Nota: me saqué 8, marco las cosas que corrigió mal a otros compañeros.

1. De la instrucción sethi dar microcódigo y localización de cada línea. Después para la instrucción sethi 2CBh, %r2 dar contenido de cada bus, bus del sistema, MIR y multiplexor del control store.

Bueno, tomó sethi que nunca lo había tomado, así que vamos con eso.

Para comenzar, es necesario ubicar la localización en memoria ROM del microcódigo de esta instrucción, utilizando el DECODE, que se completa de la siguiente manera:

1 OP OP3/OP2 00

Según el IR, el decode queda:

1 00 100000 00 = 1152

MAL poner los 000 luego del OP2 del sethi

Correción de Mazzeo:

La localizacion en la memoria de control queda determinada por 11 bits que dependen de op y op3. Dado que sethi no tiene op3 sino op2 , hay 3 bits que no estan definidos. Es decir, esos tres bits dependen del valor de la constante imm22 que pase el programador. Se espera que cualquiera sea el valor de esa constante el microprocesador sea capaz de ejecutar la instruccion sethi. Si presupones que esos 3 bit valen 000 el microprocesador funcionara correctamente con sethi solo cuando el valor de imm22 que el programador decida presuponga 000 en esos bits. Que deberias hacer para que funcione en cualquier caso? Podrias repertir el codigo en todas la posiciones correspondientes, esto esta tambien relacionado con el tema de la nanoprogramacion

El microcódigo correspondiente a sethi es:

1152: R[RD] <- LSHIFT10(R[IR]);
GOTO 2047;

¿Cómo se integra esta microinstrucción dentro del ciclo de Fetch? Con las siguientes líneas:

```
0:    R[IR] <- AND(R[PC], R[PC]);
      READ;
1:    DECODE

1152: R[RD] <- LSHIFT10(R[IR]);
      GOTO 2047;
```

2047: R[PC]<- INCPC(R[PC], R[PC]);
GOTO 0;

Este algoritmo coordinado por la unidad de control permite la ejecución de las instrucciones. Consta de los pasos:

1. Buscar en memoria la próxima instrucción a ejecutar (línea 0).
2. Decodifico (línea 1).
3. Ejecuto (líneas correspondientes al microcódigo de una instrucción).
4. Vuelvo al paso 1 (línea 2047).

Recordar que no todos los microcódigos necesitan el salto a la línea 2047.

sethi 2CBh, %r2

Contenido de cada bus:

Bus A: contenido del registro %ir (donde se encuentra guardada la constante 2CB)

Bus B: indistinto su valor.

Bus C: contenido del %r2 (resultado de la operación del sethi).

Acá lo que agregué fue *cómo llegaba el contenido de los registros a cada bus, con los decodificadores y multiplexores de los buses A, B y C.*

Bus del sistema:

Puse literalmente lo que era el bus de sistema, diferencia con el modelo de Von Neumann, hice el dibujo de cómo se conectaban los buses de datos, direcciones y control con el CPU, la memoria y los dispositivos de I/O.

MIR de la 1° microinstrucción:

Expliqué qué era cada campo y cuáles eran las opciones que había para escribir en los mismos.

1152: R[RD] <- LSHIFT10(R[IR]);
GOTO 2047;

A: 100101 (37), el registro %ir

AMUX: 0 (para que me tome el registro del campo A)

B - BMUX: indistinto

C: indistinto

CMUX: 1 (para que tome el rd del %ir, el %r2)

RD/WR: 0 (no leo ni escribo)

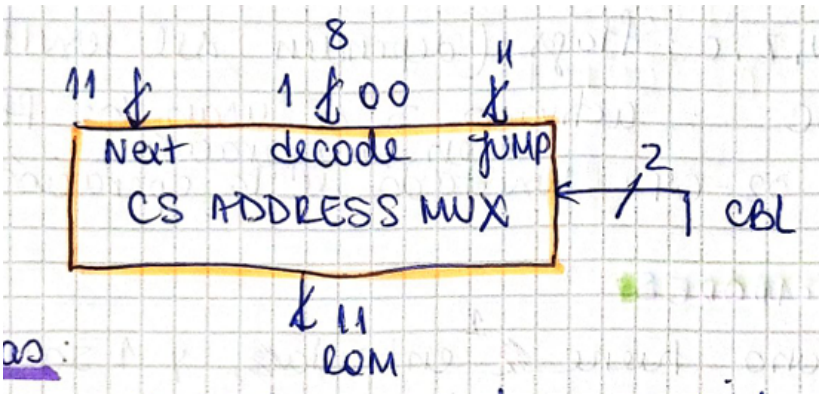
ALU: 1010 operación LSHIFT10

COND: 110 (salto sin condición)

