FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 1
Conjunto de Instrucciones MIPS

[8637/6620] Organización de Computadoras
Curso 2
Segundo Cuatrimestre de 2020

Alumno:	Padrón
BIANCARDI, Julián	103945
CZOP, Santiago Nicolás	104057
OUTEIRO, Sebastián	92108

Índice

1. Introducción

El presente informe reúne la documentación de la solución del primer trabajo práctico de la materia Organización de Computadoras que consiste en desarrollar un programa capaz de calcular el mínimo común múltiplo (m.c.m.) y máximo común divisor (M.C.D) entre dos números utilizando los lenguajes de programación C y Assembly de MIPS aplicando todos los conocimientos adquiridos hasta el momento en la materia.

2. Supuestos

Para la implementación del programa solicitado se realizaron los siguientes supuestos:

1. Se definió como "MAXINT", el número máximo aceptado como parámetro, a la constante *UINT_MAX* del módulo *limits.h*, que representa el máximo valor de un entero sin signo.
2. En caso de que el resultado del m.c.m. exceda la máxima representación posible de un entero sin signo, el comando ingresado se tomará como válido pero el comportamiento será indefinido.
3. Dado que la opción -m indica que solamente se debe calcular el m.c.m. (exclusivamente) y la opción -d indica que solo se debe calcular el M.C.D. (exclusivamente), pedir que solamente se calculen ambos es ilógico. Por lo tanto, un comando que incluye tanto la opción -m como -d es inválido y se devolverá un error por salida estándar de error.

3. Desarrollo

Para el desarrollo del trabajo se dividió en bloques el programa a realizar. Se detectaron 3 partes principales: parseo y procesamiento del comando, cálculo del mínimo común múltiplo y cálculo del máximo común divisor.

3.1. Línea de comando

Para parsear la línea de comando se optó por usar la librería "getopt.h" (ver manual: `https://man7.org/linux/man-pages/man3/getopt.3.html`). En específico, se utilizó la función *getopt_long* que nos simplifica el procesamiento de los argumentos, pudiendo así definir una lógica distinta a partir de cada parámetro. El comportamiento se definió en su totalidad en el archivo *args_parser.c*.

A diferencia del trabajo práctico 0, esta vez fue necesario lógica extra para validar los argumentos. Se decidió que esta tarea le correspondía al módulo *args_parser*, aumentando así sus responsabilidades (no sólo debe devolver los parámetros, sino además avisarnos si son válidos). Para no cargar demasiado a la función *get_arguments()* se dividió su lógica en leer los argumentos, validar el modo y obtener los números a utilizar.

3.2. Máximo Común Divisor

Para calcular el máximo común divisor entre los números m y n se utilizó el algoritmo de Euclides. El mismo consiste en realizar los siguientes pasos

1. Si $n = 0$ entonces $mcd(m, n) = m$ y el algoritmo termina.
2. En otro caso, $mcd(a, b) = mcd(b, r)$ donde r es el resto de dividir a a entre b.

Se optó por implementar en Assembly únicamente el algoritmo iterativo, aunque también exista uno recursivo.

3.3. Mínimo Común Múltiplo

Para calcular el mínimo común múltiplo entre los números m y n se reutiliza el algoritmo M.C.D. antes mencionado. Para esto se procede con la siguiente fórmula:

$$mcm(m, n) = (m \cdot n)/mcd(m, n)$$

Con la implementación indicada, mcm(m, n) no es una función hoja, por lo que en Assembly requirió consideraciones adicionales al M.C.D.

3.4. Salida

La salida de las operaciones antes mencionadas serán escritas en un archivo especificado por la opción "-o". En caso de que el contenido de la opción sea "-", la salida se interpretará como "stdout". A diferencia del trabajo práctico 0, esta vez debido a la sencillez de la operación de escritura en la salida, se optó por no implementar un TDA de escritura a archivos. Para poder diferenciar las salidas, por ejemplo para un comando como *./common -o - 4 5*, el cual imprimiría tanto el M.C.D. como el m.c.m., se agregó una pequeña etiqueta que precede a la impresión.

4. Diagramas

4.1. Diagrama de Secuencia

Se compuso un diagrama de secuencia. Éste intenta mostrar el funcionamiento del programa a partir de una hipotética ejecución con el comando *common -o - 28 7*.

