Proyecto de Arquitectura de Computadores

Angeles Barazorda, Jean Pier (100%); Mori Ortiz, Pedro Enrique (100%); Neira Riveros, Jorge Luis (100%)

1. Implementación de MUL, UMULL y SMULL en la ALU

Para añadir estas operaciones a nuestro ALU, debemos modificar ligeramente nuestro código. Para empezar, debemos aumentar la cantidad de bits de $ALUControl$ para que reconozca las 3 nuevas operaciones.

- $ALUControl = 000$: Suma
- $ALUControl = 001$: Resta
- $ALUControl = 010$: And
- $ALUControl = 011$: Or
- $ALUControl = 100$: Multiplicación
- $ALUControl = 101$: Unsigned multiplicación
- $ALUControl = 110$: Signed multiplicación

Además, debemos añadir una salida más, por lo que, en lugar de tener una salida $Result$, tendremos 2 salidas $Result1$ y $Resul2$.

Para la operación multiplicación, multiplicamos las entradas a y b, y lo igualamos a $Result1$.

Para la multiplicación unsigned, simplemente multiplicamos las entradas a y b y el resultado lo enviamos a $\{Result1, Result2\}$.

Para la multiplicación signed, creamos dos wire signed de 32 bits, igualamos a las entradas y tendremos las representaciones signed de estas, El resultado de multiplicar ambos wire signed lo enviamos a $\{Result1, Result2\}$.

A. Implementación y Testeo

En esta parte, se testeó nuestro ALU como unidad separada de los procesadores. Este ahora tiene la capacidad de ejecutar instrucciones de tipo MUL, UMULL y SMULL. Para ello, se utilizó un *testvector* en el cual se indicaban los *inputs* como el ALUControl y los *outputs* esperados, tal y como se puede observar en nuestro código fuente adjuntado. A continuación, se muestran las operaciones testeadas:

- 1 * 0 = 0
- 2147483648 * 2 = 4294967296 (Unsigned multiplicación, se guarda en dos registros)
- 1073741824 * 2 = 2147483648 (Signed multiplicación, se guarda en dos registros donde uno de ellos debe contener ceros al tratarse de una operación con signo y el resultado ser positivo)

Al probar nuestro *testvector*, todo fue procesado exitosamente ya que no existieron errores.

Fig 1.1: Simulación del ALU con instrucciones MUL, UMULL y SMULL

2. Implementación de la unidad de punto flotante

Las operaciones de números en punto flotante se realizan en FPU (Floating Point Unit), el cual es un módulo separado del ALU. A continuación presentamos los detalles de su implementación.

- Dos entradas a y b de 32 bits cada uno. Deben estar en el formato de floating point (32 bits) o de half floating point (16 bits).
- Una entrada $FPUControl$ de 2 bits. El bit $FPUControl[0]$ indica si los números están en FP de 16 bits (0) o de 32 bits (1). El bit $FPUControl[1]$ indica si la operación es una suma (0) o un producto (1).
- Una salida $Result$ de 32 bits cuyo formato depende de si las entradas están en floating point (32 bits) o de half floating point (16 bits).
- Una salida $FPUFlags$ de 4 bits. Si bien solo se usarán 2 bits (Neg y Zero) se mantiene el formato de 4 bits para implementarlo de forma similar a los ALUFlags.
- Tiene señales internas que permiten el desarrollo de los algoritmos.
- Evaluará los casos base, tales como una suma o producto con 0; y evaluará números negativos.
- Los casos de overflow o underflow son resueltos realizando desplazamientos (shift) en los que es posible perder información de los bits menos significativos, es decir, de los decimales que estén al final.

A. Algoritmo para sumar números en Floating Point

El algoritmo para sumar números en Floating Point fue desarrollado durante el Laboratorio 03 del curso, sin embargo, presentaremos modificaciones para tratar los casos base y los números negativos.

function FloatingPointAddition($n1, n2$)
  $s1, e1, m1 \leftarrow n1.Signo, n1.Exp, n1.Mant$
  $s2, e2, m2 \leftarrow n2.Signo, n2.Exp, n2.Mant$
  $m1 \leftarrow 1.m1$ ▷ Agregar el 1 normalizado
  $m2 \leftarrow 1.m2$ ▷ Agregar el 1 normalizado
  if $n1 = 0 \ \& \ n2 \neq 0$ **then**
    return $n2$
  end if
  if $n1 \neq 0 \ \& \ n2 = 0$ **then**
    return $n1$

end if
if $(n1 = 0 \ \ \& \ \ n2 = 0) \| (s1 \neq s2 \ \ \& \ \ e1 = e2 \ \ \& \ \ m1 = m2)$ **then**
return 0
end if
if $e1 \geq e2$ **then**
$e3 \leftarrow e1$
$m2 \leftarrow RightShift(m2, e1 - e2)$
else
$e3 \leftarrow e2$
$m1 \leftarrow RightShift(m1, e1 - e2)$
end if
if $s1 \neq s2$ **then**
if $m1 > m2$ **then**
$m3 = m1 - m2$
$s3 = s1$
else
$m3 = m2 - m1$
$s3 = s2$
end if
else
$m3 = m1 + m2$
$s3 = s1$
end if
if $s1 \neq s2$ **then**
if $m3.Underflow()$ **then**
$m3 \leftarrow LeftShift(m3, x)$ ▷ x son los bits necesarios para que m3 tenga la forma de mantisa
$e3 \leftarrow e3 - x$
end if
else
if $m3.Overflow()$ **then**
$m3 \leftarrow RightShift(m3, 1)$
$e3 \leftarrow e3 + 1$
end if
end if
$m3 \leftarrow m3.RemoveLeading1()$ ▷ Remueve el 1 normalizado
$n3 = FloatingPoint(s3, e3, m3)$ ▷ Forma el FP
return $n3$
end function

Consideraciones:

- Si una entrada es 0, entonces devuelve la otra.
- Si ambas entradas son 0, o una entrada es el negativo de la otra, devuelve cero.
- Luego de emparejar las mantisas, las sumamos si tienen el mismo signo; caso contrario, restamos la mayor menos la menor, y el signo resultante corresponderá con la mayor.
- Si sumamos, se puede dar Overflow en máximo 1 bit, por lo que debemos desplazar a la derecha dicho bit, y sumarle 1 al exponente. En este paso, es posible perder el decimal menos significativo.
- Si restamos, se puede dar Underflow en varios bits. Nunca da cero porque ese caso ya fue verificado al inicio, entonces siempre habrá un bit de valor 1 para normalizar. La normalización consiste en desplazar a la izquierda hasta que el bit 23, o el 10 según sea el caso de formato FP, tenga el valor de 1.
- Este algoritmo aplica para ambos formatos de FP, solo se debe diferenciar los bits a extraer, a evaluar y a devolver en el resultado en cada caso.

