



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 14/09/2024

**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 de 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 con la intención, precisamente, de hacer más fácil de entender a los alumnos de aquel entonces cómo funcionaba una computadora; el manual del modelo se puede consultar en [3]. 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>59</b>
3.1. E-CARDIAC : Electronic CARDboard Illustrative Aid to Computation . . . . .	61
3.2. E-CARDIAC C: Electronic CARDboard Illustrative Aid to Concurrent Computing . . . . .	68
3.2.1. Necesidad de un sistema operativo . . . . .	68
3.2.2. Mejoras necesarias en el <i>Hardware</i> . . . . .	69
3.2.3. Cambios en el lenguaje . . . . .	72
3.2.4. Sistema Operativo Mínimo C: Aspectos Generales . . . . .	74
3.2.5. Sistema Operativo Mínimo C: Un nuevo proceso en la cola . . . . .	83
3.2.6. Sistema Operativo Mínimo C: Lanzamiento de procesos . . . . .	98
3.2.7. Sistema Operativo Mínimo C: Actualización y borrado . . . . .	111
3.2.8. Sistema Operativo Mínimo C: Guía rápida de uso . . . . .	123
3.3. E-CARDIAC PC: Electronic CARDboard Illustrative Aid to Concurrent Parallel Computation . . . . .	125
3.3.1. Arquitectura renovada para un modelo paralelo . . . . .	126
3.3.2. Un sistema operativo para dos procesadores . . . . .	130
3.3.3. Funcionamiento del sistema operativo mínimo . . . . .	132
3.3.4. Ejecutando procesos en E-CARDIAC PC . . . . .	146
3.3.5. Sistema Operativo Mínimo PC: Guía rápida de uso . . . . .	152
<b>4. Conclusiones</b>	<b>154</b>
<b>Apéndices</b>	<b>161</b>
<b>A. Historia de la computación</b>	<b>162</b>
<b>B. Sistema Operativo Concurrente para E-CARDIAC</b>	<b>171</b>
<b>C. Sistema Operativo Concurrente y Paralelo para E-CARDIAC</b>	<b>176</b>

<b>D. Programas para E-CARDIAC</b>	<b>181</b>
<b>E. Tarjetas para cargar el Sistema Operativo</b>	<b>187</b>
<b>F. Guía de acceso al repositorio de GitHUb</b>	<b>197</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 . . . . .	60
3.2.	Arquitectura de CARDIAC por Jorge Vasconcelos, 2018 . . . . .	61
3.4.	Listas desplegables de E-CARDIAC . . . . .	62
3.3.	Pantalla de inicio de E-CARDIAC . . . . .	63
3.5.	E-CARDIAC Muestra de Output . . . . .	65
3.6.	E-CARDIAC Carga masiva por tarjetas . . . . .	66
3.7.	E-CARDIAC Carga individual de instrucciones . . . . .	66
3.8.	E-CARDIAC Cola de instrucciones/datos . . . . .	66
3.9.	E-CARDIAC después de iniciar sus funciones . . . . .	67

3.10. Diagrama de Arquitectura de E-CARDIAC C . . . . .	70
3.11. Diagrama de flujo de SOMC . . . . .	76
3.12. SOMC: Preámbulo . . . . .	77
3.13. E-CARDIAC C Apagada . . . . .	80
3.14. E-CARDIAC C durante el arranque . . . . .	81
3.15. E-CARDIAC C Sistema operativo mínimo cargado . . . . .	82
3.16. Formas de entrar al sistema operativo mínimo C . . . . .	83
3.17. SOMC : Variables del sistema . . . . .	86
3.18. Diagrama de segmento para añadir un proceso . . . . .	88
3.19. SOMC: Contenidos generales del núcleo . . . . .	88
3.20. SOMC: Añadir un proceso, parte 1 . . . . .	89
3.21. SOMC: Añadir un proceso, parte 2 . . . . .	89
3.22. SOMC: Zona de procesos del sistema operativo . . . . .	91
3.23. E-CARDIAC C Programa en <i>deck mode</i> . . . . .	94
3.24. E-CARDIAC C Programa en cola . . . . .	95
3.25. E-CARDIAC C Instrucción para añadir proceso . . . . .	96
3.26. E-CARDIAC C Programa 1 cargado en memoria . . . . .	96
3.27. E-CARDIAC C Proceso 1 cargado . . . . .	97
3.28. E-CARDIAC C Tres procesos agregados . . . . .	97
3.29. Acercamiento al preámbulo en el diagrama . . . . .	99
3.30. E-CARDIAC: Preámbulo después de una detención en P0 . . . . .	99
3.31. Acercamiento a la parte del bloqueo al proceso 0 . . . . .	101
3.32. SOMC segmento del proceso 0 . . . . .	102
3.33. E-CARDIAC C Borrado de proceso parte 1 . . . . .	103
3.34. SOMC Lanzamiento de procesos . . . . .	105
3.35. SOMC Lanzamiento de procesos parte 1 . . . . .	106
3.36. SOMC Lanzamiento de procesos parte 2 . . . . .	107
3.37. SOMC Lanzamiento de procesos parte 3 . . . . .	107
3.38. Estatus de organizadores antes de lanzar el proceso . . . . .	108
3.39. Valores del SOM antes de lanzar el proceso . . . . .	109

3.40. E-CARDIAC C: Ejecución de procesos . . . . .	110
3.41. Diagrama de actualización de proceso . . . . .	112
3.42. SOMC Actualizar proceso . . . . .	113
3.43. Actualizar proceso 1 . . . . .	113
3.44. Conexión entre actualización de procesos y lanzamiento . . . . .	114
3.45. Contexto del proceso 1 actualizado . . . . .	114
3.46. Proceso 1 antes de finalizar . . . . .	115
3.47. Preámbulo en la finalización del proceso 1 . . . . .	116
3.48. Preámbulo en la finalización del proceso 1 a punto de saltar . . . . .	116
3.49. Acercamiento a zona de borrado en diagrama . . . . .	117
3.50. Zona de procesos antes de que el proceso 1 sea borrado . . . . .	118
3.51. Zona de procesos después de borrar el proceso 1 . . . . .	118
3.52. SOMC Borrar proceso parte 2 . . . . .	119
3.53. SOMC Borrar proceso parte 3 . . . . .	120
3.54. Variables del sistema antes de borrar el proceso 1 . . . . .	121
3.55. Variables del sistema después de borrar el proceso 1 . . . . .	121
3.56. Salidas finales de los procesos . . . . .	122
3.57. Salidas finales de los procesos en texto plano . . . . .	122
3.58. Arquitectura de cómputo paralela. . . . .	128
3.59. Diagrama de Sistema Operativo Mínimo Paralelo . . . . .	131
3.60. Añadir proceso en SOMP . . . . .	133
3.61. Preámbulo sistema operativo mínimo paralelo . . . . .	134
3.62. Añadir proceso en un sistema operativo mínimo paralelo . . . . .	135
3.63. E-CARDIAC PC sin iniciar . . . . .	136
3.64. E-CARDIAC PC Booteo . . . . .	137
3.65. E-CARDIAC PC Iniciado . . . . .	138
3.66. Acercamiento al preámbulo del sistema operativo mínimo . . . . .	139
3.67. Acercamiento a segmento de <i>waiter</i> y control de procesos . . . . .	140
3.68. Segmento de control de procesos . . . . .	140
3.69. Acercamiento a fracción de lanzamiento de procesos . . . . .	141

3.70. Sistema operativo mínimo paralelo, fracción de lanzamiento . . . . .	141
3.71. Sistema operativo mínimo paralelo, fracción de lanzamiento, parte 2 . . . . .	142
3.73. Sistema operativo mínimo actualización de procesos . . . . .	142
3.72. Acercamiento al segmento de actualización de procesos . . . . .	143
3.74. Acercamiento a la fracción de borrado . . . . .	144
3.75. Sistema operativo mínimo borrar proceso . . . . .	144
3.76. Sistema operativo mínimo borrar proceso: parte 2 . . . . .	145
3.78. Proceso de Fibonacci cargado . . . . .	146
3.79. Subrutina de proceso de Fibonacci . . . . .	146
3.77. Añadiendo un nuevo proceso en E-CARDIAC PC . . . . .	147
3.80. Programa pintor en el <i>deck</i> . . . . .	148
3.81. Agregando cantidad de números para serie de Fibonacci . . . . .	149
3.82. Primeras salidas de “pintor” . . . . .	150
3.83. Ejecución finalizada . . . . .	150
3.84. Salida en texto de los procesos ejecutados . . . . .	151

# 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*, el cual se puede consultar en [10]. 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 [11].

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*[11].

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[11]. 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.

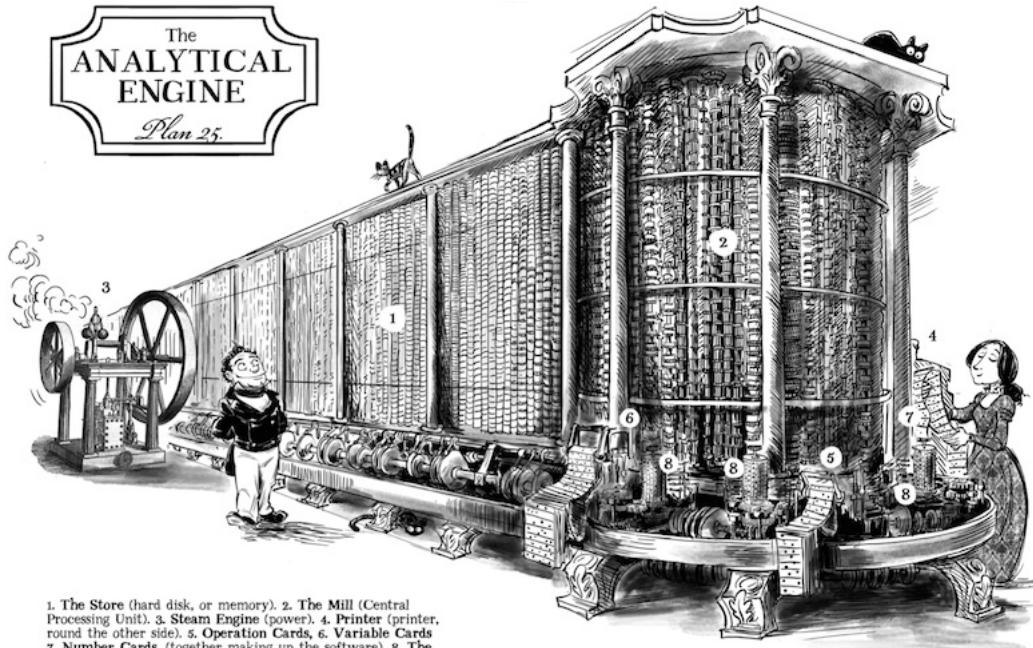


Figura 1.1: Fuente: Sydney Padua (2015), The Analytical Engine.

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], [12].

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], [12].

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 [13] 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 Analyser*, 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 [14].

---

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

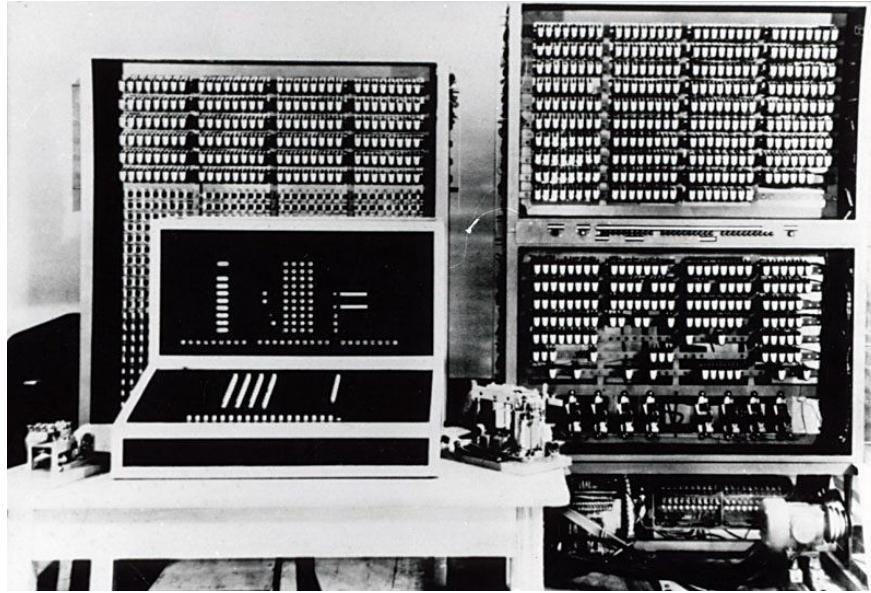


Figura 1.2: Fuente: Computer History Museum, The Zuse Z3 Computer.

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 [15].

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 [16].

Después de revisar algunas computadoras un tanto desconocidas, nos quedan dos que son quizás 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.

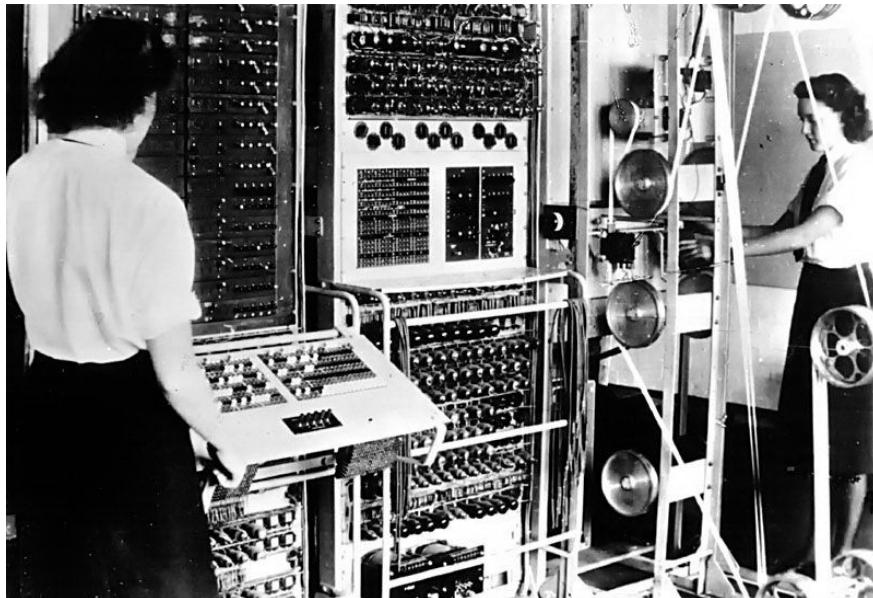


Figura 1.3: Fuente: Computer History Museum, Colossus Mark 1.

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 [17]. 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 [18], 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 [18].

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

---

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



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

del 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], [18].

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*, realmente veloz y que representaba los avances más grandes que tenía IBM en ese momento [8], [15].

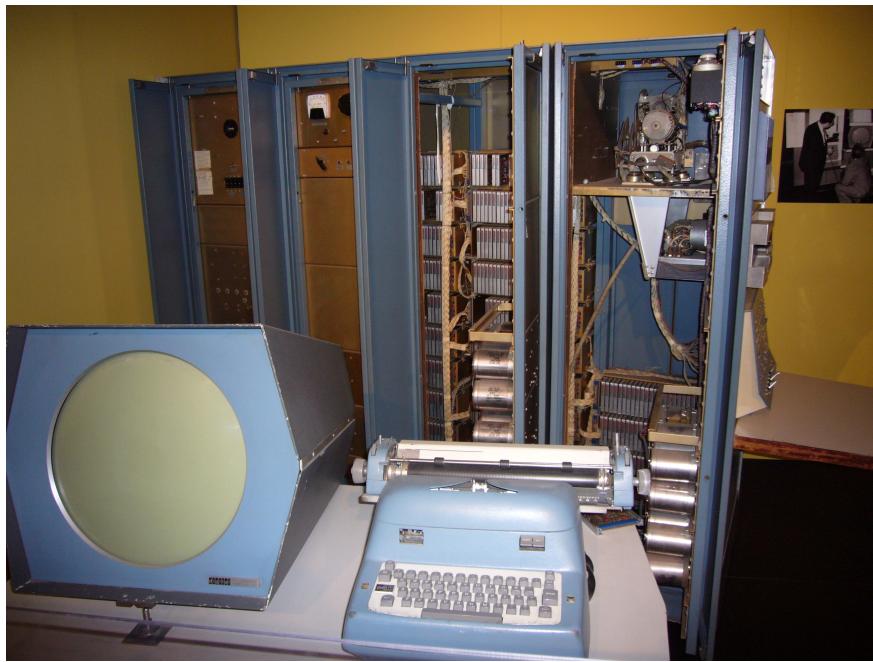


Figura 1.6: Fuente: Matthew Hutchinson - Flickr, Ordenador PDP-1.

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 [15].

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 relevantes. El hardware primaba sobre cualquier avance en software, pero en esta época la importancia del software comenzó a ser cada vez más fuerte [18].



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

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 [19]. 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 [18, 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 [20] 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 [21], 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 [20], [22].

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 [20].

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 [18].

#### Primeras apariciones de sistemas operativos

En el libro [18], 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 [20].

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 [18].

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 [18].

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 [18].

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 [18].

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 [18].

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 [18] y [23],

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 [18].

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 [18].

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 [18].

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 [18].

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 [18].

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 [18].

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 [18].

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 [18].

## 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 [23, 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 [24].

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 [25].

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 [18].

#### 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 [26, 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 [27].

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, lanzaron 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 [28].

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 [29]. 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 acompañaba el video era un modelo de cómputo llamado **CARDIAC** (CARDboard Illustrative Aid to Computation). 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 [30].

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 simplificada, 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.

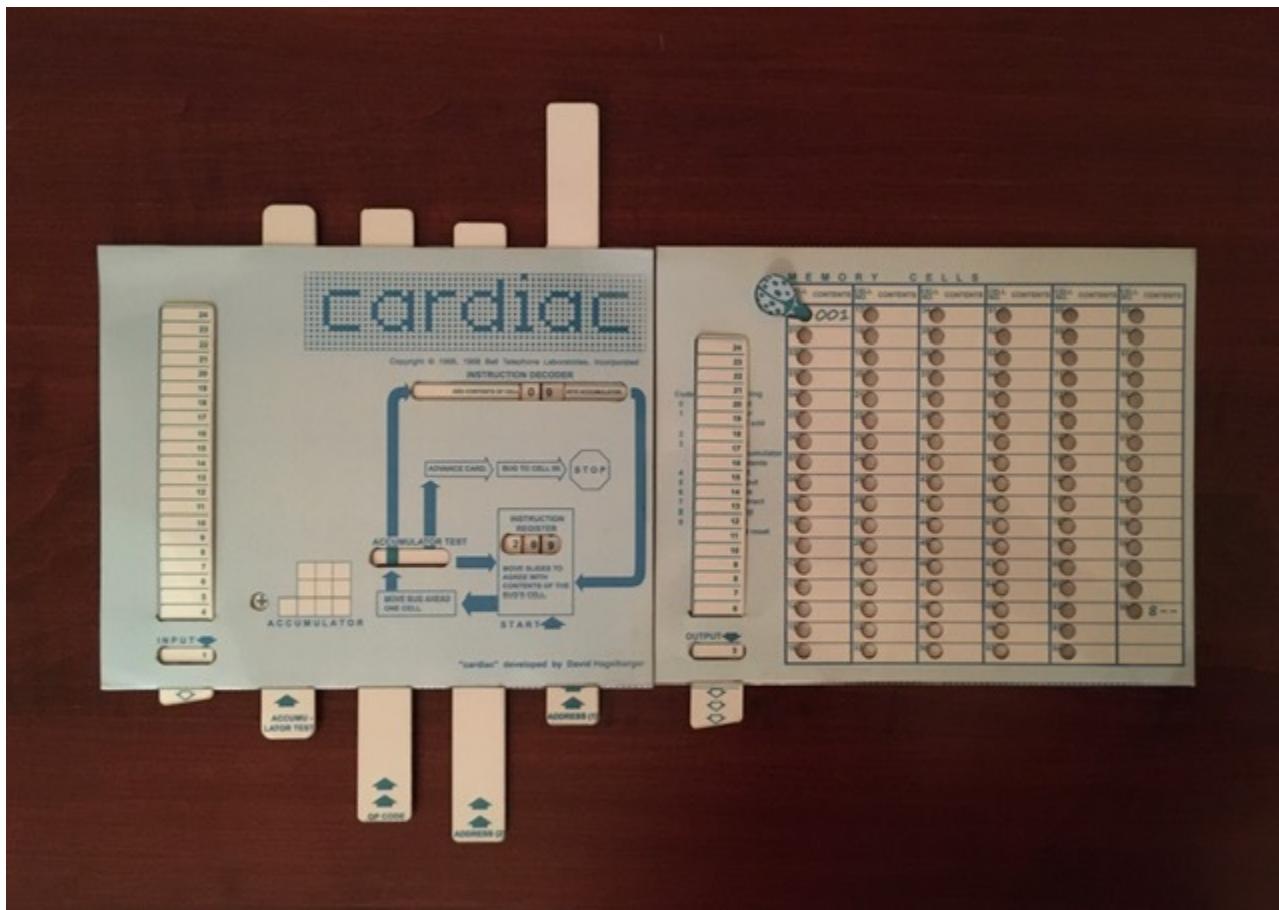


Figura 1.9: Modelo de CARDIAC construido.

### 1.2.5. Actualidad de las computadoras

La evolución desde los años 80 hasta la actualidad ha sido sorprendente. Es muy interesante leer o escuchar 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 más actuales 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 las computadoras actuales conservan de sus antepasados, y otros que claramente han mejorado. Incluso la “moderna” arquitectura que se ha vuelto muy popular con los teléfonos móviles, utilizada en los procesadores *ARM*, no es más que una versión actualizada de la ya lejana arquitectura Harvard [31, p. 109].

Una de las mejoras más notables es el cómputo en la nube o cómputo distribuido, que toma la herencia del ahora innecesario tiempo compartido (debido a la potencia de las computadoras actuales). Esta tecnología toma el concepto base y lo lleva mucho más lejos de lo que siquiera llegaron a imaginar 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, junto con otros avances a nivel de hardware como 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 sirven para recordar que en el mundo de la computación nada es estático y la evolución siempre continúa. Por esa razón, incluso un modelo tan bueno como *CARDIAC*, que ha servido tan bien durante tantos años, requiere de una actualización.

# Capítulo 2

## Arquitectura básica de las computadoras

En este capítulo, revisaremos la composición de una computadora, es decir, los elementos que contiene y su organización. Tanenbaum nos dice en [32] que la *organización o arquitectura de computadoras* no incluye los aspectos de implementación ni el tipo de tecnología usada en los diferentes componentes, para poder centrarse en los elementos que dan forma a la computadora. Además, en este texto utilizaré un enfoque basado en modelos para mostrar solo algunas partes relevantes de la computadora, dejando fuera muchos componentes importantes en ella, pero logrando así una simplificación necesaria para un acercamiento claro a la estructura de una computadora.

### 2.1. Funcionamiento de las computadoras

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

#### 2.1.1. Arquitectura von Neumann

En [33], von Neumann nos describe cómo debería ser una computadora de propósito general completamente digital (sin usar el término computadora), estableciendo que, en ese 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, y *EDSAC* fue una de las primeras máquinas que las implementó, destacando especialmente la idea de los *programas almacenados* (en memoria).

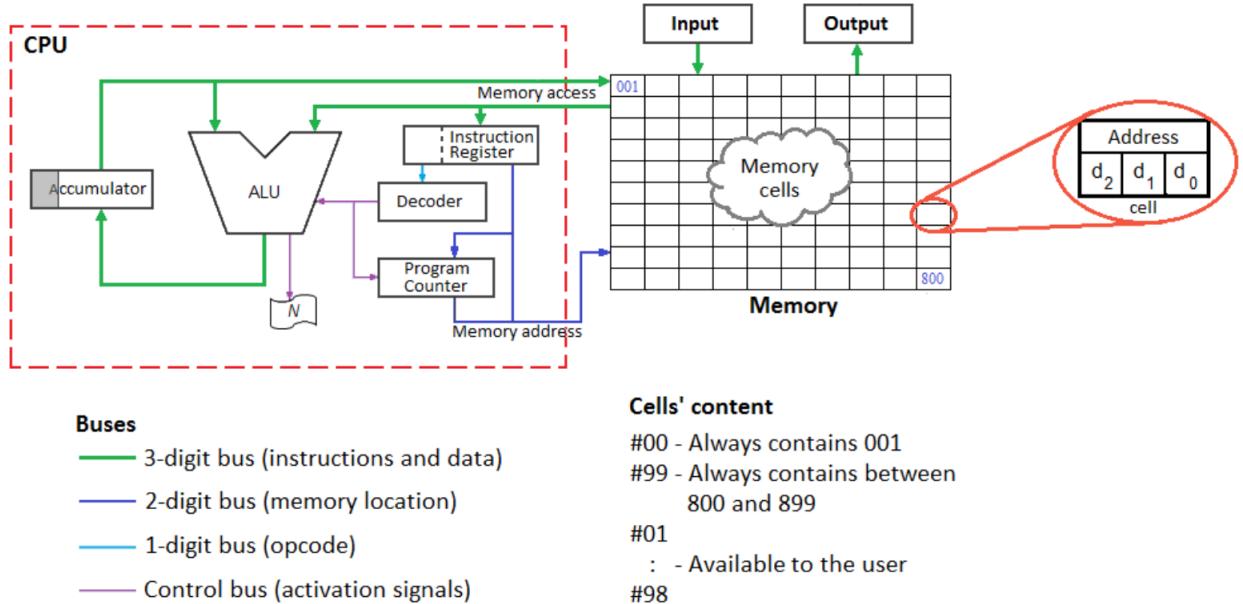


Figura 2.1: Arquitectura CARDIAC, Jorge Vasconcelos(2018)

La figura 2.1 nos ayudará a seguir lo que se conoce como arquitectura von Neumann, que, como se mencionó al principio, tiene como idea representativa la de los programas almacenados, la cual se refiere a que en la **memoria principal** se almacenen los datos de los programas, pero también las instrucciones. Esta idea es quizás una de las más polémicas del modelo debido a que plantea problemas en la seguridad e integridad de los programas al permitir que las instrucciones de un programa en ejecución puedan ser sustituidas por datos, situación que podría hacer que el programa falle si la implementación no es adecuada.

Al mismo tiempo, esta fue una de las ideas clave para llevar las máquinas de cómputo más lejos de lo que se podía imaginar en los años 40, ya que permitía ahorrar muchos costos al almacenar todo en la misma memoria y hacer la programación más eficiente. 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 almacena [32].

Esto nos lleva a su segundo punto: si tanto los datos como las 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 deben ser clasificados por el **Instruction Register**, o *registro de instrucciones*, como puramente datos o como instrucciones. En caso de ser instrucciones, deben 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) [33].

La *ALU* necesita resguardar sus resultados en un espacio particular llamado **Accumulator**, o *acumulador*, un espacio de almacenamiento especial en el que terminan todos los cálculos de esta unidad. Conocido igualmente como **registro**, es una unidad separada de la memoria principal en la que se puede almacenar poca información, pero de forma muy rápida. No se utilizará muy a menudo ese nombre porque todos los objetos dentro de la unidad de control también son llamados registros [33].

Además, se necesita de una ejecución secuencial en la que, una vez se lea una instrucción o dato de la memoria principal, se continúe con 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 se incremente para lograr la ejecución secuencial. El **Program Counter**, o *contador de programa*, logra esto en *CARDIAC* al incrementar de uno en uno su valor y apuntar a la siguiente dirección de memoria, a menos que la unidad de control, a través de sus **buses** que pueden llevar información del decodificador, transmita una instrucción que indique un salto [33].

Los buses son el sistema de comunicación interno de la computadora, cada uno con una especificación de acuerdo al tipo de datos que pueden transportar para comunicar los diferentes registros dentro de la máquina [33].

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 requieren dispositivos de entrada y salida (**Input/Output**), para que el usuario sea capaz tanto de dar instrucciones a la máquina como de recibir los resultados de esta [33].

Con estos puntos básicos, tenemos un modelo de computación que representa de forma correcta a las computadoras actuales y que es muy cercano a las computadoras de los años 60. En su artículo [33], von Neumann describió con más profundidad su modelo, que acompañó con detalles técnicos a nivel arquitectura y a nivel de implementación, donde también incluyó una *jerarquía de memorias* que es muy interesante de analizar, en la cual, además de la memoria principal, añade una memoria *caché* y una *memoria secundaria*. Estas no aparecen en el modelo de *CARDIAC* para simplificarlo .

Cada una de estas memorias es 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 y 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, mejor conocida ahora como *Random Access Memory (RAM)*, y la memoria secundaria, permitiendo aumentar la velocidad en la ejecución de los procesos [33].

Una vez que se tiene la arquitectura, se define el lenguaje de la máquina, aunque ciertamente hay una correspondencia bilateral entre estos dos conceptos, dado que tanto el lenguaje define a la máquina como la máquina al lenguaje [32]. El lenguaje de una máquina con esta arquitectura debe incluir las operaciones básicas: sumar, restar, multiplicar y dividir; pero si tratamos de ser eficientes, podemos omitir 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, programas que son utilizados por otros, 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 requiere instrucciones para que el usuario pueda ingresar datos a una dirección particular de memoria, y también para que pueda enviar datos desde la memoria principal o desde el acumulador al dispositivo de salida [33].

Con todas estas unidades, registros y un lenguaje, tenemos un modelo bien definido que cumple con la arquitectura de von Neumann. En este breve texto se puede apreciar otra de las características que han hecho 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. Dejando de lado los aspectos más técnicos del hardware, que requieren un estudio avanzado en ingeniería y física, junto a aquellos elementos del software que dividen en muchas más capas las unidades que mencionamos, logramos realizar una conveniente simplificación de una computadora para explicar esta arquitectura. Con ello, podemos tener una vista clara y concisa de esta, que es la arquitectura que siguen la mayoría de las computadoras actuales con procesadores Intel o AMD<sup>1</sup>.

### 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 [18], 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, y general, con el que

---

<sup>1</sup>Estas compañías han realizado sus correspondientes adecuaciones y actualizaciones a lo largo de los años a la arquitectura original presentada por von Neumann [32].

la comunicación sea más sencilla para buscar la imagen en cuestión [18].

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 [18].

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 [18].

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** [18].

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 [18].

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 [18].

### 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>2</sup>, la acción de cargar un programa inicial que permita al usuario interactuar con la computadora. Muchas computadoras en el pasado no necesitaban esto, especialmente cuando se movían 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 [18].

La idea general es tener almacenado un programa en una memoria no volátil y un sistema que cargué esas instrucciones en la máquina. Así, cuando encienda la máquina, se cargarán estas instrucciones, iniciando la máquina con un programa ya cargado. Este es el programa a través del cual se cargarán el resto de programas a la memoria principal para su ejecución. El no tener un sistema de arranque implica que el usuario debe realizar un proceso más tardado modificando directamente las direcciones de memoria, o buscar una alternativa para interactuar con la computadora [18].

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 la dirección #00, en el modelo de CARDIAC de la figura 2.1, está en la memoria ROM, puesto que no se puede editar [18].

---

<sup>2</sup>Forma en la que se le dice al inicio de operaciones de una computadora.

Este pequeño sistema de arranque funciona para preparar las operaciones básicas de la computadora y, en caso de tener un sistema operativo, de iniciararlo. El sistema operativo, que debe estar almacenado en un espacio físico de memoria, 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 [18].

## 2.2. Modelos de computación

Como ya mencionamos, los modelos son una abstracción que busca generalizar objetos complejos para entender mejor conceptos específicos de ellos. En esta sección, presentaré dos modelos de cómputo, es decir, dos formas diferentes de computación, que cambiaron drásticamente la forma en la que se computa.

### 2.2.1. Modelo de cómputo concurrente

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 una misma *CPU* de forma que todos tengan tiempo de ejecución. De esta manera, si tengo dos procesos, A y B, con 15 etapas cada uno, se pueden ir ejecutando intercaladamente estas etapas para que ambos se ejecuten casi “al mismo tiempo” [18].

Las computadoras actuales logran esto, e incluso, por su velocidad, nos hacen creer que varios procesos de verdad se ejecutan al mismo tiempo. Cabe aclarar que esta área sigue evolucionando, ahora no solo los procesos se ejecutan concurrentemente, sino también los **hilos** (partes de un proceso) para conseguir aún más eficiencia en la ejecución de programas [18].

La concurrencia ofrece grandes ventajas para la ejecución de procesos, pero de la misma manera exige lidiar con otros problemas. Los más relevantes son: la gestión de los recursos para repartir entre N procesos; el control de seguridad para que esos procesos no invadan el espacio de memoria de los demás; la correcta sincronización de estos; y que en cada momento que un proceso sea sacado de la ejecución, toda su información, incluidas las variables globales que este pueda tener, sea resguardada en un espacio de memoria seguro. De esta forma, cuando vuelva a tener tiempo de ejecución, este proceso encontrará sus datos donde espera que estén, y esa “pausa” no afectará su comportamiento. 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 se espera [34].

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”. Este es un problema de recursos y sincronización, que consta de varios procesos compitiendo por ellos, y en el que se debe asegurar el acceso exclusivo de cada proceso a los recursos [35].

Problemas como el de exclusión mutua, y los demás mencionados anteriormente, son temas comunes en la concurrencia. Gran parte del esfuerzo en el desarrollo de procesos concurrentes se enfoca en mitigar al máximo estos problemas de sincronización y gestión de recursos.

### 2.2.2. Modelo computó paralelo

Paralelismo es una palabra que hoy es usada en muchas situaciones, y su noción general la tenemos medianamente clara, dos o más procesos ejecutándose al mismo tiempo. Sin embargo, no existe una única forma de paralelismo. Quizá la forma más común en la que pensamos sobre paralelismo es la de múltiples *CPUs* que pueden ejecutar múltiples instrucciones al mismo tiempo [5].

Dependiendo de la arquitectura, se pueden lograr diferentes formas de paralelismo. Sin embargo, con la evolución de la computación, la distancia entre las diversas arquitecturas se fue reduciendo. Cada una fue adoptando elementos de otras para volverse más eficiente, por lo que hay formas de paralelismo que podrían parecer exclusivas de las arquitecturas Harvard, pero que también se pueden ver en procesadores con arquitectura von Neumann [5].

El **pipelining** es uno de ellos, especialmente común en arquitecturas Harvard, pero también habitual en arquitecturas von Neumann. Es 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 sí 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 manera que, cuando una instrucción está en la etapa de cálculo aritmético, otra debe estar 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 abarca 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 tiempo, pero el manejo de los recursos es realmente complicado para el sistema operativo debido a los recursos compartidos que se tienen y que se están modificando [5, p.421].

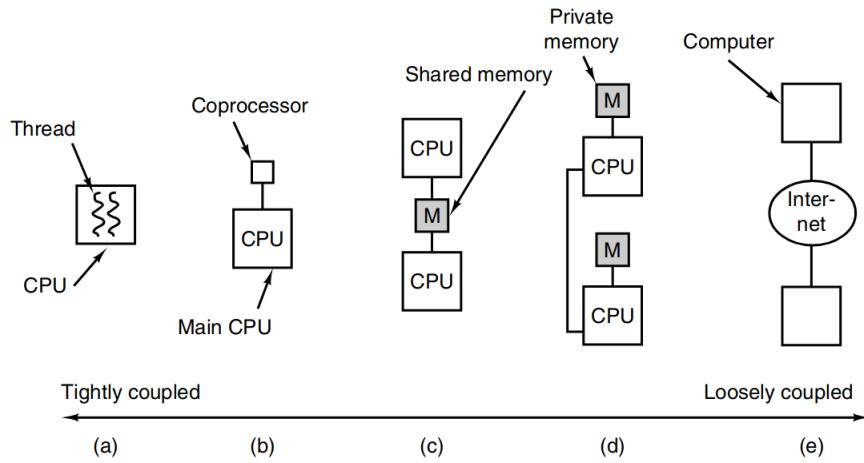


Figura 2.2: Obtenida de (Tanenbaum, 20202), a. Paralelismo On chip, b. CoProcesador, c. Multiprocesador, d. Memorias independientes, e. Múltiples computadoras.

Ahora, si pensamos en la alternativa de múltiples procesadores, aún puede haber variantes. Una de ellas es el concepto de memoria compartida en múltiples procesadores, que data de la década de 1970. En este modelo, 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 caché individual para cada procesador, o podría ser que esta también fuese compartida. Otra alternativa es 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, utilizando un solo procesador, o varios, tratando de explotar al máximo el tiempo de ejecución, generalmente utilizando la línea de producción para conseguirlo. Luego, en la parte b, se observa un procesador principal y uno auxiliar para ciertas ejecuciones. Este es un concepto realmente interesante que ya mostraban los *mainframes* de la serie *IBM 360*, con procesadores independientes para las entradas y salidas, ya que suelen

ser algunos de los procesos más complejos que tiene una máquina [32].

Finalmente, la parte *e* nos presenta el paralelismo con diferentes computadoras conectadas a través de internet, el cual es el que ofrece el nivel más alto de paralelismo al involucrar más de una computadora realizando cálculos de forma síncrona: es decir, realizando un cómputo distribuido [32].

Las computadoras actuales generalmente utilizan todos estos recursos, ya que si la concurrencia y el *pipelining* son técnicas para eficientar al máximo un solo *CPU*, cuando se tienen varios *CPUs* lo ideal es utilizarlas en todos para optimizar sus capacidades; con la intención de maximizar la eficiencia de cada uno y luego maximizar la eficiencia de su sincronización. Claramente, no se trata solo de combinar diversas técnicas y formas de paralelismo; cada procesador o computadora tiene características que permiten ciertas formas de paralelismo o concurrencia diferentes, y su implementación depende del diseño que estos posean [32].

## 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 específicas el. 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 a un bajo nivel, y cómo 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 derecha a izquierda. 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. Y dado que sigue una arquitectura von Neumann, los datos y las instrucciones se almacenan en la misma memoria, la cual no puede distinguir directamente entre estos; solo la *CPU* puede [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 (*d2*), es el código de operación, y los demás (*d1* y *d0*) 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*; mientras que, para la última dirección tenemos una especie

*EEPROM* (*Electrically Erasable Programmable ROM*), porque siempre tendrá un número entre el #800 y el #899, debido a que el dígito  $d_2$  se mantiene fijo, pero cambian los dos últimos. 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, necesitamos darle conexiones a la memoria para que pueda hacer uso de él. 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 cuatro tipos de buses identificados con colores diferentes; los de color verde transfieren datos e instrucciones, ya que son los que tienen mayor capacidad de transferencia (tres dígitos); después, están los de color morado, que transfieren direcciones de memoria (ocupan dos dígitos); para los códigos de operación, se utilizan los de color azul, que solo ocupan un dígito; por último, en color rosa están los buses de control, 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 a la *CPU* [3].