JUMP ADDR: 11111111111 (2047, línea a la que voy a saltar, el GOTO)

A AMUX B BMUX C CMUX RD WR ALU COND JUMP ADDR
100101 0 000000 0 000000 1 0 0 1010 110 11111111111

Multiplexor de la control store:



Entradas:

Next: dirección de la siguiente microinstrucción inmediata.
Decode: dirección de la 1° microinstrucción (1152).
Jump: dirección de dónde voy a saltar (2047)
CBL (selector): 2 bits que vienen de la salida de la lógica de control de saltos, pueden ser:

00 Next
01 Jump
10 Inst. Dec

La que ingresa en este caso es 01 Jump.
Salida: 2047, a la ROM.

Le agregué como extra *cómo funcionaba la lógica de control de saltos, entradas y salidas y cómo se elegía la próxima microinstrucción.*

2. En un programa cargar un arreglo de 48 elementos, mandar su largo y dirección por pila a una subrutina declarada en el mismo módulo que devuelva por stack la suma de los números cuyos primeros 2 bits sean 1 y cargar en el main el resultado en un periférico de dirección X.

Usé 2 máscaras para obtener los valores que tenían los primeros 2 bits en 1.

máscara 1: 80000000 - 1000 0000 0000 … 0000
máscara 2: 40000000 - 0100 0000 0000 … 0000

```
                .begin
                .org 2048
                .macro push arg
                add %r14,-4, %r14
                st arg, %r14
                .end macro
                .macro pop arg
                ld %r14, arg
                add %r14, 4, %r14
                .end macro
array:          .dwb 48 !declaro arreglo
largo:          .equ 192 !declarlo largo (48*4)
main:           ld [periferico], %r1 !cargo dirección del periférico
                ld [mascara1], %r2
                ld [mascara2], %r3
                add %r0, largo, %r4
                add %r0, [array], %r5
                push %r5
                push %r4
                add %r0, %r15, %r31
                call loop
                pop %r6
                st %r4, %r1
                jmpl %r31 + 4, %r0
loop:           pop %r7 !recupero largo
                pop %r8 !recupero dirección de inicio
                andcc %r7, %r7, %r0
                be fin_array
                add, %r7, -4, %r7
                ld %r7, %r8, %r9
                andcc %r9, %r2, %r0 !compruebo 1° bit
                be loop
                andcc %r9, %r3, %r0 !compruebo 2° bit
                bne sumar
                ba loop
sumar:          add %r9, %r10, %r10
                ba loop
fin_array:      push %r10
                jmpl %r15 + 4, %r0
periferico:     0xC34A0000
mascara1.       0x80000000
mascara2:       0x40000000
                .end
```

3. Describir linker editor, linker loader y linker loader dinámico y ordenarlos según la ocupación de memoria ram en ejecución.

Ranking de ocupación de RAM en ejecución (del que ocupa menos al que ocupa más)

Expliqué los 3 que pedía y le hice el ranking.

Link editor: combina programas que fueron ensamblados por separado. Resuelve referencias globales y externas y reubica direcciones internas en caso de superponerse entre módulos.

Linking Loader: linkea en tiempo de carga (enlace + carga) y busca automáticamente las librerías, de esta manera no repito una misma librería varias veces.

Linking Loader dinámico: linkea en tiempo de ejecución, carga las librerías según el programa las necesita, se realiza con DDL que pospone el enlace hasta que sean requeridas en tiempo de ejecución.

1. Linking loader dinámico (uso las librerías solo cuando las necesito).
2. Linking loader (cargo 1 vez la librería externa y no las repito).
3. Link editor (linkeo puro)