DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

ORGANIZACIÓN DE COMPUTADORES

Laboratorio 2: Acercándose al Hardware: Programación en Lenguaje Ensamblador

Profesor: Viktor Tapia.
Alumno: Rodrigo Pereira Yánez

12 de mayo de 2025

DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

Índice

1. Introducción.

El principal desafío de esta experiencia radica en comprender y aplicar de forma práctica el lenguaje ensamblador MIPS, el cual es un paso fundamental para entender la iteración directa con la arquitectura de un procesador.

Para abordar este problema, se utilizó en simulador MARS (MIPS Assambler and Runtime Simulator) como herramienta principal de desarrollo. Con esta herramienta se procedió a la escritura, ensamblaje y depuración de diversos programas en MIPS. Estos programas implicaron la traducción desde C a MIPS de ciclos, estructuras de control, arreglos, funciones y llamadas al sistema (syscall) para la entrada y salida de datos.

Los objetivos de este trabajo son:

- Usar MARS (un simulador para MIPS) para escribir, ensamblar y depurar programas MIPS.
- Escribir programas MIPS incluyendo instrucciones aritméticas, de salto y memoria.
- Comprender el uso de subrutinas en MIPS, incluyendo el manejo del stack.
- Realizar llamas al sistema en MIPS mediante “syscall”.

A lo largo de este documento, se presentará el marco teórico relevante, se explicará el desarrollo de las soluciones implementadas para cada parte del laboratorio, se mostrarán los resultados obtenidos y, finalmente, se expondrán las conclusiones derivadas de esta experiencia práctica.

2. Marco Teórico.

2.1 Lenguaje Ensamblador y Arquitectura MIPS.

El **lenguaje ensamblador** es un lenguaje de programación de bajo nivel que representa las instrucciones básicas que un procesador puede ejecutar. Cada instrucción en ensamblador tiene una correspondencia directa con una instrucción en código máquina. Programar en ensamblador permite un control preciso sobre el hardware, aunque es más complejo y menos portable que los lenguajes de alto nivel.

La **arquitectura MIPS** (Microprocessor without Interlocked Pipeline Stages) es una arquitectura de conjunto de instrucciones reducido (RISC) ampliamente utilizada en entornos académicos debido a su relativa simplicidad y claridad. Se caracteriza por tener un conjunto de instrucciones regular y un gran número de registros de propósito general.

La arquitectura MIPS se basa en 4 principios de diseño fundamentales:

1. La simplicidad favorece la regularidad.
2. Hacer el caso común más rápido.

3. Lo pequeño es rápido.
4. Buen diseño demanda buenos compromisos.

2.2 Registros en MIPS.

La arquitectura MIPS dispone de un conjunto de 32 registros de propósito general, los cuales se clasifican y utilizan convencionalmente de la siguiente manera:

Tabla 2.1: Listado de registros y su descripción.

Registro	**Descripción/Uso Convencional**
`$zero`	Valor constante 0 (no se puede modificar)
`$at`	Registro temporal
`$v0 - $v1`	Retorno de valores de una función/subrutina. `$v0` también se usa para el código de `syscall`.
`$a0 - $a3`	Argumentos de funciones/subrutinas.
`$t0 - $t7`	Registros temporales.
`$s0 - $s7`	Registros guardados.
`$t8 - $t9`	Registros temporales adicionales
`$k0 - $k1`	Reservados para el kernel del sistema operativo.
`$gp`	Puntero global (apunta al área de datos estáticos).
`$sp`	Puntero de pila (Stack Pointer).
`$fp`	Puntero de marco (Frame Pointer).
`$ra`	Dirección de retorno de una función/subrutina.

2.3 Instrucciones MIPS.

El conjunto de instrucciones de MIPS se organiza en diversas categorías, incluyendo:

- **Carga (Load)**: Transfieren datos desde una ubicación en la memoria principal hacia los registros del procesador para su procesamiento.
- **Almacenamiento (Store)**: Mueven datos desde los registros del procesador de vuelta a una ubicación específica en la memoria principal.
- **Aritméticas**: Ejecutan operaciones matemáticas básicas como la suma, resta, multiplicación y división, almacenando los resultados en registros del procesador.

