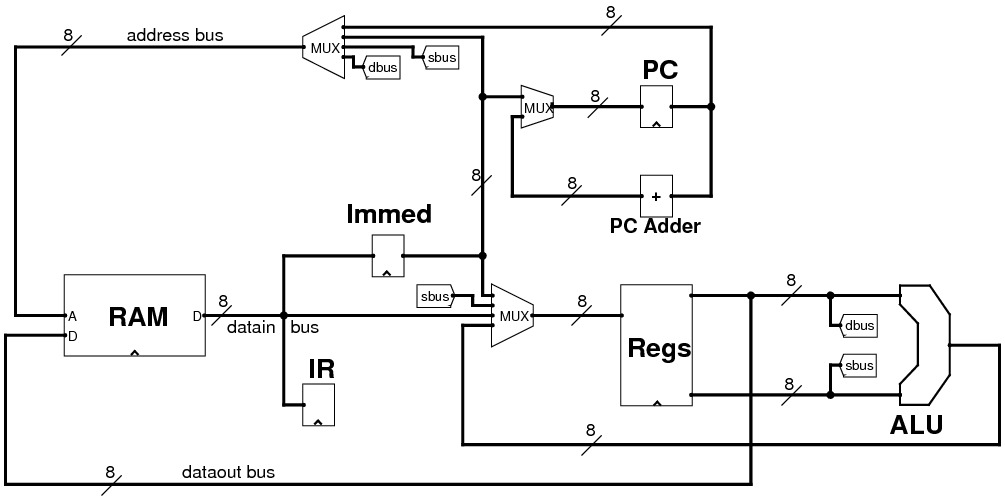
**Un ejemplo de CPU cableada**

**1   Introducción**

* En el tutorial de esta semana, veremos una CPU cableada simple, para demostrar que la lógica de control en una CPU puede construirse utilizando algunas puertas y multiplexores simples.
* El ejemplo de CPU es uno que diseñé e implementé en Logisim durante aproximadamente una semana a fines de 2010. El objetivo era crear una arquitectura razonablemente tradicional pero con la menor lógica de control posible.

**2   arquitectura**

* La CPU tiene un bus de datos de 8 bits y un bus de direcciones de 8 bits, por lo que solo puede admitir 256 bytes de memoria para almacenar tanto instrucciones como datos.
* Internamente, hay cuatro registros de 8 bits, R0 a R3, más un registro de instrucciones, el contador del programa y un registro de 8 bits que contiene valores inmediatos.
* La ALU es la misma que diseñamos la semana pasada. Realiza las cuatro operaciones AND, OR, ADD y SUB en dos valores de 8 bits, y admite ADD y SUB firmados.
* La CPU es una arquitectura de carga / almacenamiento: los datos deben incorporarse a los registros para su manipulación, ya que la ALU solo lee y escribe en los registros.
* Las operaciones de ALU tienen dos operandos: un registro es un registro de origen, y el segundo registro es el registro de origen y de destino, es decir, registro de destino = registro de destino registro de fuente OP.
* Todas las operaciones de salto realizan saltos absolutos; No hay ramas relativas a PC. Hay saltos condicionales basados ​​en el zeroness o negatividad del registro de destino, así como una instrucción de "saltar siempre".
* El siguiente diagrama muestra las rutas de datos en la CPU:



* Las etiquetas *dbus* y *sbus* indican las líneas que salen del archivo de registro que contienen el valor de los registros de destino y fuente.
* Tenga en cuenta el bucle de datos que involucra los registros y la ALU, cuya salida solo puede volver a un registro.
* El bus de datos solo está conectado a la línea *dbus* , por lo que el único valor que se puede escribir en la memoria es el registro de destino.
* También tenga en cuenta que solo hay 3 multiplexores:
  + el multiplexor de bus de dirección puede obtener una dirección de memoria de la PC, el registro inmediato (para direccionamiento directo) o de los registros de origen o destino (para direccionamiento indirecto de registro).
  + el multiplexor de PC permite al PC incrementar o saltar al valor en el registro inmediato.
  + El multiplexor delante de los registros determina de dónde proviene una escritura de registro: la ALU, el registro inmediato, otro registro o el bus de datos.

**Set de 3 instrucciones**

* La mitad de las instrucciones en el conjunto de instrucciones encajan en un byte:

|  |  |  |  |
| --- | --- | --- | --- |
| op1 | op2 | Rd | Rs |
| 2 | 2 | 2 | 2 |

* Estas instrucciones se identifican con un 0 en el bit más significativo de la instrucción, es decir, *op1* = *0X* .
* Los 4 bits de opcode se dividen en *op1* y *op2* : más detalles pronto.
* *Rd* es el registro de destino y *Rs* es el registro de origen.
* La otra mitad del conjunto de instrucciones son instrucciones de dos bytes. El primer byte tiene el mismo formato que el anterior y le sigue una constante de 8 bits o un valor inmediato:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| op1 | op2 | Rd | Rs | inmediato |
| 2 | 2 | 2 | 2 | 8 |

