7-22-2016

Implementación de una CPU de una dirección en Logisim

Charles W. Kann
Universidad de GeDysburg

Implementación de una CPU de una dirección en Logisim

Descripción

La mayoría de los usuarios de ordenadores tienen una metáfora cognitiva incorrecta, pero útil, de los ordenadores en la que el usuario dice (o teclea o hace clic) algo y se produce un comportamiento místico, casi inteligente o mágico. No es exagerado describir a los usuarios de ordenadores como personas que creen que los ordenadores siguen las *leyes de la magia, en las que* se introduce algún *conjuro mágico* y el ordenador responde con un comportamiento esperado, pero mágico.

Este ordenador mágico no existe en realidad. En realidad, los ordenadores son máquinas, y cada acción que realiza un ordenador se reduce a un conjunto de operaciones mecánicas. De hecho, la primera definición completa de un ordenador en funcionamiento fue una máquina mecánica diseñada por Charles Babbage en 1834, que funcionaba a vapor.

Probablemente, el mayor éxito de las Ciencias de la Computación (CS) en el siglo ^{XX} fue el desarrollo de abstracciones que ocultan la naturaleza mecánica de los ordenadores. El hecho de que la gente corriente utilice los ordenadores sin plantearse que son mecánicos es un triunfo de los diseñadores de la informática.

El propósito de esta monografía es romper la comprensión abstracta de un ordenador y explicar su comportamiento en términos completamente mecanicistas. Tratará específicamente de la Unidad Central de Procesamiento (CPU) del ordenador, ya que es ahí donde se produce la magia. Todas las demás partes de un ordenador pueden considerarse como meras fuentes de información para la CPU.

Esta monografía tratará de un tipo específico de CPU, una CPU de una dirección, y explicará esta CPU usando sólo compuertas estándar, específicamente compuertas AND, OR, NOT, NAND y XOR, y 4 Circuitos Integrados (CIs) básicos, el Decodificador, Multiplexor, Sumador, y Flip Flop. Todas estas puertas y componentes pueden describirse como transformaciones mecánicas de datos de entrada en datos de salida, y la CPU en su conjunto puede considerarse un dispositivo mecánico.

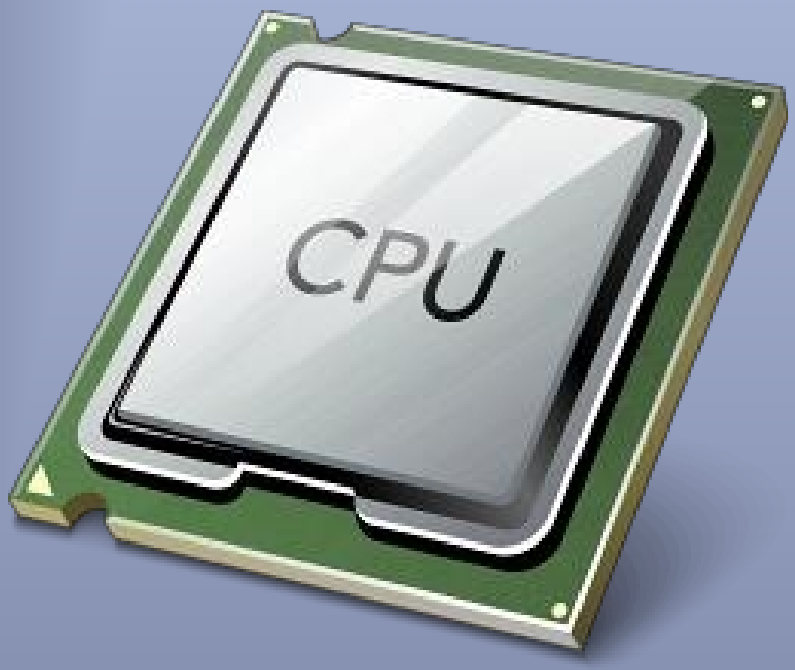
Palabras clave

Circuitos Digitales, Arquitectura de Sistemas, Organización de Computadoras, Circuitos Integrados, Lógica de Computadoras, Unidad Central de Procesamiento (CPU), Arquitectura de Procesadores, Multiplexor, Decodificador, Unidad Lógica Aritmética, Archivo de Registro, Flip-Flop, Memoria, Enclavamiento de Memoria, Sumador, Sumador Completo, Medio Sumador, Computadora de Estado, Máquina de Estado, Contador Mod 4, 7400, Serie 7400, Manual de Laboratorio de Circuitos Digitales, Circuitos Electrónicos, Proyectos Electrónicos, Proyectos de Circuitos Digitales, Informática Online, Manual de Laboratorio Online, Manual de Laboratorio

Comentarios

El archivo zip incluido con esta entrada debería tener todos los materiales para el texto, incluidos el ensamblador, los circuitos Logisim, los programas y las figuras. A medida que se desarrollen nuevos materiales, podrán estar disponibles como versiones beta en http://chuckkann.com.

Por favor, envíeme un correo electrónico a ckann@gettysburg.edu si utiliza este libro como libro de texto en una clase.



Implementación de una CPU unidireccional en Logisim

Vol. 1: Implementación de CPU en la serie Logisim

La monografía implementa una CPU sencilla de una dirección utilizando Logisim. Se crea y explica una CPU programable de una dirección, incluyendo el lenguaje ensamblador utilizado para la CPU, un ensamblador para traducir el lenguaje ensamblador a código máquina, y cómo la CPU utiliza el código máquina para implementar el programa.

© Charles W. Kann III
277 E. Lincoln Ave.
Gettysburg, Pa

Última actualización: domingo, 06 de noviembre de 2016

Un conjunto de problemas para el ordenador en este texto se está desarrollando, y si usted los desea envíeme un correo electrónico a ckann@gettysburg.edu

Este libro puede descargarse gratuitamente de: http://chuckkann.com/books/.

Otros libros de Charles Kann

Kann, Charles W., "Proyectos de circuitos digitales: Una visión general de los circuitos digitales a través de la implementación de circuitos integrados - Segunda edición" (2014). *Recursos educativos abiertos del Gettysburg College.* Libro 1. http://cupola.gettysburg.edu/oer/1

Kann, Charles W., "Introducción a la programación en lenguaje ensamblador MIPS" (2015). *Recursos educativos abiertos del Gettysburg College.* Book 2.
http://cupola.gettysburg.edu/oer/2

Adelante

El objetivo de las monografías de esta serie es ayudar a los estudiantes y demás interesados en la informática a comprender la naturaleza mecánica de la parte más importante de un ordenador, la CPU. Esta serie está pensada para estudiantes y profesionales de la informática que deseen profundizar en el diseño de una CPU, pero también para aficionados a la informática interesados en el funcionamiento real de los ordenadores.

Esta es la primera monografía de una serie prevista de documentos que describen e implementan diferentes arquitecturas de CPU. Esta monografía implementa una Unidad Central de Procesamiento (CPU) de una dirección que el autor ha implementado en Logisim. Excepto por algunos Circuitos Integrados (ICs) simples, específicamente un decodificador, multiplexor, sumador y flip flop, esta CPU está diseñada usando sólo compuertas simples (AND, OR, NOT, XOR y NAND) y lógica booleana simple, y por lo tanto debería estar a un nivel que pueda ser entendido por un aficionado que quiera profundizar en qué es una CPU, y qué puede hacer una CPU.

Esta CPU también puede utilizarse en clases de Organización de Computadores o Arquitectura de Computadores. Un libro de trabajo previsto para esta CPU tendrá una serie de proyectos que los estudiantes o aficionados pueden implementar en Logisim para modificar y ampliar las capacidades de la CPU, para ayudar en la comprensión de cómo funciona una CPU. Estos proyectos incluirán la mejora del lenguaje ensamblador y la modificación del ensamblador para manejar las modificaciones, así como la modificación del hardware para manejar las nuevas características.

Este libro también puede utilizarse con otros libros del autor para crear una clase completa de Organización de Computadoras. El libro Digital Circuits Projects implementa los circuitos integrados básicos utilizados en este libro de texto utilizando chips breadboards para ilustrar el funcionamiento de estos circuitos. El libro Introducción a la Programación en Lenguaje Ensamblador MIPS introduce a los estudiantes en un lenguaje ensamblador real para el ordenador MIPS, e integra el lenguaje ensamblador en construcciones de programación más amplias como la programación estructurada, la llamada a subprogramas y convenciones, y conceptos de memoria como pila, montón, estática y memoria de texto. El autor utiliza estos tres textos en su curso de arquitectura de ordenadores. Todos ellos pueden descargarse gratuitamente (excepto el libro de ejercicios asociado a este libro de texto, por el que se cobra una cantidad simbólica) desde los enlaces proporcionados. Se pueden encontrar recursos adicionales para estos libros de texto en el sitio web del autor, http://chuckkann.com.

Esta monografía es la primera de una serie planeada de monografías, que se espera sean escritas como proyectos de investigación de estudiantes, que implementarán diferentes diseños de CPU, incluyendo la diferencia entre las arquitecturas Von Neumann y Harvard, y las arquitecturas de computación de 0 direcciones, una dirección y 2/3 direcciones. Esto ayudará a los lectores a comprender algunas de las decisiones de diseño que intervienen en la implementación de una CPU. Todos los diseños de CPU estarán basados en RISC, aunque se puede implementar una arquitectura CISC de 3 direcciones para mostrar la complejidad de los diseños CISC y hablar de por qué ya no se utilizan.

El autor es profesor adjunto en dos escuelas, Gettysburg College y Johns Hopkins University, programa de Profesionales de la Ingeniería. Ninguna de las dos escuelas cuenta con un gran número de estudiantes con los que trabajar, por lo que si algún estudiante que utilice este libro está interesado en implementar una CPU como parte de esta serie, estaría dispuesto a trabajar con él y su asesor en ello

como proyecto de investigación. Por favor, póngase en contacto conmigo en ckann(at)gettysburg.edu y yo estaría encantado de discutir esto.

Agradecimientos

Las siguientes personas han colaborado en la elaboración de este texto.

Me gustaría dar las gracias a Amrit Dhakal por su ayuda en la escritura y depuración del código ensamblador, y por sus ideas en el desarrollo de la CPU.

El diseño de la portada incluye la imagen de una CPU genérica extraída de la web http://www.freeiconspng.com/free-images/microprocessor-icon-9584

Contenido

Cifras

1 Introducción

La mayoría de los usuarios de ordenadores tienen una metáfora cognitiva incorrecta, pero útil, de los ordenadores en la que el usuario dice (o teclea o hace clic) algo y se produce un comportamiento místico, casi inteligente o mágico. No es exagerado describir a los usuarios de ordenadores como personas que creen que los ordenadores siguen las *leyes de la magia*, en las que se introduce algún *conjuro mágico* y el ordenador responde con un comportamiento esperado, pero mágico.

Este ordenador mágico no existe en realidad. En realidad, los ordenadores son máquinas, y cada acción que realiza un ordenador se reduce a un conjunto de operaciones mecánicas. De hecho, la primera definición completa de un ordenador en funcionamiento fue una máquina mecánica diseñada por Charles Babbage en 1834, que funcionaba a vapor.

Probablemente, el mayor éxito de las Ciencias de la Computación (CS) en el siglo 20th fue el desarrollo de abstracciones que ocultan la naturaleza mecánica de los ordenadores. El hecho de que la gente corriente utilice los ordenadores sin plantearse que son mecánicos es un triunfo de los diseñadores de la informática.

El propósito de esta monografía es romper la comprensión abstracta de un ordenador y explicar su comportamiento en términos completamente mecanicistas. Tratará específicamente de la Unidad Central de Procesamiento (CPU) del ordenador, ya que es ahí donde se produce la magia. Todas las demás partes de un ordenador pueden considerarse como meras fuentes de información para la CPU.

Esta monografía tratará de un tipo específico de CPU, una CPU de una dirección, y explicará esta CPU usando sólo compuertas estándar, específicamente compuertas AND, OR, NOT, NAND y XOR, y 4 Circuitos Integrados (CIs) básicos, el Decodificador, Multiplexor, Sumador, y Flip Flop. Todas estas puertas y componentes pueden describirse como transformaciones mecánicas de datos de entrada en datos de salida, y la CPU en su conjunto puede considerarse un dispositivo mecánico.

Aunque no es necesario conocer los detalles de la implementación de estos circuitos integrados para leer este texto, sólo cómo se utilizan los circuitos integrados, la implementación de estos 4 circuitos integrados no es difícil. Un libro gratuito sobre la implementación de estos circuitos integrados está disponible en http://cupola.gettysburg.edu/oer/1/. El resto de este capítulo proporcionará una visión básica de las puertas y los circuitos integrados utilizados en este texto (Sección 1.1) y luego dará una visión general de las diferentes formas en que una CPU puede ser organizada y diseñada.

1.1 Componentes básicos de una CPU

Esta sección trata de los componentes básicos de una CPU. Cubre las puertas que se utilizan en la CPU, y cuatro circuitos integrados comunes utilizados en una CPU, el sumador, decodificador, multiplexor, y el registro.

1.1.1 Operaciones booleanas

Las puertas son implementaciones hardware de las operaciones booleanas. Las operaciones booleanas son operaciones que toman uno o más valores binarios y calculan un resultado. Por ejemplo, la operación AND

toma 2 valores binarios (con 0 = falso y 1 = verdadero) y calcula una salida binaria. Para las operaciones AND, las entradas `0 AND 0`, `0 AND 1` y `1 AND 0`dan `0` (falso), y la entrada `1 AND 1` da `1` (verdadero). Esto se implementa normalmente utilizando una tabla verdadero-falso, como sigue:

Entrada		Salida
A	**B**	**Y**
0	0	0
0	1	0
1	0	0
1	1	1

Tabla 1-1 : Tabla verdadero-falso de la puerta AND

En este texto se utilizarán 5 operadores booleanos: AND, OR, NOT, XOR y NAND. El NOT es una función unaria (sólo toma una entrada), y así se indica en la Tabla 1-2.

Entrada	Salida
A	**NO**
0	1
1	0

Tabla 1-2: Tabla verdadero-falso de la puerta NOT

Los operadores AND, OR, XOR y NAND son binarios (toman dos entradas) y se muestran en la Tabla 1- 3 a continuación.

Entrada		Salida			
A	**B**	**Y**	**O**	**XOR**	**NAND**
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	1
1	1	1	1	0	0

Tabla 1-3: Tabla verdadero-falso de las puertas AND, OR, XOR y NAND

1.1.2 Circuitos integrados

Un Circuito Integrado (CI) es una colección de puertas que se utilizan para construir componentes para implementar un comportamiento. Los componentes de un CI son compuertas simples, y todos los CI de este capítulo pueden reducirse fácilmente a compuertas AND, OR, NOT, XOR y NAND. Así como las compuertas transforman mecánicamente la entrada en salida, los CIs también hacen una transformación mecánica de entradas en salidas.

Los circuitos integrados que se describirán en este capítulo son el Sumador, el Decodificador, el Multiplexor y el Flip Flop.

1.1.3 ALU (Sumador)

La unidad aritmética lógica (ALU) es el componente central de la CPU. Realiza todas las operaciones aritméticas y lógicas sobre los datos. Todo lo demás en la CPU está diseñado para proporcionar datos sobre los que opera la ALU.