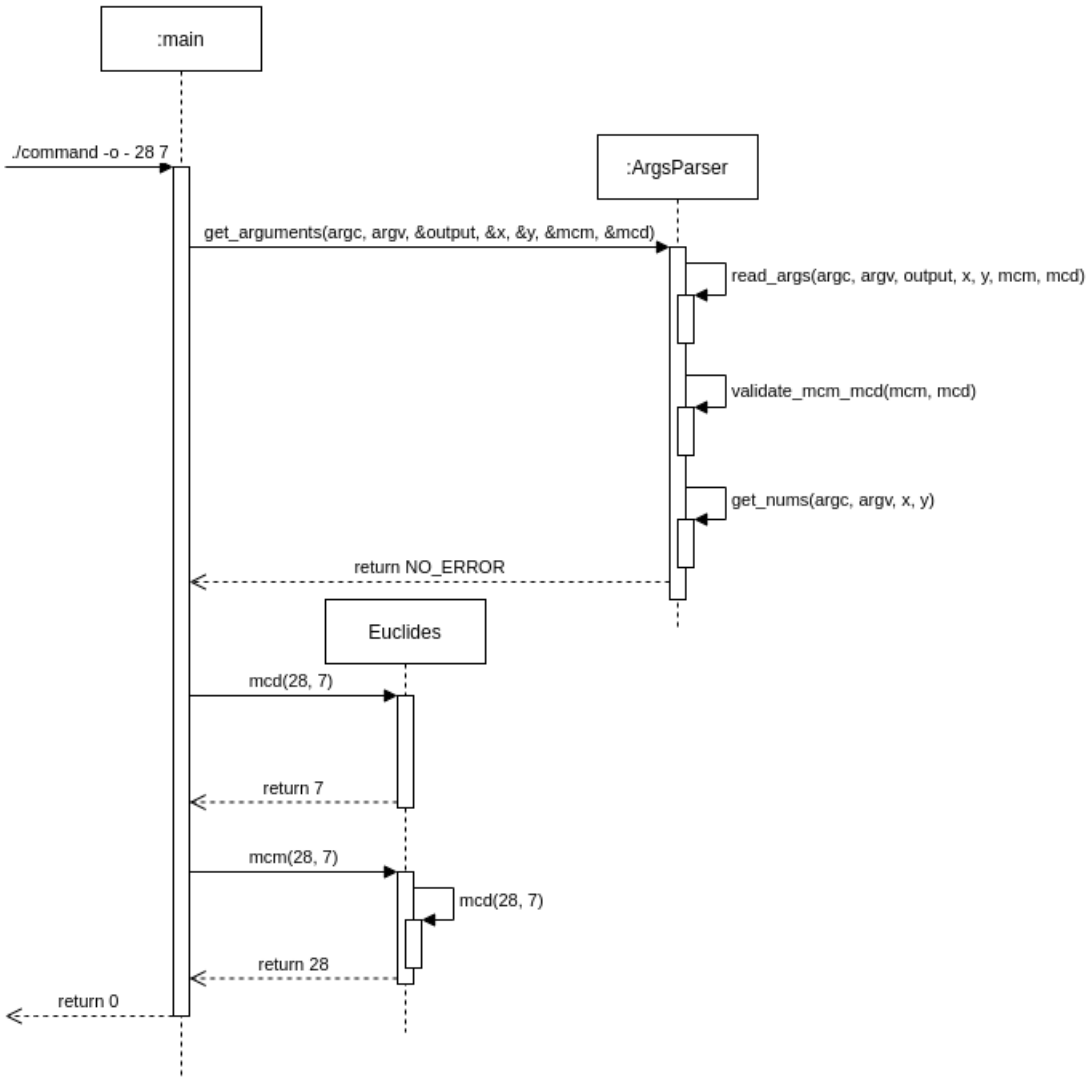

Figura 1: Diagrama de Secuencia para el Comando "./common -o - 28 7".

4.2. Diagrama de Stack

Se compuso un diagrama de stack, que intenta representar gráficamente cómo está compuesto el stack para las funciones mcm() y mcd() cuando se las llama con los parámetros 1111 y 1294.

