Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación
IIC2343 - Arquitectura de Computadores

Tarea 2

15 de marzo de 2019

Entrega: 23:59:59 del viernes 19.04.2019

1º semestre 2019 - **Profesor:** Yadran Eterovic

Requisitos

- Esta tarea es estrictamente individual. Cualquier tipo de falta a la honestidad académica será sancionada con la **reprobación** del curso con la nota mínima.
- Los ejercicios de tipo **programación** deberán ser realizados en `Python` 3.6.X.
- Los nombre de archivos y el cómo deben ser ejecutados son parte del formato, no respetarlo será penalizado.
- Los ejercicios de tipo **placa** deberán ser realizados en `VHDL`.
- Los ejercicios de tipo **teóricos** deberán ser contestados en un archivo `Markdown` y subirlo junto a su tarea, de nombre Respuestas.md, en el mismo repositorio.
- Toda tarea debe estar acompañada de un README.md que explique de forma clara el funcionamiento de la misma.
- Esta tarea deberá ser subida a su repositorio personal de *GitHub* correspondiente en la fecha y hora dada.

Ejercicios - Placa

1. Construcción de un computador.

Objetivos del ejercicio:

- familiarizarse con el concepto de **lenguaje de descripción de *hardware***, en particular, **VHDL**;
- familiarizarse con **Vivado**, un *software* desarrollado para la síntesis, análisis y programación de circuitos lógicos integrados;
- construir su propio **computador básico de 16 *bits***[1] cumpliendo con una serie de requisitos (explicados más adelante).

2. Introducción

En los últimos días se ha notado una falla generalizada en todos los dispositivos electrónicos alrededor del globo sin ninguna explicación aparente. Para calmar a la prensa todos los mayores expertos computacionales de distintas universidades y empresas han dicho que solo son tormentas solares. Sin embargo, la verdad es mucho más aterradora, ¡alguien viajo al pasado y se robo piezas y cables fundamentales del computador primigenio! Sin su existencia, el presente esta condenado.

De forma clandestina estudiantes del curso de arquitectura de computadores han sido escogidos para viajar en el tiempo y volver a armar este computador. Pero el viaje temporal es muy costoso e inestable, por lo que deberán demostrar que son aptos para hacer dicha travesía. Para eso hemos preparado una recreación lo más similar posible usando Vivado y VHDL. Donde deberán completar el computador primigenio, construyendo las piezas que faltan, y conectando todo de forma que se puedan ejecutar programas dentro de la máquina.

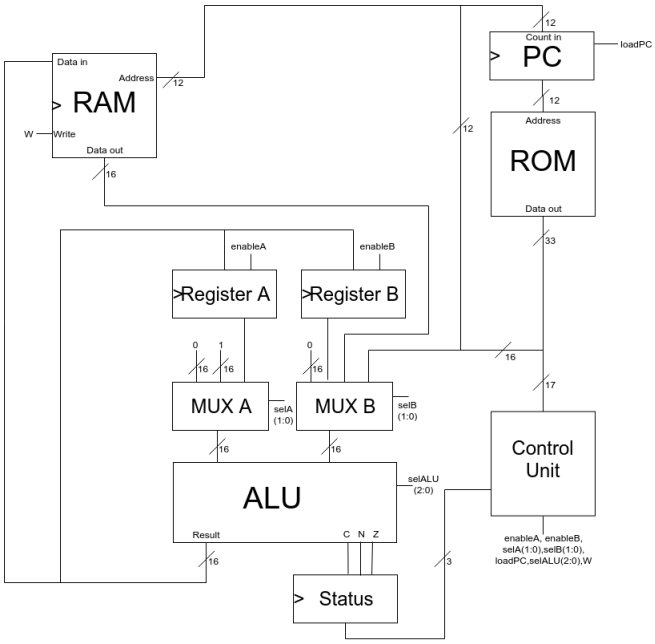
La nación, no, el mundo tecnológico tal y como lo conocemos los necesitan, completen esta labor ¡para ir al pasado lo más pronto posible!

[1] :o

Especificaciones técnicas del computador primigenio.