En la imagen 2.3, podemos ver un diagrama más centrado en los buses y sus conexiones; en la parte inferior izquierda, los vemos con la etiqueta que indica su significado. 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 morado (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 transmitir códigos de operación y los buses rosas para enviar señales de activación a los componentes que utiliza.

La *CPU*, es la unidad donde se concentran todos los elementos que hacen posible el cómputo de las instrucciones y datos que se almacenan en la memoria principal. Está conformada por los elementos que están dentro de la línea punteada en rojo en la imagen 2.3. Dada la importancia de cada elemento dentro de la unidad de procesamiento central (*Central Processing Unit*), se hará un repaso individual de cada uno.

Revisaremos cada elemento de la *CPU* analizando sus funciones. Para ello, es preciso

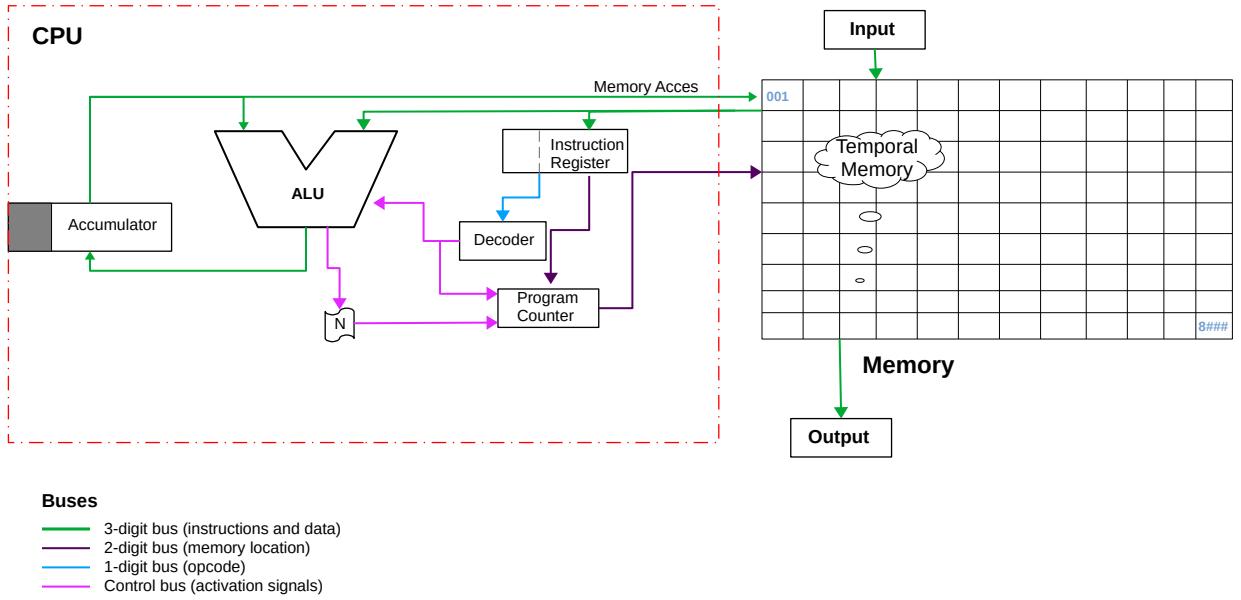


Figura 2.3: Arquitectura de CARDIAC, concepto original Bell Labs & Jorge Vasconcelos

tener claro el concepto de **ciclo**, el cual podemos entender vagamente como la situación en la que regresas a la posición inicial una y otra vez. Un ciclo comienza cuando el *Program Counter*, o contador de programa, apunta hacia una dirección; la primera sería la #00. Continúa, cuando la CPU toma el contenido de esa dirección y, por medio del bus verde, la transporta hacia el *instruction register*, o registro de instrucciones, que se encarga de separar el dígito a extrema izquierda,  $d_2$ , para enviarlo al *decoder*, o decodificador, por medio de un bus azul; este se encargará de decodificarlo para transmitirle esa instrucción a la unidad aritmético-lógica. Cuando el contador de programa salte a otra dirección de memoria, se habrá terminado un ciclo y comenzará otro [3].

Por otra parte, el mismo registro de instrucciones transfiere al contador de programa los dígitos  $d_1$  y  $d_2$ , en caso de que los necesite para saltar a una dirección específica. Si los

requiere, el decodificador le envía una señal, por medio del bus rosa, que le indica saltar a una dirección particular [3].

Es turno de la *ALU*, que realiza todos los cálculos de la computadora, según lo indicado por el código de operación. Si el resultado que va a guardar en el acumulador es negativo, envía una señal a *Negativo (N)*, una especie de booleano que indica si un número que ha sido depositado en el acumulador es o no negativo. Lo último que realiza la *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 donde se almacenarán todos los cómputos que realice la *ALU* [3].

En caso de que la unidad aritmética requiera información de la memoria para ejecutar una instrucción, puede recuperarla directamente por medio de los buses verdes. Dado que, si se requiere información de la memoria, significa que la instrucción así lo indicaba, por lo tanto, los dígitos *d1* y *d0*, del dato recogido por el registro de instrucciones, tienen la dirección de donde se debe recuperar. Únicamente si la *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, se termina un ciclo cuando el contador de programa salta, que puede ser porque una instrucción se lo indique, o bien porque la *ALU* terminó sus cómputos, y entonces el contador de programa puede continuar a la siguiente dirección de forma incremental sin ningún 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, posteriormente, se pueda realizar la operación de impresión desde un espacio de la memoria principal [3].

### 2.3.2. Funcionamiento y lenguaje en CARDIAC

Ahora, conocemos los elementos de este modelo y cómo interactúan, pero lo visitamos de cierta forma “a ciegas”, no conocemos el lenguaje que utiliza esta máquina y, como mencio-

namos 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 para *CARDIAC* en código máquina (Código de operación) y en ensamblador (Mnemotecnia), para lograr una mejor comprensión. Analizaremos cada una de las instrucciones, presentadas originalmente en [3], para entender la razón de su existencia.

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*.

Para empezar, necesitamos establecer una conexión entre el usuario y la máquina, de modo que el usuario, a través del dispositivo de entrada, pueda transmitir información a la memoria principal de la máquina; esto se logra con el código *0/INP*. Por ejemplo, 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. Podemos notar 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 se realiza con el código *1/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 a la máquina al utilizar el código *012*, y que se guarda en la dirección #12, es *005*; entonces, al ejecutar la instrucción *112*, el valor *005* se cargará 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, donde se encuentra el dato *001*, que podemos utilizar con el código *2/ADD* de la siguiente forma: *200*. Esto indica que hay que sumar el contenido de la dirección #00 a lo que tiene el acumulador, obteniendo un *006* en el acumulador si se ejecuta posterior a la instrucción *112*.

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

Tabla 2.2: Programa principal (In-Util).

Para seguir analizando los códigos de operación disponibles, tenemos un código en la tabla 2.2, donde podemos visualizar los códigos, su equivalente en ensamblador, y cómo afectan al acumulador. Este programa empieza en la dirección #20, por lo que podemos seguir una ejecución secuencial de cada instrucción empezando en #20. Si nos damos cuenta, las primeras tres instrucciones son las que revisamos en el párrafo anterior, y el estado del acumulador, en la tercera fila, expresa precisamente el valor mencionado: *006*.

El contador de programa se ubica en la dirección #22, desde donde se moverá a la #23, en la que utilizaremos la instrucción *7/SUB* para restar. Restaremos el número que cargamos en la dirección #12 de lo que tenemos en el acumulador, de hecho lo haremos dos veces seguidas para conseguir un número negativo: *-004*. Si lo viéramos en operaciones

aritméticas, la operación sería la siguiente:  $6 - 5 - 5 = -4$ .

Los números negativos en este modelo son especiales. Como se habrá observado, coloqué el signo a la izquierda para lograr una mayor simplicidad del modelo en el tratamiento de los números negativos, por lo que ocupan el mismo espacio que los positivos. El signo siempre estará más a la izquierda que cualquier otro dígito, y dado que las direcciones no pueden ser negativas, no nos encontraremos con un caso del estilo 1-42. Siguiendo con las simplificaciones, el modelo tiene la libertad de considerar al cero como positivo, situación que será importante con las siguientes instrucciones.

El contador de programa avanza a la dirección #25, en la que se suma uno al acumulador, y en la siguiente (#26), utilizamos el código *3/BLZ* en la instrucción 325 para obtener un salto condicional. Es decir, si el contenido del acumulador es menor a cero, salta a la dirección #25, formándose así un bucle en el que el acumulador cambiará de valor de uno en uno hasta llegar a cero después de cuatro ciclos. Esto ocurre debido a que pasar por la dirección #25 significa aumentar en uno el valor del acumulador, por lo tanto, después de cuatro ciclos el valor será positivo y el condicional dejará al contador de programa seguir secuencialmente.

El siguiente paso es entender el funcionamiento de la operación de salto *8/JMP*. Para esto, siguiendo el avance secuencial, vemos que en la dirección #27 se le suma uno al acumulador para tener un *001* en este registro. En la siguiente dirección (#28) está la operación de salto, que indica saltar la dirección #80, donde se encuentra una **subrutina**, un programa “pequeño” generalmente usado por otros programas para reproducir un resultado. En este caso, la tarea de la subrutina es sumar un seis a lo que tenga el acumulador en el momento que empiece su ejecución. Si lo que queremos en nuestro caso es sumarle un seis al acumulador, la forma más fácil de hacerlo es saltando a esa subrutina. Una vez que la subrutina termine, regresará a la dirección siguiente desde donde saltó el contador de programa, en este caso, regresará a la #29.

Este funcionamiento se logró fácilmente porque la instrucción *JMP*, al momento de saltar, guarda la dirección de memoria desde la cual saltó en la dirección #99 con el sufijo *8*. Si miramos la tabla 2.3, donde se encuentra esta subrutina, notaremos que lo primero que hace es cargar en el acumulador el contenido de la dirección #99, que en este caso es *828*, pues saltó desde la dirección #28. Posteriormente, se le suma un uno, para así tener la instrucción

*829*, que indica saltar a la dirección *#29*. Para asegurar ese regreso desde la subrutina, en la dirección *#82* se indica que se guarde el resultado del acumulador, el *829*, en la última dirección de la subrutina, de modo que, cuando terminen todas las adiciones, se realizará un salto 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 desde la que saltó más uno, es decir, si saltó desde la dirección *#28*, lo que se guardará en la *#99* será un *829*. En nuestro ejemplo particular, esto ahorraría código; sin embargo, por los usos que se le dará en los modelos siguientes, decidí que sería más eficiente como lo he definido en esta sección. Esto se debe a que se obtiene mayor precisión al conservar la información directa, sin modificaciones automáticas. 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.

Continuamos con el programa principal de la tabla 2.2, el contador de programa apunta a la dirección *#29*; y el valor en el acumulador es de *006*. La operación que se realiza en la *#29* es una suma para dejar el valor del acumulador en *012*, valor que será muy interesante para analizar el siguiente código de operación: *4/SHF*.

Dejé este código para casi el final porque es el menos intuitivo, 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 que en una decimal por las operaciones entre bits que permite 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, observamos el efecto de aplicar esta operación con la instrucción *432*, que cambia el número *012* por *020*. Esto sucede porque 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. Es importante destacar 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** que las celdas de la memoria; en este caso, el acumulador puede almacenar hasta 4 dígitos.

Si consideramos lo anterior en la instrucción *432*, lo primero se indica 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 de forma permanente. Si ahora aplicamos el desplazamiento de dos lugares a la derecha, indicado por el dígito *d0*, terminaremos con 0020 en el acumulador. Como podemos ver, el 1 se ha perdido, y aunque desplacemos en el sentido contrario (derecha), si este se ha quedado fuera del espacio del acumulador, no podrá recuperarse. Para simplificar, y dado que prácticamente no se usa el cuarto dígito, dejamos indicado el resultado solo como 020 en la tabla 2.2.

Para almacenar de manera persistente los datos, en la dirección #31 tenemos la instrucción *613* para guardar la información del acumulador, que emplea el código *6/STO* para indicar que en la dirección de memoria #13 se guardará dicho valor. Posteriormente, el código *5/OUT* establece la conexión con el dispositivo de salida para exportar los resultados almacenados en la dirección #13, ya que la instrucción *513*, en la dirección #32, indica con sus últimos dos dígitos la dirección de memoria de donde se tomará la información para exportar.

Finalmente, el programa se termina con el código *9/HLT* en la instrucción *900*, marcando el final del programa y reiniciando el contador de 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, excepto en el caso de *SHF*, en la cual tienen un funcionamiento especial.

Con este lenguaje, simple, pero eficaz, podemos construir cualquier programa que queramos, puesto que al tener ciclos, condicionales y la posibilidad de realizar las operaciones aritméticas básicas (suma y resta), podemos decir que es Turing-completo. Por lo tanto, nuestras limitantes no están en el lenguaje, sino en la memoria y el poder de procesamiento.

# Capítulo 3

## Evolución del Modelo

Como pudimos ver en los capítulos anteriores, *CARDIAC* es sumamente útil para explicar aspectos importantes de la computación y cómo se organizan sus componentes; sin embargo, no podemos negar que hoy en día es un tanto insuficiente para representar a las computadoras más modernas que cuentan con concurrencia y paralelismo sin los cuales no las entenderíamos, pues 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 Saul Fingerman y David Hagelbarger, es necesaria [3]. Esto implica diferentes retos, retos que se abordarán en este capítulo.

Aprovechando la disponibilidad actual de las computadoras, he realizado una simulación en *Java* para ilustrar las mejoras realizadas al modelo original, facilitando la demostración de la interacción entre 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 crear 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 del programa donde se puede seleccionar la máquina virtual que queremos probar. Como se podrá notar, he agregado la *E* (de electrónico) como sufijo de las tres máquinas virtuales, resaltando su aspecto electrónico en contraste con el de los años 60.

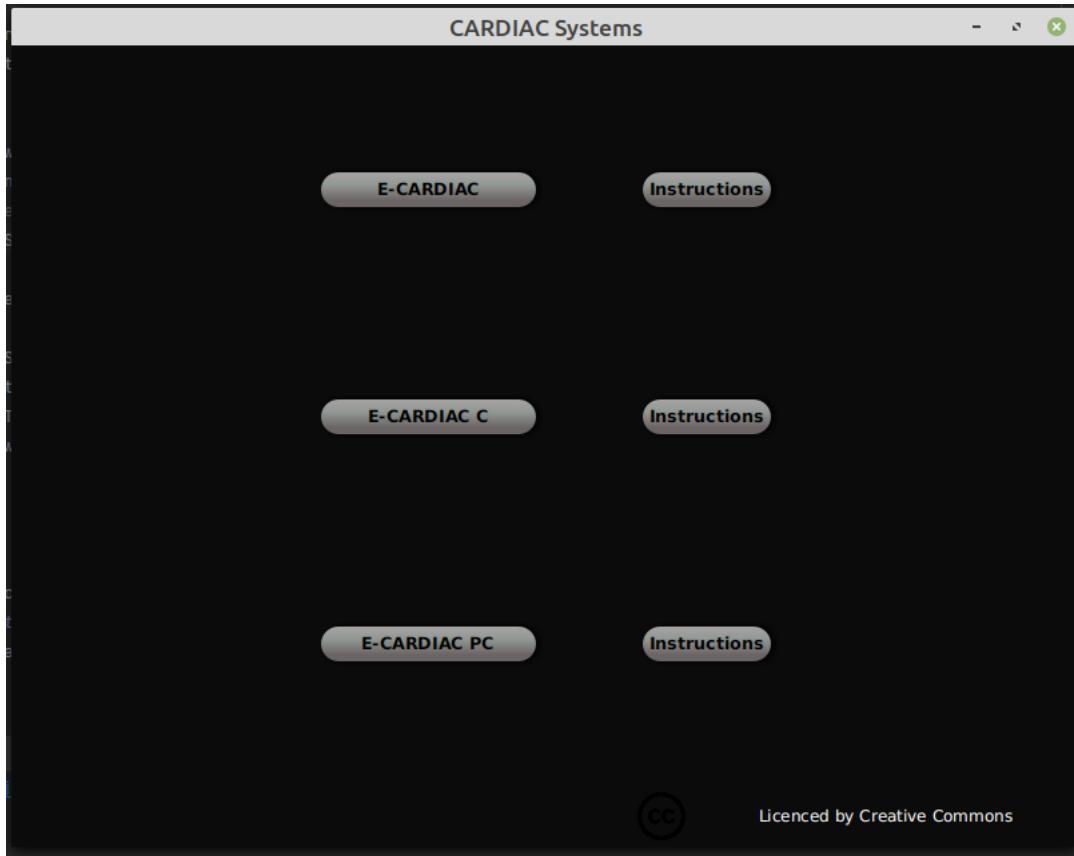


Figura 3.1: Inicio para selección de máquinas virtuales

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

Una vez que entremos al software de simulación y veamos 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 prácticamente replica el modelo original llevado a un software de simulación. En términos generales, las tres máquinas compartirán una estructura similar. Por lo tanto, una vez que conozcamos el funcionamiento de esta primera, será muy sencillo entender el funcionamiento de las otras dos, aunque estas tengan más componentes.

Para comenzar, será útil visualizar el diagrama de la arquitectura de CARDIAC, que se puede apreciar en la figura 3.2 y se comentará conforme se haga referencia a sus componentes. En una vista 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

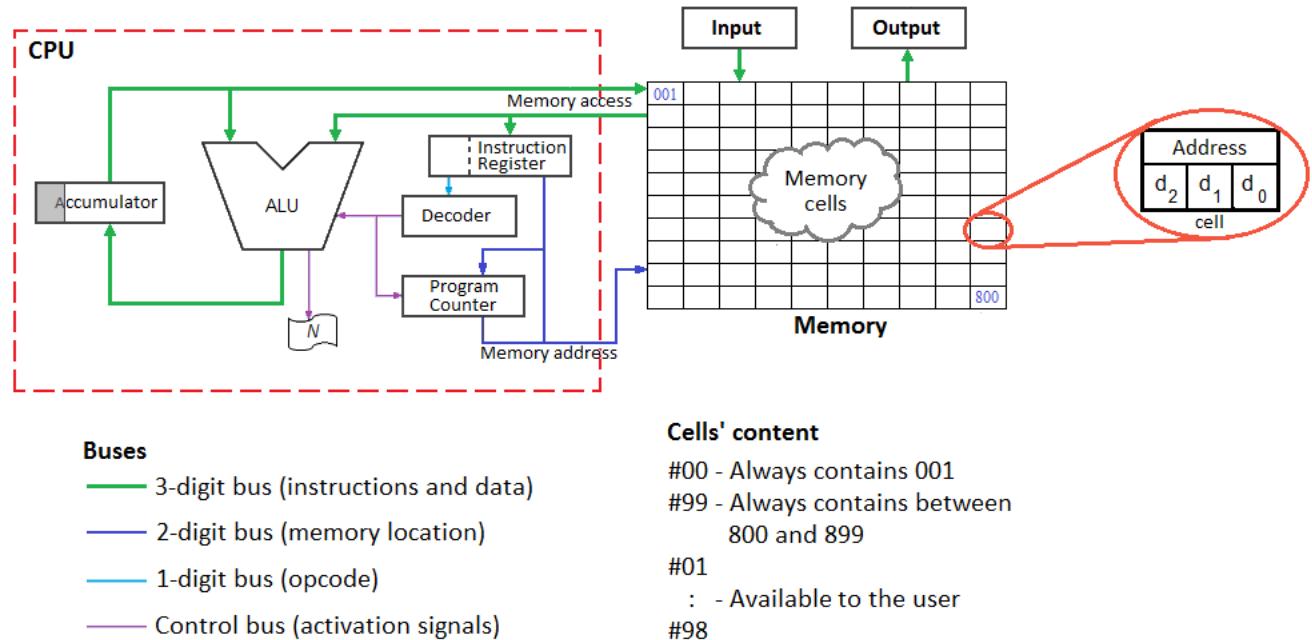


Figura 3.2: Arquitectura de CARDIAC por Jorge Vasconcelos, 2018

En la figura 3.3, podemos ver el esqueleto de lo que será nuestra máquina, que aún se encuentra apagada, pero que podemos encender al hacer clic sobre el botón *start*. Pero antes de hacer clic ahí, podemos ver que en esa barra principal tenemos varias opciones, una de ellas está a la izquierda, un símbolo de casa y un botón que dice *CARDIAC Systems*, el cual nos permitirá regresar a la página de inicio.

Continuando en el centro, tenemos tres botones, el que ya vimos para encenderla, y otro para pausarla; algo poco común en una máquina de verdad, pero bastante usual en los programas de máquinas virtuales. Este botón de pausa permite detener el funcionamiento completo sin afectar los procesos internos, analizar su estado y luego 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 cantidad de celdas con las que la máquina funcionará, es decir, la memoria disponible; y la segunda permite elegir la velocidad de cada ciclo de la máquina. De esa forma podemos decidir cuánto va a tardar cada ciclo en ser completado con el fin de observar su comportamiento.

En la figura 3.4, podemos observar las listas desplegadas con las opciones que tienen disponibles. Como podemos ver, hay distintas velocidades; la opción *slow* permite observar el proceso con más detalle, mientras que la opción *instant* ofrece un resultado inmediato. 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. Al elegir la cantidad de memoria y luego encender la máquina, esta se configura en su arquitectura para trabajar con dicha cantidad y recibir instrucciones con direcciones más grandes.



Figura 3.4: Listas desplegables de E-CARDIAC

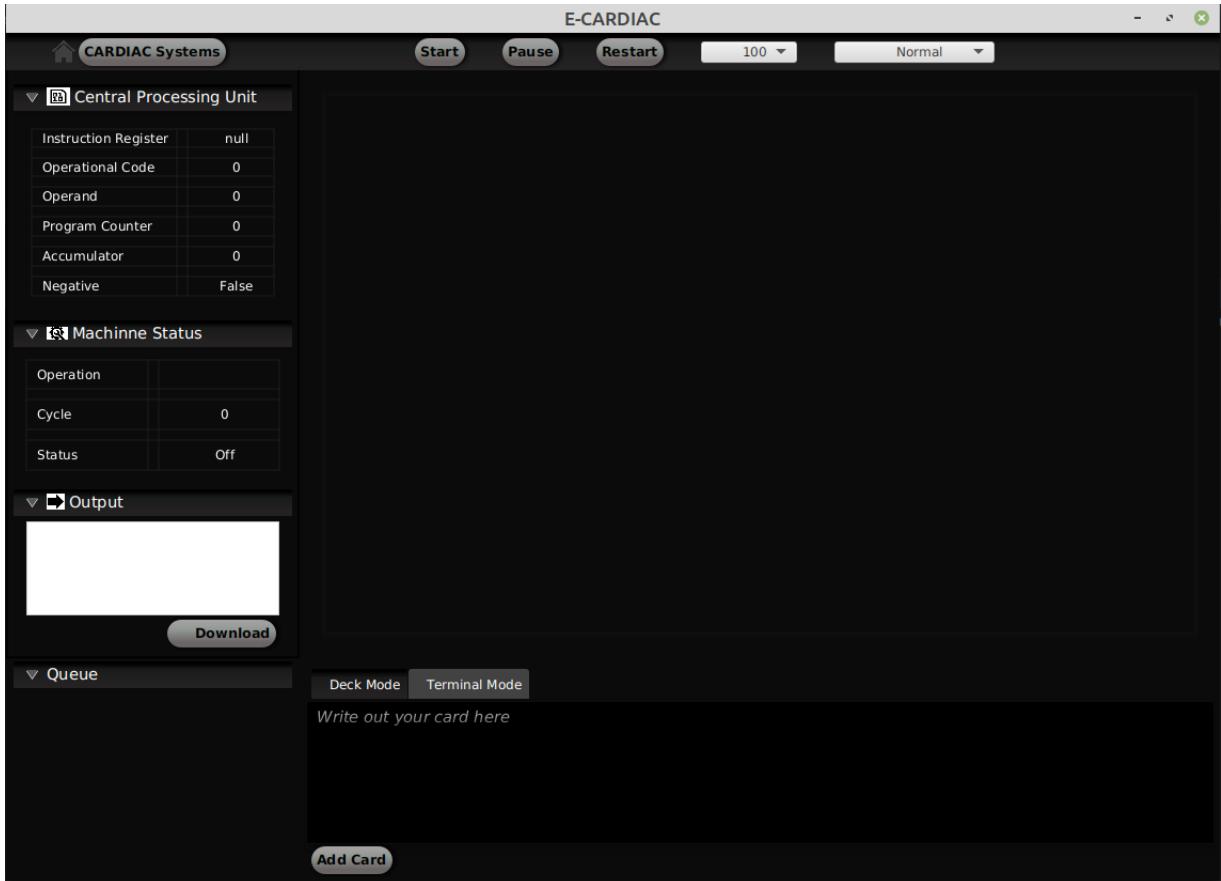


Figura 3.3: Pantalla de inicio de E-CARDIAC

## Unidad central de procesamiento

Si continuamos 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)*, o unidad central de procesamiento; 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 llamado *queue*, que servirá como cola de espera para la ejecución de las instrucciones.

En la primera sección, que es la *CPU*, podemos notar el registro de instrucciones (*instruction register*), que registrara las instrucciones que viajan a través del bus verde desde la memoria a la *CPU*. Luego, se encuentran el código de operación (*Operational Code*) y el operando (*Operand*). Estos son la separación del dato que llegó al registro de instrucciones como uno sólo, separándose en dos partes.

Posteriormente, se encuentra el contador de programa (*Program Counter*), el componente que se encarga de indicar la dirección de la memoria que será leída. 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 la dirección que es “apuntada” por el contador será transmitido a la *CPU*.

Más abajo, se encuentra el acumulador (*Accumulator*) y la bandera para saber si un número es negativo (*Negative*). El acumulador contendrá los resultados que arroje la unidad aritmético-lógica (*ALU* en diagrama de CARDIAC), o bien algún valor que sea cargado directamente desde memoria.

### Estado de la máquina y dispositivo de salida

En la sección *Machine 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*). Este último, puede ser causado por diversas razones que hagan que *CARDIAC* realice operaciones indebidas, como el intentar pasar un negativo como instrucción.

En esta misma sección está el apartado *Operation*, que indica la operación que está ejecutando la *CPU*, junto al apartado *Cycle*, que contiene el ciclo en el que se encuentra la máquina.

La tercera sección muestra la salida en forma de lista, como podemos ver en la figura 3.5. Esto emula 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 de la computadora física del usuario.

### Entrada de datos e instrucciones

Toda la parte inferior de la máquina virtual está relacionada, puesto que en *Queue* está la lista de instrucciones en cola por ser leídas, como en la figura 3.8 se muestra. El usuario podrá añadir estas instrucciones a la cola desde la pestaña *Deck Mode (modo tarjeta)*, y se podrán

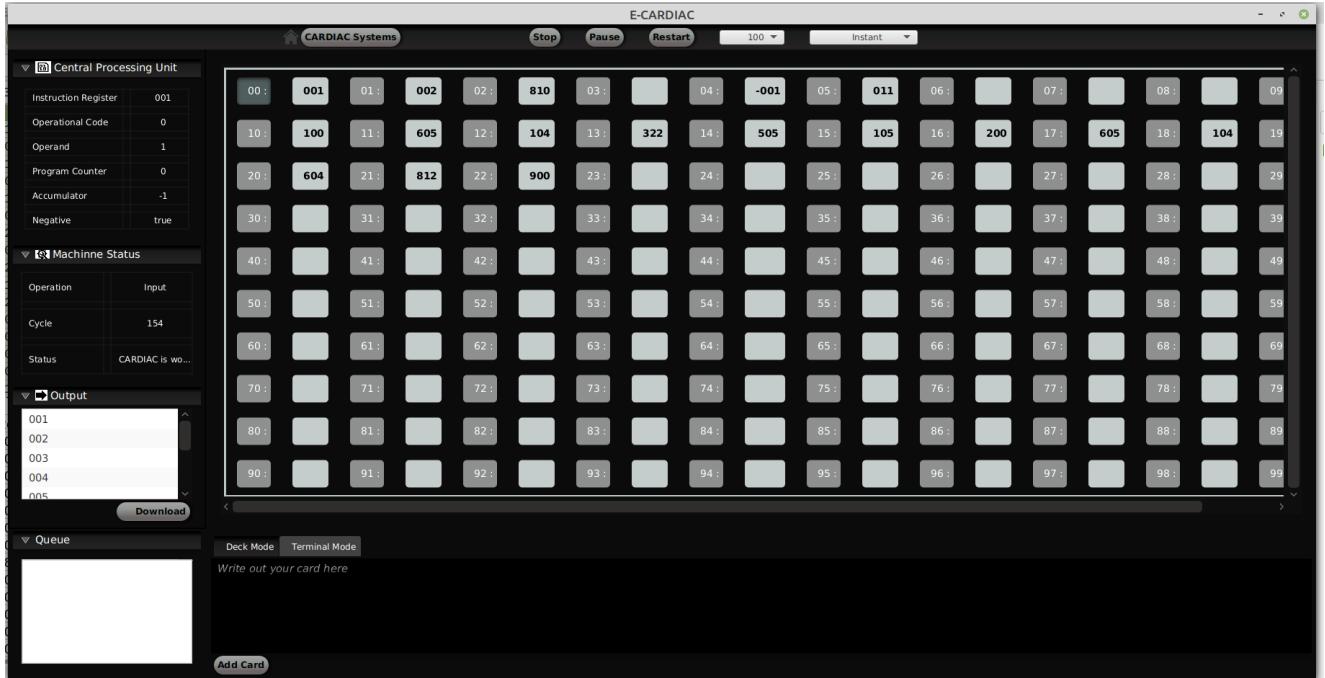


Figura 3.5: E-CARDIAC Muestra de Output

cargar desde ahí con solo escribirlas en forma de lista y hacer clic en *Add Card* (agregar tarjeta). En la figura 3.6, podemos ver una lista de instrucciones previa a ser añadida a la cola.

Por otra parte, si se desea añadir instrucciones o datos de forma individual, uno a uno, se cuenta con la otra pestaña que dice *Terminal Mode* (modo terminal). Sin embargo, para hacer uso de esta pestaña es necesario que exista una instrucción indicando que se espera un dato en una celda de la memoria. Por ejemplo, si se inicia la máquina, la primera dirección apuntada será la #00, que espera un valor del usuario para cargarlo en la celda #01. En esa situación, se puede usar el modo terminal, como en el caso que nos presenta la figura 3.7.

En caso de no tener una solicitud de carga de información, no se permitirá agregar datos de esa manera. Esto sucede debido a que este modo transfiere directo los datos o instrucciones a la memoria, a diferencia del modo de tarjetas, que siempre las manda primero a la cola de espera, la cual las transferirá posteriormente a la memoria principal cuando se solicite.

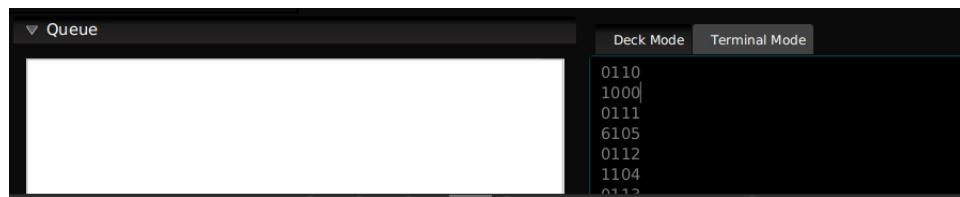


Figura 3.6: E-CARDIAC Carga masiva por tarjetas

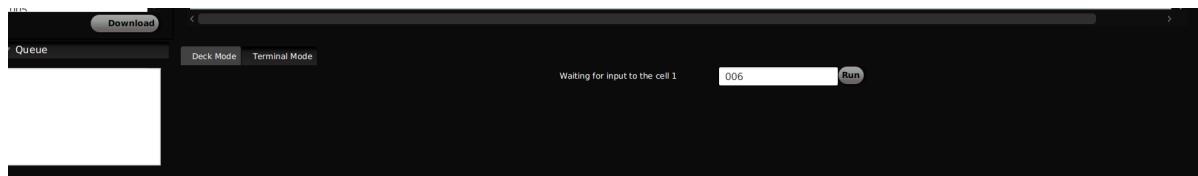


Figura 3.7: E-CARDIAC Carga individual de instrucciones

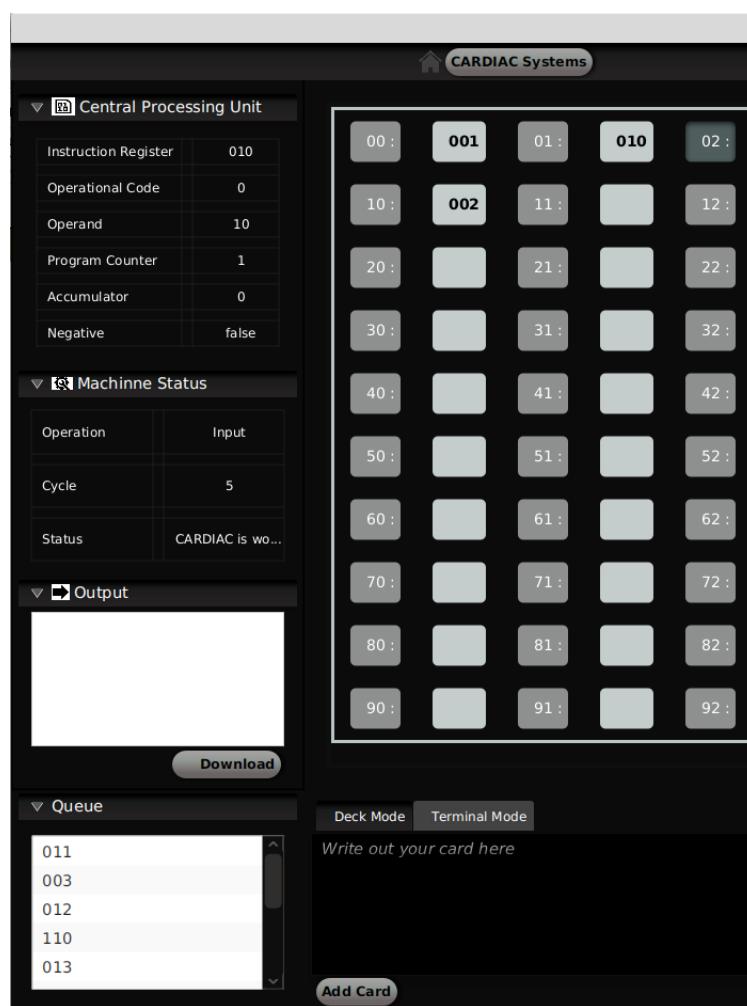


Figura 3.8: E-CARDIAC Cola de instrucciones/datos

## Encendiendo E-CARDIAC

En varias de las imágenes mencionadas en párrafos anteriores, se mostró cómo se ve la memoria principal. Veamos ahora la figura 3.9, para ver la máquina cuando acaba de encender y poder centrarnos en la memoria principal, compuesta de 100 celdas numeradas del #00 al #99 con los datos por defecto cargados. 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 cuyo contenido se está transmitiendo a la *CPU*, se usa el color azul marino, como en el caso del ejemplo, que indica que se está leyendo la instrucción en la dirección #00.

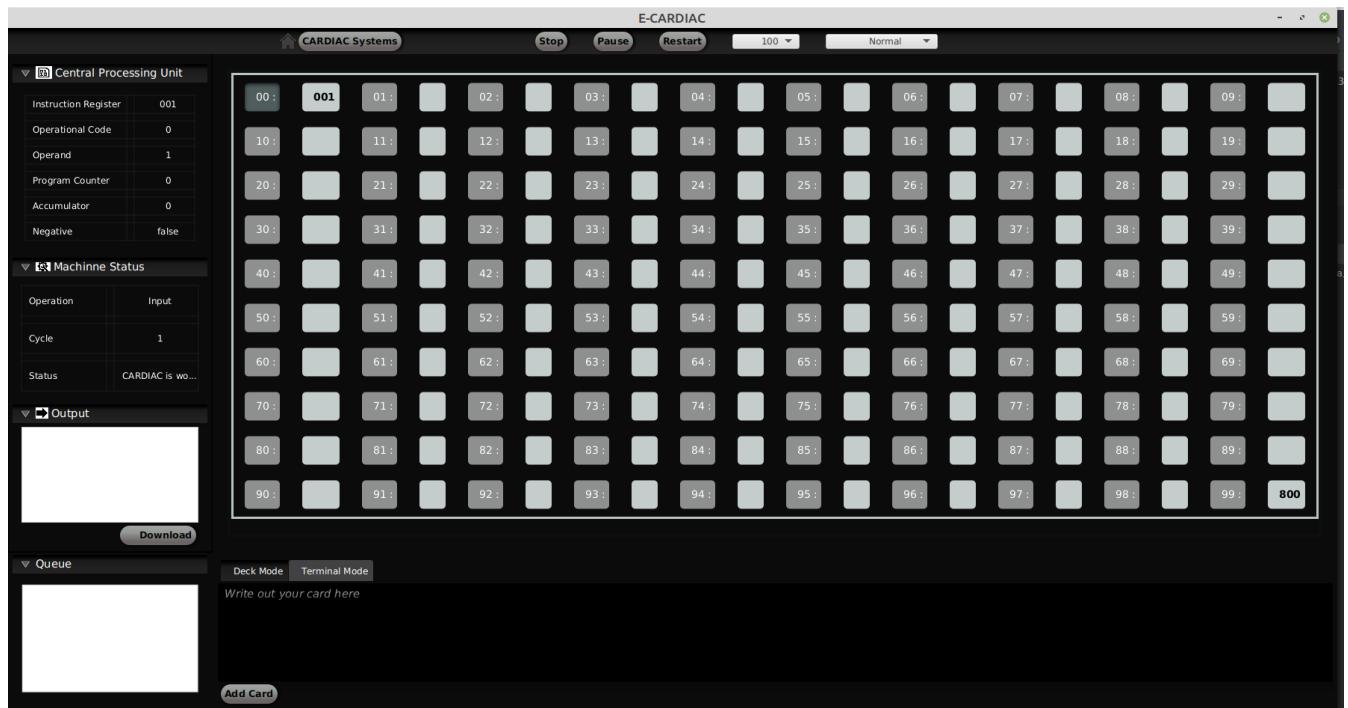


Figura 3.9: E-CARDIAC después de iniciar sus funciones

De esta forma, terminamos el recorrido por la máquina virtual de **E-CARDIAC**, que nos permitirá experimentar con más soltura la programación en un lenguaje ensamblador. La siguiente evolución del modelo requerirá de un aumento en la memoria de CARDIAC, lo cual es muy fácil de aplicar en 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. Además, al agregar esa “C” al final del nombre del modelo original, se puede denotar que la diferencia entre estos dos modelos es la concurrencia. *E-CARDIAC C*, siglas en inglés de ayuda ilustrativa de cartulina electrónica para la computación concurrente, tiene la intención de ser un modelo que represente la organización de una computadora capaz de tener operaciones concurrentes y de mostrar cómo interactúan las distintas partes de la computadora para lograr dicha 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 tanto en el hardware como en el software, además de diseñar un sistema operativo básico que permita 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 y ejecutada por nosotros. De cierta manera, seríamos nosotros mismos ese sistema de administración. En [3, p. 42], Fingerman y Hagelbarger nos presentan un sistema de carga de programas para el modelo original, un cargador (*bootloader*). Fue diseñado para poder escribir en “tarjetas” un programa y que este se cargue en la memoria de CARDIAC sin la necesidad de colocar literalmente las instrucciones una a una en los espacios de memoria; sin embargo, 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 esas tareas será un **sistema operativo mínimo**. Y necesita el calificativo de “mínimo” porque 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 pretende ser una capa de abstracción tan marcada; aunque tendrá algunos aspectos que pueden ser

considerados como tal, no será su principal cualidad. Por tal razón, lo nombraremos como sistema operativo mínimo, o *SOM* en el resto del texto.