* Estas instrucciones de dos bytes se identifican con un 1 en el bit más significativo de la instrucción, es decir, *op1* = 1X.
* Con 4 bits de operación, hay 16 instrucciones:

|  |  |  |  |
| --- | --- | --- | --- |
| **op1** | **op2** | **Mnemotécnico** | **Propósito** |
|  |  |  |  |
| 00 | 00 | Y Rd, Rs | Rd = Rd y Rs |
| 00 | 01 | O Rd, Rs | Rd = Rd O Rs |
| 00 | 10 | ADD Rd, Rs | Rd = Rd + Rs |
| 00 | 11 | SUB Rd, Rs | Rd = Rd - Rs |
| 01 | 00 | LW Rd, (Rs) | Rd = Mem [Rs] |
| 01 | 01 | SW Rd, (Rs) | Mem [Rs] = Rd |
| 01 | 10 | MOV Rd, Rs | Rd = Rs |
| 01 | 11 | NOP | Hacer nada |
| 10 | 00 | JEQ Rd, immed | PC = immed si Rd == 0 |
| 10 | 01 | JNE Rd, immed | PC = immed si Rd! = 0 |
| 10 | 10 | JGT Rd, immed | PC = immed si Rd> 0 |
| 10 | 11 | JLT Rd, immed | PC = immed si Rd <0 |
| 11 | 00 | LW Rd, immed | Rd = Mem [immed] |
| 11 | 01 | SW Rd, immed | Mem [immed] = Rd |
| 11 | 10 | LI Rd, immed | Rd = immed |
| 11 | 11 | JMP immed | PC = immed |

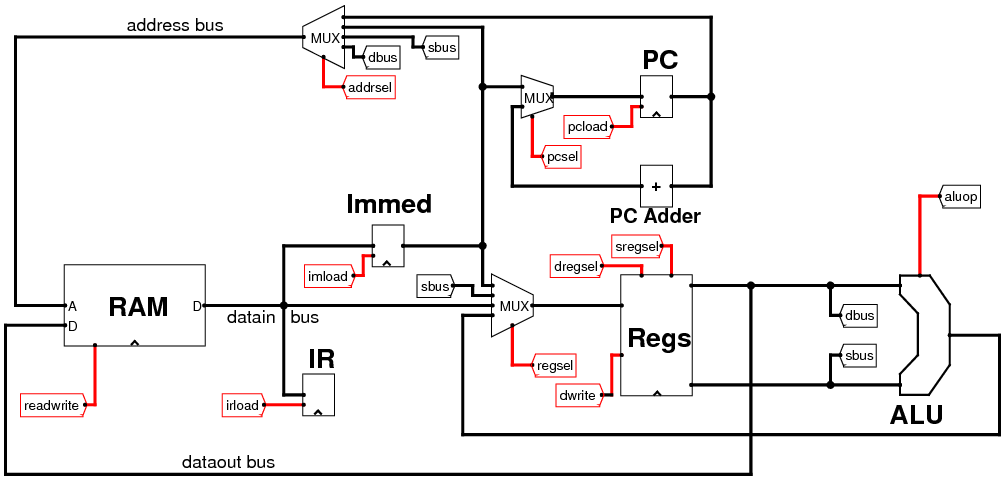
* Tenga en cuenta la regularidad de las operaciones de ALU y las operaciones de salto: podemos alimentar los bits *op2* directamente en la ALU y usar *op2* para controlar la decisión de bifurcación.
* El resto del conjunto de instrucciones es menos regular, lo que requerirá una decodificación especial para algunas de las 16 instrucciones.

**4   fases de instrucciones**

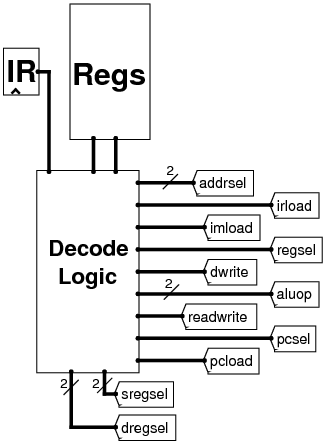
* La CPU tiene internamente tres fases para la ejecución de cada instrucción.
* En la fase 0, la instrucción se recupera de la memoria y se almacena en el Registro de instrucciones.
* En la fase 1, si la instrucción recuperada es una instrucción de dos bytes, el segundo byte se recupera de la memoria y se almacena en el Registro Inmediato. Para instrucciones de un byte, nada ocurre en la fase 1.
* En la fase 2, todo lo demás se realiza según sea necesario, que puede incluir:
  + Una operación ALU, leyendo de dos registros.
  + Una decisión de salto que actualiza la PC.
  + un registro de escritura.
  + una lectura desde una ubicación de memoria.
  + una escritura en una ubicación de memoria.
* Después de la fase 2, la CPU inicia la siguiente instrucción en la fase 0.
* La lógica de control será simple para el trabajo de la fase 0, no difícil para el trabajo de la fase 1, pero complicada para el trabajo de la fase 2.

**5   líneas de control de CPU**

* A continuación se muestra nuevamente el diagrama de la CPU principal, esta vez con las líneas de control mostradas.

