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

Tarea 3

26 de abril de 2019

Entrega: 21:59:59 del viernes 10.05.2019

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.
- El uso material web sin referencia y tareas pasadas será sancionado con la nota mínima y citación con el profesor a cargo.
- 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 con la nota mínima.
- 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, el no hacerlo conllevará un descuento en su nota.
- Esta tarea deberá ser subida a su repositorio personal de *GitHub* correspondiente en la fecha y hora dada.

1. Introducción

Hace algunos meses, durante el duro inverno, Jessica, Felipe y José se vieron atrapados por una tormenta de nieve en las montañas nevadas del norte mientras buscaban el misterioso computador primogenio para corregir las tareas de sus adorados ayudantados. Sin capacidad de comunicarnse con el resto del cuerpo de ayudantes, Jessica ordenó crear un computador básico que permitiera amplificar las señales y codificarlas para ser rescatados. Felipe se encargó de la microarquitectura, José de la ISA que usarían y Jessica se fue a nadar a las aguas termales cercanas para supervisar su trabajo ~~o eso dijo~~.

Entre las piezas que consiguieron armar y reunir de entre la espesa nieve, se toparon con ciertas limitaciones: tenían solamente una memoria y los registros que habían disponibles eran de 12 bits, no de 8. Es por esto que no nos quedó más opción que hacer un computador con una microarquitectura *Von Neumann* con palabras de 12 bits.

Su trabajo será ayudarles a crear el computador que intentan construir los ayudantes, con sus escasos recursos, para salvar sus vidas ~~y quizás, ganarse su favor~~.

2. Aspectos generales del diseño

El computador, bautizado como *VonTwelve*, está compuesto por una única memoria de 2^{12} palabras de 12 bits (todas direccionables), en la que se almacenan tanto los datos como las instrucciones, además de tener soporte para saltos y subrutinas. Por lo tanto, todos los datos/literales (que pueden ser tanto números naturales como enteros) y todos los *opcode*s son de 12 bits. Además, cuenta con registros de uso general Ax, Bx, un registro para manejo de *stack* convencional (SP) y otro de ***stack* avanzado (BP)** cada uno de 12 bits.[1]

Nota: Este tipo de arquitectura donde se utiliza una sola memoria compartida entre las instrucciones y los datos, se llama **Arquitectura *Von Neumann***, la cual permite escribir instrucciones como si estas fueran datos. Una ventaja de esto es la capacidad de poder crear programas para el computador, ocupando el propio computador[2]. En otras palabras, al poder trabajar los programas como si fueran datos, permite crearlos directamente en el mismo equipo. **Podría ser útil investigar sobre este tipo de arquitectura.** [3]

[1] Se irán explicando a medida que se avance en el enunciado

[2] Mind Blown O:!

[3] Se recomienda leer capitulo 2.1 de la sección Arquitectura de Computadores en los apuntes del syllabus.

3. Manejo de *stack* avanzado y uso del registro BP

Con la finalidad de poder codificar las señales de la mejor forma, *VonTwelve* tiene un mayor poder en el uso de las subrutinas en comparación al computador visto en la tarea pasada. En ese equipo, el *stack* almacenaba la dirección de retorno. No había una convención sobre donde almacenar los parámetros y valores de retorno. En *VonTwelve* se utiliza el *stack* de manera más explícita con : **parámetros, retorno y variables locales.**

Para hacer todo esto posible, el uso del registro BP (base pointer) es fundamental para facilitar el manejo de todos estos datos. Al igual de la definición de una **convención de llamada.** Las convenciones definen la interfaz sobre la cual trabajará el código de la subrutina.[4]

Aquí ocuparemos la convención de llamada *stdcall*, que es la usada por la *API Win32* de *Microsoft*. La convención stdcall especifica los siguientes tres puntos:

1. Los parámetros son pasados de derecha a izquierda, usando el *stack*.
2. El retorno se almacenará en el registro Ax.
3. La subrutina se debe encargar de dejar SP apuntando en la misma posición que estaba antes de pasar los parámetros.

Gracias a stdcall, SP y BP, *VonTwelve* permiten tener llamadas anidadas de subrutinas (recursión) y variables locales.