- Dos **registros de propósito general** de 16 *bits* (registros `A` y `B`);
- una **memoria de instrucciones** (ROM) de instrucciones de 4096 palabras, cada una de 33 *bits*;
- una **memoria de datos** (RAM) de 4096 palabras, cada una de 16 *bits*;
- una **unidad aritmética lógica** (ALU) de dos entradas, cada una de 16 *bits*, y que soporte las operaciones `ADD`, `SUB` `AND`, `XOR`, `SHL`, `SHR` y `NOT`;
- un **program counter** (PC) de 12 *bits* de direccionamiento;
- dos **multiplexores** (mux) que seleccionen entre cuatro entradas, cada una de 16 *bits*;
- una **unidad de control** (CU) de lógica combinacional (es decir, sin ciclos en los circuitos);
- un **registro *status*** que almacene los valores de *carry*, *zero* y *negative* (*bits* `c`, `z` y `n`, respectivamente).

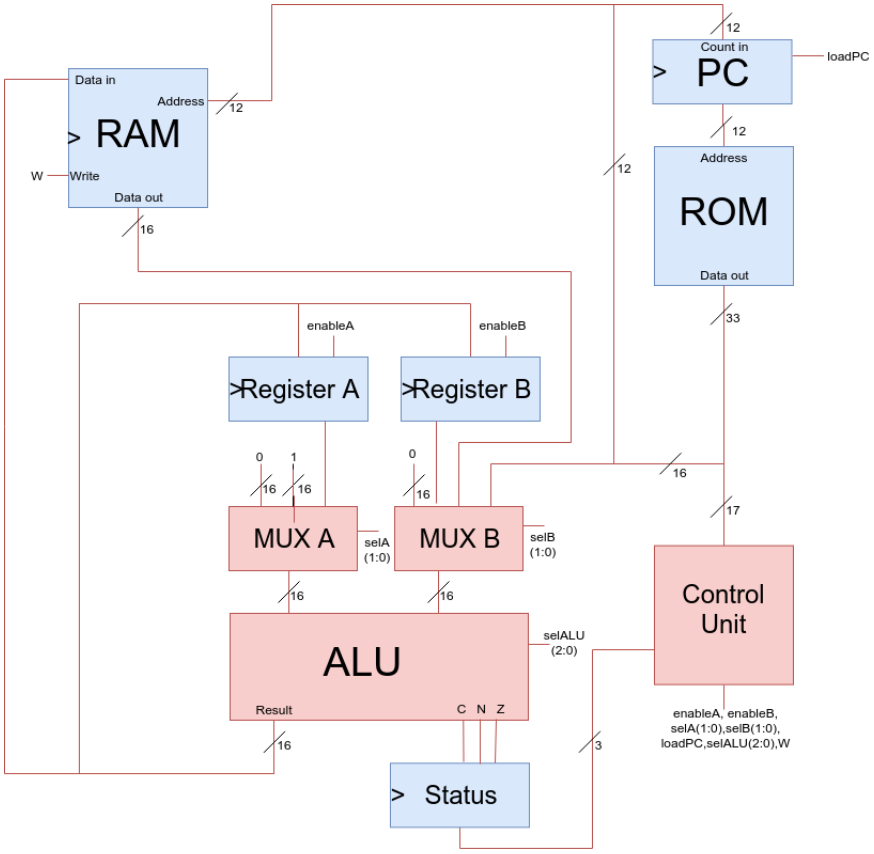
Diagrama del computador.



Componentes disponibles y por construir

En azul se puede ver los componentes que ustedes dispondrán, mientras que todo lo que esta en rojo fue lo que ladrón temporal se llevo y deben volver a construir. Estos siendo una la ALU de ocho operaciones, la control unit, los selectores de los resgistros y todas las conexiones entre componentes (los cables o como en VHDL se conoce señales).

NOTA: Si bien el componente de ROM estará dado, las instrucciones serán distintas a la hora de corregir.



Aspectos a considerar (requisitos).

Para implementar declaraciones condicionales **solamente** se permite hacer uso de bloques `with/select`. El uso de los *statements* `process`, `case` e `if/else` quedan absolutamente **prohibidos**. Esto porque se privilegia el uso de selectores y operaciones lógcias básicas para el desarrollo de esta tarea. (solo ocupar MUX, NOT, XOR, OR y AND).