- **Lógicas**: Realizan operaciones lógicas bit a bit, como AND, OR, XOR y NOR, entre operandos y guardan el resultado en un registro.
- **Comparaciones**: Comparan los valores de dos registros y, basándose en el resultado, pueden establecer ciertos indicadores o utilizarse para decisiones en instrucciones de salto condicional.
- **Movimiento entre registros**: Permiten copiar el valor contenido en un registro a otro.
- **Desplazamiento (Shift)**: Realizan desplazamientos de bits a la izquierda o a la derecha dentro de un registro, lo que puede utilizarse para multiplicaciones o divisiones rápidas por potencias de dos.
- **Salto y Bifurcaciones (Jump and Branch)**: Alteran el flujo secuencial de la ejecución del programa. Los saltos incondicionales transfieren el control a una nueva dirección de memoria sin condiciones previas. Los saltos condicionales transfieren el control solo si se cumple una determinada condición (resultado de una comparación entre registros).

2.4 Segmentos de Memorias y Directivas de Ensamblador.

Un programa en ensamblador MIPS típicamente se organiza en segmentos:

- **Segmento de Datos** (`.data`): Se utiliza para declarar variables estáticas y constantes que el programa utilizará. Por ejemplo, se pueden definir cadenas de texto (con la directiva `.asciiz`) o arreglos de números (con la directiva `.word`).
- **Segmento de Texto** (`.text`): Contiene las instrucciones ejecutables del programa. Es aquí donde se escribe la lógica principal del código.

Las directivas como `.data`, `.text`, `.word`, `.asciiz`, `.space` (para reservar espacio en memoria) y `.globl` (para declarar etiquetas globales) son instrucciones para el lenguaje ensamblador que ayudan a organizar el código y los datos.

2.5 Subrutinas (Funciones).

Las subrutinas son bloques de código que realizan una tarea específica y pueden ser llamados desde diferentes partes del programa. En MIPS, la gestión de subrutinas implica:

- **Llamada**: La instrucción `jal <nombre_etiqueta>` (jump and link) salta a la subrutina y guarda la dirección de la siguiente instrucción (dirección de retorno) en el registro `$ra`.
- **Paso de Argumentos**: Generalmente, se utilizan los registros `$a0 - $a3` para pasar argumentos.
- **Retorno de Valores**: Los registros `$v0 - $v1` se usan para devolver resultados.
- **Retorno de la Subrutina**: La instrucción `jr $ra` (jump register) se utiliza para regresar al punto desde donde se llamó la subrutina, utilizando la dirección almacenada en `$ra`.

2.6 Llamas al sistema (syscall).

Las llamadas al sistema son el mecanismo mediante el cual un programa en MIPS interactúa con el sistema operativo o el simulador (MARS) para realizar operaciones de entrada/salida (E/S) o para terminar la ejecución. Para realizar una `syscall`:

- Se carga el código del servicio deseado en el registro `$v0`. Por ejemplo:
  - Código 1: Imprimir un entero
  - Código 4: Imprimir una cadena
  - Código 5: Leer un entero
  - Código 10: Terminar el programa.
- Si el servicio requiere argumentos, estos se cargan en los registros `$a0 - $a3` según corresponda.
- Se ejecuta la instrucción `syscall`. El simulador o el SO se encarga de realizar la operación solicitada.

2.7 Simulador MARS.

El simulador MARS (MIPS Assembler and Runtime Simulator) es un Entorno de Desarrollo Integrado (IDE, por sus siglas en inglés) específicamente diseñado para facilitar la educación y el aprendizaje del lenguaje ensamblador MIPS. Desarrollado por los doctores Pete Sanderson y Kenneth Vollmar, MARS ofrece un entorno donde los estudiantes pueden experimentar directamente con la programación en MIPS sin la necesidad de hardware físico. Permite visualizar el funcionamiento interno de un procesador MIPS simulado, comprender el flujo de ejecución de los programas de manera detallada (paso a paso) y facilita significativamente la identificación y corrección de errores a través de sus herramientas de depuración.

3. Desarrollo de la Solución.

La resolución de los problemas se centró en la traducción directa de la lógica de alto nivel a instrucciones MIPS y el uso de subrutinas y syscall para la modularidad e interacción, con pruebas en MARS.

