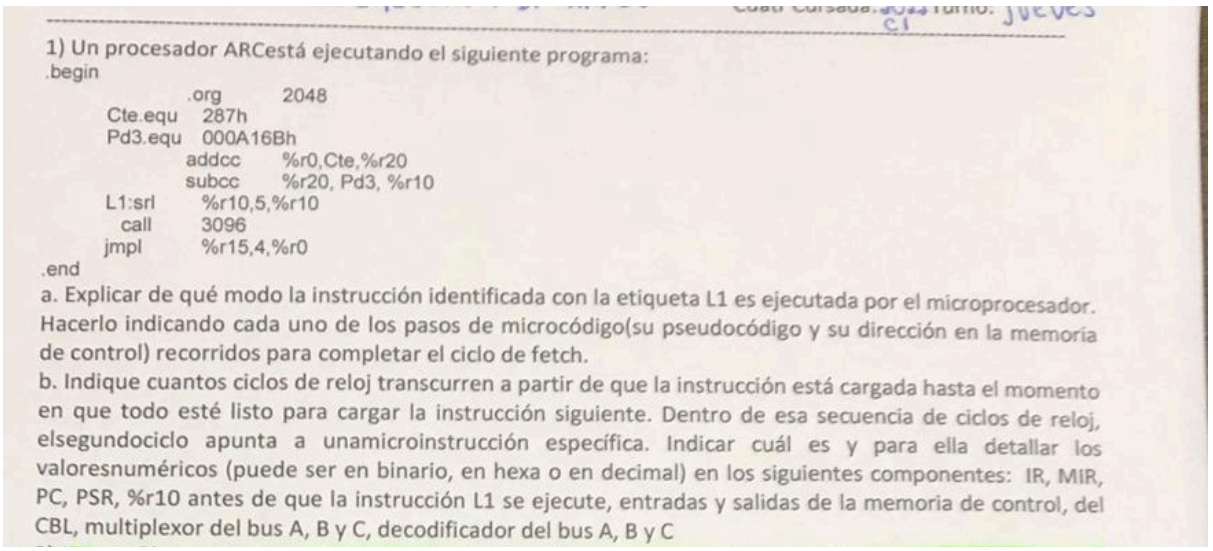


1) Un procesador ARCestá ejecutando el siguiente programa:

```
.begin
            .org        2048
      Cte.equ   287h
      Pd3.equ   000A16Bh
            addcc       %r0,Cte,%r20
            subcc       %r20, Pd3, %r10
      L1:srl      %r10,5,%r10
        call      3096
      jmpl        %r15,4,%r0
.end
```

a. Explicar de qué modo la instrucción identificada con la etiqueta L1 es ejecutada por el microprocesador. Hacerlo indicando cada uno de los pasos de microcódigo(su pseudocódigo y su dirección en la memoria de control) recorridos para completar el ciclo de fetch.

b. Indique cuantos ciclos de reloj transcurren a partir de que la instrucción está cargada hasta el momento en que todo esté listo para cargar la instrucción siguiente. Dentro de esa secuencia de ciclos de reloj, elsegundociclo apunta a unamicroinstrucción específica. Indicar cuál es y para ella detallar los valoresnuméricos (puede ser en binario, en hexa o en decimal) en los siguientes componentes: IR, MIR, PC, PSR, %r10 antes de que la instrucción L1 se ejecute, entradas y salidas de la memoria de control, del CBL, multiplexor del bus A, B y C, decodificador del bus A, B y C

a.

La instrucción es ejecutada por el microprocesador a partir del Ciclo de Fetch, que es el algoritmo mediante el cual las instrucciones de un programa son ejecutadas, siendo este un proceso dirigido por la Unidad de Control.

El Ciclo de Fetch realiza los siguientes pasos:

1. Buscar la siguiente instruccion en memoria a ejecutar.
2. Decodificar esa instrucción.
3. Ejecutarla.
4. Volver al paso 1.

Para entender cómo en este proceso se logra ejecutar la microinstrucción, realizaré el microcódigo:

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

/MICROCODIGO DE SRL

X:      IF R[IR[13]] THEN GOTO X+2;
X+1:    R[RD] <- SRL(R[RS1], R[RS2]);
        GOTO 2047;
X+2:    R[TEMP0] <- SIMM13(R[IR]);
X+3:    R[RD] <- SRL(R[RS1], R[TEMP0]);
        GOTO 2047;

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

La dirección X se dará por el DECODE de la instrucción SRL:

1 10 100110 00 = 1688
La idea es que en la microinstrucción 0, se guarda a lo que está apuntando el program counter PC en el registro IR, que será la próxima instrucción a ejecutar y la leerá. Luego en la línea 1, se decodifica para encontrar la dirección de la primer microinstrucción del microcódigo correspondiente a esa instrucción, en nuestro caso, será de la instrucción SRL.

