Computer Architecture Project

Rodrigo Gabriel Salazar Alva 100%, Juan Sebastián Sara Junco 100%, Francisco Javier Magot Barrera 100%

I. Introducción y Objetivos

A lo largo del curso, se ha podido lograr implementar nuestro propio "ALU" de un procesador ARM, así como completar un procesador "Single Cycle" y finalmente construir nuestro propio procesador "Multicycle". Por lo tanto, tomando como base estas previas actividades, se ha tomado como iniciativa, ampliar nuestros conocimientos.
Es por esta razón, que en el presente trabajo se tiene como finalidad dos objetivos. En primer lugar, se tendrá que implementar en el ALU de nuestro procesador ARM, las siguientes operaciones.

- MUL
- UMULL
- SMULL

En segundo lugar, se buscara extender nuestro propio microprocesador, implementando un "Floating Point Unit" (FPU) que opere para valores en "Single Precision" (32 bits) y "Half Precision" (16 bits), de tal manera que pueda realizar las siguientes operaciones:

- ADD
- MUL
- Solucionar el Overflow

Tomando en cuenta lo previamente indicado, para la realización de este trabajo, se utilizara un HDL, en este caso "Verilog", asimismo, se contara con el software de "GTKWave" que nos brinde los "Waveforms" de nuestras nuevas señales que se incorporaran en nuestro procesador.

II. Implementación en Verilog: Modificación de ALU

A. Encoding

El ALU sobre el cual se esta trabajando soporta cuatro instrucciones (*sum*, *sub*, *or* y *and*), teniendo un encoding de ALUControl de 2 bits. No obstante, con la adición de tres nuevas instrucciones de multiplicación, no es posible representar el nuevo total de siete instrucciones con solo 2 bits. Por ello, es necesario agrandar las dimensiones de este puerto a 3 bits. El encoding propuesto para esta nuevo repertorio de instrucciones se presenta en la siguiente.

Instrucción	Enconding	Flags
sum	000	NZCV
sub	001	NZ
and	010	NZ
or	011	NZ
mul	100	NZ
umull	101	NZ
smull	111	NZ

Para el planteamiento de este nuevo encoding se emplearon las siguientes consideraciones para simplificar el circuito resultante y disminuir el numero de modificaciones necesarias sobre el ALU previo.

- El encoding de las instrucciones se conservo extendiéndolas únicamente con un 0 en su bit más significativa. De esta forma es posible conservar el bloque casex previamente implementado con la adición de los 0's correspondiente.
- Todas las operaciones de multiplicación tienen un 1 en su bit mas significativo. Esto nos permite implemntar todas estas operaciones en un único caso en el bloque casex de Result.
- De las operaciones de multiplicación, únicamente las que tiene resultados de 64 bits tienen un 1 en su bit menos significativo. Esto ayuda a simplificar los casos utilizados para la detección de flags.
- Se distingue entre operaciones de multiplicación con signo y sin signo mediante su segundo bit. Esto resulta útil tanto para determinar que resultado de multiplicación es dirigido al wire mull como para simplificar la lógica de casos para la detección de flags.

B. Modificación de puertos

Cabe destacar que dentro de las consideraciones explicadas en la sección anterior se mencionan un aspectos importante que representan un cambio significativo para el ALU: existen operación cuyo resultado puede ser de 64 bits. Para ello existen múltiples posibles soluciones pero en este caso se decidió añadir un nuevo puerto de 32 bits Auxiliar que se encargaría de guardar los bits mas significativos de estos resultados. Esta decisión de diseño se sustenta en el hecho de que estos bits son rellenados en operaciones de multiplicación. Si se guardase todo en un mismo puerto esto resultaría en que los 32 bits mas significativos estarían rellenados de valores X o valores basura en una gran mayoría de las operaciones. En cambio, se considera que aislar este valor y asignarle siempre el resultado de la multiplicación (aun si no es la operación) es una mejor opción que permite aislar la lógica de multiplicación en la micro arquitectura de los procesadores y facilita la comprensibilidad de los outputs producidos. A continuación se muestra entonces la nueva cabecera del modulo ALU.

Listing 1. ALU: Nueva cabecera del modulo en alu.v

```
module ALU(a,b,Result,
           ALUFlags,ALUControl,
           Auxiliar);
  input [31:0] a;
  input [31:0] b;
  input [2:0] ALUControl;
```

```
output reg [31:0] Result;
output wire [3:0] ALUFlags;
// ADDED: Auxiliar output
// for long multiplication
output wire [31:0] Auxiliar;
```

C. Operaciones de Multiplicación

Con estas consideraciones y modificaciones se procede a implementar la multiplicación en si. Para esto se utiliza el operador behavioral * de verilog que permite asignar el producto entre dos puertos de tamaño N a un nuevo puerto de tamaño 2N. Esto permite una rápida implementación de UMULL y MUL siendo esta segunda operación una versión corta de la primera. No obstante, para SMULL es necesario hacer que los inputs a y b sean interpretados como números con signo. Para ello se agregan elementos "wire signed" intermediarios que permiten ahora si implementar SMULL de forma similar a lo descrito anteriormente. Posteriormente, en un wire mull se escoge entre los dos output producidos (con y sin signo) utilizando un operador ternario que revisa el segundo bit del ALUControl. A continuación se muestra el fragmento de código que implementa estas especificaciones.

Listing 2. ALU: Calculo de multiplicaciones en alu.v

```
// ADDED: Singed multiplication
// calculation
wire signed [31:0]sa,sb;
wire signed [63:0]smull;
assign sa = a;
assign sb = b;
assign smull = sa*sb;

// ADDED: Unsinged multiplication
// calculation
wire [63:0]umull;
assign umull = a*b;

// ADDED: Assign to mull
assign mull = (ALUControl[1]?smull:umull);
```

D. Logica de Flags

Finalmente, habiendo implementando el calculo de la multiplicación en si, es necesario ahora implementar la lógica de las Flags. Para esto, dado a que no se ven afectadas, la lógica de CV no se modificada de forma considerable, únicamente extendiendo un 0 para la comparación de ALUControl. Por otro lado, las flags de neg y zero si sufren cambios considerables con la adición de casos. Para la flag zero (Z) se agrega un caso adicional correspondiente a las multiplicaciones long: en estas no solo basta con confirmar que Result=0, sino que también es necesario validar que Auxiliar=0. Asimismo, para la flag neg(N) se agregan dos casos: si es una operación MUL o UMULL entonces siempre es 0 y si es SMULL se revisa Auxiliar[31] en vez de Result[31]. La implementación de estos cambios de se presenta a continuación:

Listing 3. ALU: Nueva logica de Flags en alu.v

```
//Flags
// NZ: Afectadas por todas las operaciones
// ADDED: always block for cases of
// neg and zero
always @(*) begin
casex (ALUControl[2:0])
  3'b1?1: // SMULL & UMULL
    zero = (
      (Result == 32'b0)
      &
      (Auxiliar == 32'b0)
    );
  default: // All other operations
    zero = (Result == 32'b0);
endcase
end
always @(*) begin
casex(ALUControl[2:0])
  3'b10?: // MUL & UMULL
    neg = 1'b0;
  3'b111: // SMULL
    neg = Auxiliar[31];
  default: // All other operations
    neg = Result[31];
endcase
end
// CV: Solo afectadas por ADD y SUB
assign carry = (
  (ALUControl[2:1] == 2'b00)
  &
  sum[32]
);
assign overflow = (
  (ALUControl[2:1] == 2'b00)
  &
  (sum[31] ^ a[31])
  &
  ~(ALUControl[0] ^ a[31] ^ b[31])
);
```

El código fuente completo resultante de esta implementación se encuentra disponible en el archivo alu.v adjunto.

III. Implementación en Verilog: Modulo FPU

A. Encoding

Para el diseño del Floating Point Unit (FPU) se plantea le siguiente encoding de 2bits para las 4 operaciones que debe soportar:

Instrucción	Enconding	Flags
add.fp16	00	NZCV
mul.fp16	01	NZ
add.fp32	10	NZCV
mul.fp32	11	NZ

Las principales consideraciones utilizadas para el planteamiento de este encoding son dos:

- El bit mas significativo representa las dimensiones de la operación (16 bits o 32 bits).
- El bit menos significativo representa la operación a realizar (ADD=0 y MUL=1).

B. Submodulos parametrizados

Para la implementación de las operaciones de ADD y MUL con floating point en si se diseñaron submodulos parametrizados fp_add y fp_mult, instanciados dos veces en el FPU, una para 16bits y otra para 32 bits.