La ALU es normalmente una caja *negra* que proporciona las operaciones para la CPU sobre dos operandos. Esta caja negra es responsable de todas las operaciones que realiza la CPU, incluyendo no sólo las operaciones con enteros y lógicas, sino también los cálculos en coma flotante. Operaciones como los cálculos en coma flotante son muy complejas, y a menudo se implementan en coprocesadores. Para mantener las cosas simples, los únicos tipos de datos permitidos para la CPU en este texto serán enteros, y sólo se permitirán operaciones lógicas y de enteros.

Una visión general de una ALU se puede ver en la ALU típica que se muestra a continuación. Una ALU toma dos argumentos e implementa operaciones como la suma, la resta, la multiplicación y la división de estos dos operandos. La ALU también permite realizar operaciones booleanas (AND, OR, XOR, etc.), desplazamiento de bits y comparación.

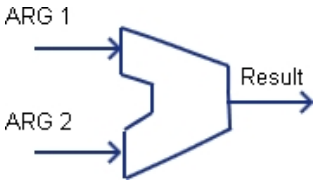


Figura 1-1: ALU

Debido a que la única operación ALU cubierta en el texto recomendado sobre circuitos integrados es un sumador, la ALU utilizada en la implementación Logisim de la CPU contendrá sólo un circuito sumador. Utilizando un sumador, se implementan tanto la suma como la resta. Una configuración más robusta para una ALU se puede encontrar en las notas adicionales a las que se puede acceder para este texto.

1.1.4 Decodificador

Un descodificador es un CI que divide un número de n bits en 2^n líneas de salida separadas. Por ejemplo, consideremos un número de 2 bits, que puede tener 4 valores, 0x0 ... 0x3. Un descodificador tomaría como entrada 2 líneas de entrada que representan el número de 2 bits, y encendería una (y sólo una) de las cuatro líneas de salida. La línea que se enciende corresponde al valor de la entrada de 2 bits. Así que en el siguiente diagrama, si la entrada de 2 bits tiene ambas líneas altas (representando "11"), y la línea de salida 3 se enciende.

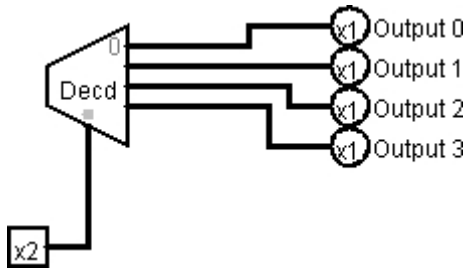


Figura 1-2: Decodificador

1.1.5 Multiplexor

Un multiplexor es un circuito integrado que selecciona entre diferentes entradas. En el siguiente diagrama, los 8 bits utilizados por la Salida pueden provenir del Registro 1 o del Registro 2. El MUX selecciona qué valor de 8 bits utilizar. El MUX selecciona que valor de 8 bits utilizar. Si la Entrada Seleccionada es 0, se elige el Registro 1, y si la Entrada Seleccionada es 1 se elige el Registro 2.

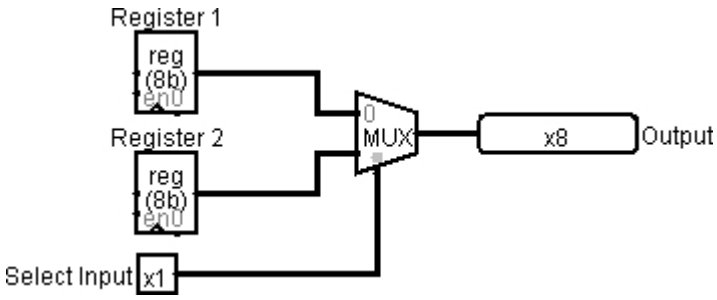


Figura 1-3: Multiplexor

1.1.6 Registros (D Flip Flops) y Memoria

La memoria es diferente de los otros ICs en que es síncrona, donde síncrono significa que la celda de memoria tiene un valor que en intervalos de tiempo discretos. Un ejemplo de este comportamiento es el `$ac` en el siguiente fragmento de programa:

```
clac            ← tiempo = t0, $ac = 5
addi 5          ← tiempo = t1, $ac = 5
addi 7          ← tiempo = t2, $ac = 5
subi 2          ← tiempo = t3, $ac = 5
```

Este programa muestra que el valor de la memoria, `$ac`, cambia discretamente con el tiempo. Este comportamiento discreto se logra mediante un reloj del sistema. Un reloj del sistema es un circuito oscilador electrónico que produce una onda cuadrada con una frecuencia precisa. La siguiente es una ilustración de una onda cuadrada.

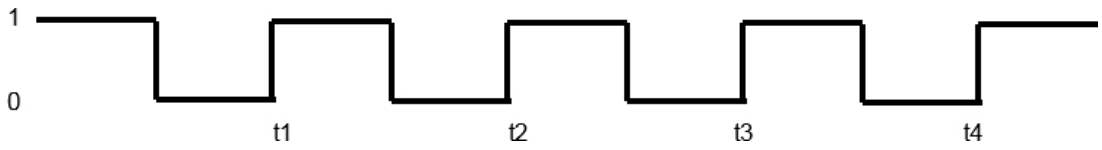


Figura 1-4: Onda cuadrada

En una onda cuadrada, el valor es siempre 0 ó 1, y la memoria utiliza la transición de 0 a 1 (el flanco positivo) para cambiar el valor de todos los componentes de la memoria. Así, las celdas de memoria tienen valores discretos que cambian en cada pulso de reloj.

Cualquier celda de memoria de una CPU se denomina normalmente registro. La memoria de registro suele consistir en memoria RAM estática (SRAM) y se implementa mediante Flip Flops. La memoria del ordenador principal suele ser RAM dinámica (DRAM), aunque algunas memorias, especialmente la memoria caché, pueden implementarse con SRAM. Los detalles específicos de la memoria están más allá del alcance de este texto, y todo lo que el lector necesita saber es que la memoria de registro es típicamente SRAM y está localizada dentro de la CPU.

1.2 Comparación de arquitecturas informáticas

Esta monografía es la primera de una serie de monografías que cubrirán diferentes tipos de CPU, donde las dos grandes diferencias entre los tipos de CPU es el formato de dirección de las instrucciones, y cómo se divide la memoria de instrucciones y datos para el procesador. La siguiente sección tratará sobre el formato de dirección de las instrucciones. La siguiente sección tratará sobre los diseños en los que la memoria de instrucciones y la de datos están combinadas (arquitectura Von Neumann) frente a las memorias de instrucciones y datos separadas (arquitectura Harvard).

1.2.1 Arquitectura de cero, una y dos/tres direcciones

La principal diferencia entre las arquitecturas de 0, 1 y 2/3 direcciones es la procedencia de los operandos para la ALU. Esta sección describe cada una de estas arquitecturas.

Tenga en cuenta que en todas estas arquitecturas, los operandos pueden proceder de registros/memoria, o los operandos pueden formar parte de la propia instrucción. Por ejemplo, el valor utilizado en la instrucción `add A` en un ordenador de una dirección proviene de la celda de memoria en una dirección correspondiente a la etiqueta `A`, y la instrucción añade *el valor en la* posición de memoria `A` a la `$ac`. En la instrucción `adi 5` el valor del operando se incluye en la instrucción, y se denomina valor *inmediato*. En esta serie de monografías, los operadores que utilizan un valor inmediato irán acompañados de una "*i*". Por ejemplo, como se muestra arriba, la instrucción *add* utiliza un valor de memoria, y `addi` utiliza un valor inmediato.

1.2.1.1 Arquitectura de direcciones 0

Cuando se habla de la arquitectura de direcciones de un ordenador, la cuestión central es cómo se recuperan los argumentos para la ALU y dónde se almacenan los resultados de la ALU. Una arquitectura de dirección 0 recupera (pops) los dos argumentos de la parte superior de una *pila* de operandos, realiza la operación, y luego almacena (pushes) el resultado de nuevo en la pila de operandos. Los dos operandos de la ALU están implícitos como los dos operandos de la parte superior de la pila, y la operación, en este caso sumar, no especifica ningún operando. Debido a que el operador no toma ningún operando explícito, las direcciones 0 se incluyen como parte de la operación y esto se denomina arquitectura de dirección 0. Nótese que una arquitectura de 0 direcciones es a menudo referida como una arquitectura de pila porque usa una pila para los operandos hacia/desde la ALU.

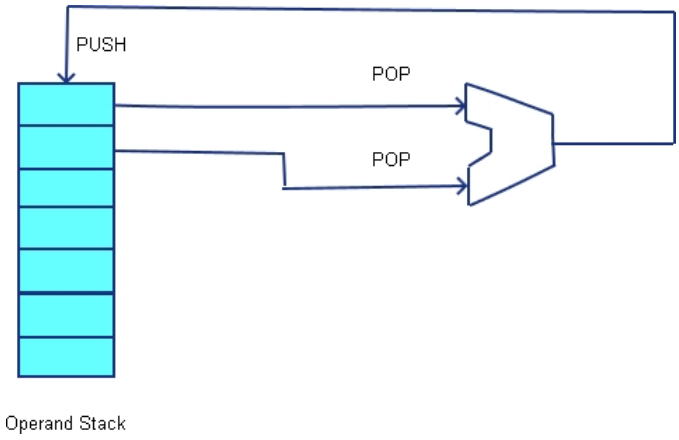


Figura 1-5: Arquitectura de dirección 0

Cuando se escribe código ensamblador para esta arquitectura, los operandos se introducen primero en la pila (desde la memoria o valores inmediatos) utilizando dos operaciones *push*. Se ejecuta la operación que consiste en sacar los dos operandos de la pila, ejecutar la ALU y empujar el resultado de nuevo a la pila. A continuación, la respuesta se almacena en memoria mediante una operación *pop*. El siguiente programa, que suma el valor de la variable A y el valor 5, y luego almacena el resultado de nuevo en la variable B, ilustra un sencillo programa de dirección 0.

```
PUSH A
PUSHI
5 ADD
POP B
```

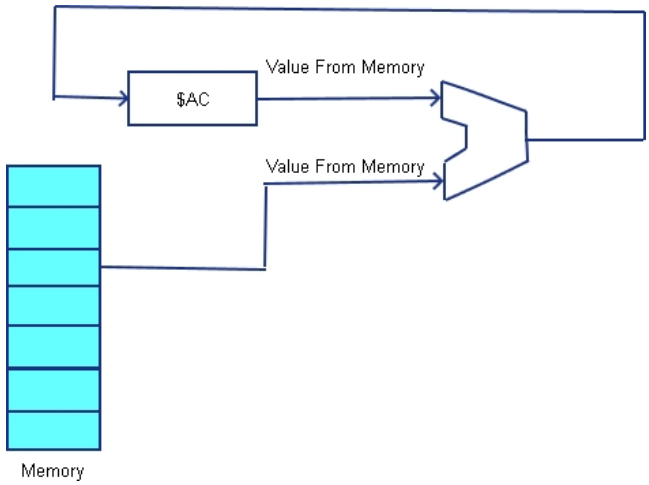
Programa 1-1: programa de dirección 0 para sumar dos números

Históricamente ha habido ordenadores implementados con arquitecturas de dirección 0, como las series 6500 y 7500 de Burroughs, pero rara vez o nunca se utiliza en las arquitecturas de hardware modernas.

Sin embargo, la mayoría de los lenguajes modernos que se ejecutan en Máquinas Virtuales (MV), como la Máquina Virtual Java (MVJ) o el Common Language Runtime (CLR) de .Net, implementan arquitecturas de dirección 0, o de pila.

1.2.2 Arquitectura de una dirección

En una arquitectura de una dirección se mantiene en la CPU un registro especial, llamado *Acumulador* o `$ac`. El `$ac` es siempre un operando de entrada implícito a la ALU, y es también el destino implícito del resultado de la operación de la ALU. El segundo operando de entrada es el valor de una variable de memoria o un valor inmediato. Esto se muestra en el siguiente diagrama.



Programa 1-2: arquitectura de una dirección

El siguiente es un sencillo programa de ordenador de una dirección para sumar el valor 5 y el valor de la variable A, y almacenar el resultado de nuevo en la variable B.

```
        CLR// Poner el AC a 0
ADDI 5 // Suma 5 al $AC. Como antes era 0, esto carga 5 ADD A / /
Suma A+5, y almacena el resultado en el $AC
STOR B // Almacena el valor de $AC en la variable de memoria B
```

Programa 1-3: programa de una dirección para sumar dos números

Debido a que sólo se especifica un valor en la instrucción del operador ALU, este tipo de arquitectura se denomina arquitectura de una dirección. Dado que una arquitectura de una dirección siempre tiene un acumulador, también se denomina arquitectura *de acumulador*.

Históricamente, muchas de las primeras CPU utilizaban diseños de una dirección, como el Intel 8080 y el PDP-8, que utilizaban arquitecturas de acumuladores. Debido a su simplicidad y a que pueden ser más rápidas que otras arquitecturas, algunos diseños de microordenadores siguen utilizando una arquitectura de acumuladores, aunque la mayoría de los ordenadores implementan diseños de registros de propósito general.

1.2.3 Dos/Tres - Arquitectura de direcciones

Las arquitecturas de dos y tres direcciones se denominan arquitecturas de *registro de propósito general*. Los diseños de dos y tres direcciones funcionan de forma similar. Ambas arquitecturas tienen un cierto número de registros de propósito general que se pueden utilizar para seleccionar las dos entradas de la ALU, y el resultado de la operación de la ALU se escribe de nuevo en un registro de propósito general.
La diferencia está en cómo se especifica el resultado de la operación de la ALU (el registro de destino). En una arquitectura de tres direcciones, los 3 registros son el destino (donde escribir los resultados de la ALU), `Rd`, y los dos registros de origen que proporcionan los valores a la ALU, `Rs` y `Rt`. Esto se muestra en la siguiente figura.

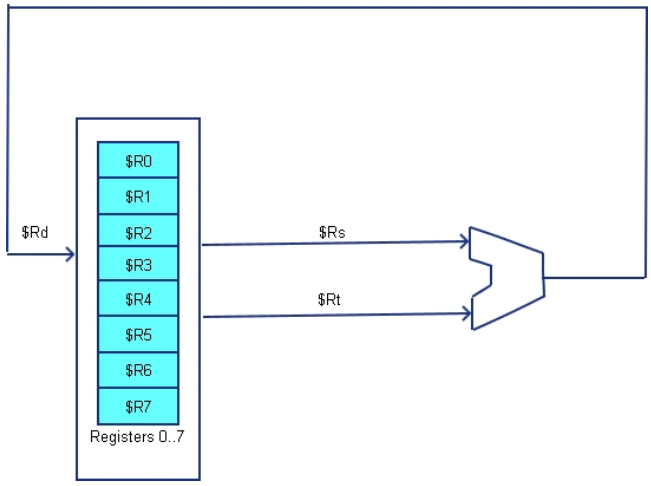


Figura 1-6: Arquitectura de 3 direcciones

Una arquitectura de 2 direcciones es similar a una arquitectura de 3 direcciones, y la única diferencia es que sólo se especifican 2 registros en la instrucción, el primero se utiliza tanto para el destino de la operación como para la primera fuente a la ALU.