Una vez que llegamos a esa dirección de memoria, ejecutamos las líneas que correspondan (de la 1688 a la 1692). Observemos que hay 2 flujos de caminos posibles para la instrucción, siendo si el modo de direccionamiento es inmediato o no.

Sea cual sea el modo de direccionamiento, luego de cargar en el registro destino el resultado de la operación, vamos a la línea 2047 donde incrementamos el program counter para poder dirigirnos a la siguiente instrucción volviendo a la línea 0.

b.

Cada microinstrucción tarda generalmente un ciclo de reloj en ejecutarse, solo las instrucciones que involucren el acceso a memoria pueden tardar más.

1: DECODE; **/+1**

/MICROCODIGO DE SRL

1688: IF R[IR[13]] THEN GOTO 1690; **/+1**
1689: R[RD] <- SRL(R[RS1], R[RS2]);
GOTO 2047;
1690: R[TEMP0] <- SIMM13(R[IR]); **/+1**
1691: R[RD] <- SRL(R[RS1], R[TEMP0]); **/+1**
GOTO 2047;

En total tendremos 4 ciclos de reloj.

Para esta microinstrucción:

1690: R[TEMP0] <- SIMM13(R[IR]);

IR

srl %r10, 5, %r10

10 01010 100110 01010 1 0000000000101
op %r10 op3 rs1 i 5

MIR
A AMUX B BMUX C CMUX RD WR ALU COND JUMP ADDR
100101 0 000000 0 100001 0 0 0 1011 000 00000000000

PC Program Counter 2056

PSR Flags
n, v, z, c

Con la operación de addcc no se activa ningún flag, ahora con la de subcc me dará un valor negativo así que n = 1.

Entonces los flags quedan:

n = 1
v = 0
z = 0
c = 0

%r10 antes de que se realice L1

Resta entre el contenido de %r10 y la constante 000A16B (0 - esa cte)

Memoria de Control ROM (Control Store)

Entrada: 1691
Salida: MIR de la microinstrucción 1691

1691: R[10] <- SRL(R[10], R[TEMP0]);
GOTO 2047;

A AMUX B BMUX C CMUX RD WR ALU COND JUMPADDR
001010 0 000000 0 001010 0 0 0 0100 110 2047

CBL

Entradas:
flags: como estaban
ir: 1
COND: 000
Salidas:
NEXT: 00

Multiplexor Bus A

Entradas:
RS1: rs1 del IR 01010
A (MIR): 100101
Selector AMUX: 0

Salida: 100101

Multiplexor Bus B -> No interviene

Multiplexor Bus C

Entradas:
rd: rd del IR 01010 (%r10)
C (MIR): 100001 (33 temp0)
Selector CMUX: 0

Salida: 100001

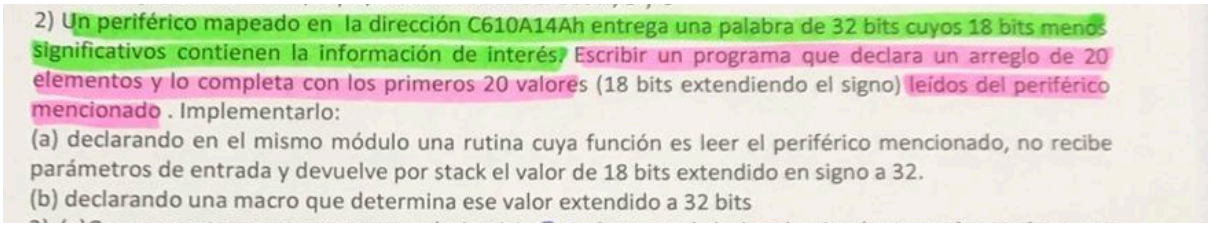
Decodificador Bus A

Entrada: 100101 (IR 37)
Salida: 38 bits, todos en cero menos el bit 37

Decodificador Bus B -> No interviene

Decodificador Bus C

Entrada: 100101 (TEMP0 33)
Salida: 38 bits, todos en cero menos el bit 33


2) Un periférico mapeado en la dirección C610A14Ah entrega una palabra de 32 bits cuyos 18 bits menos significativos contienen la información de interés. Escribir un programa que declara un arreglo de 20 elementos y lo completa con los primeros 20 valores (18 bits extendiendo el signo) leídos del periférico mencionado . Implementarlo:
(a) declarando en el mismo módulo una rutina cuya función es leer el periférico mencionado, no recibe parámetros de entrada y devuelve por stack el valor de 18 bits extendido en signo a 32.
(b) declarando una macro que determina ese valor extendido a 32 bits

a.