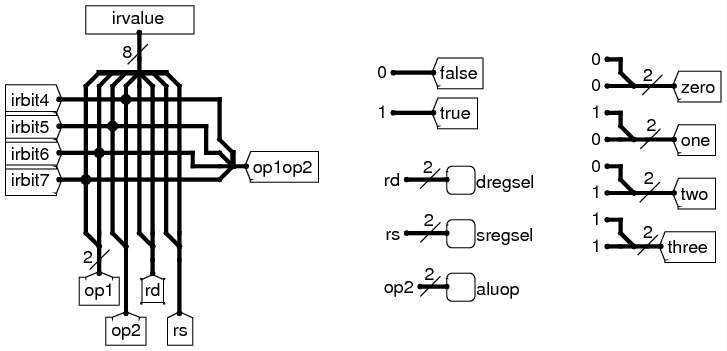


* Hay varias líneas de control de 1 bit:
  + *pcsel* , incrementa PC o carga el valor de salto del Registro Inmediato.
  + *cargue* , carga la PC con un nuevo valor, o no carga un nuevo valor.
  + *cargue* , carga el registro de instrucciones con una nueva instrucción.
  + *imload* , carga el registro inmediato con un nuevo valor.
  + *lectura-escritura* , lectura de la memoria, o escribir en la memoria.
  + *escriba* , *vuelva a* escribir un valor en un registro o no escriba un valor.
* También hay varias líneas de control de 2 bits:
  + *addrsel* , seleccione una dirección de la PC, el registro inmediato, el registro de origen o el registro de destino.
  + *Regsele* , seleccione un valor para escribir en un registro desde el Registro inmediato, otro registro, el bus de datos o desde la ALU.
  + *dregsel* y *sregsel* , seleccione dos registros cuyos valores se envíen a la ALU.
  + *aluop* , que son los bits *op2* que controlan el funcionamiento de la ALU.
* Los valores para todas estas líneas de control son generados por la lógica de decodificación, que obtiene como entrada el valor del registro de instrucciones, y las líneas cero y negativas del registro de destino.

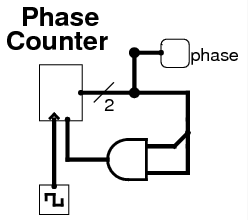


* Ahora es el momento de mirar dentro de la lógica de decodificación para ver cómo crea los valores en las líneas de control.

**6   Dentro de la lógica de decodificación**



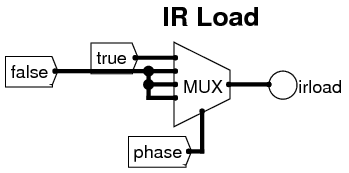
* Dentro del bloque de lógica de decodificación, el valor del registro de instrucciones se divide en varias líneas individuales *irbit4* , *irbit5* , *irbit6* e *irbit7* . *op1* y *op2* se dividen, con *op2* exportado como *aluop* . Finalmente, los 4 bits de código de operación de la instrucción se dividen como la línea *op1op2* .
* Varios de los bits del valor del registro de instrucciones se conectan directamente a estas salidas de 2 bits: *dregsel* , *sregsel* y *aluop* .
* El diagrama parece un poco feo, ya que esta es la forma en que Logisim divide los grupos de líneas en líneas individuales.
* En la implementación de Logisim de la CPU, hay dos líneas "constantes" de 1 bit definidas: *verdadera* y *falsa* , así como varias líneas de 2 bits: *cero* , *una* , *dos* y *tres* .
* También necesitamos algo de lógica para generar la fase actual de ejecución. Esto se hace con un simple contador de 2 bits que está controlado por el ciclo del reloj y que genera una línea de *fase de* 2 bits .



* No necesitamos la fase 3, por lo que los dos bits de la línea de fase pueden ser ANDed. Cuando ambos son verdaderos, esto restablece el contador a cero.
* Para el resto de la lógica de decodificación, debemos observar qué se debe realizar para las distintas fases de la CPU y también qué se debe realizar para cada instrucción específica.

**6.1   fase cero**

* En la fase cero, el valor del PC tiene que ser colocado en el bus de direcciones, por lo que el *addrsel* línea debe ser 0. El *irload* línea tiene que ser 1 de manera que la IR se carga desde el *DATAIN* bus. Finalmente, la PC debe incrementarse en caso de que necesitemos obtener un valor inmediato en la fase 1.
* Todo esto se puede hacer utilizando multiplexores que emiten diferentes valores dependiendo de la fase actual. Aquí está la lógica de control para la línea de *carga* .



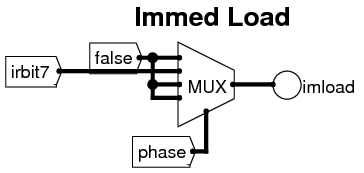
* Solo necesitamos cargar el IR en la fase 0, por lo que podemos conectar verdadero a la entrada 0 del multiplexor de *carga* y falso a las otras entradas. **Nota: la** entrada 11 (es decir, el decimal 3) al multiplexor nunca se usa, ya que nunca llegamos a la fase 3, pero Logisim quiere que todas las entradas del multiplexor sean válidas.
* Otra forma de ver cada fase es el valor que se debe establecer para cada línea de control, para cada instrucción.
* Para la fase cero, estos valores de línea de control se pueden configurar para todas las instrucciones:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **op1** | **op2** | **Instruir** | **pcsel** | **pcload** | **cargar** | **imload** | **rw** | **escribir** | **jumpsel** | **addrsel** | **regsel** | **hez** | **sreg** | **aluop** |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| xx | xx | todos | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | X | X | X | X |

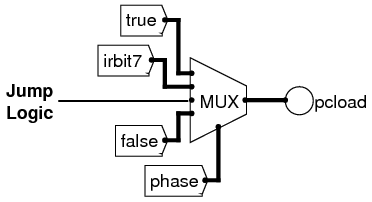
* 'x' representa cualquier valor, es decir, acepta cualquier valor de código de operación, genera cualquier valor de línea de control.