1) Floating Point Addition (fp_add): Se implemento el siguiente algoritmo en forma de circuito:

- Ajuste de mantisas:
  La mantisa correspondiente al exponente mas pequeño se ajusta por la diferencia entre los exponentes mediante un Logical Shift Right (LSR).

```
if ( ExponenteA > ExponenteB ) {
  DifExp = ExponenteA-ExponenteB
  MantisaB = LSR(MantisaB, DifExp)
  ExponenteMayor = ExponenteA
} else {
  DifExp = ExponenteB-ExponenteA
  MantisaA = LSR(MantisaA, DifExp)
  ExponenteMayor = ExponenteB
}
```

- Suma de mantisas:
  Se suman la mantisa en base a casos: Si son de signos distinto se guarda el valor absoluto de la diferencia de las mantisas y se usa el signo de la mantisa mayor. Si son del mismo signo, se suman las mantisas y se guarda el signo de cualquiera de las dos.

```
if (SignoA XOR SignoB){
  if (MantisaB > MantisaA){
    SignoRes = SignoB
    MantSuma = MantisaB - MantisaA
  } else {
    SignoRes = SignoA
    MantSuma = MantisaA - MantisaB
  }
} else {
  SignoRes = SignoA
  MantSuma = MantisaA + MantisaB
}
```

- Ajuste de mantisa y exponente (Normalización):
  Si existe carry se guarda el rango [MANTISA_WIDTH:1], se suma 1 al exponente y se activa la flag carry. Caso contrario, no se apaga la flag y el rango empleado es [MANTISA_WIDTH-1:0].

```
if (MantSuma[MANTISA_WIDTH+1]){
  carry = 1
  MantRes = MantSuma[MANTISA_WIDTH:1]
  ExponenteAdj = ExponenteMayor+1
} else {
  carry = 0
  MantRes=MantSuma[MANTISA_WIDTH-1:0]
  ExponenteAdj = ExponenteMayor
}
```

- Deteccion y Resolucion de overflow:
  El Overflow se detecta en dos casos: Si existe carry al resultado de la suma del exponente o si en el rango [EXPONENT_WIDTH-1:0] todos los valores son 1 (Si la mantisa no esta llena de 0's entonces seria un resultado NaN). En estos casos, para solucionar el overflow se hace set 1 a la flag de overflow y se fuerza el valor de la suma resultante a infinito (signo determinado previamente) estableciendo la mantisa como 0's y el exponentes como 1's.

```
if (
 ExponenteAdj[EXPONENT_WIDTH] = 1
 OR
 ExponentAdj[EXPONENT_WIDTH-1:0]
  = EXPONENT_WIDTH-1*{1}
) {
  overflow = 1
  ExponentRes = EXPONENT_WIDTH*{1}
  MantRes = MANTISA_WIDTH*{0}
} else {
  overflow = 0
  ExponenteRes
   = ExponenteAdj[EXPONENT_WIDTH-1:0]
}
```

El codigo resultante de la traducción de los pseudo-codigos presentados a verilog se encuentra implementada en el archivo fp_add.v.

2) Floating Point Multiplication (fp_mult): Se implemento el siguiente algoritmo en forma de circuito:

- Calculo de signo:
  El signo se calcula mediante una operacion XOR. Si son iguales entonces es + (=0). Si son distintos es - (=1).

```
SignoRes = SignoA XOR SignoB
```

- Calculo de exponente:
  El exponente se calcula como la suma de los exponentes restados por el BIAS

```
ExponenteRes
 = ExponenteA + ExponenteB - BIAS
```

- Calculo de mantisa:
  Para el calculo de la nueva mantisa primero se multiplican las mantisas extraídas y se guardan en MantisaMult. Los N/2 bits mas significativos son seleccionados de este resultado.

```
MantisaMult = MantisaA * mantisaB
MantisaParcial = MantisaMult[
  2*MANTISA_WIDTH+1:MANTISA_WIDTH+1
]
```

- Normalización:

```
if (MantisaParcial[MANTISA_WIDTH]==1) {
  MantisaRes = {
    MantisaParcial[MANTISA_WIDTH-2:0],
    MantisaMult[size]
  }
  ExponenteRes = ExponenteRes + 1
}  else {
  MantisaRes = MantisaParcial[
    MANTISA_WIDTH-1:0
  ]
}
```

- Edge-case de ZERO:
  Existe un edge case que es la multiplicación por zero. En este caso se fuerza el resultado a 0.

```
if (
  (ExponenteA = 0 && MantisaA = 0)
  ||
  (ExponenteB = 0 && MantisaB = 0)
) {
  ExponenteRes = EXPONENT_WIDTH*{0}
  MantisaRes = MANTISA_WIDTH*{0}
}
```

El codigo resultante de la traducción de los pseudo-codigos presentados a verilog se encuentra implementada en el archivo fp_add.v.

C. Instanciación de submodulos

Habiendo completado las implementación de los submódulos fp_add y fp_mult, estos se instancian dentro del FPU de la siguiente forma.

Listing 4. FPU: Instanciación de submodulos parametrizados en fpu.v

```
// Single precision (32 bits)
wire [31:0]single_add;
wire [3:0]single_add_flags;
add_fp #(
  .MANTISA_WIDTH(23), .EXPONENT_WIDTH(8)
) single_fp_adder(
  .a(a), .b(b), .res_add(single_add),
  .flags_add(single_add_flags)
);

wire [31:0]single_mult;
wire [3:0]single_mult_flags;
mult_fp #(
  .MANTISA_WIDTH(23), .EXPONENT_WIDTH(8)
) single_fp_mult(
  .a(a), .b(b), .res_mult(single_mult),
  .flags_mult(single_mult_flags)
);

// Half precision (16 bits)
wire [15:0]half_add;
wire [3:0]half_add_flags;
add_fp #(
  .MANTISA_WIDTH(10), .EXPONENT_WIDTH(5)
) half_fp_adder(
  .a(a[15:0]), .b(b[15:0]),
  .res_add(half_add),
  .flags_add(half_add_flags)
);

wire [15:0]half_mult;
wire [3:0]half_mult_flags;
mult_fp #(
  .MANTISA_WIDTH(10), .EXPONENT_WIDTH(5)
) half_fp_mult(
  .a(a[15:0]), .b(b[15:0]),
  .res_mult(half_mult),
  .flags_mult(half_mult_flags)
);
```

D. Selección de Result y Flags

La selección del output se realiza entonces en base a casos definidos por el input FPUControl, donde el Result y FPUFlags deben igualres a los obtenidos por el submodulo correspondiente a la operación codificada por este input. La implementación de esta logica en verilog se presenta a continuación.

Listing 5. FPU: Selección de resultados en fpu.v

```
// Selector de result y flags
always @(*) begin
casex (FPUControl[1:0])
  2'b10: begin
    Result = single_add;
    FPUFlags = single_add_flags;
  end
  2'b11: begin
    Result = single_mult;
    FPUFlags = single_mult_flags;
  end
  2'b00: begin
    Result = {
      16'b0000000000000000,
      half_add
    };
    FPUFlags = half_add_flags;
  end
  2'b01: begin
    Result = {
      16'b0000000000000000,
      half_mult
    };
    FPUFlags = half_mult_flags;
  end
endcase
end
```

El código completo resultante de esta implementación se encuentra disponible en el archivo fpu.v adjunto.

IV. Implementación en Verilog: Integración en Procesadores

A. DPUS

Para simplificar el proceso de adicion del nuevo ALU y FPU en los procesadores que se tiene se decidió diseñar un modulo unificador de estas unidades de procesamiento. Este modulo lo hemos llamado DPUS (Data Processor Unit Selector) y su tabla de encoding es la siguiente.

Instrucción	Enconding	Flags
sum	0000	NZCV
sub	0001	NZ
and	0010	NZ
or	0011	NZ
mul	0100	NZ
umull	0101	NZ
smull	0111	NZ
add.fp16	1000	NZCV
mul.fp16	1001	NZ
add.fp32	1010	NZCV
mul.fp32	1011	NZ

Cabe destacar que se tomaron las siguientes consideraciones para plantear este encoding:

- El bit mas significativo define si el output del DPUS corresponde al output del ALU (0) o al del FPU (1)
- Los bits [3:0] corresponden al control de ALU y los bits [2:0] corresponde al control del FPU.

Con estas consideraciones es posible implementar este modulo mediante el instanciamiento de un ALU y un FPU donde se conectan los inputs recibidos directamente a estas unidades y de los resultados obtenidos de ambas unidades de procesamiento se elige cual se va a usar mediante un if con el bit mas significativo del input DPUSControl. De esta forma, el funcionamiento del nuevo modulo creado es análogo al que actualmente tiene el ALU, permitiendo remplazarlo en la micro arquitectura sin necesidad de realizar mucha modificaciones.

El código resultante completo se encuentra disponible en el archivo dpus.v.

B. Procesador SingleCycle

En esta sección se explica como se integraron los cambios realizados en el ALU y el nuevo modulo FPU al procesador SingleCycle.

Como se explico en la seccion de diseño del DPUS, para simplificar el proceso de la integración de los cambios realizados, esto se realizaran mediante el remplazo del ALU por el DPUS en el procesador. El esquema de los cambios realizados al procesador para poder soportar este intercambio de módulos se presenta a continuación.

Al revisar este esquema se pueden encontrar tres grupos de cambios principales.

1) ***CONTROLLER****:* En primer lugar, el controller sufre modificaciones en la producción de cuatro señales: redimensionamiento de ALUControl, adición de dos nuevos outputs

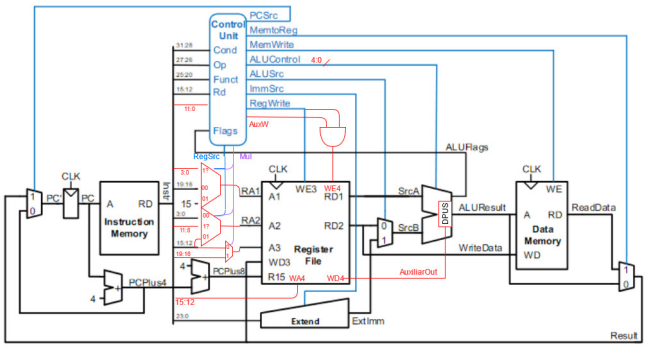

Fig. 1. Nuevo Esquema Single Cycle

(AuxW y Mul) y adición de un nuevo input (Src2 = Instr[11:0]).

El primer cambio se produce debido a que el modulo DPUS que ahora remplaza el ALU tiene un mayor numero de instrucciones. Por ello es necesario extender y ajustar los casos actuales para poder incluir el decoding de las nuevas instrucciones añadidas al ALU y las correspondientes al FPU. Cabe destacar que se agrego un bloque case intermediario que usa el nuevo input Src2 para distinguir el decoding de las operaciones de multiplicación del resto de operaciones al tener estas un encoding particular.