4. Funcionamiento del computador

En *VonTwelve* 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 al registro CU, el cual lo almacena y libera una señal de control $stalingPC = 0$ para que se ejecute y continúe con el siguiente valor en memoria (que será un literal)
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. Durante este ciclo el registro CU no almacena el valor que obtiene de entrada. Luego, se libera una señal de control $stalingPC = 1$ para que se propague el resto de señales de control en el siguiente ciclo.
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. En este ciclo solo se ejecuta, no se lee la siguiente instrucción de memoria.

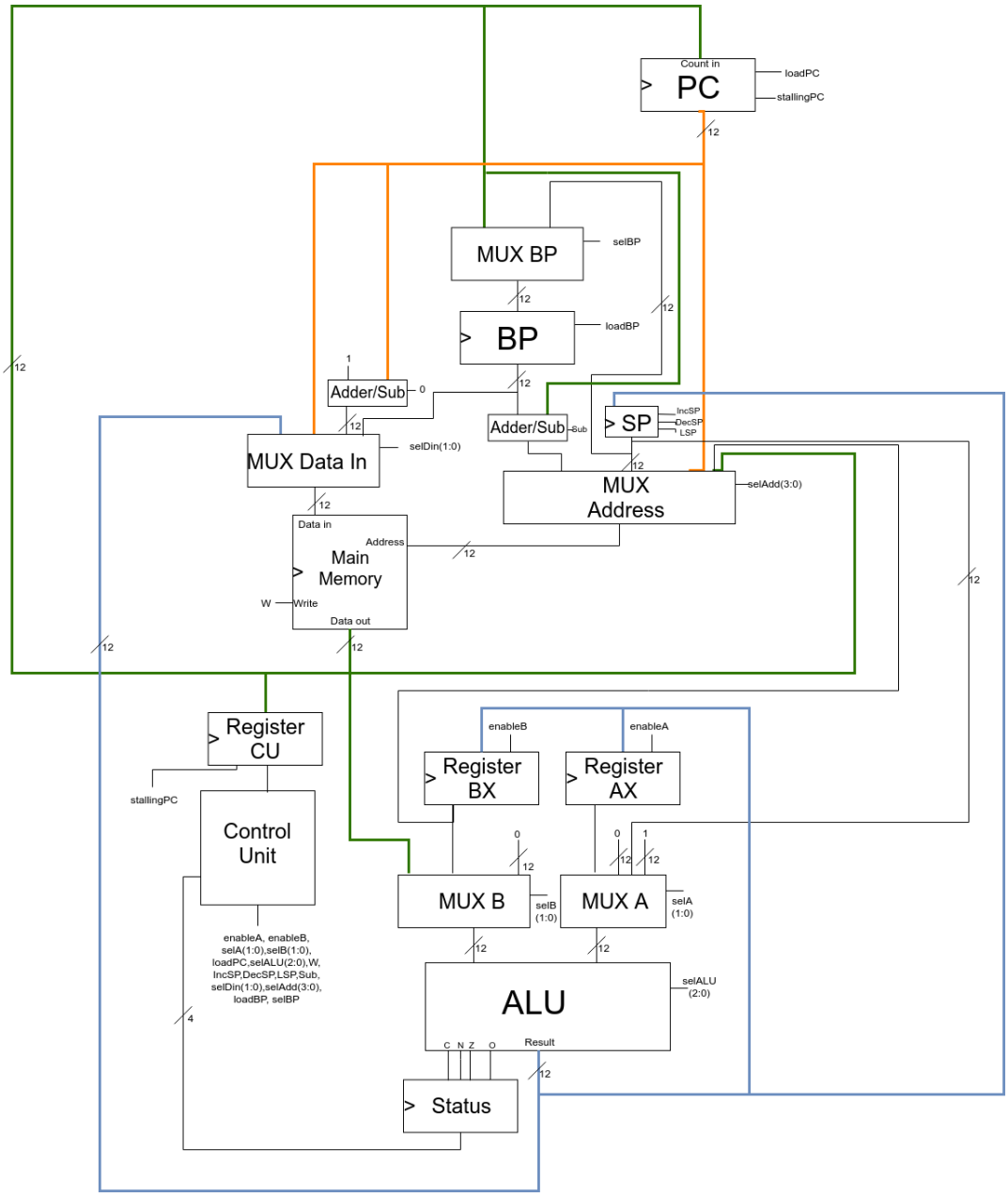
[4]Es **fuertemente recomendado** leer capitulo 2.4 de la sección Arquitectura de Computadores en los apuntes del syllabus.