B. Algoritmo para multiplicar números en Floating Point

Este algoritmo es similar al de sumar, pues hay que extraer el signo, exponente y mantisa de los números, y evaluar.

function FLOATINGPOINTMUTIPLICATION($n1, n2$)
$s1, e1, m1 \leftarrow n1.Signo, n1.Exp, n1.Mant$
$s2, e2, m2 \leftarrow n2.Signo, n2.Exp, n2.Mant$
$m1 \leftarrow 1.m1$ ▷ Agregar el 1 normalizado
$m2 \leftarrow 1.m2$ ▷ Agregar el 1 normalizado
if $n1 = 0 \| n2 = 0$ **then**
return 0
end if
$e3 \leftarrow e1 + e2 - bias$
$m1, bits1 \leftarrow RightShiftMantisa(m1)$ ▷ Desplazamos a la derecha las mantisas quitando los ceros decimales.
$m2, bits2 \leftarrow RightShiftMantisa(m2)$
$m3 \leftarrow m1 * m2$
if $m3.Overflow()$ **then**
$exp3 = exp3 + 1$
end if
$m3 \leftarrow Mantisa(m3)$ ▷ Damos forma de mantisa al resultado
$s3 \leftarrow s1 \ EXOR \ s2$
$m3 \leftarrow m3.RemoveLeading1()$ ▷ Remueve el 1 normalizado
$n3 = FloatingPoint(s1, e3, m3)$ ▷ Forma el FP
return $n3$
end function

Consideraciones:

- Si una entrada es 0, entonces devuelve 0.
- Ejecutamos desplazamientos a la derecha en las mantisas para quitar los ceros del final, ya que estos ceros generan una multiplicación muy grande y podría no entrar en los bits del resultado.
- El resultado debe tener tantos decimales como la suma de decimales efectivos (sin contar los ceros que se desplazaron en el paso anterior) de ambas mantisas, por lo tanto, la parte entera puede tener 1 o 2 bits. En caso de tener 2 bits, sumar 1 al exponente para dejarlo con un solo bit.
- Se presentan 3 casos:
  - CASO 1: El primer bit en 1 de la multiplicación está en el bit 23, o bit 10 según sea el caso, entonces no hay nada que hacer.
  - CASO 2: El primer bit en 1 de la multiplicación está a la derecha del bit 23 (Overflow), entonces ejecutar desplazamientos a la izquierda hasta dejarlo en el

bit 23. En este pueden perderse los decimales menos significativos.
  - CASO 3: El primer bit en 1 de la multiplicación está a la izquierda del bit 23 (UnderFlow), entonces ejecutar desplazamientos a la derecha hasta dejarlo en el bit 23.
- Este algoritmo aplica para ambos formatos de FP, solo se debe diferenciar los bits a extraer, a evaluar y a devolver en el resultado en cada caso.

C. Implementación y testeo de la FPU

Los algoritmos descritos han sido implementados en $Verilog$, y para testearlo utilizamos un $test\_vector$ con las siguientes entradas. Cabe resaltar que, al $test\_vector$ se le añade el resultado esperado y los flags esperados.

- Suma de FP de 32 bits
  - $1.0 + 1.0 = 2.0$
  - $23.0 + 15.5 = 38.5$
  - $7.875 + 0.1875 = 8.0625$
  - $0.125 + 100032.125 = 100032.25$
  - $1.5 + 1.5 = 3.0$
  - $1.5 + 0 = 1.5$
  - $0 + 1.5 = 1.5$
  - $-1.5 + 1.5 = 0$
  - $1.5 - 1.5 = 0$
  - $6.875 - 15.5 = -8.625$
  - $15.5 - 6.875 = 8.625$
- Suma de FP de 16 bits
  - $1.625 + 0.5 = 2.125$
  - $0.875 + 10.5 = 11.375$
  - $64.4375 + 18.015625 = 82.453125$
- Multiplicación de FP de 32 bits
  - $1.0 * 1.0 = 1.0$
  - $1.5 * 1.5 = 2.25$
  - $312.4375 * 124.8125 = 38996.10546875$
  - $0.03125 * 7.8125 = 0.244140625$
  - $-1.0 * -1.0 = 1.0$
  - $-1.0 * 1.0 = -1.0$
  - $1.0 * -1.0 = -1.0$
  - $1.0 * 0.0 = 0.0$
  - $0.0 * 0.0 = 0.0$
- Multiplicación de FP de 16 bits
  - $1.625 * 0.5 = 0.8125$
  - $0.03125 * 7.8125 = 0.244140625$
  - $0.0 * 0.0 = 0.0$

El resultado fue positivo ya que no se obtuvieron errores.

```
PS D:\Insync\UTEC\06 - 2021-1\Arquitectura de computadoras\Proyecto\ProyectoArch\FP Unit> vvp.exe .\fpu.out
WARNING: .\fpu_tb.v:59: $readmemh: The behaviour for reg[...] mem[N:0]; $readmemh("...", mem); changed in the 1364-2005 standard. To avoid
ambiguity, use mem[0:N] or explicit range parameters $readmemh("...", mem, start, stop); Defaulting to 1364-2005 behavior.
WARNING: .\fpu_tb.v:59: $readmemh(testvector.tv): Not enough words in the file for the requested range [0:1000].
VCD info: dumpfile fpu.vcd opened for output.
total errors:        0
.\fpu_tb.v:92: $finish called at 275 (1ns)
```

Fig 2.1: Simulación de la FPU

3. Adicionar los cambios en los procesadores

A. Nuevas instrucciones

Nuestros procesadores reconocerán las siguientes nuevas instrucciones:

- MUL: Multiplicación de números de 32 bits a 32 bits.
- $UMULL$: Multiplicación unsigned de números de 32 bits a 64 bits.
- $SMULL$: Multiplicación signed de números de 32 bits a 64 bits.
- $VADDH$: Suma de números en Half FP (16 bits), con y sin Immediate.
- $VADD$: Suma de números en FP (32 bits), con y sin Immediate.
- $VMULH$: Multiplicación de números en Half FP (16 bits), con y sin Immediate.
- $VMUL$: Multiplicación de números en Half FP (16 bits), con y sin Immediate.

Estas instrucciones siempre se darán sobre registros, ya que nuestro inmmediate tiene un límite de bits. Entonces, planteamos la siguiente estructura de instrucción para estas nuevas instrucciones.

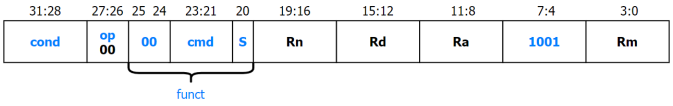

Fig 3.1: Instrucción MUL

donde $\{Rd, Ra\} = Rn \times Rm$ para las operaciones $UMULL$ y $SMULL$.

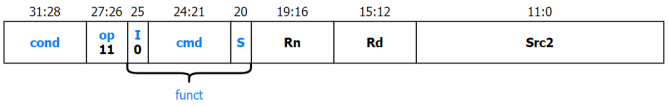

Fig 3.2: Instrucción FP

Donde $I = 0$ demuestra que siempre leeremos los datos de registros, S será 1 cuando necesitamos actualizar los flags y 0 en caso contrario, y cmd está definido por:

cmd	$Operación$
000	MUL
100	$UMULL$
110	$SMULL$

Table 3.1: cmd en instrucciones MUL

cmd	$Operación$
0000	$VADDH$
0001	$VADD$
0010	$VMULH$
0011	$VMUL$

Table 3.2: cmd en instrucciones FP

B. *Modificaciones en el Single-Cycle Processor*

Para que nuestro procesador reconozca correctamente las nuevas instrucciones, necesitamos modificar tanto el datapath como el controller. Los cambios mostrados son los siguientes, de color verde los cambios para las instrucciones MUL y de color rojo los cambios para las instrucciones FP.

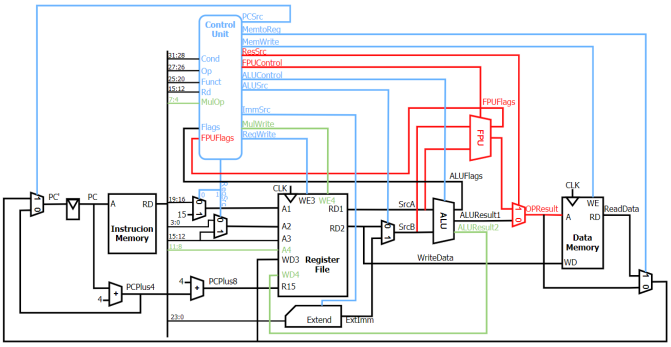

Fig 3.3: Single-Cycle Processor Modificado

Debemos añadir una salida $ResSrc$ al controlador que nos indicará cuándo escogemos el resultado del FPU o del ALU.