3.1 Parte 1: Escribir programa MIPS.

3.1.1 Traducción de Bucle while (program1.asm).

- Se inicializaron `$s0 #a=0`, `$s1 #b=5` y `$s2 #dirección base de D[]`.
- El bucle `while (a < 10)` se implementó con `bge $s0, $t0, end_while #$t0 es 10`.
- `D[a] = b + a`: Se calculó la dirección de `D[a] = Base + (4 * Posición)` usando `sll` y `add`. La suma `b+a` se guardó con `sw`.

- `a++`: Se realizó con `addi $s0, $s0, 1`.
- Se añadió una subrutina `imprimirNumero` para mostrar el valor almacenado en cada iteración.

3.1.2 Traducción de Bucle for y suma de pares (program2.asm).

- Se cargó la dirección base de arreglo (arr) en `$s0.` La longitud del arreglo (arrlen) se calculó obteniendo la diferencia de direcciones entre end y arr, y luego `srl $s1, $s1, 2`.
- `evensum` se inicializó en `$t0` y el contador `i` en `$s2`.
- El bucle `for (i < arrlen)` se controló con `bge $s2, $s1, end_for_loop`.
- La condición de paridad (`arr[i] & 1 == 0`) se implementó cargando `arr[i]` en `$t3`, aplicando `andi $t4, $t3, 1` y saltando con `beq $t4, 1, end_if` (saltar si es impar).
- Si es par, `evensum += arr[i]` se hizo con `add $t0, $t0, $t3`.
- Se utilizó una subrutina `imprimirNumero` para mostrar los números pares sumados.

3.2 Parte 2: Uso de syscall.

3.2.1 Implementación de Diferencia par/impar (program3a.asm).

El objetivo fue leer dos enteros, calcular su diferencia absoluta, e indicar si esta era par o impar.

- **Entrada y Salida**: Se utilizaron `syscall` (códigos 4 y 5) para solicitar y leer los dos enteros del usuario, almacenándolos en `$s0` y `$s1`. La salida de mensajes y resultados también se manejó con syscall (códigos 4 y 1).
- Subrutina `diferencia_absoluta`:
  - Recibió los dos enteros (copiados a `$a1, $a2` desde `$s0, $s1`).
  - Calculó la diferencia absoluta (`$v0 = |$a1 - $a2|`) mediante una comparación `blt $a1, $a2, end_if` y resta condicional `sub $v0, $a1, $a2` o `sub $v0, $a2, $a1`
- Subrutina `par_impar`:
  - Recibió la diferencia (copiada a `$a3` desde el resultado de `diferencia_absoluta`).
  - Determinó la paridad usando `andi $t1, $a3, 1`.
  - Imprime "`(Par)`" o "`(Impar)`" según el resultado de `andi` usando `syscall`.
- Flujo Principal: `main` orquestó la lectura, llamó a `diferencia_absoluta`, imprimió la diferencia, y luego llamó a `par_impar` para mostrar la paridad.

3.2.2 Implementación de nueva subrutina condicional(program3b.asm).

Este programa extendió el anterior, sumando la diferencia a uno de los enteros originales según su paridad, e introdujo una mayor modularización.

- Modularización de E/S: Se crearon subrutinas específicas para la interacción con el usuario:
- Subrutina `diferencia_absoluta`: Se mantuvo sin cambios funcionales respecto a program3a.asm.
- Subrutina `par_impar` (Modificada):
  - Además de imprimir "`(Par)`" o "`(Impar)`", retornó un flag en `$v0` (0 si par, 1 si impar).
- Nueva Subrutina `primer_segundo_entero`:
  - Recibió la diferencia (`$a0`), los enteros originales (`$a1, $a2`), y el flag de paridad (`$a3` proveniente de `par_impar`).
  - Si el flag indicaba par (`$a3 == 0`), sumó la diferencia al primer entero (`$a1 + $a0`) e imprimió el resultado.
  - Si el flag indicaba impar (`$a3 == 1`), sumó la diferencia al segundo entero (`$a2 + $a0`) e imprimió el resultado.
- Flujo Principal Mejorado: `main` utilizó las nuevas subrutinas de E/S. Después de obtener la diferencia y el flag de paridad (de `diferencia_absoluta` y `par_impar` respectivamente), llamó a `primer_segundo_entero` para realizar la suma condicional y mostrar el resultado final.