**6.2   Fase uno**

* En la fase 1, debemos cargar el registro inmediato con un valor de la memoria si el *irbit7* del IR es verdadero. El valor de la PC tiene que ser colocado en el bus de direcciones, por lo que el *addrsel*línea debe ser 0. El *IMLOAD* línea tiene que ser 1, de modo que el Registro inmediata se carga desde el *DATAIN* autobús. Finalmente, la PC debe incrementarse para que estemos listos para buscar la próxima instrucción en la siguiente fase 0.



* La lógica *imload* se muestra arriba. Es muy similar a la lógica de *carga* , pero esta vez se *genera* un valor de habilitación solo en la fase 1, y solo si se configura *irbit7* .



* Parte de la lógica de *pcload* se muestra arriba. La PC siempre se incrementa en la fase 0. Se incrementa en la fase 1 si se establece *irbit7* , es decir, una instrucción de dos bytes. Finalmente, la PC puede cargarse con un valor inmediato en la fase 2 si estamos realizando una instrucción de salto y la prueba de salto es verdadera. Volveremos a la lógica del salto más tarde.
* Podemos tabular los valores de las líneas de control para la fase 1. Esta vez, lo que se *genera* depende del bit superior del valor *op1* :

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **op1** | **op2** | **Instruir** | **pcsel** | **pcload** | **cargar** | **imload** | **rw** | **escribir** | **jumpsel** | **addrsel** | **regsel** | **hez** | **sreg** | **aluop** |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 0x | xx | todos | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | X | X | X | X |
| 1x | xx | todos | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | X | X | X | X |

**6.3   Fase dos**

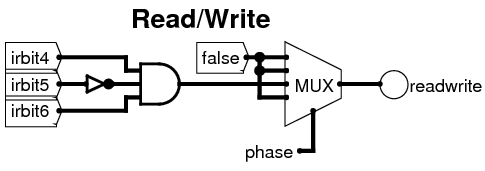
* Aquí, los valores de las líneas de control dependen en gran medida de la instrucción específica que estamos realizando. Aquí está la tabla de salidas de línea de control dependiendo de la instrucción:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **op1** | **op2** | **Instruir** | **pcsel** | **pcload** | **cargar** | **imload** | **rw** | **escribir** | **addrsel** | **regsel** | **hez** | **sreg** | **aluop** |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 00 | 00 | Y Rd, Rs | X | 0 | 0 | 0 | 0 | 1 | X | 3 | Rd | Rs | op2 |
| 00 | 01 | O Rd, Rs | X | 0 | 0 | 0 | 0 | 1 | X | 3 | Rd | Rs | op2 |
| 00 | 10 | ADD Rd, Rs | X | 0 | 0 | 0 | 0 | 1 | X | 3 | Rd | Rs | op2 |
| 00 | 11 | SUB Rd, Rs | X | 0 | 0 | 0 | 0 | 1 | X | 3 | Rd | Rs | op2 |
| 01 | 00 | LW Rd, (Rs) | X | 0 | 0 | 0 | 0 | 1 | 2 | 2 | Rd | Rs | X |
| 01 | 01 | SW Rd, (Rs) | X | 0 | 0 | 0 | 1 | 0 | 3 | X | Rd | Rs | X |
| 01 | 10 | MOV Rd, Rs | X | 0 | 0 | 0 | 0 | 1 | X | 1 | Rd | Rs | X |
| 01 | 11 | NOP | X | 0 | 0 | 0 | 0 | 0 | X | X | X | X | X |
| 10 | 00 | JEQ Rd, immed | 0 | j | 0 | 0 | 0 | 0 | X | X | Rd | X | op2 |
| 10 | 01 | JNE Rd, immed | 0 | j | 0 | 0 | 0 | 0 | X | X | Rd | X | op2 |
| 10 | 10 | JGT Rd, immed | 0 | j | 0 | 0 | 0 | 0 | X | X | Rd | X | op2 |
| 10 | 11 | JLT Rd, immed | 0 | j | 0 | 0 | 0 | 0 | X | X | Rd | X | op2 |
| 11 | 00 | LW Rd, immed | X | 0 | 0 | 0 | 0 | 1 | 1 | 2 | Rd | X | X |
| 11 | 01 | SW Rd, immed | X | 0 | 0 | 0 | 1 | 0 | 1 | X | Rd | X | X |
| 11 | 10 | LI Rd, immed | X | 0 | 0 | 0 | 0 | 1 | X | 0 | Rd | X | X |
| 11 | 11 | JMP immed | 0 | 1 | 0 | 0 | 0 | 0 | X | X | X | X | X |