Op	$Funct_5$	$Funct_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp	ResSrc
00	0	X	DP Reg	0	0	0	0	XX	1	00	1	0
00	1	X	DP Imm	0	0	0	1	00	1	X0	1	0
01	X	0	STR	0	X	1	1	01	0	10	0	0
01	X	1	LDR	0	1	0	1	01	1	X0	0	0
10	X	X	B	1	0	0	1	10	0	X1	0	0
11	X	X	FP	0	0	0	0	XX	1	00	0	1

Fig 3.4: Lógica de Decoder en el Single-Cycle Processor

Para las instrucciones MUL, añadimos la entrada $MulOp$ para detectar cuándo una instrucción es de multiplicación, y una salida $MulWrite$ para escribir el segundo registro en el $regfile$.

Para las instrucciones FP, añadimos los flags del FPU al $ConditionalLogic$, la señal $FPUFlagW_{1:0}$ cuya función es similar al $FlagW_{1:0}$. Como salidas, añadimos a $ResSrc$, para escoger cuáles son los flags a actualizar y cuál salida entre ALU y FPU continuará el path; y $FPUControl_{1:0}$.

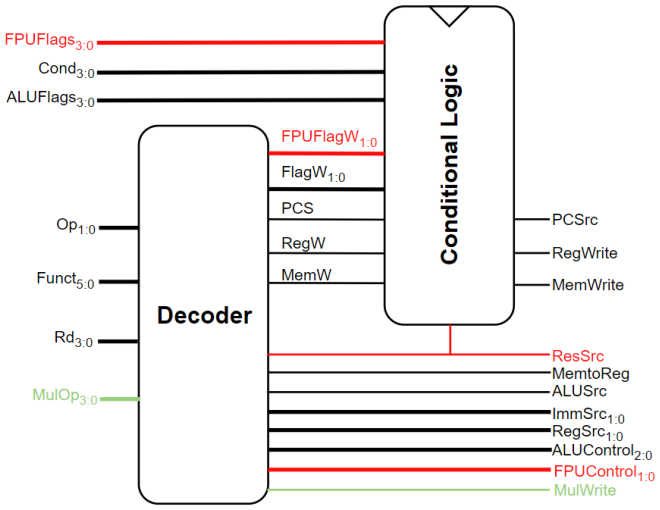

Fig 3.5: Controller del Single-Cycle Processor

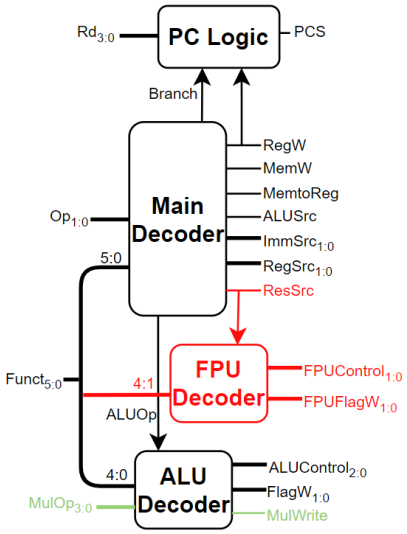

Fig 3.6: Decoder del Single-Cycle Processor

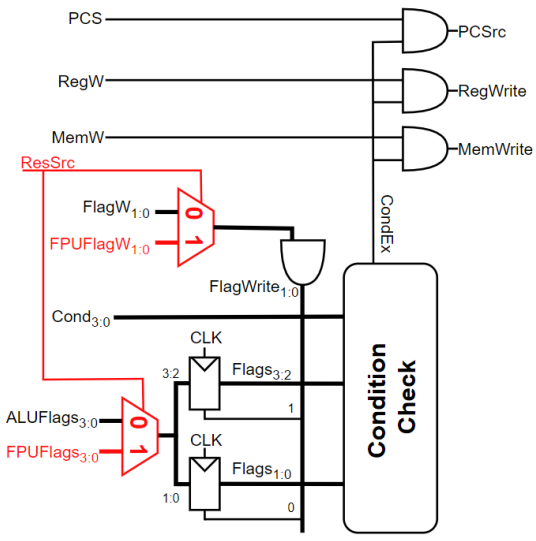

Fig 3.7: CondLogic del Single-Cycle Processor

C. Modificaciones en el Multi-Cycle Processor

Las modificaciones en el Multi-Cycle Processor son muy parecidas que en el Single-Cycle Processor.

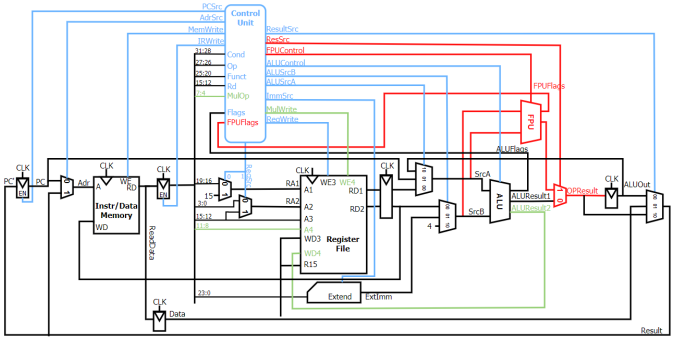

Fig 3.8: Multi-Cycle Processor Modificado

Sin embargo, cambiamos un poco la lógica del Decoder, donde unimos el FPU Decoder y el ALU Decoder.

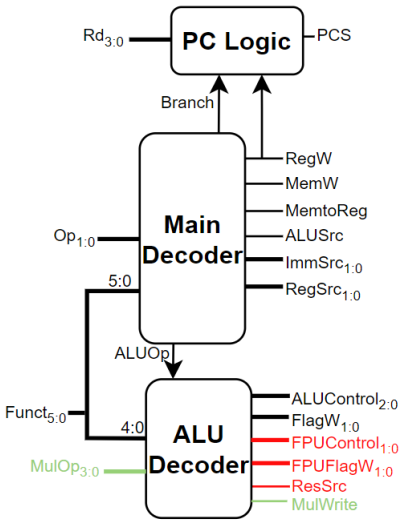

Fig 3.9: Decoder del Multi-Cycle Processor

4. Test de los procesadores

Primero probamos el programa del laboratorio 4 en ambos procesadores, cuyo resultado fue exitoso. De esta forma, comprobamos que los cambios realizados en los procesadores no alteraron sus comportamientos.

Luego, creamos un testbench para testear las nuevas funcionalidades. El programa y su detalle están al final del informe. Los pasos que realiza son los siguientes,

1) Carga el número en FP 4283.25 en R3 y 1.5 en R4.
2) Realiza la suma y multiplicación de números en FP de 32 bits y los guarda en memoria. Si los valores guardados no son los esperados, termina el programa.
3) Aprovechamos los números en R4 y R5 para obtener números grandes de 32 bits cuyo primer bit sea 1, para notar la diferencia entre UMULL y SMULL.
4) Realizamos UMULL y guardamos las dos partes del número en memoria. Si los valores no son los esperados, termina el programa.
5) Realizamos SMULL y guardamos las dos partes del número en memoria. Si los valores no son los esperados, termina el programa.
6) Carga el número en HFP 1.625 en R3 y 0.5 en R4. Realizamos el mismo procedimiento que el punto 2.
7) Cargamos el número en FP 1.5 en R4 y -1.5 en R5.
8) Realizamos la suma de números en FP. Como el segundo es negativo se puede considerar una resta. Guardamos los flags del FPU.
9) Si los flags indican 0, entonces guardará el valor 0 en memoria, de lo contrario, guardará 8 y terminará el programa con error.
10) Si el valor guardado es 0, el programa termina exitosamente.
11) En la formación de los números testeados se utilizó el comando MUL, con lo cual mostramos su correcto funcionamiento.