El segundo cambio se debe a que, con la adición de las operaciones de multiplicación, ahora es necesario considerar que todas estas requieren de un esquema de inputs distinto a los permitidos por los mux2 actuales, para lo que se usa Mul para el control de nuevos mux3, y algunas de estas (SMULL y UMULL) requieren escribir en un segundo registro al ejecutarse, lo cual se ve controlado por AuxW que esta conectado a un nuevo puerto en regfile mediante un AND con RegWrite. Para estas nuevas señales se define la siguiente tabla de nuestro main Decoder:

TABLA DE DECODER				
Input			**Output**	
Op	Src[7:4]	Funct[3:1]	Mul	AuxW
00	1001	0xx	1	0
00	1001	100	1	1
00	1001	110	1	1
00	1001	1x1	1	0
00	0xxx	xxx	0	0
00	x1xx	xxx	0	0
00	xx1x	xxx	0	0
00	xxx0	xxx	0	0
1x	xxxx	xxx	0	0
01	xxxx	xxx	0	0

Fig. 2. Decoder Table

Adicionalmente, para el enable del write de las flags CV es necesario cambiar la lógica para incluir también los casos de ADD16 y ADD32.

A continuación se muestra la sección principal de la implementación de estos cambios en decode.v.

Listing 6. Single-Cycle: Bloque always principal de decode.v

```
if (DPUSOp) begin
  // ADDED: Updated cases
  //for DPUS and added MUL's detection
  case (Src2[7:4])
    4'b1001:
    begin
    // MUL's
    Mul = 1'b1;
    case (Funct[3:1])
      // Only UMULL and SMULL
      // have AuxW on
      3'b000: begin
      DPUSControl = 4'b0100;
      AuxW = 1'b0; end // MUL
      3'b100: begin
      DPUSControl = 4'b0101;
      AuxW = 1'b1;
      end// UMULL
      3'b110: begin
      DPUSControl = 4'b0111;
      AuxW = 1'b1; end// SMULL
      3'b101: begin
      DPUSControl = 4'b1011;
      AuxW = 1'b0; end// MULL32
      3'b111: begin
      DPUSControl = 4'b1001;
      AuxW = 1'b0; end// MULL16
      default: begin
      DPUSControl = 4'bxxxx;
      AuxW = 1'b0; end//Default
    endcase
    end
    default:
    begin
    // Standard
    Mul = 1'b0;
    // Only MULs might
    // use auxiliary output
    AuxW = 1'b0;
    case (Funct[4:1])
      4'b0100: // ADD
      DPUSControl = 4'b0000;
      4'b0010: // SUB
      DPUSControl = 4'b0001;
      4'b0000:// AND
      DPUSControl = 4'b0010;
      4'b1100: // ORR
      DPUSControl = 4'b0011;
      4'b1010:// ADD32
      DPUSControl = 4'b1010;
      4'b1011: // ADD16
      DPUSControl = 4'b1000;
      default: // Default
      DPUSControl = 4'bxxxx;
    endcase
    end
  endcase
  // FlagW[1]: NZ should be saved
  // FlagW[1]: CV should be saved
  // Siempre qe S este activo
  FlagW[1] = Funct[0];

  // ADDED: Updated FlagW to
  //include also ADD16 and ADD32
  FlagW[0] = Funct[0] & (
    (DPUSControl == 4'b0000)
    |
    (DPUSControl == 4'b0001)
    |
    (DPUSControl == 4'b1000)
    |
    (DPUSControl == 4'b1010)
  );
  // Solo para ADD16, ADD32,
  // ADD y SUBB
end
else begin
  DPUSControl = 4'b0000;
  FlagW = 2'b00; // No se escribe
end
```

*2) **MUX REGISTER ADDRESS**:* El segundo grupo principal de modificaciones se ve en los mux de selección de los addresses de los registros de lectura y escritura (RA1, RA2 y WA3). Con las modificaciones realizadas al ALU y la adición del FPU ahora existen operaciones de multiplicación. Estas operaciones extraen los addresses de los registros utilizados de una sección distinta de la instrucción a las dos opciones actualmente disponibles en el procesador. Para implementar estos nuevos casos se remplazaron los mux2 de selección de registros de lectura por mux3 donde la segunda entrada selectora (Mul) fuerza las direcciones seleccionadas a las requeridas por una operaciones de multiplicación y se agrego un mux2 para la selección del registro de escritura WA3. Por ende, las tablas que define este nuevo comportarmiento en el datapath son las siguientes.

Tabla de MUX RA1		
Inputs		Outputs
Mul	RegSrc[0]	Selected RA1
1	x	Ins[3:0]
0	0	Ins[19:16]
0	1	R15

Fig. 3. Mux RA1 Table

Estos cambios se ven implementados en el archivo datapath.v. A continuación se presenta el fragmento de código principal que refleja estos cambios.

Listing 7. Single-Cycle: Muxes de selección de direcciones de registro en datapath.v

```
// ADDED: Modificacion de Register
// Adress a mux3 para incluir
// logica de Mul
```

Tabla de MUX RA2		
Inputs		Outputs
Mul	RegSrc[1]	Selected RA2
1	x	Ins[11:8]
0	0	Ins[3:0]
0	1	Ins[15:12]

Fig. 4. Mux RA2 Table

Tabla de MUX A3	
Inputs	Outputs
Mul	Selected A3
1	Ins[19:16]
0	Ins[15:12]

Fig. 5. Mux A3 Table

```
// Seleccion de RegisterAdrress1
mux3 #(4) ra1mux(
        .d0(Instr[19:16]), // Rn (Src1)
        .d1(4'b1111), // PC
        .d2(Instr[3:0]), // Rn (Mul)
        .s({Mul, RegSrc[0]}),
        .y(RA1)
);

// Seleccion de RegisterAddress2
mux3 #(4) ra2mux(
        .d0(Instr[3:0]), // Rm (Src2)
        .d1(Instr[15:12]), // Rd (STR)
        .d2(Instr[11:8]), // Rm (Mul)
        .s({Mul,RegSrc[1]}),
        .y(RA2)
);

// ADDED: Logica para seleccion de
// output de escritura para incluir
// logica de Mul
mux2 #(4) wa3mux(
        .d0(Instr[15:12]), // Rd (Src2)
        .d1(Instr[19:16]), // Rd (Mul)
        .s(Mul),
        .y(WA3)
);
```

*3) **REGISTER FILE**:* El ultimo grupo de cambios corresponde al register file. A este se le ha añadido tres nuevos inputs para la escritura paralela en un cuarto register address. Estos inputs son su address (WA4), el cual se conecta directamente de un fragmento de la instrucción, el data que se desea escribir (WD4), conectado al output Auxiliar producido por el DPUS, y el wirte enable (WE4), conectado a RegWrite & AuxW.

Para implementar la estructura paralela de dos registros se tuvieron que realizar ligeros cambios al archivo regfile.v, que se muestran continuación.

Listing 8. Single-Cycle: Escritura paralela de registros en regfile.v

```
always @(posedge clk)
        if (we3 & we4) begin
                rf[wa3] <= wd3;
                rf[wa4] <= wd4;
        end
        else if (we3) begin
                rf[wa3] <= wd3;
        end
        else if (we4) begin
                rf[wa4] <= wd4;
        end
```

Con las modificaciones realizadas se obtiene un procesador single cycle funcional mediante el cual se pueden emplear el ALU extendido y el modulo FPU implementados en este proyecto. El codigo fuente completo se encuentra disponible en la carpeta ”single” adjunta.

C. Procesador MultiCycle:

En esta sección se integraron los cambios realizados en el ALU y el nuevo modulo FPU SingleCycle.

El esquema de los cambios realizados al procesador para poder soportar este intercambio de módulos se presenta a continuación.

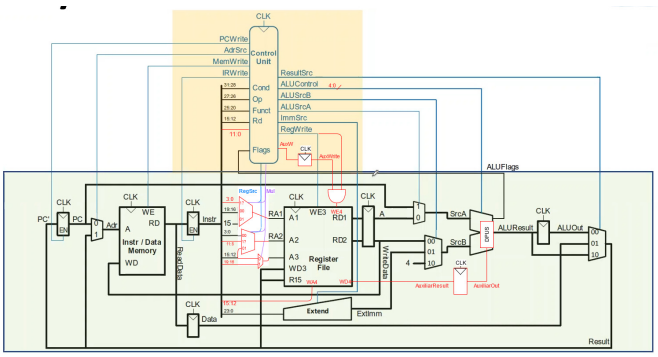

Fig. 6. Nuevo Esquema Multi Cycle

Al revisar el esquema se puede identificar que los cambios realizados al procesador MultiCycle para poder soportar el nuevo modulo unificador DPUS son similares a los realizados en el SingleCycle. En este sentido, los cambios en el regfile y los muxes son exactamente los mismos. No obstante, son necesarias dos consideraciones adicionales.

*1) **CONTROLLER**:* Si bien las modificaciones de puertos realizada sobre el controller del MultiCycle son las mismas que las realizadas sobre el SingleCycle, el método mediante el cual se calcula la señal Mul difiere.

En el procesador MultiCycle DPUSOp se encuentra activo solo durante las fases de ExecuteI o ExecuteR, mientras que la lectura de registros se realiza en la fase anterior. No obstante, la lógica de calculo de la señal Mul (que define que registros se van a leer) se enceuntra dentro de un bloque IF(DPUSOp) Esta consideración implica que la lógica de la señal de Mul debe ser extraída fuera del este bloque en el que se encontraba actualmente para una correcta lectura de registros.

Para solucionar este problema se ha trasladado esta lógica fuera del bloque de decoding de ALUControl al bloque correspondiente al decoding del RegSrc. A continuación, se muestra el fragmento de código que refleja esta nueva lógica de Mul.