* Para hacer que la lógica de la línea de control sea lo más simple posible, un diseñador de CPU siempre se esfuerza por la regularidad. Sin embargo, esto a menudo está en conflicto con la funcionalidad de la CPU deseada.
* De la tabla anterior, las instrucciones de ALU ( *op1* = 00) y las instrucciones de salto ( *op1* = 10) son agradables y regulares. Todas las instrucciones *op1* = 1x utilizan el registro inmediato, mientras que las instrucciones *op1* = 0x no lo hacen.
* Siempre podemos vincular *dregsel* a *Rd a* partir de la instrucción, y lo mismo ocurre con *sregsel* = *Rs* y *aluop* = *op2* . E *irload* e *imload* siempre son 0 para la fase 2.
* Con las restantes líneas de control, cesan las regularidades.

**6.4   Lógica de lectura / escritura**

* La línea de lectura / escritura en la memoria solo debe habilitarse cuando estamos realizando operaciones de SW (store word), y solo en la fase 2.
* Los valores *op1op2* para las dos instrucciones de SW son 0101 y 1101, por lo que podemos tratar esto como x101, y establecer *readwrite* true cuando *irbit6* está *activado* , *irbit5* está desactivado e *irbit4*está *activado* .

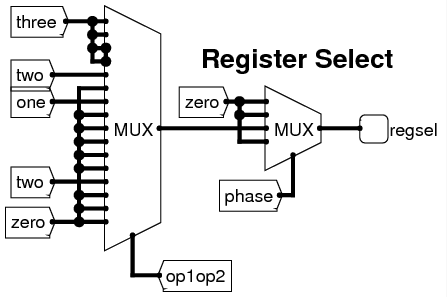


**6.5   Registrar la lógica de selección**

* Ahora llegamos a las líneas de control que están desordenadas. La primera es la línea *regsel* , que selecciona la entrada que se escribirá en el registro de destino. Esto puede ser:

|  |  |
| --- | --- |
| 00 (cero) | Registro inmediato |
| 01 (uno) | *sbus* , es decir, el registro fuente |
| 10 (dos) | bus de datos |
| 11 (tres) | Salida ALU |

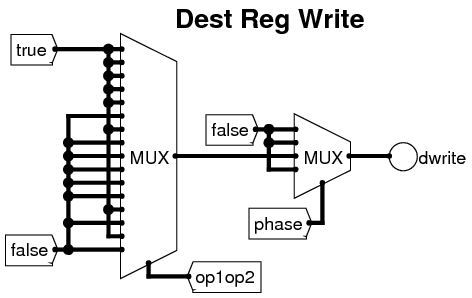
* Al consultar la tabla grande en la subsección de la Fase Dos, todas las instrucciones de ALU establecen *regsel* en 3, pero, aparte de eso, no existe una regla lógica simple para generar todos los valores posibles.
* Además, para los *valores* de *regsel* 'x' en la tabla, podemos optar por generar cualquier valor, ya que el registro no se cargará en estas instrucciones.
* La solución más sencilla aquí es usar un multiplexor para la fase de operación, y un segundo multiplexor para el código de operación de la instrucción, es decir, *op1op2* se usa para seleccionar el valor de salida.



* Cada una de las 16 entradas al multiplexor grande establece un valor de *regsel* para una instrucción específica basada en el valor *op1op2* , y esto solo se elimina durante la fase 2. De lo contrario, *regsel*se establece en cero.

**6.6   Destino Registro Escribir Lógica**

* Siguiendo con *Regsel* , necesitamos controlar el modo de *escritura* , es decir, cuando se escribe el registro de destino.
* Esto puede ocurrir solo en la fase 2, y nuevamente, no hay una regla simple que nos permita cablear el valor con puertas simples: vea la columna *dwrite* en la tabla grande de arriba.
* Nuevamente, podemos usar un multiplexor de 16 entradas, con *op1op2* para elegir el valor de *dwrite* correcto para la salida en la fase 2.

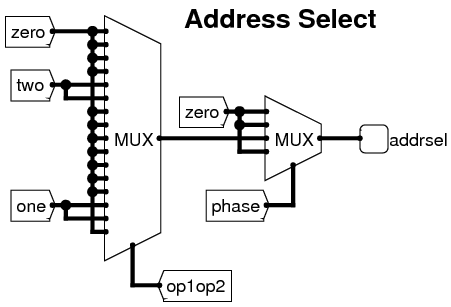


**6.7   Lógica de selección de dirección**

* Cuando queremos leer datos de la memoria principal, la dirección de la que queremos leer se puede seleccionar desde estas entradas:

|  |  |
| --- | --- |
| 00 (cero) | Contador de programa |
| 01 (uno) | Registro inmediato |
| 10 (dos) | *sbus* , es decir, el registro fuente |
| 11 (tres) | *dbus* , es decir, el registro fuente |

* La lógica de decodificación debe generar un valor para *addrsel* que seleccione la dirección correcta para afirmar en el bus de direcciones para cada instrucción durante la fase 0, la fase 1 y la fase 2.
* Al igual que con las dos líneas de control anteriores, no existe una lógica simple para producir el valor en esta línea basado en el código de operación de la instrucción, por lo que recurrimos nuevamente a un multiplexor de 16 entradas.