Al ejecutar el programa obtenes una ejecución exitosa. Mostramos las partes principales de la simulación en GTKWave.

A. Simulación Single-Cycle Processor

Ejecución de $4284.75 + 1.5$ y 4284.75×1.5 en punto flotante de 32 bits. En operaciones FP, la señal $ResSrc$ se activa. El resultado es procesado en $FPUResult$ y guardado en memoria ($WriteData$) en la siguiente iteración.

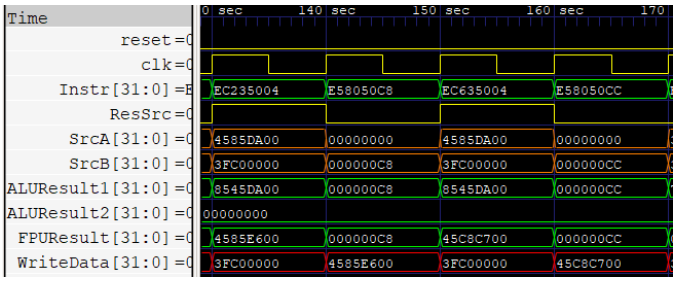

Fig 4.1: Single-Cycle Processor Simulación parte 1

En la operación UMULL, se genera un número de 64 bits que es guardado en 2 registros. Note que, en los dos siguientes ciclos, se guardan sus valores en memoria ($WriteData$).

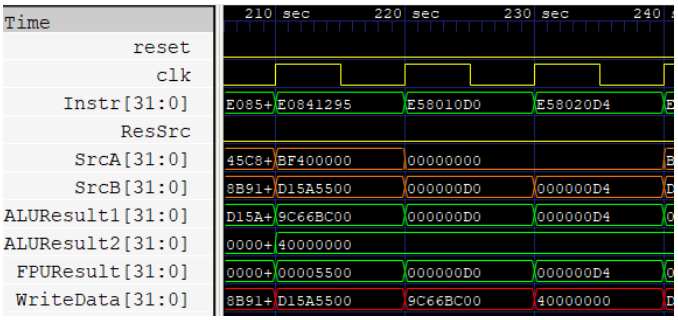

Fig 4.2: Single-Cycle Processor Simulación parte 2

En la operación SMULL, se operan los números $SrcA$ y $SrcB$ en signed, lo que significa que su bit más significativo es 1 para negativos y 0 para positivos. Note que, en los dos siguientes ciclos, se guardan sus valores en memoria ($WriteData$).

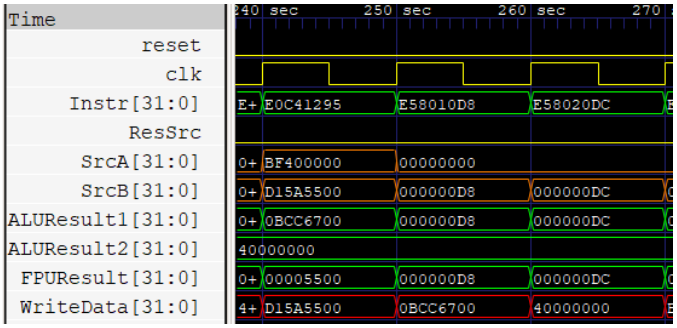

Fig 4.3: Single-Cycle Processor Simulación parte 3

Ejecución de $1.625 + 0.5$ y 1.625×0.5 en punto flotante de 16 bits (Half Floating Point), En operaciones FP, la señal $ResSrc$ se activa. El resultado es procesado en $FPUResult$ y guardado en memoria ($WriteData$) en la siguiente iteración.

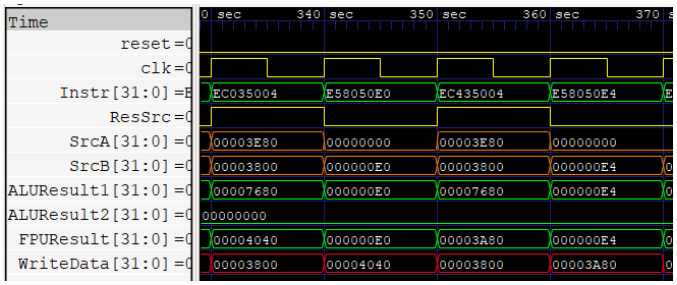

Fig 4.4: Single-Cycle Processor Simulación parte 4

Ejecución de $1.5 + (-1.5)$ activando los flags en el FPU. Note que solo se ejecuta la primera de las dos instrucciones STR (según $MemWrite$) debido al flag Z.

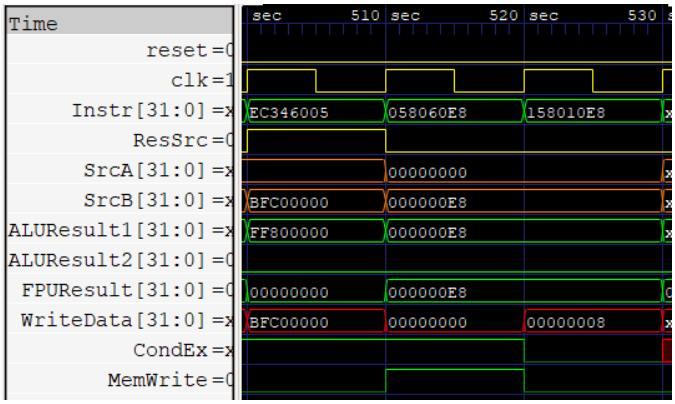

Fig 4.5: Single-Cycle Processor Simulación parte 5

B. Simulación Multi-Cycle Processor

Ejecución de $4284.75 + 1.5$ en punto flotante de 32 bits. La señal $ResSrc$ se activa en la etapa ExecuteR. En la siguiente instrucción, el resultado se guarda en memoria.

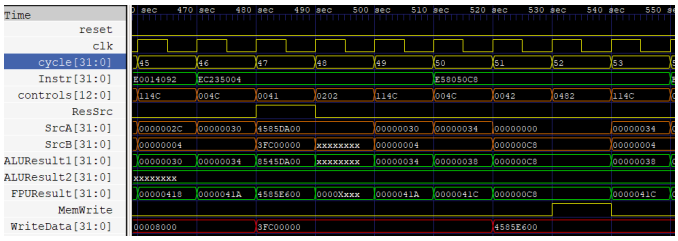

Fig 4.6: Multi-Cycle Processor Simulación parte 1a

Ejecución de 4284.75×1.5 en punto flotante de 32 bits. La señal $ResSrc$ se activa en la etapa ExecuteR. En la siguiente instrucción, el resultado se guarda en memoria.

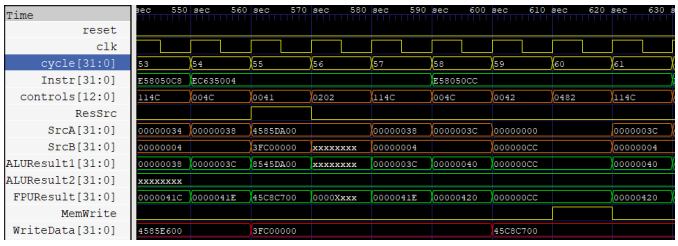

Fig 4.7: Multi-Cycle Processor Simulación parte 1b

Ejecución de UMULL. Note que, en los dos siguientes ciclos, se guardan sus valores en memoria ($WriteData$).

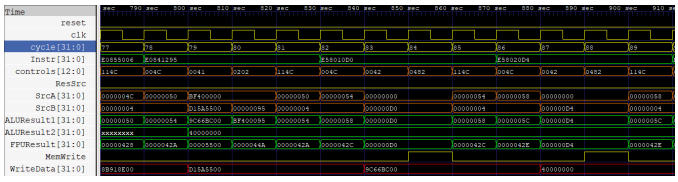

Fig 4.8: Multi-Cycle Processor Simulación parte 2

Ejecución de SMULL. Note que, en los dos siguientes ciclos, se guardan sus valores en memoria ($WriteData$).