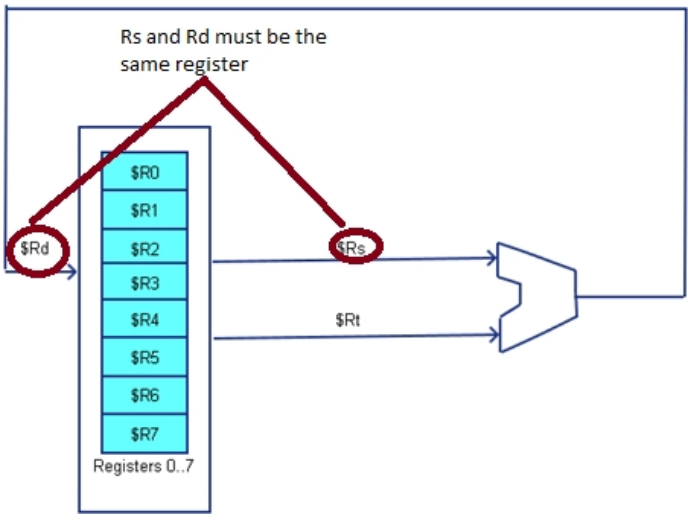


Figura 1-7: Arquitectura de 2 direcciones

Como se muestra en los diagramas, la CPU selecciona dos de los registros de propósito general los valores a enviar a la ALU, y otra selección se hace para escribir el valor de la ALU de nuevo a un registro. En este diseño, todos los valores pasados a la ALU deben venir de un registro de propósito general, y los resultados de la ALU deben ser almacenados en un registro de propósito general. Esto requiere que se acceda a la memoria mediante operaciones de carga y almacenamiento, y el nombre correcto para un ordenador de dos/tres direcciones es "ordenador de carga/almacenamiento de dos/tres direcciones".

Los dos programas siguientes ejecutan el mismo programa, B=A+5, que en los ejemplos anteriores. El primer ejemplo utiliza el formato de 3 direcciones, y el segundo el formato de 2 direcciones.

```
LOAD $R0, A
LOADI $R1, 5
ADD$R0, $R0, $R1
STORE B,   $R0
```

Programa 1-4: programa de 3 direcciones para sumar dos números

```
LOAD $ R 0 , A
LOADI $R1, 5
ADD$R0, $R1
STORE B,   $R0
```

Programa 1-5: programa de 2 direcciones para sumar dos números

1.3 Arquitecturas Von Neumann y Harvard

Cuando se habla de cómo se accede a la memoria a nivel de CPU, hay que tener en cuenta dos diseños. El primero es una arquitectura Von Neumann, y el segundo es una arquitectura Harvard. La principal diferencia entre las dos arquitecturas es que en una arquitectura Von Neumann toda la memoria es

capaz de almacenar todos los elementos del programa, datos e instrucciones; en una arquitectura Harvard la memoria se divide en dos memorias, una para datos y otra para instrucciones.

Para esta monografía, el principal problema a la hora de decidir qué arquitectura utilizar es que algunas operaciones tienen que acceder a la memoria tanto para obtener la instrucción que se va a ejecutar como para acceder a los datos sobre los que se va a operar. Dado que sólo se puede acceder a la memoria una vez por ciclo de reloj, en principio una arquitectura Von Neumann requiere al menos dos ciclos de reloj para ejecutar una instrucción, mientras que una arquitectura Harvard puede ejecutar instrucciones en un solo ciclo.

La capacidad en una arquitectura Harvard de ejecutar una instrucción en una sola instrucción conduce a un diseño mucho más simple y limpio para una CPU que una implementada usando una arquitectura Von Neumann. Para esta primera monografía se implementará una implementación Harvard. En monografías posteriores se estudiará la implementación de la CPU utilizando la arquitectura Von Neumann.

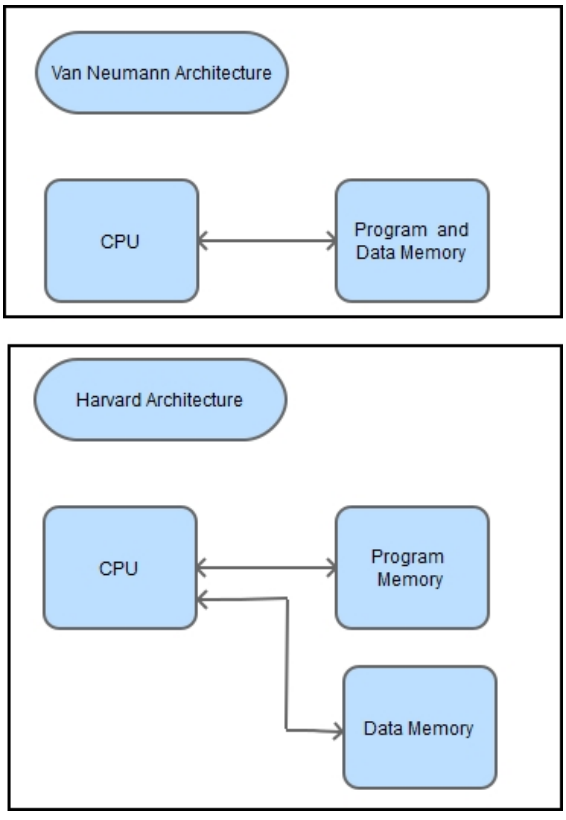


Figura 1-8: Diferencia entre una arquitectura Von Neumann y una Harvard

2 Lenguaje ensamblador

2.1 Qué es el lenguaje ensamblador

El lenguaje ensamblador es un lenguaje de muy bajo nivel, legible y programable por el ser humano, en el que cada instrucción del lenguaje ensamblador corresponde a una instrucción del código máquina del ordenador. Los programas en lenguaje ensamblador se traducen directamente en instrucciones en código máquina, y cada instrucción en ensamblador se traduce en una única instrucción en código máquina[1] .

Tras haber elegido un formato básico de direcciones para la arquitectura, se define el formato del lenguaje ensamblador, denominado Arquitectura de Conjuntos de Instrucciones (ISA). El siguiente paso consiste en diseñar todo el lenguaje ensamblador que se traducirá y ejecutará en la CPU.

A continuación se describen los pasos del diseño de la CPU.

1. En primer lugar, se diseña el lenguaje ensamblador que puede utilizarse para escribir programas para esta CPU. Este lenguaje se prueba implementando programas sencillos en el lenguaje ensamblador.
2. Se escribirá una representación en código máquina para el lenguaje ensamblador. La CPU sólo tiene capacidad para interpretar información que sean datos binarios, por lo que el lenguaje ensamblador debe traducirse a datos binarios que sean comprensibles para la CPU.
3. A continuación, la CPU se diseña para ejecutar las instrucciones del código máquina.

Este proceso de diseño del lenguaje, creación del código máquina para el lenguaje e implementación de la CPU es normalmente iterativo, pero el único producto final de estos pasos se incluirá en esta monografía.

El primer paso, la creación del lenguaje ensamblador, es el tema de este capítulo.

Para crear un lenguaje ensamblador hay que definir tres restricciones principales del lenguaje.

1. Es necesario definir los datos que se tratarán en esta CPU. En lenguajes de alto nivel esto serían tipos, como entero, flotante o cadena. En una CPU, los tipos no existen. En su lugar, la CPU se ocupa de cuestiones como el tamaño de una palabra en el ordenador, y cómo se utilizarán las direcciones de memoria para recuperar datos.
2. Conjunto de directivas de ensamblador para controlar el programa ensamblador mientras se ejecuta. Directivas para definir cuestiones como el tipo de memoria a la que se accede (texto o datos), etiquetas para especificar direcciones en el programa, cómo asignar y almacenar datos del programa y cómo se definen los comentarios.

[1] Normalmente se elimina esta correspondencia uno a uno entre las instrucciones del lenguaje ensamblador y las instrucciones del código máquina. Los lenguajes ensambladores definen pseudoinstrucciones que se traducen en múltiples instrucciones de código máquina. Estas pseudoinstrucciones están diseñadas para facilitar la escritura de programas en el lenguaje ensamblador. Esta monografía está diseñada para mostrar al lector cómo funciona la CPU, por lo que cada instrucción del lenguaje ensamblador se corresponderá con una única instrucción del

lenguaje máquina.

3. Debe definirse un conjunto completo de instrucciones de ensamblador.

La siguiente sección de este capítulo dará algunas advertencias a los programadores que vienen de un lenguaje de nivel superior sobre cuestiones que deben tener en cuenta al programar en lenguaje ensamblador. Las 3 secciones siguientes definirán los datos utilizados en el ensamblador, las directivas del ensamblador y las instrucciones del ensamblador.

2.2 Advertencias sobre el lenguaje ensamblador

Los programadores que han aprendido lenguajes de alto nivel, como Java, C/C++, C# o Ada, a menudo han desarrollado formas de pensar sobre un programa que son inapropiadas para lenguajes y sistemas de bajo nivel como el lenguaje ensamblador. Esta sección dará algunas sugerencias a los programadores que se acerquen al lenguaje ensamblador por primera vez.

Lo primero que hay que tener en cuenta es que todas las instrucciones deben implementar operaciones primitivas. Los lenguajes de alto nivel permiten una abreviatura que implica muchas instrucciones. Por ejemplo, la sentencia `B=A+5` implica operaciones de carga que preparan las variables `A` y `5` para ser enviadas a la ALU. A continuación se debe realizar una operación de `suma` por parte de la ALU. Finalmente debe ejecutarse una operación para almacenar el resultado de la ALU de nuevo en la variable `B`. En ensamblador el programador debe especificar todas las operaciones primitivas necesarias. No hay atajos.

Lo segundo que hay que tener en cuenta es que, a pesar de lo que pueda haber oído acerca de que las sentencias goto son malas, no hay forma de implementar el control del programa, como las sentencias if o los bucles, sin utilizar una instrucción branch, que es el equivalente a una sentencia goto. Esto no significa que las construcciones de programación estructurada no puedan utilizarse de forma eficaz. Si un programa está confundido sobre cómo implementar construcciones de programación estructurada en ensamblador, hay un capítulo en un libro gratuito sobre programación en ensamblador MIPS escrito por el autor de esta monografía que explica cómo se puede lograr esto.

El tercer punto importante sobre el lenguaje ensamblador es que los datos no tienen contexto. En un lenguaje de nivel superior, normalmente las variables A y B, y el número 5, se especifican como números enteros. El lenguaje de alto nivel sabe que son enteros y proporciona un contexto para interpretarlos. La operación de suma se sabe que es una suma de enteros, y el compilador generará una instrucción para hacer la opción de enteros y no una operación de coma flotante. Si la declaración de los números se cambiara a float, la operación de suma en el lenguaje de alto nivel se cambiaría a una suma en coma flotante. El lenguaje de nivel superior conoce las variables de tipo, y puede proporcionar el contexto adecuado para interpretarlas.

En lenguaje ensamblador, no hay contexto para ningún dato. Los datos pueden ser un número entero, un valor booleano, un número de coma flotante, caracteres ASCII o incluso instrucciones de programa. El ensamblador no tiene ni idea del tipo, y simplemente ejecutará la operación especificada. En ensamblador es posible realizar operaciones sin sentido, como sumar dos instrucciones de programa. El lenguaje ensamblador gustosamente le permitirá hacer cosas sin sentido y completamente absurdas, y no le advertirá de ninguna manera de que

no tiene sentido. El ensamblador no tiene contexto para los datos, y no hay forma de corregir este problema porque desde el punto de vista del ensamblador, no hay problema.

Cuando se programa en lenguaje ensamblador, es importante que el programador mantenga el conocimiento del *contexto* actual del programa. Es el programador quien sabe si dos elementos de datos son enteros, y por lo tanto una operación de suma de enteros es apropiada. Es el programador quien debe saber si los valores con los que se está trabajando son direcciones o valores, y realizar las operaciones de desreferenciación adecuadas. No hay nada más que el conocimiento del programador para asegurar que un programa ejecutará operaciones correctas en los tipos de datos apropiados.

2.3 Directivas del ensamblador

Las directivas de ensamblador son instrucciones para que el ensamblador realice una acción o cambie una configuración. Las directivas de ensamblador no representan instrucciones y no se traducen a código máquina.

Para este ensamblador, todas las directivas comienzan con "." o "#" (el comentario es un #), y la directiva debe existir en una línea separada de cualquier otra directiva de ensamblador o instrucción de ensamblador. Hay 4 directivas de ensamblador y la etiqueta de comentario.

- .text - La directiva .text indica al ensamblador que la información que sigue es texto de programa (instrucciones de ensamblado), y que el código máquina traducido debe escribirse en el segmento de texto de la memoria.

- .data - La directiva .data indica al ensamblador que la información que sigue son datos de programa. La información que sigue a una instrucción .data serán valores de datos, y se almacenarán en el segmento de datos.

- .label - Una etiqueta es una dirección de memoria que corresponde a una instrucción o a un valor de datos. Es sólo una conveniencia para que el programador pueda hacer referencia a una dirección por un nombre. Se utilizará de la siguiente manera:

  .label nombre

  La etiqueta es una etiqueta que puede ser referenciada en lugar de una dirección en cualquier instrucción en ensamblador que pueda tomar una etiqueta/dirección. Las etiquetas y las direcciones pueden utilizarse indistintamente.

- .number - La directiva number indica al ensamblador que reserve 2 bytes de memoria para un valor de datos y que inicialice la memoria con el valor dado. A menudo se utiliza con la directiva
  .label para establecer una etiqueta a un valor de memoria de 2 bytes, e inicializar el valor, como se muestra en el siguiente fragmento de código.

```
.label var1
    .número 5
```

Esta sentencia asigna espacio para la variable `var1`, y asigna a ese espacio en memoria el valor 5. Los datos para esta CPU sólo funcionarán con números enteros de 2 bytes (16 bits), por lo que se pueden utilizar valores discretos desde -32768..32767 (inclusive). Todos los valores de datos están en decimal; el ensamblador no reconocerá valores hexadecimales.

- # - el # (hashtag) se utiliza para especificar un comentario. Cualquier cosa en una línea que comience con "#" es una línea de comentario y es ignorada por el ensamblador.

2.4 Tipos de datos

Aunque un lenguaje ensamblador no tiene tipos de datos explícitos, existen reglas sobre cómo se accede a los datos y cómo se almacenan. Esta sección define las reglas de acceso a los datos.

En esta CPU, una palabra son 16 bits. Todas las posiciones de memoria tienen 16 bits de ancho, y las palabras, no los bytes, son direccionables. Así, el valor en la dirección 0 está contenido en los bits 0...15, el valor en la dirección 1 está contenido en los bits 16...31, etc.

Cada dirección se refiere a una cantidad o palabra de 2 bytes. Si esa posición de memoria está en memoria de datos el valor es un número entero; si la dirección está en memoria de texto es una instrucción de 2 bytes.

Hay un total de 256 posiciones de memoria (palabras direccionables) tanto en la memoria de datos como en la memoria de texto (programa). Las direcciones de ambas memorias comienzan en 0 y van hasta 255, que corresponde a un valor de 8 bits sin signo. Aunque las direcciones de memoria se solapan, el contexto de la petición determinará qué memoria utilizar. Sólo la $pc se utilizará para acceder a la memoria de texto, y todas las demás direcciones se referirán a la memoria de datos.

