PUNTO 1

a) Un programa declara un arreglo de 16 elementos en punto flotante y lo completa con los primeros 16 valores positivos leídos de un periférico que esta mapeado en la dirección A3C20140h. La lectura de datos del periférico debe ser implementada mediante una rutina declarada en el mismo modulo a la cual no se le pasan parámetros y que devuelve por stack el valor leído. Una vez completado el arreglo, el programa termina.

b) Describir en palabras de qué forma debería replantear la escritura del código si se pidiera que la rutina este declarada en un módulo diferente al del programa principal, cuáles segmentos de código eliminaría, cuáles cambiaría y de qué modo, y en qué consistiría el código de la rutina declarada en el segundo módulo.

a.

```
            .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 16
largo:      .equ 64
main:       add %r0, largo, %r1
            add %r15, %r0, %r31
            call loop
loop:       andcc %r1, %r1, %r0
            be fin
            call lectura
            pop %r2
            bneg loop
            add %r1, -4, %r1
            st %r2, %r1, [array]
            ba loop
lectura:    ld [periferico], %r3
            st %r3, %r4
            push %r4
            jmpl %r15+4, %r0
periferico: 0xA3C20140
fin:        jmpl %r31+4, %r0
            .end
```

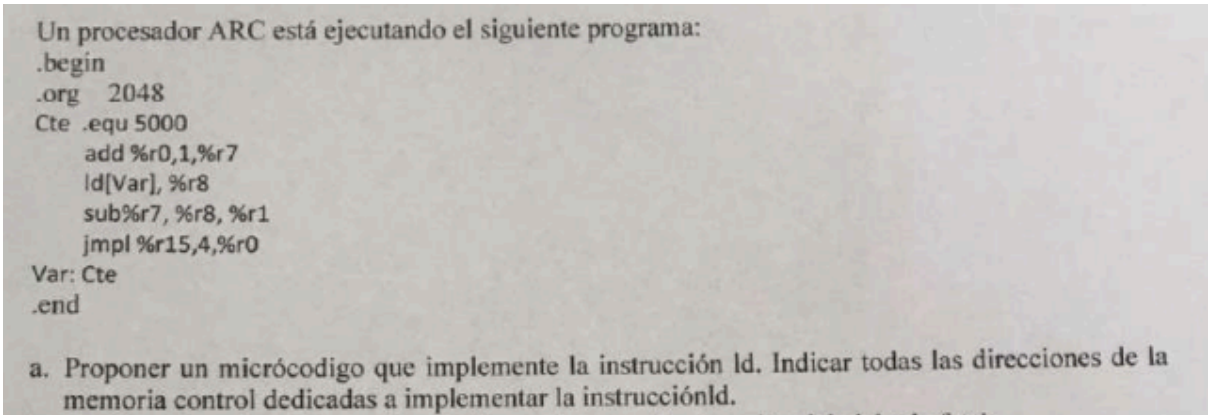
b.

Si realizamos la rutina de lectura en un módulo externo al que contiene el programa principal, como cambio principal en el módulo principal deberemos declarar a la subrutina lectura como .extern de esta manera está habilitado a hacerle el call y en el módulo de la subrutina, .global para que pueda ser llamada.

Luego, como el contenido de la subrutina es:

```
ld [periferico], %r3
st %r3, %r4
push %r4
jmpl %r15+4, %r0
```

Habría que volver a declarar la macro en el módulo de ella o directamente usar las líneas de código que representan el push del stack, también declarar la dirección del periférico.



Un procesador ARC está ejecutando el siguiente programa:

```
.begin
.org   2048
Cte  .equ 5000
     add %r0,1,%r7
     ld[Var], %r8
     sub%r7, %r8, %r1
     jmpl %r15,4,%r0
Var: Cte
.end
```

a. Proponer un micrócodigo que implemente la instrucción ld. Indicar todas las direcciones de la memoria control dedicadas a implementar la instrucciónld.

```
        .begin
        .org 2048
cte:    .equ 5000
        add %r0, 1 %r7          2048
        ld [Var], %r8           2052
        sub %r7, %r8, %r1       2056
        jmpl %15+4, %r0         2060
Var:    cte                     2064
        .end
```

a.