Para 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 también a un cambio en el lenguaje, que está 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, lo que a su vez, aumentaría la cantidad de buses.

Además, al ser un programa que requiera los privilegios de control más elevados sobre la computadora, quizás requiera piezas de hardware adicionales que en su versión original 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 esta forma, cada dirección posible tendrá tres dígitos. Así, un número completo de cuatro dígitos puede representar tanto un valor numérico 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 cómo serán los buses: los de color verde, que son los de mayor capacidad, transmitirán instrucciones y datos; los que están en color morado, sólo las direcciones; y los que están en color azul, únicamente 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, el cual tiene el mismo significado que en el modelo original, pero con un cero extra por el crecimiento en el tamaño de la memoria. El contenido al final de la memoria tiene una situación similar, ya que sólo

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 donde se ejecutó una instrucción de salto, al igual que en el modelo original.

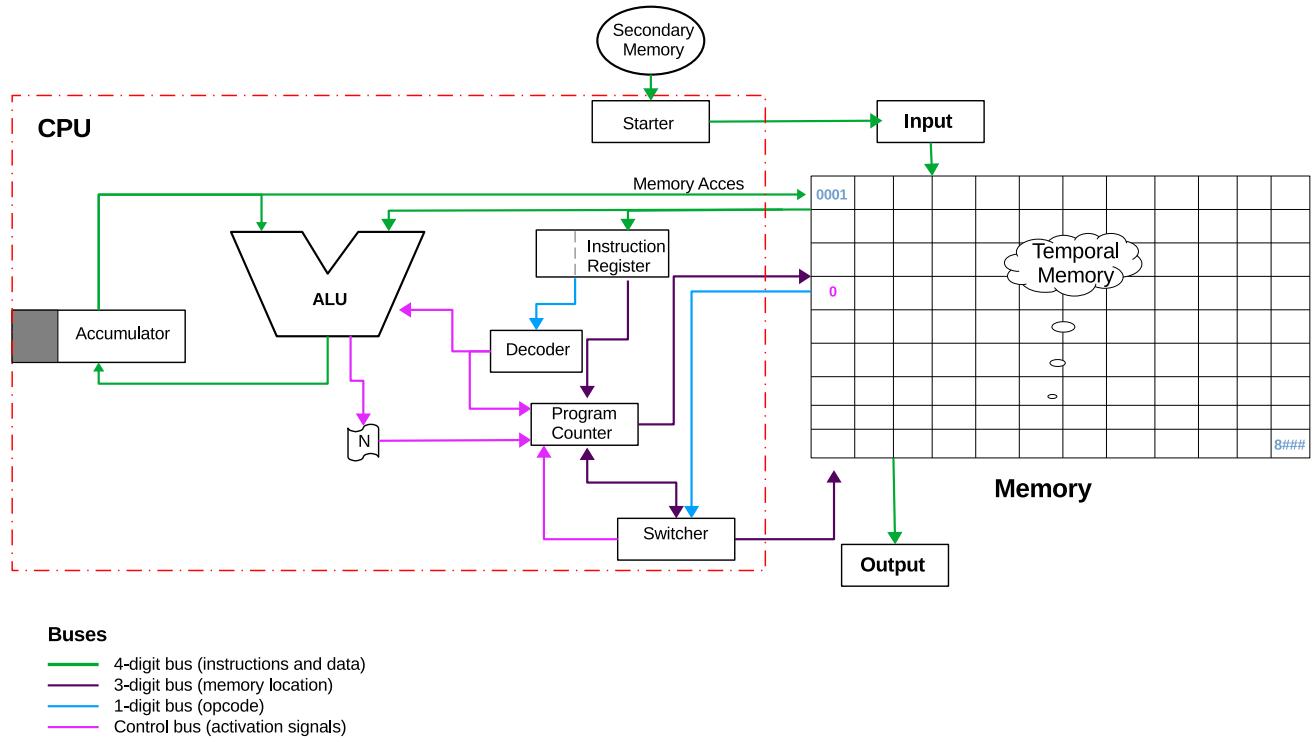


Figura 3.10: Diagrama de Arquitectura de E-CARDIAC C

Esto permite mantener varios elementos del modelo original sin cambios. Pero, podemos notar tres 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. Este tiene por nombre **starter** (iniciador), y de igual manera, está conectado con otro elemento nuevo: la **secondary memory**, o memoria secundaria.

La memoria secundaria es la representación de lo que sería el disco duro en las computadoras actuales, pero más limitada, porque sirve como un almacenamiento puntual para el sistema operativo, como si de una tarjeta de programas se tratase. Esta tiene conexión con

el iniciador, mismo que se conecta al dispositivo de entrada, para que al iniciar la máquina lo primero que haga sea agregar las instrucciones guardadas en la memoria secundaria a la cola de ejecución.

De esta manera, podemos mantener simple el modelo. Pero, a su vez, establecer una conexión con otra fuente de información diferente a la memoria principal, y ser capaces de cargar el sistema operativo mínimo de una forma más natural sin que esté presente 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 de la *CPU*. Este elemento es la solución al problema de incluir un programa que requiere los privilegios para controlar las ejecuciones, ceder el control a otros programas, y poder recuperarlo en un momento particular.

Como se puede observar en la imagen 3.10, el comutador tiene cuatro diferentes conexiones, siendo uno de los que más posee. 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**. Esta dirección tiene que ser establecida en la configuración inicial de la misma arquitectura. Este salto lo hace siempre y cuando en la dirección #003, el contenido sea 1, pues es la bandera que indica si es tiempo de ejecución del usuario y por lo tanto cada N ciclos los recursos deben regresar 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 realiza a través de un bus 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, indicarle al contador de programa cuándo debe regresar los recursos al SOM, y a qué dirección debe saltar para devolver el control de los recursos. El *comutador* tiene un contador propio, el cual sigue al contador de programa, pero que se reinicia cada que la bandera de saltos, que se encuentra en #003, cambia de valor. Para todo esto, se requiere un bus de color morado para transmitir las direcciones y datos, y uno de color rosa para mandar la señal de interrupción al contador de programa para que detenga su proceso natural.

Una razón adicional para esta conexión bidireccional es que el *conmutador* recibe del contador de programa la última dirección apuntada por el proceso del usuario, antes de que fuera ejecutada. Esto sucede debido a que, en ese momento, al cumplirse los N ciclos, se da la instrucción de regresar los recursos al SOM, y entonces esa última dirección es transmitida al *conmutador*. La intención de esto es que, 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, y cuando regrese los recursos a ese proceso, pueda continuar justo en la instrucción que no pudo ejecutar.

El sistema operativo mínimo en su misma ejecución 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. La aclaración de “aproximadamente” es porque una vez que se cambia 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 el proceso. 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 de forma concurrente, el lenguaje también requiere modificaciones, en su mayoría mínimas, pero que van orientadas a funcionar en conjunto 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 ofrece. 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 interactuarán entre sí.

Para E-CARDIAC C agregamos un nivel adicional, la consideración del sistema operativo mínimo. Como podemos notar en la tabla 3.1, uno de los únicos dos cambios que se realizó es en la instrucción *halt*. Esta está diseñada ahora para funcionar con un sistema operativo, ya que, por sí misma, 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. Esto se debe a que ya no finalizará programas la instrucción, sino procesos, y es la razón por la cual esta instrucción no se entiende sin un sistema operativo. Con esto, podemos considerar que nuestro SOM es, en cierta medida, 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 de la memoria.
3	BLZ	Saltar si la información en el acumulador es menor que cero.
4	SHF	Mover d1 veces a la izquierda el contenido del acumulador y d0 veces a la derecha.
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 contador de programa 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*.

Si consideramos 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 cuando se trate de una instrucción. Para *shift* esto será importante, puesto que, en esta instrucción sólo importarán los dígitos *d1* y *d0*, ya que el *d2* no representará nada. Se tendrá en *d1* la cantidad de lugares que se desplazará el número a la izquierda, y en *d0* la cantidad de lugares que se desplazará a la derecha.

El último cambio a destacar es que el valor del contador de programa, que se guardaba en

la #99 cuando se hacía un salto con la instrucción *jump*, ahora será guardado en la dirección #999 para adaptarse a esta arquitectura. Aun con esto, para el resto de instrucciones el comportamiento no cambiará, sólo que ahora tienen tres dígitos a la derecha que representan la dirección, ya 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 este modelo. Sin embargo, como se mencionó, ya se había pensado en qué características necesitaba la computadora para poder implementar un SOM.

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

Ahora conocemos 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, en el desarrollo de este sistema operativo mínimo, al que llamaremos *SOMC*, está una lista de tareas que debe ser capaz de realizar para poder ejecutar de manera concurrente diferentes procesos; lo cual 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 lograrlo, era necesario que las direcciones no fuesen fijas en el diseño del programa. Por lo que usé variables para las direcciones, de forma que en el diseño pude tener una dirección como #s1, pero en la implementación esa dirección se transforma en #801. Así, si tengo una instrucción de la forma *1(s1)*, su mapeo 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 avancé, descubrí que no, y esta flexibilidad en cuanto a las direcciones me permitió seguir escribiendo para más de 100 direcciones sin tener que reescribir lo que ya tenía. Lo único que tenía que hacer es cambiar la dirección de inicio de mi programa, si antes #s0, la dirección de inicio del SOM, era #900, ahora sería #800, y el resto aumentaría secuencialmente.

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

ejecutan en la máquina. También hay una **zona de variables**, que contendrá variables o constantes de uso recurrente, para 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 así prepara ciertas variables para que cuando el núcleo del SOM esté en ejecución, todas las variables estén en la posición correcta. Las ventajas de segmentar el programa son que se pueden cambiar las direcciones de inicio de cada segmento sin afectar a los demás.

Adicionalmente, el núcleo del SOM se fraccionó dependiendo de las tareas que realiza en cada parte, separándolas en: añadir proceso, actualizar proceso, borrar proceso, ejecutar proceso y gestión del arrancador (*bootloader*). Para poder seguir todos estos conceptos con más claridad, diseñé un diagrama de flujo, que se puede observar en la imagen 3.11 , en la cual cada fracción del núcleo está con un color diferente. Por el tamaño del diagrama es difícil apreciar lo que dicen las letras, sin embargo, en cada sección que requiramos del diagrama se hará un acercamiento. 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, que 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. En la siguiente columna 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 tenemos la descripción completa de la instrucción, en caso de que esta sea necesaria.

Cada segmento del SOM tendrá variables de dirección diferentes. Veamos el caso del preámbulo, donde las variables que indiquen las direcciones empezarán por una *e* y continuarán de forma serial. En el recuadro de la imagen 3.12 podemos ver cómo se usan. Si nos fijamos en la fila donde está la dirección #e10, veremos la instrucción *1(e(0))* que indica

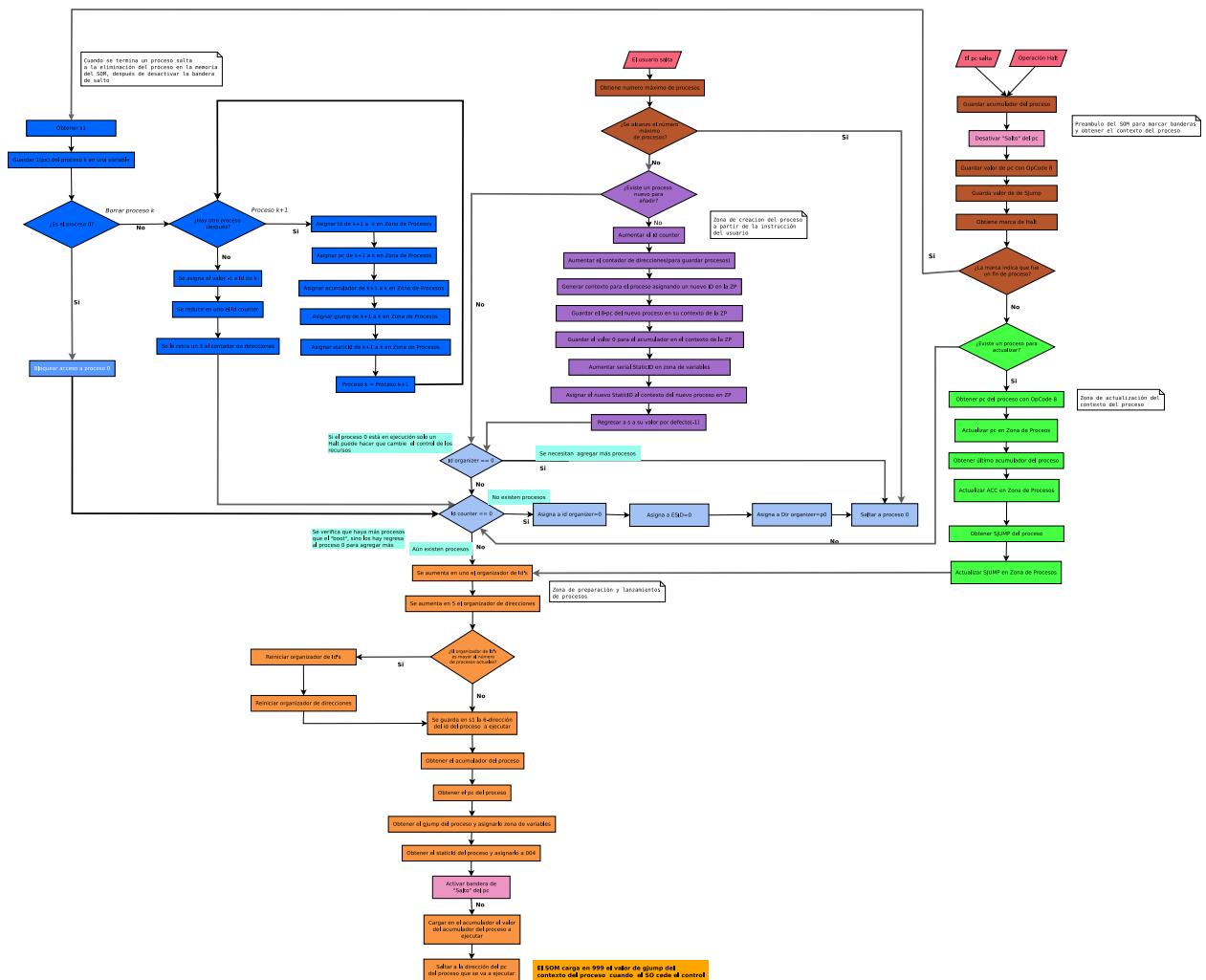


Figura 3.11: Diagrama de flujo de SOMC

que se cargue el contenido de la dirección #e0 en el acumulador. En la implementación, la variable de dirección será 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 empezarán por *p*, en la zona de variables del sistema empezarán 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 nos indica que trabajará con un valor de la zona de variables del sistema, debido a la letra *c* que aparece 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 brindan más información para entender mejor su funcionamiento.

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>				
<i>Cambiar bandera a "No" permitir</i>	<b>e1</b>	<b>6(c2)</b>	<b>STO c2</b>	Manda a c2 el último acc del proceso antes de saltar
	<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
<i>Mandar pc a C</i>	<b>e5</b>	<b>1(e0)</b>	<b>LDA e0</b>	Carga en el acumulador el último pc del proceso antes de saltar
	<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 Verificar marca</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
	<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
	<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
	<b>e13</b>	<b>8(s19)</b>	<b>JMP s19</b>	Salta al área de borrado
<i>Salto Salto</i>	<b>e14</b>	<b>1(c16)</b>	<b>LDA c16</b>	Obtiene el numero máximo de procesos
<i>Preámbulo para añadir procesos</i>	<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 almacenado en la memoria secundaria, por lo que, para poder proceder con su carga de manera transparente al usuario, se requiere de un sistema de arranque. La forma de cargar datos de *CARDIAC* está diseñada para tener una especie de sistema de arranque si se siguen ciertas reglas al momento de cargar los datos, como comparten Fingerman y Hagelbarger en [3]. En esta versión del modelo se han adaptado esas

reglas para funcionar con una *memoria secundaria* y un sistema operativo. Para comenzar, las instrucciones que el *iniciador* envíe desde la memoria secundaria a la cola de ejecución deben estar en formato de tarjeta, es decir, en una lista de instrucciones o datos ordenados secuencialmente. Además, la tarjeta debe empezar siempre con las siguientes instrucciones:

0002

8000

Esto para formar un bucle, en el que primero se ejecutará la instrucción *0001*, la cual se encuentra en la dirección #000. Con esto, se cargará *0002* en la dirección #001, que será la siguiente dirección en ser apuntada. Por lo tanto, la siguiente instrucción en ser ejecutada será *0002*, que repetirá el proceso anterior, sólo que ahora dejando en la dirección #002 una instrucción de salto a la #000. Lo que generará un bucle para regresar siempre al inicio, a menos que se ejecute un salto en la dirección #001. Estas dos instrucciones sólo son necesarias en el primer programa en ser cargado a la memoria, ya que a partir de ese momento el sistema de arranque estará instalado.

Con estas instrucciones de inicio conseguimos un sistema de arranque, donde el resto de instrucciones de la tarjeta sólo tengan que seguir un formato de instrucciones pareadas para obtener una carga automática. Es decir, que se necesitan dos instrucciones para añadir un dato a la memoria principal, la primera instrucción define la dirección de destino de la segunda, y la segunda instrucción es la que será un dato o un elemento del programa a añadir a la memoria. Abajo vemos cuáles serían las siguientes instrucciones en la tarjeta:

0003

0000

0004

0000

El primer par de instrucciones es para cargar la bandera que indica si el proceso en ejecución es del usuario o del SOM. Como notamos, la primera instrucción del par establece que la segunda sea cargada en la dirección #003, y que el valor cargado ahí es un 0000. Este valor se coloca por defecto para señalar que los recursos son del SOM. El siguiente par lo que nos indica es el identificador estático o, *id estático*, del proceso que se está ejecutando en ese mismo momento. El valor 0 para este identificador no es sólo un número por defecto, se coloca porque justamente se está ejecutando el **proceso 0**.

Se considera así, porque para el SOM este sistema de arranque será tratado como un proceso una vez que se haya completado la carga del sistema, y la primera dirección que se apuntará después de dicha carga será la #000. Por lo tanto, el primer proceso en ejecutarse es el *0*, y es reconocido por el identificador estático desde el inicio de la carga del sistema. Si bien es considerado un proceso más para el sistema, cuenta con algunas características especiales por su relevancia para el SOM. En el diagrama, los elementos relacionados con el proceso 0 se identifican con un color azul, en la fracción *ejecución del arrancador*.

El resto de los datos que tiene la tarjeta es todo el contenido del sistema operativo mínimo. La tarjeta utilizada para cargar el sistema se encuentra completa en el apéndice E para su consulta, y básicamente consiste en pares de instrucciones para colocar cada segmento del sistema en su lugar.

## Encendiendo la máquina

En la figura 3.13, podemos observar cómo será la máquina virtual para E-CARDIAC C, y nos ayudará como guía para comprender muchos de los aspectos del modelo. Notamos que ahora, en la parte inferior derecha, ya no hay un espacio vacío como en la primera máquina, sino que se muestra el contenido de la memoria secundaria, conformada por direcciones de esta memoria y su 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. Ya que, al ser está una memoria no volátil, la información permanece estática. No cuenta como memoria ROM porque es posible reescribirla, aunque de forma externa. La podemos pensar como una tarjeta que se puede cargar, similar a un disco de instalación.

Como se notará en las imágenes, otra parte que cambió fue la del estatus de la máquina, que ahora tiene un campo llamado *Starter*. 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*) coloca automáticamente la tarjeta de la memoria secundaria en la cola, solo es cuestión de dejar avanzar la ejecución para que la carga se complete. Por lo tanto, se considera que la máquina está “arrancada”.

Además del *Starter*, tenemos otros tres campos nuevos, los cuales nos ayudarán a ver

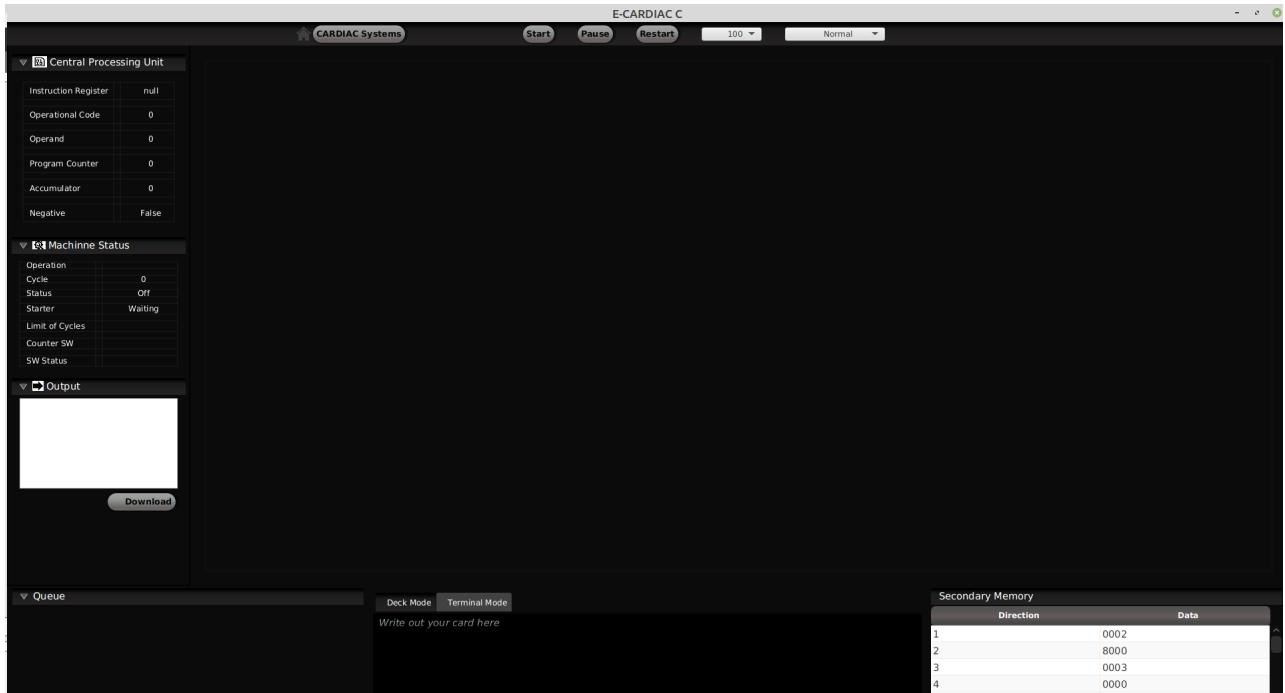


Figura 3.13: E-CARDIAC C Apagada

quién tiene el control de los recursos y por cuántos ciclos. El campo *Limit of Cycles* (límite de ciclos) contiene el valor máximo de ciclos que tendrá un proceso del usuario antes que obligadamente tenga que ceder los recursos de nuevo al SOM. En la implementación que se muestra, tiene un máximo de 30 ciclos para cada proceso del usuario.

El *Counter SW* es el contador del conmutador y tiene un contador de ciclos, pero a diferencia de *Cycle*, este se reinicia cada vez 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 cuántos ciclos lleva el SOM desde que le cedieron los recursos. Puesto que, 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, 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

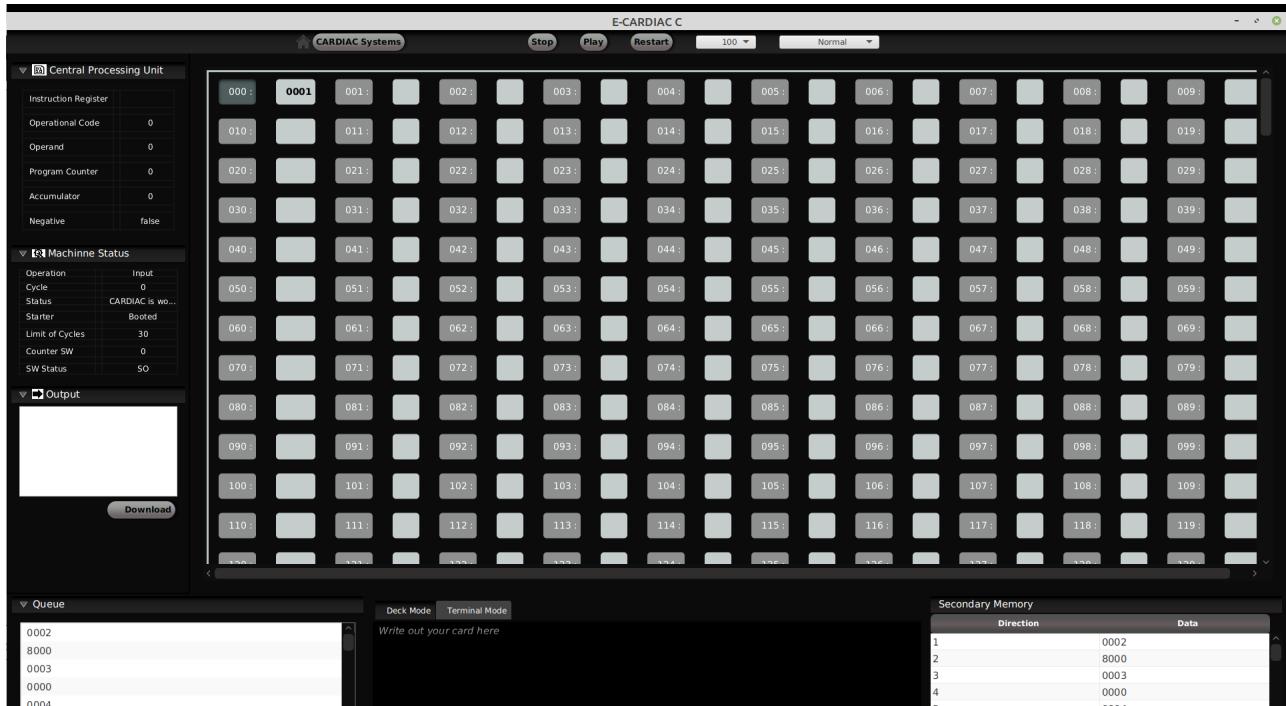


Figura 3.14: E-CARDIAC C durante el arranque

instrucciones se cambia el valor de la bandera, 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, donde podemos notar que el puntero está en la dirección #000. Es decir, está esperando un dato para cargarlo en la dirección #001, la cual ya tiene un dato que fue ingresado en algún momento durante la carga del SOM en memoria, pero que ahora es “basura”.

También podemos observar en la imagen que las banderas están cargadas correctamente, y algo que puede llamarnos la atención es que el estatus del conmutador marca que el control de los recursos lo tiene el sistema operativo. 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 permitirá al usuario cargar programas para que sean ejecutados, y no sería nada óptimo que cada  $N$  ciclos tuviera que detener la carga del 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, tras haber cargado los procesos deseados, cederá

el control al SOM para que administre los recursos y empiece la ejecución de los procesos previamente cargados.

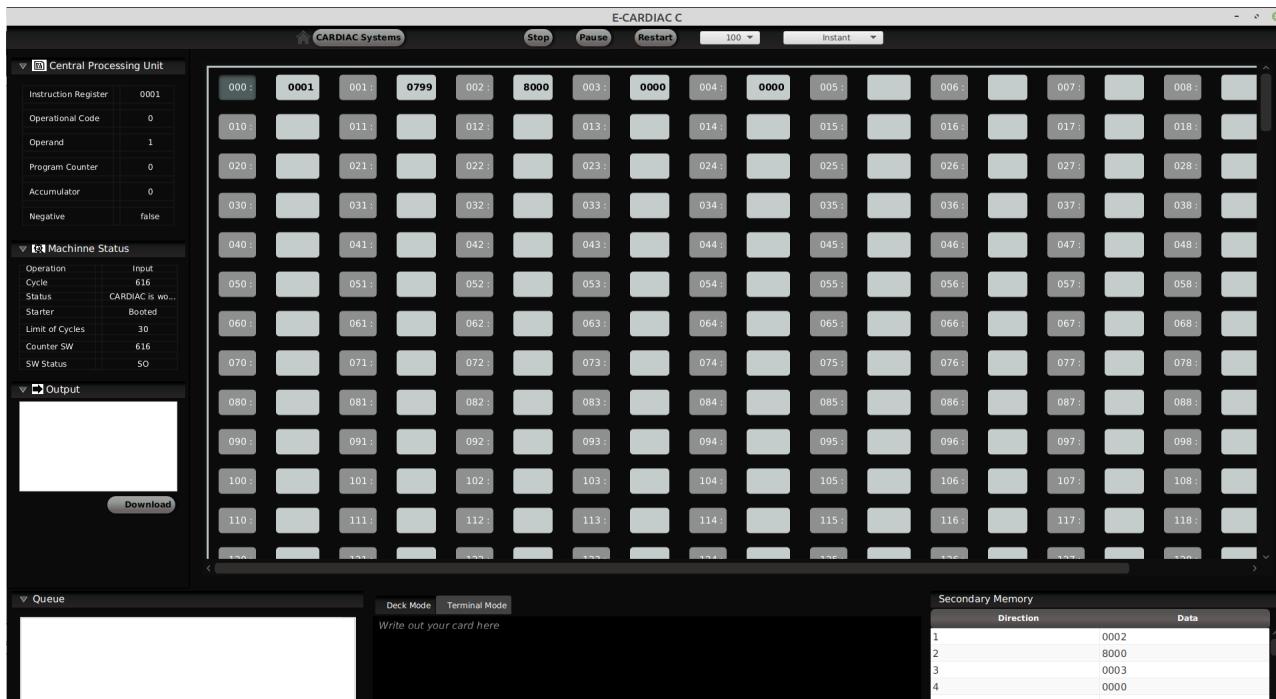


Figura 3.15: E-CARDIAC C Sistema operativo mínimo cargado

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

Como pudimos ver en las imágenes anteriores, a pesar de que para el *conmutador* el control de los recursos actualmente está del lado del SOM, en realidad es el usuario quien los está usando y controlando, aunque de manera limitada, hasta que decida cederlos al iniciar la ejecución de los procesos que ha cargado.

En la figura 3.11, se nos presenta un diagrama de flujo (muy general) en el que podemos alcanzar a ver tres recuadros rosas, con una forma más inclinada que la de un rectángulo normal. Esto es porque son entradas de datos por parte del usuario, utilizadas en el diagrama para exemplificar que son 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 y el usuario ordena el salto; la segunda, cuando el contador de programa salta automáticamente por medio del conmutador; y la tercera, si se da la indicación de borrado de un proceso con la operación *Halt*, es decir, si se lee la instrucción #9000.

En la figura 3.16, apreciamos estas tres formas en que el sistema toma control de los recursos, y podemos observar que todas van a instrucciones de color café. Esto sucede siempre, ya que antes de acceder al núcleo del SOM, es necesario pasar por el preámbulo, que se encarga de las actividades previas para que el sistema operativo mínimo pueda funcionar con normalidad y seguridad.

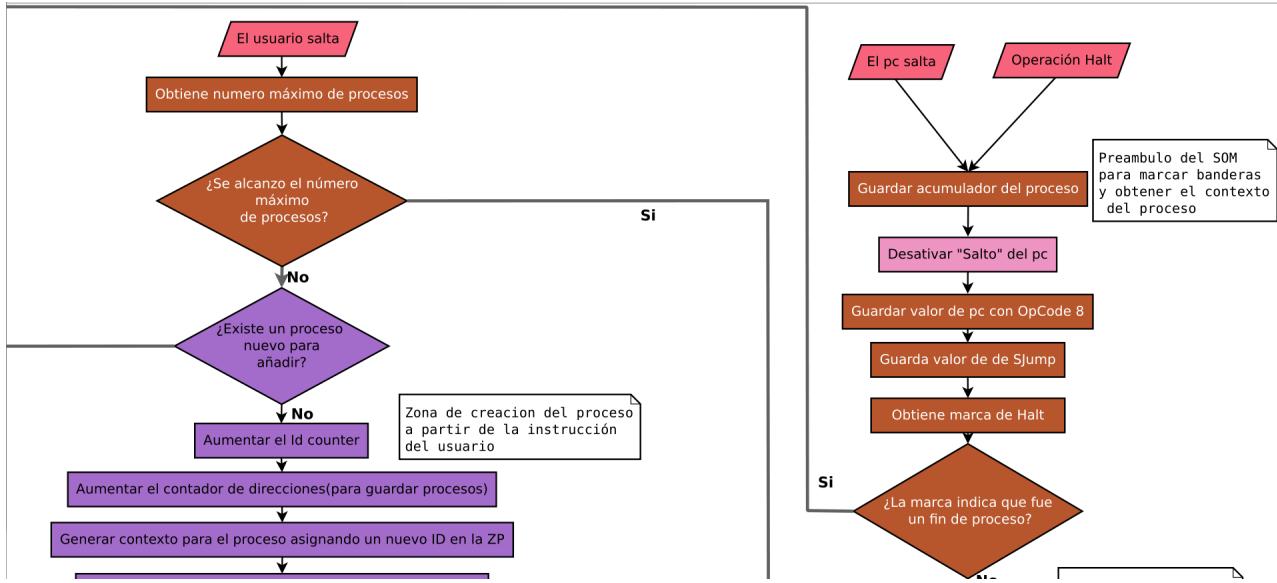


Figura 3.16: Formas de entrar al sistema operativo mínimo C

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

Veremos en orden cómo el sistema operativo mínimo interactuará con los programas que queramos ejecutar en E-CARDIAC C. Para ello, partiremos de un ejemplo práctico. Lo primero que haremos es añadir un par de procesos e iniciar su ejecución. Estos tendrán que actualizar su contexto en la zona de procesos, así como ser eliminados de ahí cuando hayan finalizado. Así, podremos observar todo el ciclo de vida de un proceso en la computadora.

#### Añadir un proceso nuevo: tareas del usuario

El programa que añadiremos es para imprimir números del 1 al 10, el cual se llamará *pintador*, pero lo haremos en tres versiones: una que iniciará después de la dirección #100, otra después de la dirección #300, y otra después de la #500. Observemos la tarjeta de la

versión 1 en la tabla 3.2, que utiliza direcciones posteriores a la #100. En ella observamos un fragmento de la tarjeta que el usuario colocará en la máquina virtual para añadir este programa a la zona de procesos. La primera instrucción de la tarjeta es para indicar que se cargue en la dirección #110 la primera instrucción de *pintador v1*. El resto de la tarjeta sigue el mismo concepto de instrucciones pareadas visto antes.

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

Pero, al final hay una sola instrucción que se sale de esta regla: la instrucción *JMP 985*, que sirve para saltar directamente a la dirección donde el sistema operativo comenzará a añadir el programa a la zona de procesos. En ese momento, el usuario cede por completo el control al SOM para que la información del programa, que ya cargó en memoria, sea añadida a la zona de procesos y este se convierta en un **proceso**. Esta dirección, la #985, es la única del sistema operativo mínimo a la que el usuario tiene permiso para saltar, puesto que el usuario tiene permisos restringidos en todas las direcciones propias del sistema operativo y sus segmentos.

Para ver la tarjeta completa, podemos consultar la tabla 3.3, la cual se lee de arriba hacia abajo y de izquierda a derecha. En esta, se puede observar que todas las instrucciones que se encuentran en la mitad, aquellas omitidas en la tabla 3.2, son efectivamente pares de instrucciones para ir cargando el programa en la memoria principal, a excepción de la última, que es para añadir el proceso a la **zona de procesos**.

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 un proceso nuevo: tareas del Preámbulo

El siguiente paso está del lado del sistema operativo, y lo primero que se necesita saber es a qué parte del sistema se salta con la última instrucción de la tarjeta del usuario. El salto es al preámbulo del sistema, como ya podíamos suponer por lo revisado, y más precisamente a la dirección que corresponde a la variable *e14*. Así, podemos decir que, en nuestra implementación, el valor de *e14* equivale a la dirección #985. Si observamos la imagen 3.12, donde empieza *e14*, el nombre clave asocia a esta y las siguientes direcciones a la fracción **añadir un proceso**, lo que lo hace más claro.

Es en este punto donde también observamos la aparición de las *variables del sistema*. Podemos ver en la imagen 3.17 que hay 19 variables o constantes que el sistema usará a lo largo de su ejecución. En este momento nos interesan particularmente las que se usan en esta parte del preámbulo, la variable *c4* y la variable *c16*; la primera es un **contador de los procesos del usuario**, que por defecto tiene 0, pues se empieza con 0 procesos y el *proceso 0* no se considera del usuario, por lo que no suma en el contador; la segunda contiene el **número máximo de procesos** que se pueden cargar en el sistema operativo. El número máximo para la implementación es 5; para establecerlo, en *c16* se coloca el número máximo que queremos menos uno.

