b>0938</b>
270	6003	270	0998
271	<b>0932</b>	271	<b>0939</b>
272	0998	272	6972
273	<b>0933</b>	273	<b>0940</b>
274	0998	274	8910
275	<b>0934</b>	275	<b>0941</b>

## Tarjetas del sistema Operativo

276	1968	276	1000
277	<b>0935</b>	277	<b>0942</b>
278	6971	278	7000
279	<b>0936</b>	279	<b>0943</b>
280	2973	280	6972
281	<b>0937</b>	281	<b>0944</b>
282	6938	282	6004
283	<b>0938</b>	283	<b>0945</b>
284	0998	284	1968
285	<b>0939</b>	285	<b>0946</b>
286	6972	286	7976
287	<b>0940</b>	287	<b>0947</b>
288	8910	288	6971
289	<b>0941</b>	289	<b>0948</b>
290	1000	290	8897
291	<b>0942</b>	291	<b>0950</b>
292	7000	292	0998
293	<b>0943</b>	293	<b>0951</b>
294	6972	294	6967
295	<b>0944</b>	295	<b>0952</b>
296	6004	296	1000
297	<b>0945</b>	297	<b>0953</b>
298	1968	298	7000
299	<b>0946</b>	299	<b>0954</b>
300	7976	300	6003
301	<b>0947</b>	301	<b>0955</b>
302	6971	302	1950
303	<b>0948</b>	303	<b>0956</b>
304	8000	304	2978
305	<b>0950</b>	305	<b>0957</b>
306	0998	306	6966
307	<b>0951</b>	307	<b>0958</b>
308	6967	308	1999
309	<b>0952</b>	309	<b>0959</b>
310	1000	310	6979
311	<b>0953</b>	311	<b>0960</b>
312	7000	312	1950
313	<b>0954</b>	313	<b>0961</b>
314	6003	314	3963
315	<b>0955</b>	315	<b>0962</b>
316	1950	316	8802
317	<b>0956</b>	317	<b>0963</b>
318	2978	318	8824
319	<b>0957</b>	319	<b>0985</b>
320	6966	320	1981
321	<b>0958</b>	321	<b>0986</b>

## Tarjetas del sistema Operativo

322	1999	322	7969
323	<b>0959</b>	323	<b>0987</b>
324	6979	324	3000
325	<b>0960</b>	325	<b>0988</b>
326	1950	326	8866
327	<b>0961</b>	327	<b>0965</b>
328	3963	328	0998
329	<b>0962</b>	329	<b>0966</b>
330	8802	330	0998
331	<b>0963</b>	331	<b>0967</b>
332	8819	332	0998
333	<b>0985</b>	333	<b>0968</b>
334	1981	334	0774
335	<b>0986</b>	335	<b>0969</b>
336	7969	336	0000
337	<b>0987</b>	337	<b>0970</b>
338	3000	338	0769
339	<b>0988</b>	339	<b>0971</b>
340	8866	340	0769
341	<b>0965</b>	341	<b>0972</b>
342	0998	342	0000
343	<b>0966</b>	343	<b>0973</b>
344	0998	344	1000
345	<b>0967</b>	345	<b>0974</b>
346	0998	346	6000
347	<b>0968</b>	347	<b>0975</b>
348	0774	348	-0001
349	<b>0969</b>	349	<b>0976</b>
350	0000	350	0005
351	<b>0970</b>	351	<b>0977</b>
352	0769	352	0002
353	<b>0971</b>	353	<b>0978</b>
354	0769	354	8000
355	<b>0972</b>	355	<b>0979</b>
356	0000	356	8000
357	<b>0973</b>	357	<b>0980</b>
358	1000	358	0000
359	<b>0974</b>	359	<b>0981</b>
360	6000	360	0004
361	<b>0975</b>	361	<b>0982</b>
362	-0001	362	0998
363	<b>0976</b>	363	<b>0983</b>
364	0005	364	0004
365	<b>0977</b>	365	<b>0769</b>
366	0002	366	0000
367	<b>0978</b>	367	<b>0770</b>

## Tarjetas del sistema Operativo

368	8000	368	8000
369	<b>0979</b>	369	<b>0771</b>
370	8000	370	0000
371	<b>0980</b>	371	<b>0772</b>
372	0000	372	8000
373	<b>0981</b>	373	<b>0773</b>
374	0004	374	0000
375	<b>0982</b>	375	<b>0774</b>
376	0998	376	-0001
377	<b>0983</b>	377	<b>0777</b>
378	0004	378	8000
379	<b>0769</b>	379	<b>0779</b>
380	0000	380	-0001
381	<b>0770</b>	381	<b>0782</b>
382	8000	382	8000
383	<b>0771</b>	383	<b>0784</b>
384	0000	384	-0001
385	<b>0772</b>	385	<b>0787</b>
386	8000	386	8000
387	<b>0773</b>	387	<b>0789</b>
388	0000	388	-0001
389	<b>0774</b>	389	<b>0792</b>
390	-0001	390	8000
391	<b>0777</b>	391	<b>0794</b>
392	8000	392	-0001
393	<b>0779</b>	393	<b>0797</b>
394	-0001	394	8000
395	<b>0782</b>	395	<b>0799</b>
396	8000	396	-0001
397	<b>0784</b>		
398	-0001		
399	<b>0787</b>		
400	8000		
401	<b>0789</b>		
402	-0001		
403	<b>0792</b>		
404	8000		
405	<b>0794</b>		
406	-0001		
407	<b>0797</b>		
408	8000		
409	<b>0799</b>		
410	-0001		