Listing 9. Multi-Cycle: Aislamiento de logica de señal Mul en decode.v

```
// RegSrc Logic
// ADDED: Mul logic extracted
// from previos block
always @(*)
casex (Op)
  // Data processing instructions
  2'b00:
    begin
    RegSrc = 2'b00;
    if (Src2[7:4] == 4'b1001)
      Mul = 1'b1;
    else
      Mul = 1'b0;
    end
  // Memory instructions
  2'b01: begin
    RegSrc = 2'b10;
    Mul=1'b0;
  end
  // Branch instructions
  2'b10: begin
    RegSrc = 2'bx1;
    Mul=1'b0;
  end
  default: begin
    RegSrc = 2'bxx;
    Mul=1'bx;
  end
endcase
```

*2) **DELAY DE AUXILIAR**:* En lo que respecta a la conexión del output AuxiliarOut obtenido de DPUS y el input WD4 del register file, esta no puede hacerse directamente debido a que la señal RegWrite aun no esta habilitada durante el ciclo en el que en el que se obtiene este output (Estado: ExecuteR) y en cambio se estaría escribiendo otros datos el siguiente ciclo (Estado: ALUWB). Asimismo, el mismo problema aplica con la señal AuxW que es actualmente recibida durante el estado ExecuteR en vez de ALUWB. Para solucionar esto se propone entonces agregar una instancia de flopr en el medio de estas conexiones para hacer delay a estas señales de forma que así lleguen al register file en el ciclo correcto. A continuación, se muestra el fragmento de código correspondiente a la implementación de este bloques adicionales en el datapath.

Listing 10. Multi-Cycle: Delay de AuxW y AuxiliarOut en decode.v

```
// ADDED: Delay de AuxW y AuxiliarOut
wire AuxWrite;
wire [31:0] AuxiliarResult;
flopr #(1) auxwdelay(
        .clk(clk),
        .reset(reset),
        .d(AuxW),
        .q(AuxWrite)
);
flopr #(32) auxoutdelay(
        .clk(clk),
        .reset(reset),
        .d(AuxiliarOut),
        .q(AuxiliarResult)
);
```

Con estas adaptaciones de los cambios realizados en el SingleCycle para el MultiCycle, obtenemos ahora un procesador MultiCycle funcional que permite la utilización de los modulos FPU y ALU implementados a lo largo de este proyecto. El codigo fuente completo se encuentra disponible en la carpeta ”multi” adjunta.

V. Test Case A: FPU

En esta sección se dará un breve contexto sobre el programa creado en assembly y posteriormente incorporado en nuestro modulo de test bench en Verilog, para luego ser probado en nuestro FPU del Single Cycle y Muli Cycle.

A. Consideraciones

Antes de presentar el programa en codigo Assembly, tenemos que considerar que el encoding del ”cmd”, para las operaciones ADD y MUL de los test case del Floating Point, es diferente a las habituales, esto debido a que se realizaran operaciones para valores en notacion de ”Floating Point”. Tomando en consideracion lo previamente mencionado, para poder darle un cmd unico a estas operaciones, se utilizaran las cmd de otras operaciones que no soportan nuestros procesadores. A continuación, se mostrara la tabla de los nuevos encodings para operaciones con ”Floating Point”

Instruccion	Encoding de CMD Nuevo	Sobreescrituras
ADDFP32	1010	Sobreescribe Compare
MULFP32	101	Sobreescribe UMLAL
ADDFP16	1011	Sobreescribe Compare Negative
MULFP16	111	Sobreescribe SMLAL

Fig. 7. Tabla Nuevos Encoding

B. Floating Point Test Case

El programa que se tuvo en consideración para poder realizar los test case respectivos, de tal manera que se utilicen las operaciones de ADD y MUL para Floating Point, se basa en realizar un ”Producto punto” entre dos vectores o arreglos. Recordemos, que este tipo de operaciones en vectores contienen multiplicación y suma, por lo que, es el caso perfecto en donde podremos aprovechar ambas operaciones para evidenciar la eficiencia de FPU. A continuación, se mostrara el código Assembly de nuestro programa.

Cabe destacar que en este código se encuentra comentado como se podría inicializar los arreglos 1 y 2 sobre los cuales se va a hacer el producto en un procesador con arquitectura ARM. Sin embargo, por simplicidad del programa y del testing, al traducir del programa y ejecutarlo se asumirá que esta información se encuentra ya en memoria.

Listing 11. prodpunto.asm

```
MAIN:
   //Registro para iniciar valores
   SUB R0,R15,R15
   // NOTE: Se evita usar el MOV
   //mediante operacion 0+Immediate
   // Memoria base de arreglo 1
   ADD R1,R0,#0x070 // base1 = 112
   // Memoria base de arreglo 2
   ADD R2,R0,#0x080 // base2 = 128
   // Direccion final de resultado
   ADD R7,R0,#0xA0
   // Tama o de arreglos
   ADD R4,R0,#3 //size=3

   // Ejemplo de inicializacion de valores
   //LDR R8,=0x4096ecc0 //4.7164
   //LDR R9,=0x404ccccd // 3.2
   //LDR R12,=0x00000000 //0

   //LDR R10,=0x40b00000 //5.5
   //LDR R11,=0x40066666 //2.1
   //LDR R7,=0x41073333 // 8.45

   // //arreglo1 = [4.7164, 3.2, 0]
   //STR R8, [R1]
   //STR R9, [R1,#4]
   //STR R12, [R1,#8]

   // //arreglo2 = [5.5, 2.1, 8.45]
   //STR R10, [R2]
   //STR R11, [R2,#4]
   //STR R7,[R2,#8]
   B PROD_PUNTO

// PROD PUNTO: Halla el producto punto
// de dos arreglos de tama o size
// R1 = direcci n de memoria base
// para arreglo1
// R2 = direcci n de memoria base
// para arreglo1
// R3 = Tama o de los arreglos (size)
PROD_PUNTO:
   ADD R3,R0,#0//iterador=0
//Loop para recorrer los arreglos
//Y realizar las operaciones necesarias
LOOP:
   SUBS R6,R4,R3 // (iterador == 3)?
   BEQ END // IF (iterador == 3) => END
   // A1 = arreglo1[BASE1+4*iterador]
   LDR R5,[R1]
   // A2 = arreglo2[BASE2+4*iterador]
   LDR R6,[R2]
   MUL32 R12,R5,R6 // TEMP= A1 * A2
   ADD32 R0,R0,R12 // RESULT=RESULT + TEMP
   ADD R1,R1,#4 // BASE1 + 4
   ADD R2,R2,#4 // BASE2 + 4
   ADD R3,R3,#1 // ITERADOR ++
   B LOOP // REPEAT LOOP
END:
   // Almacenar respuesta en memoria
   STR R0, [R7, #0]
```

Ahora, se mostrara la tabla con los respectivos encodings para cada instrucción del programa.

Instrucciones	Encoding
SUB R0, R15, R15	1110_00_0_0010_0_1111_0000_0000_0000_1111
ADD R1, R0, #0x070	1110-00-1-0100-0-0000-0001-0000-01110000
ADD R2, R0, #0x080	1110-00-1-0100-0-0000-0010-0000-10000000
ADD R7, R0, #0xA0	1110-00-1-0100-0-0000-0111-0000-10100000
ADD R4, R0, #3	1110-00-1-0100-0-0000-0100-0000-00000011
ADD R3, R0, #0	1110-00-1-0100-0-0000-0011-0000-00000000
B PROD_PUNTO	1110-10-10-11111111111111111111111111
SUBS R6,R4,R3	1110_00_0_0010_1_0100_1000_00000_00_0_0011
BEQ END	0000-10-10-00000000000000000000000111
LDR R5,[R1]	1110-01-011001-0001-0101-000000000000
LDR R6,[R2]	1110-01-011001-0010-0110-000000000000
MULFP32 R12,R5,R6	1110-00-00-101-0-1100-0000-0110-1001-0101
ADDFP32 R0,R0,R12	1110-00-0-1010-0-0000-0000-00000-00-0-1100
ADD R1,R1,#4	1110-00-1-0100-0-0001-0001-0000-00000100
ADD R2,R2,#4	1110-00-1-0100-0-0010-0010-0000-00000100
ADD R3,R3,#1	1110-00-1-0100-0-0011-0011-0000-00000001
B LOOP	1110-10-10-11111111111111111110101
STR R0, [R7, #0]	1110-01-011000-0111-0000-000000000000

Fig. 8. Encoding Programa "Producto Punto"

C. Computo Manual

En este apartado, se brindara un procedimiento paso a paso de como se ejecutara este programa, utilizando Floating Point, hasta llegar a la respuesta correcta.

Como mencionó anteriormente, para el producto punto hace falta dos vectores/arreglos con dimensiones idénticas para poder hacer un producto punto de manera correcta. Por esta razón, se inicializa en los registros R1 y R2 las posiciones base en memoria donde se encontrarán los primeros valores de los vectores/arreglos. Estas posiciones de memoria estan lo sufucientemente espaciadas para poder almacenar Se optó por tener dos vectores de 3 números, siendo estos: [4.7164, 3.2, 0] y [5.5, 2.1, 8.45].

Para calcular el producto punto se requiere la suma de la multiplicación de cada par respectivo de números en los vectores. Para esto se cargan valores de memoria a los registros R5 y R6 utilizando los registros con las posiciones de memoria base. Luego, estos dos números se multiplican y se añaden al registro 0, donde se almacenará la respuesta final. Las 3 multiplicaciones que se harán son las siguientes:

- $4.7164 \times 5.5 = 25.9402$
- $3.2 \times 2.1 = 6.72$
- $0 \times 8.45 = 0$

En el código cada vez que se hace una multiplicación, el resultado es agregado al registro 0 que esta inicializado con un valor de 0. De esta manera, el restulado final calculado es de 32.6602. La representación hexadecimal correspondiente de este numero en single precision es $0x4202a40b4$ y $01000010000000101010010000001011$ en binario.

