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

IIC2343 – Arquitectura de Computadores (II/2018)

Tarea 5

Fecha de entrega: lunes 24 de septiembre de 2018 a las 11:59 AM

Introducción

Una máquina de *stack* es un computador que utiliza un *stack* en vez de registros para almacenar los resultados de las operaciones. Esto significa que cada instrucción aritmética o lógica de dos parámetros, toma los dos valores en el tope del *stack* y luego los elimina, sustituyéndolos por el valor de la operación recién realizada. Para el caso de las operaciones de un parámetro, por ejemplo `NOT`, el computador solo sustituye el valor en el tope del *stack* por el nuevo valor. Además, una máquina de *stack* es capaz de cargar valores literales en el tope del *stack* y también descartarlos.

El diagrama de esta máquina es el siguiente:

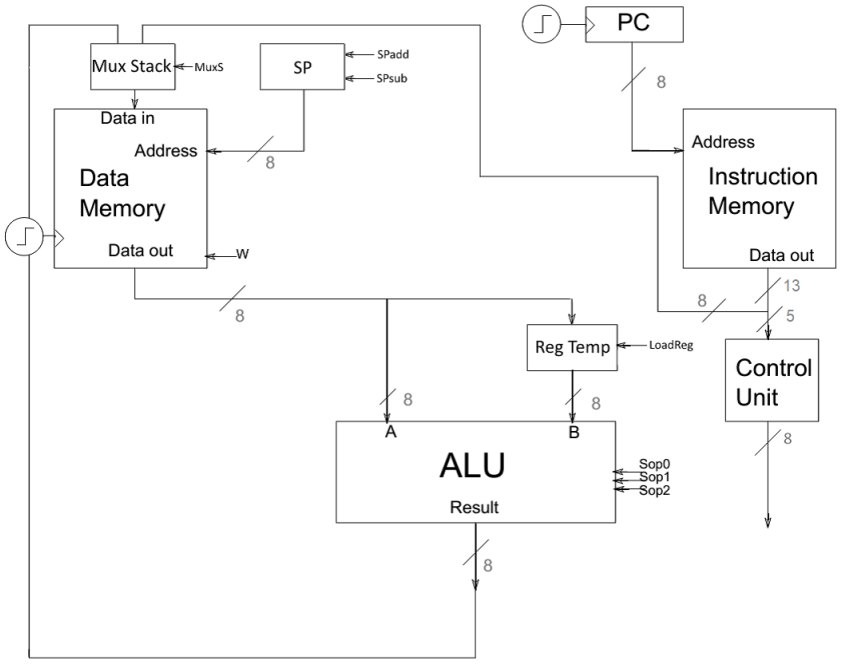

Figura 1: Diagrama de la máquina de *stack*.

De aquí es importante notar que:

- `Reg Temp` es el único registro de datos necesario para este computador. En particular, almacena el valor que se encuentra en el tope del *stack* para luego ser operado con el que se encuentra justo debajo de él (en caso de ser una operación de dos parámetros). Note que como estas operaciones requieren de dos accesos a memoria, requieren de dos ciclos para ejecutarse, justificando el uso de este registro para mantener uno de los valores leídos de la RAM.
- `Mux Stack` se utiliza para escoger lo que ingresa al tope del *stack*: un literal o el resultado de una operación, tal como lo indica la descripción de la máquina.

Por otra parte, la `ISA` asociada a este computador es la siguiente:

Instrucción	***Opcode***	**Señales de control**
`PUSH Lit`	00000	`SPsub = 1`
	00001	`W = 1, MuxS = 1`
`POP`	00010	`SPadd = 1`
`ADD`	00011	`SPadd = 1, LoadReg = 1`
	00100	`Sop = ADD, W = 1, MuxS = 0`
`SUB`	00101	`SPadd = 1, LoadReg = 1`
	00110	`Sop = SUB, W = 1, MuxS = 0`
`AND`	00111	`SPadd = 1, LoadReg = 1`
	01000	`Sop = AND, W = 1, MuxS = 0`
`OR`	01001	`SPadd = 1, LoadReg = 1`
	01010	`Sop = OR, W = 1, MuxS = 0`
`XOR`	01011	`SPadd = 1, LoadReg = 1`
	01100	`Sop = XOR, W = 1, MuxS = 0`
`SHL`	01101	`Sop = SHL, W = 1, MuxS = 0`
`SHR`	01110	`Sop = SHR, W = 1, MuxS = 0`
`NOT`	01111	`Sop = NOT, W = 1, MuxS = 0`
`NOP`	10000	`W = 0`

Tabla 1: `ISA` de la máquina de *stack*.

