Universitat de les
Illes Balears

Estructura de Computadores

Curso académico: 2014-2015

Nombre: Alberto Gelabert Mena

DNI: 43193979W

Email: albertogelabert95@gmail.com

Variables más importantes

A continuación resumiremos las variables que encontramos en el proyecto.

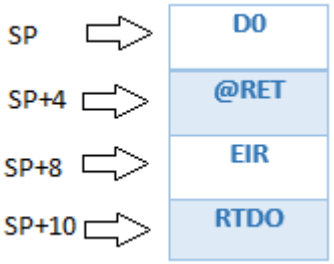
- EPROG: Contiene el eprograma a emular. Aquí deberemos poner nuestras instrucciones codificadas en hexadecimal.
- Variables de la PEPA4r:
    - EIR: contiene la instrucción a ejecutar
    - EPC: el contador de programa de la PEPA4r, de 1 en 1.
    - Registros de la PEPA4r:
        - ER0. Su función es la de registro acumulador en la mayoría de operaciones con la ALU.
        - ER1. Se utiliza como interfaz con la memoria (escribir y leer).
        - ER2 y ER3. Solo se utilizan para llevar a cabo operaciones ALU.

Subrutinas

Seguidamente, pasaremos a explicar brevemente las subrutinas que he utilizado.

- **Subrutina de descodificación** (DESCO). Tal cual como se indica en el enunciado de la práctica, se necesita de una subrutina para realizar la fase de descodificación. Dicha subrutina tiene que ser de librería, es decir, cualquiera conociendo como se le pasan los parámetros podría utilizarla.

    - **Espacio resultado**: antes de pasar el parámetro a la pila, hemos de guardar un Word en la pila para que la subrutina deje ahí el resultado.
    - **Paso de parámetros**: le hemos de pasar por parámetro un Word que contendrá la instrucción para descodificar. En la Figura podemos mirar la estructura que tendría la pila.



    - **Funcionamiento**: mediante un árbol de comprobaciones de test (mediante la instrucción BTST), vamos mirando cada uno de los bits de la instrucción, y así vamos por un camino u otro dependiendo del valor del bit, hasta que finalmente llegamos a descodificar la instrucción identificando de cuál se trata. Podemos ver el árbol en la figura 1.
    - **Resultado**: una vez sabemos cuál es la instrucción, escribimos en el espacio para el resultado un valor numérico que identifica de qué instrucción se trata.

- **Subrutina de flags**. He decidido modificar el valor del ESR mediante una subrutina, para ahorrar código y hacerlo así más entendible. Para ello, he creado 3 subrutinas, una para cada flag, debido a que ciertas instrucciones no actualizaban todos los flags, así, solo actualizo los flags que se deben actualizar. No se le deben de pasar ningún parámetro, tan solo debemos asegurarnos que lo tenemos en el registro D0. De igual manera, no debemos coger ningún valor de retorno, directamente modifica el ESR.

- **Flag Z**. Para determinar si se trata de un cero, nos basta con hacer una comparación de los 12 bits menos significativos. Para ello, eliminamos previamente los otros 4 bits que no nos interesan mediante una máscara.
- **Flag N**. Para determinar el flag N, miraremos únicamente el valor del bit 11, ya que trabajamos con números en binario con signo. Si el bit es 1, quiere decir que el número es negativo; de lo contrario, positivo.
- **Flag C**. Finalmente, el flag C se obtiene de mirar los bits 12 y 13. La única manera de que se haya producido acarreo es que el bit 12 y 13 sean 1 y 0 respectivamente.