Cuando se hace referencia a valores en instrucciones (valores inmediatos y direcciones) se utiliza un valor de 8 bits. Este valor de 8 bits se puede dar como un valor numérico, o como una etiqueta válida para una dirección en algún lugar del programa. Cuando se utiliza como una dirección, este valor de 8 bits será sin signo, y se refiere a un número entre 0...255. Para las instrucciones `add`, `sub`, y `stor`, esta es una dirección en el segmento de datos. Para las instrucciones branch, `beqz, la dirección de 8 bits se refiere al segmento de texto`.

Para las instrucciones inmediatas, `addi` y `subi`, el valor del operando es un valor entero de 8 bits, y tiene un valor comprendido entre -128...127.

2.5 Diseño de un lenguaje ensamblador

Cuando se diseña un lenguaje ensamblador, un lenguaje para manipular una CPU, hay 3 preocupaciones principales:

1. Transferencia de datos de la memoria principal a la memoria interna de la CPU (registros o pila de operandos).
2. Conjunto de operaciones que puede realizar la ALU con los datos, por ejemplo, `suma`,

`resta`, y, `desplazamiento`, etcétera.

3. Una forma de proporcionar control al programa, por ejemplo para implementar estructuras de tipo *rama (if)* y *bucle (for* o *while)* en un programa. Normalmente la estructura de control será proporcionada por una operación de bifurcación.

Estas tres preocupaciones principales, y cómo se abordan en el lenguaje ensamblador, se discutirán en las siguientes secciones. La última sección de este capítulo dará algunos programas que ilustrarán cómo se escribirá un programa en este lenguaje ensamblador.

2.5.1 Transferencia de datos de la memoria principal a la memoria interna de la CPU

La cantidad de memoria directamente accesible a un programador en la CPU (por ejemplo, registros) es muy limitada. En el caso de la arquitectura de una dirección, sólo una ranura de memoria, la `$ac`, es directamente utilizable por un programador. Por lo tanto, los programas tienen que depender de la memoria principal para almacenar las instrucciones y los datos del programa.

Para transferir elementos de la memoria de datos a la $ac se utilizan las instrucciones add, sub y stor.

Para las instrucciones `add` y `sub`, el segundo operando de la instrucción es la etiqueta o dirección de memoria del valor a recuperar de la memoria y enviado a la ALU. Así, por ejemplo, para cargar un valor en `$ac` desde una posición de memoria etiquetada como `A` se utilizaría el siguiente código.

```
clac
add A
```

Tenga en cuenta que el `$ac` siempre debe ponerse a 0 (utilizando el `clac`) antes de cargar un valor en el `$ac`, o el valor almacenado en el $ac será el resultado de sumar el valor en la posición de memoria A con el valor actual en el $ac.

Para la instrucción `stor`, el segundo operando es la etiqueta o dirección de memoria en la que `almacenar` el valor del `$ac`. Por ejemplo, para almacenar el valor del `$ac` en la memoria en la dirección de la etiqueta `B`, se utilizaría el siguiente código.

```
stor B
```

2.5.2 Conjunto de operaciones ALU válidas

La siguiente consideración es el conjunto de operaciones que la ALU puede realizar con los datos de entrada. Esta lista depende de la complejidad de la ALU. La ALU de este ordenador es muy simple, por lo que sólo admite las operaciones suma y resta.

2.5.3 Control del programa (ramificación)

Para hacer cualquier programa útil, las construcciones if y loop deben ser soportadas. En la CPU de una dirección implementada esto se consigue mediante la operación Branch-if-equal-zero (`beqz`). Para esta operación, si `$ac` es 0, el programa se bifurcará a la dirección de memoria de texto contenida en

la sentencia branch. Esta dirección puede ser una etiqueta que represente la dirección, o el valor numérico de la dirección de bifurcación. Así en la siguiente instrucción, el programa se bifurcará a la dirección de la etiqueta EndLoop si el valor en `$ac` es `0`.

```
beqz EndLoop
```

A menudo se utiliza una sentencia de bifurcación incondicional, pero puede simularse estableciendo primero `$ac` a
`0` antes de la instrucción de bifurcación. La siguiente instrucción implementa una bifurcación incondicional.

```
clac
beqz StartLoop
```

2.5.4 Instrucciones del ensamblador

Basándose en los criterios de la sección anterior, se define un conjunto mínimo de instrucciones de ensamblador para crear programas útiles. Estas instrucciones son suficientes para crear programas útiles, y se mostrarán varios ejemplos al final de este capítulo.

- `add [label/address]` - Añade un valor de la memoria de datos (dm) al $ac actual.

  ```
  $ac <- $ac + dm[dirección]
  ```

  En esta instrucción se puede utilizar tanto una etiqueta como la dirección real del valor. Así, si la etiqueta `A` se refiere a la dirección dm de `5`, las dos instrucciones siguientes son iguales.

  ```
  añadir A
  añade 5
  ```

- `addi immediate` - Añade el valor inmediato de esta instrucción a `$ac`. El valor inmediato es un valor entero de 8 bits entre -128...127

  ```
  $ac <- $ac + inmediato
  ```

  A continuación se muestra un ejemplo de una instrucción `addi` que suma 15 al valor de `$ac`.

  ```
  adi 15
  ```

- `beqz [etiqueta/dirección]` - La instrucción `beqz` cambia el valor en el Contador de Programa (`$pc`) a la dirección de memoria de texto en la instrucción si el valor en el `$ac` es `0`. En esta CPU, el `$pc` siempre especifica la siguiente instrucción a ejecutar, por lo que esto tiene el efecto de cambiar la siguiente instrucción a ejecutar a la dirección en la instrucción. Esto se llama *bifurcación de programa*, o simplemente *bifurcación*.

  ```
  $pc <- dirección SI $ac es 0
  ```

  A continuación se muestra un ejemplo de la instrucción `beqz` que se bifurca a la dirección 16 si `$ac` es `0`:

  ```
  beqz 16
  ```

- `clac` - La instrucción `clac` establece `$ac` a `0`. Esto podría hacerse con un conjunto de operaciones stor y sub, por lo que la instrucción es más que nada por conveniencia.

  ```
  $ac <- 0
  ```

- `sub[etiqueta/dirección]` - Resta un valor de la memoria de datos al $ac actual.

  ```
  $ac <- $ac - dm[dirección]
  ```

  En esta instrucción se puede utilizar tanto una etiqueta como la dirección real del valor. Así, si la etiqueta `A` se refiere a la dirección dm de 5, las dos instrucciones siguientes son iguales.

  ```
  sub A
  sub 5
  ```

- `subi immediate` - Sub el valor inmediato en esta instrucción al `$ac`. El valor inmediato es un valor entero de 8 bits entre -128...128.

  ```
  $ac <- $ac - inmediato
  ```

  A continuación se muestra un ejemplo de una instrucción `subi` que suma 15 al valor de `$ac`.

  ```
  subi 15
  ```

- `stor [etiqueta/dirección]` - Almacena el valor actual de `$ac` en la memoria de datos.

  ```
  dm[dirección] <- $ac
  ```

  En esta instrucción se puede utilizar tanto una etiqueta como la dirección real del valor. Así, si la etiqueta `A` se refiere a la dirección dm de `5`, las dos instrucciones siguientes son iguales.

  ```
  stor A
  stor 5
  ```

- `noop` - esta sentencia no hace nada. Ejecutar esta sentencia no cambia el valor de ninguna memoria o registro (excepto el `$pc`) en el sistema. Se incluye para que la memoria de texto pueda ponerse a 0 y ejecutarse sin cambiar el estado interno del ordenador.

2.6 Programas ensambladores

Los siguientes programas en ensamblador ilustran cómo se puede utilizar el lenguaje ensamblador definido en este capítulo para implementar algunos programas sencillos.

2.6.1 Cargar un valor en la CA

Este primer programa carga un valor inmediato de `5` en el registro `$ac`. Después de ejecutar el programa, el valor en $ac será 5.

```
.text
clac
addi 5
```

Programa 2-1: Carga de un valor en $ac a partir de un valor inmediato

Este segundo programa carga el valor de la dirección de memoria correspondiente a la etiqueta `var1` en `$ac`. Dado que el valor en la dirección de `var1` es `5`, el programa carga el valor `5` en el campo `$ac`.

```
.text
clac
añadir var1
.datos
.label var1
     .número 5
```

Programa 2-2: Carga de un valor en la memoria $ac mediante una etiqueta

Este tercer programa añade el valor en la dirección `0` en el segmento de datos al `$ac`. Como el valor `5` ha sido cargado como el primer valor en el segmento `.data`, el valor `5` es cargado al `$ac`.

```
.text
clac
add 0
.datos
    .número 5
```

Programa 2-3: Carga de un valor en $ac desde la memoria utilizando una referencia

2.6.2 Suma de dos valores inmediatos

Este programa ilustra la suma de 2 valores inmediatos en `$ac`. El `$ac` se inicializa a `0`, y luego el primer valor se carga en el `$ac` desde el valor inmediato de en la instrucción. El valor inmediato de la segunda instrucción se suma a `$ac`, y `$ac` contiene el resultado final.

```
.text
clac
addi 5
adi 2
# Respuesta en el $ac es 7
```

Programa 2-4: Suma de dos valores inmediatos

2.6.3 Suma de dos valores de la memoria y almacenamiento de los resultados

Este programa suma dos valores de la memoria en las etiquetas `var1` y `var2`, los suma y almacena el resultado de nuevo en el valor de la etiqueta `ans`. Este programa también introduce una nueva construcción, que llamaremos *halt*.

El halt es un conjunto de instrucciones que crea un bucle infinito al final del programa para que el programa no continúe simplemente ejecutando instrucciones `noop` hasta que se quede sin memoria. Esta construcción pone `$ac` a `0`, y luego se bifurca a la misma sentencia una y otra vez. El programa se está ejecutando, pero no está progresando el `$pc` o cambiando el estado de la computadora.

```
.text
clac
add var1
add var2
stor ans
.label halt
    clac
    beqz halt
# La respuesta está en la memoria de datos en la dirección 0.
.datos
.label ans
    .número 0
.label var1
     .número 5
.label var2
    .número 2
```

Programa 2-5: Suma de dos valores de memoria y almacenamiento en memoria

2.6.4 Multiplicación por suma iterativa

Este programa multiplica dos números por iteración. Esto significa que n*m se calcula sumando n a sí mismo m veces, por ejemplo n + n + n... etc.

```
.texto
# Comienza el bucle de multiplicación.
.label startLoop

# Cuando se ensambla el programa el multiplicador y el
multiplicando se # inicializan en memoria. El contador se
inicializa a 0
# y se incrementa en 1 cada vez que se pasa por el bucle. Cuando el
# contador == multiplicador, # el valor del contador-
multiplicador = 0, # y la instrucción beqz se bifurca al final
del bucle. clac
add multiplier
sub counter
beqz endLoop # Ir al final cuando termine

# calcular el producto. El producto es inicialmente cero,
pero cada # vez a través del bucle el producto
```

```
# se incrementa con el valor del multiplicando. El producto se #
almacena de nuevo en la memoria para su uso en la siguiente
pasada.
clac
añadir producto
añadir
multiplicando stor
producto

# El contador se incrementa y el programa vuelve # al
principio del bucle para la siguiente pasada. clac
añadir
contador adi 1
stor contador
clac
beqz startLoop

# Cuando contador == multiplicador, el programa se ha completado.
# La respuesta está en la posición de memoria del producto de la etiqueta.
.label endLoop
clac
    beqz endLoop
# el resultado está en ans (dirección de datos 0)

.datos
.label multiplicando
     .número 5
      .label
  multiplicador
    .número 4
.label contador
    .número 0
.label producto
    .número 0
```

Programa 2-6: Multiplicación mediante suma iterativa

3 Código máquina

El código máquina es una representación de un programa en lenguaje ensamblador que el hardware de la CPU puede entender. Como la CPU sólo entiende binario, el código máquina es un lenguaje binario que controla la CPU. Cuando escribimos el código binario máquina, para facilitar su lectura a un humano, el código se reunirá en grupos de 4 bits, y se escribirá el resultado hexadecimal (base 16).

3.1 Visión general del formato de las instrucciones del código máquina

Todas las instrucciones de código máquina de nuestro ordenador constarán de dos segmentos de 4 bits y uno de 8 bits, como se muestra a continuación.

Figura 3-1: Formato de instrucción de máquina de 16 bits

El primer segmento de 4 bits representará el tipo de operación. Los posibles tipos de operaciones son los siguientes:

- 0 - Esta es una instrucción de no operación (o *noop*). No cambia el estado actual del ordenador, y simplemente mueve la CPU a la siguiente instrucción.
- 1 -Este opcode representa una operación inmediata que utiliza la ALU para producir un resultado. Esta instrucción consta de un opcode de 4 bits, una opción ALU de 4 bits (ALUopt) para indicar a la ALU qué operación debe ejecutar y un valor inmediato de datos de 8 bits para un operando. Tal y como está implementada la ALU sólo ejecuta 2 operaciones, 0x0 es suma y 0x1 es resta, aunque los ejercicios al final del texto añaden más operaciones. Se pueden implementar un máximo de 16 operaciones en la CPU.

  A continuación se ofrecen ejemplos de traducción de estas instrucciones en

  ensamblador a código máquina. La instrucción:

  ```
  adi 2
  ```

  se traduce en el siguiente código máquina:

  ```
  0x1002
  ```

  La instrucción:

```
subi 15
```

se traduce al siguiente código máquina

```
0x110f
```

- 2 - Este opcode representa una operación de dirección de memoria que utiliza la ALU para producir un resultado. Esta instrucción se compone del opcode de 4 bits, un ALUopt de 4 bits para indicar a la ALU qué operación debe ejecutar, y una dirección de memoria de datos de 8 bits para un operando. Tal y como está implementada la ALU sólo ejecuta 2 operaciones, 0x0 es suma y 0x1 es resta, aunque los ejercicios al final del texto añaden más operaciones. Se pueden implementar un máximo de 16 operaciones en la CPU.

  A continuación se ofrecen ejemplos de traducción de estas instrucciones en

  ensamblador a código máquina. La instrucción:

  ```
  añadir 2
  ```

  se traduce en el siguiente código máquina:

  ```
  0x2002
  ```

  La instrucción

  ```
  sub 15
  ```

  se traduce en el siguiente código máquina

  ```
  0x210f
  ```

  Tenga en cuenta que durante el proceso de ensamblaje las etiquetas en código ensamblador se traducen a direcciones, por lo que las etiquetas nunca aparecerán en código máquina.

- 3 - Este opcode ejecuta la operación clac (por ejemplo, pone $ac a 0). En esta instrucción se ignoran todos los bits posteriores al 0x3, por lo que pueden contener cualquier valor. Por convención, los bits adicionales deben ponerse siempre a 0.

  Por ejemplo, la siguiente instrucción de montaje

  ```
  clac
  ```

se traduce por

```
0x3000
```

- 4 - Este opcode ejecuta la operación stor. En esta instrucción no se utiliza la ALU opt de 4 bits, y debe ponerse a 0. El valor de la dirección es la dirección en la que se almacenará el valor en $ac. Por ejemplo, la siguiente instrucción

  ```
  stor 15
  ```

  se traduce como:

  ```
  0x400f
  ```

- 0x5 - El opcode ejecuta la operación beqz. En esta instrucción no se utiliza la operación ALU de 4 bits, y debe ponerse en blanco. El resultado de esta operación es que $pc se establece en el valor de la dirección si $ac es cero. Establecer el valor de $pc hace que el programa se bifurque a esa dirección.

