Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2343 – Arquitectura de Computadores (II/2018)

Tarea 6

Fecha de entrega: jueves 11 de octubre de 2018 a las 11:59 AM

Introducción

Uno de los paradigmas arquitectónicos predominantes hoy en día corresponde a la arquitectura Von Neumann. En esta, a diferencia de la arquitectura Harvard utilizada en el computador básico del curso, tanto las instrucciones como los datos están almacenados juntos en la memoria principal del computador. La ventaja de esto, además de la reducción de costos, es la capacidad de poder crear programas para el computador, ocupando el propio computador. Al poder trabajar los programas como si fueran datos, permite crearlos directamente en el mismo equipo.

A partir del computador básico visto en clases, es posible construir una nueva versión que siga la arquitectura Von Neumann. A continuación, el diagrama de una **posible** implementación.

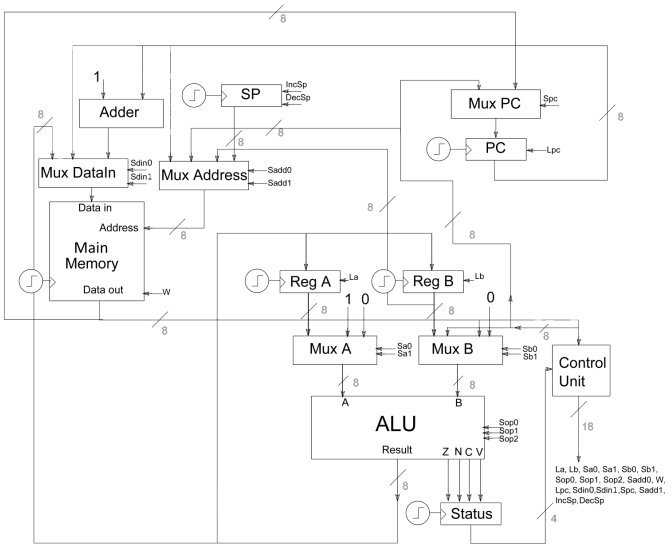

Figura 1: Diagrama del computador básico con arquitectura Von Neumann.

De esta arquitectura es importante notar que:

- Ahora existe una memoria común (`Main Memory`) que contiene tanto las instrucciones del programa como los literales y los datos de memoria.
- Ahora el valor `PC+1` se conecta directamente con el `Mux DataIn` dado que la dirección de retorno de una subrutina se almacena en la memoria principal.
- La salida de la memoria principal puede ser tanto un *opcode* como un literal o un dato de memoria, por lo que la Unidad de Control debe tratar con cuidado dichos casos.

Utilizando un subconjunto de la ISA del computador básico visto en clases, tanto la parte de programación como la parte práctica tendrán como objetivo trabajar con la arquitectura aquí presentada. Por otra parte, **se trabajará bajo el supuesto de que cada instrucción es ejecutada en tres ciclos** de la siguiente forma:

1. Se obtiene el *opcode* de la instrucción a ejecutar desde la memoria y se envía a la Unidad de Control.
2. Se obtiene el literal de la memoria principal, ubicado en la palabra contigua al opcode, y se propaga a todas las componentes conectadas a este.
3. Se deja un ciclo adicional para que se ejecute la instrucción correspondiente ya habiendo propagado las señales de control y el literal. Además, en este se permite un tercer acceso a la memoria principal en caso de que la instrucción lea o escriba sobre una palabra de esta.

La ISA del computador a implementar, entonces, se puede segmentar por funcionalidad como sigue:

Instrucciones de carga, aritméticas y lógicas

Instrucción	Operandos	Operación	*Opcode*
`MOV`	`A, B`	`A=B`	00000000
	`B,A`	`B=A`	00000001
	`A,Lit`	`A=Lit`	00000010
	`B,Lit`	`B=Lit`	00000011
`ADD`	`A,B`	`A=A+B`	00000100
	`B,A`	`B=A+B`	00000101
	`A,Lit`	`A=A+Lit`	00000110
`SUB`	`A,B`	`A=A-B`	00000111
	`B,A`	`B=A-B`	00001000
	`A,Lit`	`A=A-Lit`	00001001
`AND`	`A,B`	`A=A and B`	00001010
	`B,A`	`B=A and B`	00001011
	`A,Lit`	`A=A and Lit`	00001100
`OR`	`A,B`	`A=A or B`	00001101
	`B,A`	`B=A or B`	00001110
	`A,Lit`	`A=A or Lit`	00001111
`NOT`	`A,A`	`A=not A`	00010000
	`B,A`	`B=not A`	00010001
	`A,Lit`	`A=not Lit`	00010010
`XOR`	`A,B`	`A=A xor B`	00010011
	`B,A`	`B=A xor B`	00010100
	`A,Lit`	`A=A xor Lit`	00010101
`SHL`	`A,A`	`A=shift left A`	00010110
	`B,A`	`B=shift left A`	00010111
	`A,Lit`	`A=shift left Lit`	00011000
`SHR`	`A,A`	`A=shift right A`	00011001
	`B,A`	`B=shift right A`	00011010
	`A,Lit`	`A=shift right Lit`	00011011

Instrucciones de salto y comparación

Instrucción	Operandos	Operación	Condición	*Opcode*
CMP	A,B	A-B	-	00011100
	A,Lit	A-Lit	-	00011101
JMP	Dir	PC = Dir	-	00011110
JEQ	Dir	PC = Dir	Z=1	00011111
JNE	Dir	PC = Dir	Z=0	00100000
JGT	Dir	PC = Dir	N=0 y Z=0	00100001
JLT	Dir	PC = Dir	N=1	00100010
JGE	Dir	PC = Dir	N=0	00100011
JLE	Dir	PC = Dir	Z=1 o N=1	00100100
JCR	Dir	PC = Dir	C=1	00100101
JOV	Dir	PC = Dir	V=1	00100110

Instrucciones de memoria y direccionamiento

Instrucción	Operandos	Operación	*Opcode*
MOV	A,(Dir)	A=Mem[Dir]	00100111
	B,(Dir)	B=Mem[Dir]	00101000
	(Dir),A	Mem[Dir]=A	00101001
	(Dir),B	Mem[Dir]=B	00101010
	A,(B)	A=Mem[B]	00101011
	B,(B)	B=Mem[B]	00101100
	(B),A	Mem[B]=A	00101101

Instrucciones adicionales

Instrucción	Operandos	Operación	*Opcode*
INC	B	B=B+1	00101110
NOP	Ninguno	Nada	11111111

Además de la descripción anterior, cabe destacar lo siguiente:

- Si bien el diagrama presentado de referencia lo incluye, **no es necesario añadir** un *stack pointer* dado que no es necesario que implemente subrutinas. Si desea hacerlo, vea la sección bonus al final del enunciado.
- Notará que hay funcionalidades que no se explicitan (por ejemplo, la protección de la Unidad de Control al recibir literales o valores de memoria). Esto, dado que **usted debe pensar en cómo implementarlas.**
- Si un computador recibe un *opcode* desconocido para él, debe detener la ejecución en ese instante.

Parte programación

Su tarea se separará en tres partes programadas y una de preguntas, las que serán descritas a continuación.

Parte 1 - *Assembler*

Deberá programar el *Assembler* del computador básico con arquitectura Von Neumann.

Formato de entrada

Su programa recibirá como entrada un archivo `.txt` con el código escrito en un subconjunto del `Assembly` del computador básico, siguiendo exclusivamente la `ISA` detallada anteriormente. A continuación, un ejemplo de un código válido:

```
;Se carga A = 3, B = 5 y luego se guarda en la direccion de var1 el valor A+B.
MOV A,3
MOV B,5
ADD A,B
MOV (var1),A
;Note que ya no es necesario el segmento DATA, por lo que los
;labels de datos puede estar en cualquier linea del codigo.
;Mucho ojo con esto!!!!!
var1 0
...
```

Formato de salida

Su programa deberá retornar como salida un archivo `.txt` correspondiente a la composición completa de la memoria principal. En este, cada línea será un *string* de 8 bits que puede corresponder a: un *opcode* o a un literal por cada instrucción, o bien a un dato de memoria. Por ello, deberá seguir los siguientes supuestos:

1. El código del programa parte desde la posición de memoria 0, sin importar si la primera línea es un *label* o una instrucción.
2. El literal de un *opcode* siempre irá en la posición de memoria siguiente.