D. TestBench 0.A: Test sobre fpu.v

En este apartado se implementa el test inicial que nos permite confirmar el correcto funcionamiento de nuestro FPU antes de probarlo dentro de los procesadores.

Para esto, se ha escrito en el archivo fpu_tv los inputs y outputs esperados del modulo fpu por cada una de las operaciones ADD32 y MUL32 que se espera que realice durante la ejecución de este primer programa.

A continuación se mostrara el código que representa el modulo implementado para hacer el testing utilizando el test vector previamente mencionado.

Listing 12. fpu_tb.v

```
module fpu_tb();
  reg  [31:0] a,b;
  wire [3:0] FPUFlags;
  wire [31:0] Result;
  reg  [1:0] FPUControl;
  reg  clk, reset;

  reg [101:0] testvector[15:0];
  reg [31:0] Result_expected;
  reg [3:0] FPUFlags_expected;
  reg [31:0] vectornum;
  reg [31:0] errors;

 FPU fpu_dut (
   .a(a), .b(b),
   .Result(Result),
   .FPUFlags(FPUFlags),
   .FPUControl(FPUControl)
 );

  always
    begin
     clk=1; #5; clk=0; #5;
    end

  initial
    begin
      $readmemh(
      "FPU/fpu_tv.tv",
      testvector);
      errors=0;
      vectornum=0;
      reset =1; #27; reset = 0;
    end

  always @(posedge clk)
    begin
     FPUFlags_expected=
     testvector[vectornum][3:0];
     Result_expected=
     testvector[vectornum][35:4];
     a=testvector[vectornum][67:36];
     b=testvector[vectornum][99:68];
     FPUControl=
     testvector[vectornum][101:100];
    end

 always @(negedge clk)
  begin
   if (~reset)
    begin
    // ==, ===
    if (
      (Result!==Result_expected)
      ||
      (FPUFlags!==FPUFlags_expected)
    )
      begin
      //$multiple displays with outputs,
      //inputs and expected values
      $display("Vectornum=%d",vectornum);
      $display(
        "For: a=%h",a,
        " b=%h",b,
        "Control=%b",FPUControl);
      $display(
        "Got: Res=%h", Result,
        "Flags=%b",FPUFlags);
      $display(
        "Exp: Res=%h",Result_expected,
        "Flags=%b",FPUFlags_expected,
        "\n");
      errors = errors+1;
      end
      vectornum=vectornum+1;
       if (
         testvector[vectornum][0]===1'bx
       )
        begin
         $display("errors :%d",errors);
         $dumpfile("test.vcd");
         $dumpvars();
         $finish;
         end
       end
    end
endmodule
```

Este test a sido incluido de forma adicional dentro de la carpeta "extra/fp".

E. TestBench 1.A.1 y 2.A: Test de FPU sobre procesadores

En este apartado se implementa el Test 1.A y Test 2.A que nos permite confirmar la correcta integración del FPU a nuestros procesadores SingleCycle y MultiCycle respectivamente.

Para poder corroborar que el FPU funciona correctamente dentro de los procesadores se ejecutara el programa previamente explicado en cada uno de estos procesadores. Para ello, su codificación en binario se encuentra disponible en el archivo memfile_fp.dat. Cabe destacar que a este archivo también se le han agregado los valores de los arreglos en su posición correspondiente, rellenando los intermedios con don't cares (X). De esta forma es posible cargar este programa a memoria habilitando la linea:

```
initial $readmemb("memfile_fp.dat", RAM);
```

en imem.v y dmem.v para el procesador SingleCycle y en mem.v para el procesador MultiCycle.

Al finalizar de ejecutar el programa debería de ejecutar un STR en la dirección 0xA0 (160) con el valor de producto punto computado. Es esta la instrucción que se captura el testbench mediante un if con el DataAdr y revisa si el WriteData concuerda con el valor computado manualmente ($32.6602 = 0x4202a40b4$).

La implementación de este testbench en verilog se presenta a continuación.

Listing 13. testbench_fp.v

```
module testbench_fp;
  reg clk;
  reg reset;
  wire [31:0] WriteData;
  wire [31:0] DataAdr;
  wire MemWrite;
  top dut(
    .clk(clk),
    .reset(reset),
    .WriteData(WriteData),
    // Para MultiCycle remplazar
    // .DataAdr por .Adr
    .DataAdr(DataAdr),
    .MemWrite(MemWrite)
  );
  initial begin
    reset <= 1;
    #(22);
    reset <= 0;
  end
  always begin
    clk <= 1;
    #(5);
    clk <= 0;
    #(5);
  end
  always @(negedge clk)
    if (MemWrite)
      if (DataAdr === 160) begin
        $display("wd=%h", WriteData);
          if (WriteData == 32'h4202a40b)
          begin
            $display(
            "Simulation succeeded"
            );
            $stop;
          end
          else begin
            $display("Simulation failed");
            $stop;
          end
      end
  initial begin
    $dumpfile("arm_single_fp.vcd");
    $dumpvars();
  end
endmodule
```

VI. Test Case B: ALU

En esta sección se explicara el programa que se utilizo a manera de testeo para nuestro ALU extendido. Este programa en assembly contiene las operaciones MUL, SMULL y UMULL tal y como se indico al inicio de nuestro trabajo.

A. Extended ALU Test Case

Nuestro programa, construido en lenguaje Assembly, tiene como finalidad multiplicar todos los nodos derechos de la estructura de datos "Heap". Recordemos que para poder encontrar el nodo derecho de un "Heap" se tiene la siguiente formula:

- Indice del Nodo derecho = Indice * 2 + 1

Donde "Indice" es el indice del padre representado en el Array. En adición, tomando como referencia esta formula, se construyo el siguiente programa.

Cabe destacar que este programa asume que el heap se encuentra ya en memoria. Asimismo, por fines de testeo, el 1 inicial del multiplicatorio se esta inicializando mediante un load de memoria que asume que la posición 0xA0 es 1.

Listing 14. multrightheap.asm

```
MAIN:
  //Registro para iniciar valores
  SUB R5,R15,R15
  // NOTE: Se evita usar el MOV
  // mediante operacion 0+Immediate
  ADD R0,R5,#28 // NWord=28
  ADD R1,R5,#3 // n=3,Profundidad del heap
  // Direccion final de resultado
  ADD R10,R5,#0xA0
  B MULT_RIGHT_HEAP

//MULT RIGHT HEAP: Halla el
// multiplicatorio de todos los
// elementos derecho de un un heap
//R0 = Numero de word
//R1 = Nieveles de profundidad del heap
MULT_RIGHT_HEAP:
  // Constante utilizada para cambiar el offset
  ADD R7,R5,#2 // Constante2  = 2
  // Inicializar el registro de respuesta
  LDR R2,[R10] //Total = 1
  // Inicializar variable de loop
  ADD R3,R5,#1 // i = 1 iterador
  // Inicializar Offset
  ADD R4,R5,#8 // Constante8 = 8
  // Valor temporal
  ADD R5,R5,#4 // temp = 4
  // Transformar numero de word
  // proporcionado a direccion de memoria
  UMULL R0, R0, R5, R9 // Dir = NWord * 4
// Loop para recorrer los nodos
// derechos del heap
```

```
LOOP:
   SUBS R9, R1, R3 // alternativa de CMP
   BLT END // IF (n < iterador) => END
   LDR R5, [R0] // temp = Load(Mem)
   // Total = Total x temp
   SMULL R2, R2, R5, R9
   ADD R0, R0, R4 // Mem = Mem + offset
   MUL R4, R4, R7 // offest = offset * 2
   ADD R3, R3, #1 //i++
   B LOOP
END:
   // Almacenar respuesta en memoria
   STR R2,[R10,#0]
// NOTE: R9 = Garbage for multiplication
//and comparison
```

A continuación, se mostrara una tabla en donde se podrá evidenciar el encoding de las instrucciones de nuestro programa, esta incluye dichas instrucciones que utilizan las operaciones de SMULL, UMULL Y MUL que soportara nuestro ALU extendido.

Instrucciones	Encoding
SUB R5, R15, R15	1110_00_0_0010_0_1111_0101_0000_0000_1111
ADD R0,R5,#28	1110-00-1-0100-0-0101-0000-0000-00011100
ADD R1,R5,#3	1110-00-1-0100-0-0101-0001-0000-00000011
ADD R10, R5, #0xA0	1110-00-1-0100-0-0101-1010-0000-10100000
B MULTI_RIGHT_HEAP	1110-10-10-11111111111111111111111111
ADD R7,R5,#2	1110-00-1-0100-0-0101-0111-0000-00000010
LDR R2, [R10]	1110-01-011001-1010-0010-000000000000
ADD R3,R5,#1	1110-00-1-0100-0-0101-0011-0000-00000001
ADD R4,R5,#8	1110-00-1-0100-0-0101-0100-0000-00001000
ADD R5,R5,#4	1110-00-1-0100-0-0101-0101-0000-00000100
UMULL R0, R0, R5, R9	1110-00-00-100-0-0000-1001-0101-1001-0000
SUBS R9, R1, R3	1110-00-0-0010-1-0001-1001-00000-00-0-0011
BLT END	1011-10-10-00000000000000000000000101
LDR R5, [R0]	1110-01-011001-0000-0101-000000000000
SMULL R2, R2, R5, R9	1110-00-00-110-0-0010-1001-0101-1001-0010
ADD R0, R0, R4	1110-00-0-0100-0-0000-0000-00000-00-0-0100
MUL R4, R4, R7	1110-00-00-000-0-0100-0000-0111-1001-0100
ADD R3, R3, #1	1110-00-1-0100-0-0011-0011-0000-00000001
B LOOP	1110-10-10-11111111111111111110111
STR R2, [R10, #0]	1110-01-011000-1010-0010-000000000000

Fig. 9. Encoding Programa "Multiplicacion de nodos derechos de un heap"

B. Computo Manual

En esta parte se procederá a hacer el computo manual del programa. Como fue mencionado anteriormente, el propósito del programa es calcular la multiplicación de todos los hijos derechos de un heap. Los valores que se multiplicarán son: -8 (0xFFFFFFF8), 5 (0x00000005) y -1 (FFFFFFFF). Estos valores han sido especificados en el archivo de memoria previo a la ejecución. El valor de la multiplicación de estos números es 40. Esta multiplicación ha sido ejectuado utilizando SMULL debido a que se requiere trabajar con números con signo (positivo o negativo).