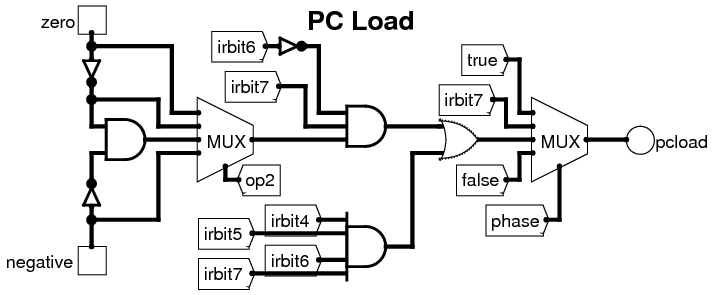
* Verifique el diagrama lógico anterior con la tabla grande de arriba para asegurarse de que producirá la salida correcta para *addrsel,* dado el valor *op1op2* de la instrucción.

**6.8   Jump Logic**

* Hemos llegado a la última y probablemente la línea de control más complicada de la CPU, *pcload* , que determina cuándo se actualiza el contador de programas.
* En realidad, no es tan complicado, solo se ve feo. Usted ha visto la lógica para la fase 0 y la fase 1. Ahora debemos observar la lógica para las instrucciones de salto.
* Una de las instrucciones de salto, *op1op2* = 1111, siempre establece la PC en el Registro Inmediato.
* Las otras instrucciones de salto solo configuran la PC en Registro inmediato cuando una prueba específica es verdadera:

|  |  |  |  |
| --- | --- | --- | --- |
| **op1op2** | **Prueba** | **Cero** | **Negativo** |
|  |  |  |  |
| 1000 (JEQ) | Rd == 0 | 1 | X |
| 1001 (JNE) | Rd! = 0 | 0 | X |
| 1010 (JGT) | Rd> 0 | 0 | 0 |
| 1011 (JLT) | Rd <0 | X | 1 |

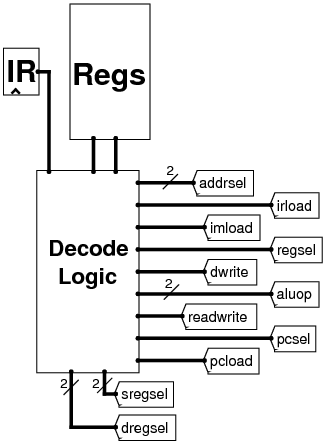
* Aquí está la lógica para configurar *pcload* para todas las fases:



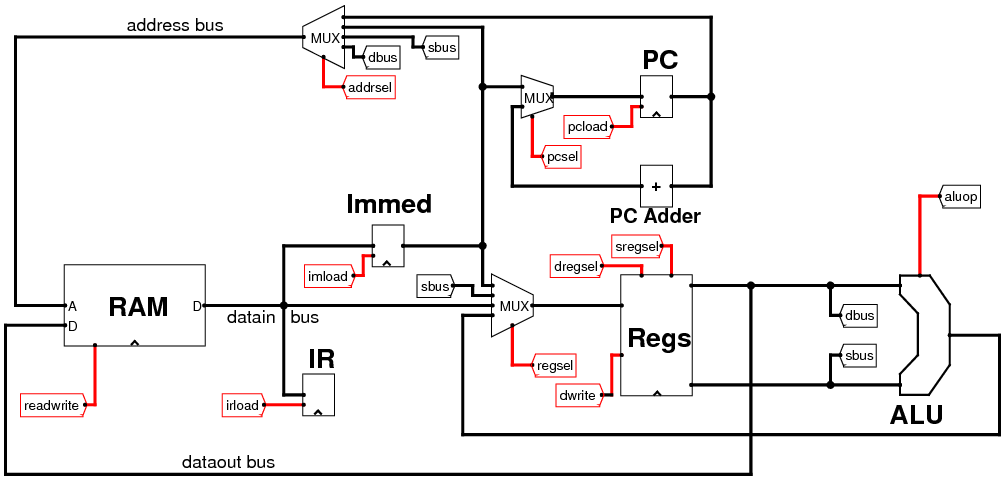
* La compuerta OR que se conecta a la entrada de fase 2 del multiplexor derecho elige una prueba de salto exitosa (entrada superior izquierda) O una instrucción de "saltar siempre" (entrada inferior derecha).
* La instrucción "saltar siempre" es *op1op2* = 1111, por lo que se utiliza una puerta AND de 4 entradas para seleccionar solo este código de operación.
* El multiplexor de la izquierda produce verdadero cuando la decisión de EQ / NE / GT / LT es verdadera, para cualquier instrucción (no solo una instrucción de salto). Compara la lógica de la compuerta aquí con la tabla de verdad de arriba.
* Pero debemos asegurarnos de que solo *emitimos la* decisión EQ / NE / GT / LT sobre las instrucciones de salto, es decir, cuando *op1op2* = 10xx. Por lo tanto, utilizamos la puerta AND de 3 vías y también utilizamos las líneas *irbit6* e *irbit7* como entradas.