Volviendo al preámbulo, podemos ver por qué se hace ese ajuste. La primera acción del preámbulo es tomar el número máximo de procesos permitidos menos uno, restarle la

cantidad de procesos que el usuario ha agregado y toma una decisión; si el resultado es menor que cero, se reiniciará saltando al sistema de arranque, pues se ha alcanzado el número máximo de procesos; en caso contrario, saltará directamente a la fracción *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 de la resta entre ambos será 0, lo que permitirá añadir el quinto proceso. En cambio, si el usuario ya tuviera 5 procesos, el resultado sería negativo y el condicional indicaría reiniciar en el proceso 0. Por lo tanto, si queremos como máximo 10 procesos ejecutándose de manera concurrente, hay que considerar este funcionamiento y colocar en *c16* un 9.

Variables del sistema				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
	<i>c0</i>			Espacio para el SOM
<i>8-pc</i>	<i>c1</i>			Espacio para el SOM/Preámbulo, recibe un valor en cada iteración
<i>acc</i>	<i>c2</i>			Espacio para el SOM/Preámbulo, recibe un valor en cada iteración
<i>Inicial</i>	<i>c3</i>	<i>p5</i>		Es la dir de inicio de la zona de procesos(usuario)
<i>Id counter</i>	<i>c4</i>	0		El contador de procesos de usuario
<i>Dir counter</i>	<i>c5</i>	<i>p0</i>		El contador de direcciones
<i>Dir organizer</i>	<i>c6</i>	<i>p0</i>		Contiene a la dirección del proceso que se ejecutará
<i>Id organizer</i>	<i>c7</i>	0		Contiene el id del proceso que se ejecutará
	<i>c8</i>	1000		Valor para convertir op-code en LOAD
	<i>c9</i>	6000		Valor para convertir op-code en STORE
	<i>c10</i>	-1		Valor de uso recurrente
	<i>c11</i>	5		Valor de uso recurrente, para el salto de los procesos
	<i>c12</i>	2		Valor de uso recurrente
	<i>c13</i>	8000		Valor para convertir op-code en JUMP
<i>Saver Jump</i>	<i>c14</i>			Guardará el valor de 999 por parte de la arquitectura
<i>Serial Id Proces</i>	<i>c15</i>	0000		Contendrá un valor serial para los id de los procesos
<i>Máximo numero</i>	<i>c16</i>	0004		Máximo numero de procesos disponibles (Menos 1 para la resta)
	<i>c17</i>	998		Espacio para el SOM/Área de borrado
	<i>c18</i>	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, se saltará al segmento del núcleo del SOM para añadir un nuevo proceso, puesto que hay 0 procesos del usuario y se permite añadir otro. El salto se realiza a la dirección *s66*, la cual funciona para recordar que la letra “s” en las variables representa direcciones del núcleo del sistema operativo.

Pero observemos primero el diagrama, con un acercamiento en la fracción de añadir un nuevo proceso, de la 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, pues hay que validar que no se haya saltado por accidente.

La forma en que el usuario deja una marca para indicar que sí hay un proceso por añadir, es colocando un valor en la dirección `#s0`. La tarjeta del programa *pintor v1* realiza esto en las dos últimas instrucciones previas al salto hacia el preámbulo. Este par contiene la instrucción *0800*, que significa cargar un dato en la primera posición del núcleo del sistema, y la instrucción *8110*, que es la primera dirección del programa *pintor v1*. Por lo tanto, este par de instrucciones sirve para colocar en `#s0` la dirección de inicio del proceso a ser cargado. Cabe aclarar que `#s0` es **la única dirección del SOM en la que el usuario puede directamente escribir** sin restricciones.

En la imagen 3.19 podemos ver que *s0*, que en nuestra implementación es *800*, tiene un valor por defecto de *-0001*. Esto se debe a que dicho valor indica que no hay un proceso por cargar, y cada vez que se termina de cargar un nuevo proceso, el contenido de esta dirección regresa a su valor por defecto para preservar la integridad del sistema.

En caso de que el valor de *s0* fuese negativo, el flujo posterior al salto desde el preámbulo sería más corto, como se muestra en la figura 3.18. El flujo comenzaría verificando el valor del **Id organizer**, u “organizador de identificadores”, que contiene el identificador del proceso que se está ejecutando. Si el valor es 0, es porque la orden de añadir un nuevo proceso fue lanzada desde el cargador<sup>1</sup>, por lo que regresará al proceso 0 mediante el recorrido más corto.

Sin embargo, la última operación de la fracción *añadir un proceso* (morada), continúa su flujo también en la condicional del organizador de identificadores. Así, en ambos resultados posibles del condicional “¿Hay un nuevo proceso para añadir?”, se termina revisando si la orden fue lanzada desde el proceso 0. Esta verificación es relevante, ya que, si la orden de añadir un nuevo proceso no fue lanzada desde el proceso 0, como es lo normal, la fracción de gestión del proceso 0, toma mayor relevancia.

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 nuevo proceso. Si fue un proceso el que cedió los recursos, el sistema operativo tiene que elegir al siguiente proceso a ejecutar, y no sólo saltar

---

<sup>1</sup>Se utilizarán de forma indistinta “proceso 0”, “cargador”, “arrancador”, o “sistema de arranque”, aunque estos dos últimos, se utilizarán principalmente cuando se trate de iniciar operaciones.

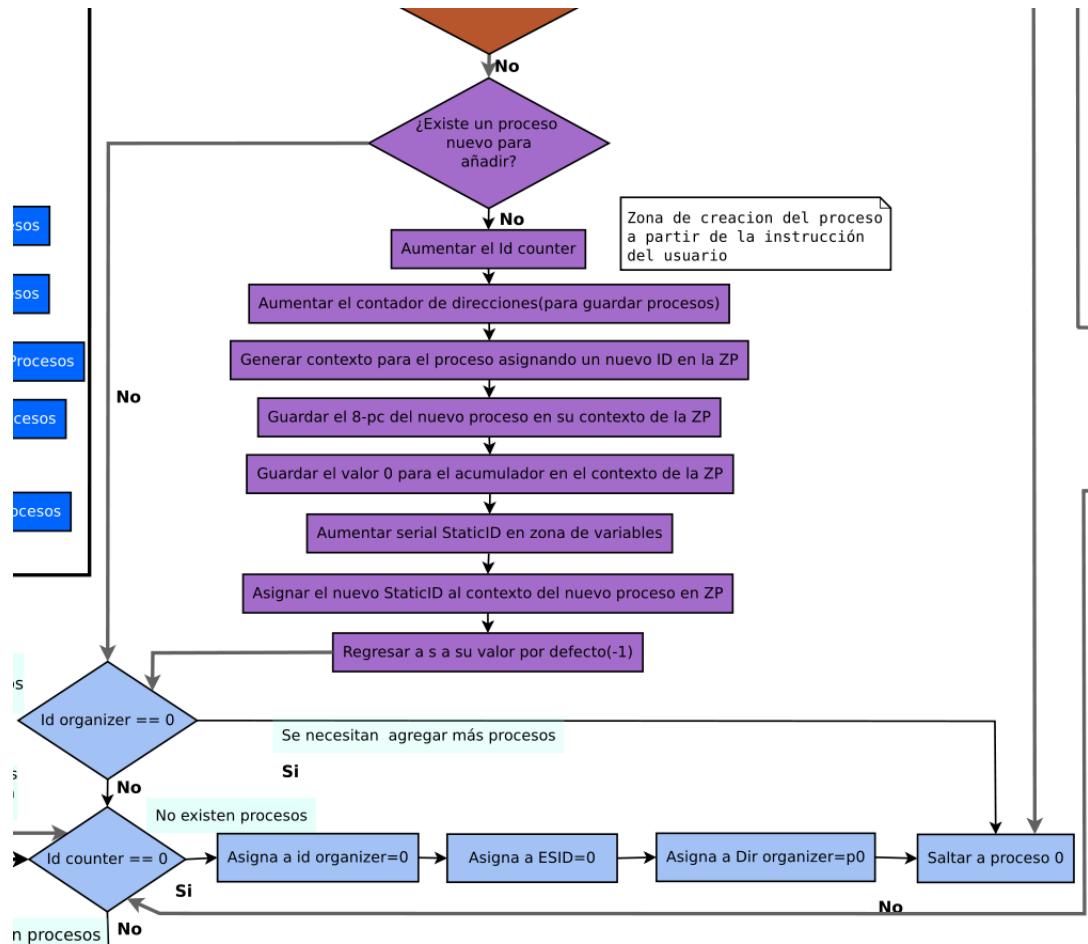


Figura 3.18: Diagrama de segmento para añadir un proceso

al proceso 0. Si el organizador tiene un valor diferente de 0, se pasa a la siguiente condicional de la fracción azul claro, donde se verifica si hay más procesos por lanzar con el *id counter*, o “contador de identificadores”. Si no hay procesos por lanzar su valor será cero, por lo que reiniciará algunas variables y saltará al proceso 0. Pero, si el resultado es diferente de cero, el flujo se enlazará con otra fracción del núcleo del sistema, que es el lanzamiento de procesos. Esta fracción la veremos más adelante, y su función es “elegir” qué proceso será el siguiente a ejecutar.

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

Figura 3.19: SOMC: Contenidos generales del núcleo

## Fracción Añadir proceso: Contexto del proceso

Regresando a la fracción morada del diagrama, después de verificar que hay un nuevo proceso por 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 se actualiza el valor de  $c_4$ , el contador de procesos aumenta en uno. Por otra parte, tenemos a  $c_5$ , el “contador de direcciones”; este contador contiene la dirección del identificador del último proceso añadido, por lo tanto, contiene direcciones de la zona de procesos. La variable  $c_5$  tiene como valor por defecto a  $p_0$ , dado que toda variable que inicia 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
	s66	1(s)	LDA s	Se carga en el acumulador la 8-dir de inicio del proceso
	s67	3(s95)	BLZ s94	si acc<0 no hay nuevo proceso
Aumenta el Id counter M[c4]++	s68	1(c4)	LDA c4	Se obtiene el valor del Id counter
	s69	2000	ADD 000	
	s70	6(c4)	STO c4	
Aumenta el Dir counter M[c5]+=5	s71	1(c5)	LDA c5	Se obtiene el valor del Dir counter
	s72	2(c11)	ADD c11	Se le suma 5 para tener la siguiente dirección
	s73	6(c5)	STO c5	
Previa de ID	s74	2(c9)	ADD c9	Convierte la nueva dirección en dir counter en 6-dir
	s75	6(s87)	STO s86	En s70 se guarda 6(px)
Previa de pc	s76	2000	ADD 000	Se obtiene la 6-dir del gpc del nuevo proceso
	s77	6(s89)	STO s78	En s72 se guarda 6(px)+1, la 6-dir del gpc del nuevo proceso
Previa de acc	s78	2000	ADD 000	Se obtiene la 6-dir del gacc del nuevo proceso
	s79	6(s92)	STO s81	En s75 se guarda 6(px)+2

Figura 3.20: SOMC: Añadir un proceso, parte 1

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

Figura 3.21: SOMC: Añadir un proceso, parte 2

En la figura 3.22 podemos ver cómo está compuesta la zona de procesos. Llamaremos “contexto del proceso” a todas las variables asociadas al proceso, las cuales se almacenarán en la zona de procesos bajo un mismo identificador. E-CARDIAC C requiere de cinco variables para el contexto de sus procesos, siempre asociadas a un identificador. En la zona de procesos podemos observar el valor por defecto para  $p5$  y  $p10$ , el cual es  $-0001$  en ambos, ya que son los inicios de cada contexto, y ese valor por defecto indica que no hay un proceso en ese lugar.

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  $p5$ . En nuestro caso es justo lo que hace: cambia a  $p5$  el valor del contador de direcciones, porque ahora el último proceso es el que se encontrará en el contexto de  $p5$ .

Revisemos el resto de variables que están en el contexto del proceso. Primero tenemos a las dos más evidentes: el contador de programa, guardado en  $gpc$ , y el acumulador, guardado en  $gacc$ . Estas variables contienen el último valor del contador de programa y del acumulador, respectivamente, antes de que el proceso cediera los recursos. Al crear el proceso, a  $gpc$  se le asigna la dirección de inicio del proceso, y a  $gacc$  un cero.

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, durante la ejecución del proceso al que hace referencia. En caso de que se esté creando el proceso se coloca el valor por defecto que tiene el modelo.

Esta variable es necesaria debido a que el valor de la última dirección de la computadora cambia de acuerdo a las instrucciones de cada proceso y es vital guardarla en el contexto de cada proceso para evitar 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 e intente usar el valor que hay en #999 para ir a la dirección #819, termine en la #514.

Por último, para identificar a los procesos tenemos su ID principal, con el que inicia el contexto, y que tiene por nombre completo *ID counter*, “ID contador”, o simplemente “ID

principal". Este va en orden ascendente y también sirve para contar cuántos 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. Por lo pronto, y desde este punto, siempre haremos la distinción entre cada identificador, pues sus funciones son distintas.

Zona de Procesos				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>Id Counter</i>	<i>p0</i>	0000		Id del primer programa
<i>gpc</i>	<i>p1</i>	8000		Es el pc del proceso 0 con op-code 8
<i>gacc</i>	<i>p2</i>	0000		Es el acumulador del proceso
<i>gjump</i>	<i>p3</i>	8000		Contiene el valor del que se tenía en la última dirección de memoria antes de saltar
<i>Static ID</i>	<i>p4</i>	0000		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 esta 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: Zona de procesos del sistema operativo

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

Las figuras 3.18, 3.20, y 3.21 seguirán siendo nuestra guía para comprender cómo se va agregando la información faltante en el proceso. Primero, se actualizan las variables del sistema, para que este conozca cuántos procesos hay, dónde están y cuál es el último contexto de la zona de procesos activo, es decir, con un *ID contador* no negativo. Después, desde la dirección #s74 hasta la #s92, lo que hace es llenar el contexto del proceso. Se definió a #p5 como la última dirección de inicio de un contexto de proceso activo, por lo que el primer paso es asignar el número 1 como contenido de #p5, ya que el *ID contador* es un contador de identificadores con valores ascendentes. Si agrego otro proceso, el contenido de #p10 será 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, con sólo ese valor, conocer la cantidad de procesos que hay y saber el lugar que ocupa cada proceso en la lista de procesos de la ZP, o zona de procesos.

Para hacer la carga de los valores del contexto del proceso en su correspondiente lugar, es el mismo sistema operativo quien coloca 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. Es decir, como programador, no puedo colocar de manera estática la dirección `#p0` o `#p10` para agregar un proceso; tienen que ser direcciones dinámicas que coloque el SOM.

Analicemos 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. Ese resultado, con la instrucción de `#s75` lo guardamos en la dirección `#s87`. En la imagen 3.21, la dirección `#s87` tiene un contenido genérico en color rojo con el código de operación 6, acompañado de un `px`, indicando que se trata de una dirección tipo `p`, que en nuestro caso sería la `#p5`, pues es una dirección dinámica.

Dejamos un poco esa parte de las instrucciones y nos movemos hasta la `#s86`, donde se carga en el acumulador el valor de `c4`, que contiene el número de procesos que se han agregado hasta el momento. Cuando se apunte a `#s87`, que tiene como contenido `6(p5)`, se guardará el número 1 en `#p5`, pues es el número de procesos que se han agregado hasta ese momento de acuerdo con la variable `c4`; ya que como recordaremos, las variables del sistema se actualizaron primero.

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 se cargan estos valores en orden, 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.

Luego, se requiere colocar en `p6`, que representa a la variable *gpc*, la dirección de inicio del proceso antecedida por un código de operación 8, lo que ayuda en la simplificación del lanzamiento del proceso. En *gacc*, que se encuentra en `p7`, el valor por defecto para el acumulador es 0. La variable que no se actualiza es *gjump*, en `p8`, porque al inicio del proceso

es razonable suponer que la última dirección de la máquina podría tener cualquier valor posible.

Lo que sigue en el flujo es modificar el identificador estático, que se debe cambiar tanto en la zona de variables del sistema como en el contexto del proceso que estamos agregando. Ya que, mientras el *ID contador* no guarda ninguna relación con el proceso una vez que este termina, debido a que 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 sólo va aumentando a medida que agregamos procesos; este valor se encuentra en *c15*. 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 luego guardar ese valor en *#p9*, que es la dirección donde se guarda el *ID estático* del nuevo proceso que se está añadiendo.

Para finalizar, se reinicia el valor de *#s0* a un negativo, indicando que ya no hay un nuevo proceso por añadir. El flujo continúa hacia 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 es necesario tomar el flujo más largo. Como notarán, en ningún momento se realiza cambio de bandera, ya que desde que el usuario agrega el programa hasta que el SOM lo añade a la zona de procesos, para el *comutador* el dueño de los recursos sigue siendo el SOM.

### Práctica: añadir un proceso nuevo

Analicemos en la máquina virtual cómo se añade un nuevo proceso. Estamos en la situación de la máquina iniciada, con el sistema operativo cargado, y el contador de programa apuntando a la dirección *#000*, esperando para cargar un nuevo proceso. Pero, si observamos los valores con los que ha sido iniciada la computadora, en la figura 3.15, veremos algo interesante. En las últimas dos casillas superiores, que son listas desplegables, tenemos un *100*, que hace referencia al número de celdas, y *Instant*, que hace referencia a la velocidad a la que cada ciclo se completa.

A diferencia del primer modelo, en este la computadora no puede tener *100* celdas de memoria, ya que cuando se inicia la máquina virtual, esta realiza una validación sobre las celdas indicadas en la lista desplegable, y verifica si son suficientes para el tipo de procesos

que llevará a cabo. *E-CARDIAC C* sólo puede funcionar con mil celdas o más; por esa razón, si la lista desplegable tiene un valor menor, la máquina se iniciará con el mínimo permitido, dejando en la lista desplegable el valor que ignoró.

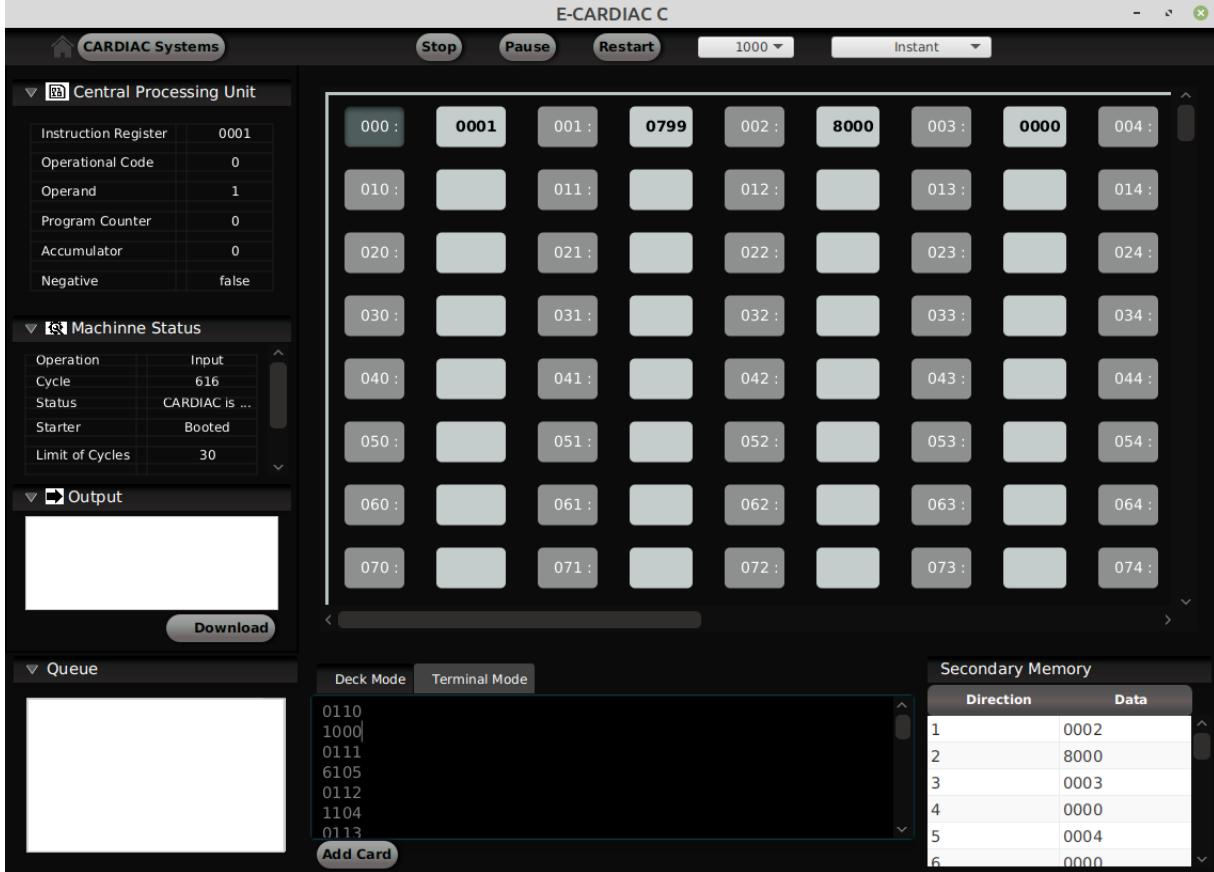


Figura 3.23: E-CARDIAC C Programa en *deck mode*

En la figura 3.23 observamos la máquina cargada nuevamente, y ahora si tiene el valor 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 cómo se va ejecutando cada instrucción. Los cambios de velocidad se pueden aplicar en cualquier momento del proceso dependiendo de las necesidades del usuario. Sin embargo, el cambio en la cantidad de celdas solo puede efectuarse al iniciar la máquina.

En esa misma imagen, la 3.23, podemos ver en el modo de tarjetas una lista de instrucciones en texto escrita por el usuario, previo a oprimir el botón *Add Card*. Una vez que se oprime el botón, esa tarjeta es trasladada a la cola, y lo podemos ver en la imagen 3.24, don-

de aparece una lista de instrucciones que están siendo transmitidas a la memoria principal desde la cola.

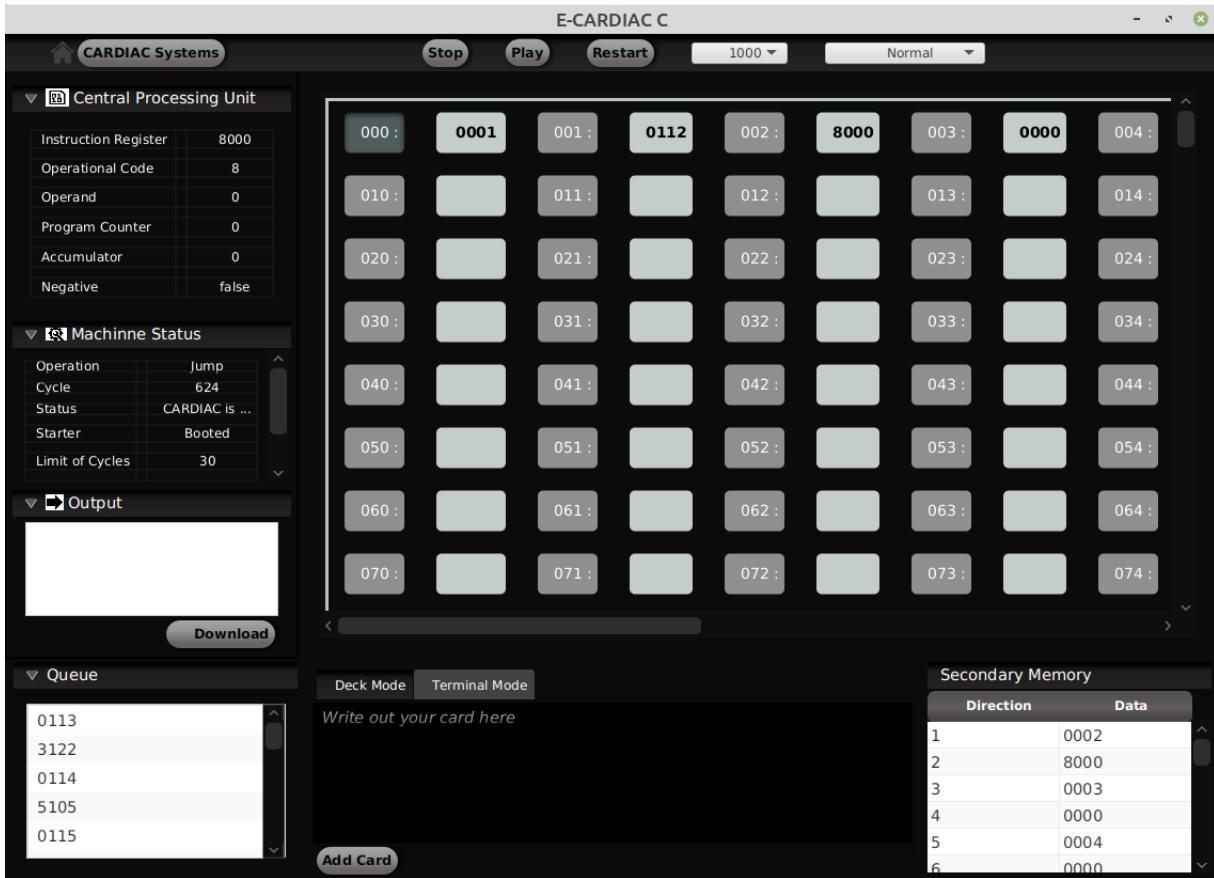


Figura 3.24: E-CARDIAC C Programa en cola

Luego, en la imagen 3.25, observamos que se vació la cola y la instrucción que se va a ejecutar es para saltar al preámbulo y añadir un nuevo proceso a la lista. En este punto, el proceso de la carga del programa se ha completado, y lo podemos ver en la figura 3.26, donde se observa el programa cargado en memoria. Cuando el sistema termina de agregar los datos de este programa a la zona de procesos es cuando, en sí, se convierte en un proceso, ya no son sólo instrucciones de código. El contexto del proceso ha quedado definido, y con ello todas las variables que necesita gestionar el SOM para la ejecución segura del proceso.

En la figura 3.27 observamos el contexto del *proceso 0*, que inicia en la dirección #769, y también el del *proceso 1*, que inicia en la dirección #774. El contexto del *proceso 1* cuenta con: un *ID contador* con un valor de 1, un *gpc* que indica la dirección de inicio del proceso, un *gacc* con ceros en su valor inicial, el *gjump* con el valor por defecto que se tiene para

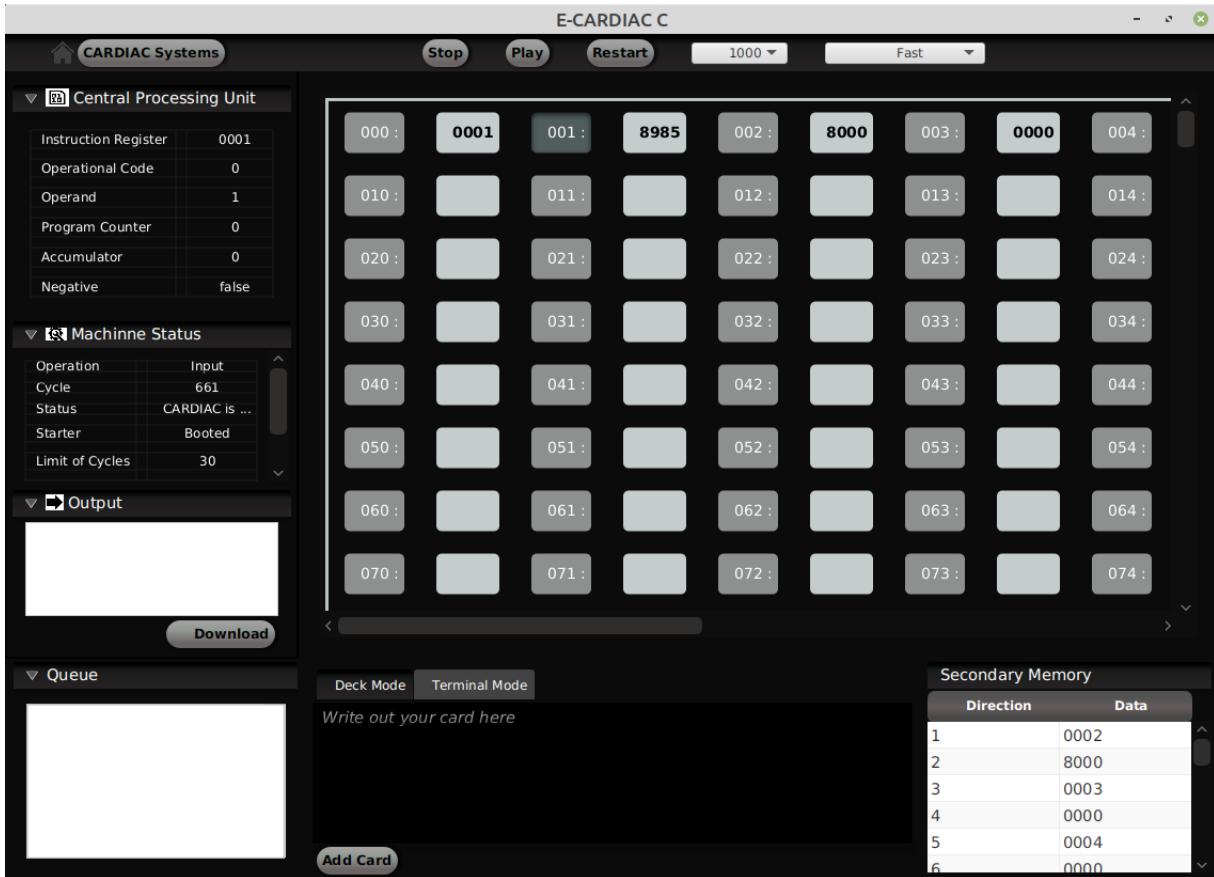


Figura 3.25: E-CARDIAC C Instrucción para añadir proceso

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

la dirección #999, y un 1 para el ID estático, puesto que es el primer proceso que se está agregando. Notamos también que la dirección #779, el siguiente inicio del contexto de un proceso, tiene un valor de -1, ya que no hay otro proceso más hasta este momento.

Para finalizar, agregaremos los otros dos *pintores*, que iniciarán en #310 y #510, respectivamente. Como una vez que se termina de agregar un proceso se regresa al proceso 0, podemos repetir el procedimiento sin ningún problema hasta alcanzar el número máximo de procesos. En la imagen 3.28 vemos la zona de procesos después de agregar estos dos, donde ahora el identificador de #799 es un 2, y el de #784 es un 3. Los tres procesos que agregamos

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

tienen casi los mismos valores de arranque en su contexto, salvo las direcciones de inicio y sus identificadores.

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 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 para detener un proceso, en este caso, el proceso 0. Lo que hace después es saltar al preámbulo, por lo que hay que revisar las figura 3.12 y 3.29 para ver en más detalle las instrucciones que ejecuta para “finalizar” el *proceso 0* y saltar al núcleo del SOM.

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

Después de ejecutar una instrucción *Halt*, el salto es directo a la dirección #e1, que sigue exactamente el mismo flujo que se sigue cuando el contador de programa salta de forma automática. Lo primero que se realiza es guardar en una variable del sistema el último valor que el acumulador del proceso tuvo antes de ceder los recursos. Después, cambia la bandera para que ya no permita saltos, y a continuación, guarda en otra variable del sistema el último valor que tuvo el contador de programa cuando el proceso estaba en ejecución.

Este valor se puede guardar gracias a que el conmutador en automático, cuando da la indicación de saltar al preámbulo, coloca en #e0 el último valor que el *pc* tuvo durante el proceso. Mientras que si fue la instrucción *halt* la que indicó el salto, el conmutador coloca un valor negativo en #e0.

La siguiente acción es guardar el valor del *gjump*. Para ello, se toma directamente el valor del contenido de #999, porque los saltos hacia #e1 por parte del conmutador y de la instrucción *halt* no dejan ninguna marca en #999. Entonces, cuando el apuntador llega a #e8, simplemente toma el valor de #999 y lo guarda en la zona de variables del sistema, más precisamente en #c14, para resguardarlo.

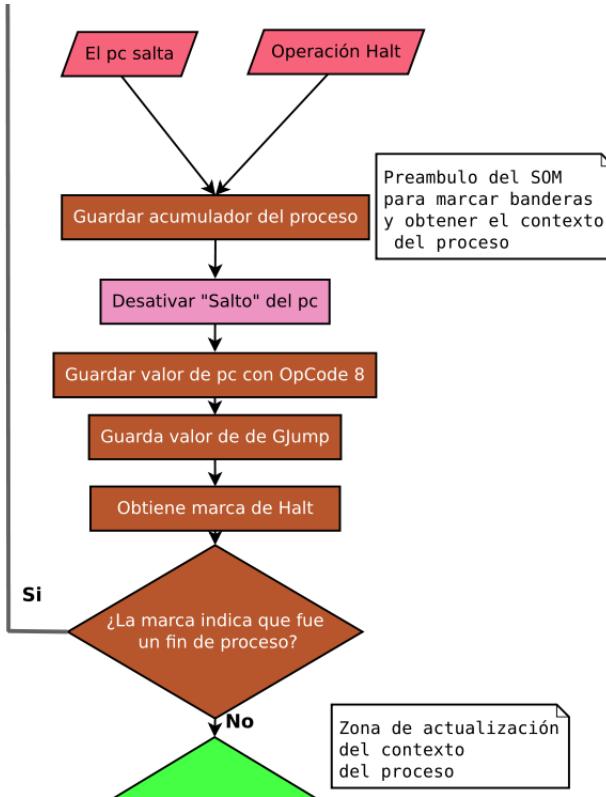


Figura 3.29: Acercamiento al preámbulo en el diagrama

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 este caso, #e0 es negativo, por lo que saltará a la fracción de borrado. En la figura 3.30 podemos ver que en la dirección #950, que es la dirección #e0 en la implementación, hay un valor negativo. Por lo tanto, cuando llega al condicional en #961 salta la dirección #963, la cual contiene la instrucción para saltar a la fracción de borrado. Si observamos el diagrama en las figuras 3.29 y 3.11, notaremos que es en esta parte donde la flecha sale de la fracción del preámbulo hasta el extremo izquierdo, donde está la fracción de borrado.

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

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

## Fracción *borrar proceso*: Características especiales para el proceso 0

Como vemos en el diagrama de la figura 3.31, una vez que se llega a la fracción de borrado, la primera acción es el respaldo de una variable del sistema, pero inmediatamente después está la condicional que pregunta si la instrucción de borrado fue ejecutada desde el *proceso 0*. Si no lo fue, continúa con la ejecución normal para borrar el proceso. En caso contrario, salta a la fracción de *gestión del proceso 0*, donde el proceso es bloqueado para que se puedan ejecutar otros. Por lo tanto, el proceso no es borrado; sino que al ser bloqueado, permite la conexión con la fracción del lanzamiento de procesos. Es por eso que, para lanzar los procesos que se han cargado, el usuario tiene que 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 un buen momento para entender qué función tiene esta variable. Si observamos la figura 3.19, podemos ver que en el comentario ya se nos da una pequeña explicación de lo que es: el valor que guarda esa dirección siempre será la dirección del *ID contador* del último proceso de usuario que se ha ejecutado, con un código de operación *6*. Para simplificar, usaremos la nomenclatura *6-dir*, utilizando como prefijo el código de operación que tiene la dirección a la que hacemos referencia, y como sufijo la palabra *dir*, haciendo referencia a dirección.

Observemos ahora la imagen 3.33, que muestra el inicio de la zona de borrado. Lo primero que se realiza es obtener el valor de `#s1`, para que sea convertido en una *1-dir*, es decir, una dirección con un código de operación *1*. Si no se trata del *proceso 0*, este cambio será útil. El siguiente paso es obtener el *ID contador* del proceso que se estaba 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 se ejecute esa instrucción, se obtenga el contenido de la dirección de la zona de procesos a la que hace referencia, que será el *ID contador* del proceso que se requiere borrar.

En este caso, como sí es el proceso 0, salta a la fracción azul claro en la dirección `#s98` para bloquear el proceso, y no para borrarlo. Al bloquearlo, se impide su ejecución infinita y se permite que otros procesos se ejecuten hasta que sea necesario que este proceso vuelva a ejecutarse.

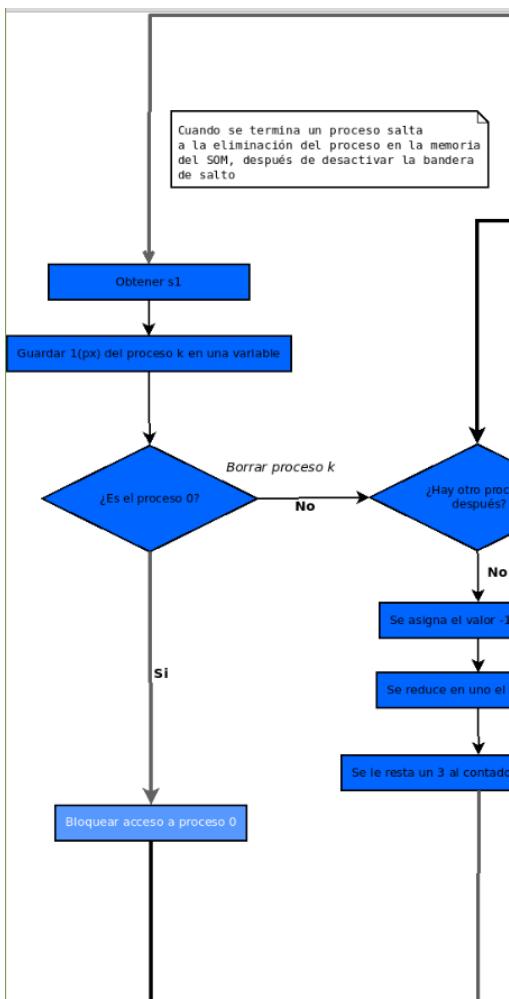


Figura 3.31: Acercamiento a la parte del bloqueo al proceso 0

Si observamos la imagen 3.32, podremos notar que en #s98, dirección a la que salta el proceso de borrado, primero se verifica si hay más procesos. Esto se logra consultando 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 este tiene un valor de cero, quiere decir que no hay más procesos, y lo que haría es regresar al proceso 0.