Tanto la parte de programación como la parte práctica tendrán como objetivo trabajar con la arquitectura aquí presentada.

Parte programación

Su tarea se separará en dos partes programadas, las que serán descritas a continuación.

Parte 1 - *Assembler*

Deberá programar el *Assembler* de la máquina de *stack*.

Formato de entrada

Su programa recibirá como entrada un archivo `.txt` con el código escrito en el `Assembly` de la máquina de *stack*, siguiendo exclusivamente la `ISA` señalada en la introducción. A continuación, un ejemplo de un código válido:

```
;Se agrega 3 y luego 5 al stack. Con ADD se suman los valores del tope: 3+5 = 8.
PUSH 3
PUSH 5
ADD
```

Formato de salida

Su programa deberá retornar como salida un archivo `.txt` donde cada línea será el *opcode* y el literal de cada instrucción. Para el mismo ejemplo anterior, se tendría entonces el siguiente resultado:

```
0000000000000
0000100000011
0000000000000
0000100000101
0001100000000
0010000000000
1000000000000
1000000000000
1000000000000
...
```

Note que los primeros cinco bits corresponden al *opcode*, mientras los 8 restantes corresponden al literal.

Por otra parte, podrá notar que las últimas líneas generadas corresponden a la instrucción `NOP`. Esta en particular no genera ningún cambio en la ejecución de la máquina de *stack* ni en sus registros, pero es necesario que el archivo generado contemple un total de 2^8 líneas de instrucciones dado que, finalmente, este representará la memoria de instrucciones completa de la máquina de *stack*. Por lo tanto, debe llenar el resto del archivo `.txt` con esta instrucción hasta completar el tamaño esperado, en caso de ser necesario.

Ejecución

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

```
C:\User\IIC2343\>python stack_assembler.py program.txt rom.txt
```

En este caso, `program.txt` corresponde a la ruta del archivo de entrada y `rom.txt` a la ruta del nuevo archivo de salida a generar.

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. En el ejemplo, se utiliza arbitrariamente un valor igual a cero.
- 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 de entrada nunca superará la cantidad de direcciones de la memoria de instrucciones.

Parte 2 - Simulador

Deberá programar un simulador para la máquina de *stack*.

Formato de entrada

Su programa recibirá como parámetro por línea de comandos un archivo `.txt` con el contenido de la memoria de instrucciones, la que contiene código `binario` para la máquina de acuerdo a la tabla de *opcodes* entregada. 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 en base binaria y en base decimal del del contenido de la dirección a la que apunta el registro `SP` (*i.e.* el tope del *stack*), siguiendo el formato `t = nº ciclo clock valor_binario valor_decimal`. Por ejemplo, para el caso anterior, se tiene:

```
t = 0 00000000 0
t = 1 00000011 3
t = 2 00000000 0
t = 3 00000101 5
t = 4 00000011 3
t = 5 00001000 8
t = 6 00001000 8
t = 7 00001000 8
t = 8 00001000 8
...
```

Ejecución

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

```
C:\User\IIC2343\>python stack_simulator.py rom.txt result.txt
```

En este caso, `rom.txt` corresponde a la ruta del archivo de entrada y `result.txt` a la ruta del nuevo archivo de salida a generar.

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*.

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* una máquina de *stack*, dando solución a los desafíos que involucra una arquitectura de ese tipo.

Descripción de la actividad

Deben implementar la máquina de *stack* descrita en la sección común del enunciado, implementando la `ISA` entregada en la tabla. Para comenzar a partir de una arquitectura previa, se les facilitará un proyecto base con el computador básico, el que pueden[1] modificar para lograr la nueva máquina.

La `ISA` de su máquina de *stack* debe ser **solo** la indicada en la tabla 1. Es decir, **no puede exponer** instrucciones que trabajen de forma directa con el registro intermedio de la máquina, como por ejemplo `MOV Temp,(Dir)` (siendo `Dir` el valor del registro `SP`). Si bien la combinación de señales lo permiten, su máquina no debe aceptar este tipo de instrucciones, restringiendo al programador que desarrolla el código `Assembly`. **El no respeto de esta restricción significará la evaluación con la nota mínima.**

Para evaluar la correctitud de su arquitectura, los *displays* de la placa deben mostrar, en todo momento, el valor del tope del *stack* de su máquina. Al alimentar el registro `PC` con la señal `clock` del proyecto, podrá ver cómo va cambiando este valor 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 intrucciones `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.