4. Resultados.

Todos los programas desarrollados fueron ensamblados y ejecutados satisfactoriamente en el simulador MARS. Las pruebas realizadas, incluyendo los casos de ejemplo proporcionados en el enunciado y variaciones con distintos valores de entrada (incluyendo números negativos para la Parte 2), demostraron el correcto funcionamiento de cada solución.

4.1 Parte 1: Escribir programa MIPS.

4.1.1 Traducción de Bucle while (program1.asm).

El programa implementó exitosamente (Figura 4.1.1) el bucle while especificado, mostró en la consola los valores almacenados en el arreglo D a medida que se calculaban, separados por comas. La salida final fue:

```
Unset
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ] Loop finalizado
```

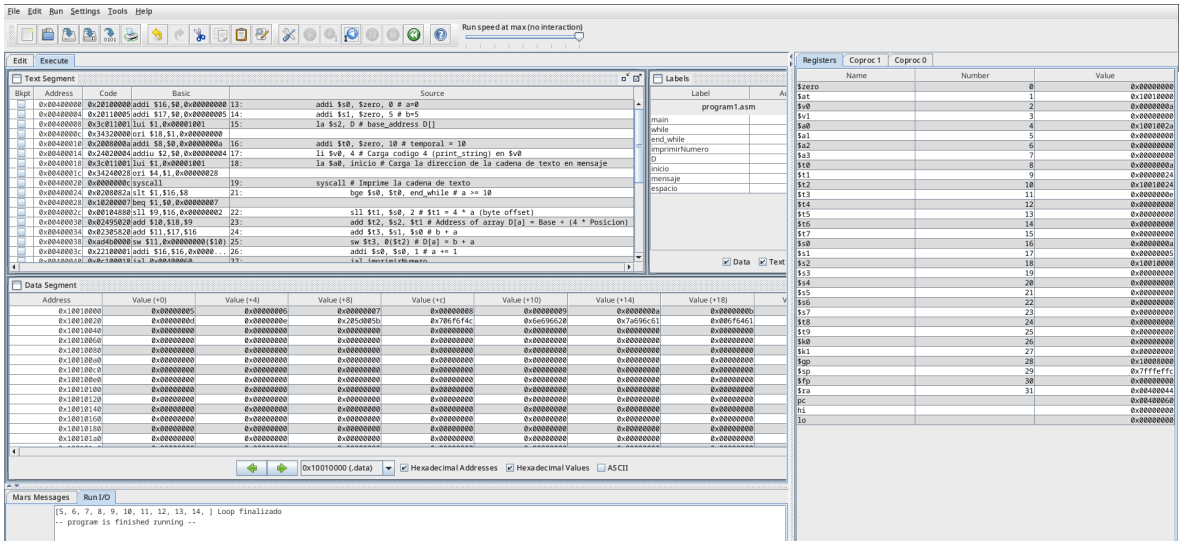

Figura 4.1.1: Execute - MARS, program1.asm.

4.1.2 Traducción de Bucle for y suma de pares (program2.asm).

El programa determinó correctamente la longitud del arreglo `arr`. Posteriormente, recorrió el arreglo utilizando una estructura de bucle for. Dentro del bucle, identificó los elementos pares y los sumó acumulativamente en el registro `$t0` (representando `evensum`). La subrutina `imprimirNumero` mostró los números pares que se iban sumando. Para el arreglo {11, 22, 33, 44}, la salida observada fue:

```
Unset
22 + 44 +  = 66
```