**6.9   finalmente ...**

* Poniendo todo esto de nuevo juntos, ahora tenemos este dispositivo:

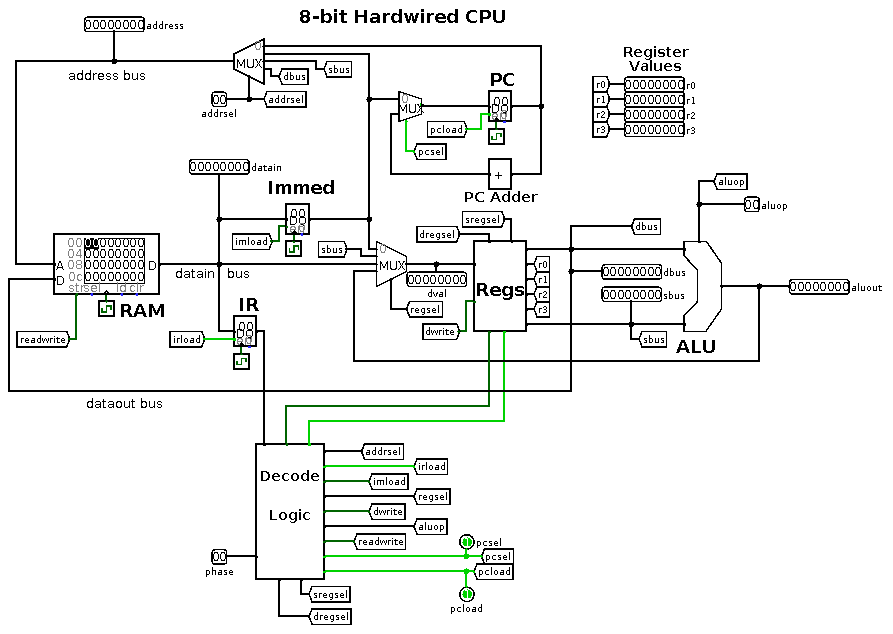


que controla el flujo de datos para toda la CPU:



**7   Implementando la CPU**

* Aquí están los archivos de implementación de Logisim para la CPU, junto con la imagen de memoria para el programa en la siguiente sección.
  + [CPU.circ](https://minnie.tuhs.org/CompArch/Tutes/Files/CPU.circ)
  + [ALU.circ](https://minnie.tuhs.org/CompArch/Tutes/Files/ALU.circ)
  + [cpumem.img](https://minnie.tuhs.org/CompArch/Tutes/Files/cpumem.img)
* Descarga los tres en la misma carpeta.
* Ejecute *Logisim* y abra el archivo *CPU.circ* . Verá la CPU, pero hay pantallas adicionales que muestran los valores en los registros y en algunas de las rutas de datos y líneas de control.



* Para cargar la memoria RAM con la imagen de la memoria, haga clic con el botón derecho en el dispositivo RAM, elija *Cargar imagen* , navegue hasta el archivo *cpumem.img* y luego haga clic en *Aceptar* .
* Para ejecutar el programa, escriba control-T repetidamente. Cada control-T realiza la mitad de un ciclo de reloj.

**8   un programa ejemplo**

* Es hora de ver un programa de ejemplo escrito para esta CPU.
* En la memoria que comienza en la ubicación 0x80 hay una lista de números de 8 bits; El último número en la lista es 0.
* Queremos que un programa sume los números, almacene el resultado en la ubicación de memoria 0x40 y luego realice un ciclo indefinido.
* Tenemos 4 registros para usar. Se asignan de la siguiente manera:
  + R0 mantiene el puntero al siguiente número para agregar.
  + R1 tiene la suma corriente.
  + R2 tiene el siguiente número para agregar a la suma corriente.
  + R3 se utiliza como un registro temporal.
* Aquí está el código de estilo de montaje para el programa.
* LI R1,0x00 # Poner suma corriente a cero
* LI R0,0x80 # Comenzar al principio de la lista
* bucle: LW R2, (R0) # Obtener el siguiente número
* JEQ R2, final # Salir del bucle si número == 0
* AGREGAR R1, R2 # Agregar número a la suma corriente
* LI R3, 0x01 # Poner 1 en R3, así podemos hacerlo
* AÑADIR R0, R3 # R0 ++
* JMP loop # Loop back
* final: SW R1, 0x40 # Almacene el resultado en la dirección 0x40
* inf: JMP inf # bucle infinito
* Convirtiendo a código de máquina, aquí están los valores hexadecimales para poner en memoria comenzando en la ubicación 0:

|  |  |
| --- | --- |
| LI R1,0x00 | e4 00 |
| LI R0,0x80 | e0 80 |
| LW R2, (R0) | 48 |
| JEQ R2, fin | 88 0d |
| AÑADIR R1, R2 | 26 |
| LI R3, 0x01 | ec 01 |
| AÑADIR R0, R3 | 23 |
| Bucle JMP | ff 04 |
| SW R1, 0x40 | d4 40 |
| JMP inf | ff 0f |