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

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:

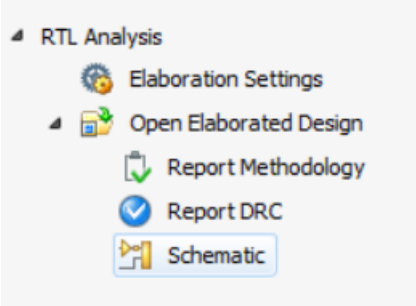

Figura 2: Menú de selección para ingresar al *Schematic*.

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

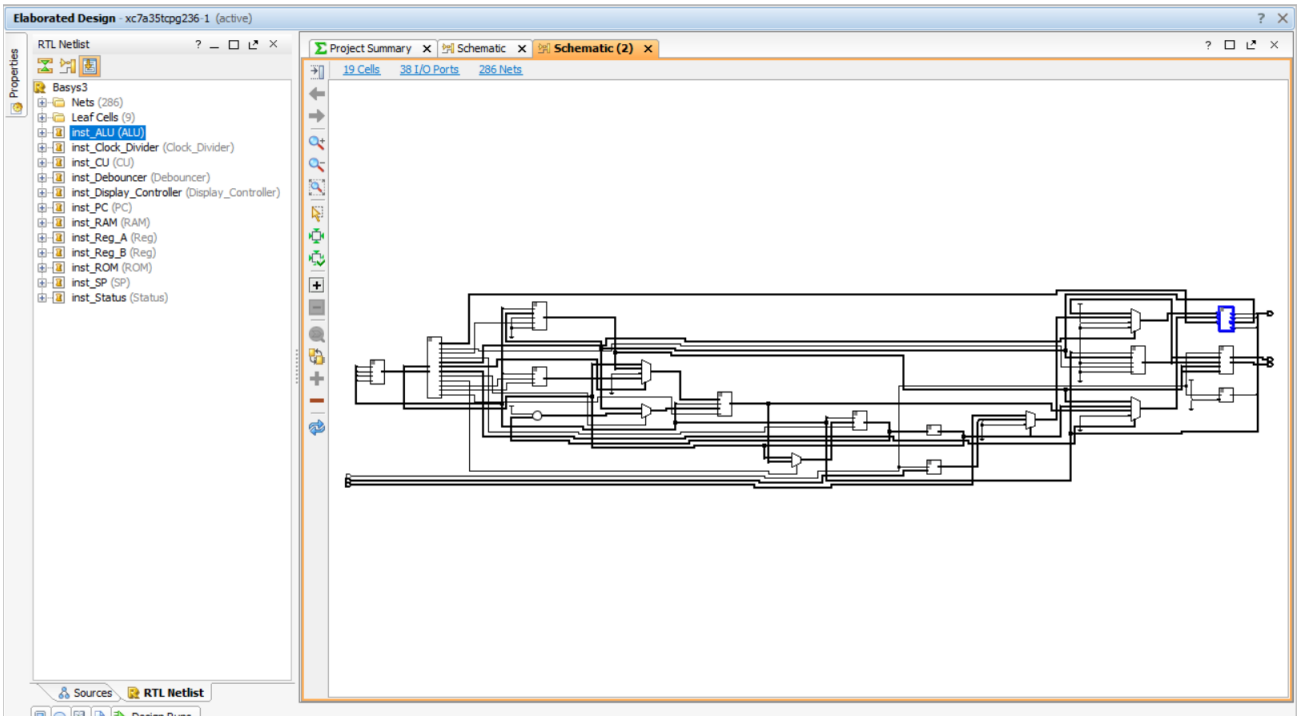

Figura 3: *Schematic* del computador básico.

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 a la máquina de *stack* **todas las instrucciones de salto vistas en clases.** Para ello, deberá añadir:

- Un diagrama de la arquitectura modificada.
- La nueva `ISA` de su arquitectura.
- Detalles de su implementación.

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.**

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 `stack_assembler.py` y `stack_simulator.py`, como se estipuló anteriormente.

Parte práctica

La tarea se debe 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`. El repositorio, además, 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 el sistema operativo usado para realizar la tarea. **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. 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.