Hago de DECODE

1 11 000000 00

```
1792:  R[TEMP0] <- ADD(R[RS1], R[RS2]);
       IF R[IR[13]] THEN GOTO 1794;
1793:  R[RD] <- AND(R[TEMP0], R[TEMP0]);
       GOTO 2047;
1794:  R[TEMP0] <- SEXT13(R[IR]);
```

1795: R[TEMP0] <- ADD(R[RS1], R[TEMP0]);
GOTO 1793;

b. Explicar de qué modo ese microcódigo se integra a la ejecución del ciclo de fetch.

El microcódigo de la instrucción se integra a la ejecución del ciclo de fetch a partir de las líneas 0, 1 y 2047 del microcódigo.

0: R[IR] <- AND(R[PC], R[PC]); READ;
1: DECODE
1792: R[TEMP0] <- ADD(R[RS1], R[RS2]);
IF R[IR[13]] THEN GOTO 1794;
1793: R[RD] <- AND(R[TEMP0], R[TEMP0]);
GOTO 2047;
1794: R[TEMP0] <- SEXT13(R[IR]);
1795: R[TEMP0] <- ADD(R[RS1], R[TEMP0]);
GOTO 1793;
2047: R[PC] <- INCPC(R[PC], R[PC]);
GOTO 0;

c. Una vez que el microprograma saltó a la primera dirección del microcódigo propuesto en (a) indicar cuál es la microinstrucción siguiente y para ella detallar los valores numéricos (puede ser en binario, en hexa o en decimal) en los siguientes componentes:

-Registro de instrucciones
- Registro de microinstrucciones
- ProgramCounter
- Entradas y salidas de la ALU
- Entradas y salidas del multiplexor de la memoria de control
- Entradas y salidas de la lógica de control de saltos
- Multiplexor del Bus A
- Multiplexor del Bus B
- Decodificador del Bus A
- Decodificador del Bus B
- Decodificador del Bus C

**1794: R[TEMP0] <- SEXT13(R[IR]);**
**ld [Var], %r8**

**IR**

11 01000 000000 00000 1 2064

**MIR**

A AMUX B BMUX C CMUX RD WR ALU COND JUMP ADDR
37 0 000000 0 33 0 0 0 1100 000 00000000000

**Program Counter:** 2052

**ALU**

Entradas:
- F0, …, F3: 1100
- BUS A: contenido del IR 11 01000 000000 00000 1 2064
- BUS B: no importa

Salidas:
- BUS C: (resultado de la operación) extensión de signo de la constante de sim 13 del IR
- Flags
- SCC:

**Multiplexor de la memoria de control**

Entradas:
- NEXT 1795
- JUMP 00
- DECODE 1 OP OP3/OP2 00 1792
- SELECTOR **NEXT**, JUMP O INST. DEC.

Salida:
- 1795 (Al Control Store)

**Control Branch Logic**

Entradas:
- PSR, depende de las operaciones.
- IR[13] = 1
- COND = 000

Salida: **NEXT 00**, JUMP 01 O INST. DEC 10

**MUX A**

Entradas:
- A (MIR) 37 (Registro IR)
- RS1 (IR) (000000)
- Selector AMUX = 0

Salida: 37

**MUX B** no importa

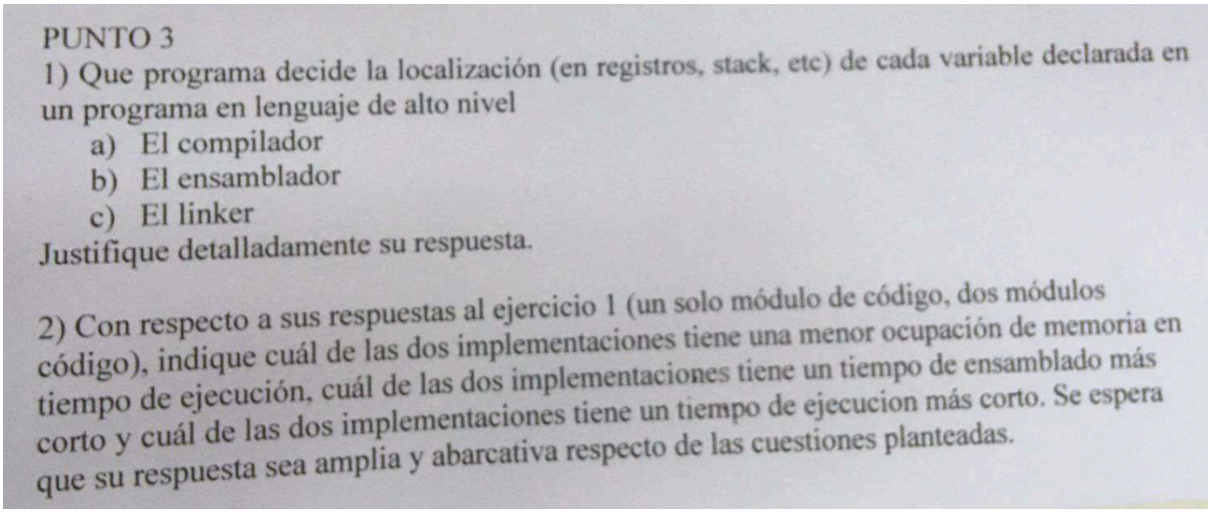
**DECO A**
- Entrada: 37
- Salida: 38 bits, todos en 0 menos el 37