  Por ejemplo, la siguiente instrucción

  `beqz 40` se

  traduce

  como:

  ```
  0x5028
  ```

4 Programa ensamblador

El ensamblador es el programa que traduce un programa en lenguaje ensamblador a código máquina. El ensamblador leerá un programa fuente en ensamblador consistente en instrucciones en ensamblador. A continuación, el ensamblador escribirá dos archivos que se utilizarán para ejecutar el programa en la CPU Logisim.

El primer archivo de salida contiene el segmento de código máquina que contiene el programa de código máquina que se utilizará en la CPU. Se trata de una traducción de las instrucciones en ensamblador a instrucciones de máquina. El segundo archivo de salida contiene el segmento de datos inicializados que se utilizará en la CPU. Esto se ilustra en la siguiente figura.

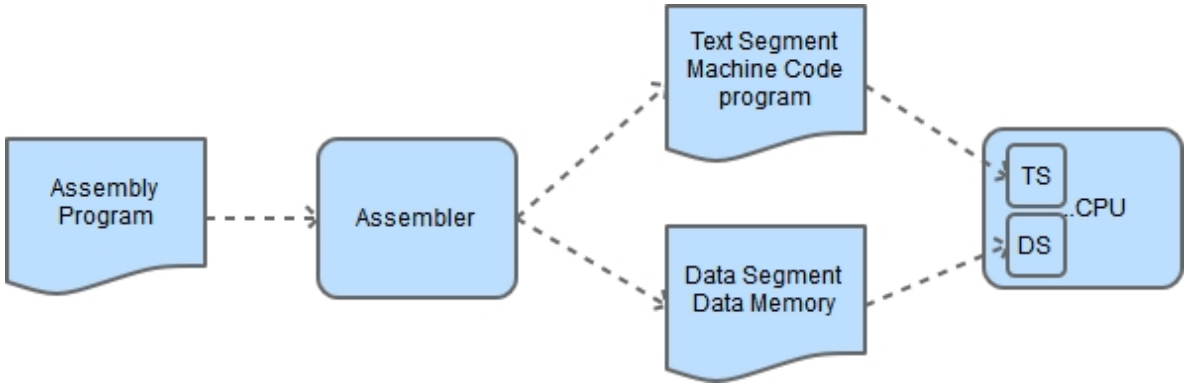


Figura 4-1: Proceso de montaje

El ensamblador es un *ensamblador de dos pasadas*. Un ensamblador de dos pasadas lee el fichero de entrada dos veces, o en 2 pasadas desde el principio hasta el final del fichero de ensamblado fuente. La primera pasada calculará una dirección para cada etiqueta del programa para crear una tabla de símbolos. Una tabla de símbolos es una lista que contiene etiquetas en el programa y su dirección en la memoria. La segunda pasada traducirá cada instrucción en el archivo de entrada a código máquina, y escribirá un archivo que corresponde a código máquinapara los datos y los segmentos de texto para el programa. La segunda pasada por el fichero utiliza la tabla de símbolos para resolver cualquier referencia de etiqueta que se utilice en las instrucciones de ensamblado. El formato del programa se muestra en la siguiente figura.

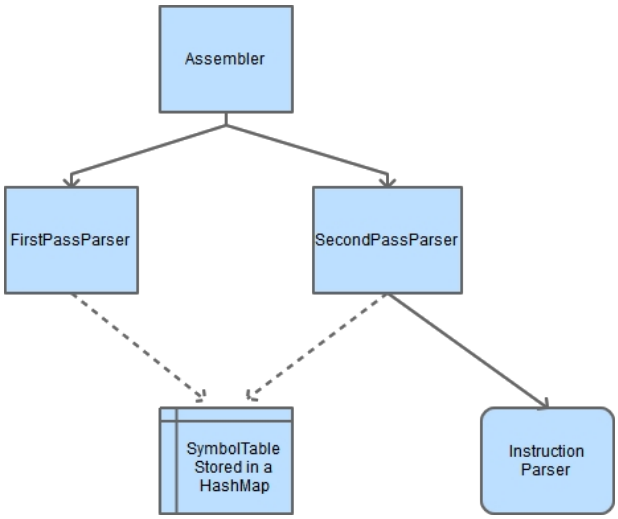


Figura 4-2: Visión general del ensamblador

El ensamblador está escrito en Java. No hay ninguna razón o ventaja particular para escribir este ensamblador en Java. El mayor problema del ensamblador es la escritura de call backs o métodos/funciones virtuales para analizar las instrucciones individuales. Esto sería probablemente más fácil en C/C++, y definitivamente más rápido usando punteros de función en lugar de polimorfismo. Pero el concepto de devolución de llamada es difícil en cualquier lenguaje para los lectores no familiarizados con el concepto, y este ensamblador es más que suficientemente rápido.

El ensamblador utiliza inicializadores estáticos con patrones de fábrica y objetos singleton para implementar la estructura polimórfica de devolución de llamada para procesar los comandos. Esto podría hacerse en una sentencia "if" grande, pero esta solución polimórfica es más limpia y fácil de extender. También es instructivo para los lectores que no estén familiarizados con el patrón de fábrica, el patrón singleton y los inicializadores estáticos. Pero el ensamblador es un programa relativamente corto y directo, y los lectores son bienvenidos a reescribirlo usando cualquier estructura o lenguaje que prefieran.

El ensamblador puede obtenerse en el sitio web del autor, http://chuckkann.com. El archivo zip contiene el ensamblador, el archivo de definición del circuito para la CPU de una dirección, y algunos programas y archivos para que el usuario textee y trabaje con la CPU.

4.1 Ejecutar un programa en la CPU de una dirección

Las siguientes instrucciones detallan cómo utilizar la CPU de una dirección.

1. Primero descarga el archivo One-AddressCPU.zip de la web del autor, http://chuckkann.com. Descomprime el archivo en una ubicación adecuada.

2. En el directorio raíz de este zip hay un archivo JAR ejecutable llamado OneAddressAssembler.jar. Haga doble clic en este icono y debería ver la siguiente pantalla.

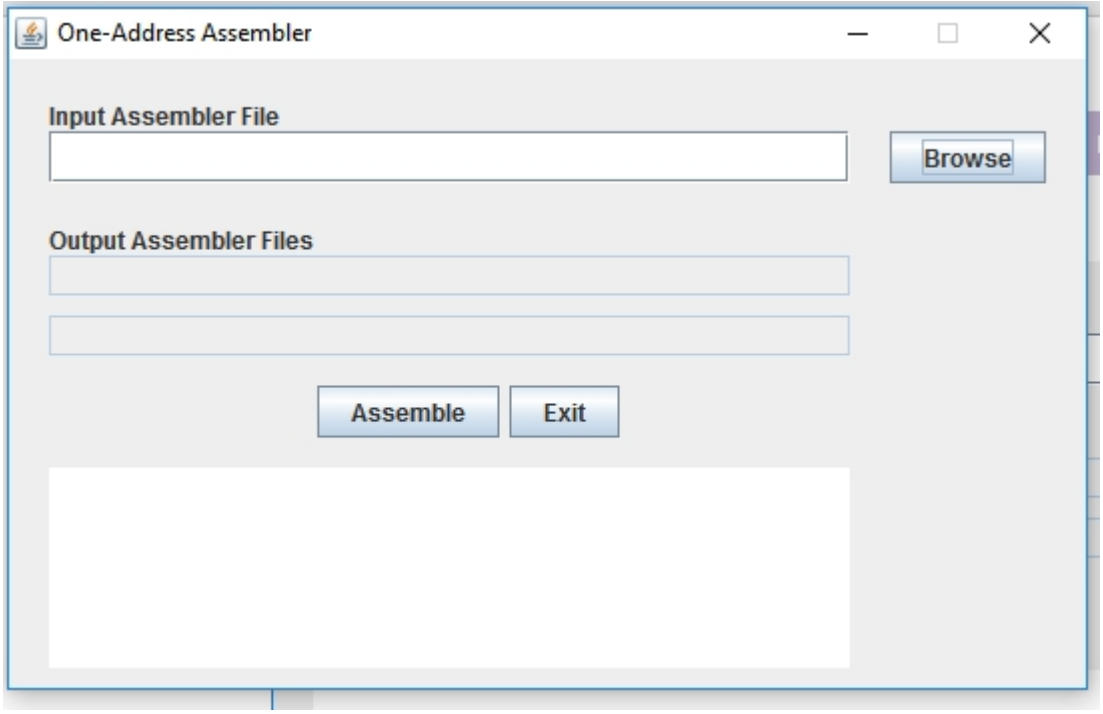


Figura 4-3: Ejecución del ensamblador - paso 1

3. Haga clic en el botón *Examinar* y aparecerá un cuadro de diálogo de archivos. Seleccione el directorio AssemblyPrograms.

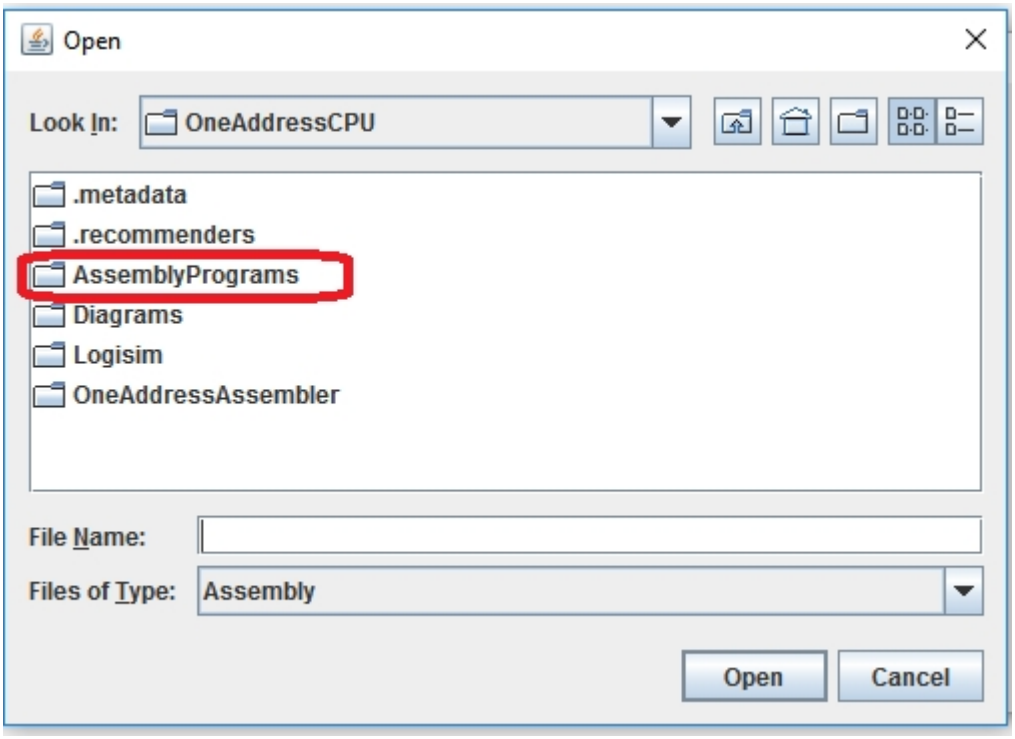


Figura 4-4: Ejecución del ensamblador - paso 2

4. Seleccione el programa AdditionExample.asm. Este es el Programa 2.5 del Capítulo 2, y suma dos valores de la memoria y almacena el resultado en la memoria.

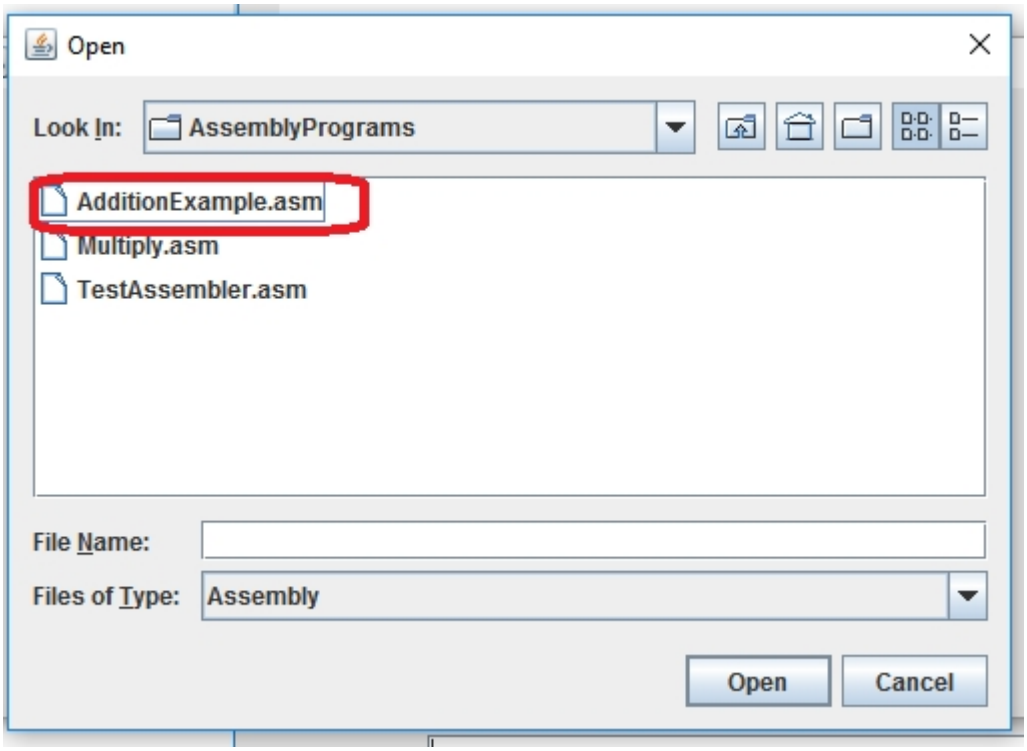


Figura 4-5: Ejecución del ensamblador - paso 3

5. El programa mostrará la siguiente pantalla, que dice que el ensamblador utilizará como entrada el fichero AdditionExample.asm, y si se completa con éxito producirá dos ficheros de salida, AdditionExample.mc (el código máquina) y AdditionExample.dat (los datos). Pulse el botón Ensamblar.