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>¿está en el proceso 0?</i>	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
<i>¿Se acabaron los programas?</i>	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
<i>Reiniciar id organizer a 0</i>	s99	7(000)	SUB 000	Se le resta un 1 , acc=-1
	s100	3(s141)	BLZ s140	Si acc<0 salta al proceso 0
<i>Asigna id organizer=0</i>	s141	1(000)	LDA 000	
	s142	7000	SUB 000	Carga el id=0 del proceso 0
<i>Asigna a 004 el 0</i>	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
<i>Asigna a dir organizer=p0</i>	s145	1(c3)	LDA c3	Obtiene p4
	s146	7(c11)	SUB c11	Obtiene p1 al restarle 5
	s147	6(c6)	STO c6	Asignar a dir organizer=p0
	s148	8(000)	JMP 000	Saltar a proceso 0

Figura 3.32: SOMC segmento del proceso 0

En la misma imagen (3.32), está la condicional de salto a #s141, el salto que se realiza cuanto no hay más procesos. En esa tabla, hay una continuación directa de la dirección #s100 a la #s141, puesto que entre esas dos direcciones 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 *organizador de ID* y del *organizador de direcciones*. También coloca en la dirección #004 el valor de 0, ya que ahí se encuentra el valor del **identificador estático** del proceso que se está ejecutando, en este caso será 0; esta dirección, asimismo, está reservada para el SOM. Posteriormente, salta de regreso al proceso 0. En caso contrario, si hay más procesos para ejecutar y el *ID contador general* de la zona de variables del sistema tiene un valor mayor a 0, continúa la ejecución en la fracción de lanzamiento de los procesos.

### Fracción *lanzamiento de procesos*: exploración del diagrama

En este caso, la ejecución continuará en la fracción naranja del diagrama, en las direcciones que están entre #s100 y #s141, debido a que sí hay más procesos para ejecutar. Como vemos en la imagen 3.34, continuamos en la fracción naranja después de verificar que el *ID contador* no es cero. Aquí, las variables del sistema que serán muy importantes son las que están entre #c3 y #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, son las direcciones para encontrar el contexto del proceso.

Sistema operativo				
<i>Guardar 1(px)</i>	s19	1(s1)	LDA s1	Llamemos k al proceso asociado a la dir px
	s20	4011	SHT 11	Se convierte 6(px) en 0(px)
	s21	2(c8)	ADD c8	Se convierte 0(px) en 1(px)
	s22	6(c0)	STO c0	Guarda en c0 1(px)
<i>Verificar si es el proceso 0</i>	s23	1c7	LDA c7	Se obtiene el id del proceso que se estaba ejecutando
	s24	7(000)	SUB 000	Se le resta 1 al id, acc=-1
	s25	3(s98)	BLZ s	Si acc<0 es el proceso 0 y se bloquea, no se borra
	s26	1(c0)	LDA c0	Se obtiene 1(px)
<i>Si no lo es</i>	s27	2(c11)	ADD c11	Se obtiene 1(px)+5, la 1-dir del id del proceso siguiente (k+1)
	s28	6(s29)	STO s29	En s24 se guarda 1(px)+5, al que llamamos 1(py)
	s29	1(py)	LDA py	Se obtiene el id del proceso (k+1)
	s30	3(s52)	BLZ s52	Si acc<0 no hay otro proceso adelante
<i>¿Hay otro?</i>	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
<i>Iniciar 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

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

Si consultamos la imagen 3.17, observamos que en #c3 se guarda siempre la dirección de inicio de la zona de procesos propios del usuario. Por esa razón contiene un #p5 y no un #p0, porque el *proceso 0* es un proceso del SOM. La dirección de inicio de la zona de procesos se decide en la implementación del sistema operativo; en ese momento también se llenarán los espacios que requieran esa información.

Después, en #c4, como ya habíamos visto, tenemos el *ID contador general*, o *id counter* en la imagen, 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. La variable #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 manera, podemos conocer el último proceso. Sin embargo, para conocer la información del proceso a ejecutar, necesitamos de los **organizadores**, de #c6 y #c7. Estos, en lugar de ser “contadores” de identificadores y direcciones, contienen la dirección del *ID contador* del proceso que se va a ejecutar y el valor de ese identificador, respectivamente.

Con todo esto, podemos lograr una concurrencia de procesos. Primero, se colocan los valores correspondientes en los **organizadores** #c6 y #c7, con ello, se puede tener acceso al contexto completo del proceso y ejecutarlo. Cuando el proceso alcance el límite de ciclos permitidos, se regresa a la zona de lanzamiento, y se aumenta el valor de los “organizadores” para ejecutar el siguiente proceso en cola.

En este sistema, se sigue un paradigma *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 reinicia el lanzamiento en el proceso 1, hasta que terminen su ejecución todos los procesos. 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.

### Fracción *lanzamiento de procesos*: análisis del código

En la imagen 3.35, podemos seguir los pasos descritos anteriormente. Desde el aumento de los organizadores para acceder al siguiente proceso en cola, hasta la condicional (de #s109) en la que puede saltar a #s134, en caso de que sea necesario reiniciar los organizadores para lanzar de nuevo el *proceso 1*. Para ver cómo se realiza este reinicio, podemos observar cómo 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 para lanzarlo. En caso de que ya no hubiera procesos, ni siquiera se llegaría a la fracción naranja.

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

En la parte siguiente del rombo naranja en el diagrama de la figura 3.34 se puede ver el flujo de ejecución que continúa después de esa verificación; en código, está en la dirección #s110. En esta, se guarda en #s1 la *6-dir* de donde se encuentra el identificador del proceso a ejecutar, 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, en 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 para asignarlos a sus respectivos lugares en la ejecución del proceso.

Para lograr esto, se usa como pivote la dirección que se encuentra almacenada en #c6, en el organizador de identificadores. Ya que, con solo ir sumando uno a esa dirección, se puede

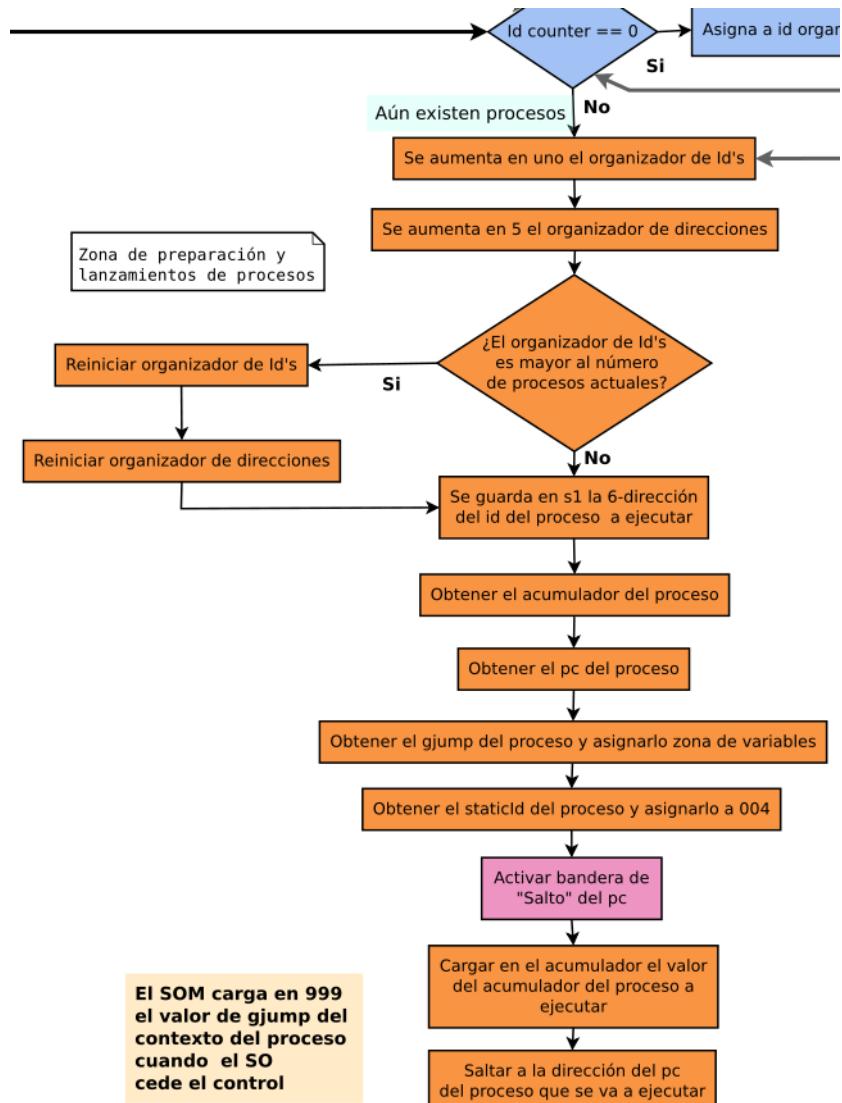


Figura 3.34: SOMC Lanzamiento de procesos

obtener la dirección de cada uno de los elementos del contexto del proceso. A estas direcciones se les asigna 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.

Analicemos 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  $1(p5)/1774$ , es decir, la  $1\text{-dir}$  del identificador del proceso. Ese resultado,  $1(p5+1)/1775$ , se guarda en #s119. Cuando el apuntador llega a esta dirección, la instrucción que lee es para obtener el contenido de la dirección #775, que es el contador de programa del proceso a ejecutar con un código de operación de 8. Este es guardado en #s133, la última dirección propia del sistema operativo

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
	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"
<i>Actualizar s1 para salir al proceso con su dc v acc</i>	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

mínimo. Se guarda en esta dirección porque, con ese código (8110 en este proceso), se realiza el salto desde la zona del sistema operativo al proceso del usuario.

Si regresamos a la dirección #s117, vemos cómo se suma un uno al contenido del acumulador, que en ese momento tenía un  $1(px+1)/1775$ . Con esto, se obtiene  $1(px+2)/1776$ , es decir, la 1-dir del acumulador del proceso, y ese resultado se guarda en #s132, la penúltima dirección del SOM. Lo que significa que, justo antes de saltar, se 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. Así, el proceso toma el control de los recursos con la instrucción de salto que está en la última dirección del SOM, y el acumulador de la máquina en ese momento ya es igual al acumulador del contexto del proceso.

Por otra parte, en #s121, lo que sucede es que se carga en el acumulador el valor  $1(p5+2)/1776$ , para sumarle otro uno y obtener  $1(p5+3)/1777$ . Este es guardado en #s128 para poder obtener el *gJump* y guardarlo en la variable del sistema #c14. Esto es de suma importancia, porque es la máquina quien, a través del conmutador, coloca en #999 el valor de *gJump* utilizando la variable #c14. 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 la dirección desde donde salta. Pero lo que queremos que guarde es el valor que contiene la variable *gJump* del contexto del proceso a lanzar.

Lo único pendiente es el *ID estático*. Su dirección se obtiene en #s124 al sumar otro uno al acumulador, porque en ese momento el valor que tenía era  $1(p5+3)/1777$ . De esta

manera, se consigue el valor  $1(p5+4)/1778$ , que se guarda en #s126 para obtener el valor del identificador estático y poderlo guardar en #004. Esta dirección está reservada para el sistema, ya que es usada por la máquina para identificar que proceso se está ejecutando. La máquina virtual utiliza al identificador estático para identificar los procesos, porque este nunca cambia ni se repite. Así, en el *output*, se puede colocar a que identificador estático pertenece cada salida. De ahí la necesidad de un identificador estático, para que el usuario conozca qué proceso escribió cada salida, ya que, como los procesos se van a ir intercalando, será fácil perder de vista a quién corresponde cada una.

Por último, antes de cargar el acumulador del contexto del proceso y saltar, lo que se hace en #s130 y #s131 es cambiar el valor de la bandera para permitir saltos. Es importante notar que, para el *conmutador*, las dos últimas instrucciones del sistema son consideradas como instrucciones del proceso de usuario, y hay que tenerlo en cuenta al contar los ciclos que cada proceso tiene.

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>Continuación</i>	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
	s117	2000	ADD 000	Se obtiene la 1-dir del gacc del proceso a ejecutar
<i>Preparación gpc y gcc</i>	s118	6(s132)	STO s131	En s112 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
<i>Preparación gjump</i>	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 113
<i>Salvar Static ID en 004</i>	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

Figura 3.36: SOMC Lanzamiento de procesos parte 2

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>Salvar gjump</i>	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
	s130	1(000)	LDA 000	Obtiene el número 1
<i>bandera</i>	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
	s133	8(xx)	JMP xx	La dir xx representa el valor donde el proceso se ejecutará
<i>LastDirectionSO</i>	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
<i>Reiniciar id organizer</i>	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
<i>Regresar</i>	s140	8(s110)	JMP s109	Regresar para lanzar el proceso

Figura 3.37: SOMC Lanzamiento de procesos parte 3

## Lanzamiento de procesos en la máquina virtual

Observemos ahora la figura 3.38, en la cual las variables del sistema que representan a los *organizadores*,  $c6$  y  $c7$ , contienen la dirección del *ID contador* del proceso que se va a lanzar y el contenido del mismo, respectivamente.



Figura 3.38: Estatus de organizadores antes de lanzar el proceso

Para observar la situación del sistema antes de lanzar el proceso, veamos la imagen 3.39, donde se muestra que en  $s133$ , la instrucción que se va a ejecutar es un  $8110$ , es decir, ya va a saltar al proceso. Saltará 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, ya 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  $\#932$  verificó la bandera, y como esta ya permite saltos, reinició su valor a cero; antes de ejecutar  $\#933$  aún no ha sumado nada a ese ciclo, cuando lo termine, le sumará un uno. Por lo tanto, el proceso del usuario iniciará con un ciclo ya utilizado.

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

Avanzamos hasta el final del proceso en la imagen 3.40b, donde vemos al proceso justo antes de ceder los recursos al SOM nuevamente. Como el contador del conmutador ya tiene el valor de 30, la instrucción que se encuentra apuntada en  $\#119$  ya no será ejecutada y el contador de programa saltará directamente al preámbulo.

Pero primero, observemos el estatus del proceso. Vemos que en el *output* ya hay impresos cuatro valores. El primero lo imprimió cuando “finalizó” el proceso 0, por eso aparece como

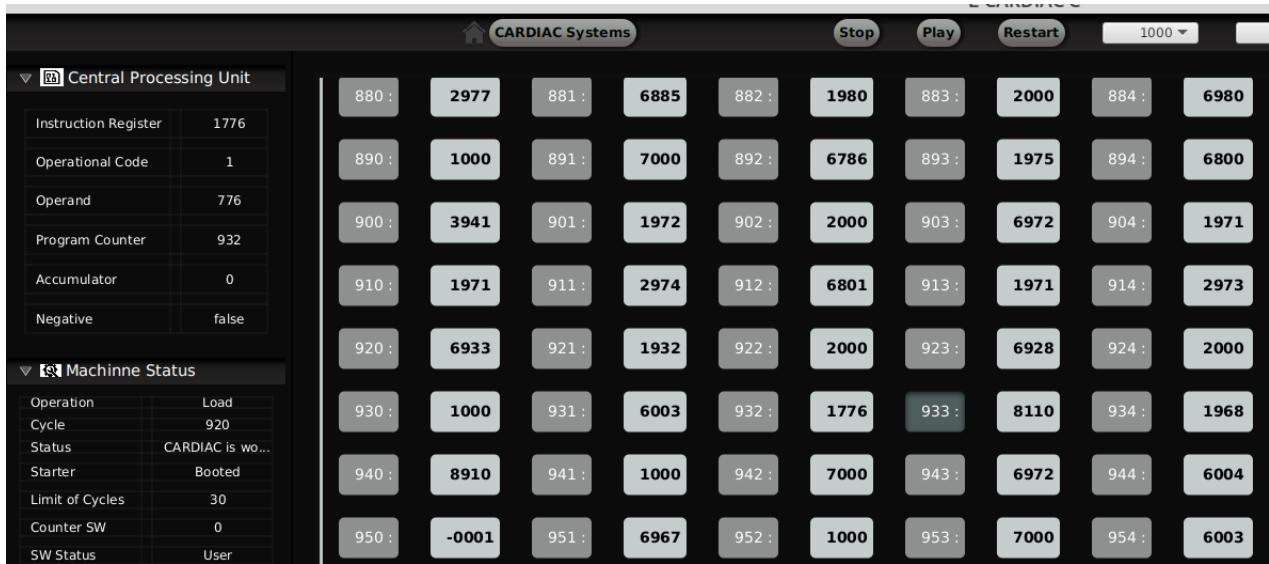


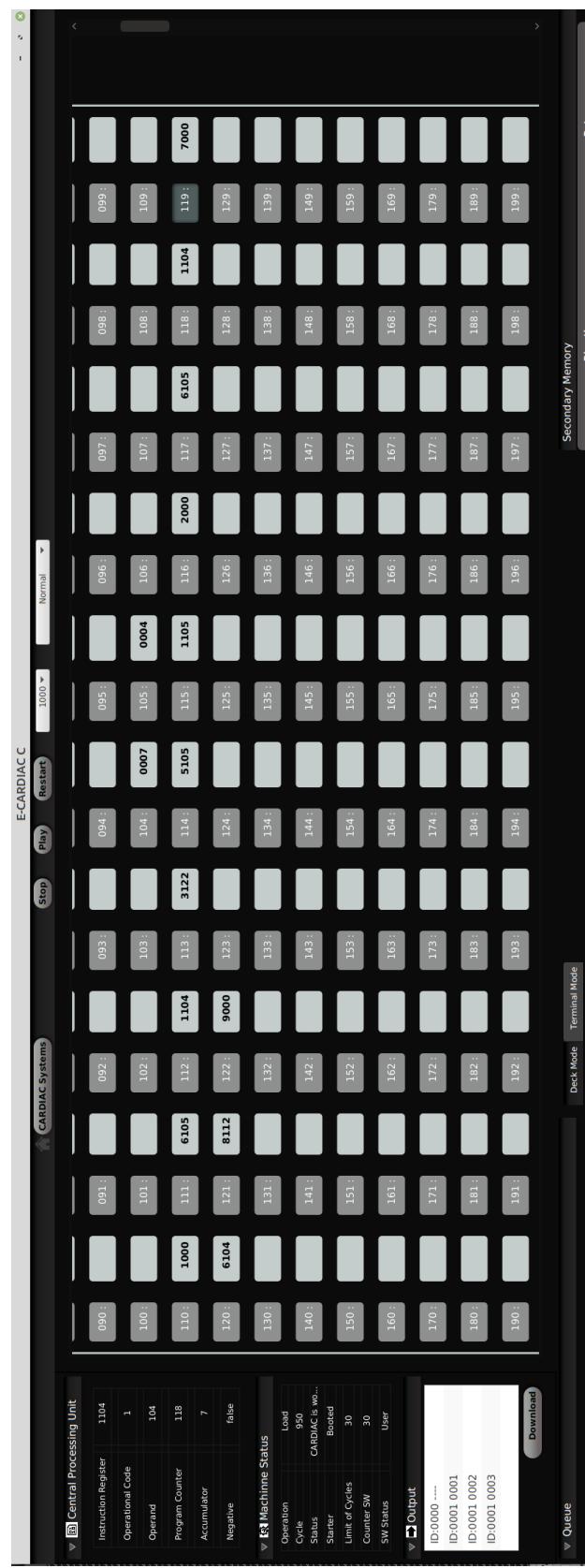
Figura 3.39: Valores del SOM antes de lanzar el proceso

identificador ese número seguido de guiones, que indican que ese proceso ha finalizado. Luego, hay salidas 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 vez que nos refiramos a un proceso, no lo haremos 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 uno, 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, no se alcanzó a escribir el 4 en esta vuelta, pero en la siguiente ejecución lo hará.

▼ 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



(b) Final de la primera vuelta de *pintor v1*

Figura 3.40: E-CARDIAC C: Ejecución de procesos

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

Tenemos un proceso que acaba de ceder sus recursos nuevamente al sistema operativo. Por lo tanto, la primera acción del sistema es guardar los valores asociados a él. El salto que hace el conmutador es directo a  $e1/\#951$ , el preámbulo, que guarda el valor del contador de programa, del acumulador y del valor que hay en  $\#999$ ; el cual no se ve afectado por este salto, ya que no se hace con la instrucción de *jump*, sino con el conmutador. También, se cambia el valor de la bandera para no permitir saltos. Y como en este caso  $e0$  no tiene un valor negativo, puesto que no se llegó al preámbulo por una instrucción *halt*, el salto que se haga desde el preámbulo será a  $s2/\#802$ : *la zona de actualización*.

#### Fracción *actualizar proceso*

Podemos observar el flujo de ejecución en la fracción verde del diagrama general y en la figura 3.41, donde se hace un acercamiento a esa zona, a la cual se llega después de haber pasado, necesariamente, por el preámbulo.

Para analizar cómo se hace la actualización, veamos la imagen 3.42, que contiene el código de la fracción, en conjunto con el diagrama. Lo que hace en términos generales es actualizar el contexto del proceso, es decir, actualiza los valores del contador de programa, del 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* del proceso a actualizar a  $s1$ , pues será un valor pivote para conocer en qué dirección está el contexto del proceso que estaba en ejecución.

Lo que sucede cuando la dirección  $\#s2$  es apuntada es la obtención de la *6-dir* del *ID contador* del proceso, dirección que se encuentra en  $\#s1$ . Suponiendo que pasa la validación, la siguiente acción es la suma de un *1* al valor del acumulador. Con ello, se obtiene la *6-dir* del *gpc* del proceso, es decir, 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 la dirección  $s7/\#807$ , se encuentra  $6(px+1)/6775$ . La instrucción que ejecuta antes de esta es la que recupera de  $\#c1$  el último contador de programa que tuvo el proceso antes de saltar, contador que fue resguardado en  $\#c1$  por el



Figura 3.41: Diagrama de actualización de proceso

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 la fracción de actualización coloca 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 hacia el lanzamiento de procesos, regresemos a la validación y veamos qué sucede si no es aprobada. 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 lo tanto, como vemos en la imagen 3.44, el flujo sigue hacia el apartado que verifica si ya no hay más procesos para ejecutar y, por lo tanto, si es que es necesario lanzar de nuevo el proceso 0 y reiniciar

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

Figura 3.42: SOMC Actualizar proceso

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

Figura 3.43: Actualizar proceso 1

los organizadores; en caso de que sí haya más procesos para ejecutar, seguirá a la zona de lanzamiento de procesos. Para la situación en que el contenido de #s1 no sea negativo, el flujo actualiza el contexto del proceso, 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.

El siguiente paso es en la fracción naranja, donde los “organizadores” dictan el siguiente proceso a ser lanzado. Para este caso, será el proceso 2, luego el proceso 3, y entonces vuelve a tocarle el turno al proceso 1 para seguir en el mismo orden de ejecuciones hasta que alguno finalice.

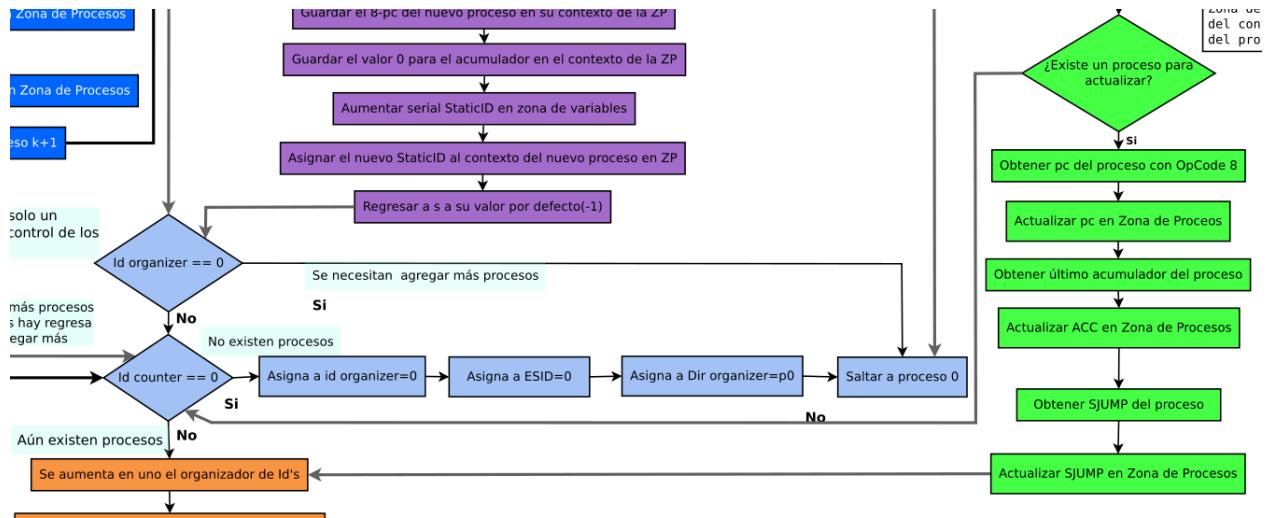


Figura 3.44: Conexión entre actualización de procesos y lanzamiento



Figura 3.45: Contexto del proceso 1 actualizado

### Fracción *borrar proceso* : Funcionamiento del borrado de un proceso

Después de varias iteraciones, tenemos el estatus que se muestra 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 *proceso 2*. Actualmente, los recursos los tiene el *proceso 1*, que va a saltar a la dirección #122, porque cuando llegue al condicional de la dirección #113, el valor que contenga el acumulador será negativo, indicando el final del *proceso 1*. Como la instrucción que está en #122 es un *halt*, 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 e0/#950 está la marca de un número negativo, y que en el *output* lo último que se encuentra es una salida correspondiente al *proceso 1*, con unas líneas que indican que el proceso terminó satisfactoriamente.

En la siguiente figura, la figura 3.48, observamos que el salto para salir del preámbulo es hacia la dirección #819, la zona de borrado de procesos, debido a que e0 tiene un valor negativo. En el diagrama general y en la imagen 3.49, podemos ver el flujo que sigue un proceso para ser borrado. Como ya vimos anteriormente, los primeros pasos son convertir el valor que está en #s1 de una 6-dir a una 1-dir, y verificar que no se trate del *proceso 0*. Para

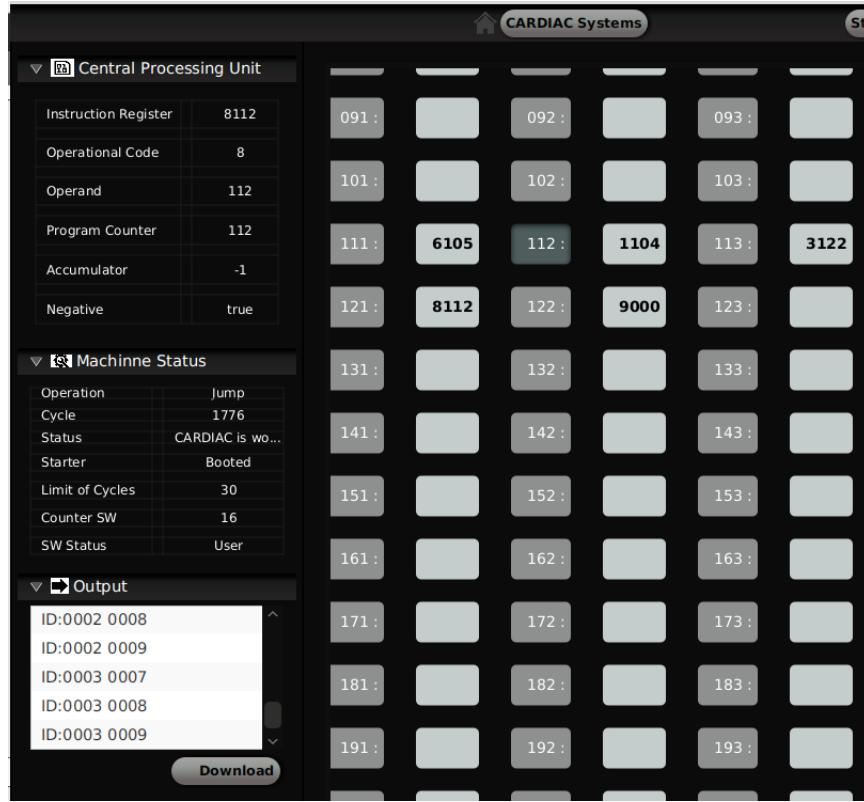


Figura 3.46: Proceso 1 antes de finalizar

seguir el código, podemos ver la imagen 3.33, además de la 3.52 y la 3.53, 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 hay una condicional que verifica que no se trate del proceso 0; si no se trata de él, permite el borrado. Llamaremos al proceso a borrar *proceso k*, en este caso será el proceso 1. El significado de borrar un proceso en este modelo, 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 los disponibles.

Notarán, asimismo, que no he mencionado el programa como tal que está en la dirección #110. Esto se debe a que no importa lo que haya en esa dirección, para el sistema lo que importa es lo que se encuentra en la zona de procesos. No importa que exista un programa en alguna parte de la memoria; si no está ligado a la zona de procesos con su correspondiente contexto, para el sistema solo es basura.

Entonces, para eliminar un proceso del usuario, lo que se necesita es volver negativo el

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 :	
985 :		986 :		987 :		988 :		989 :	

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

*ID contador* de dicho proceso, si es que es el último en la lista, y disminuir el *ID contador general* (#c4) que está en la zona de variables. No hay necesidad de borrar el contenido del resto del contexto, ya que se sobrescribirá cuando haya otro proceso; lo importante es el *ID contador*, porque así es como el sistema nota si un proceso existe. Si el proceso a borrar en este caso fuese el número 3, ese es el flujo que seguiría, y después continuaría hacia la fracción azul claro, como en la imagen 3.49.

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, ya que el sistema espera que los procesos sean consecutivos y con un *ID contador* ascendente, puesto que está preparado para buscar procesos de forma ascendente según los identificadores y detenerse cuando encuentre un número negativo como identificador.

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

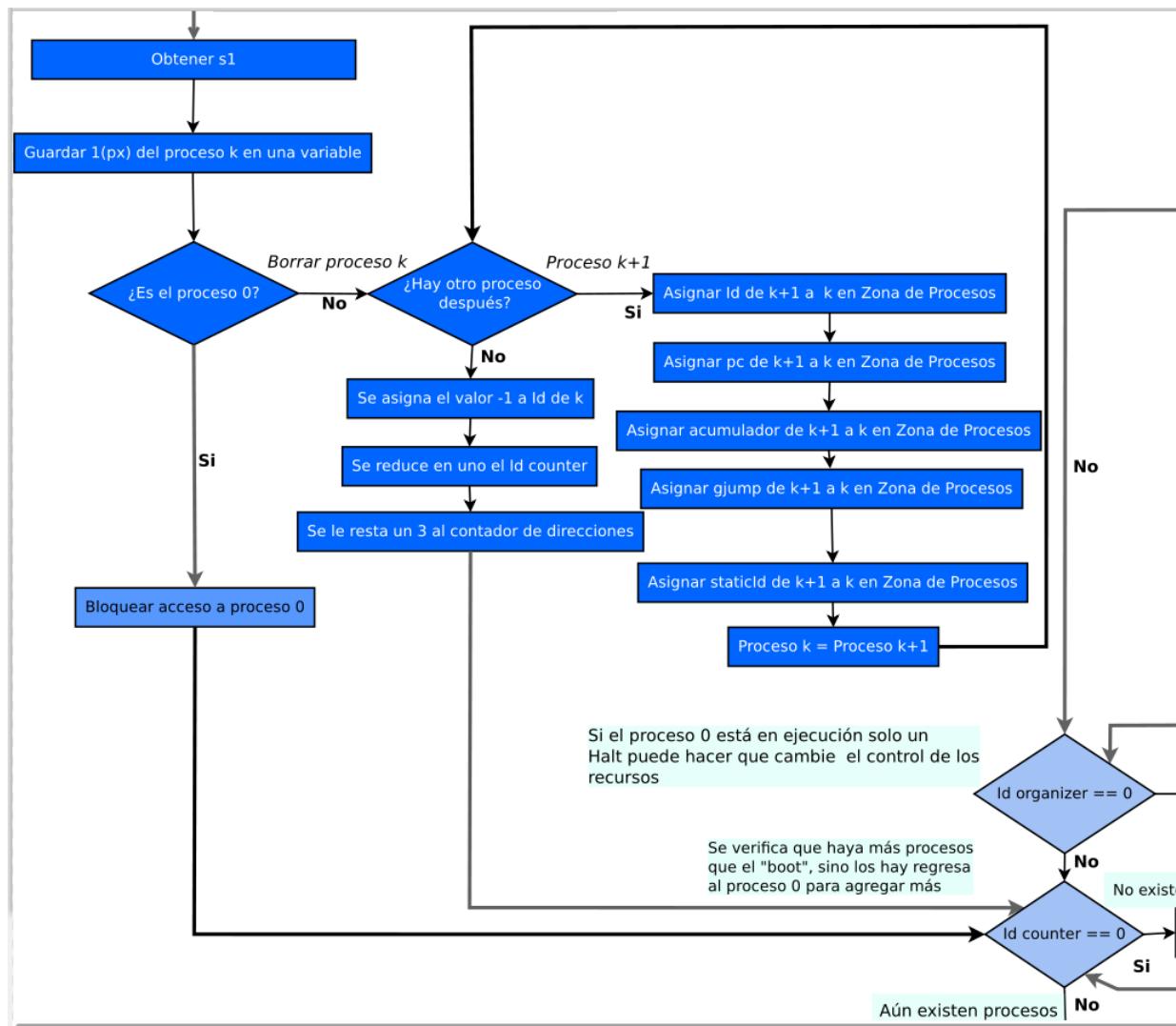


Figura 3.49: Acercamiento a zona de borrado en diagrama

valor ascendente y continuo; no puede existir un *proceso 2* si no existe un *proceso 1*. Lo que se hace para borrar el proceso, es convertir el ID contador con el valor más alto en *-1*; en el caso que revisamos, el valor más alto es el *3*. Por ende, el *proceso 3* ahora tendrá un *ID contador* con un valor de *2*, pero guarda su identidad por el identificador estático; así, tanto el sistema como nosotros podremos identificar siempre al *proceso 3*, sin importar el valor de su *ID contador*.

En la imagen 3.50, vemos la zona de procesos antes de que el *proceso 1* sea borrado, aún todos los *ID contadores* y los *ID estáticos* son iguales. Mientras que en la imagen 3.51, que muestra la zona de procesos después de borrar el *proceso 1*, vemos que ahora el identificador

estático que se encuentra en #778 es el 2, diferente al *ID contador* del proceso que ocupa ese contexto. Lo mismo ocurre para el *proceso 3*, que está en el contexto que inicia con el *ID contador* 2. Podemos notar fácilmente cómo se han recorrido los procesos. Además, podemos ver que donde antes estaba el *ID contador* 3, en #784, ahora hay un -1. Esto marca el fin de la lista de procesos, aunque el resto de las variables de contexto aún contengan valores; de hecho, son los mismos valores que tiene el *proceso 3* en sus nuevas direcciones. Esto es porque se quedan ahí como basura y se sobrescribirán 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 :	<b>0000</b>
770 :	<b>8000</b>	771 :	<b>0000</b>	772 :	<b>8000</b>	773 :	<b>0000</b>	774 :	<b>0001</b>
775 :	<b>8117</b>	776 :	<b>0010</b>	777 :	<b>8121</b>	778 :	<b>0001</b>	779 :	<b>0002</b>
780 :	<b>8317</b>	781 :	<b>0010</b>	782 :	<b>8321</b>	783 :	<b>0002</b>	784 :	<b>0003</b>
785 :	<b>8517</b>	786 :	<b>0010</b>	787 :	<b>8521</b>	788 :	<b>0003</b>	789 :	<b>-0001</b>

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

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>8317</b>	776 :	<b>0010</b>	777 :	<b>8321</b>	778 :	<b>0002</b>	779 :	<b>0002</b>
780 :	<b>8517</b>	781 :	<b>0010</b>	782 :	<b>8521</b>	783 :	<b>0003</b>	784 :	<b>-0001</b>
785 :	<b>8517</b>	786 :	<b>0010</b>	787 :	<b>8521</b>	788 :	<b>0003</b>	789 :	<b>-0001</b>

Figura 3.51: Zona de procesos después de borrar el proceso 1

## ¿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, lo que necesitamos primero es verificar si hay más procesos con *ID contadores* mayores; para ello, 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 3.52 y en la 3.53. Se toma como pivote la dirección que fue obtenida en #s27 y que es guardada en #c17, para el fácil acceso de la subrutina, ya que 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 ( $k+1$ ), y si a esa dirección se le resta 5, se obtendrá la dirección donde está el *gpc* del proceso que está siendo borrado ( $k$ ). 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  $k$ .

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,gju mp,staticid de py a px</i>	s39	6(s44)	STO s44	Se guarda en s44 para tener obtener el valor de la sección del proceso
	s40	7c11	SUB C11	Obtiene la dirección correspondiente del gpc(u otra sección) del proceso px
	s41	4011	SHT 11	Se convierte 1(px)+u en 0(px)+u
	s42	2(c9)	ADD c9	Se convierte 0(px)+u en 6(px)+u
	s43	6(s45)	STO s45	Se guarda en s45 para que sea cargada la sección en px
	s44	1(py)+u	LDA py+u	
	s45	6(px)+u	STO px+u	
<i>Aumentar contador de secciones</i>	s46	1(c0)	LDA c0	Obtener contador de contextos
	s47	7000	ADD 000	Se le resta uno al contador de contextos
	s48	6(c0)	STO c0	Lo guarda en c0 de nuevo
<i>Regresar a recorrer</i>	s49	8(s35)	JMP s35	

Figura 3.52: SOMC Borrar proceso parte 2

Después de completar el copiado de contexto del proceso  $k+1$  a la zona del proceso  $k$ , viene otra iteración, pues la subrutina de copiado está anidada en la iteración para recorrer 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. Esto provoca que ahora el proceso  $k+1$  se convierta en el proceso  $k$ . Por lo tanto, la dirección que ese proceso ocupaba ahora tiene que ser limpiada, ya sea recorriendo otro proceso (si existe) o dejar 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 del *proceso k*. Entonces, salta a #s52, donde se obtiene la dirección del *ID contador* del proceso  $k$  y se actualiza con un valor de -1. En este caso ocurre sólo cuando el proceso a borrar es el último de la lista y no se tienen que recorrer procesos. Por lo tanto, cada vez que se da la instrucción de borrar un proceso de usuario, esto sucede; aunque se recorran los procesos, siempre quedará un contexto de proceso al final que hay que limpiar.