Para el mismo ejemplo anterior, se tendría entonces el siguiente resultado.[1]

```
00000010 -- Opcode MOV A, Lit
00000011 -- Literal 3
00000011 -- Opcode MOV B, Lit
00000101 -- Literal 5
00000100 -- Opcode ADD A,B
00000000 -- Literal 0
00101001 -- Opcode MOV (Dir),A
00001000 -- Literal 0, ubicacion de var1
00000000 -- var1
11111111 -- Termino programa
11111111 -- Termino programa
...
```

Como se mencionó en los supuestos, podrá notar que: i) siempre hay pares de líneas en la que el primer valor es el *opcode* de una instrucción y el segundo es su literal, y ii) el programa se ubica al principio de la memoria principal. Cabe destacar, no obstante, que la definición de datos en memoria ocupa **solo** una palabra en la memoria principal, donde el valor de esta corresponde al literal asignado y su ubicación dependerá de la

[1] Los comentarios en el archivo de ejemplo son para ilustrar, en su respuesta solo debe estar el código binario.

cantidad de variables e instrucciones que la antecedan en el código `Assembly`.

Además de lo anterior, podrá notar que las últimas líneas generadas corresponden a la palabra 255. Esta indicará el término de código y no generará ningún cambio en la ejecución del computador Von Neumann ni en sus registros, pero es necesario que el archivo binario generado contemple un total de 2^8 líneas dado que este representará la memoria principal. Por lo tanto, debe llenar el resto del archivo `.txt` con esta palabra hasta completar el tamaño esperado, en caso de ser necesario.

Ejecución

Su programa debe tener como nombre `von_neumann_assembler.py` y debe ser ejecutado por línea de comando de la siguiente forma:

```
C:\User\IIC2343\>python von_neumann_assembler.py program.txt mem.txt
```

En este caso, `program.txt` corresponde a la ruta del archivo de entrada y `mem.txt` a la ruta del nuevo archivo de salida a generar, correspondiente a la memoria principal resultante.

A considerar

Además de lo anterior, deben considerar lo siguiente:

- Si no se hace uso de un literal en la instrucción, puede poner el que quiera acompañando al *opcode* en el archivo binario generado, dado que no genera ningún efecto en la ejecución.
- El programa **puede** tener comentarios. Estos se ubican en líneas separadas de las instrucciones, es decir, nunca habrá un comentario antes o después de una instrucción en una misma línea. Estos parten con el caracter "*;*", igual que en el ejemplo.
- El número de instrucciones del código, sumado al número de labels declarados, nunca superará la cantidad de direcciones de la memoria principal.
- Para las instrucciones de salto sí se hace uso de *labels*, por lo que se debe cuidar que el literal correspondiente a la línea del código sea la dirección del primer *opcode* del segmento a ejecutar.

Parte 2 - Simulador

Deberá programar un simulador para el computador Von Neumann.

Formato de entrada

Su programa recibirá como parámetro por línea de comandos un archivo `.txt` con el contenido de la memoria principal, la que contiene código `binario` para el computador de acuerdo a las tablas de *opcodes* entregadas, además de literales y datos de memoria. Este corresponde **al mismo formato** del archivo de salida de la primera parte.