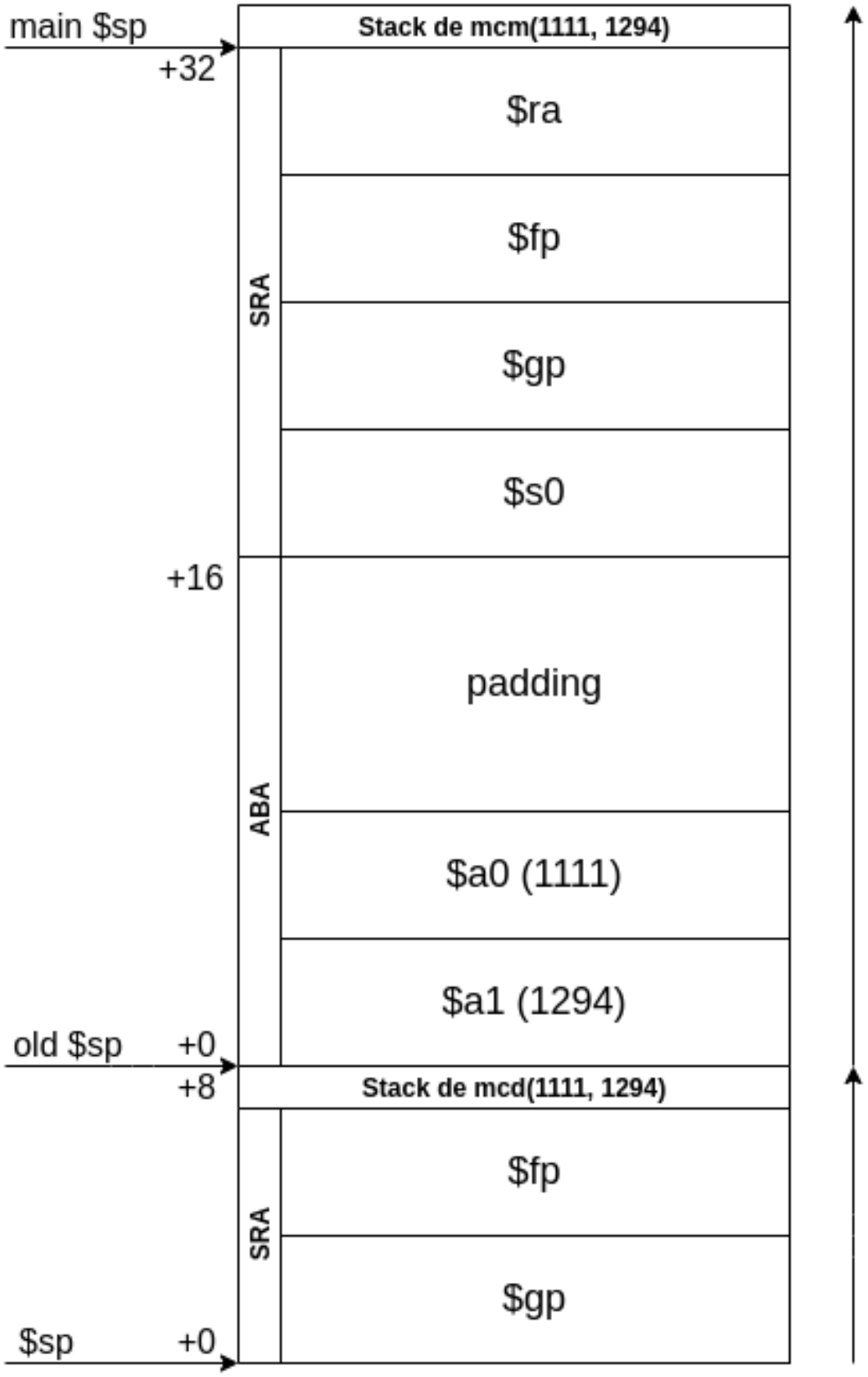

Figura 2: Diagrama de Stack para las funciones mcm y mcd.

5. Ejecución de Pruebas

En el siguiente apartado se describirán las pruebas realizadas para probar el código del trabajo y garantizar así el correcto funcionamiento en todos los escenarios posibles. Las pruebas fueron escritas en C y hacen uso de módulos adicionales que permiten una salida prolija de los resultados.

5.1. Pruebas Unitarias

Pruebas de MCD

```
MCD TESTS

MCD entre 5 y 10 devuelve 5: OK
MCD entre 256 y 192 devuelve : OK
MCD entre 1111 y 1294 devuelve : OK
MCD entre 12 y 20 devuelve : OK
MCD entre 205 y 87 devuelve : OK
```

Pruebas de MCM

```
MCM TESTS

MCM entre 5 y 10 devuelve 10: OK
MCM entre 256 y 192 devuelve : OK
MCM entre 1111 y 1294 devuelve : OK
MCM entre 12 y 20 devuelve : OK
MCM entre 205 y 87 devuelve : OK
```

Pruebas de Args Parser

```
ArgsParser

Commands with a single argument returns an error: OK
Commands with two arguments returns an error: OK
Commands with three arguments returns an error: OK
Commands with four arguments returns an error: OK
Commands with five arguments are valid: OK
Commands with huge valid numbers are valid: OK
Commands with huge invalid numbers return error: OK
Commands with negative numbers return error: OK
Commands with number zero return error: OK
Commands with number one return error: OK
Commands with number two are valid: OK
Commands with specified output file are valid: OK
```

5.2. Pruebas de Benchmark

En adición a las pruebas unitarias, se implementaron pruebas sencillas que calculan cuánto tiempo tarda la computadora en ejecutar muchas veces *mcm(m, n)* y *mcd(m, n)*.