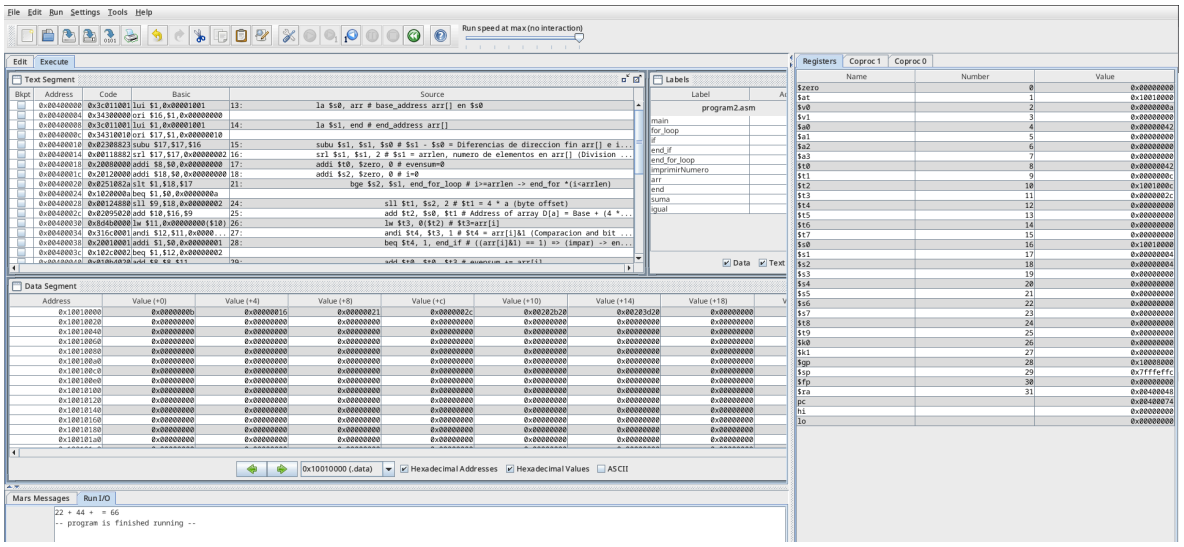

Figura 4.1.2: Execute - MARS, program2.asm.

4.2 Parte 2: Uso de syscal.

4.2.1 Implementación de Diferencia par/impar (program3a.asm).

El programa interactuó correctamente con el usuario, solicitando y leyendo dos enteros. La subrutina `diferencia_absoluta` calculó la diferencia no negativa entre ambos. Posteriormente, la subrutina `par_impar` determinó si esta diferencia era par o impar, mostrando el resultado en consola. La salida final fue:

```
Unset
Por favor ingrese el primer entero: 31
Por favor ingrese el segundo entero: 11
La diferencia es: 20 (Par)
```

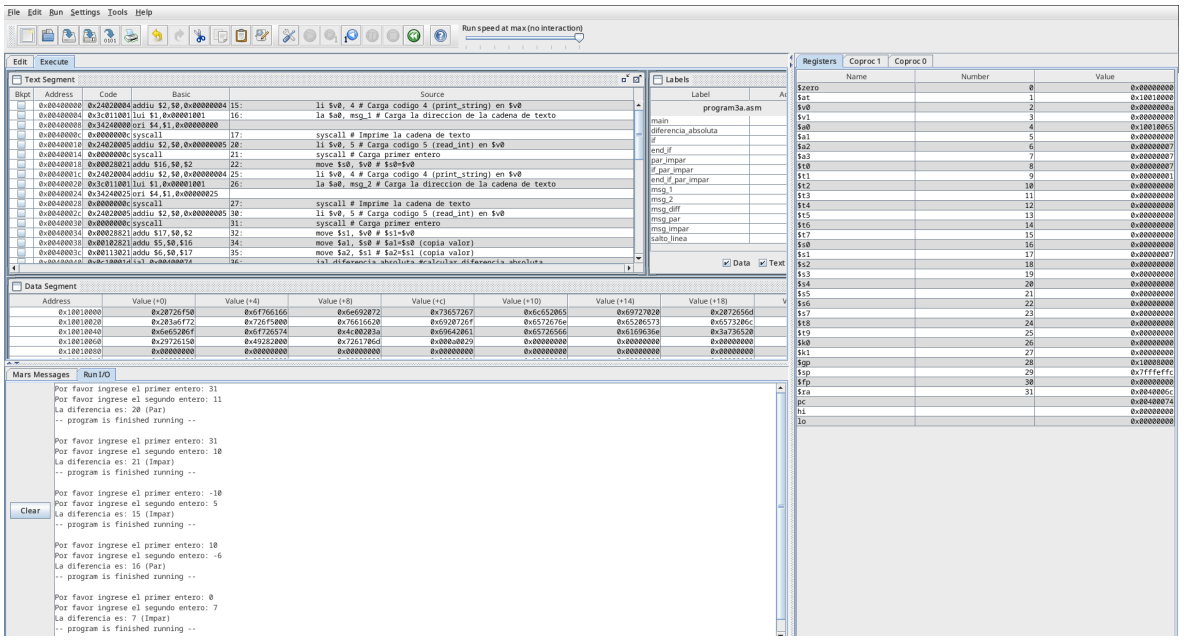

Figura 4.2.1: Execute - MARS, program3a.asm.