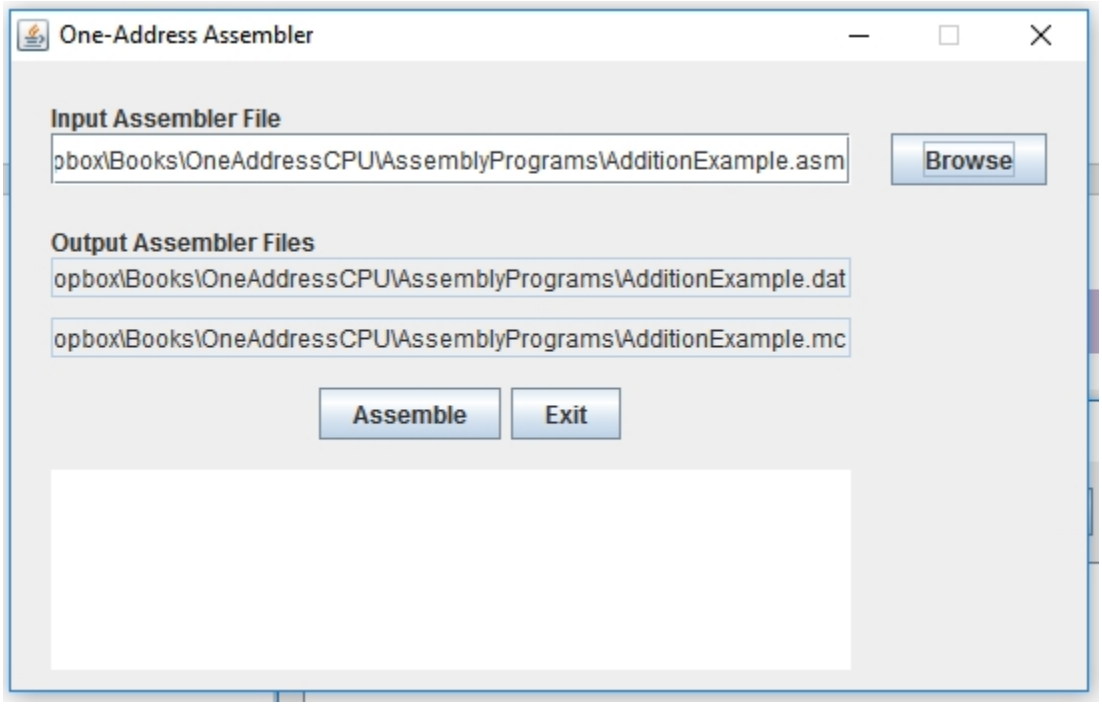


Figura 4-6: Ejecución del ensamblador - paso 4

6. Debería recibir el siguiente mensaje indicando que el ensamblaje se ha completado correctamente. Haga clic en Salir para cerrar el ensamblador.

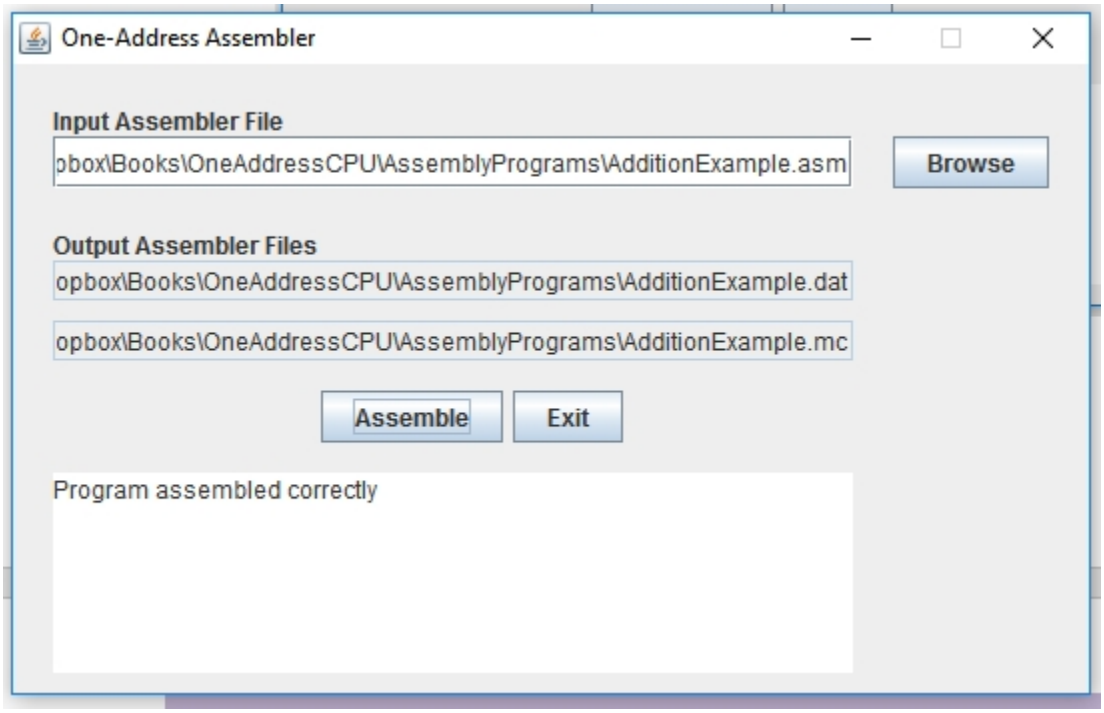


Figura 4-7: Ejecución del ensamblador - paso 5

7. Vaya al subdirectorio *Logisim* y haga clic en el icono logisim-generic-2.7.1.jar. Esto abrirá Logisim, y debería ver la siguiente pantalla. Seleccione el *archivo* >Opción *Abrir*, seleccione el directorio Logisim y abra el archivo OneAddressCPU.circ.

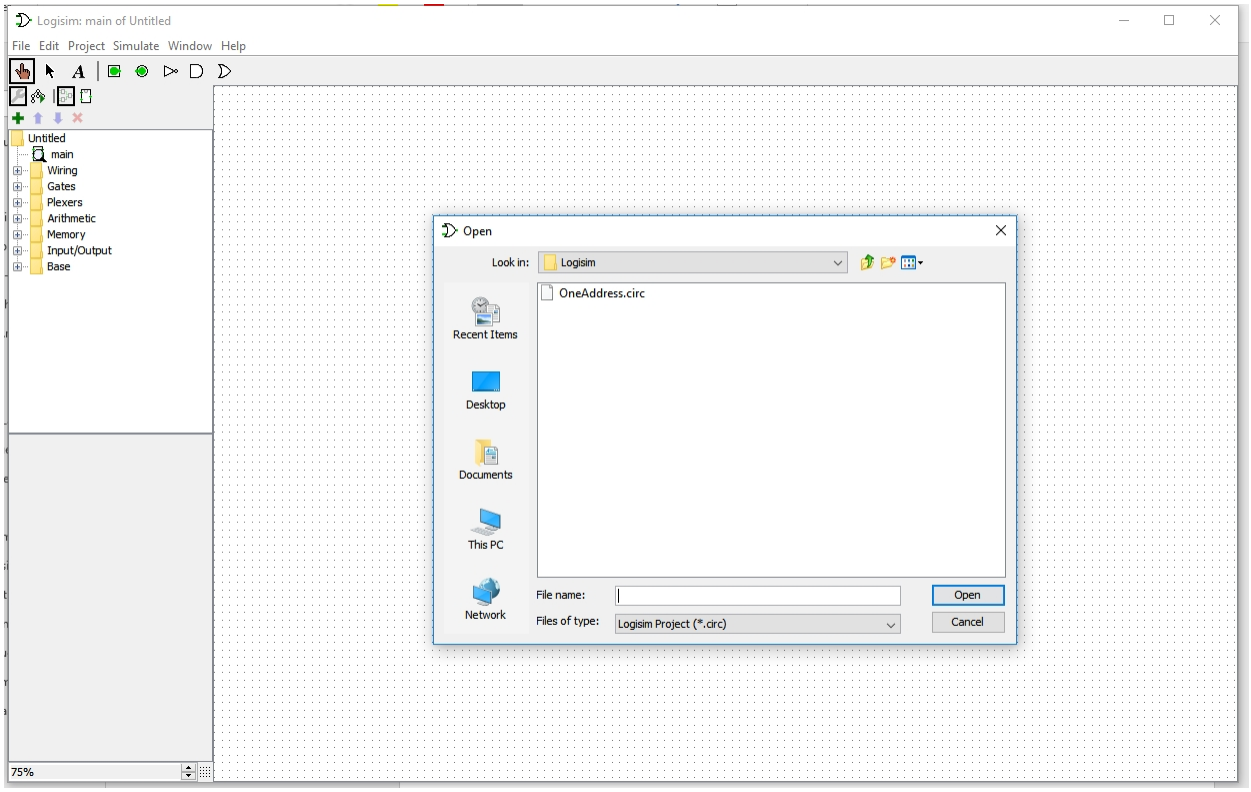


Figura 4-8: Ejecución de la CPU - paso 1

8. Ahora debería ver una pantalla con la CPU de una dirección. Ajuste el tamaño del Zoom (marcado con un círculo en la imagen) al 75% (o al 50% si es necesario) para ver todo el diagrama. Por ahora las únicas dos áreas de la CPU que nos interesan son la Memoria de Texto y la Memoria de Datos, que están marcadas con un círculo en el diagrama.

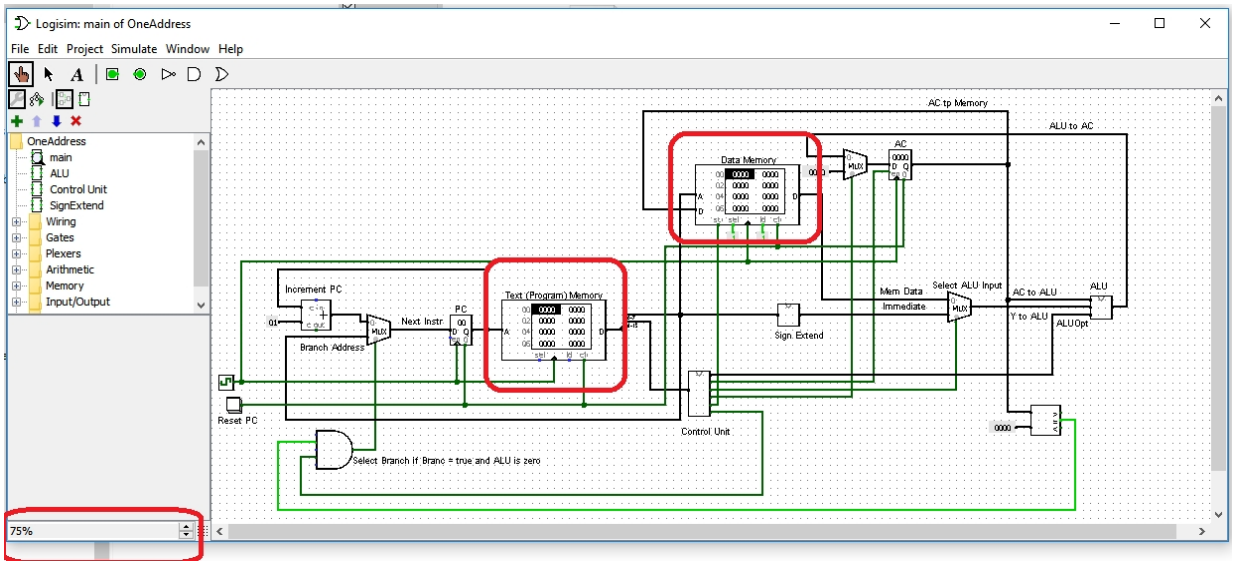


Figura 4-9 Ejecución de la CPU - paso 2

9. Haga clic con el botón derecho del ratón en el cuadro Memoria de texto y seleccione la opción *Editar contenido*. Debería ver aparecer en su pantalla el siguiente editor para la memoria. Tenga en cuenta que los valores de entrada para esta memoria están en hexadecimal, pero vamos a rellenarla a partir de los archivos que fueron creados por el ensamblador, y el ensamblador ha escrito los archivos en hexadecimal.

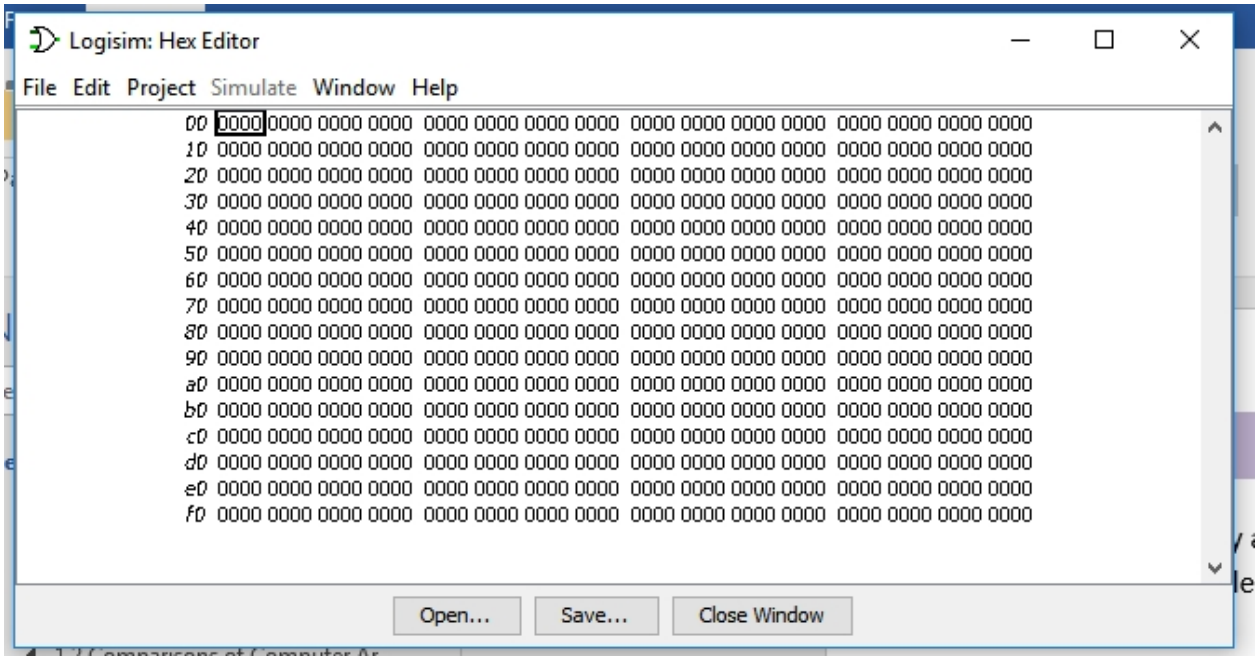


Figura 4-10: Ejecución de la CPU - paso 2

10. Elija la opción Abrir, vaya al directorio AssemblyPrograms y seleccione el archivo AdditionExample.mc.

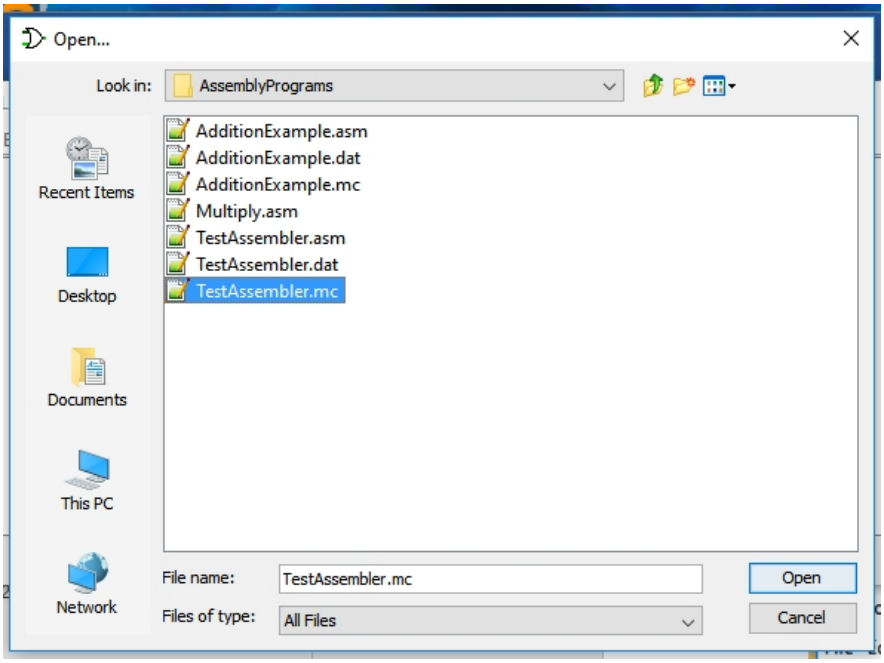


Figura 4-11: Ejecución de la CPU - paso 3

11. Ahora deberías tener el código máquina de AdditionExample en este bloque de datos, como en el siguiente ejemplo. Seleccione la opción cerrar, y el código máquina estará ahora disponible en la CPU.