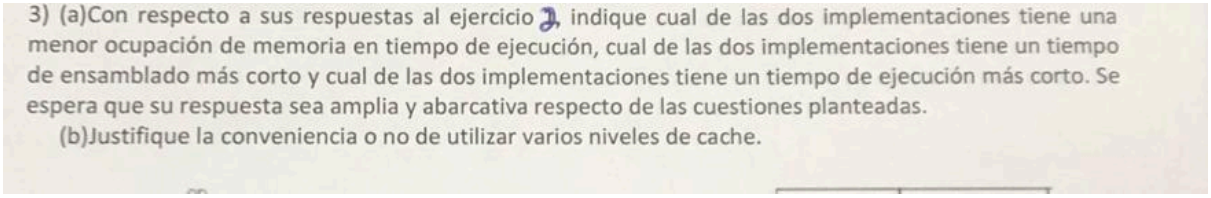
```
                .begin
                .org 2048
                .macro push arg
                add %r14, -4, %r14
                st arg, %r14
                .end macro
                .macro pop arg
                ld %r14, arg
                .end macro
array:          .dwb 20
largo:          .equ 80
main:           ld [periferico], %r1 !dirección periférico
                add %r0, largo, %r2 !largo del array
                add %r15, %r0, %r31
                call loop
loop:           andcc %r2, %r2; %r0
```

```
                be fin
                call lectura
                pop %r3 !recupero el valor leído extendido de signo a 32 bits
                add %r2, -4, %r2
                st %r3, %r2, [array]
                ba loop
lectura:        st %r1, %r4
                sla %r4, 14, %r4 !me quiero deshacer de los 12 bits de adelante
                sra %r4, 14, %r4
                push %r4
                jmpl %r15+4, %r0
fin:            jmpl %r31+4, %r0
periferico:     0xC610A14A
                .end
```

b.

Con la macro:

```
                .begin
                .org 2048
                .macro push arg
                add %r14, -4, %r14
                st arg, %r14
                .end macro
                .macro pop arg
                ld %r14, arg
                .end macro
                .macro valor arg
                sla arg, 12, arg
                sra arg, 12, arg
                .end macro
periferico:     0xC610A14A
array:          .dwb 20
largo:          .equ 80
main:           ld [perif], %r1
                add %r0, largo, %r2 !largo del array
loop:           andcc %r2, %r2; %r0
                be fin
                add %r2, -4, %r2
                st %r1, %r4
                valor %r4
                st %r4, %r2, [array]
                ba loop
fin:            jmpl %r15+4, %r0
```


3) (a)Con respecto a sus respuestas al ejercicio 2, indique cual de las dos implementaciones tiene una menor ocupación de memoria en tiempo de ejecución, cual de las dos implementaciones tiene un tiempo de ensamblado más corto y cual de las dos implementaciones tiene un tiempo de ejecución más corto. Se espera que su respuesta sea amplia y abarcativa respecto de las cuestiones planteadas.
(b)Justifique la conveniencia o no de utilizar varios niveles de cache.

a.

Ocupación de memoria en tiempo de ejecución:

Al tener una macro que reemplaza una de las rutinas, tendré menos símbolos para almacenar en memoria, así que la que tiene la macro tendrá el menor espacio de memoria en tiempo de ejecución.

Tiempo de ensamblado:

En cuestiones del ensamblado, en ambos tendríamos 2 pasadas ya que son de un único módulo. Pero si tenemos una macro extra, habrá que utilizar tiempo para reemplazarla, por lo tanto, la versión con la macro adicional, tendrá mayor tiempo de ensamblado, y la que no la tiene, menor tiempo.

Tiempo de ejecución:

Al ejecutar el programa, ya habremos pasado por el ensamblado, el cual expandirá las macros. Si pensamos en cuestiones de saltos de memoria, utilizando subrutinas hacemos más saltos e idas y vueltas en la memoria, causando mayor tiempo de ejecución.

b.

Utilizar diferentes niveles de Caché, se conoce como Caché Multinivel.

Consta de tener varios niveles de memorias caché, donde tendremos el primer nivel más cerca del CPU (ya sabemos que la caché está conectada físicamente a él) siendo el más veloz, y a medida que avanzamos los niveles, vamos reduciendo la velocidad de los mismos. Generalmente se utilizan hasta 2 o 3 niveles.

La ventaja que tiene esta organización de memoria caché, es que a la hora de encontrar un dato allí, iremos recorriendo los niveles y la última instancia será el acceso a la memoria principal, lo que conocemos como fallo de caché. De esta manera estamos retrasando el error ya que si un dato o instrucción no se encuentra en el primer nivel, irá al segundo y podría seguir al tercero.

Una desventaja es que a medida que aumentamos los niveles, estos ocupan mayor espacio físico y complejizan el sistema.