Formato de salida

Su programa deberá retornar como salida un archivo `.txt` donde cada línea contendrá: el número del ciclo del *clock*, el valor del registro `A`, el valor del registro `B`, el valor del registro `PC`, y el valor de memoria apuntado por `PC` (`Mem[PC]`). Por ejemplo, para el caso anterior, se tiene:

```
t = 0 00000000 00000000 00000000 00000010
t = 1 00000000 00000000 00000001 00000011
t = 2 00000000 00000000 00000001 00000011
t = 3 00000011 00000000 00000010 00000011
t = 4 00000011 00000000 00000011 00000101
t = 5 00000011 00000000 00000011 00000101
t = 6 00000011 00000101 00000100 00000100
t = 7 00000011 00000101 00000101 00000000
t = 8 00000011 00000101 00000101 00000000
t = 9 00001000 00000101 00000110 00101001
...
```

En el caso de que la simulación reciba una palabra **no identificada** como un *opcode*, debe detener su ejecución y generar un mensaje de error.

Ejecución

Su programa debe tener como nombre `von_neumann_simulator.py` y debe ser ejecutado por línea de comando de la siguiente forma:

```
C:\User\IIC2343\>python von_neumann_simulator.py mem.txt result.txt
```

En este caso, `mem.txt` corresponde a la ruta del archivo de entrada y `result.txt` a la ruta del nuevo archivo de salida a generar. Si esta ejecución tuvo un error, la última línea de `result.txt` debe indicarlo.

A considerar

Además de lo anterior, deben considerar lo siguiente:

- El archivo de entrada siempre tendrá un formato válido y correspondiente al establecido en este enunciado.
- Si bien el objetivo de esta parte es hacer uso de los resultados de la anterior, deben funcionar de forma independiente, es decir, puede realizar el simulador sin hacer el *assembler*.

Parte 3 - *Disassembler*

Deberá programar un *Disassembler* para el computador básico con arquitectura Von Neumann.

Formato de entrada

Su programa recibirá como parámetro por línea de comandos un archivo `.txt` con el contenido de la memoria principal, la que contiene código `binario` para el computador de acuerdo a las tablas de *opcodes* entregadas, además de literales y datos de memoria.

Formato de salida

Su programa deberá retornar como salida un archivo `.txt` donde cada línea corresponderá a una instrucción reconstruida del archivo binario recibido como *input*. Se sigue el mismo formato que el archivo de entrada del *Assembler*.

En el caso de que el *Disassembler* no pueda reconstruir una instrucción al **no identificar** un *opcode*, debe detener su ejecución y añadir un mensaje de error al final del archivo de salida, como comentario.

Ejecución

Su programa debe tener como nombre `von_neumann_disassembler.py` y debe ser ejecutado por línea de comando de la siguiente forma:

```
C:\User\IIC2343\>python von_neumann_disassembler.py mem.txt code.txt
```

En este caso, `mem.txt` corresponde a la ruta del archivo de entrada y `code.txt` a la ruta del nuevo archivo de salida a generar. Si esta ejecución tuvo un error, la última línea de `code.txt` debe indicarlo.

Parte 4 - Preguntas

A partir de sus resultados en las partes anteriores, deberá ejecutar los archivos de *input* disponibles en el Syllabus del curso y contestar las siguientes preguntas:

1. **Pregunta 1.** ¿Existen diferencias entre el código `Assembly` original y el generado por el *Disassembler*? ¿Cuáles? ¿Cómo y por qué se genera cada una de ellas?
2. **Pregunta 2.** Explique qué sucede al ejecutar en su simulador el código binario generado por su *Assembler* a partir de los archivos de *input* disponibles en el Syllabus.

Estas preguntas deben ser respondidas **al principio** de su `README`, para posteriormente incluir la información adicional descrita en la sección "Entrega y Evaluación" más adelante. Además, en una carpeta llamada `preguntas` debe adjuntar los archivos `.txt` generados por *su tarea*, que apoyen sus respuestas.

Supuestos

Tanto para la primera como para la segunda parte, puede hacer uso de **supuestos** en casos límite o elementos que no hayan sido explicitados en el enunciado. No obstante, para que estos sean válidos durante la corrección, **debe** explicitarlo en su `README`.

No se aceptarán supuestos sobre criterios que hayan sido establecidos en el enunciado.

Parte práctica

Objetivo

Implementar en *hardware* el computador básico con la estructura de Von Neumann, dando solución a los desafíos que involucra una arquitectura de ese tipo.

