PRÁCTICA 5

ARM

El objetivo de esta práctica, es escribir un programa en lenguaje ensamblador para encontrar los números primos entre 2 y 32768, empleando el método de la **“Criba de Eratóstenes”.**

La Criba de Eratóstenes es un procedimiento para determinar todos los números primos hasta cierto número natural dado. Esto se hace recorriendo una tabla de números usando el siguiente algoritmo:

- Empezamos en el número 2, resaltamos el número 2 como primo pero *tachamos* todos los múltiplos de 2 (es decir, tachamos 4, 6, 8, etc.).
- Se continua con el siguiente número no tachado en la tabla, en este caso el número 3, resaltamos el número 3 como primo y *tachamos* todos los múltiplos de 3 (es decir tachamos 6, 9, 12, etc.).
- El siguiente número no tachado en la tabla es el 5, resaltamos el número 5 como primo y *tachamos* todos los múltiplos de 5 (es decir tachamos 10, 15, 20, etc.).

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120
121	122	123	124	125	126	127	128	129	130
131	132	133	134	135	136	137	138	139	140
141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160
161	162	163	164	165	166	167	168	169	170
171	172	173	174	175	176	177	178	179	180
181	182	183	184	185	186	187	188	189	190
191	192	193	194	195	196	197	198	199	200

Por lo tanto, el método de la criba de Eratóstenes requiere una matriz con un bit por cada número entero. Inicialmente todos los bits están en 1, indicando que cada número es potencialmente un número primo.
Comenzando por el número 2, se consulta la matriz de bits. Si el bit correspondiente es 1 el número es primo. A continuación se marcan como no primos todos los números múltiplos del primo actual. Para ello hay que escribir un 0 en la posición que corresponde a cada múltiplo en la matriz de bits. Cuando se llega al último número sólo quedan en 1 los bits correspondientes a los números primos en la matriz de bits.

Para nuestra práctica, organizaremos la matriz de bits como una matriz de 1024 posiciones de 32 bits cada una, comenzando en la dirección de memoria 0x40000800. Por lo tanto, lo primero que el programa deberá de hacer, es escribir dicha matriz en la memoria de datos.
A continuación, escribiremos dos subrutinas cuyo nombre y función son los siguientes:

1. *tstbit:* Prueba el bit indicado en R0, y devuelve 1 en el flag Z si el bit de la matriz es 0. Hay que tener en cuenta que los 5 bits de menor peso de R0 seleccionan uno de los 32 bits del dato de la matriz indicado por los bits 6 al 15 de R0.
2. *clrbit*: Pone en cero el bit indicado por R0. El significado de los bits de R0 es idéntico al caso anterior.

La estructura del programa puede ser esta:

- Escribimos la matriz rellena de 1 inicialmente en la memoria de datos.
- Comenzamos probando el número 2, utilizando la función *tstbit*. Si nos dice que en esa posición hay un 1, dicho número es primo.
    - Si el número es primo, se imprime y también un salto de línea (utilizando las funciones *_U0prtnum* y *_U0putch*, como se explicará más adelante).
    - A continuación calculamos todos los múltiplos sin salirnos de la matriz, y borramos esas posiciones utilizando la función *clrbit*.
    - Cuando acabemos con todos los múltiplos (nos salimos de la matriz), pasamos al siguiente número, y volvemos a comprobarlo utilizando la función *tstbit*.
- Si cuando probamos un número utilizando la función *tstbit* nos dice que ese número no es primo, simplemente pasamos al siguiente.

PENSAR EL PROBLEMA AUNQUE EN CLASE VEREMOS ALGO MÁS SOBRE TODO ESTO.

Rutinas de entrada-salida en crt.S:

_Halt: No tiene parámetros. Se detiene la ejecución hasta un nuevo reset.
_U0putch: parámetro en R0: código ASCII del carácter a transmitir. Envía el caracter almacenado en R0 a través del puerto serie al terminal.