5. Parte placa

Especificaciones técnicas del computador VonTwelve.

- Dos **registros de propósito general** de 12 *bits* (registros `Ax` y `Bx`).
- Un **registro de instrucciones** de 12 *bits* (registro `CU`).
- Un **registro de *stack*** de 12 *bits* (registro `SP`).
- Un **registro de puntero a base** de 12 *bits* (registro `BP`).
- Una **memoria principal** (*Main Memory*) de $2^{12} = 4096$ palabras, cada una de 12 *bits*.
- Una **unidad aritmética lógica** (ALU) de dos entradas, cada una de 12 *bits*, y que soporte las operaciones `ADD`, `SUB`, `AND`, `SHL 1`, `SHR 1` y `NOT`.
- Un ***program counter*** (PC) de 12 *bits* de direccionamiento.
- Dos **multiplexores** (MUX1 y MUX2) que seleccionen entre dos entradas, cada una de 12 *bits*.
- Un **multiplexor** (MUX3) que selecciona el dato de entrada de la *Main Memory,* cada una de 12 *bits*.
- Un **multiplexor** (MUX4) que seleccionen entre la dirección a que apuntará la *Main Memory*, cada una de 12 *bits*.
- Dos ***Adder/Sub*** que dados dos valores de 12 bits da el resultado de su suma o su resta dependiendo de una señal de entrada que puede ser cero o uno; y su salida también es de 12 *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*, *negative* y *overflow* (*bits* `c`, `z`, `n` y `o`, respectivamente).

Diagrama del computador.