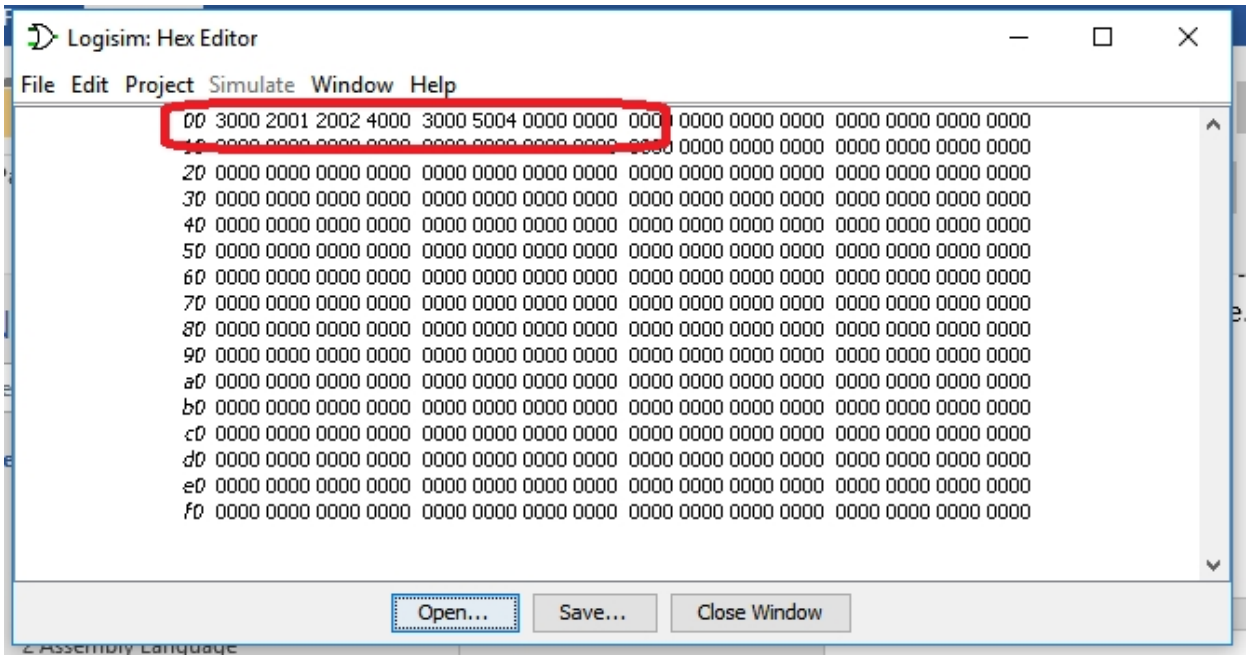


Figura 4-12: Ejecución de la CPU - paso 4

12. Repita los pasos 9-11 para la memoria de datos utilizando el fichero AdditionExample.dat. La memoria de datos y la memoria de programa ya deberían estar inicializadas.

13. Pulsando dos veces sobre el reloj (recuerda que los registros y la memoria sólo cambian en un disparo de flanco positivo, no en un disparo de flanco negativo), y la instrucción clac se ejecutará. Esto no resultará en un cambio porque el $ac ya es cero. Pulsa el reloj dos veces más, y deberías ver que el programa está en la instrucción 2, y el $ac cambia a 5.

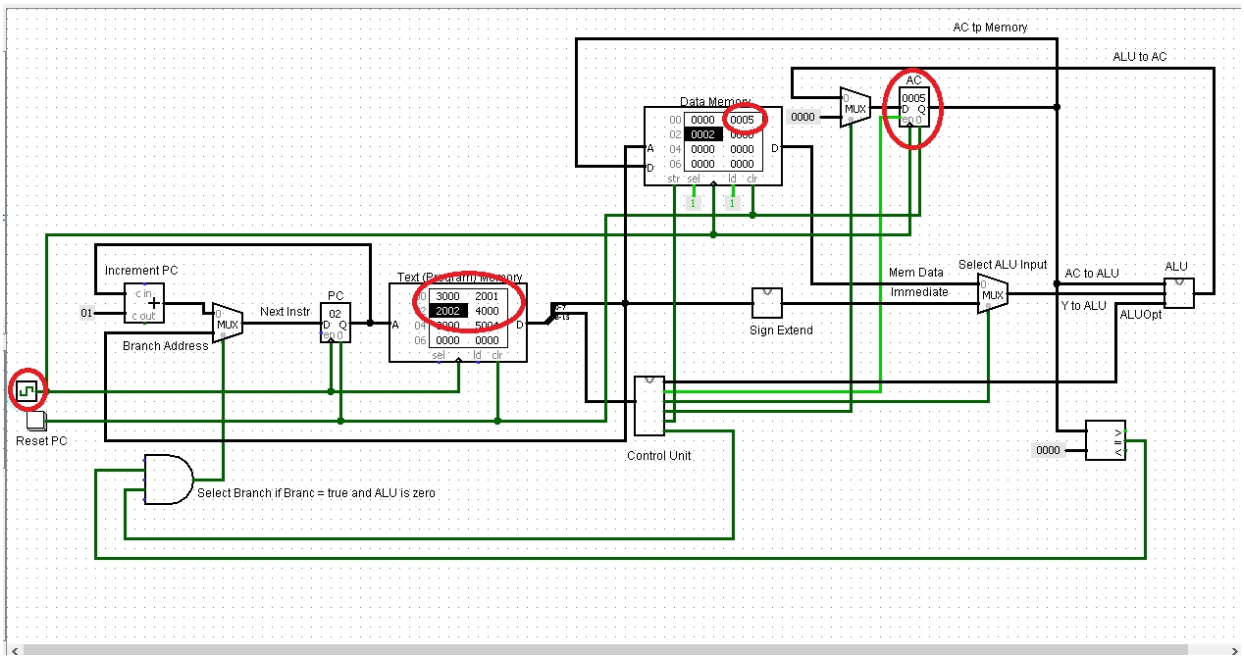


Figura 4-13: Ejecución de la CPU - paso 5

14. Pulsando sobre el reloj 6 veces más se muestra que el programa ha finalizado, y el valor de 7 (5+2) ha sido calculado y almacenado en la dirección de memoria 0. Las dos últimas instrucciones, clac y beqz halt, simplemente ponen el programa en un bucle infinito al final para que no continúe ejecutándose por el resto de la memoria.

En Logisim hay varias formas de controlar la simulación utilizando la opción *Simulación*. Son especialmente útiles la *frecuencia de tick* (cada cuánto tiempo se produce un clic en el reloj), la *activación de tick* (que ejecuta el reloj a la frecuencia de tick) y el *registro* (que proporciona una forma sencilla de revisar los resultados).

5 Implementación de la CPU

Este capítulo cubrirá la implementación Logisim de la CPU One-Address. Esta implementación consistirá en 3 subcomponentes Logisim que son necesarios para implementar la CPU, y el componente principal que es la CPU. Los 3 subcomponentes, la unidad de extensión de signo, la ALU, y la Unidad de Control (CU) se explicarán primero, y luego el componente principal será desglosado y examinado en detalle.

5.1 La unidad de extensión del signo

Los valores inmediatos que pueden formar parte de una instrucción son de 8 bits y pueden utilizarse como entrada para la ALU. Sin embargo, la ALU acepta entradas de 16 bits. Por lo tanto, los valores inmediatos que se pasan a la CPU deben expandirse para llenar 16 bits. La cuestión es cómo rellenar los 8 bits altos cuando se expanden valores inmediatos de 8 a 16 bits.

Recuerde que todos los valores inmediatos pasados a la CPU son enteros; el bit superior (más a la izquierda) del valor determina el signo. Si el número en complemento a 2 es positivo, los 0 a la izquierda no tienen efecto sobre el número. Por ejemplo, $01012 = 0000\ 01012 = 510$. En un número negativo del complemento a 2, los 1 a la izquierda no afectan al número. Así $10112 = 1111\ 10112 = -510$. Para extender un número entero, el bit situado más a la izquierda se extiende a los nuevos dígitos binarios. Esto es lo que hace la unidad de extensión de signo, extendiendo el bit 7^{th} a las posiciones 8-15 para convertir el entero de 8 bits en un entero de 16 bits.

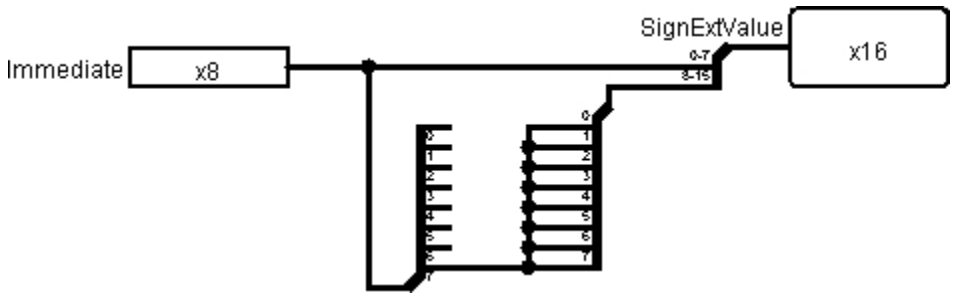


Figura 5-1: La unidad de ampliación de señales

5.2 La ALU

La ALU de esta unidad admite sumas y restas, y también implementa un indicador para saber si la operación actual ha producido un *desbordamiento*. Y el desbordamiento se produce cuando se suman dos números que son demasiado grandes para ser almacenados en los enteros de 16 bits implementados en la CPU. Por ejemplo, sumar 27000 + 25000 = 52000, un valor mayor que el entero máximo que se puede manejar, que es 32767. Del mismo modo -27000 + -25000 = -52000, un número que está muy por debajo del número entero mínimo que se puede manejar, que es -32768. Cómo la ALU maneja estas situaciones se discutirá más adelante.

Una ALU simple que implementaría sólo la suma de 16 bits es fácil de implementar, y se muestra en la siguiente figura. Dos valores de 16 bits ($ac e Y) se envían a la CPU, y un sumador se utiliza para sumar los valores y producir un resultado.

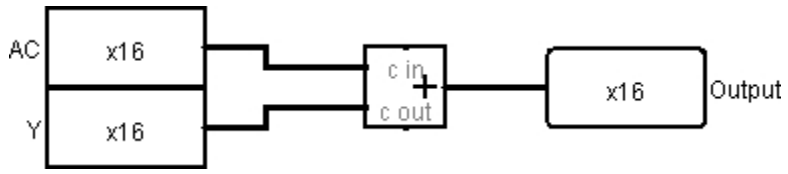


Figura 5-2: Sumador simple

Para crear la sustracción de implementos, se utiliza un poco de matemática creativa.

1. Recuerda que X ⊕ 0 = X; y X ⊕ 1 = X'.
2. La aritmética simple dice que X + Y = X + (-Y)
3. -Y = Y' + 1 (operación de negación en complemento a 2).
4. La sustracción se puede implementar tomando cada bit de Y, XOR con 1 (obteniendo el complemento), a continuación, añadiendo 1. Para sumar 1, pasa este bit al carry in del sumador.
5. La suma se puede implementar tomando cada bit de Y, XORing con 0 (por lo que no cambia) y añadiendo 0. Para añadir 0, pasar este bit en el carry in del sumador.
6. Así, una unidad de suma/resta puede implementarse pasando un bit de bandera. Si el bit es 0, se realiza una operación de suma; si el bit es 1, se realiza una operación de resta.

Este procedimiento se implementa en el siguiente circuito Logisim. Tenga en cuenta que todo lo que hace es XOR los bits Y con el valor de la bandera 0/1, y luego añadir la bandera a la sumadora a través del carry en la sumadora.

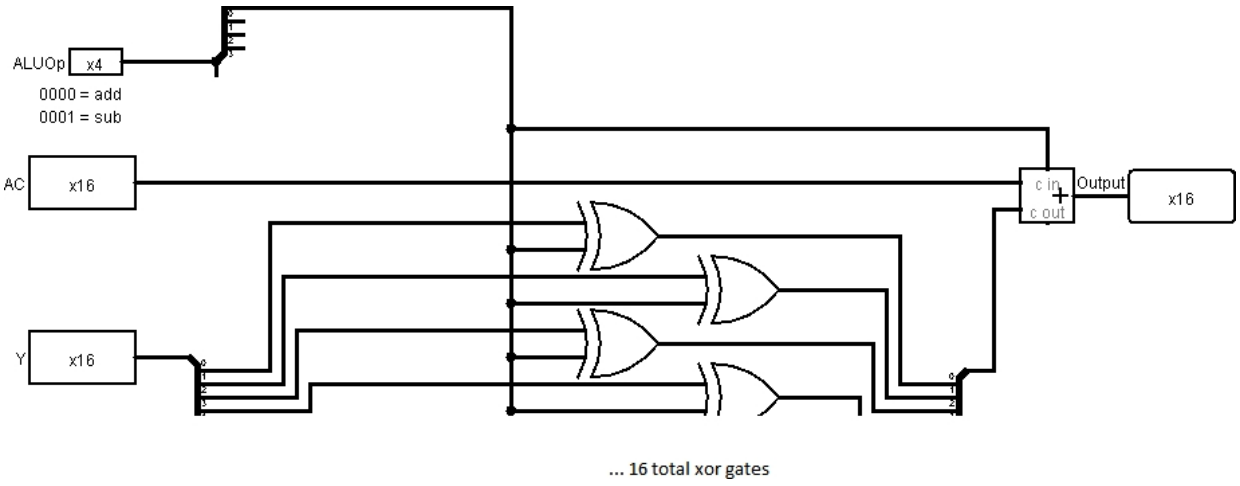


Figura 5-3: Sumador/Restar

Hay una última adición que hacer a la ALU. Queremos comprobar si hay desbordamiento o no. La forma más sencilla de hacerlo es comprobar los bits de entrada y salida al último sumador completo. Si son iguales

iguales, no hay desbordamiento, pero si son diferentes, entonces se ha producido desbordamiento. Estos dos bits pueden comprobarse utilizando una puerta XOR. Si son iguales (no hay desbordamiento), el XOR producirá 0, y si son diferentes (desbordamiento) el XOR producirá 1.

El sumador de Logisim no señala el desbordamiento, por lo que una vez más hay que hacer un uso creativo del circuito. En lugar de utilizar un sumador de 16 bits, la ALU utilizará un sumador de 15 bits y otro de 1 bit. Esto permite comprobar la entrada y salida del último bit, pero requiere una serie de conmutadores para que el número de líneas de cada componente sea correcto. Sin embargo, éste es el único cambio entre la última versión de la ALU y la versión final presentada en la figura siguiente.