Por otra parte, se ha utilizado el MUL para calcular el siguiente indice del heap y así poder leerlo de memoria. La dirección de acceso a memoria esta dado en el registro R0, y por cada iteración del loop este valor es multiplicado por 2. Luego se le agrega uno para conseguir la posición de su hijo derecho.

Por último, el UMULL es utilizado para transformar el numero de word proporcionado a una dirección de memoria mediante una multiplicación x4.

C. TestBench 0.B: Test sobre alu.v

En este apartado se implementa el test inicial que nos permite confirmar el correcto funcionamiento de nuestro ALU extendido antes de probarlo dentro de los procesadores.

Para esto, se ha escrito en el archivo alu_tv los inputs y outputs esperados del modulo alu por cada una de las operaciones MUL, UMULL y SMULL que se espera que realice durante la ejecución de este primer programa.

A continuación se mostrara el código que representa el modulo implementado para hacer el testing utilizando el test vector previamente mencionado.

Listing 15. alu_tb.v

```
module alu_tb();
 reg [31:0] a,b;
 wire [3:0] ALUFlags;
 wire [31:0] Result;
 wire [31:0] Auxiliar;
 reg [2:0]ALUControl;
 reg clk, reset;

 reg [134:0] testvector[15:0];//103:0 - 10:0
 reg [31:0] Result_expected;
 reg [3:0] ALUFlags_expected;
 reg [31:0] Auxiliar_expected;
 reg [31:0] vectornum;
 reg [31:0] errors;

ALU alu_dut (
  .a(a), .b(b),
  .Result(Result),
  .Auxiliar(Auxiliar),
  .ALUFlags(ALUFlags),
  .ALUControl(ALUControl)
 );

 always
  begin
    clk=1; #5; clk=0; #5;
  end
 initial
  begin
   $readmemh("alu_tv.tv",testvector);
   errors=0;
   vectornum=0;
   reset =1; #27; reset = 0;
  end

 always @(posedge clk)
  begin
   ALUFlags_expected=
   testvector[vectornum][3:0];
   Result_expected=
   testvector[vectornum][35:4];
```

```
  Auxiliar_expected=
  testvector[vectornum][67:36];
  a=testvector[vectornum][99:68];
  b=testvector[vectornum][131:100];
  ALUControl=
  testvector[vectornum][134:132];
end

always @(negedge clk)
  begin
   if (~reset)
    begin
      if (
       (Result!==Result_expected)
       ||
       (ALUFlags!==ALUFlags_expected)
       ||
       (
       Auxiliar!==Auxiliar_expected
       &&
       Auxiliar_expected!==32'bx
       )
      )
       begin
        // $multiple displays with
        // outputs, inputs and
        // expected values
        $display(
        "Vectornum=%d",vectornum
        );

        $display(
          "For: a=%h",a,
          " b=%h",b,
          " Control=%b",ALUControl
        );
        $display(
          "Got: Res=%h", Result,
          " Aux=%h", Auxiliar,
          " Flags=%b",ALUFlags);
        $display(
          "Exp: Res=%h",
          Result_expected,
          " Aux=%h",
          Auxiliar_expected,
          " Flags=%b",
          ALUFlags_expected,
          "\n"
        );
        errors = errors+1;
       end
      vectornum=vectornum+1;
      if (
       testvector[vectornum][0]===1'bx
      )
       begin
        $display("errors :%d",errors);
        $finish;
       end
     end
   end
endmodule
```

Este test a sido incluido de forma adicional dentro de la carpeta "extra/alu".

D. TestBench 1.B y 2.B: Test de ALU extendido sobre procesadores

En este apartado se implementa el Test 1.B y Test 2.B que nos permite confirmar la correcta integración del FPU a nuestros procesadores SingleCycle y MultiCycle respectivamente.

Para poder corroborar que el ALU extendido funciona correctamente dentro de los procesadores se ejecutara el programa previamente explicado en cada uno de estos procesadores. Para ello, su codificación en binario se encuentra disponible en el archivo memfile_mul.dat. Cabe destacar que a este archivo también se le ha agregado los valores de los elementos derechos del heap en su posición correspondiente, rellenando los intermedios con don't cares (X). De esta forma es posible cargar este programa a memoria habilitando la linea:

```
initial $readmemb("memfile_mul.dat", RAM);
```

en imem.v y dmem.v para el procesador SingleCycle y en mem.v para el procesador MultiCycle.

Al finalizar de ejecutar el programa debería de ejecutar un STR en la dirección 0xA0 (160) con el valor computado correspondiente al multiplicatorio de todos los elementos derechos del heap . Es esta la instrucción que se captura el testbench mediante un if con el DataAdr y revisa si el WriteData concuerda con el valor computado manualmente ($40 = 0x00000028$).

La implementación de este testbench en verilog se presenta a continuación.

Listing 16. testbench_mul.v

```
module testbench_fp;
  reg clk;
  reg reset;
  wire [31:0] WriteData;
  wire [31:0] DataAdr;
  wire MemWrite;
  top dut(
    .clk(clk),
    .reset(reset),
    .WriteData(WriteData),
    // Para MultiCycle remplazar
    // .DataAdr por .Adr
    .DataAdr(DataAdr),
    .MemWrite(MemWrite)
  );
  initial begin
    reset <= 1;
    #(22);
    reset <= 0;
  end
  always begin
```

```
    clk <= 1;
    #(5);
    clk <= 0;
    #(5);
  end
  always @(negedge clk)
    if (MemWrite)
      if (DataAdr === 160) begin
        $display("wd=%h", WriteData);
          if (WriteData == 32'h4202a40b)
          begin
            $display(
            "Simulation succeeded"
            );
            $stop;
          end
          else begin
            $display("Simulation failed");
            $stop;
          end
      end
  initial begin
    $dumpfile("arm_single_mul.vcd");
    $dumpvars();
  end
endmodule
```

VII. Discusión de Resultados

A. Test 0: Testing de ALU extendido y FPU

En esta sección se ejecutan los tests preliminares para confirmar el correcto funcionamiento de los nuevos módulos ALU y FPU diseñados a lo largo de este proyecto.

1) FPU: Para corroborar que el FPU implementado funciona correctamente se ejecuta el archivo "fpu_tb.v", que simula todas las operaciones de floating point que necesitan realizar los procesadores para ejecutar el programa prodpunto.asm utilizando el testvector "fpu_tv.tv".

El resultado obtenido es el siguiente:

```
D:\Rodrigo\Rodrigo\UTEC\ciclo\2020-1\Arqui\proyecto\UPLOAD\Source\fpu>vvp a.out
WARNING: fpu_tb.v:24: $readmemh: The behaviour for reg[...] mem[N:0]; $readmemh(
t, stop);. Defaulting to 1364-2005 behavior.
WARNING: fpu_tb.v:24: $readmemh(fpu_tv.tv): Not enough words in the file for the
errors :        0
VCD info: dumpfile test.vcd opened for output.
fpu_tb.v:59: $finish called at 75 (1s)
```

Fig. 10. Resultado de Test 0.A

Como se ve en la imagen, el resultado obtenido indica que no han habido errores. Por lo tanto podemos decir que todas las operaciones de floating point que se necesitan realizar para ejecutar el primer programa funcionan correctamente en el FPU.

2) ALU: Para verificar el correcto funcionamiento del ALU se ejecuta el archivo "alu_tb.v", que simula todas las operaciones de multiplicación que necesitan realizar los procesadores para ejecutar el programa multrightheap.asm utilizando el testvector "alu_tv.tv".

```
D:\Rodrigo\Rodrigo\UTEC\ciclo\2020-1\Arqui\proyecto\UPLOAD\Source\alu>vvp a.out
WARNING: alu_tb.v:31: $readmemh: The behaviour for reg[...] mem[N:0]; $readmemh("
t, stop);. Defaulting to 1364-2005 behavior.
WARNING: alu_tb.v:31: $readmemh(alu_tv.tv): Not enough words in the file for the
errors :        0
alu_tb.v:66: $finish called at 95 (1s)
```

Fig. 11. Resultado de Test 0.B

El resultado obtenido es el siguiente:

Este resultado indica que no han habido errores y que todas las operaciones de multiplicación que se necesitan realizar para ejecutar el segundo programa funcionan correctamente en el ALU.

3) Conclusión: Habiendo tenido resultados positivos en los test preliminares del ALU extendido y el nuevo modulo FPU, podemos continuar con el testing de la implementación de estos en los procesadores con la seguridad de que no son la causa directa de los problemas que sucedieran en este proceso.

B. Test A: Testing de implementación en procesador SingleCycle

1) Test 1.A Test de integración de FPU: Para el test de integración del FPU se utilizo el programa prodpunto.asm, el cual es cargado a memoria descomentando la linea

```
initial $readmemb("memfile_mul.dat", RAM);
```

y comentando

```
initial $readmemb("memfile_fp.dat", RAM);
```

en los archivos imem.v y dmem.v. Para validar que se ejecutó correctamente el test es controlado mediante el modulo testbench_fp implementado en la sección previa.