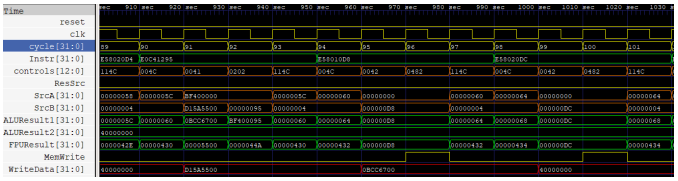

Fig 4.9: Multi-Cycle Processor Simulación parte 3

Ejecución de $1.625 + 0.5$ en punto flotante de 16 bits. La señal $ResSrc$ se activa en la etapa ExecuteR. En la siguiente instrucción, el resultado se guarda en memoria.

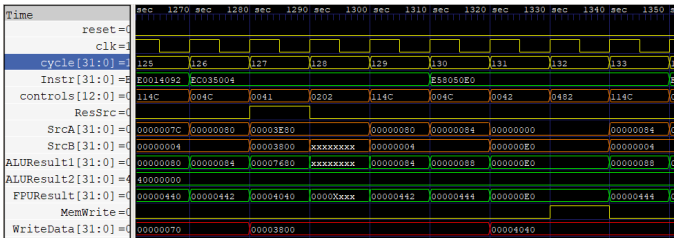

Fig 4.10: Multi-Cycle Processor Simulación parte 4a

Ejecución de 1.625×0.5 en punto flotante de 16 bits. La señal $ResSrc$ se activa en la etapa ExecuteR. En la siguiente instrucción, el resultado se guarda en memoria.

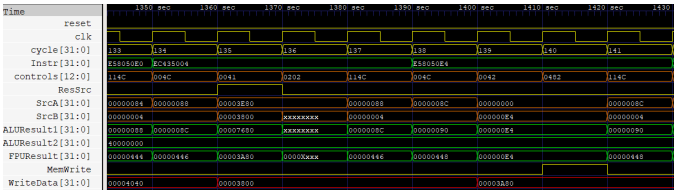

Fig 4.11: Multi-Cycle Processor Simulación parte 4b

Ejecución de $1.5 + (-1.5)$ y activación de flags. El primer STR es ejecutado debido al flag Z, el segundo no se ejecuta por la misma razón, además que el programa finaliza cuando se ejecuta correctamente el primer STR.

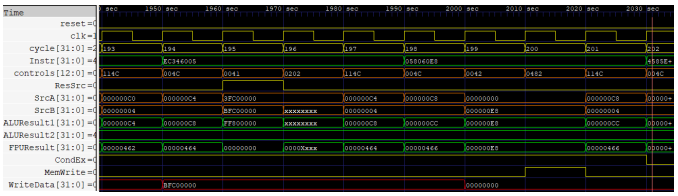

Fig 4.12: Multi-Cycle Processor Simulación parte 5

5. Programa 1: Factorial grande

El siguiente programa utiliza las funciones de multiplicación entera añadidas a nuestro ALU, el cual trata de calcular el factorial de un número n, donde $13 \leq n < 20$ utilizando loops.

A. Código en ASM

```
.global _start
_start:
    SUB R0, R15, R15        // R0 = 0
    ADD R1, R0, #19          // R1 = n = 19
    ADD R2, R0, #12         // R2 = m = 12
    SUB R3, R1, #20         // R3 = -1
    ADD R4, R0, #1          // R4 = i = 1
    ADD R5, R0, #1          // R5 = fact1 = 1
LOOP1:
    SUBS R6, R4, R2         // i <= m?
    BGT END1                // Case: Yes, End loop
    MUL R5, R5, R4          // fact1 = fact1 * i
    ADD R4, R4, #1          // i = i + 1
    B LOOP1                 // Branch to LOOP1
END1:
    ADD R6, R0, #1          // R6 = fact2 = 1
LOOP2:
    SUBS R7, R4, R1         // i <= n?
    BGT END2                // Case: Yes, End loop
    MUL R6, R6, R4          // fact2 = fact2 * i
    ADD R4, R4, #1          // i = i + 1
    B LOOP2                 // Branch to LOOP2
END2:
    UMULL R7, R5, R6, R8    // fact1 * fact2 = n!
    SMULL R9, R5, R3, R10   // m! * -1 = -m!
    STR R7, [R0, #100]      // Store at mem[100]
    STR R8, [R0, #104]      // Store at mem[104]
    STR R9, [R0, #108]      // Store at mem[108]
    STR R10, [R0, #112]     // Store at mem[112]
```

B. Explicación

La forma de hallar el factorial es un tanto peculiar, debido al hecho de que primero calculamos 12! utilizando la instrucción MUL, luego hallamos el resultado de multiplicar desde el 13 hasta n también utilizando MUL y, finalmente, unimos los dos resultados, pero esta vez utilizando la instrucción UMULL. Como extra, se calcula el negativo de 12! con la finalidad de utilizar la instrucción SMULL al final del programa.

- Inicializamos R1, R2, R3, R4 y R5 en 19, 12, -1, 1, 1 respectivamente.
- R1 será nuestro número n, el cual, en este caso, vale 19.
- R2 equivaldrá a 12 por lo explicado hace un momento.
- R3 guardará el valor de -1 para utilizarlo al final del programa y obtener el negativo de 12!.
- R4 representará al contador de nuestros loops, al cual se le denomina comúnmente como i.
- R5 alojará el valor de 12!.
- El contador i, en el primer loop, irá de 1 a 12 y cuando llegue a 13, saldrá del loop.
- Luego, inicializamos R6 en 1 para allí almacenar la segunda parte de nuestro factorial. En este caso será $13 * 14 * 15 * 16 * 17 * 18 * 19$, el cual se hallará en el segundo loop.
- Nuestro contador i, en este caso, irá de 13 a 20. Cuando llegue a 20, saldrá del loop.
- En la parte final, se multiplican R5 y R6 para hallar $n!$. En nuestro ejemplo, 19!.
- También se halla el negativo de 12! multiplicando R5 con R3.
- Por último, se guardan los resultados en memoria.

C. Ejecución

Ejecutamos el programa en ambos procesadores, con las condiciones descritas previamente. Los resultados fueron como esperábamos.

```
PS D:\ProyectoArch\Single cycle processor\src> iverilog -o top.out top.v testbenchFactorial.v
PS D:\ProyectoArch\Single cycle processor\src> vvp top.out
WARNING: ./imem.v:8: $readmemh: The behaviour for reg[...] mem[N:0]; $readmemh(*...*, mem); changed in the 1364-2005 standard. To avoid ambiguity, u
se mem[0:N] or explicit range parameters $readmemh(*...*, mem, start, stop). Defaulting to 1364-2005 behavior.
WARNING: ./imem.v:8: $readmemh(memfileFactorial.mse): Not enough words in the file for the requested range [0:63].
VCD info: dumpfile top.vcd opened for output.
Simulation succeded Factorial LSB
Simulation succeded Factorial MSB
Simulation succeded Negative Factorial LSB
Simulation succeded Negative Factorial MSB
```

Fig 5.1: Programa 1 en Single-Cycle Processor

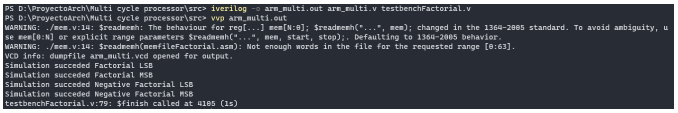

Fig 5.2: Programa 1 en Multi-Cycle Processor

Para ver los resultados de la ejecución del programa 1, por favor visitar la sección D de los Anexos.