Ejemplo with/select

```
with f select h <=
    "1000" when "00",
    "0100" when "01",
    "0010" when "10",
    "0001" when "11";
```

Esto se traduce a un **MUX**. En este caso cuando el valor se seleccion con el valor de f igual a 00 entonces seleccionaremos 1000 como valor de h. De la misma forma cuando f es igual 01 entonces seleccionaremos 0100 para el valor de h, y así sucesivamente

Además, las entidades PC, RAM, Reg y Status solamente pueden ser **instanciadas**, mas **no modificadas.**

Y por último, para evaluar la correctitud de su arquitectura, los *displays* de la placa deben mostrar, en todo momento, el valor de ambos registros de su computador (dos *displays* por cada registro, $display_a$ para los 8 bits más significativos del registro A, $display_b$ para los 8 bits menos significativos del registro A, $display_c$ para los 8 bits más significativos del registro B y por último $display_d$ para los 8 bits menos significativos del registro B).

Recomendaciones.

Se recomienda **revisar los tutoriales** de Vivado y VDHL del *syllabus* antes de comenzar a programar[2].

En caso de que las dudas persistan, recomendamos **agendar una reunión** presencial con alguno de los ayudantes. Dichas reuniones se encuentran sujetas a la disponibilidad de los ayudantes, por lo que se dará preferencia a quienes las soliciten **con anticipación**. Para agendar se debe completar un form que estará disponible en el README del syllabus.

Entrega

La tarea debe ser realizada por los grupos asignados y la entrega se realizará a través de GitHub. El repositorio debe contener una carpeta con su proyecto de Vivado y el archivo .bit. En el caso de la carpeta del proyecto, deben subir solo la carpeta basic computer.srcs y el archivo basic computer.xpr

[2]Técnicamente no sería programar, sino que describir el comportamiento de un circuito.

ISA a utilizar (placa)

Instrucción	Operandos	Operación	Opcode
MOV	A, B	A = B	00000000000000001
	B, A	B = A	00000000000000010
	A, Lit	A = Lit	00000000000000011
	B, Lit	B = Lit	00000000000000100
	A, (Dir)	A = Memory[Dir]	00000000000000101
	B, (Dir)	B = Memory[Dir]	00000000000000110
	(Dir), A	Memory[Dir] = A	00000000000000111
	(Dir), B	Memory[Dir] = B	00000000000001000
ADD	A, B	A = A + B	00000000000001001
	B, A	B = A + B	00000000000001010
	A, Lit	A = A + Lit	00000000000001011
	B, Lit	B = A + Lit	00000000000001100
	A, (Dir)	A = A + Memory[Dir]	00000000000001101
	B, (Dir)	B = A + Memory[Dir]	00000000000001110
	(Dir)	Memory[Dir] = A + B	00000000000001000

Instrucción	**Operandos**	**Operación**	**Opcode**
SUB	A, B	A = A - B	00000000000001001
	B, A	B = A - B	00000000000001010
	A, Lit	A = A - Lit	00000000000001011
	B, Lit	B = A - Lit	00000000000001100
	A, (Dir)	A = A - Memory[Dir]	00000000000001101
	B, (Dir)	B = A - Memory[Dir]	00000000000001110
	(Dir)	Memory[Dir] = A - B	00000000000001111
AND	A, B	A = A AND B	00000000000010000
	B, A	B = A AND B	00000000000010001
	A, Lit	A = A AND Lit	00000000000010010
	B, Lit	B = A AND Lit	00000000000010011
	A, (Dir)	A = A AND Memory[Dir]	00000000000010100
	B, (Dir)	B = A AND Memory[Dir]	00000000000010101
	(Dir)	Memory[Dir] = A AND B	00000000000010110
OR	A, B	A = A OR B	00000000000010111
	B, A	B = A OR B	00000000000011000
	A, Lit	A = A OR Lit	00000000000011001
	B, Lit	B = A OR Lit	00000000000011010
	A, (Dir)	A = A OR Memory[Dir]	00000000000011011
	B, (Dir)	B = A OR Memory[Dir]	00000000000011100
	(Dir)	Memory[Dir] = A OR B	00000000000011101
XOR	A, B	A = A XOR B	00000000000011110
	B, A	B = A XOR B	00000000000011111
	A, Lit	A = A XOR Lit	00000000000100000
	B, Lit	B = A XOR Lit	00000000000100001
	A, (Dir)	A = A XOR Memory[Dir]	00000000000100010
	B, (Dir)	B = A XOR Memory[Dir]	00000000000100011
	(Dir)	Memory[Dir] = A XOR B	00000000000100100
NOT	A, B	A = NOT A	00000000000100101
	B, A	B = NOT A	00000000000100110
	(Dir)	Memory[Dir] = NOT A	00000000000100111
SHL	A, B	A = SHL A	00000000000101000
	B, A	B = SHL A	00000000000101001
	(Dir)	Memory[Dir] = SHL A	00000000000101010
SHR	A, B	A = SHR A	00000000000101011
	B, A	B = SHR A	00000000000101100
	(Dir)	Memory[Dir] = SHR A	00000000000101101
INC	B	B = B + 1	00000000000101110
CMP	A, B	A - B	00000000000101111
	A, Lit	A - Lit	00000000000110000