**DECO B** no importa

**DECO C**

Entrada: 33

Salida: 38 bits, todos en 0 menos el 33



PUNTO 3

1) Que programa decide la localización (en registros, stack, etc) de cada variable declarada en un programa en lenguaje de alto nivel

a) El compilador
b) El ensamblador
c) El linker

Justifique detalladamente su respuesta.

2) Con respecto a sus respuestas al ejercicio 1 (un solo módulo de código, dos módulos código), indique cuál de las dos implementaciones tiene una menor ocupación de memoria en tiempo de ejecución, cuál de las dos implementaciones tiene un tiempo de ensamblado más corto y cuál de las dos implementaciones tiene un tiempo de ejecucion más corto. Se espera que su respuesta sea amplia y abarcativa respecto de las cuestiones planteadas.

1.

Recordemos qué realiza cada uno de estos programas:

El compilador realiza el pasaje de lenguaje de alto nivel a lenguaje ensamblador a través de los siguientes análisis:

- Análisis lexicográfico: reconoce los elementos básicos del programa.
- Análisis sintáctico: a partir de los elementos básicos, reconoce la estructura del programa.
- Análisis semántico:
  - Análisis de nombre: a cada nombre se le asigna una variable particular y una dirección de memoria.
  - Análisis de tipos: se reconocen todos los tipos de variables que el programa requiere.

Una vez realizados los análisis, se asignan las acciones y se escribe por cada línea de código en alto nivel, 1 o más líneas de lenguaje ensamblador.

**El compilador es la respuesta correcta, ya que es el que define, al traducir a Assembler los registros para cada variable.**

El ensamblador se encarga de pasar de lenguaje ensamblador a código de máquina (1 y 0). En ensamblador de ARC es de 2 pasadas, es decir que lee 2 veces el texto y eso permite el uso de símbolos de un programa antes de definirlos. Además de permitir el uso de macros, la ubicación de palabras en memoria, poder definir etiquetas para direcciones de memoria constantes, entre otras prestaciones.

En la primera pasada, principalmente se genera la tabla de símbolos, donde para cada uno de ellos se determina su valor, si es reubicable en memoria o no y si es global o extern. Esta tabla de símbolos será útil desde su creación hasta la ejecución del programa.

En la segunda pasada, se genera el código de máquina, siendo este y el listado que contiene la tabla de símbolos y por ejemplo, la ubicación de la primera línea a ejecutar lo que saldrá del ensamblador.

El linker se encarga de enlazar 2 o más módulos que han sido ensamblados de forma separada, además se encarga de resolver referencias globales y externas y además en caso de que dirección de memoria se superpongan entre módulos, este las separa y reubica.

2.

Ocupación de memoria en tiempo de ejecución: si pensamos en que se trata del mismo programa solo que todo en 1 y separado en 2, teniendo en cuenta que durante el enlace el programa se volverá a unir, podemos decir que la ocupación en memoria será la misma para ambos casos.

Tiempo de ensamblado: recordemos que el ensamblador de ARC es de 2 pasadas, por lo tanto para el caso de tener 2 módulos tendrán que hacer 4 pasadas, retrasando el tiempo de ensamblado.

Tiempo de ejecución: tomando en cuenta que al tener 2 módulos se realizan reubicaciones de direcciones de memoria, podemos decir que en ese caso tendremos más tiempo de ejecución ya que tendremos que realizar más saltos.