6. Programa 2: Sumatoria de 0.5^i

El siguiente programa utiliza las funciones de suma y multiplicación de punto flotante implementadas. La función calcula la siguiente sumatoria:

$$\sum_{i=1}^{n} 0.5^{i-1}$$

, para alguna entrada $n > 0$

A. Código en ASM

```
.global _start
_start:
    SUB R0, R15, R15
        //R1 = 1.0 = 0x3f800000 = 1065353216 = 2^23*127
    ADD R1, R0, #32
    MUL R1, R1, R1          //R1 = 2^10
    MUL R1, R1, R1          //R1 = 2^20
    ADD R2, R0, #8
    MUL R1, R1, R2          //R1 = 2^23
    ADD R2, R0, #127
    MUL R1, R1, R2          //R1 = 2^23*127 = 1.0
        //R2 = 0.5 = 0X3f000000 = 1056964608 = 2^24*63
    ADD R2, R0, #64         //R2 = 2^6
    MUL R2, R2, R2          //R2 = 2^12
    MUL R2, R2, R2          //R2 = 2^24
    ADD R3, R0, #63
    MUL R2, R2, R3          //R2 = 2^24*63 = 0.5 en FP

    ADD R5, R1, #0          //R5 = Respuesta

    ADD R3, R0, #3          //i = 3
    LOOP:
        SUBS R3, R3, #1     //i = i - 1
        BEQ END             //i = 0? Goto End
        VMUL R1, R1, R2     //R1 = R1 * R2
        VADD R5, R5, R1     //R5 = R5 + R1
        B LOOP              //Loop
    END:
    STR R5, [R0, #200]      //mem[200] = Respuesta
```

B. Explicación

- Inicializa R1 en 1.0 y R2 en 0.5.
- En R5 guardamos la respuesta, que empieza en 1.0.
- Asignamos i a R3, el cual será el valor de n para la sumatoria. El índice i disminuye en 1 en cada iteración, cuando sea 0, termina el Loop.
- En cada iteración, se calula en R1 la siguiente potencia de 0.5, y lo suma al resultado.
- Al terminar el loop, guarda el resultado final en memoria.

C. Ejecución

Ejecutamos el programa en ambos procesadores, para distintos valores de n. Obtenemos resultados satisfactorios.

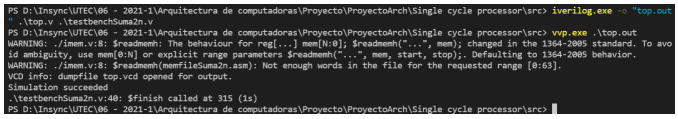

Fig 6.1: Programa 2 en Single-Cycle Processor

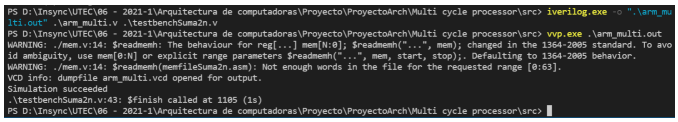

Fig 6.2: Programa 2 en Multi-Cycle Processor

n	*Resultado*	FP
1	1.0	$0x3f800000$
2	1.5	$0x3fc00000$
3	1.75	$0x3fe00000$
4	1.875	$0x3ff00000$
5	1.9375	$0x3ff80000$
6	1.96875	$0x3ffc0000$