Los resultados obtenidos de ejecutar el test se muestran en la Fig 12.

```
D:\Rodrigo\Rodrigo\UTEC\ciclo\2020-1\Arqui\proyecto\UPLOAD\Source\MultiCycle>vvp a.out
WARNING: mem.v:16: $readmemb: The behaviour for reg[...] mem[N:0]; $readmemb("...", mem
stop);. Defaulting to 1364-2005 behavior.
WARNING: mem.v:16: $readmemb(memfile_fp.dat): Not enough words in the file for the requ
VCD info: dumpfile arm_multi_fp.vcd opened for output.
wd=4202a40b
Simulation succeeded
testbench_fp.v:42: $stop called at 1195 (1s)
** VVP Stop(0) **
** Flushing output streams.
** Current simulation time is 1195 ticks.
> finish
** Continue **
```

Fig. 12. Resultado de Test 1.A

Como podemos ver en el output de consola la ejecución fue exitosa retornando como valor final el resultado esperado 0x4202a40b.

Asimismo, el waveform producido por el testbench se encuentra almacenado en el archivo de salida arm_single_fp.vcd. A continuación se muestra gráficamente el waveform considerando las señales más importantes (clk, reset, PC, Instr, ALUResult, AuxiliarOut, Mul, AuxW, WriteData, MemWrite y ReadData) utilizando el programa GTKWave. Estas son visible en las Figuras 13-17

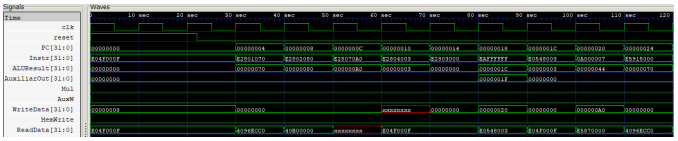

Fig. 13. Waveform Test 1.A parte 1

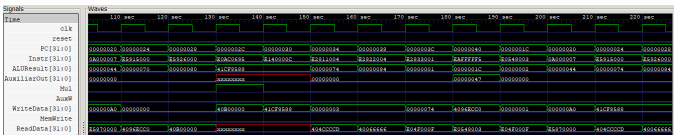

Fig. 14. Waveform Test 1.A parte 2

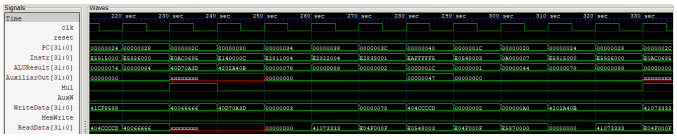

Fig. 15. Waveform Test 1.A parte 3

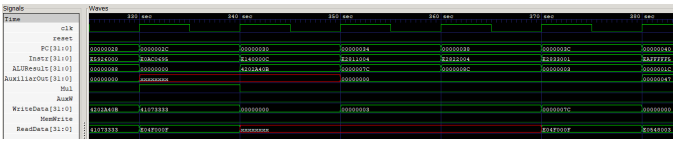

Fig. 16. Waveform Test 1.A parte 4

2) Test 1.B Test de integración de ALU: Para el test de integración del ALU se utilizo el programa multrightheap.asm, el cual es cargado a memoria descomentando la linea

```
initial $readmemb("memfile_fp.dat", RAM);
```

y comentando

```
initial $readmemb("memfile_mul.dat", RAM);
```

en los archivos imem.v y dmem.v. Para validar que se ejecutó correctamente el test es controlado mediante el modulo testbench_alu implementado en la sección previa.

Los resultados obtenidos de ejecutar el test se muestran en la Fig 18.

Como podemos ver en el output de consola la ejecución fue exitosa retornando como valor final el resultado esperado 0x00000028.

Asimismo, el waveform producido por el testbench se encuentra almacenado en el archivo de salida arm_single_mul.vcd. A continuación se muestra gráficamente el waveform considerando las señales más importantes (clk, reset, PC, Instr, ALUResult, AuxiliarOut, Mul, AuxW, WriteData, MemWrite y ReadData) utilizando el programa GTKWave. Estas son visible en las Figuras 19-23

3) Conclusión: En base a los resultados obtenidos de la ejecución del los tests 2.A y 2.B podemos ver que ambos tienen resultados positivos. Esto indica entonces que se logro integrar exitosamente los módulos de FPU y ALU extendido al procesador SingleCycle.

C. Test 2: Testing de implementación en procesador MultiCycle

1) Test 2.A Test de integración de FPU: Para el test de integración del FPU se utilizo el programa prodpunto.asm, el cual es cargado a memoria descomentando la linea

```
initial $readmemb("memfile_mul.dat", RAM);
```

y comentando

```
initial $readmemb("memfile_fp.dat", RAM);
```

en mem.v. Para validar que se ejecutó correctamente el test es controlado mediante el modulo testbench_fp implementado en la sección previa.

Los resultados obtenidos de ejecutar el test se muestran en la Fig 24.

Como podemos ver en el output de consola la ejecución fue exitosa retornando como valor final el resultado esperado 0x4202a40b.

Asimismo, el waveform producido por el testbench se encuentra almacenado en el archivo de salida arm_multi_fp.vcd. A continuación se muestra gráficamente el waveform considerando las señales más importantes (clk, reset, PC, state, Instr, ALUResult, AuxiliarOut, Mul, AuxW, WriteData, MemWrite y ReadData) utilizando el programa GTKWave. Estas son visible en las Figuras 25-33

2) Test 2.B Test de integración de ALU: Para el test de integración del ALU se utilizo el programa multrightheap.asm, el cual es cargado a memoria descomentando la linea

```
initial $readmemb("memfile_fp.dat", RAM);
```

y comentando

```
initial $readmemb("memfile_mul.dat", RAM);
```

en mem.v. Para validar que se ejecutó correctamente el test es controlado mediante el modulo testbench_alu implementado en la sección previa.

Los resultados obtenidos de ejecutar el test se muestran en la Fig 34.

Como podemos ver en el output de consola la ejecución fue exitosa retornando como valor final el resultado esperado 0x00000028.

Asimismo, el waveform producido por el testbench se encuentra almacenado en el archivo de salida arm_multi_mul.vcd. A continuación se muestra gráficamente el waveform considerando las señales más importantes (clk, reset, PC, Instr, ALUResult, AuxiliarOut, Mul, AuxW, WriteData, MemWrite y ReadData) utilizando el programa GTKWave. Estas son visible en las Figuras 35-44

3) Conclusión: De los resultados obtenidos de ejecutar los tests 2.A y 2.B podemos encontrar que estos son positivos. Esto indica entonces que se logro integrar exitosamente los módulos de FPU y ALU extendido al procesador MultiCycle.

D. Conclusión

Al ejecutar los tests planteados en la sección anterior obtenemos que estos son exitosos en los diferentes contextos

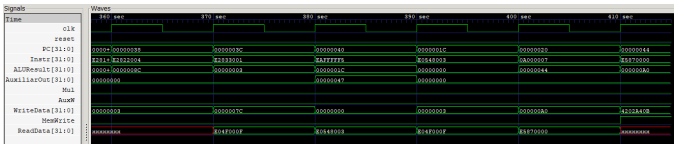

Fig. 17. Waveform Test 1.A parte 5

```
D:\Rodrigo\Rodrigo\UTEC\ciclo\2020-1\Arqui\proyecto\UPLOAD\Source\SingleCycle>vvp a.out
WARNING: imem.v:12: $readmemb: The behaviour for reg[...] mem[N:0]; $readmemb("...", mem)
g to 1364-2005 behavior.
WARNING: imem.v:12: $readmemb(memfile_mul.dat): Not enough words in the file for the requ
WARNING: dmem.v:18: $readmemb: The behaviour for reg[...] mem[N:0]; $readmemb("...", mem)
g to 1364-2005 behavior.
WARNING: dmem.v:18: $readmemb(memfile_mul.dat): Not enough words in the file for the requ
VCD info: dumpfile arm_single_mul.vcd opened for output.
wd=00000028
Simulation succeeded
testbench_mul.v:43: $stop called at 395 (1s)
** VVP Stop(0) **
** Flushing output streams.
** Current simulation time is 395 ticks.
> finish
** Continue **
```

Fig. 18. Resultado de Test 1.B

planteados. Esto implica se logro no solo exitosamente diseñar e implementar un modelo extendido de ALU y un nuevo modulo de FPU, sino que también se logro integrar correctamente estos módulos a los procesadores SingleCycle y MultiCycle presentados, realizando modificación pertinentes para ajustar estos procesadores a los nuevos diseños y usando un diseño modular en todo momento para facilitar el orden y desarrollo.