El siguiente paso 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 3.53, a partir de la dirección *s59*. Al finalizar, salta hacia la fracción de gestión del proceso 0.

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>¿Hay proceso después de (k+1)?</i> "Si" de s30	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
	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)
<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
<i>M[c4]--</i>	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
	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
<i>M[c5]=M[c5]-5</i>	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 3.53: SOMC Borrar proceso parte 3

En la figura 3.54, vemos los contadores de identificadores y direcciones,  $c4/\#969$  y  $c5/\#970$ , antes de que el *proceso 1* fuese borrado. Se marcaba un 3 en el contador de identificadores, indicando que había 3 procesos activos, y un 784 en el contador de direcciones, indicando la dirección del último *ID 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.

Algo curioso pasa con los organizadores, ya que, como los organizadores lanzan los procesos de acuerdo a su posición en la lista, es decir, de acuerdo a su *ID contador*, esto puede

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

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

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

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

causar desfases en la elección del siguiente proceso a ejecutar. Si nos colocamos en la situación donde se acaba de ejecutar el *proceso 1*, mismo que ha sido borrado, el que sigue es el proceso con *ID contador 2*, por lo que se estaría saltando para esta ejecución al proceso con *identificador estático 2*, que ahora tiene un *ID contador* igual a 1. Por lo tanto, el siguiente proceso a ejecutar sería el proceso con *identificador estático 3*, pero 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 el proceso 2.