Table 6.1: Resultados del programa 2

7. ANEXOS

A. Diagrama de los procesadores

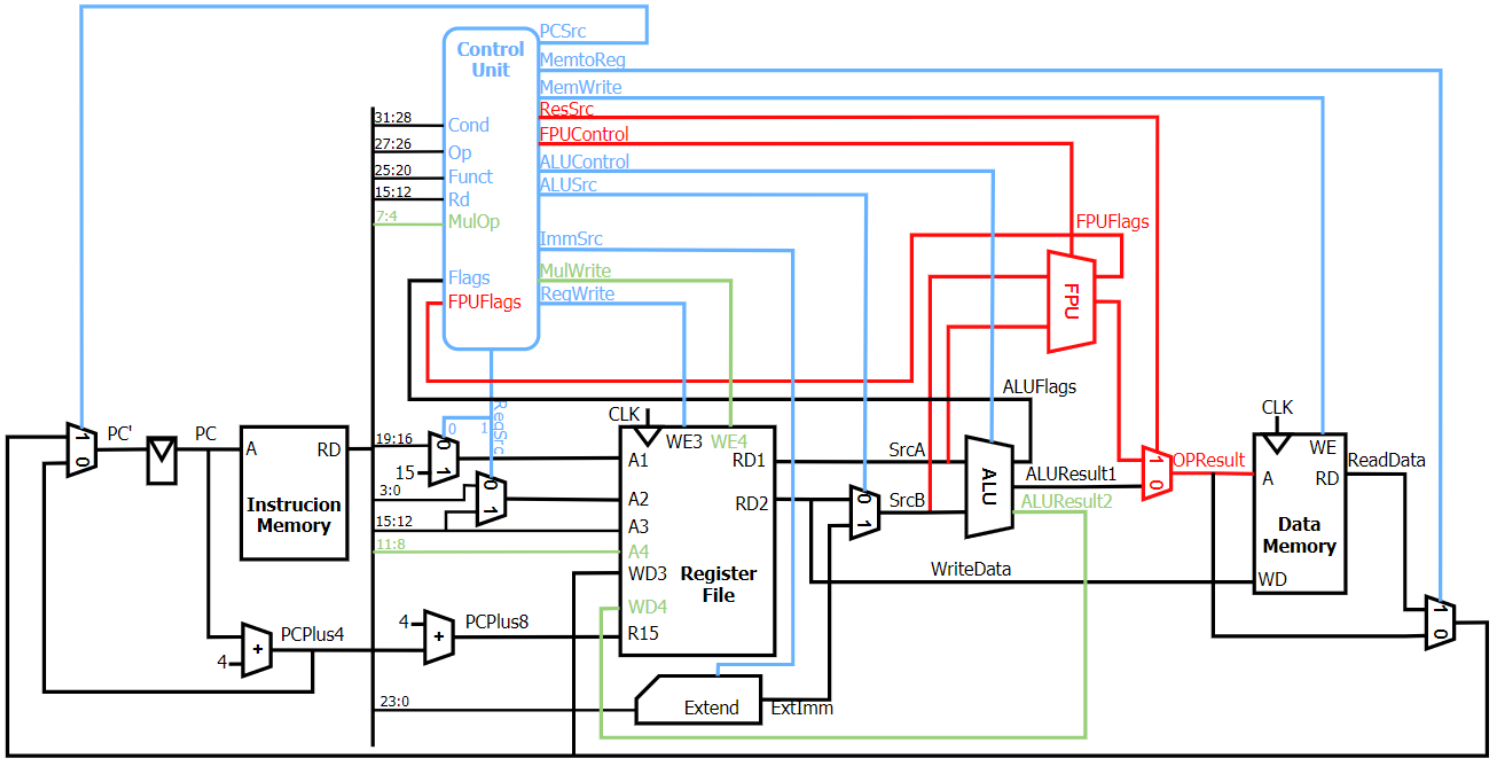

Fig 7.1: Single-Cycle Processor

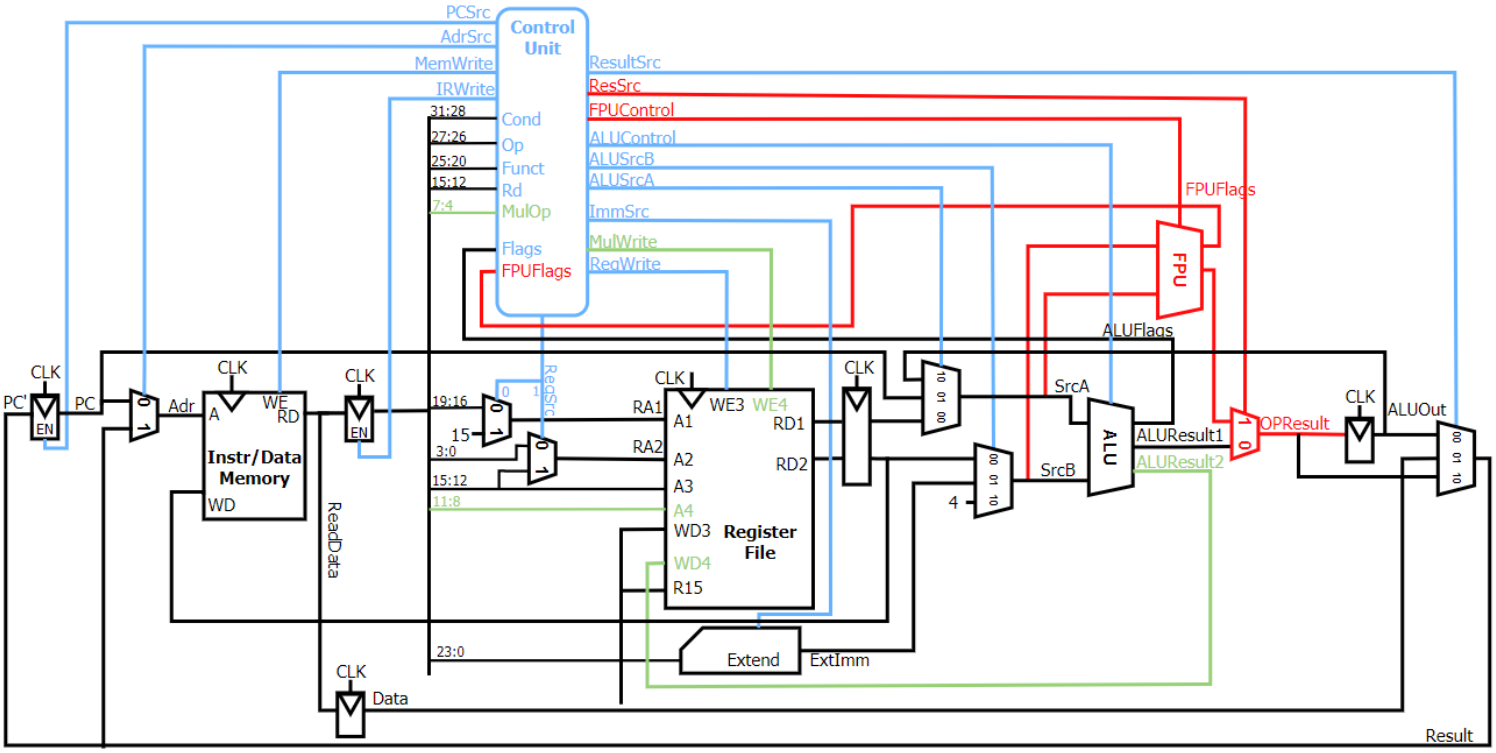

Fig 7.2: Multi-Cycle Processor

.

B. Programa Sumatoria

```
.global _start
_start:
    SUB R0, R15, R15     //R0 = 0                                  1110 00 000100 1111 0000 00000000 1111     E04F000F
    //-------------------------------------------------------------
    //--------------------ADD y MUL en FP 32 bits-------------------
    //-------------------------------------------------------------
        //Buscamos R3 = 4 283.25 = 0x4585DA00 = 1 166 400 000
    ADD R1, R0, #180     //R1 = 180                                1110 00 101000 0000 0001 00001011 0100     E28010B4
    ADD R2, R0, #200     //R2 = 200                                1110 00 101000 0000 0010 00001100 1000     E28020C8
    MUL R3, R1, R1       //R3 = 32 400                             1110 00 000000 0001 0011 00001001 0001     E0013091
    MUL R3, R3, R1       //R3 = 5 832 000                          1110 00 000000 0011 0011 00001001 0001     E0033091
    MUL R3, R3, R2       //R3 = 1 166 400 000                      1110 00 000000 0011 0011 00001001 0010     E0033092
        //Buscamos R4 = 1.5 = 0x3fc00000 = 1 069 547 520
    ADD R1, R0, #128     //R1 = 128                                1110 00 101000 0000 0001 00001000 0000     E2801080
    MUL R2, R1, R1       //R2 = 16 384                             1110 00 000000 0001 0010 00001001 0001     E0012091
    ADD R2, R2, R2       //R2 = 32 768                             1110 00 001000 0010 0010 00000000 0010     E0822002
    SUB R1, R2, R1       //R1 = 32 640                             1110 00 000100 0010 0001 00000000 0001     E0421001
    MUL R4, R1, R2       //R4 = 1 069 547 520                      1110 00 000000 0001 0100 00001001 0010     E0014092

    VADD R5, R3, R4      //R5 = R3 + R4 (FP) = 4 284.75            1110 11 000010 0011 0101 00000000 0100     EC235004
    STR R5, [R0, #200]   //mem[200] = 4 284.75 = 0x4585e600        1110 01 011000 0000 0101 00001100 1000     E58050C8
    VMUL R5, R3, R4      //R5 = R3 * R4 (FP) = 6 424.875           1110 11 000110 0011 0101 00000000 0100     EC635004
    STR R5, [R0, #204]   //mem[204] = 6 424.875 = 0x45c8c700       1110 01 011000 0000 0101 00001100 1100     E58050CC

    //-------------------------------------------------------------
    //-------------------------UMULL y SMULL-----------------------
    //-------------------------------------------------------------
        //Tenemos  R4 = 0x3fc00000 = 1 069 547 520
        //Buscamos R4 = 0xD15A5500 = 3 208 642 560 = 3*R4
    ADD R6, R4, R4       //R6 = 2*R4                               1110 00 001000 0100 0110 00000000 0100     E0846004
    ADD R4, R4, R6       //R4 = 3*R4 = 0xBF400000                  1110 00 001000 0100 0100 00000000 0110     E0844006
        //Tenemos  R5 = 0x45c8c700 = 1 170 786 048
        //Buscamos R5 = 0xD15A5500 = 3 512 358 144 = 3*R5
    ADD R6, R5, R5       //R6 = 2*R5                               1110 00 001000 0101 0110 00000000 0101     E0856005
    ADD R5, R5, R6       //R5 = 3*R5 = 0xD15A5500                  1110 00 001000 0101 0101 00000000 0110     E0855006
        //Unsigned R4*R5 = 11 269 901 826 801 008 640  = 0x9C66BC00 40000000 = {R1 R2}
        //R4*R5 = R1,R2
    UMULL R1, R4, R5, R2//{R1, R2} = R4 x R5                     1110 00 001000 0100 0001 00101001 0101     E0841295
    STR R1, [R0, #208]   //mem[208] = R1 = 0x9C66BC00              1110 01 011000 0000 0001 00001101 0000     E58010D0
    STR R2, [R0, #212]   //mem[212] = R2 = 0x40000000              1110 01 011000 0000 0010 00001101 0100     E58020D4

        //R4 = BF40 0000 = -1086324736 (en Twos Complement, signed)
        //R5 = D15A 5500 = -782609152  (en Twos Complement, signed)
        //Signed R4*R5 = 850 167 680 437 583 872 = 0x0BCC6700 40000000 = {R1, R2}
    SMULL R1, R4, R5, R2//{R1, R2} = R4 x R5                     1110 00 001100 0100 0001 00101001 0101     E0C41295
    STR R1, [R0, #216]   //mem[216] = R1 = 0x0BCC6700              1110 01 011000 0000 0001 00001101 1000     E58010D8
    STR R2, [R0, #220]   //mem[220] = R2 = 0x40000000              1110 01 011000 0000 0010 00001101 1100     E58020DC

    //-------------------------------------------------------------
    //--------------------ADD y MUL en FP 16 bits------------------
    //-------------------------------------------------------------
        //Buscamos R3 = 1.625 = 0x00003E80 = 16 000 = 100 * 160
    ADD R1, R0, #100     //R1 = 100                                1110 00 101000 0000 0001 00000110 0100     E2801064
    ADD R2, R0, #160     //R2 = 160                                1110 00 101000 0000 0010 00001010 0000     E28020A0
    MUL R3, R1, R2       //R3 = R1 * R2                            1110 00 000000 0001 0011 00001001 0010     E0013092
        //Buscamos R4 = 0.5 = 0x00003800 = 14 336 = 128 * 112
    ADD R1, R0, #128     //R1 = 128                                1110 00 101000 0000 0001 00001000 0000     E2801080
    ADD R2, R0, #112     //R2 = 112                                1110 00 101000 0000 0010 00000111 0000     E2802070
    MUL R4, R1, R2       //R4 = R1 * R2                            1110 00 000000 0001 0100 00001001 0010     E0014092

    VADDH R5, R3, R4     //R5 = R3 + R4 (HFP) = 2.125              1110 11 000000 0011 0101 00000000 0100     EC035004
    STR R5, [R0, #224]   //mem[224] = 2.125 = 0x00004040           1110 01 011000 0000 0101 00001110 0000     E58050E0
    VMULH R5, R3, R4     //R5 = R3 * R4 (HFP) = 0.8125             1110 11 000100 0011 0101 00000000 0100     EC435004
    STR R5, [R0, #228]   //mem[228] = 0.8125 = 0x00003A80          1110 01 011000 0000 0101 00001110 0100     E58050E4

    //-------------------------------------------------------------
    //---------------------------Flags en FP-----------------------
    //-------------------------------------------------------------
        //Buscamos R4 = 1.5 = 0x3FC00000 = 1 069 547 520
    ADD R1, R0, #128     //R1 = 2^7 = 128                          1110 00 101000 0000 0001 00001000 0000     E2801080
    MUL R2, R1, R1       //R2 = 2^14 = 16 384                      1110 00 000000 0001 0010 00001001 0001     E0012091
    ADD R2, R2, R2       //R2 = 2^15 = 32 768                      1110 00 001000 0010 0010 00000000 0010     E0822002
    SUB R1, R2, R1       //R1 = 32 768 - 128 = 32640               1110 00 000100 0010 0001 00000000 0001     E0421001
    MUL R4, R1, R2       //R4 = 1 069 547 520                      1110 00 000000 0001 0100 00001001 0010     E0014092

        //Buscamos R5 = -1.5 = 0xBFC00000 = 3 217 031 168 = 8 * 16^7 + R4
    ADD R1, R0, #16      //R1 = 16                                 1110 00 101000 0000 0001 00000001 0000     E2801010
    MUL R2, R1, R1       //R2 = 16^2                               1110 00 000000 0001 0010 00001001 0001     E0012091
    MUL R3, R2, R2       //R3 = 16^4                               1110 00 000000 0010 0011 00001001 0010     E0023092
    MUL R5, R1, R2       //R5 = 16^3                               1110 00 000000 0001 0101 00001001 0010     E0015092
    MUL R5, R5, R3       //R5 = 16^7                               1110 00 000000 0101 0101 00001001 0011     E0055093
    ADD R1, R0, #8       //R1 = 8                                  1110 00 101000 0000 0001 00000000 1000     E2801008
    MUL R5, R5, R1       //R5 = 8*16^7                             1110 00 000000 0101 0101 00001001 0001     E0055091
    ADD R5, R4, R5       //R5 = R5 + R4 = -1.5                     1110 00 001000 0100 0101 00000000 0101     E0845005

    VADDS R6, R4, R5     //R6 = 1.5 + (-1.5) = 0                   1110 11 000011 0100 0110 00000000 0101     EC346005
    STREQ R6, [R0, #232]//mem[232] = 0                           0000 01 011000 0000 0110 00001110 1000     058060E8
    STRNE R1, [R0, #232]//mem[232] = 8                           0001 01 011000 0000 0001 00001110 1000     158010E8
```