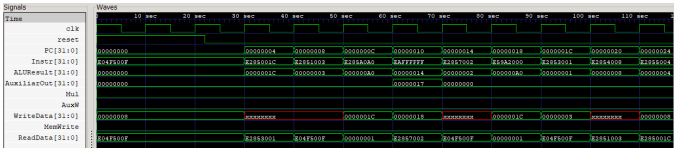

Fig. 19. Waveform Test 1.B parte 1

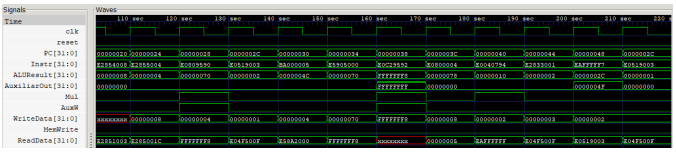

Fig. 20. Waveform Test 1.B parte 2

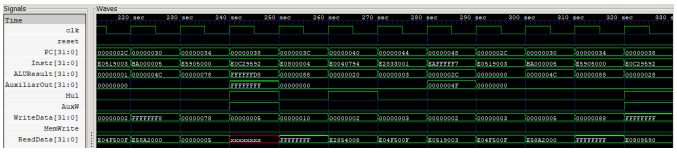

Fig. 21. Waveform Test 1.B parte 3

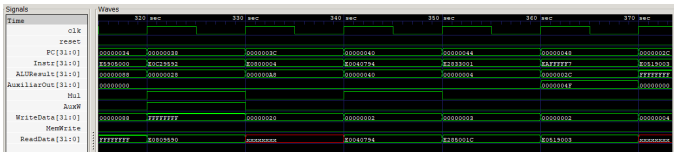

Fig. 22. Waveform Test 1.B parte 4

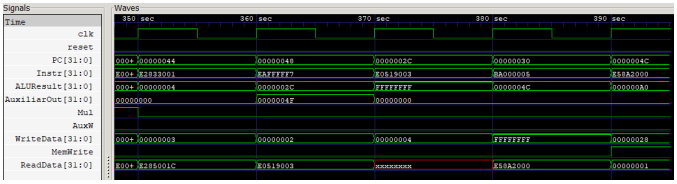

Fig. 23. Waveform Test 1.B parte 5

```
D:\Rodrigo\Rodrigo\UTEC\ciclo\2020-1\Arqui\proyecto\UPLOAD\Source\MultiCycle>vvp a.out
WARNING: mem.v:16: $readmemb: The behaviour for reg[...] mem[N:0]; $readmemb("...", mem
stop);. Defaulting to 1364-2005 behavior.
WARNING: mem.v:16: $readmemb(memfile_fp.dat): Not enough words in the file for the requ
VCD info: dumpfile arm_multi_fp.vcd opened for output.
wd=4202a40b
Simulation succeeded
testbench_fp.v:42: $stop called at 1195 (1s)
** VVP Stop(0) **
** Flushing output streams.
** Current simulation time is 1195 ticks.
> finish
** Continue **
```

Fig. 24. Resultado de Test 2.A

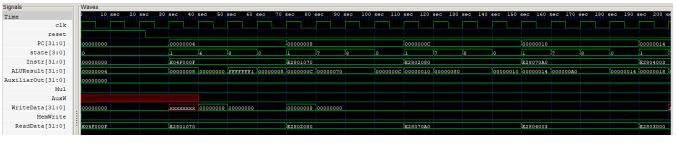

Fig. 25. Waveform Test 2.A parte 1

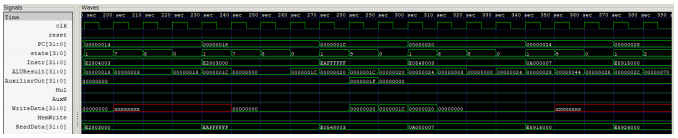

Fig. 26. Waveform Test 2.A parte 2

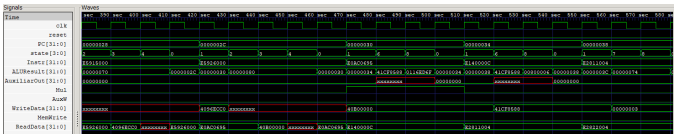

Fig. 27. Waveform Test 2.A parte 3

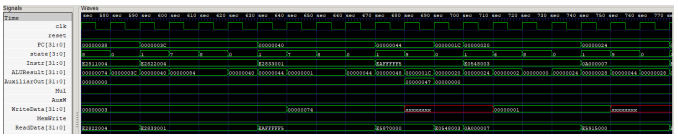

Fig. 28. Waveform Test 2.A parte 4

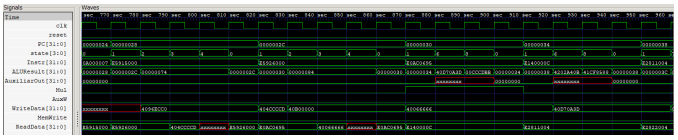

Fig. 29. Waveform Test 2.A parte 5

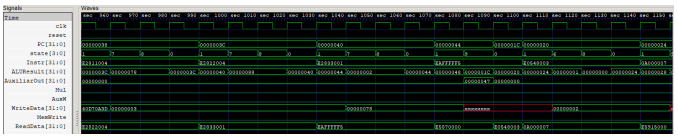

Fig. 30. Waveform Test 2.A parte 6

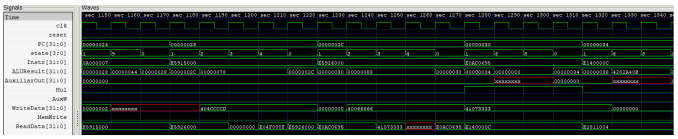

Fig. 31. Waveform Test 2.A parte 7

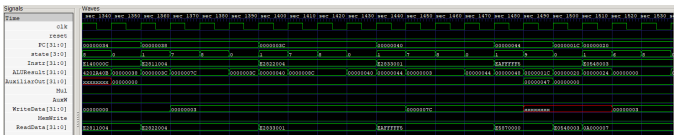

Fig. 32. Waveform Test 2.A parte 8

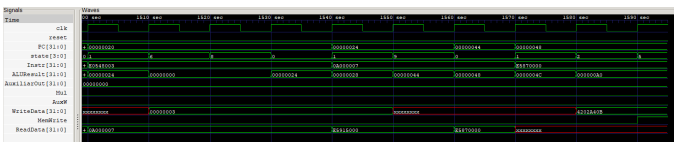

Fig. 33. Waveform Test 2.A parte 9

```
D:\Rodrigo\Rodrigo\UTEC\ciclo\2020-1\Arqui\proyecto\UPLOAD\Source\MultiCycle>vvp a.out
WARNING: mem.v:18: $readmemb: The behaviour for reg[...] mem[N:0]; $readmemb("...", mem
stop);. Defaulting to 1364-2005 behavior.
WARNING: mem.v:18: $readmemb(memfile_mul.dat): Not enough words in the file for the req
VCD info: dumpfile arm_multi_mul.vcd opened for output.
wd=00000028
Simulation succeeded
testbench_mul.v:43: $stop called at 1495 (1s)
** VVP Stop(0) **
** Flushing output streams.
** Current simulation time is 1495 ticks.
> finish
** Continue **
```

Fig. 34. Resultado de Test 2.B

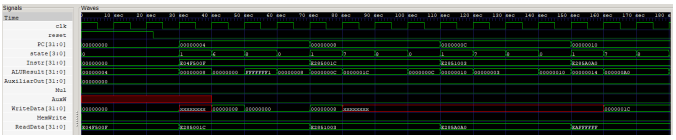

Fig. 35. Waveform Test 2.B parte 1

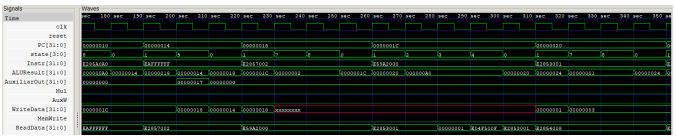

Fig. 36. Waveform Test 2.B parte 2

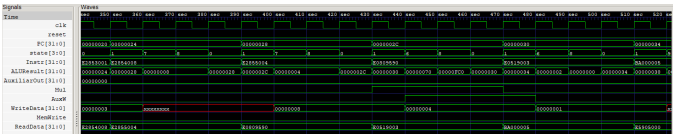

Fig. 37. Waveform Test 2.B parte 3

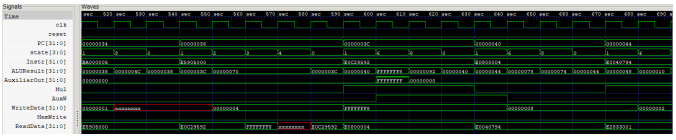

Fig. 38. Waveform Test 2.B parte 4

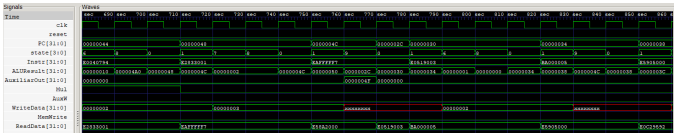

Fig. 39. Waveform Test 2.A parte 5

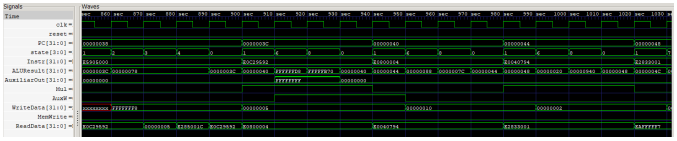

Fig. 40. Waveform Test 2.B parte 6

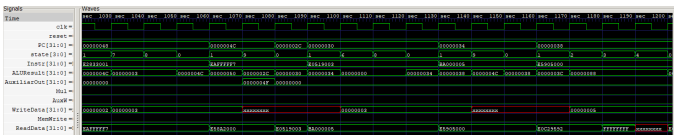

Fig. 41. Waveform Test 2.B parte 7

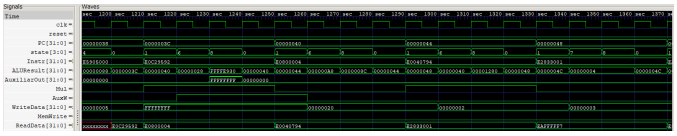

Fig. 42. Waveform Test 2.B parte 8

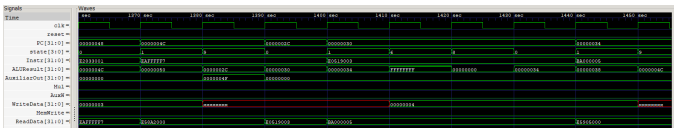

Fig. 43. Waveform Test 2.B parte 9

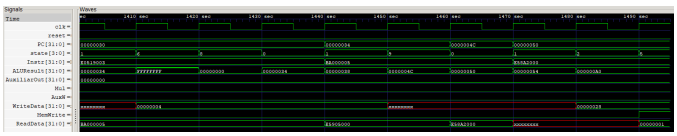

Fig. 44. Waveform Test 2.B parte 10