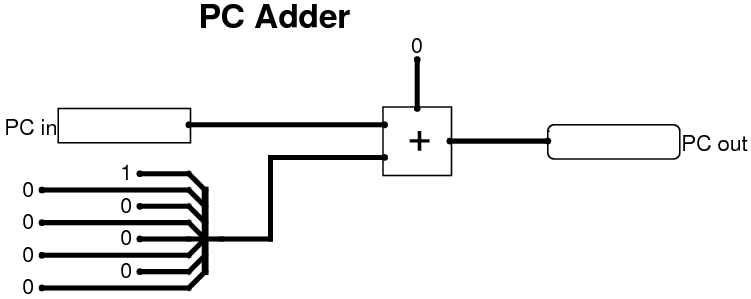
* Con la CPU cargada en Logisim y la memoria cargada con los valores de datos anteriores, podemos iniciar el programa en ejecución.
* Mira las fases de operación. Mira cómo el IR se carga con una instrucción.
* Ver el Registro Inmediato se carga con un valor.
* En la instrucción LW, observe cómo se selecciona el valor del *bus sbus* para ubicarse en el bus de direcciones, y el valor del datain se escribe en el registro de destino.
* En las instrucciones de ALU, observe los valores de *sbus* y *dbus* , el *aluop* y el resultado que se vuelve a escribir en el registro de destino.
* En la instrucción JEQ, observe el valor de N y Z en la lógica de decodificación, y los *valores* resultantes de *pcsel* y *pcload* .

**9   Otras Áreas de la CPU.**

* Esta última sección cubre las partes restantes de la CPU. Probablemente no tengamos tiempo en el tutorial para explorar estas áreas, pero están documentadas aquí de todos modos.

**9.1   El PC Adder**

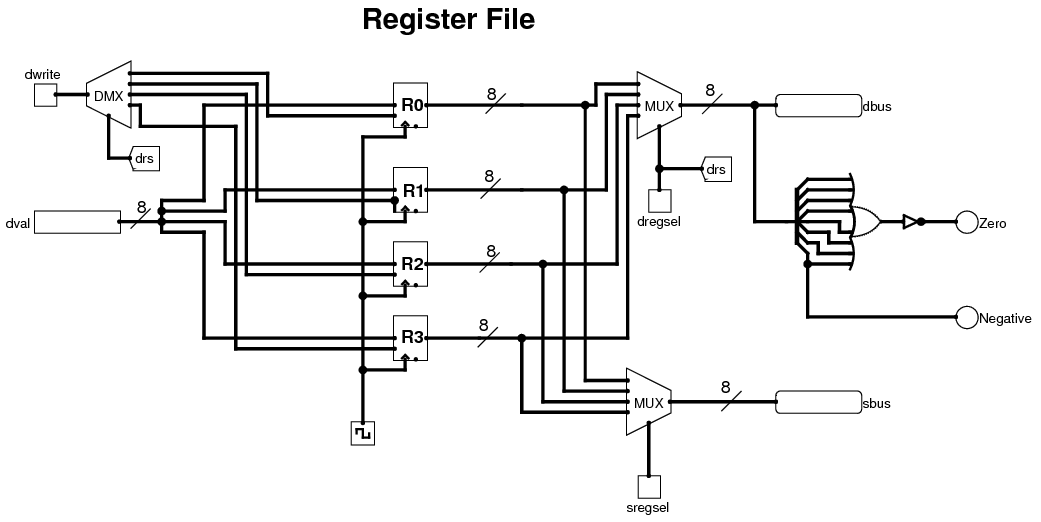
* La lógica de PC Adder es agradable y simple.



* Con la unidad sumadora Logisim incorporada, tome el antiguo valor de PC como una entrada y la constante de 8 bits 00000001 como la otra entrada, agréguelos y obtenga el nuevo valor de PC.

**9.2   El archivo de registro**

* La lógica del archivo de registro parece fea, por lo que veremos cada sección por turno.



* En el medio están los cuatro registros, que son unidades Logisim integradas.
* El valor de cada registro se envía a dos multiplexores a la derecha.
* El multiplexor superior selecciona uno de los valores de registro en función de la línea de control *dregsel* y, por lo tanto, envía un valor en la línea *dbus* a la ALU.
* De forma similar, el multiplexor inferior selecciona uno de los valores de registro en función de la línea de control de *sregsel* y, por lo tanto, envía un valor en la línea de *sbus* a la ALU.
* También hay lógica para probar si el valor de *dbus* es negativo. Los ocho bits en el *dbus* se dividen, y el bit más significativo es la salida *negativa* . Esto implica que usamos la representación de dos complementos para los valores firmados.
* Los ocho bits del *dbus* se ordenan juntos y luego se niegan.
* Cuando el *dbus* es cero, todos los bits son cero. La salida OR es cero, y por lo tanto la salida negada en la línea *Cero* es verdadera.
* A la izquierda, cada registro toma tres entradas:
  + un valor para posiblemente cargar. Todos los registros están conectados al *dval de* entrada que contiene el valor a cargar.
  + El pulso del reloj, que le indica a los registros cuándo cargar.
  + una señal de habilitación de escritura. Solo cuando la señal de activación de escritura es 1, un registro puede sobrescribir su valor anterior.
* El demultiplexor en la parte superior izquierda toma como entrada *dwrite* (es decir, la señal de escritura), y en base al valor de *dreg* , selecciona a qué registro enviar la señal *dwrite* . Todos los demás registros obtendrán una señal de escritura de 0, es decir, no realizar una operación de escritura.