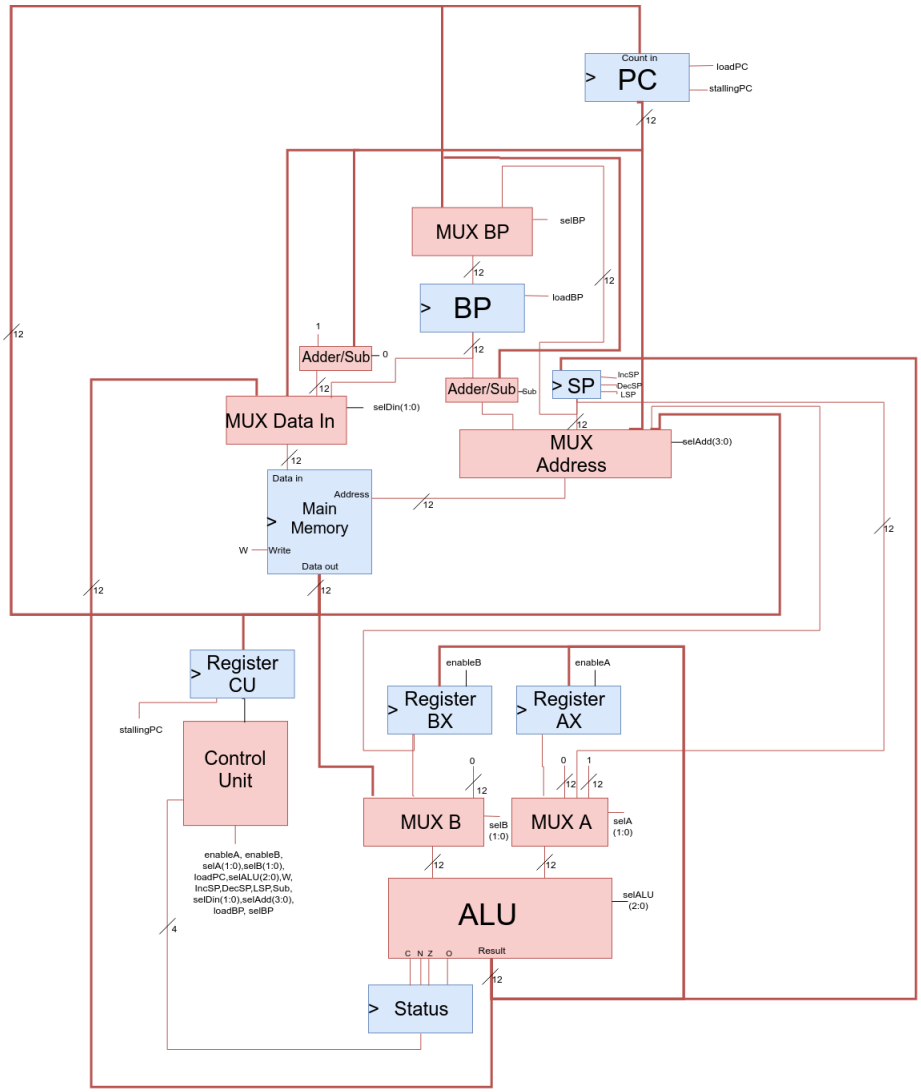


NOTA: Los cables de colores son para diferenciar entre conexiones sobrepuestas. Todo cable azul es de salida de la ALU. Todo cable verde es salida de la *Main Memory*. Todo cable naranjo es salida del PC.

Componentes disponibles y por construir

En el diagrama a continuación se utilizan colores para diferenciar las componentes ya disponibles para su uso de las componentes que ustedes deberán construir. En azul están marcadas las componentes disponibles, mientras que todo lo que esta en rojo es lo que la tormenta se llevó y deben volver a construir. Estos últimos son una ALU de seis operaciones, la *Control Unit*, los *ADDer*/sub, todos los selectores y todas las conexiones entre componentes, es decir, los cables (o, como se conoce en VHDL, las señales).

NOTA: Si bien la memoria ya está dada, 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**. Siguiendo el ejemplo, cuando el valor de `f` sea 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 para todos los casos. Además, las entidades *PC, Main Memory, Reg, RegCU, SP, BP 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 los 8 bits menos significativos de ambos registros de su computador. El *display* B mostrará los 4 bits menos significativos del registro Ax y el *display* A mostrará los siguientes 4 bits menos significativos del mismo registro. Por ejemplo, si el registro Ax guarda el valor 001001101110, el *display* B mostrará 1110 y el *display* A mostrará 0110. Análogamente, el *display* D mostrará los 4 bits menos significativos del registro Bx y el *display* C mostrará sus siguientes 4 bits menos significativos.

Recomendaciones.

Se recomienda **revisar los tutoriales** de Vivado y VDHL del *syllabus* antes de comenzar a programar[5].

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 mismos, 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.md del Syllabus.

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

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`.

6. Parte programada

Su tarea será crear tres programas en específico:

1. `assembler_x86.py`: convierte código Assembly en código máquina.
2. `disassembler_x86.py`: convierte código máquina en código Assembly.
3. `simulador_x86.py`: permite simular la ejecución de un código máquina.

Código Assembly x86

Un programa en código Assembly de x86 no está dividido en secciones `DATA` y `CODE`. Las instrucciones están al inicio del programa mientras que las variables se declaran al final, como se puede ver en el siguiente ejemplo.

```
; multipliquemos tres por cuatro

loop:
MOV  Ax, [resultado]
MOV Bx, [x]
ADD  Ax, Bx
MOV resultado,  Ax
MOV  Ax, [cont]
ADD  Ax, 1
MOV cont,  Ax
MOV  Ax, [y]
MOV Bx, [cont]

new_iteration:
CMP  Ax, Bx  ;
JG loop    ; salta si   Ax - Bx > 0

end_program:
MOV  Ax, [resultado]
RET

data:
x 3
y 4
resultado 0
cont 0
```

También debes notar que los nombres de *labels* y variables están en minúsculas.