C. Programa Factorial Grande

```
.global _start
_start:
    SUB R0, R15, R15        // R0 = 0              1110 0000 0100 1111 0000 0000 0000 1111 E04F000F
    ADD R1, R0, #19         // R1 = n = 19         1110 0010 1000 0000 0001 0000 0001 0011 E2801013
    ADD R2, R0, #12         // R2 = m = 12         1110 0010 1000 0000 0010 0000 0000 1100 E280200C
    SUB R3, R1, #20         // R3 = -1             1110 0010 0100 0001 0011 0000 0001 0100 E2413014
    ADD R4, R0, #1          // R4 = i = 1          1110 0010 1000 0000 0100 0000 0000 0001 E2804001
    ADD R5, R0, #1          // R5 = fact1 = 1      1110 0010 1000 0000 0101 0000 0000 0001 E2805001
LOOP1:
    SUBS R6, R4, R2         // i <= m?             1110 0000 0101 0100 0110 0000 0000 0010 E0546002
    BGT END1                // Case: Yes, End loop 1100 1010 0000 0000 0000 0000 0000 0010 CA000002
    MUL R5, R5, R4          // fact1 = fact1 * i   1110 0000 0000 0101 0101 0000 1001 0100 E0055094
    ADD R4, R4, #1          // i = i + 1           1110 0010 1000 0100 0100 0000 0000 0001 E2844001
    B LOOP1                 // Branch to LOOP1     1110 1010 1111 1111 1111 1111 1111 1010 EAFFFFFA
END1:
    ADD R6, R0, #1          // R6 = fact2 = 1      1110 0010 1000 0000 0110 0000 0000 0001 E2806001
LOOP2:
    SUBS R7, R4, R1         // i <= n?             1110 0000 0101 0100 0111 0000 0000 0001 E0547001
    BGT END2                // Case: Yes, End loop 1100 1010 0000 0000 0000 0000 0000 0010 CA000002
    MUL R6, R6, R4          // fact2 = fact2 * i   1110 0000 0000 0110 0110 0000 1001 0100 E0066094
    ADD R4, R4, #1          // i = i + 1           1110 0010 1000 0100 0100 0000 0000 0001 E2844001
    B LOOP2                 // Branch to LOOP2     1110 1010 1111 1111 1111 1111 1111 1010 EAFFFFFA
END2:
    UMULL R7, R5, R6, R8    // fact1 * fact2 = n!  1110 0000 1000 0101 1000 0111 1001 0110 E0858796
    SMULL R9, R5, R3, R10   // m! * -1 = -m!       1110 0000 1100 0101 1010 1001 1001 0011 E0C5A993
    STR R7, [R0, #100]      // Store at mem[100]   1110 0101 1000 0000 0111 0000 0110 0100 E5807064
    STR R8, [R0, #104]      // Store at mem[104]   1110 0101 1000 0000 1000 0000 0110 1000 E5808068
    STR R9, [R0, #108]      // Store at mem[108]   1110 0101 1000 0000 1001 0000 0110 1100 E580906C
    STR R10, [R0, #112]     // Store at mem[112]   1110 0101 1000 0000 1010 0000 0111 0000 E580A070
```

D. Resultados de Ejecución del Programa 1

V	$Result$	$ResultHexa$	MSB	LSB
12!	479001600	$0x1C8CFC00$	$0x00000000$	$0x1C8CFC00$
$-12!$	-479001600	$0xE3730400$	$0xFFFFFFFF$	$0xE3730400$
19!	121645100408832000	$0x01B02B9306890000$	$0x01B02B93$	$0x06890000$

Table 7.1: Resultados del programa 1