Instrucción	**Operando**	**Operación**	**Condición**	**Opcode**
JMP	Dir	PC=Dir	Ninguna	00000000000110001
JEQ			Z = 1	00000000000110010
JNE			Z = 0	00000000000110011
JGT			N = 0 y Z = 0	00000000000110100
JGE			N = 1	00000000000110101
JLT			N = 0	00000000000110110
JLE			N = 1 o Z = 1	00000000000110111
JCR			C = 1	00000000000111000
NOP	Ninguno	Nada	Ninguna	00000000000000000

Ejercicios - Programación

Objetivos del ejercicio:

- Familiarizarse con los conceptos de *assembler* y *disassembler*;
- **Programar** estos para una ISA en particular;
- **Simular** la ejecución de un archivo binario en un computador básico de 8 *bits*;
- **Comparar y analizar** el comportamiento del *assembler* y el *disassemler*.

¿Qué es un *assembler* y un *disassembler*?

Cuando el computador desea ejecutar un programa cualquiera, este **no** recibe el codigo en Assembly, lenguaje con el que ya deberían estar familiarizados, ni en ningún otro lenguaje. Como habrán visto en clases, lo que usa el computador finalmente para ejecutar este programa es la **representación en binario** (opcode + literal ó literal + opcode) de instrucciones en Assembly. Es por esto que es necesario tener un programa que acutúe como compilador Assembly → Binario, a este programa se le llama *Assembler*. El *Disassembler*, en cambio, corresponde al proceso inverso, toma una secuencia de *opcodes* y las traduce a su equivalente en *Assemmbly* (Es un commpilador Binario → Assembly)

Requisitos de la tarea.

- Programar un *assembler* que compile un código escrito en *assembly* a un archivo binario de *opcodes*;
- Programar un *disassembler* que haga el proceso inverso: convertir un archivo binario compilado en código *assembly*;
- Escribir un programa en *Python*, que tenga como input una archivo binario compilado representando un programa en *assembly*, y a partir de él simular el comportamiento del computador básico durante su ejecución.

Assembler

Tu assembler deberá transformar un archivo en texto plano (.txt) y entregar un archivo binario (también .txt) con las especificaciones dadas a continuación. Junto con el enunciado podrán ver archivos de ejemplo.

Estructura del código assembly

1. Iniciamos con una sección `DATA`, donde se encuentra el nombre de cada una de las variables y su valor inicial, separados por un espacio.
2. Posteriormente, en una sección `CODE`, se encuentra el código de nuestro programa.
3. Para el direccionamiento directo solo se usarán los registros o variables declaradas en la sección `DATA`, nunca un literal por sí solo.
4. Cada línea contiene exactamente una instrucción.
5. No existen dos variables que se llamen igual.
6. Los comentarios inician con un "punto y coma" (`;`). Estos pueden estar o no en la misma línea que una instrucción.
7. Si una línea contiene una instrucción y un comentario, este siempre se encontrará **después** de la instrucción y todos sus operandos.

```
; esto SÍ es un comentario válido :)
<CÓDIGO ASSEMBLY>; esto también es un comentario válido :D
; <-- esto si es un comentario válido :D <CÓDIGO ASSEMBLY QUE NO SERÁ USADO>

esto NO es un comentario válido :(
<CÓDIGO ASSEMBLY> esto tampoco es un comentario válido :c
esto menos >:c <CÓDIGO ASSEMBLY>
```

8. Siempre deberá haber una *label* `main` y una *label* `end`. Esto marca dónde comienza y termina la ejecución respectivamente. Sin embargo, puede haber código tanto antes como después de la subrutina `main`.
9. Todos los valores numéricos serán enteros **con signo**.