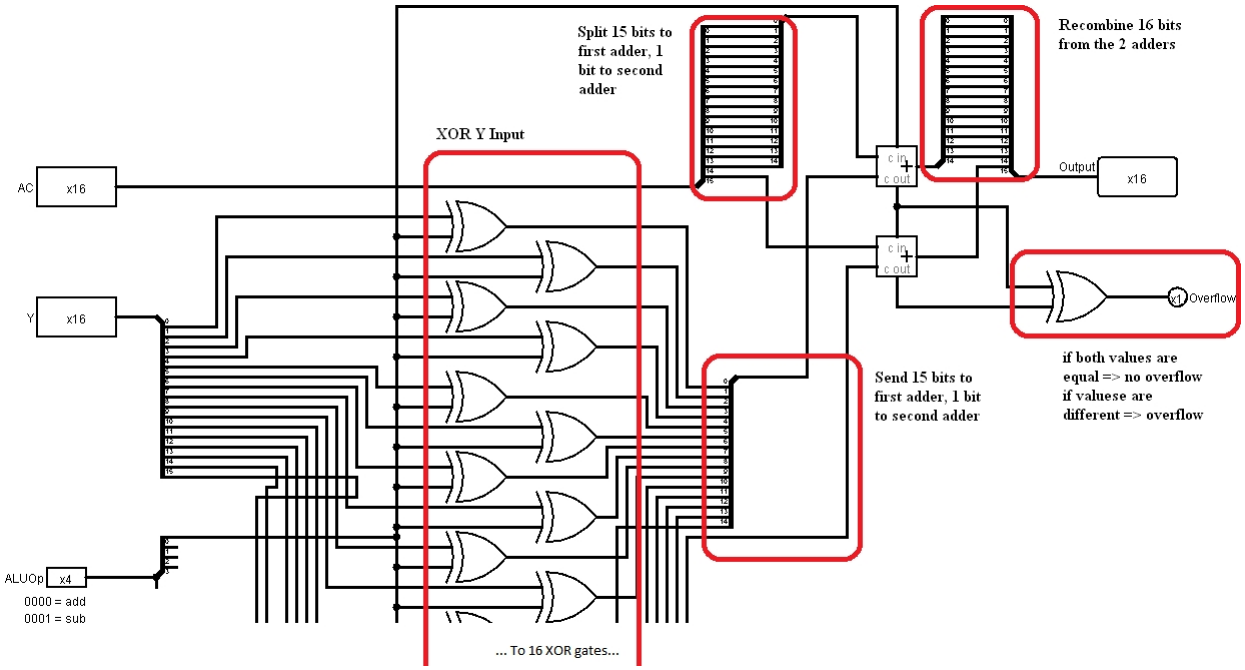


Figura 5-4: Suma/resta con desbordamiento

5.3 La unidad de control (CU)

La CU es el *cerebro* de la CPU. Toma el opcode de la instrucción y establece los cables de control que controlarán cómo procesará la CPU la instrucción. Dado que la configuración de los cables de control sólo tendrá sentido una vez que se entienda el uso de los cables de control, la explicación de la CU se hará después de explicar la CPU.

5.4 La CPU

La CPU reúne todos los componentes en un solo paquete para ejecutar programas. La CPU en este texto consta de dos subsecciones, y fue diseñada a propósito para ser lo suficientemente simple como para que los cables de control (aparte del reloj) para cada subsección de la CPU estén completamente separados el uno del otro.

La primera subsección realiza toda la aritmética y gestiona la entrada/salida a/desde la memoria de datos. Esta subsección utiliza los cables de control Clock, MemWr (escribir memoria), ClrAC (borrar $ac poniéndolo a 0),

WriteAc (escribe un valor en $ac), ALUSrc (elige una fuente para el segundo operando de la ALU, ya sea un valor de memoria o inmediato), y ALUOpt (un valor de 4 bits para especificar qué operación ejecutar en la ALU).

La segunda subsección controla la ruta de ejecución del programa. Utiliza los cables de control Clock y Beqz (branch if equal zero).

Estas dos subsecciones se examinarán por separado.

Antes de empezar, un 0 en un cable de control implica no hacer nada, de ahí que una instrucción *noop* sea un 0x0000 de 2 bytes. Si un cable no está siendo utilizado, por defecto se pone a 0.

5.4.1 La CPU - Subsección Aritmética

La subsección aritmética de la CPU cubre el registro $ac, la ALU y la memoria de datos. Esto cubrirá las operaciones en ensamblador clac, add, addi, sub, subi y stor. La subsección aritmética se muestra en el siguiente diagrama.

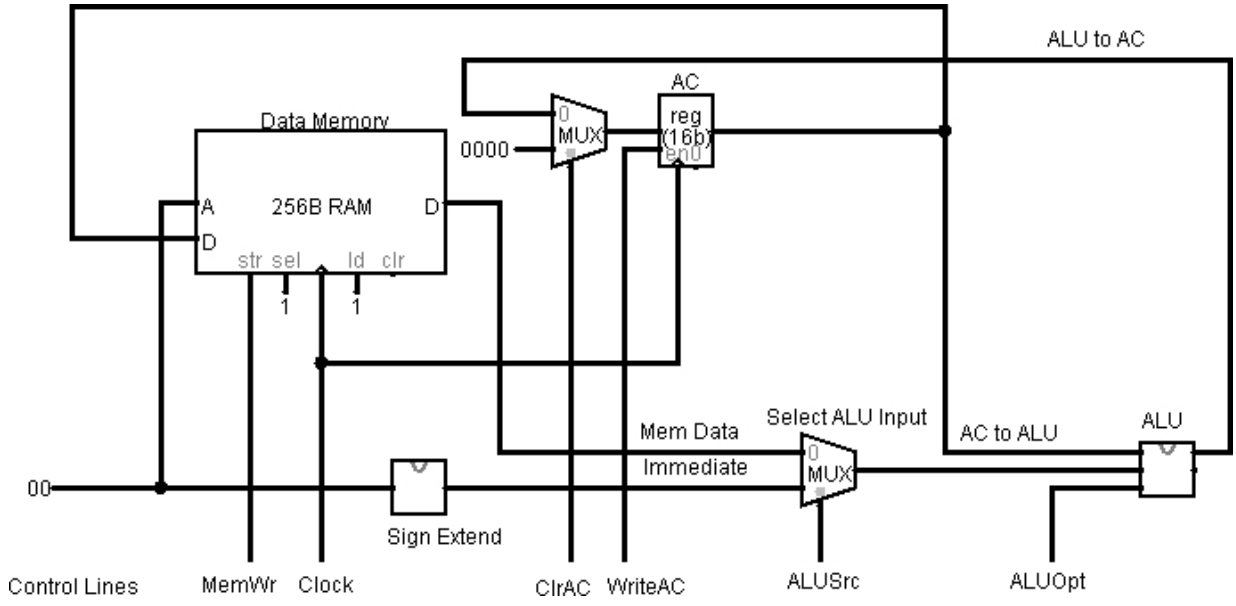


Figura 5-5: CPU - Subsección Aritmética

La operación clac selecciona la entrada constante 0x0000 usando el mux delante del $ac para fijar el valor del $ac. Para ello, la línea ClrAC al mux es 1 (seleccionando 0x0000) y la línea WriteAC es 1. Todas las demás líneas de control son 0.

Para las operaciones add y sub, la entrada a la ALU son datos de memoria, por lo que la línea ALUSrc se pone a 0 para seleccionar los Datos de Memoria. El resultado se almacena de nuevo en $ac, por lo que la línea ClrAC debe ponerse a 0 para seleccionar la salida de la ALU, y la línea WriteAC se pone a 1 para escribir el resultado de la ALU en la AC. La línea ALUOpt se pone a 0000 para sumar y a 0001 para restar. Todas las demás líneas de control se ponen a 0.

Para las operaciones add y subi, la entrada a la ALU es el valor inmediato, por lo que la línea ALUSrc se pone a 1 para seleccionar el valor inmediato de los datos. Todas las demás líneas se ajustan como en las operaciones add y sub.

Para la operación stor, la MemWr se pone a 1, lo que escribe el valor en el puerto D de la Memoria de Datos en la memoria en la dirección especificada en el puerto A (nótese que la dirección proviene de la parte inmediata de la instrucción). Todas las demás líneas de control se ponen a 0.

A algunos lectores podría preocuparles que haya valores pasados en la CPU que no se utilicen. Por ejemplo, cuando se está ejecutando la operación stor se sigue calculando un valor en la ALU, y se envía por el cable al $ac. Sin embargo, la línea WriteAC es 0, por lo que el valor de la ALU no tiene efecto, y es ignorado. Lo mismo ocurre con el valor Mem Data para operaciones inmediatas como addi. El Mem Data se genera, pero se ignora ya que el ALUSrc elige el valor inmediato. Muchas líneas se establecen en cada instrucción, y la mayoría de ellas son ignoradas, que es por lo que usando poniendo todos los cables de control a 0 una instrucción NOOP no hace nada.

5.4.2 Subsección CPU - Ruta de ejecución

La segunda subsección de la CPU es la ruta de ejecución, que se muestra en la siguiente figura.

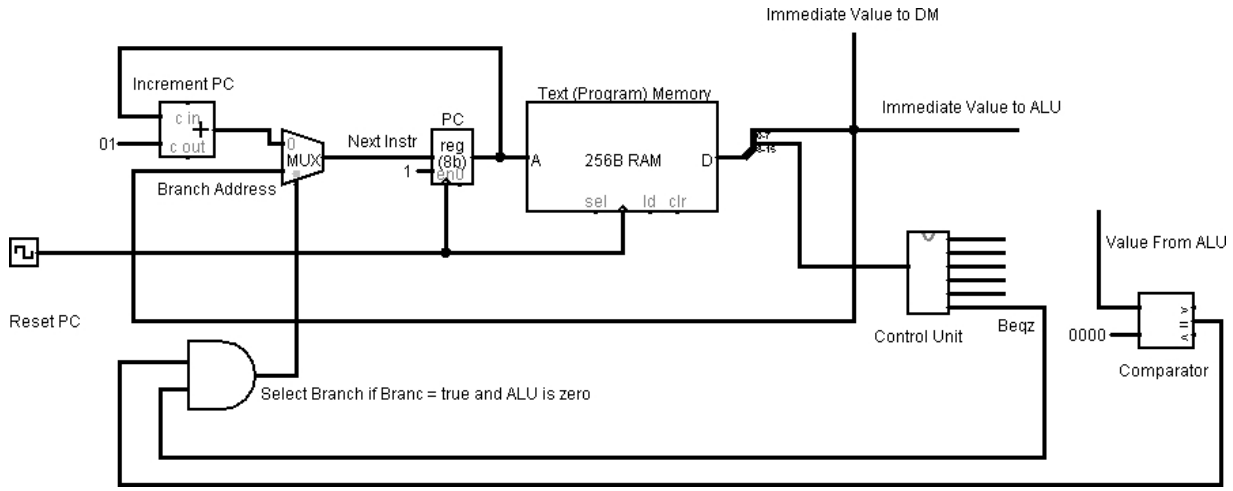


Figura 5-6: Subsección CPU - Ruta de ejecución

Como muestra esta figura, el valor del registro $pc se utiliza para establecer la dirección de lectura de la instrucción. A continuación, esta instrucción se divide en la mitad de control (bits 8-15) y el valor inmediato (bits 0-7). Los bits de control se envían a la CU para establecer los hilos de control, y el valor inmediato se envía a la memoria de datos o a la ALU para su uso en la subsección aritmética de la CPU.

Cada vez que se ejecuta una instrucción, el registro $pc se cambia para apuntar a la siguiente instrucción a ejecutar. Cuando el programa se ejecuta secuencialmente, la siguiente instrucción en memoria se selecciona sumando 1 al $pc en el nombre del sumador Incrementar PC, y el multiplexor se pone a 0 para seleccionar esta instrucción.

La única vez que no se selecciona la siguiente instrucción es si el cable Beqz está alto (lo que significa que se trata de una instrucción beqz), Y los resultados del compactador son 1 (la ALU tiene un valor de 0). Cuando esto ocurre, la

mux selecciona la dirección de ramificación, que es el valor inmediato de la instrucción, y el programa continúa ejecutándose en la instrucción en la nueva dirección.

5.5 Implantación de la UC

Ahora es posible especificar cómo configurar los cables de control desde la CU. En primer lugar, la ALU opt es el valor de los bits 8-11 de la ALU, por lo que estos se dividen y se envían para controlar la ALU.

Los 4 bits superiores, bits 12-15, se utilizan para ajustar los otros cables de control, y de la discusión anterior se puede establecer de acuerdo con la siguiente tabla[2] .

Operación	Código	EscribirAc	ALUSrc	ClrAc	MemWr	Beqz
Inmediato Operación[3]	0x1	1	1	0	0	0
Memoria Operación[4]	0x2	1	0	0	0	0
clac	0x3	1	x	1	0	0
stor	0x4	0	x	x	1	0
beqz	0x5	0	x	x	0	1

Tabla 5-1: Cables de operación y control

Para implementar esta tabla, se implementa un decodificador para dividir las operaciones individuales. A continuación, estas operaciones se combinan para producir el comportamiento de salida correcto. La CU se muestra en la siguiente figura.

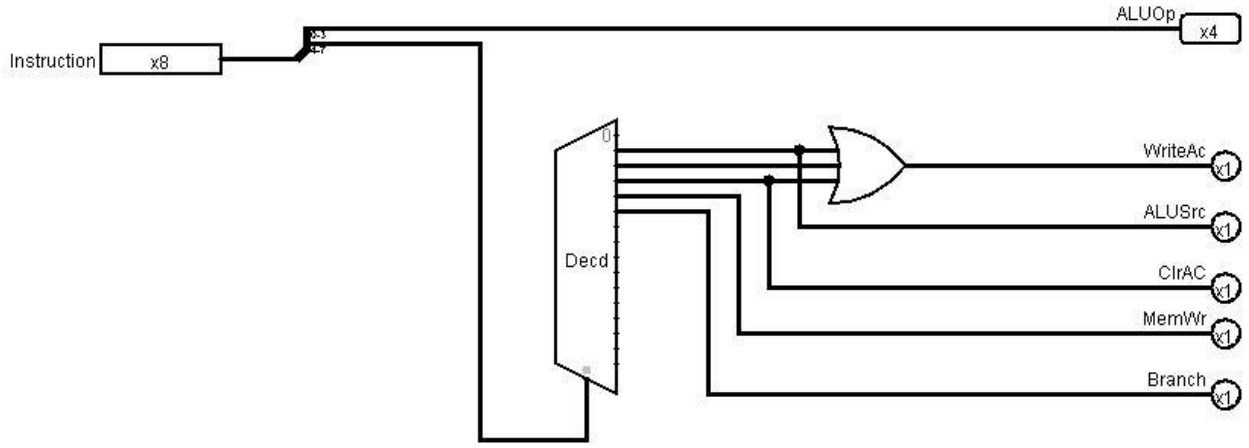


Figura 5-7: Unidad de control

[2] Una "x" en la tabla significa una condición "*don't care*", es decir, el valor puede ser 0 o 1, ya que no afecta al funcionamiento de la CPU. Por convención, todos los valores x deben codificarse como 0.
[3] addi, subi, etc.
[4] sumar, restar, etc.