5.2.1. Código Fuente de la Prueba

Las pruebas consisten en inicializar un vector de 1024 números aleatorios entre 0 e INT_MAX y luego cronometrar $2^{30} - 1$ ejecuciones de *mcm(m, n)* o *mcd(m, n)* según corresponda. En el tiempo se incluyen algunas operaciones extra que no se pueden desacoplar con facilidad que en conjunto se encargan de seleccionar los números aleatorios a utilizar como parámetro.

```
unsigned int numbers[1024];
for (int i = 0; i < 1024; i++) {
  numbers[i] = rand();
}

clock_t start = clock();
for (int i = 0; i < SAMPLE_SIZE; i++) {
  unsigned int index = rand();
  index %= 1023;
  mcd(numbers[index], numbers[index]); // mcm en la otra prueba
}
clock_t end = clock();

return (double)(end - start) / CLOCKS_PER_SEC;
}
```

5.2.2. Resultados

Procesador	Sistema Operativo	Tiempo MCM	Tiempo MCD
Intel i7	Fedora 32	13.303340 s	11.519333 s
	QEMU	N/D	N/D
AMD Ryzen 7	Ubuntu	9.840360 s	9.723055 s
	QEMU	N/D	N/D

Cuadro 1: Benchmark Results

Por motivos que exceden nuestro conocimiento, en la máquina virtual de QEMU la ejecución tardó demasiado y se decidió frenarla, por lo que no se obtuvieron resultados. Sin embargo, se obtuvieron resultados interesantes para los Host.

6. Compilación y Portabilidad

El programa se diseñó para ser portable a QEMU, de forma tal que en caso de correrse con una máquina mips64 pueda aprovecharse los módulos de mínimo común múltiplo y máximo común divisor desarrollados en Assembly. Para compilar el programa se proporcionó un *Makefile* capaz de compilar para la máquina correcta con el comando *make*. Por su lado, las pruebas pueden compilarse con *make tests* y cualquier archivo creado por estos comandos se puede borrar con *make clean*.

7. Código Fuente

7.1. Código .TEX

Se puede obtener el código .TEXutilizado para compilar el presente informe en:
`https://www.overleaf.com/read/tfqscnsxwmkq`

7.2. Código .C y .S

Se puede observar el código implementado en la siguiente dirección, en la carpeta TP1:
`https://github.com/JulianBiancardi/Organizasion-de-Computadoras-66.20`

8. Conclusiones

En conclusión, se considera que se ha logrado implementar una solución portable que cumple con las indicaciones requeridas. Se encontró una particular dificultad para lograr comprender e implementar ABI, resultando muy confuso comprender las direcciones de memoria de cada bloque y qué le corresponde al Caller y Callee. Programar en Assembly no resultó muy desafioso debido a que nos encontrabamos familiarizados con SPARC, que funcionaba similar pero con otros parámetros.

También hemos de mencionar que aunque el archivo de assembly se logró generar sin mayores problemas, nos encontramos con instrucciones que resultan extrañas y esperamos ser capaces de comprenderlas eventualmente. El trabajo práctico se coloca a si mismo como un buen punto de partida, dado que es relativamente sencillo y se trata de código que conocemos profundamente debido a que fue implementado por nosotros.

Lamentablemente, no se logró llegar a resultados conclusivos sobre si Assembly proporciona una mejora sobre el programa realmente o no, dado que no fue posible comparar contra QEMU. Se atribuye que el tiempo excesivo en su ejecución de las pruebas de Benchmark se debieron a que es una máquina virtual y por lo tanto trabaja mucho más lento.

Referencias

[1] Linux.die.net. 2020. *Getopt_Long(3): Parse Options - Linux Man Page.* [online] Disponible en: <https://linux.die.net/man/3/getopt_long>(Accedido 21 Octubre 2020).

[2] En.wikipedia.org. 2020. *Euclidean Algorithm.* [online] Disponible en: <https://en.wikipedia.org/wiki/Euclidean_algorithm>(Accedido 5 Noviembre 2020).

[3] Gnu.org. n.d. *GNU Make.* [online] Disponible en:
<https://www.gnu.org/software/make/manual/make.html>(Accedido 12 Noviembre 2020).