_U0getch: sin parámetros. Espera recibir un dato en el puerto serie. Cuando esto ocurre devuelve en R0 el código ASCII del carácter recibido. *_U0getch* no retorna hasta que no se recibe un dato.
_U0puts: parámetro: puntero a texto terminado en cero. Transmite por el puerto serie la cadena de caracteres apuntada por R0. La cadena debe estar terminada con un caracter nulo (ASCII 0).
_U0prthex: parámetro de 32 bits en R0. Imprime en hexadecimal, a través del puerto serie, el valor de R0 como un dato de 32 bits (8 dígitos).
_U0prthex8: parámetro de 8 bits en R0. Imprime en hexadecimal, a través del puerto serie, el valor de R0 como un dato de 8 bits (2 dígitos).
_U0prtnum: parámetro de 32 bits en R0. Imprime en decimal, a través del puerto serie, el valor de R0. Se imprimen los dígitos justos para representar el valor (se eliminan ceros a la izquierda).
_delay_loop: parámetro en R0. Rutina para hacer retardos. El tiempo que se tarda en retornar es (4*R0+4), medido en ciclos de reloj (16.95 ns por cada ciclo de reloj). Esta rutina se utiliza en las macros *_delay_us* y *_delay_ms*, que tenemos en los programas en "C" y que usan unas unidades de tiempo más convenientes.

ARM: convención de paso de parámetros a subrutinas

ARM ha publicado una convención para el paso de parámetros a subrutinas. Siguiendo esta convención podemos hacer que el código que escribamos en lenguaje ensamblador sea compatible con el compilado desde lenguajes de alto nivel, como "C". Para ello debemos seguir las siguientes reglas:

- Los parámetros tienen un tamaño máximo de 32 bits e incluyen tanto datos de tipo entero como punterosde memoria. No trataremos los parámetros de coma flotante.
- Los parámetros se pasan a las subrutinas en los registros R0, R1, R2 y R3, en este mismo orden. Si una subrutina necesita un único parámetro éste deberá estar en el registro R0 antes de llamar a la subrutina. Si tuviese dos parámetros, p0 y p1, p0 irá en R0 y p1 en R1, y así suvesivamente hasta un máximo de 4 parámetros.
- Si hay más de 4 parámetros hay que introducir todos los que pasen del parámetro 3 en la pila antes de llamar a la subrutina y vaciar la pila después del retorno de la subrutina.
- La pila siempre es del tipo full-descending.
- Los resultados se devuelven en los registros R0 a R3. Si hubiera que retornar un número de datos mayor habría que plantearse pasar a la subrutina un puntero (por ejemplo en R0) y almacenar los resultados en la memoria apuntada por dicho puntero.
- Las subrutinas no necesitan preservar los valores de los registros R0, R1, R2, R3, R12 ni R14. Ojo: R14 es LR y contiene la dirección de retorno cuando se llama a la subrutina. El programa que llama a las subrutinas debe tener en cuenta que el contenido de cualquiera de estos registros se puede perder en las llamadas.
- Las subrutinas **deben preservar** los valores de R4, R5, R6, R7, R8, R9, R10 y R11. Si una subrutina necesitara usar alguno de estos registros debería guardar su valor en la pila previamente y recuperarlo antes de retornar.

Ejemplo de subrutina con 1 parámetro en R0 y un valor de retorno, también en R0. No se llama a otras subrutinas:

```
suma8192:
                add r0,#0x2000 @ operación...
                mov pc,lr @ retorno
```

Ejemplo de subrutina con más registros en uso y llamada a otras subrutinas. Se usa la pila para almacenar y recuperar el valor original de los registros:

```
rutina2:
stmfd sp!,{r4-r7,lr} @ guardamos registros, incluyendo LR
...
mov r0,r1,lr, lsl #2 @ ahora LR es un reg cualquiera
...
bl rutina_l2 @ LR se usa en llamadas a subrutinas
...
ldmfd sp!,{r4-r7,pc} @ recuperamos regs. PC en lugar de LR
```

Ejemplo de llamada a subrutina con más de 4 parámetros. Los parámetros están en R0 a R6 (7 parámetros):

```
stmfd sp!,{r4-r6} @ 3 parámetros de más a la pila
bl rutina7p
add sp,sp,#12 @ reajustamos pila
...
```