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

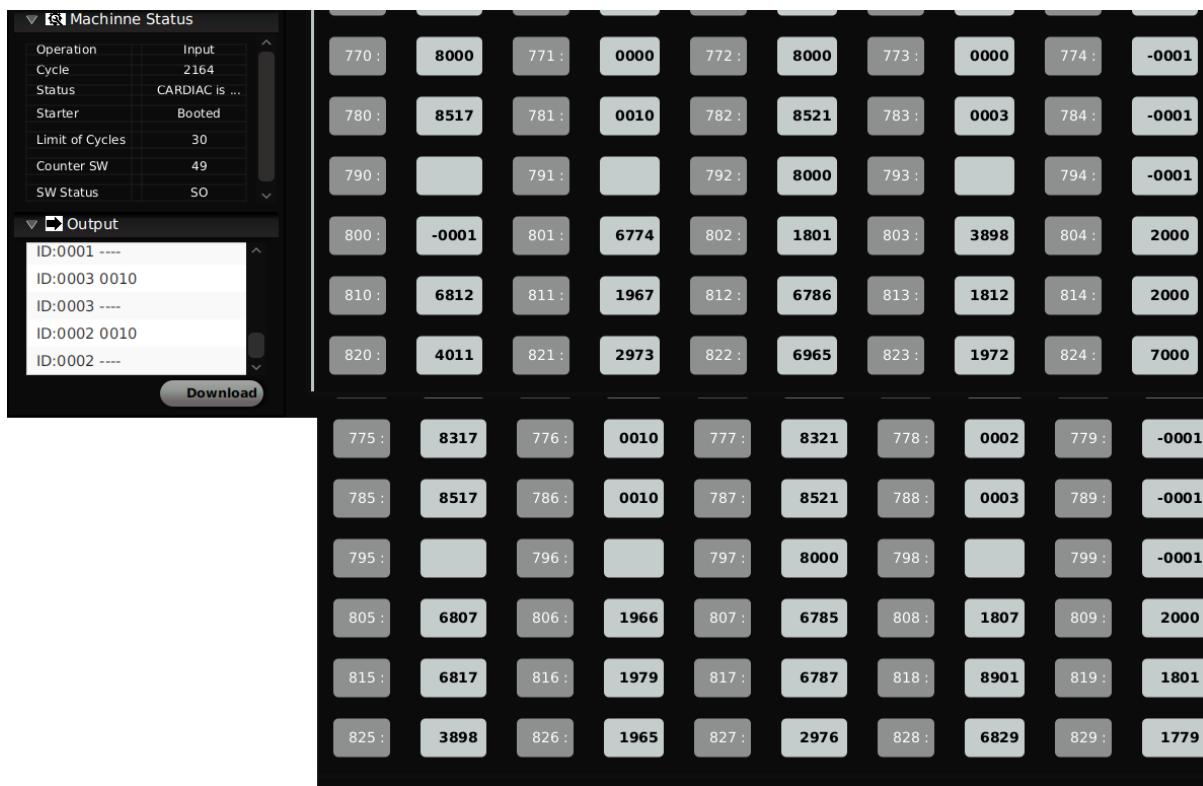


Figura 3.56: Salidas finales de los procesos

```
CARDIAC_Output_2024...-06 21:01:05.433.txt x
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. Sistema Operativo Mínimo C: 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 de **E-CARDIAC C** en conjunto con el 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 con la arquitectura de la máquina y 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 qué lugar de la memoria cargar cada instrucción, de forma que se pueda tener una “tarjeta” con instrucciones pareadas, donde 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. Donde *0800* indica que se cargue la segunda instrucción en la primera dirección del SOMC, 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 SOMC.
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 borrarlo, sino para pausarlo y empezar la ejecución de los procesos que el usuario ha cargado.

8. Esperar la ejecución de los procesos. Cuando hayan terminado, se podrán descargar los resultados del área de *output*, donde cada salida indica a qué proceso pertenece (según su identificador estático). 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 pueda apreciar 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. 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. Otro, podría ser cuando pláticas con alguien por mensaje de texto y con alguien más en persona “al mismo tiempo”, pero le das más atención a determinada conversación según el momento.

Sin embargo, también hay actividades que podemos realizar realmente al mismo tiempo, aunque para eso son importantes los diversos sentidos que tenemos y cómo los orientamos. Por ejemplo, cuando leemos un libro y escuchamos música, es difícil poner atención a ambos, 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 anti-natural. 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 de manera concurrente. Este fue el problema más importante de resolver al momento de diseñar un modelo paralelo de *CARDIAC*, pues este, 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.

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. 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 involucrados.

Para diseñar un modelo que sea intuitivo para el usuario y capaz de explicar las repercusiones que tiene la implementación del paralelismo, opté por crear un modelo que tenga

dos procesadores, pero que tengan usos diferentes. De esta manera, tendremos un procesador principal y un *co-procesador*, lo que da como resultado que el procesador principal se encargue de algunas tareas y que el *coprocesador* sea un apoyo en otras. Así logramos hacer una analogía a lo que hace nuestra mente con los sentidos, ya que cada procesador estará centrado en cierta parte del funcionamiento de la computadora.

El modelo creado lleva por nombre **Electronic CARDboard Illustrative Aid to Concurrent Parallel Computation** (*E-CARDIAC PC*), o ayuda ilustrativa de cartulina electrónica para la computación paralela y concurrente, por sus siglas en inglés. Al igual que el modelo anterior, su nombre hace referencia a un hecho histórico en la computación: la computadora personal, también conocida como *PC*.

### 3.3.1. Arquitectura renovada para un modelo paralelo

Para entender este nuevo modelo, lo primero que hay que hacer es ver el diagrama de su arquitectura. Este diagrama, aunque aún se basa en el del modelo original, ha recibido bastantes modificaciones debido a la inclusión de un *coprocesador*, el cual será difícil diferenciar del procesador principal, porque ambos realizarán tareas muy importantes para el modelo de cómputo. En la figura 3.58 se encuentra el diagrama para esta arquitectura, que mantiene varios elementos de su antecesor: la memoria principal, los buses, la memoria secundaria y el *output*. Sin embargo, también hay otros elementos modificados o totalmente nuevos.

Empecemos por lo más llamativo: las dos *CPU*. Una en azul, llamada *CPU Loader* (la cargadora), y otra en rojo, llamada *CPU Executor* (la ejecutora). Cada una tendrá tareas particulares para el modelo. La principal es *la cargadora*, puesto que esta es la que está conectada a la memoria secundaria y a la entrada (*input*) principal, la cual se conecta a la memoria secundaria y al usuario directamente por medio del modo de entrada de tarjetas (*deck mode*), que consiste en la entrada masiva de información. Mientras que *la ejecutora* es el coprocesador, porque solo realizará un número determinado de tareas que el usuario haya solicitado desde la CPU principal.

Mencionaba que sería difícil diferenciar a la CPU principal de la que no lo es porque la segunda, la ejecutora, será la encargada de ejecutar como tal los procesos; la cargadora solo se encarga de cargar el sistema operativo y de permitir al usuario la interacción con la

máquina. Cabe mencionar que, además de esas responsabilidades, la ejecutora también tiene una conexión a un método de entrada: el método que en la máquina virtual está en la pestaña de *terminal mode*, pues habrá procesos que requieran de información del usuario y, por lo tanto, la ejecutora también requerirá una forma en que el usuario pueda interactuar. Con esto, se crea una división en los métodos de entrada, división que no existía en los modelos anteriores, pues ambos métodos de entrada servían para exactamente lo mismo.

También podemos observar que la CPU principal tiene menos componentes que la CPU de *E-CARDIAC C*, puesto que, de los elementos nuevos en el modelo concurrente, solo mantiene el *starter* para poder cargar el sistema operativo mínimo, ya que es el que tiene la conexión a la memoria secundaria. Pero el *switcher/conmutador*, como no lo necesita, se quedó únicamente en la ejecutora. No necesita al conmutador porque una vez que se cargue el sistema operativo mínimo, el usuario solo usará la CPU principal para estar cargando programas por medio del *proceso 0*, y es el usuario mismo quien da la instrucción de ceder el control al SOM para que este añada el proceso que será ejecutado por la CPU secundaria. Al terminar de añadir el proceso, los recursos de la cargadora regresarán de inmediato al *proceso 0* para que el usuario pueda seguir añadiendo programas.

Por otra parte, la ejecutora sí necesita un conmutador porque estará cambiando el control de los recursos entre el sistema operativo mínimo y los procesos de usuario que esté ejecutando. Adicionalmente, requiere de un elemento extra en el hardware, un elemento que he llamado *waiter*, o vigilante, porque se encargará de vigilar si hay procesos en la *zona de procesos* para continuar con la ejecución. En caso de que no haya, mantiene a la ejecutora en una especie de pausa hasta que detecta que hay algún proceso a ejecutar. Recordemos que ambos procesadores operan en paralelo, y habrá momentos en los que no haya procesos y el coprocesador deba estar en espera.

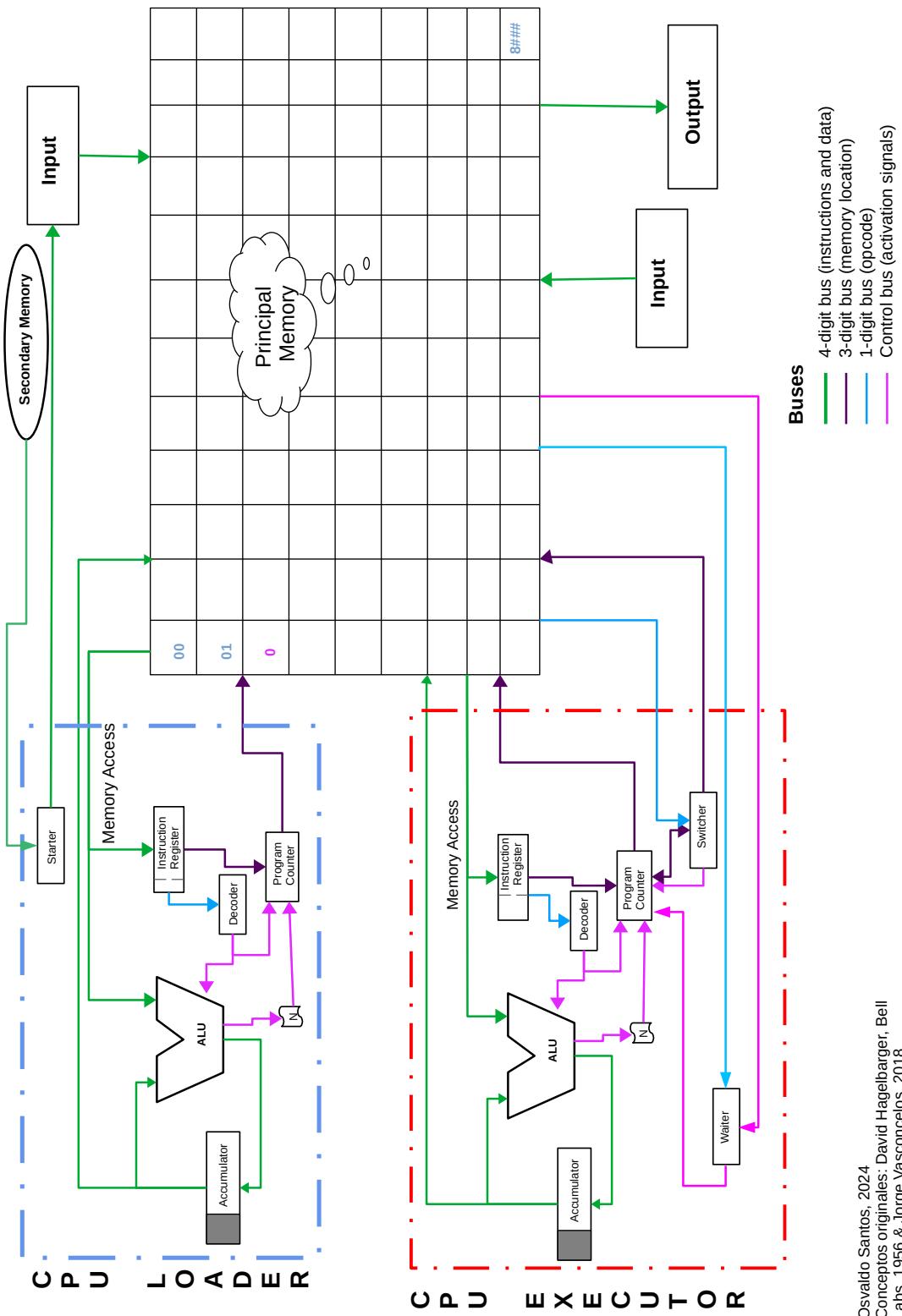


Figura 3.58: Arquitectura de cómputo paralela.

Para averiguar si hay procesos, el vigilante utiliza el bus azul, que lo conecta directamente a la memoria. Por este medio, recupera el valor que está guardado en #c4, que es el contador de identificadores general<sup>2</sup>. Si tiene un cero, significa que no hay nada para ejecutar. El mecanismo del vigilante realiza una pausa en el contador de programa si no hay procesos, e independientemente se mantiene revisando, en cada ciclo que pasa en la computadora, si ya hay algún proceso. Cuando detecta que hay al menos un proceso, deja al contador de programa avanzar a la siguiente dirección.

Por lo tanto, para ahorrar recursos, no siempre está activo. Se activa cuando el contador de programa apunta a la dirección #s97 del sistema operativo, una 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, por el cual se transfiere la información sobre si se está apuntando a la dirección #s97. Si recibe esta información, el vigilante se activa.

La otra conexión que necesita es para pausar el contador de programa. Para esto, 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, continuar con la ejecución en la dirección #s98.

Como habrán notado, se intentó duplicar lo menos posible, para que no sea una arquitectura muy costosa con dos procesadores exactamente iguales, sino dos procesadores que se complementan. De hecho, la *ALU* está diseñada para que no funcionen todas las operaciones en ambos procesadores, puesto que algunas serían innecesarias. De esta forma, 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 su complejidad en comparación con el modelo anterior.

---

<sup>2</sup>Como solo tiene valores menores a 9, no necesita más de un dígito; por eso se utiliza el bus azul.

### 3.3.2. Un sistema operativo para dos procesadores

Ahora, el reto es diseñar 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 entre ambos en la memoria. Ya que, al ser un modelo donde la memoria es compartida, se podrían dar situaciones en las que la segunda CPU cambie variables que la primera usa y produzca **errores de sincronización**, o que la segunda espere valores de la primera y la primera nunca los entregue, lo que terminará causando que la segunda 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 mínimo PC (SOMPC)*, donde podremos analizar cómo 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, la de añadir un nuevo proceso, con una parte de la fracción del *preámbulo*, el cual está partido en dos porque es parte de ambos grupos, pero en el cual no existen conexiones entre ellos.

Por lo tanto, no hay posibilidad de que las instrucciones del grupo uno, que ejecutará la segunda CPU, tengan un conflicto directo con las del grupo dos, que ejecutará la cargadora. Tienen entradas y salidas diferentes, y no hay saltos que puedan llevar de alguna fracción del grupo uno a alguna del dos; son totalmente independientes. De esta forma, cada CPU tiene restringidos espacios de memoria desde el mismo diseño del 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

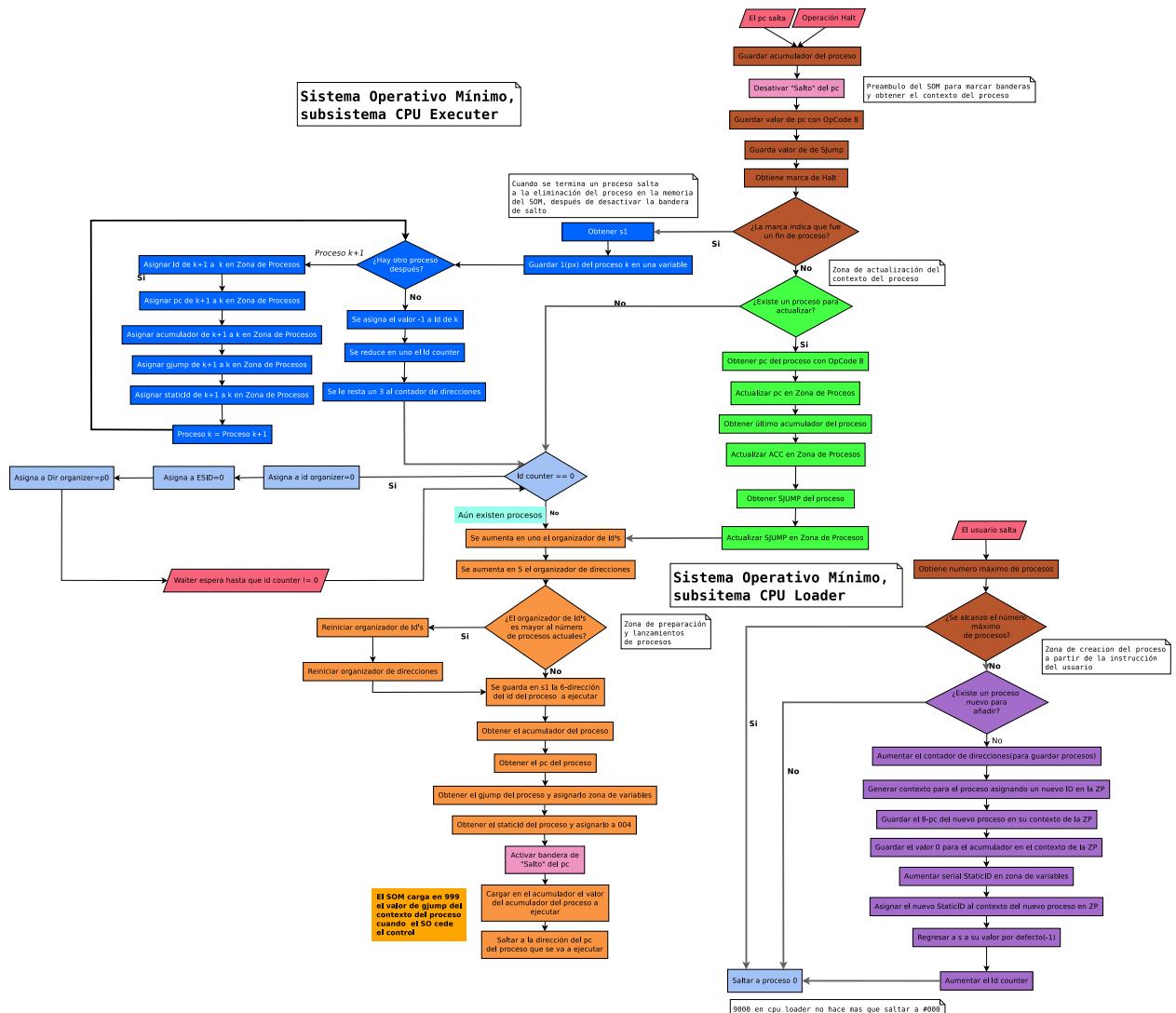


Figura 3.59: Diagrama de Sistema Operativo Mínimo Paralelo

los posibles errores o problemas de sincronización que se presentan cuando se trabaja 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, lo cual es 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 comprensión.

### 3.3.3. Funcionamiento del sistema operativo mínimo

Para explorar el sistema operativo, revisaremos por separado lo que ejecuta cada CPU, ya que al final son procesos del sistema operativo independientes entre sí, para concluir en su funcionamiento conjunto y ser capaces de entender cómo, sin interacciones directas, a través del sistema operativo las dos CPU tienen un funcionamiento homogéneo que le brinda 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. Así, fácilmente el 90 % del código y funcionamiento del sistema serán idénticos al de *C*, lo que facilitará entender cómo funciona este sistema operativo mínimo mejorado.

#### Sistema operativo para CPU Loader

Empecemos con la *CPU Loader/Cargadora*, porque es el que iniciará las operaciones de la máquina y, además, es el más corto en su funcionamiento. En la imagen 3.60, observamos un acercamiento a todo lo que realizará esta CPU para el sistema operativo. Contiene una pequeña sección de preámbulo que funciona igual 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 y, en caso de que sí, salta al *proceso 0*.

Sin embargo, ahora esta instrucción de saltar al *proceso 0* no se conecta con una fracción de gestión del *proceso 0*, sólo salta. Asimismo, si no hay un proceso nuevo para añadir, se regresa también 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 nuevamente tenga la posibilidad de añadir más procesos.

Desde la imagen 3.61, podemos observar que las instrucciones del preámbulo que corresponden al diagrama antes, visto están únicamente entre las direcciones #e14 y #e17. Tal como ocurría 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 a este por medio de la instrucción de *añadir un proceso*, el flujo iniciará necesariamente en #e1 y, de forma inevitable, terminará en un salto a #s2 o #s24.

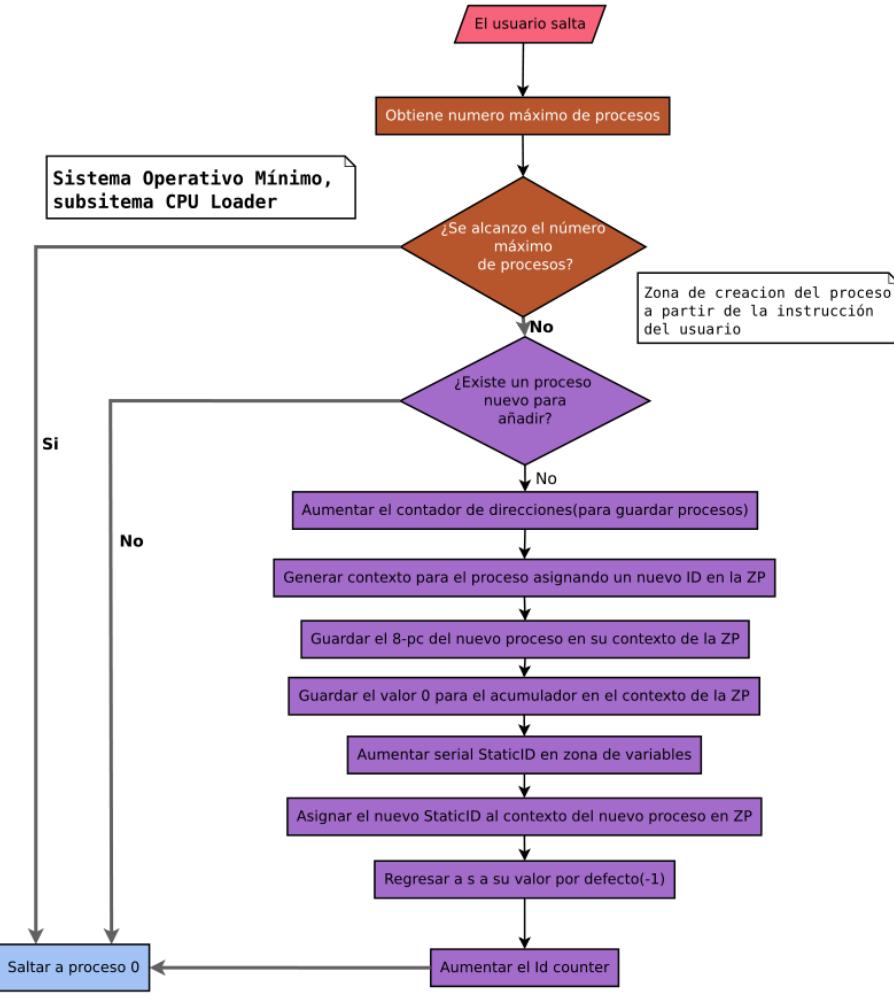


Figura 3.60: Añadir proceso en SOMP

Ahora, un detalle importante al momento de añadir un proceso, que cambia respecto de la versión para C, es el momento en que se aumenta el *ID contador* general en #c4. 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, vigilando que #c4 cambie su valor a uno distinto de 0. Cuando el *vigilante* detecta que cambió, permite que la ejecución en la *ejecutora* continúe, y realice todos los procesos necesarios para lanzar un nuevo proceso. Si el valor de #c4 cambiara al principio, como sucede en C, el vigilante permitiría que la ejecutora continúe, pero aún no estaría listo el contexto del proceso nuevo.

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 del sistema operativo en la *CPU cargadora*; lo demás, son ejecuciones del usuario y del *iniciador/starter*. El resto de las operaciones que realiza esta CPU, consisten únicamente en la carga de instrucciones que provienen del usuario a través del modo de tarjetas (*deck*), para cargar al mismo *proceso 0* y el sistema operativo en la memoria principal.

Notarán que he sido incisivo en que la carga del usuario es por medio del modo de tarjetas. Esto se debe a que, como el mismo diagrama de la arquitectura indica, habrá dos medios de entrada, a diferencia de lo que sucedía en C, donde existía un sólo 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 evitar conflictos si ambas requieran hacer uso del periférico al mismo tiempo. Así, el método de tarjetas queda reservado para la cargadora, y el método de la terminal para la ejecutora.

Las salidas también tendrán conexiones diferentes para optimizar recursos. La *ALU* de la cargadora no reconocerá la operación *output*, y simplemente no ejecutará nada, ya que el único que necesita realizar impresiones es la ejecutora. Otra instrucción que tendrá un comportamiento limitado en la cargadora será la instrucción de *halt*, ya que, 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)
	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
<i>Previa de pc</i>	s75	2000	ADD 000	Se obtiene la 6-dir del gacc del nuevo proceso
	s76	6(s87)	STO s81	En s75 se guarda 6(px)+2
	s77	2(c12)	ADD C12	Añade 2 para obtener 6(px)+4(Static ID)
<i>Guardar Nuevo Static ID para el proceso</i>	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
	s83	1(s)	LDA s	Se obtiene el 8-pc del proceso
<i>pc nuevo</i>	s84	6(px)+1	STO px+1	Se guarda el 8-pc en el gpc del proceso
	s85	1000	LDA 000	
	s86	7000	SUB 000	Se obtiene el 0
<i>acc del nuevo proceso</i>	s87	6(px)+2	STO px+2	El gacc del proceso se inicializa en 0
	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. En ella notamos que los dos métodos de entrada se mantienen, pero que ahora servirán a diferentes CPU, y por supuesto, ahora tenemos dos CPU, una en cada lateral. Al tener dos CPU, fue necesario también reordenar el contenido del estado de la máquina (*machine status*), ya que varios de sus elementos eran propios de la CPU, y ahora que hay dos, es necesario diferenciar a qué CPU pertenece cada uno. Además, en ese mismo apartado, se debe añadir otro contador de ciclos y otro visualizador de operaciones, para observar que se está ejecutando en cada CPU.

También fue necesario reordenar algunos elementos de la pantalla inicial para distribuirlos de mejor manera debido a su aumento. Aunque, como podemos notar, el esquema sigue siendo muy semejante al de *C*. De hecho, el arranque es igual, después de hacer clic en *start* se iniciará la carga de las instrucciones desde la memoria secundaria, por medio del *starter*,

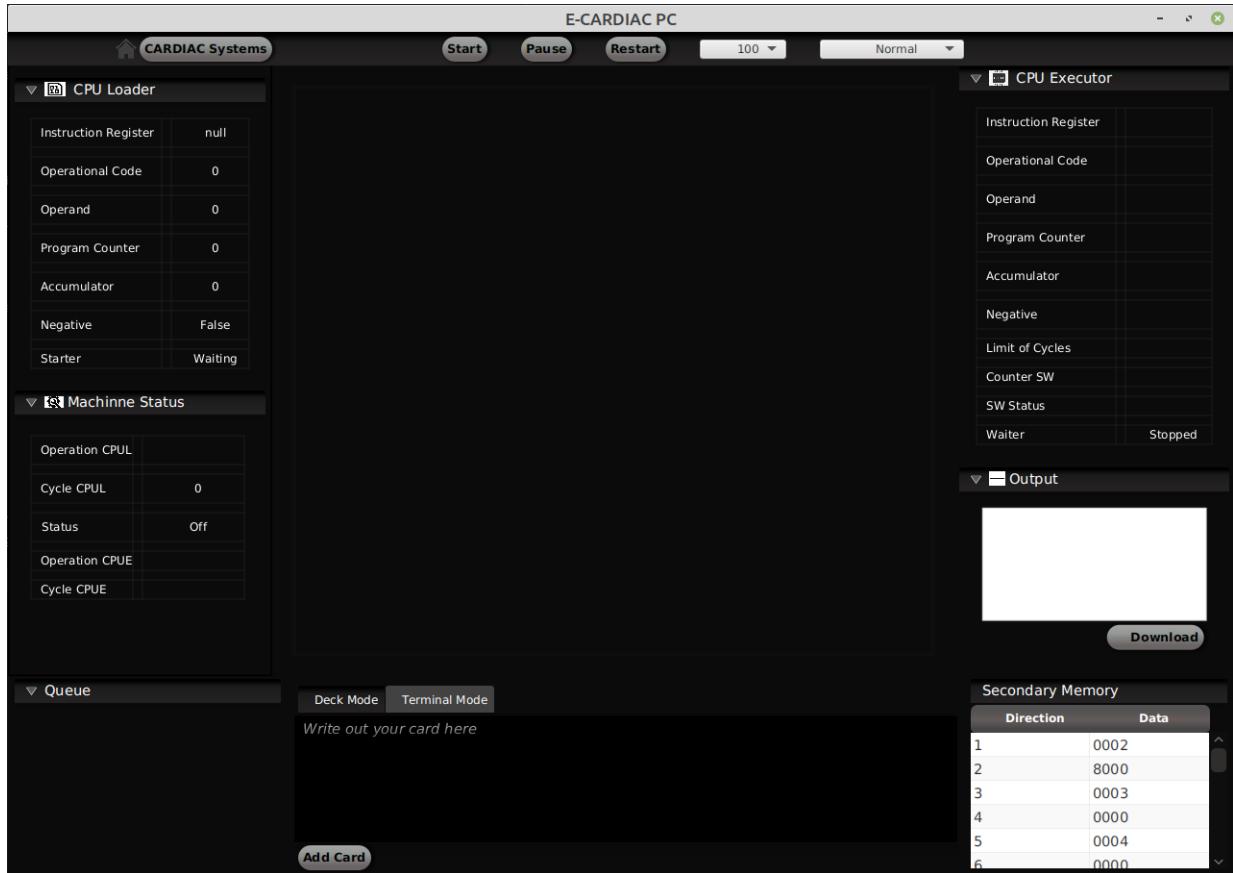


Figura 3.63: E-CARDIAC PC sin iniciar

hacia la memoria principal; todos estos movimientos los veremos en la *CPU Loader*.

En la imagen 3.64 podemos ver lo que sucede después de iniciar la carga (*booteo*). En esta, observamos básicamente lo mismo que en C, con la diferencia de que ahora podemos notar que la CPU principal está funcionando consultando al *Cycle CPUL* y *Operation CPUL*, que indican el ciclo en el que va y la instrucción que está ejecutando, respectivamente. Mientras que los correspondientes a la segunda 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 la máquina inicia, se inician las dos CPU, la primera inicia la carga a través del iniciador y continúa ejecutando instrucciones. Mientras que la segunda inicia en la dirección #s97, dirección que activa al vigilante (*waiter*), el cual impide que el contador de programa avance hasta que haya un proceso cargado por el usuario.

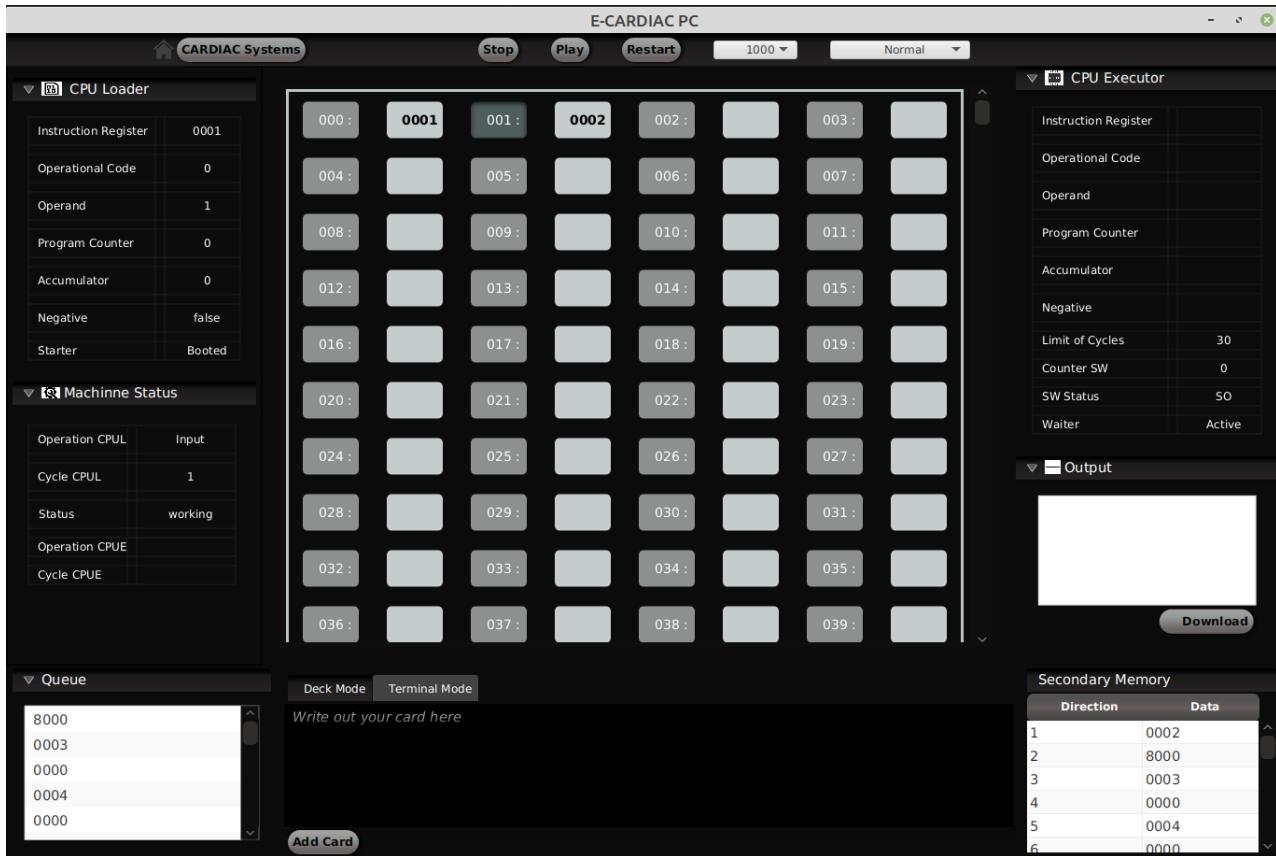


Figura 3.64: E-CARDIAC PC Booteo

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

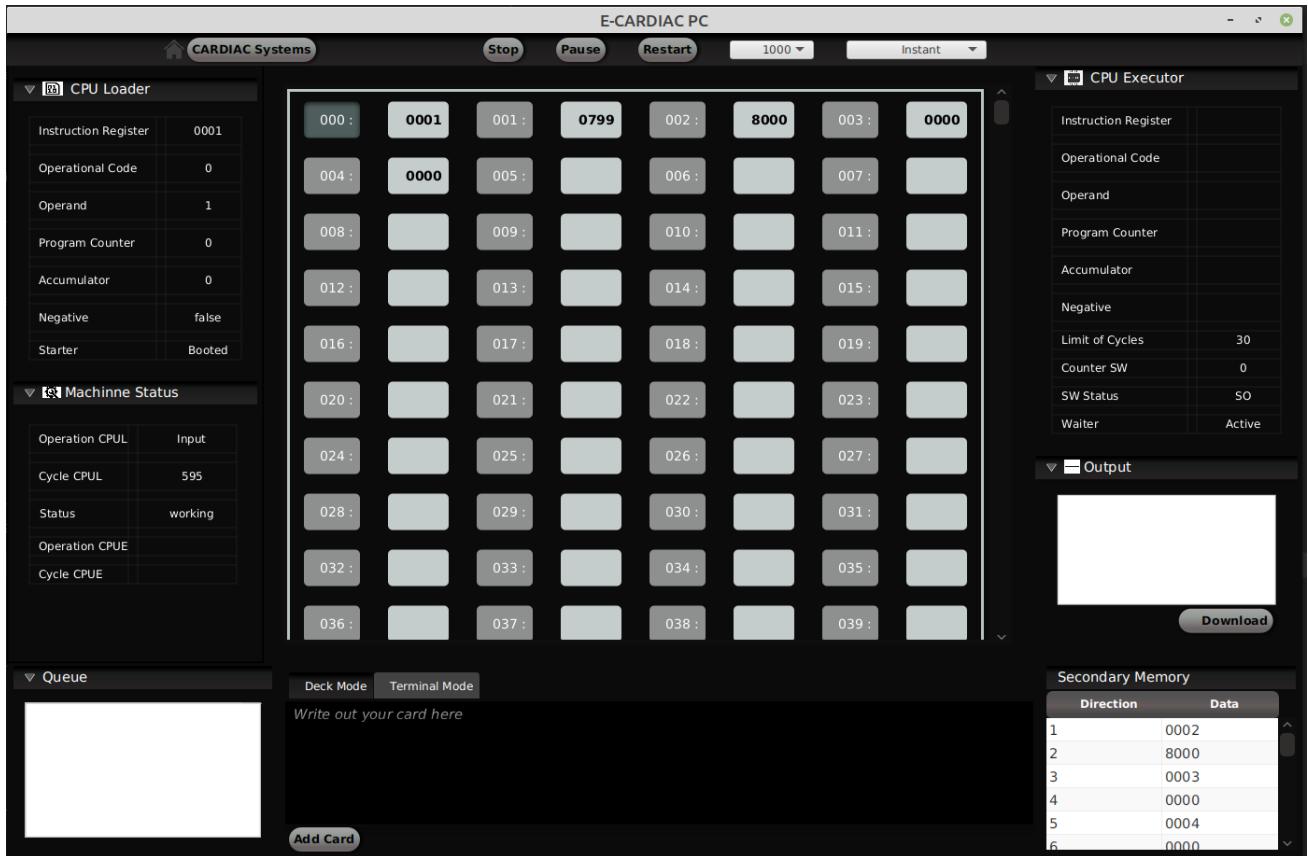


Figura 3.65: E-CARDIAC PC Iniciado

### Sistema operativo para la *CPU Executor*

Antes de cargar el primer programa en la nueva computadora, será necesario comprender cómo funciona el sistema operativo para la segunda CPU. Si observamos el diagrama general del sistema operativo (fig. 3.59) notaremos que toda la parte izquierda es ejecutada por la segunda CPU. Si nos acercamos al preámbulo (fig. 3.66) podemos notar que las dos formas de entrada que tiene son cuando el contador del programa salta en automático por medio del *conmutador* o si un proceso es terminado con la operación *halt*.

Una vez que el SOM retoma el control por cualquiera de las dos vías, realiza las operaciones que ya efectuaba para C en 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 observamos la imagen 3.61, notaremos que lo único que cambia son las direcciones a donde salta, ya que algunas instrucciones de

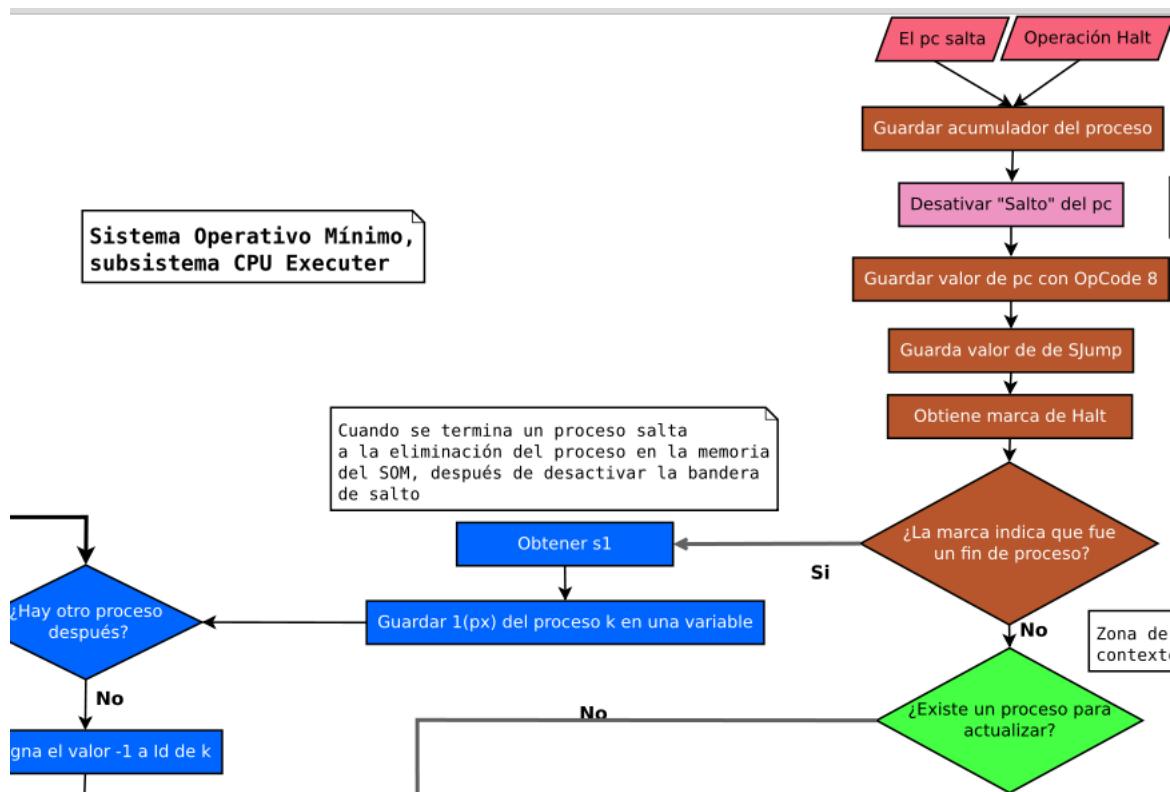


Figura 3.66: Acercamiento al preámbulo del sistema operativo mínimo

la fracción de borrado fueron modificadas, lo que cambió las posiciones para acceder a ellas. Esta fracción terminó por ser la que más redujo su tamaño respecto a su versión anterior.

Ahora, si recordamos, el sistema operativo no avanza 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 de la figura 3.67, donde está resaltado en rosa un recuadro que indica que el flujo continuará hasta que el ID contador general sea distinto de cero. El color rosa indica una especie de entrada al sistema, porque es un punto en el que este puede continuar su ejecución.

Como podemos observar, el flujo continúa desde el preámbulo hasta la fracción azul claro primero. En *C*, esta fracción 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 la llamaremos fracción de **control de procesos**. En esta fracción, se verifica la existencia de procesos; si el *ID contador general* es igual a cero, reinicia los organizadores y el identificador estático de la dirección #004 antes de activar al *vigilante*. De esta manera, cuando este permita al

contador de programa continuar porque ya existen procesos, los organizadores estarán en la posición correcta. En la imagen 3.68 se puede ver en código.

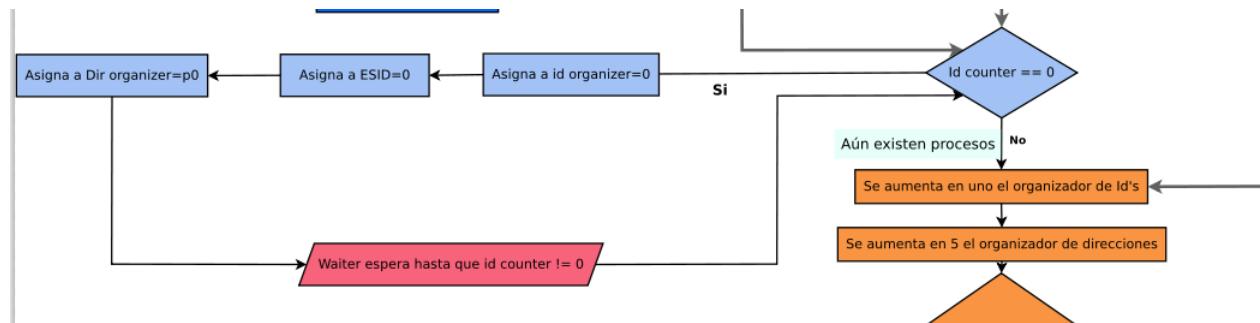


Figura 3.67: Acercamiento a segmento de *waiter* y control de procesos

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

Figura 3.68: Segmento de control de procesos

La fracción de **lanzamiento de procesos**, que se observa en el diagrama general en la parte inferior naranja y en la figura 3.69, no cambio en nada, salvo en las direcciones. Si existen procesos, se llega a este segmento, que cambia los valores de los organizadores, obtiene de la zona de procesos el contexto del proceso a ejecutar, lo coloca donde el proceso pueda tomarlo, activa el salto del *conmutador* y salta al proceso del usuario. Para conocer los códigos para el lanzamiento de procesos, podemos ver las figuras 3.70 y 3.71, que muestran que es básicamente el mismo código con el mismo funcionamiento que en C.

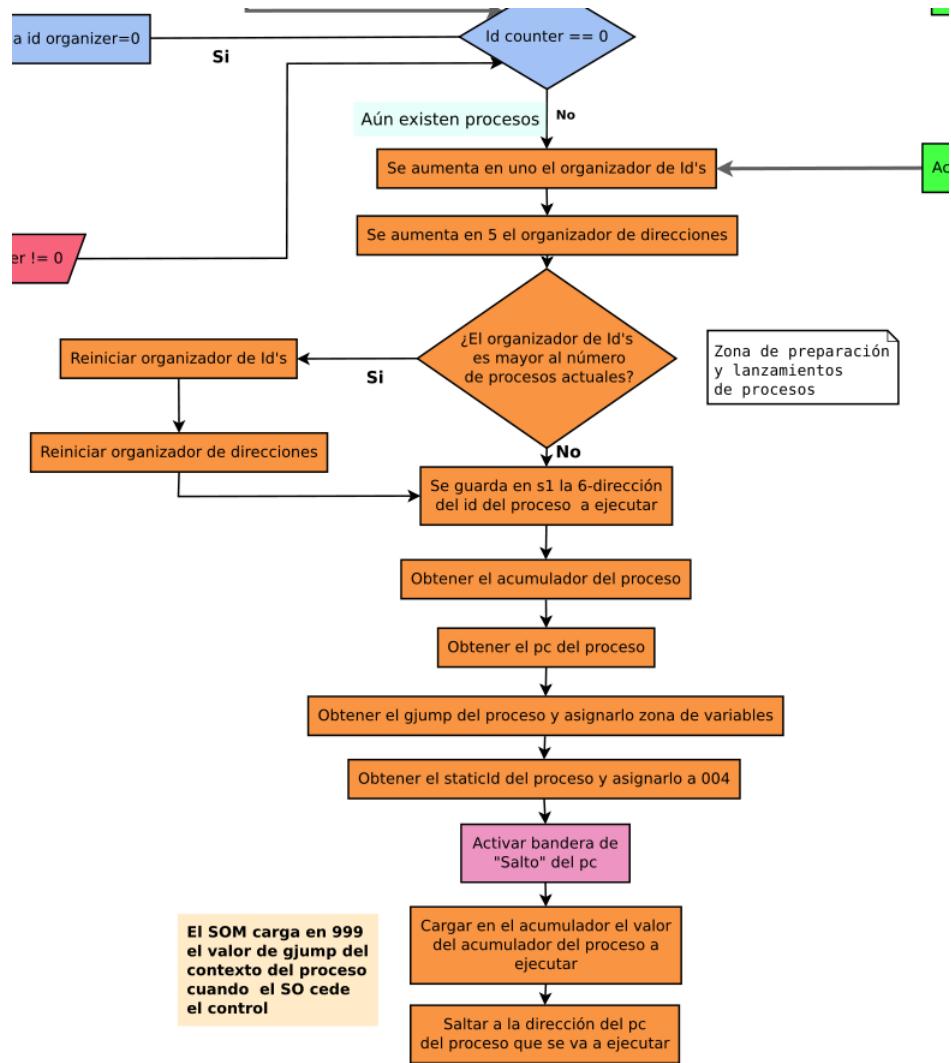


Figura 3.69: Acercamiento a fracción de lanzamiento de procesos

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
Actualizar s1 para salir al proceso con su own area	s112	6(s1)	STO s1	Se guarda en s1, será el proceso "actual"
	s113	1(c6)	LDA c6	
Continuación	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
	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
LastDirectionStar	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 general en color verde y en la figura 3.72, tampoco sufrió modificaciones, salvo el cambio de direcciones, que podemos ver en la imagen 3.73. Dado que la zona de lanzamiento procesos se mantiene sin cambios, es natural que la zona de actualización no requiera modificaciones, ya que se trata de 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)
	s2	1(s1)	LDA s1	El acumulador toma un valor de la forma 6(px)
Validación	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
Actualizar gpc	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
Actualizar gacc	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
Actualizar gjump	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
Saltar	s18	8(s101)	JMP S100	Saltamos a cambiar de proceso

Figura 3.73: Sistema operativo mínimo actualización de procesos

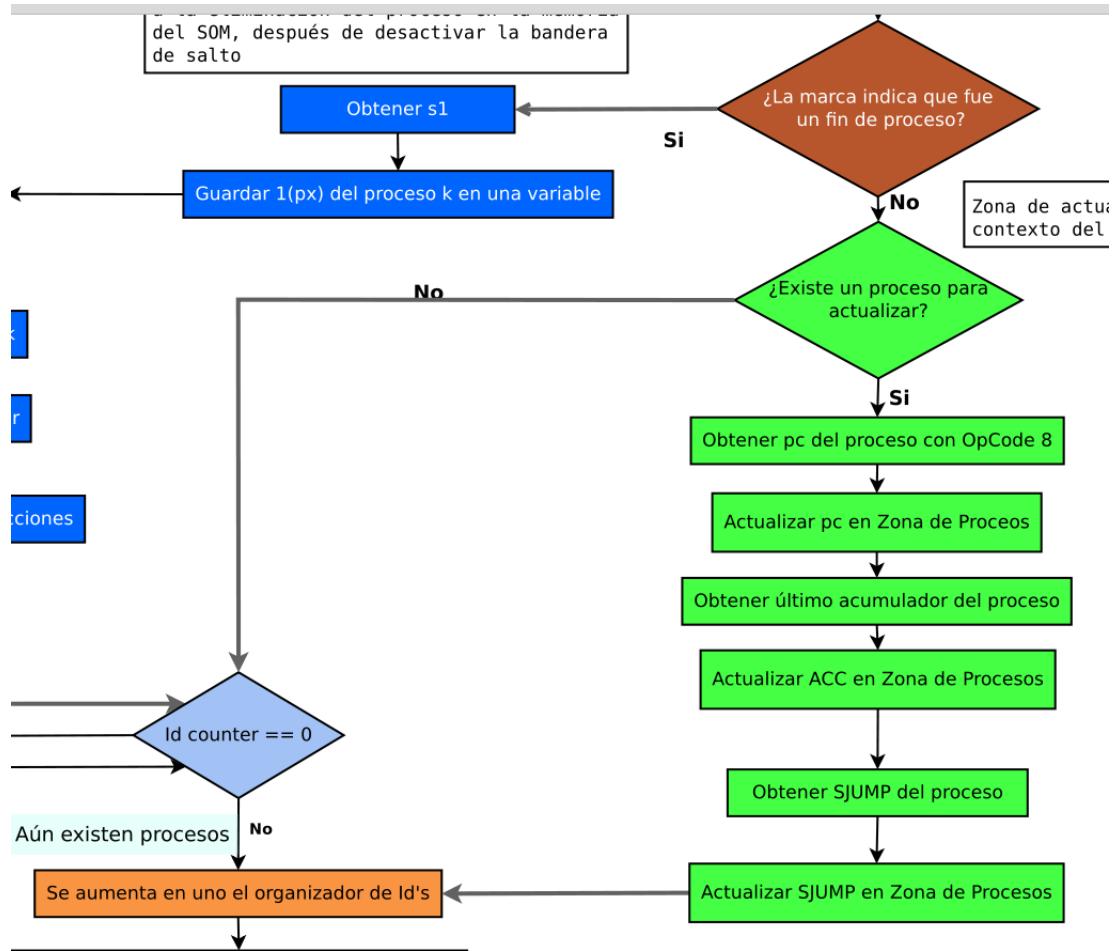


Figura 3.72: Acercamiento al segmento de actualización de procesos

Lo que sí tuvo un cambio importante es la **fracción de borrado**, que podemos observar en la figura 3.74. Aunque no cambia la forma de borrar, ya que esas subrutas permanecen sin modificaciones, debido a que se continúa con la eliminación de los valores del proceso a eliminar y, si hay más procesos con un *ID* mayor, se van desplazando para reducir el número. Esto lo vemos en el diagrama, observando el ciclo que está a la izquierda, y en código en la figura 3.76 y en la parte final de la figura 3.75.

Lo que cambia, como podemos notar en estas últimas imágenes, 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. Esta, además, ahora sólo recibe procesos de usuario para ser borrados, por lo que únicamente requiere obtener su información y continuar con la ejecución. Por esta

razón, es que se dejaron vacías cinco celdas que ya no se ocupan; se pueden observar en la figura 3.74. Se decidió no recorrer el programa para evitar la necesidad de editar el resto de fracciones. Así que, el borrado es incluso más eficiente para la versión paralela del sistema operativo.

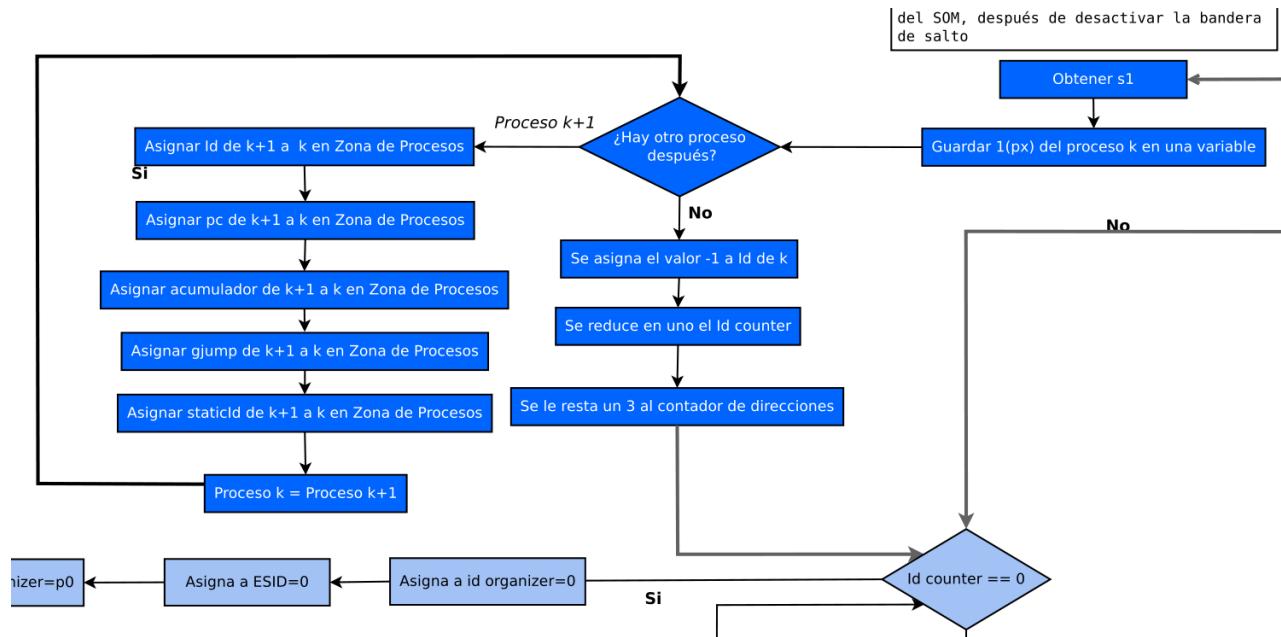


Figura 3.74: Acercamiento a la fracción de borrado

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

Figura 3.75: Sistema operativo mínimo borrar proceso

Sistema operativo				
Nombre clave	Dirección	Instrucción	Código	Descripción
<i>Recorrer las secciones gpc,gacc,gjump,staticid de py a px</i>	s37	1(c17)	LDA c17	Se obtiene 1(py) en el acumulador
	s38	2(c0)	ADD Cc0	Obtiene la dirección del gpc/gacc/gjump/staticid del proceso py
	s39	6(s44)	STO s44	Se guarda en s44 para tener obtener el valor de la sección del proceso
	s40	7c11	SUB C11	Obtiene la dirección correspondiente del gpc(u otra sección) del proceso px
	s41	4022	SHT 11	Se convierte 1(px)+u en 0(px)+u
	s42	2(c9)	ADD c9	Se convierte 0(px)+u en 6(px)+u
	s43	6(s45)	STO s45	Se guarda en s45 para que sea cargada la sección en px
<i>Aumentar contador de secciones</i>	s44	1(py)+u	LDA py+u	
	s45	6(px)+u	STO px+u	
	s46	1(c0)	LDA c0	Obtener contador de contextos
	s47	7000	ADD 000	Se le resta uno al contador de contextos
	s48	6(c0)	STO c0	Lo guarda en c0 de nuevo
<i>Regresar a recorrer</i>	s49	8(s35)	JMP s35	
<i>¿Hay proceso después de (k+1)?</i> <i>"Si" de s30</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>Borrado</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>M[c4]--</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[c5]=M[c5]-5</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

El flujo en general sería el siguiente: el usuario añade un proceso con ayuda de la primera CPU desde el modo de tarjetas. Una vez cargado, el *vigilante* permitirá avanzar al contador de programa de la CPU ejecutora para que añada el programa a la lista de procesos a ejecutar y, tal como sucedía en C, si ya hay un proceso agregado a la zona de procesos, el lanzador de procesos puede empezar a ejecutarlo siguiendo el flujo.

Después de la zona de lanzamiento de procesos, cuando el proceso ya se ha lanzado y ha terminado los ciclos que le corresponden, el flujo continúa hacia la zona de actualización del proceso, y regresa a la zona de lanzamiento si aún existen más procesos por lanzar. En caso de que el proceso haya culminado su ejecución, el flujo continúa hacia el borrado del proceso. El ciclo culmina con el *vigilante*, una vez que todos los procesos han terminado y la segunda CPU regrese a un estado de espera. Cabe destacar que, mientras la ejecutora está ejecutando procesos, la CPU cargadora puede seguir añadiendo procesos sin necesidad de esperar a que alguno complete su ejecución.

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

Añadiremos dos procesos para observar el funcionamiento de la máquina. El primero será un proceso para imprimir en reversa los siete números iniciales de la serie de Fibonacci, donde los números serán proporcionados por el usuario. El segundo será el *pintor*, el cual ya conocemos, y que imprimirá los primeros 10 dígitos. De esta forma, podremos ver cómo, mientras se ejecuta un programa, se puede añadir otro.

Para comenzar, añadimos un programa tal como se hacía en C. En la imagen 3.77, observamos que hay unas instrucciones en la cola de carga y que la CPU principal está realizando operaciones para cargarlas, mientras que la segunda CPU permanece en espera; se está cargando el programa de Fibonacci.

En la figura 3.78, observamos que el programa está cargado desde la dirección #204 hasta la #239, y una subrutina que utiliza el programa la podemos visualizar en la imagen 3.79. Además, vemos que la segunda CPU ya está en funcionamiento (fig. 3.78). De hecho, está en la zona de lanzamiento de procesos, puesto que se encuentra en la dirección #907.

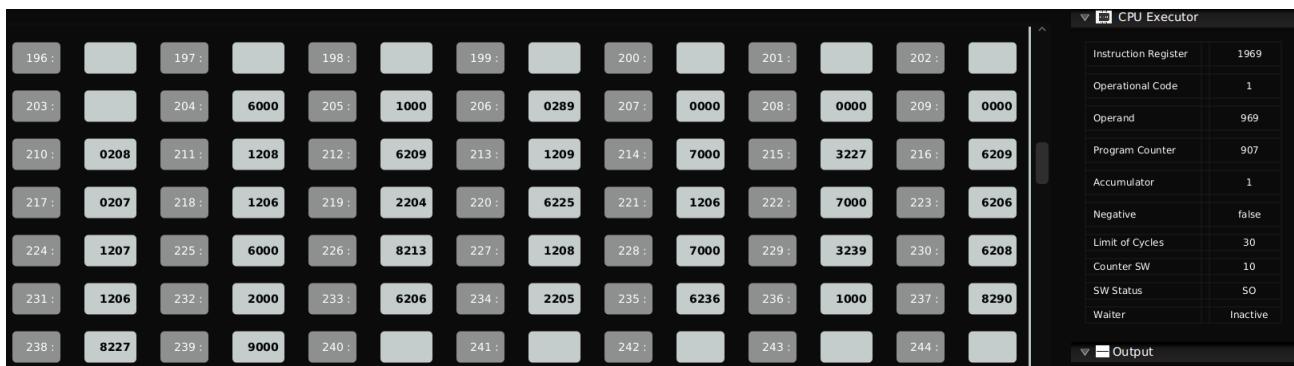


Figura 3.78: Proceso de Fibonacci cargado

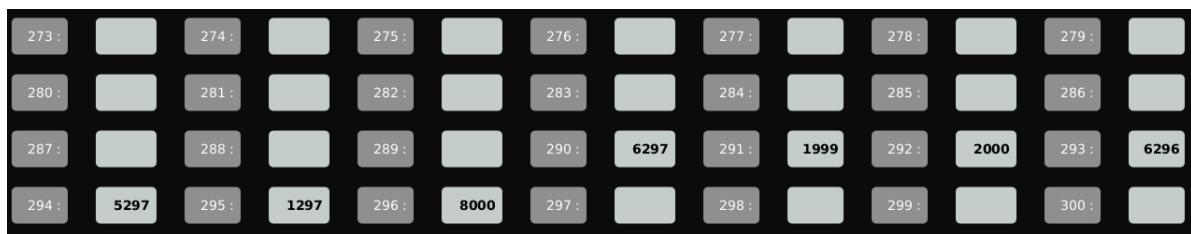


Figura 3.79: Subrutina de proceso de Fibonacci

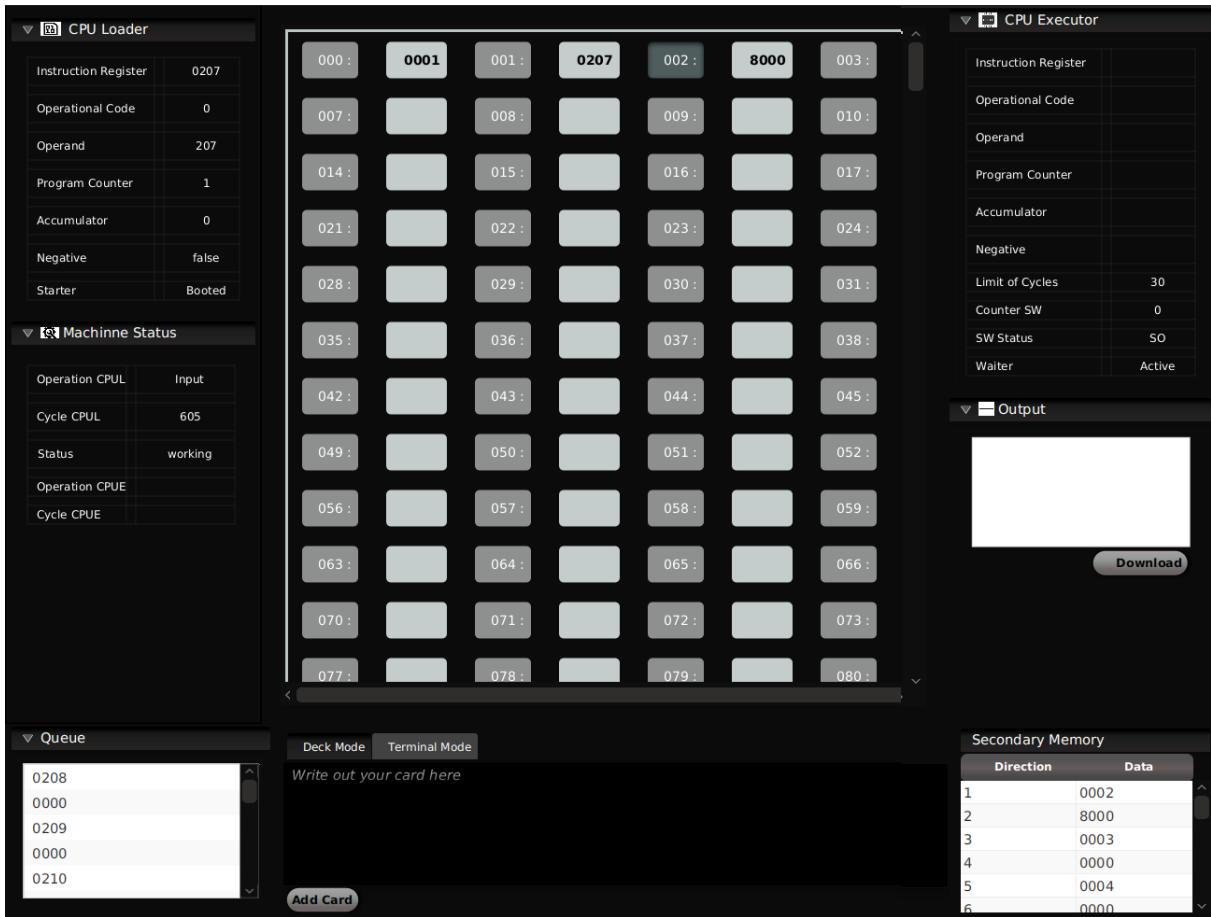


Figura 3.77: Añadiendo un nuevo proceso en E-CARDIAC PC

El siguiente paso es añadir el otro proceso. En la imagen 3.80, observamos en el *deck* la tarjeta lista para ser añadida a la cola. También, podemos ver que la segunda CPU ya ha lanzado el primer proceso que agregamos y está apuntando a la dirección #210, esperando datos del usuario. Como podrán notar, el color que resalta la dirección en la que se encuentra el contador de programa de la ejecutora no es verde, sino naranja, lo cual facilita diferenciar qué CPU está usando determinada zona de la memoria.

En la imagen 3.81, observamos que el segundo programa ya está en la cola de carga y que la CPU cargadora lo está añadiendo. Mientras, desde el modo terminal se está añadiendo la cantidad de números que el proceso de Fibonacci imprimirá. De esta forma, ya estamos avanzando en la ejecución del proceso de Fibonacci y, al mismo tiempo, se está añadiendo un nuevo proceso. Podemos notar claramente la potencia que nos ofrece el paralelismo en contraste al modelo anterior.

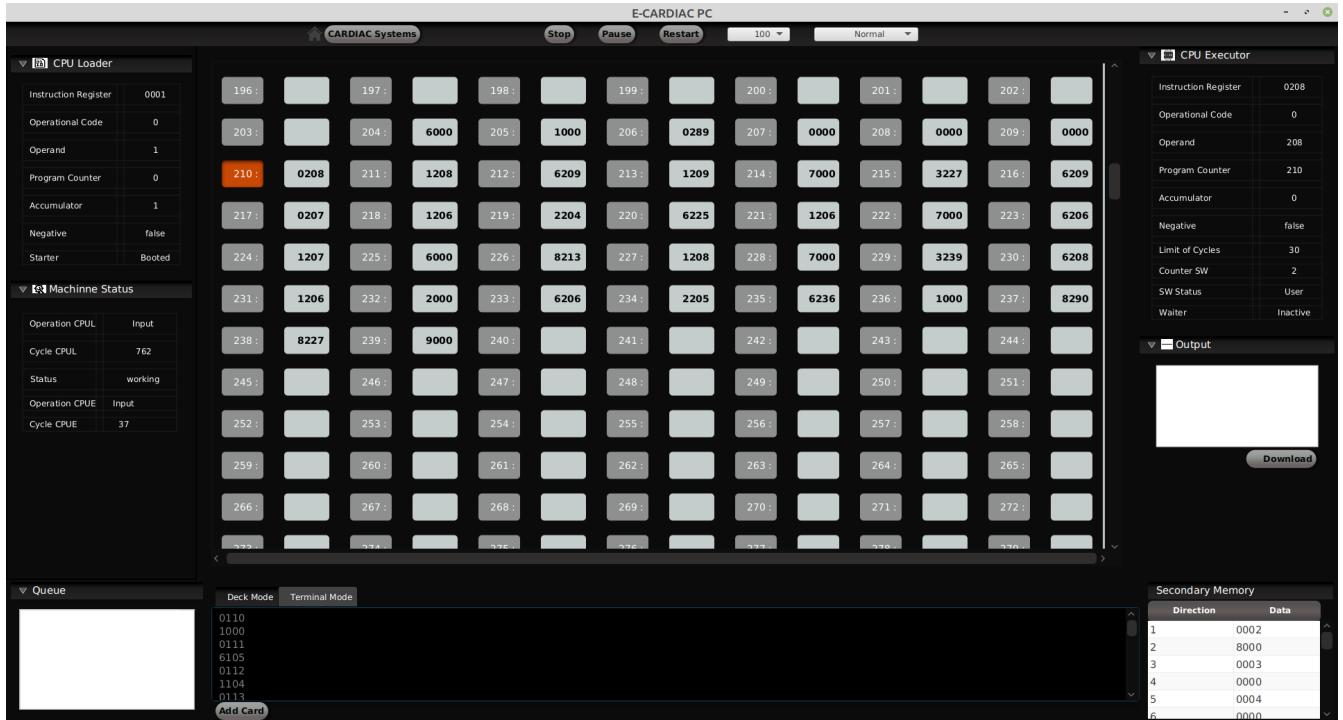


Figura 3.80: Programa pintor en el *deck*

En la imagen 3.82 observamos que ya se ha impreso parte de la salida del segundo proceso, del *pintor*, y que la ejecutora está actualizando uno de los procesos, ya que se encuentra en la dirección #817. En la siguiente imagen (3.83), vemos que la ejecución ha terminado. En ella, podemos apreciar que el *proceso 1* fue el último en terminar, lo cual es razonable, dado que es un proceso más complejo e incluso necesita de interacción con el usuario para cargar los números. Para ese momento, el vigilante (*waiter*) está activado de nuevo, porque la CPU ejecutora al no detectar más procesos, se encuentra en estado de espera. Mientras tanto, la CPU principal está lista para continuar agregando procesos.

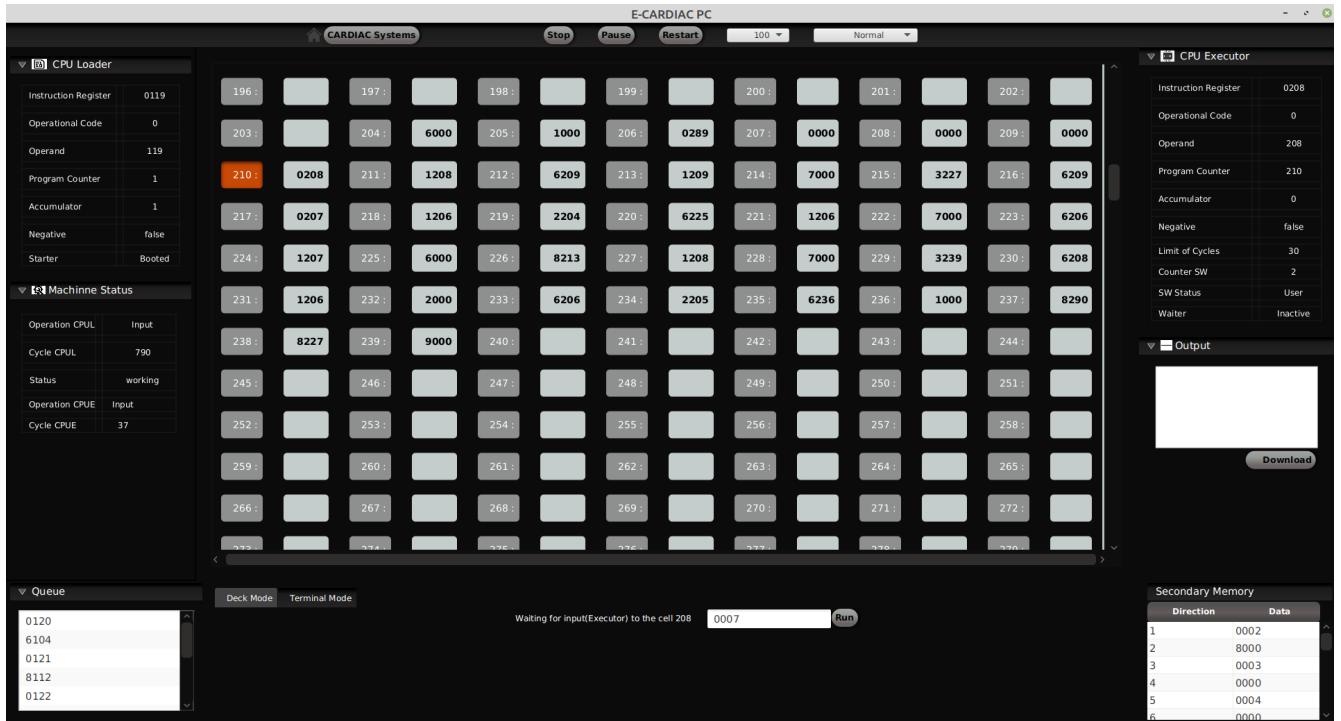


Figura 3.81: Agregando cantidad de números para serie de Fibonacci

Para visualizar mejor la salida, podemos ir a la carpeta de descargas, donde se encuentra el archivo de texto con la información de salida, como se puede ver en la imagen 3.84. En este archivo, podemos ver que primero se imprimió todo el *proceso 2*, antes de empezar a imprimir información del *proceso 1*, ya que el proceso de Fibonacci requería mucha interacción con el usuario. De esta manera, logró completar ambos sin errores y con una eficiencia notable en comparación con su antecesor: *E-CARDIAC C*.

En los anexos se pueden encontrar diversos códigos para *E-CARDIAC C* y *E-CARDIAC PC*, con los cuales podrán probar las máquinas virtuales directamente y comenzar a desarrollar sus propios códigos en lenguaje de *CARDIAC*.

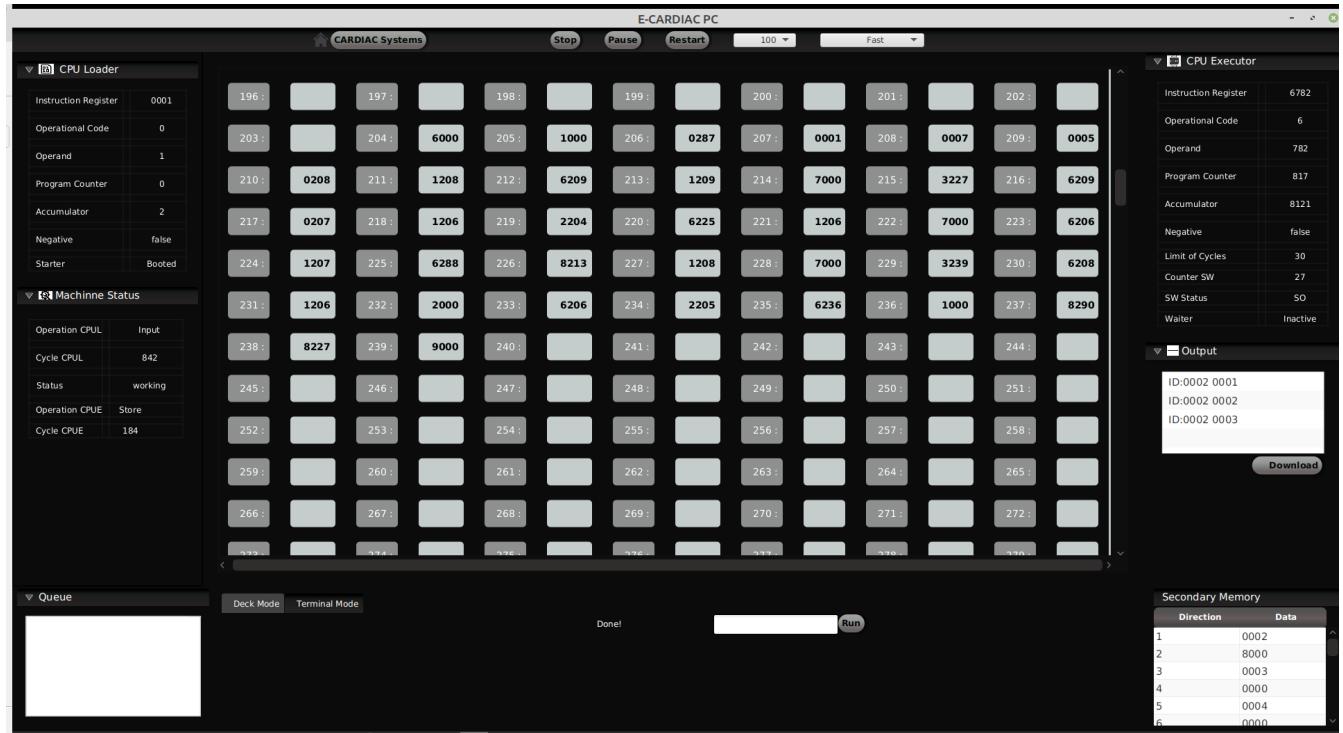


Figura 3.82: Primeras salidas de “pintor”

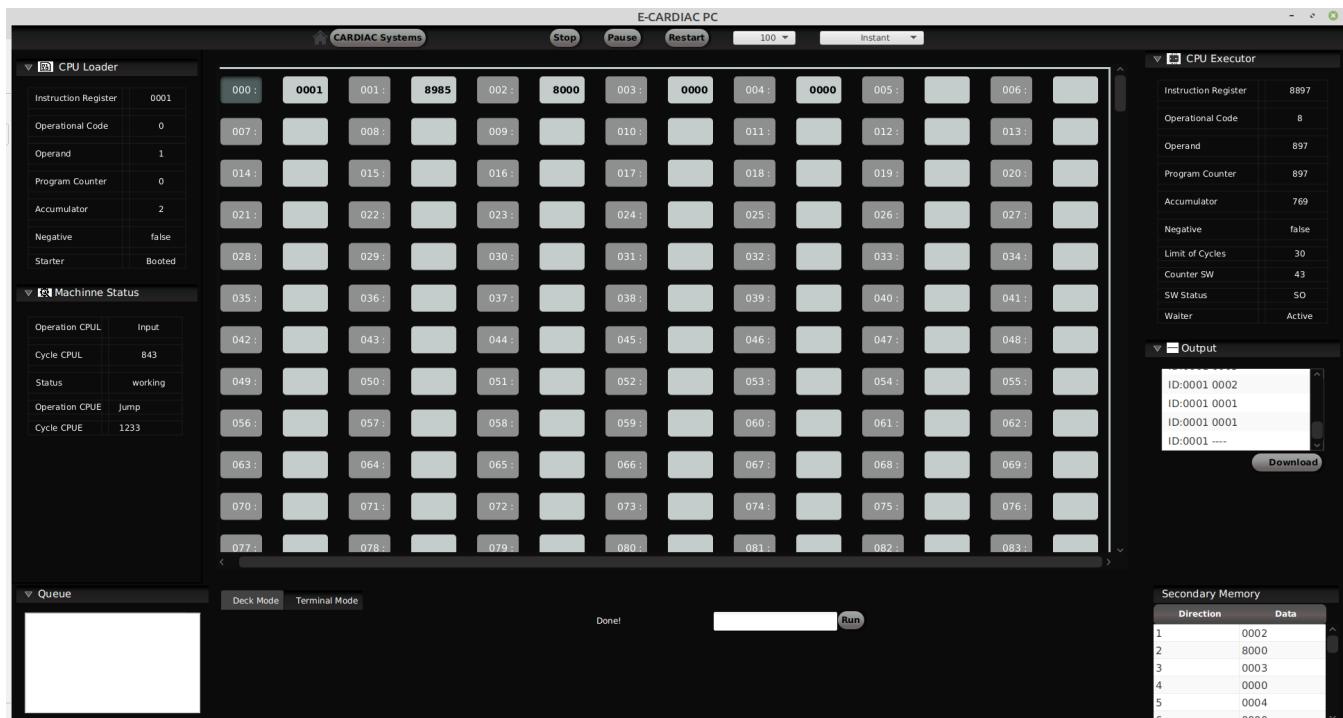


Figura 3.83: Ejecución finalizada