Código Máquina *VonTwelve*

Por convención, cuando el literal no sea usado en la instrucción, este será nulo (cero). Para dar una idea de lo que es espera, el programa anterior compila a esto (los comentarios y salto de linea son para una mejor comprensión del *output*):

```
0o0036 ; MOV  Ax, Dir
0o0021 ; Dir=var:resultado

0o0037 ; MOV Bx, Dir
0o0017 ; Dir=var:x

0o0005 ; ADD  Ax, Bx
0o0000 ; ---

0o0040 ; MOV Dir,  Ax
0o0022 ; Dir=var:resultado

0o0036 ; MOV  Ax, Dir
0o0022 ; Dir=var:cont

0o0007 ; ADD  Ax, Lit
0o0001 ; Lit=1

0o0040 ; MOV Dir,  Ax
0o0022 ; Dir=var:cont

0o0036 ; MOV  Ax, Dir
0o0020 ; Dir=var:y

0o0037 ; MOV Bx, Dir
0o0022 ; Dir=var:cont

0o0027 ; CMP  Ax, Bx
0o0000 ; ---

0o0033 ; JG Dir
0o0000 ; Dir=label:start

0o0036 ; MOV  Ax, Dir
0o0021 ; Dir=var:resultado
```

```
0o0051 ; RET (primer ciclo)
0o0000 ; ---

0c0052 ; RET (segundo ciclo)
0o0000 ; ---

0o0003 ; x 3 (x=3 inicialmente)
0o0004 ; y 4 (y=4 inicialmente)
0o0000 ; resultado 0 (resultado=0 inicialmente)
0o0000 ; cont 0 (cont=0 inicialmente)

0o0000 ; NOP
0o0000 ; ---

.
.
.

0o0000 ; NOP
0o0000 ; ---
```

Los números son representados en base 8 (octal) debido a la facilidad para transformar desde y hacia binarios de 12 *bits*. A continuación se muestra un ejemplo de dicha transformación.

```
0o5261
```

```
5: 101
2: 010
6: 110
1: 001
```

Entonces,

```
0o5261: 101 010 110 001: 101010110001 ->12 bits.
```

Es decir, que cada dígito en la representación octal, es el valor en base decimal (10) de un numero binario (base 2) de 3 *bits*. Considere además que la representación negativa de los números considera el complemento a 8[6] e implementarlo como tal. Operaciones que no sigan estas instrucciones serán reconocidas como erroneas por el computador *VonTwelve*.

[6] si, sí existe

Assembler

`Assembler_x86.py`

Recibe el nombre del archivo `.txt` con el código *Assembly* y el nombre del archivo `.txt` en el que se escribirá el *output*, en ese orden[7].

En este caso, el *output* corresponde al código compilado tal como se presentó en la sección anterior.

Disassembler

`disassembler_x86.py`

Recibe el nombre del archivo `.txt` con el código máquina y el nombre del archivo `.txt` en el que se escribirá el output, en ese orden.

En este caso, el *output* correponde al código *Assembly* tal como se presentó en la sección anterior.

Simulador

`simulador_x86.py`

Recibe el nombre del archivo `.txt` con el código máquina y el nombre del archivo `.csv` en el que se escribirá el output, en ese orden.

En este caso, el *output* corresponde a un archivo *csv* en el cual cada línea deberá contener la siguiente información (en este orden y en su representación octal) separada por coma (`,`). Considere que los registros comienzan inicializados como nulos (en cero).

- El valor almacenado en el registro `PC` **al momento** de la ejecución de la instrucción.
- El valor almacenado en el registro `SP` **previo** a la ejecución de la instrucción.
- El valor almacenado en el `BP` **previo** a la ejecución de la instrucción.
- El *opcode*.
- El literal.
- El valor almacenado en el registro `Ax` **previo** a la ejecución de la instrucción.
- El valor almacenado en el registro `Bx` **previo** a la ejecución de la instrucción.

[7] revisar `AspectosFormalesIIC2343.pdf` en el Syllabus para mas detalle en caso de tener dudas

Para el ejemplo de la sección anterior (el del programa que multiplica), las primeras cinco líneas del *output* serían:

```
0o0000,0o0000,0o0000,0o0036,0o0021,0o0000,0o0000
0o0001,0o0000,0o0000,0o0037,0o0017,0o0000,0o0000
0o0002,0o0000,0o0000,0o0005,0o0000,0o0000,0o0003
0o0003,0o0000,0o0000,0o0040,0o0022,0o0003,0o0003
0o0004,0o0000,0o0000,0o0036,0o0022,0o0003,0o0003
```

NOTA: Un archivo `csv` puede ser trabajado igual que un archivo `txt`, la única diferencia es que se debe crear/abrir con la extensión correcta. Reiteramos la importancia de que cada linea escrita debe tener los valores separados por comas (`,`) y cada fila debe estar separada con un salto de línea (`\n`) al final. La gracia de los `csv` es que pueden ser renderizados como una tabla, tal como muestra la imágen de a continuación.