Estructura del Archivo Binario

El archivo binario contará con cuatro secciones:

1. La primera corresponde a una sección dónde el código presente cargue en memoria de datos los valores de las variables. Debe realizarse en un orden incremental, es decir, la primera variable en aparecer va en la dirección 0x0000, la suguiente en 0x0001, y así sucesivamente hasta la última.
2. La segunda sección corresponde a una instrucción de salto hacia la dirección en la ROM donde comienza el bloque de instrucciones correspondiente a la subrutina main (debe ser agregada por el compilador).
3. La tercera sección, corresponde a todos los bloques de código. Estos deben estar en el orden en que fueron escritos.
4. Finalmente, la cuarta sección corresponde a cuatro *bytes* `0xFF`.

En el archivo binario, las instrucciones se presentan como pares `opcde - literal`. Los `opcodes` utilizarán un *byte*, al igual que los `literares`.

IMPORTANTE El archivo debe llamarse `basic_assembler.py`, y al igual que la tarea pasada, la ejecución de esta parte debe ser a través de la linea de comandos (terminal). Si aún no saben como lograr esto, lean el archivo AspectosFormalesIIC2343 en el Syllabus de curso.

Disassembler

Deberás crear un programa que pase de un archivo binario compilado a un código en assembly.

La primera sección del binario deberás convertirla en la sección `DATA` correspondiente. Usa nombres genéricos si lo necesitas. El `JMP` a `main` no deberá estar presente. Los bloques de código de la sección `CODE` deberán tener *labels*.

IMPORTANTE El archivo debe llamarse `basic_disassembler.py`, y al igual que la tarea pasada, la ejecución de esta parte debe ser a través de la linea de comandos (terminal). Si aún no saben como lograr esto, lean el archivo AspectosFormalesIIC2343 en el Syllabus de curso

Simulación

Deberás simular el funcionamiento del computador básico a partir de un archivo binario compilado. Todos los archivos que les proveamos cumplirán las especificaciones de formato antes declaradas.

La simulación debe tener como output un archivo `.txt`, donde por cada instrucción (t) que ejecute su programa, debe mostrar los estados de los siguientes componentes:

1. Número de iteración (servirá como guía).
2. Valor en el registro A (post ejecución).
3. Valor en el registro B (post ejecución).
4. Valor en el PC (Program counter) del `opcode`.
5. `Opcode` que se ejecuta.
6. `Literal` que acompaña al opcode.

Es importante mantener el orden especificado anteriormente. Todos los valores, salvo el de `t` deberán ser mostrados en hexadecimal sin el prefijo que lo indique. Si el valor de un registro no ha sido asignado todavía, coloque un guión en su lugar.

Un ejemplo de lo esperado es lo siguiente:

```
t  |  A   |  B  | PC | OPCODE | LITERAL
1  |  -   |   0 |  0 |   03   |    0
2  |   0  |   0 |  2 |   02   |    0
3  |   0  |   0 |  4 |   2D   |    0
4  |   0  |   1 |  6 |   2E   |    0
5  | - 1  |   1 |  8 |   02   |  - 1
6  | - 1  |   1 |  A |   2D   |    1
7  | -1F  |   2 |  C |   2E   |  -1F
8  | -1F  |   2 |  E |   02   |    0
9  | -1F  |   2 | 10 |   2D   |    2
10 | -1F  |   2 | 12 |   1E   |   22
11 |   2  |   2 | 14 |   00   |    0
.
.
.
```

IMPORTANTE El archivo debe llamarse `basic_simulator.py`, y al igual que la tarea pasada, la ejecución de esta parte debe ser a través de la linea de comandos (terminal). Si aún no saben como lograr esto, lean el archivo AspectosFormalesIIC2343 en el Syllabus de curso

ISA a utilizar (programación)