```
 CARDIAC_Output_2024...-14 13:36:38.949.txt ×
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. Sistema Operativo Mínimo PC: Guía rápida de uso

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

1. Diseñar un programa de acuerdo con la arquitectura de la máquina utilizando 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 qué lugar de la memoria cargar cada instrucción, de forma que puedas tener una “tarjeta” con instrucciones pareadas, donde 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. *0800* indica que se cargue la segunda en la primera dirección del SOMPC, en la implementación de 1000 celdas.
5. Para finalizar, agregar la instrucción preestablecida, que es *8985* en esta implementación, la cual es 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 SOMPC.
6. Esperar a que se cargue el programa y que el proceso 0 recupere el control para que el usuario pueda seguir 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 del o de los procesos, mientras el usuario puede seguir añadiendo más.

8. Si uno de los *procesos del usuario* requiere interacción con el usuario, se deberá usar el modo terminal para añadir los datos que el proceso requiera, ya que el modo *deck* es sólo 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*, donde cada salida indicará a qué proceso pertenece, identificándolo con su identificador estático. Si un proceso finaliza, se imprimirá el identificador seguido de varios guiones medios.
10. Una vez que se ha completado la ejecución de todos los procesos, el vigilante (*waiter*) se reactiva para poner a la segunda CPU en espera.
11. La CPU principal en paralelo puede seguir añadiendo procesos.

Con la guía, podemos notar que, salvo algunos puntos, lo demás sigue la misma estructura, porque finalmente es una evolución del modelo concurrente que ahora utiliza dos procesadores para lograr que 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**, no sólo hemos aprendido el funcionamiento básico de la computadora viendo cómo interactúan sus componentes, sino que nos ha permitido adentrarnos en otros aspectos relevantes de las computadoras modernas como 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 basada en una arquitectura von Neumann. El traslado del papel al software fue de vital importancia para lograr esto, ya que cuando se aumenta el número de celdas de memoria de 100 a 1000 es muy difícil de seguir el proceso sólo con papel. Contar con una máquina virtual que haga los cálculos y movimientos del apuntador por ti, permite que te concentres más en las interacciones entre los diversos componentes de la máquina y cómo reaccionan a cada instrucción leída.

La dificultad de presentar un modelo concurrente y uno paralelo fue bien subsanada al utilizar *CARDIAC* como base. La intención fue ser los más minimalistas posible en los modelos construidos, para que fueran 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, limitando el número de conceptos y elementos con los que lidiar. Además de que al analizar estos modelos, surge orgánicamente la necesidad del sistema operativo, por lo que con este par de modelos el estudiante puede ver por qué es tan importante el sistema operativo y, como sin él, construir un modelo de cómputo con capacidades concurrentes o paralelas sería una tarea demasiado complicada, y

el resultado, quizás, demasiado complejo.

Para 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, como algunos de la bibliografía. Asimismo, las máquinas virtuales diseñadas, le pueden servir para explorar más situaciones o problemas que no se plantearon en este texto. Tienen a su disposición un modelo concurrente, y uno paralelo y concurrente, de los cuales 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] L. Manabrea, "Scientific Memoirs, Selected from the Transactions of Foreign Academies of Science and Learned Societies, and from Foreign Journals," en, R. Taylor, ed., pág. 666, 1843. dirección: <http://people.csail.mit.edu/brooks/idocs/Lovelace.pdf>.
- [11] 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).
- [12] 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).
- [13] H. H. Goldstine, *The computer from Pascal to von Neumann*. Princeton, N.J.: Princeton University Press, 1972, ISBN: 978-0-691-08104-5.
- [14] 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).
- [15] Computer History Museum, *Computers / Timeline of Computer History*. dirección: <https://www.computerhistory.org/timeline/computers/#169ebbe2ad45559efbc6eb3572083fb> (visitado 30-09-2023).
- [16] 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).

- [17] 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](https://doi.org/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).
- [18] A. S. Tanenbaum, *Modern operating systems*, English. 2002, OCLC: 981051666, ISBN: 978-0-13-031358-4 978-7-111-09156-1.
- [19] 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.
- [20] 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.
- [21] Maurice Vincent Wilkes, *EDSAC 1951*, 1976. dirección: <https://www.youtube.com/watch?v=6v4Juzn10gM> (visitado 14-09-2023).
- [22] M. Richards, “EDSAC Initial Orders and Squares Program,” en, *University of Cambridge*, dirección: <https://www.cl.cam.ac.uk/~mr10/Edsac/edsacposter.pdf>.
- [23] 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.
- [24] 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](https://doi.org/10.1145/982753.982754). dirección: <https://doi.org/10.1145/982753.982754> (visitado 09-05-2023).
- [25] R. M. Hord, *The Illiac IV: the First Supercomputer*, eng. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, OCLC: 851370561, ISBN: 978-3-662-10345-6.

- [26] 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.
- [27] 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.
- [28] 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).
- [29] ——, *AT&T Archives: The Thinking Machines (Bonus Edition)*, mar. de 2012. dirección: <https://www.youtube.com/watch?v=clud9I18DXU> (visitado 17-08-2023).
- [30] megardi, *CARDIAC (CARDboard Illustrative Aid to Computation) Replica*, en. dirección: <https://www.instructables.com/CARDIAC-CARDboard-Illustrative-Aid-to-Computation-/> (visitado 17-08-2023).
- [31] 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.
- [32] 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.
- [33] 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.
- [34] 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.

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

# Apéndices

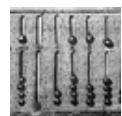
## Apéndice A

### Historia de la computación

# Historia de la computación

2700 BCE

- Creación del ábaco



Se crea una especie de ábaco en Sumeria, acorde a la numeración que utilizaban.

1300 BCE

- Ábaco en China

En este periodo se encuentran las referencias más antiguas al uso de una especie de ábaco en China.

1614



- Logaritmos

John Napier descubre los **logaritmos** y, poco después, crea tablas para recordarlos de manera más eficiente.

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 aún se utiliza en la

ingeniería actual para ciertos cálculos.

1642

### Pascaline



Blaise Pascal inventa la **primera** máquina de cálculo aritmético basada en ruedas y engranajes.

1673

### Stepped Reckoner



Gottfried Leibniz construye una calculadora mecánica llamada "Máquina de Leibniz" o Stepped Reckoner, que podía realizar las cuatro operaciones básicas de manera automática.

1801

### Telar de Jacquard



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

1820

### Arithmomètre



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

1830

### Difference Engine



Charles Babbage presenta un prototipo de lo que sería su máquina diferencial.

1841

### Analytical Engine



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

1847

## Álgebra de Boole

George Boole publica *The Mathematical Analysis of Logic* y años después *An Investigation of the Laws of Thought*. Los cuales definen lo que hoy conocemos como **Álgebra de Boole**.

1890



### Tabulating Machine

Herman Hollerith desarrolla una máquina con tarjetas perforadas para procesar la información de los censos en EE.UU.

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 Harold Locke Hazen y Vannevar Bush en el MIT, y era capaz de resolver ecuaciones diferenciales, utilizada especialmente en problemas de ingeniería y física.

1936



### Formalización del computo

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

1939



### Complex Number Calculator

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

1941

### Z3 Computer



Konrad Zuse termina la Z3, primera 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 primera computadora creada en EE.UU.

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 máquina 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 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 reemplazaba 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, y 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 al 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



Primera computadora comercial creada en EE.UU., 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 inició la cultura "hacker" en el MIT.

1961

### IBM 7030 Stretch



Fue el primer ordenador de IBM que usaba transistores. Originalmente, tenía un precio de 13.5 millones de dólares.

1964

### IBM System/360



Es una familia de computadores que se empezó a liberar en 1964 y que tenía como característica principal el compartir un mismo sistema operativo.

1965

### GE 645



AT&T y General Electric desarrollaron 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

Minicomputadora creada por Digital Equipment Corporation, fue la primera minicomputadora comercialmente exitosa y tenía un costo inicial de 18,000 dólares.



1972

## ILLIAC IV

Se pone en marcha la "primera" supercomputadora, que contaba con 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 buscaba ser más "personales".



1975

## Altair 8080

Primera computadora que puede llamarse "personal" por lo pequeña que es y que ha sido 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, dio otro avance gigante en las computadoras personales.



1981

## IBM PC

La primera computadora personal desarrollada por IBM, formalmente llamada IBM 5150, es 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 con el mercado: 16 GB de RAM, 512 GB de almacenamiento interno y con el sistema operativo Windows 11 incluido.

## Apéndice B

# Sistema Operativo Concurrente para E-CARDIAC

Sistema operativo				
Nombre clave	Dirección	Instrucción LM	Instrucción LE	Descripción
<i>Nuevo</i>	<i>s0</i>	-1		-1 Es su valor por defecto, el usuario siempre agregará una 8-dir
<i>Actual</i>	<i>s1</i>	6000		Contiene la 6-dir del id del proceso actual, controlado por el lanzador, 6(px)
<i>Validación</i>	<i>s2</i>	<b>1(s1)</b>	<b>LDA s1</b>	El acumulador toma un valor de la forma 6(px)
	<i>s3</i>	<b>3(s98)</b>	<b>BLZ s97</b>	Si acc<0 no hay proceso para actualizar
	<i>s4</i>	<b>2000</b>	<b>ADD 000</b>	Se obtiene la 6-dir del gpc del proceso
	<i>s5</i>	<b>6(s7)</b>	<b>STO s7</b>	Se guarda la instrucción 6(px)+1
	<i>s6</i>	<b>1(c1)</b>	<b>LDA C1</b>	Se obtiene el último pc del proceso con forma 8(pc)
	<i>s7</i>	<b>6(p5)</b>	<b>STO p5</b>	En p5 se actualiza gpc
	<i>s8</i>	<b>1(s7)</b>	<b>LDA s7</b>	Se obtiene la instrucción 6(px)+1
	<i>s9</i>	<b>2000</b>	<b>ADD 000</b>	Para acceder a la 6-dir del gacc del proceso
<i>Actualizar gpc</i>	<i>s10</i>	<b>6(s12)</b>	<b>STO s12</b>	En s12 se guarda 6(px)+2
	<i>s11</i>	<b>1(c2)</b>	<b>LDA c2</b>	Se obtiene el último acc del proceso
	<i>s12</i>	<b>6(p6)</b>	<b>STO p6</b>	Se actualiza el valor de gacc
	<i>s13</i>	<b>1(s12)</b>	<b>LDA s12</b>	Obtiene la 6(p6) del proceso que se está actualizando
	<i>s14</i>	<b>2000</b>	<b>ADD 000</b>	Obtiene la 6(p7) del proceso que se está ejecutando
	<i>s15</i>	<b>6(s17)</b>	<b>STO s17</b>	Guarda en e18 el código para guardar en la zona de procesos c14
	<i>s16</i>	<b>1(c14)</b>	<b>LDA c14</b>	Obtiene c14
	<i>s17</i>	<b>6(p7)</b>	<b>STO p7</b>	Guarda en la zona de procesos correspondiente al proceso el valor saber jump
<i>Actualizar gjump</i>	<i>s18</i>	<b>8(s101)</b>	<b>JMP S100</b>	Saltamos a cambiar de proceso
<i>Saltar</i>	<i>s19</i>	<b>1(s1)</b>	<b>LDA s1</b>	Llamemos k al proceso asociado a la dir px
	<i>s20</i>	<b>4011</b>	<b>SHT 11</b>	Se convierte 6(px) en 0(px)
	<i>s21</i>	<b>2(c8)</b>	<b>ADD c8</b>	Se convierte 0(px) en 1(px)
	<i>s22</i>	<b>6(c0)</b>	<b>STO c0</b>	Guarda en c0 1(px)
	<i>s23</i>	<b>1e7</b>	<b>LDA c7</b>	Se obtiene el id del proceso que se estaba ejecutando
	<i>s24</i>	<b>7(000)</b>	<b>SUB 000</b>	Se le resta 1 al id, acc=-1
	<i>s25</i>	<b>3(s98)</b>	<b>BLZ s</b>	Si acc<0 es el proceso 0 y se bloquea, no se borra
	<i>s26</i>	<b>1(c0)</b>	<b>LDA c0</b>	Se obtiene 1(px)
<i>Guardar l(px)</i>	<i>s27</i>	<b>2(c11)</b>	<b>ADD c11</b>	Se obtiene 1(px)+5, la 1-dir del id del proceso siguiente (k+1)
	<i>s28</i>	<b>6(s29)</b>	<b>STO s29</b>	En s24 se guarda 1(px)+5, al que llamamos l(py)
	<i>s29</i>	<b>1(py)</b>	<b>LDA py</b>	Se obtiene el id del proceso (k+1)
	<i>s30</i>	<b>3(s52)</b>	<b>BLZ s52</b>	Si acc<0 no hay otro proceso adelante
	<i>s31</i>	<b>1(s29)</b>	<b>LDA s29</b>	Se obtiene 1(py) en el acumulador
	<i>s32</i>	<b>6(c17)</b>	<b>STO c17</b>	Se guarda en c17 para usarla más tarde
	<i>s33</i>	<b>1(c18)</b>	<b>LDA 004</b>	Se carga 0004 para funcionar como contador
	<i>s34</i>	<b>6(c0)</b>	<b>STO C0</b>	Se guarda el contador de secciones del proceso
<i>Verificar si faltan secciones</i>	<i>s35</i>	<b>7000</b>	<b>SUB C11</b>	Se verifica si ya terminó con las secciones
	<i>s36</i>	<b>3(s50)</b>	<b>BLZ s50</b>	Si ya se pasaron las 4 secciones salta a ver si hay mas procesos
	<i>s37</i>	<b>1(c17)</b>	<b>LDA c17</b>	Se obtiene 1(py) en el acumulador
	<i>s38</i>	<b>2(c0)</b>	<b>ADD Cc0</b>	Obtiene la dirección del gpc/gacc/gjump/staticid del proceso py
	<i>s39</i>	<b>6(s44)</b>	<b>STO s44</b>	Se guarda en s44 para tener obtener el valor de la sección del proceso
	<i>s40</i>	<b>7e11</b>	<b>SUB C11</b>	Obtiene la dirección correspondiente del gpc(u otra sección) del proceso px
	<i>s41</i>	<b>4011</b>	<b>SHT 11</b>	Se convierte 1(px)+u en 0(px)+u
	<i>s42</i>	<b>2(c9)</b>	<b>ADD c9</b>	Se convierte 0(px)+u en 6(px)+u
<i>Recorrer las secciones gpc,gacc,gjump,staticid de py a px</i>	<i>s43</i>	<b>6(s45)</b>	<b>STO s45</b>	Se guarda en s45 para que sea cargada la sección en px
	<i>s44</i>	<b>1(py)+u</b>	<b>LDA py+u</b>	
	<i>s45</i>	<b>6(px)+u</b>	<b>STO px+u</b>	
	<i>s46</i>	<b>1(c0)</b>	<b>LDA c0</b>	Obtener contador de contextos
	<i>s47</i>	<b>7000</b>	<b>ADD 000</b>	Se le resta uno al contador de contextos
	<i>s48</i>	<b>6(c0)</b>	<b>STO c0</b>	Lo guarda en c0 de nuevo
	<i>s49</i>	<b>8(s35)</b>	<b>JMP s35</b>	
	<i>s50</i>	<b>1(s29)</b>	<b>LDA S29</b>	Se obtiene 1(py) en el acumulador
<i>¿Hay proceso después de (k+1)?</i>	<i>s51</i>	<b>8(s27)</b>	<b>JMP s27</b>	Va a verificar si hay otro proceso adelante
	<i>"Si" de s30</i>	<b>s52</b>	<b>1(s29)</b>	LDA s29 Se carga la 1-dir del id del proceso k+1

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)
Guardar Nuevo Static ID para el proceso	s81	6(s85)	STO s85	Guardar en s85
	s82	1(c15)	LDA c15	Obtener Serial de Static ID
	s83	2000	ADD 000	Añadir una unidad
	s84	6(c15)	STO C15	Guardar en c15 el folio actualizado
Id del nuevo 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

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 s110	Regresar para lanzar el proceso
Reinicidar 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
	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

#### 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

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>Inicial</i>	c3	p5		Es la dir de inicio de la zona de procesos(usuario)
<i>Id counter</i>	c4	0		El contador de procesos de usuario
<i>Dir counter</i>	c5	p0		El contador de direcciones
<i>Dir organizer</i>	c6	p0		Contiene 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

## Apéndice C

# 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>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

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

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 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 dir organizer=p0	s139	6(c7)	STO c7	Se reinicia el id organizer
	s140	8(s110)	JMP s109	Regresar para lanzar el proceso
Reinic平 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
Obtener Saver Jump	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

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
	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 esta 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>Inicial</i>	c3	p5		Es la dir de inicio de la zona de procesos(usuario)
<i>Id counter</i>	c4	0		El contador de procesos de usuario
<i>Dir counter</i>	c5	p0		El contador de direcciones
<i>Dir organizer</i>	c6	p0		Contiene 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

## Apéndice D

### 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

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

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	-----
810	8985
007	Entrada
001	0007
001	0001
001	0001
002	0002
002	0003
003	0005
005	0008
008	0013
013	

Potencias de 2	
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

Versión CARDIAC		Versión CARDIAC C y PC
199		2000
092		0493
200		6496
093		0494
696		1497
094		0495
197		2497
095		0800
297	----	8410
002	-----	8985
810		

## Apéndice E

### Tarjetas para cargar el 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>

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>

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>

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>

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>

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>

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>

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>

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		

## Apéndice F

### Guía de acceso al repositorio de GitHub

# Máquina Virtual CARDIAC: Un modelo didáctico para la computación

Esta es una máquina virtual diseñada para explorar la computación concurrente y paralela utilizando el reconocido modelo CARDIAC, creado por David Hagelbarger y Saul Fingerman.

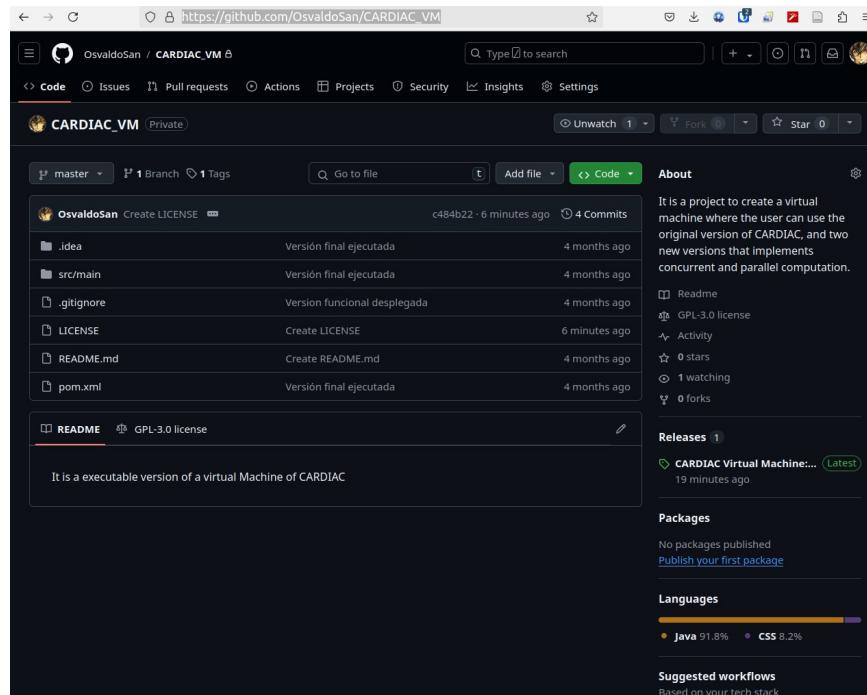
Esta aplicación fue creada para la tesis: **CARDIAC: La evolución hacia un modelo concurrente y paralelo**, y está destinada a asistir a cualquier estudiante que la necesite.

## ¿Cómo Usar Esta Aplicación?

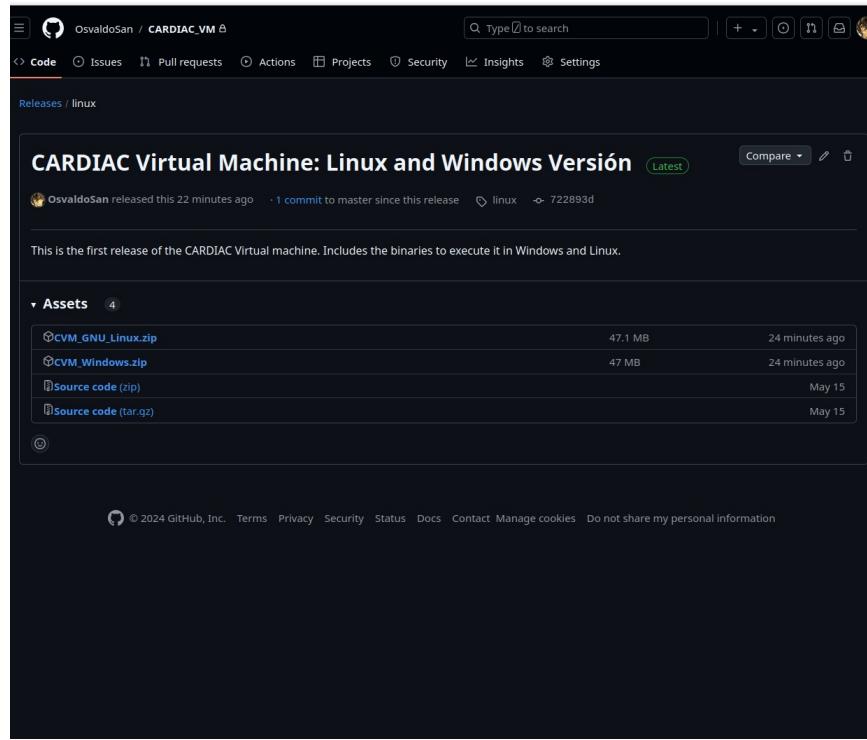
### Opción Binaria

Para usar esta versión en Windows o Linux, necesitas instalar la versión más reciente del Entorno de Ejecución de Java. Puedes encontrarla en [Java](#).

Para buscar los binarios ve al [repositorio](#) y haz clic en la sección *releases*.



Una vez que hayas hecho clic allí, verás la siguiente imagen:



Aquí podrás ver dos archivos zip que contienen el código binario para ejecutar el programa en Windows y GNU/Linux. Descarga la versión adecuada para tu sistema operativo.

### Opción Binaria: Linux

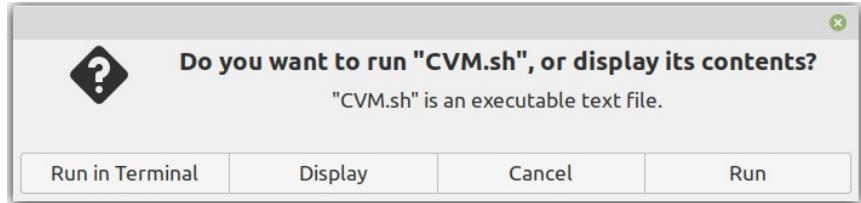
Si descargas la versión para Linux, necesitas descomprimir el archivo, y verás las siguientes carpetas:



Estas carpetas contienen toda la información necesaria para que el programa funcione. Luego, ve a la carpeta llamada *bin*, donde verás los siguientes archivos:



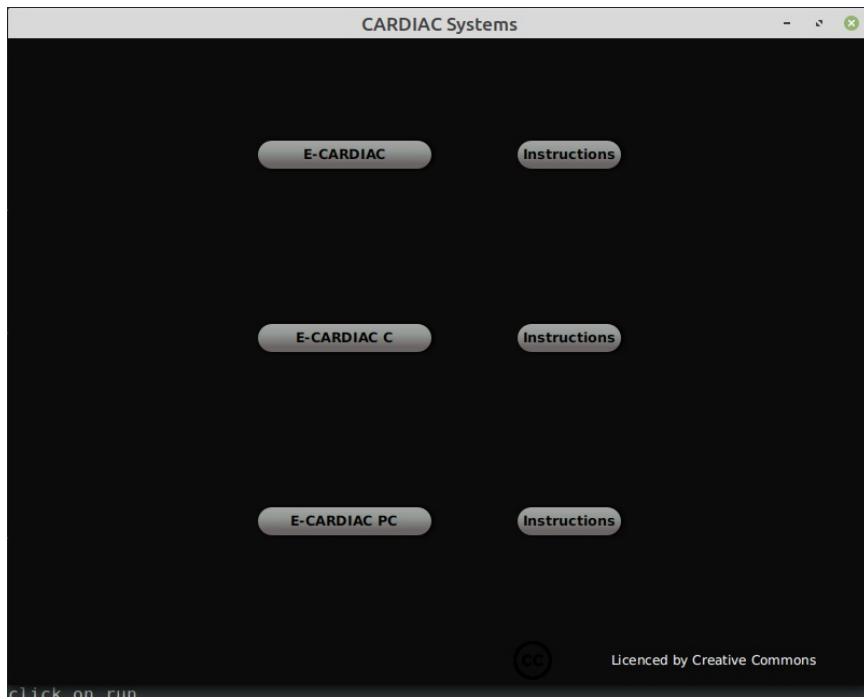
A continuación, necesitas ejecutar el archivo llamado **CVM.sh**. Puedes ejecutarlo desde el explorador de archivos haciendo doble clic en él, lo que mostrará el siguiente diálogo. Aquí, haz clic en "RUN".



Si encuentras algún problema con esto, otra opción es ejecutarlo desde la línea de comandos del terminal con el siguiente comando:

```
mrblue@mr bin ~ ls  
CVM.sh  java  jrunscript  keytool  
mrblue@mr bin ~ ./CVM.sh  
Sep 14, 2024 10:19:55 PM javafx.fxml.FXMLLoader$ValueElement p  
rocessValue  
WARNING: Loading FXML document with JavaFX API of version 19 b  
y JavaFX runtime of version 11.0.2
```

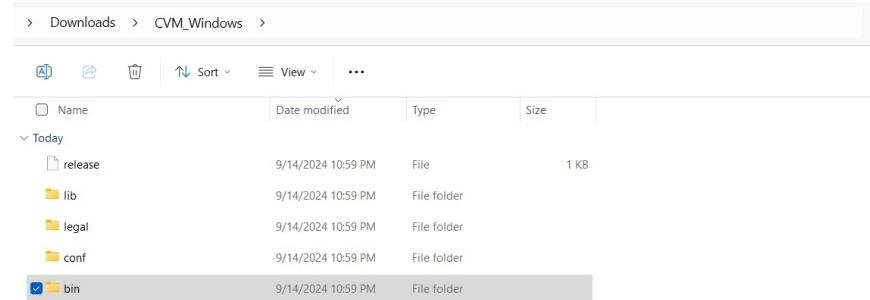
Después de eso, verás que el programa se inicia:



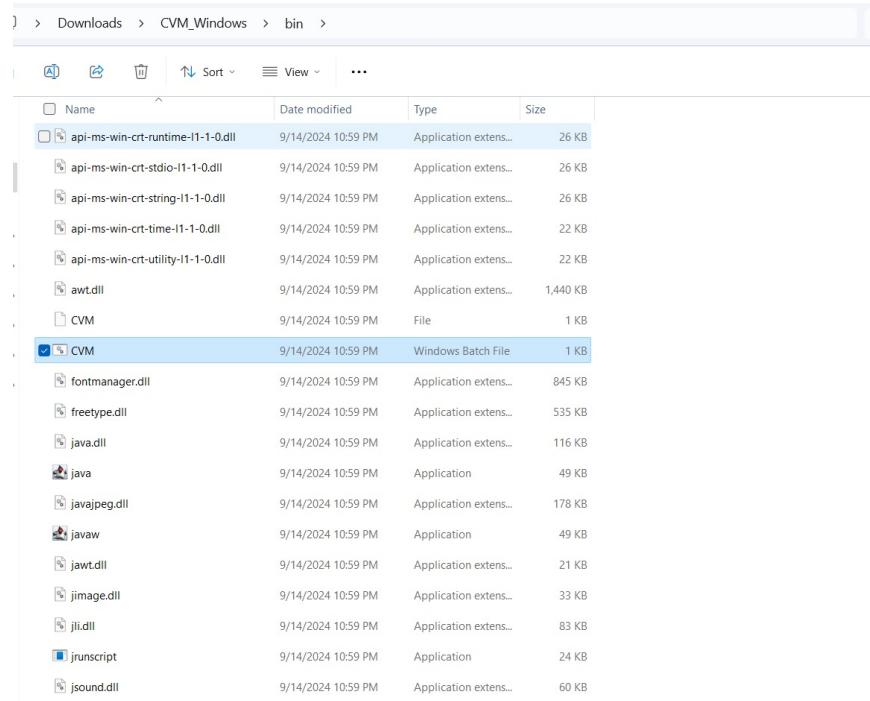
Las instrucciones sobre cómo usar la máquina virtual se encuentran en el PDF incluido en el repositorio. Este PDF hace referencia a la tesis para este proyecto.

### Opción Binaria: Windows

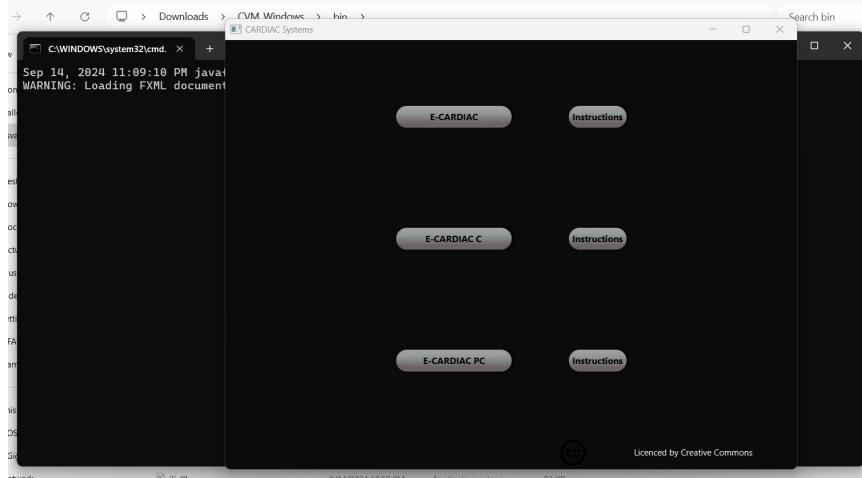
Si usas la opción para Windows, después de descargar y extraer el archivo **CVM\_Windows.zip**, verás las siguientes carpetas:



Estas carpetas contienen toda la información necesaria para que la aplicación funcione correctamente. Para ejecutar la aplicación, ve a la carpeta *bin*, donde verás varios archivos. Busca el archivo llamado *CVM*, que es un archivo tipo *Windows Batch File* como se muestra en la imagen:



Una vez que hayas identificado el archivo, simplemente haz doble clic en él, y se abrirá una ventana de *CMD* que iniciará la ventana principal de *CARDIAC*.



Las instrucciones sobre cómo usar la máquina virtual se encuentran en el PDF incluido en el repositorio. Este PDF hace referencia a la tesis para este proyecto.

## Crear un Ejecutable con el Código Fuente

Si prefieres crear un ejecutable a partir del código fuente, puedes hacerlo. Primero, accede al repositorio [CARDIAC VM](#) y clona el código.

Luego, simplemente sigue las reglas del framework [Apache Maven](#) y edita el archivo *pom* según tus especificaciones.

Aquí tienes un ejemplo de lo fácil que es desplegar la aplicación usando Maven en el terminal de Linux:

```
[~/Desktop/CVM @ master ~] $ mvn clean compile javafx:jlink
[INFO] Scanning for projects...
[INFO]
[INFO] --->< com.vm:cvm >-----
[INFO] Building cvm 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ cvm ---
[INFO] Deleting /home/mrblue/Documentos/Proyectos/JFXProjects/cvm/target
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ cvm ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 29 resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ cvm ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 14 source files to /home/mrblue/Documentos/Proyectos/JFXProjects/cvm/target/classes
[INFO] /home/mrblue/Documentos/Proyectos/JFXProjects/cvm/src/main/java/com/vm/cvm/controller/CardiacSync_controller.java: /home/mrblue/Documentos/Proyectos/JFXProjects/cvm/src/main/java/com/vm/cvm/controller/CardiacSync_controller.java uses unchecked or unsafe operations.
[INFO] /home/mrblue/Documentos/Proyectos/JFXProjects/cvm/src/main/java/com/vm/cvm/controller/CardiacSync_controller.java: Recompile with -Xlint:unchecked for details.
[INFO]
[INFO] >>> javafx-maven-plugin:0.0.8:jlink (default-cli) > process-classes @ cvm >>>
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ cvm ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 29 resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ cvm ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 14 source files to /home/mrblue/Documentos/Proyectos/JFXProjects/cvm/target/classes
```

Una de las mejores cosas de usar Maven para desplegar la aplicación es que, dependiendo del sistema operativo que utilices, se creará el archivo binario correspondiente. Así que, si deseas un binario para Windows, necesitarás desplegar la aplicación en un sistema de archivos de Windows.

Para más información, visita [Apache Maven](#).