Descripción de la actividad

Deben implementar el computador con arquitectura Von Neumann descrito en la sección común del enunciado, implementando **solo** la `ISA` entregada en las tablas. Para comenzar a partir de una arquitectura previa, se les facilitará un proyecto base con el computador básico, el que deben[2] modificar para lograr la nueva máquina.

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* para cada uno). Además, deben conectar la salida de la memoria a los 8 leds menos significativos. Al alimentar el registro `PC` con la señal `clock` del proyecto, podrá ver cómo van cambiando sus valores a través de cada instrucción.

Si lo desea, puede usar otros elementos de la placa para enriquecer el funcionamiento de su tarea. Por ejemplo, que los *switches* afecten la frecuencia del `clock` para que el *display* cambie más rápido o más lento. No obstante, esto **no es un requisito para esta tarea.**

A considerar

Admeás de los anterior, deben considerar lo siguiente:

- **Pueden** usar las instrucciones `Process` y el bloque `with\select` para su tarea.
- **No pueden** usar operaciones aritméticas como suma y resta, tienen a disposición componentes que se encargan de esto.

ProTips de Vivado

Si es la primera vez que usa Vivado, se recomienda revisar los tutoriales del `Syllabus/Vivado`, especialmente el tutorial 2 y el archivo `intro_vhdl.pdf`. Si siguen teniendo dudas, pueden solicitar una reunión presencial (estas posibilidades están sujetas a la disponibilidad de los ayudantes).

Además, pueden revisar en Vivado el diagrama de su circuito usando lo siguiente:

[2] Pueden diseñarla desde cero, si así lo desean.

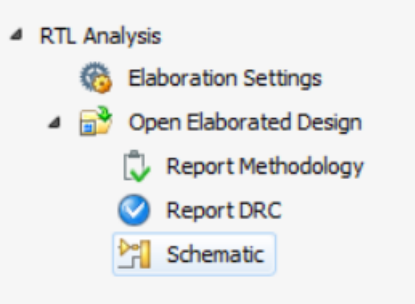

Figura 2: Menú de selección para ingresar al *Schematic*.

Verán un esquema que muestra las instancias de sus componentes y las conexiones entre ellos:

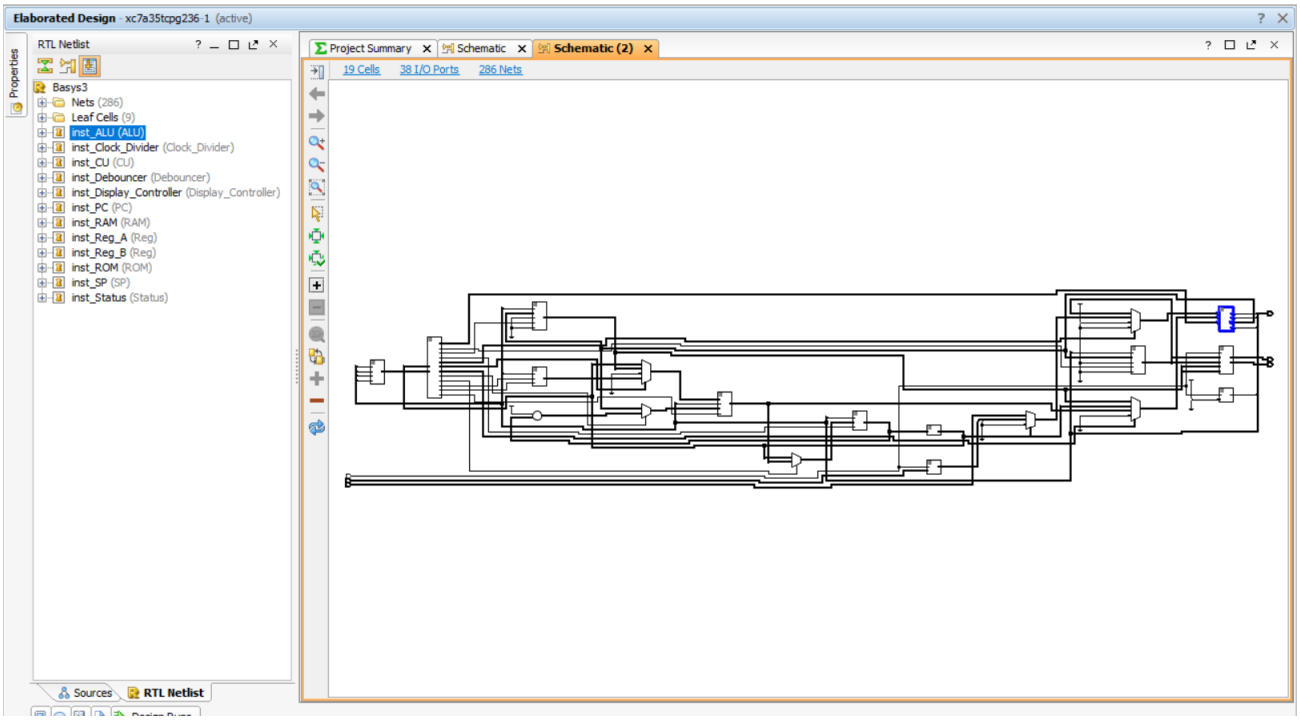

Figura 3: *Schematic* del computador básico.

Supuestos

Puede hacer uso de **supuestos** en casos límite o elementos que no hayan sido explicitados en el enunciado. No obstante, para que estos sean válidos durante la corrección, **debe** explicitarlo en su `README`.

No se aceptarán supuestos sobre criterios que hayan sido establecidos en el enunciado.

Bonus

Independiente de la parte trabajada, podrá optar a un bonus de hasta **1.0 puntos** si le añade[3] al computador con arquitectura Von Neumann **todas las instrucciones de subrutinas vistas en clases**. Para este bonus, deberá añadir:

- Un diagrama de su arquitectura modificada[4].
- La nueva `ISA` de su arquitectura.
- Detalles de su implementación. En particular, se espera una explicación detallada de cómo funciona su *stack pointer* y el *stack* de su memoria principal.

Estos elementos deben ser incluidos en un archivo llamado `Bonus.pdf` dentro del repositorio de su tarea. El puntaje asignado queda a criterio del ayudante según el trabajo realizado. Si decide no hacer el bonus, no es necesario que incluya el archivo. No obstante, si quiere optar a la bonificación **debe incluirlo.**

Se recomienda implementar el bonus después de haber terminado la tarea y no incluirlo desde el principio.

Entrega y evaluación

Parte programación

La tarea se debe realizar de **manera individual** y la entrega se realizará a través de GitHub. El formato de entrega serán los *scripts* en Python 3.5 o 3.6 correspondientes, de nombre `von_neumann_assembler.py`, `von_neumann_simulator.py` y `von_neumann_dissassembler.py` como se estipuló anteriormente, además del `README` con las respuestas de las preguntas y sus datos (nombre, número de alumno) y supuestos.

Parte práctica

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`. **Considere que esta parte incluirá una evaluación de pares, que será detallada durante la entrega.**

Independiente de la parte trabajada en esta tarea, el repositorio subido debe contar con un archivo `README.md` escrito en *Markdown* que identifique sus datos y consideraciones que el corrector deba tomar en cuenta, tales como la versión en Python utilizada, sistema operativo usado para realizar la tarea, etc. El `README` se puede subir hasta 24 horas después de la entrega. Si lo sube posterior al plazo de entrega establecido, debe crear una *issue* en el repositorio de su tarea indicándolo para que sea considerado en la corrección. Los archivos que no ejecuten o que no cumplan el formato de entrega establecido implicarán nota **1.0** en la tarea, la que podrá ser corregida con un descuento asociado dependiendo del caso. En caso de atraso, se aplicará un descuento de **1.0** punto por cada 6 horas o fracción.

[3] No basta con diseñarlo para optar al puntaje completo, es necesario que lo implemente en su programa y que funcione correctamente.

[4] Puede crear una nueva y no necesariamente hacer uso del ejemplo de este enunciado.

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.