Instrucciones básicas

Instrucción	Operandos	Operación	Opcode
MOV	A, B	A = B	0x00
	B, A	B = A	0x01
	A, Lit	A = Lit	0x02
	B, Lit	B = Lit	0x03
ADD	A, B	A = A + B	0x04
	B, A	B = A + B	0x05
	A, Lit	A = A + Lit	0x06
SUB	A, B	A = A - B	0x07
	B, A	B = A - B	0x08
	A, Lit	A = A - Lit	0x09
AND	A, B	A = A AND B	0x0A
	B, A	B = A AND B	0x0B
	A, Lit	A = A AND Lit	0x0C
OR	A, B	A = A OR B	0x0D
	B, A	B = A OR B	0x0E
	A, Lit	A = A OR Lit	0x0F
NOT	A, B	A = NOT A	0x10
	B, A	B = NOT A	0x11
	A, Lit	A = NOT Lit	0x12
XOR	A, B	A = A XOR B	0x13
	B, A	B = A XOR B	0x14
	A, Lit	A = A XOR Lit	0x15
SHL	A, B	A = shift left A	0x16
	B, A	B = shift left A	0x17
	A, Lit	A = shift left Lit	0x18
SHR	A, B	A = shift right A	0x19
	B, A	B = shift right A	0x1A
	A, Lit	A = shift right Lit	0x1B

Saltos

Instrucción	Operando	Operación	Condición	Opcode
CMP	A, B	A - B	-	0x1C
	A, Lit	A - Lit	-	0x1D
JMP			-	0x1E
JEQ			Z = 1	0x1F
JNE			Z = 0	0x20
JGT			N = 0 y Z = 0	0x21
JGE	Dir	PC=Dir	N = 1	0x22
JLT			N = 0	0x23
JLE			N = 1 o Z = 1	0x24
JCR			C = 1	0x25
JOV			V = 1	0x26

Direccionamiento

Instrucción	Operandos	Operación	Opcode
MOV	A, (Dir)	A = Memory[Dir]	0x27
	B, (Dir)	B = Memory[Dir]	0x28
	(Dir), A	Memory[Dir] = A	0x29
	(Dir), B	Memory[Dir] = B	0x2A
	A, (B)	A = Memory[B]	0x2B
	B, (B)	B = Memory[B]	0x2C
	(B), A	Memory[B] = A	0x2D

Instrucciones adicionales

Instrucción	Operandos	Operación	Opcode
INC	B	B = B + 1	0x2E
NOP	-	-	0xFF

Entrega y evaluación

La tarea se debe realizar de **manera individual** tanto si se trabaja de forma programada o como un individuo de dos personas en caso de los grupos con placa. La entrega se realizará a través de GitHub. Archivos que no compilen y/o que no cumplan con el formato de entrega implicarán nota **1.0** en la tarea. En caso de atraso, se aplicará un descuento de **1.0** punto por cada 6 horas o fracción.

Política de Integridad Académica

Los alumnos de la Escuela de Ingeniería deben mantener un comportamiento acorde al Código de Honor de la Universidad:

"Como miembro de la comunidad de la Pontificia Universidad Católica de Chile me comprometo a respetar los principios y normativas que la rigen. Asimismo, prometo actuar con rectitud y honestidad en las relaciones con los demás integrantes de la comunidad y en la realización de todo trabajo, particularmente en aquellas actividades vinculadas a la docencia, el aprendizaje y la creación, difusión y transferencia del conocimiento. Además, velaré por la integridad de las personas y cuidaré los bienes de la Universidad."

En particular, se espera que mantengan altos estándares de honestidad académica. Cualquier acto deshonesto o fraude académico está prohibido; los alumnos que incurran en este tipo de acciones se exponen a un procedimiento sumario. Específicamente, para los cursos del Departamento de Ciencia de la Computación, rige obligatoriamente la siguiente política de integridad académica. Todo trabajo presentado por un alumno (grupo) para los efectos de la evaluación de un curso debe ser hecho individualmente por el alumno (grupo), sin apoyo en material de terceros. Por "trabajo" se entiende en general las interrogaciones escritas, las tareas de programación u otras, los trabajos de laboratorio, los proyectos, el examen, entre otros. Si un alumno (grupo) copia un trabajo, los antecedentes serán enviados a la Dirección de Docencia de la Escuela de Ingeniería para evaluar posteriores sanciones en conjunto con la Universidad, las que pueden incluir reprobación del curso y un procedimiento sumario. Por "copia" se entiende incluir en el trabajo presentado como propio partes hechas por otra persona. Está permitido usar material disponible públicamente, por ejemplo, libros o contenidos tomados de Internet, siempre y cuando se incluya la cita correspondiente.