4.2.2 Implementación de Diferencia par/impar (program3a.asm).

Este programa, además de realizar las funciones de program3a.asm, implementó una lógica adicional en la subrutina `primer_segundo_entero`. Si la diferencia era par, se sumaba al primer entero ingresado; si era impar, se sumaba al segundo. La salida final fue:

```
Unset
Por favor ingrese el primer entero: 31
Por favor ingrese el segundo entero: 11
La diferencia es: 20 (Par)
Nuevo primer entero: 51
```

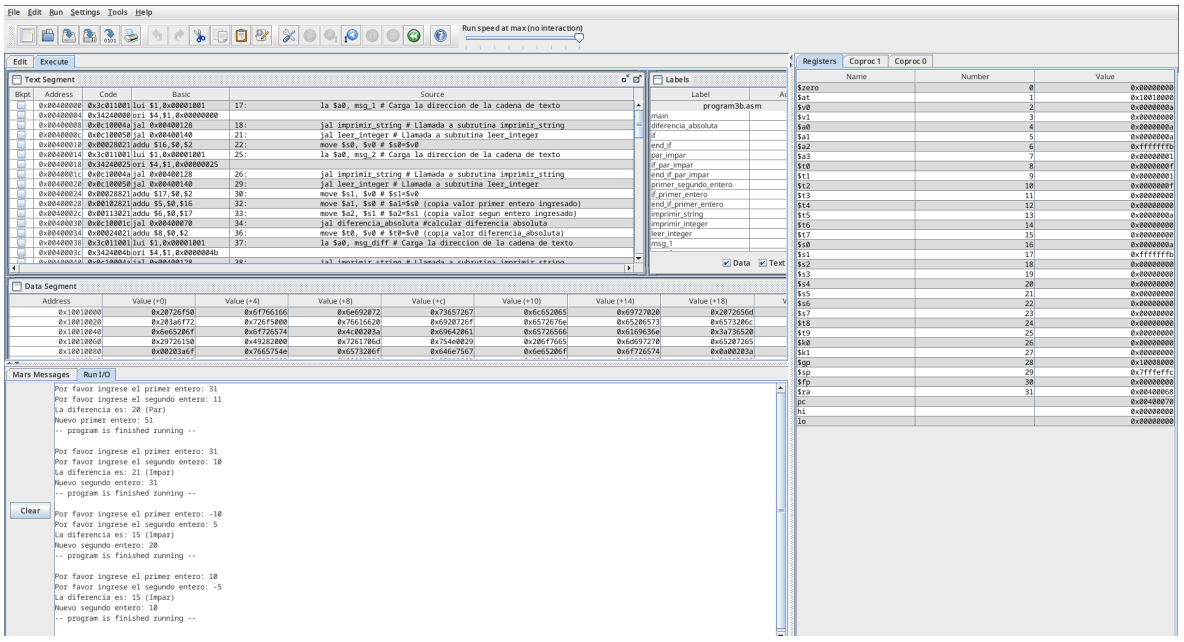

Figura 4.2.2: Execute - MARS, program3b.asm.

Conclusiones.

Este laboratorio fue fundamental para profundizar los conocimientos en programación en lenguaje ensamblador con la arquitectura MIPS mediante el simulador MARS.

- Se tradujeron con éxito estructuras de alto nivel (bucles, arreglos) a código MIPS, mejorando el manejo de registros y memoria.
- Se aplicó un variado conjunto de instrucciones MIPS (aritméticas, de salto, de memoria) para construir la lógica de los programas.
- Se implementaron subrutinas, gestionando el paso de argumentos y el retorno de valores.
- Se utilizó la instrucción syscall para la interacción con el usuario (entrada/salida de datos) y el control del programa.

En definitiva, la experiencia práctica mejoro la comprensión de la programación a bajo nivel y su relación directa con la arquitectura de computadores, cumpliendo todos los objetivos del laboratorio.

Referencias.

1. MARS MIPS Simulator. (s.f.). [https://dpetersanderson.github.io/index.html].
2. Patterson, D. A., & Hennessy, J. L. (2014). Computer organization and design: The hardware/software interface. Morgan Kaufmann.