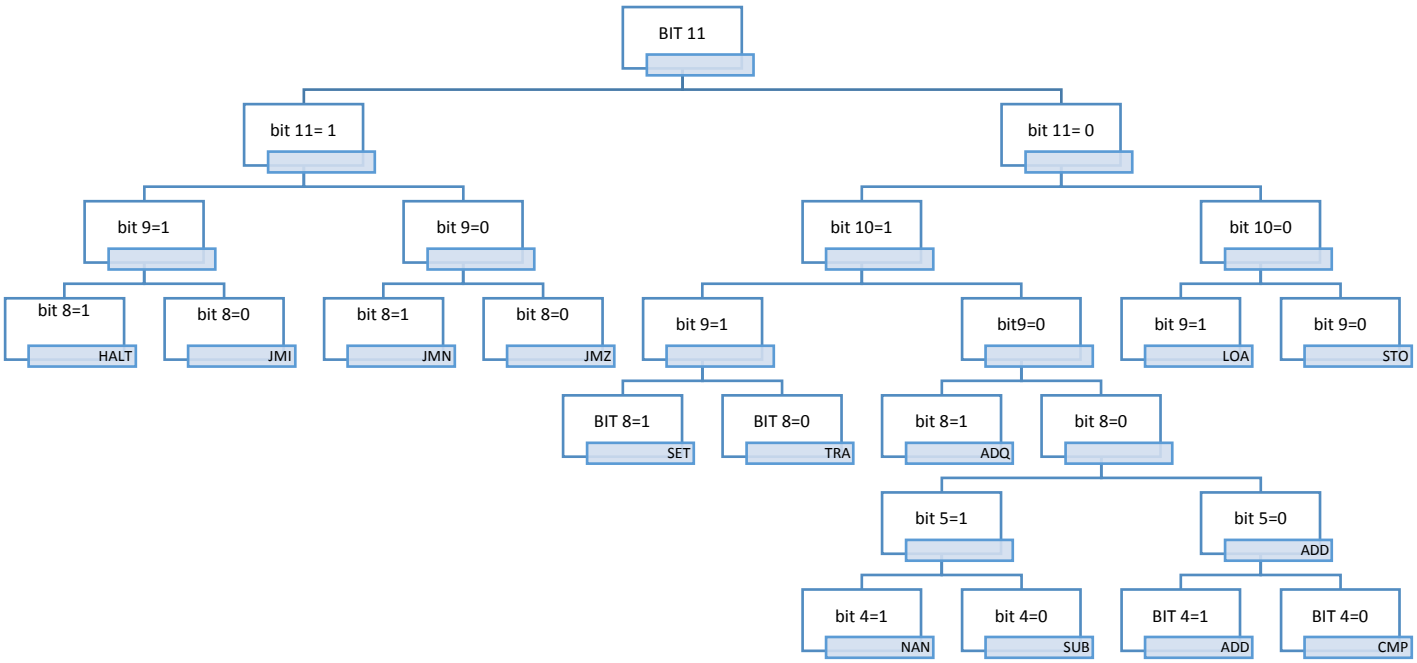


Figura 1. Árbol binario representativo de la subrutina de descodificación

Para resumir, a alto nivel, el funcionamiento del programa, primero cabe destacar que el PC de la PEPA va de 1 en 1, mientras que las posiciones que ocupan sus direcciones son 2, es decir, para acceder al EPROG, lo haremos usando el EPC como índice, pero antes lo tendremos que multiplicar por 2.

FETCH

- Acceder a EPROG con EPC
- Guardar la instr. en EIR
- Incrementar EPC

DESCODIFICACIÓN

- Reservamos espacio en la pila para RTDO (w).

- Movemos [EIR] a la pila.
- JSR DESCO
- Eliminamos EIR de la pila
- Recuperamos RTDO pila.

EJECUCIÓN

- Multiplicamos el identificador numérico de la instr. para acceder a una jumplist.
- Mediante una jumplist saltamos a la etiqueta de la instrucción que toca donde se llevará a cabo su ejecución.
- Una vez acabada la ejecución, saltamos a la fase de fetch hasta que se ejecute una instrucción HALT.

EINSTRUCCIONES

A continuación haré breves aclaraciones acerca de los aspectos más relevantes de cada una.

- **De salto**:

JMI. Mediante una máscara, sacamos la dirección de la instrucción y la metemos en EPC

JMZ. Miramos mediante BTST si el bit Z=1 (bit 2), si se cumple, mediante una máscara sacamos la dirección de salto y la metemos en EPC.

JMN. Ídem de la anterior, pero en este caso miramos el bit N (bit 1).

- **De transferencia entre registros**:

STO. Sacamos la dirección de la instrucción mediante una máscara y movemos el contenido de ER1 a la posición de memoria indicada mediante direccionamiento indexado.

LOA. Ídem a la instrucción STO pero a la hora de realizar la transferencia, movemos de la memoria al registro ER1.

SET. Sacamos el número del registro sobre el que haremos el set mediante una máscara y seguidamente movemos dos posiciones hacia la derecha con la instrucción LSR. Haciendo comparaciones, averiguo a qué registro se refiere y le meto el valor CCCCCC obtenido previamente con una máscara.

TRA. Primero sacamos el número de ambos registros y después hago comparaciones hasta llegar a saber el registro origen y destino que indica la instr. Al hacer la transferencia, llamo a las subrutinas que actualizan los flags Z y N.

- **Operaciones de tipo ALU**:

CMP. Extraemos el número del registro con el cual haremos la comparación mediante una máscara, movemos dos posiciones a la derecha y hacemos comparaciones hasta saber de qué registro se refiere. Una vez tenemos el registro, lo negamos y lo sumamos a ER0. Seguidamente, sin guardar el resultado en ningún lado, llamamos a las subrutinas que actualizan los flags Z, N y C. Acabado esto, saltamos a FETCH

ADQ. Mediante una máscara extraemos el valor de la constante dd. Haciendo comparaciones averiguamos cuál es su valor. Seguidamente extraemos el identificador del registro y hacemos las comparaciones hasta determinar qué registro es. Hacemos la suma y llamamos a las subrutinas que actualizan los flags Z, N y C

ADD. Ídem de la instrucción anterior (ADQ), con la diferencia de que en esta, sólo haremos una comparación para obtener el registro Ra, ya que el registro destino será siempre R0. El resto igual, realizamos la suma y llamamos a las subrutinas que actualizan los flags Z, N y C.

SUB. Ídem de la instrucción anterior (ADD), sólo que en esta instrucción haremos una resta en vez de una suma. La resta la llevamos a cabo negando el valor a restar y sumándolo, en vez de restarlo directamente.

NAN. Primero de todo extraemos mediante una máscara el valor del registro Ra. Una vez sabemos qué registro es realizamos la operación NAN. Para ello hacemos primero una NOT y después una AND. Seguidamente llamamos a las subrutinas que actualizan los flags Z, N y C.

- **De tipo HALT**:

HALT. Simplemente hacemos un SIMHALT y se detiene la ejecución.

Conclusión

Como comentario final, destacar que al final de cada instrucción (excepto las de tipo salto y STO y LOA) he añadido un máscara a los registros de la PEPA4r, ya que estos son de 12 bits, y el 68k trabaja con 16 bits. Para no tener problemas, los 4 bits más significativos debería ser 0 siempre, por eso, después de llamar a las subrutinas de actualización de flags, he añadido una máscara para eliminar esos 4 bits y ponerlos a 0.

Opiniones y dificultades

Como opinión personal me gustaría remarcar que no se trata de una práctica difícil, solamente tienes que tener los conceptos muy claros de emular una máquina que trabaja con 12 bits sobre una máquina que utiliza 16, como he comentado anteriormente. Además, encuentro que aunque el tiempo para realizar la práctica parece a priori justo, es suficiente. La culpa es de los alumnos que no nos tomamos enserio cuando los profesores decís que hemos de llevar la práctica al día.