1,2,3,4,5
6,7,8,9,10
11,12,13,14,15
16,17,18,19,20

$\rightarrow$

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

Instrucciones básicas

Instrucción	Operandos	Operación	*opcode*
MOV	Ax, Bx	Ax = Bx	0o0001
	Bx, Ax	Bx = Ax	0o0002
	Ax, Lit	Ax = Lit	0o0003
	Bx, Lit	Bx = Lit	0o0004
ADD	Ax, Bx	Ax = Ax + Bx	0o0005
	Bx, Ax	Bx = Ax + Bx	0o0006
	Ax, Lit	Ax = Ax + Lit	0o0007
SUB	Ax, Bx	Ax = Ax - Bx	0o0010
	Bx, Ax	Bx = Ax - Bx	0o0011
	Ax, Lit	Ax = Ax - Lit	0o0012
AND	Ax, Bx	Ax = Ax and Bx	0o0013
	Bx, Ax	Bx = Ax and Bx	0o0014
	Ax, Lit	Ax = Ax and Lit	0o0015
NOT	Ax	Ax = not Ax	0o0016
	Bx	Bx = not Bx	0o0017
	Lit	Ax = not Lit	0o0020
SHL	Ax	Ax = shift left Ax	0o0021
	Bx	Bx = shift left Bx	0o0022
	Lit	Ax = shift left Lit	0o0023
SHR	Ax	Ax = shift right Ax	0o0024
	Bx	Bx = shift right Bx	0o0025
	Lit	Ax = shift right Lit	0o0026

Saltos

Instrucción	Operando	Operación	Condición	*opcode*
CMP	Ax, Bx	Ax - Bx	-	0o0027
	Ax, Lit	Ax - Lit	-	0o0030
JMP	Dir	PC=Dir	-	0o0031
JE			Z = 1	0o0032
JG			N = 0 y Z = 0	0o0033
JC			C = 1	0o0034
JO			V = 1	0o0035

Direccionamiento

Instrucción	Operandos	Operación	*opcode*
MOV	Ax, [Dir]	Ax = Memory[Dir]	0o0036
	Bx, [Dir]	Bx = Memory[Dir]	0o0037
	[Dir], Ax	Memory[Dir] = Ax	0o0040
	[Dir], Bx	Memory[Dir] = Bx	0o0041

Subrutinas

Instrucción	Operandos	Operaciones	*opcode*
PUSH	Ax	Memory [SP] = Ax ; SP = SP - 1	0o0042
	Bx	Memory [SP] = Bx ; SP = SP - 1	0o0043
POP	Ax	SP = SP + 1	0o0044
		Ax = Memory [SP]	0o0045
	Bx	SP = SP + 1	0o0046
		Bx = Memory [SP]	0o0047
CALL	Dir	Memory [SP] = pc ; SP = SP - 1 ; pc = Dir	0o0050
RET		SP = SP + 1	0o0051
		pc = Memory [SP]	0o0052

Manejo de *stack* avanzado

Instrucción	Operandos	Operaciones	*opcode*
PUSH	BP	Memory [SP] = BP y SP = SP - 1 y PC = Dir	0o0053
POP	BP	SP = SP + 1	0o0054
		BP = Memory [SP]	0o0055
MOV	BP, SP	BP = SP	0o0056
	Ax, [BP + Lit]	Ax = BP + Lit	0o0057
	Ax, [BP - Lit]	Ax = BP - Lit	0o0060
	Bx, [BP + Lit]	Bx = BP + Lit	0o0061
	Ax, [BP - Lit]	Bx = BP - Lit	0o0062
ADD	SP, Lit	SP = SP + Lit	0o0063
SUB	SP, Lit	SP = SP - Lit	0o0064
RET	Lit	SP = SP + 1	0o0065
		PC = Memory[SP + Lit] y SP = SP + Lit	0o0066

Instrucciones adicionales

Instrucción	Operandos	Operación	*opcode*
NOP	-	-	0o0000

Para más información se recomienda ver una ISA de x86 pero RISC

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, sin excepciones. 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.