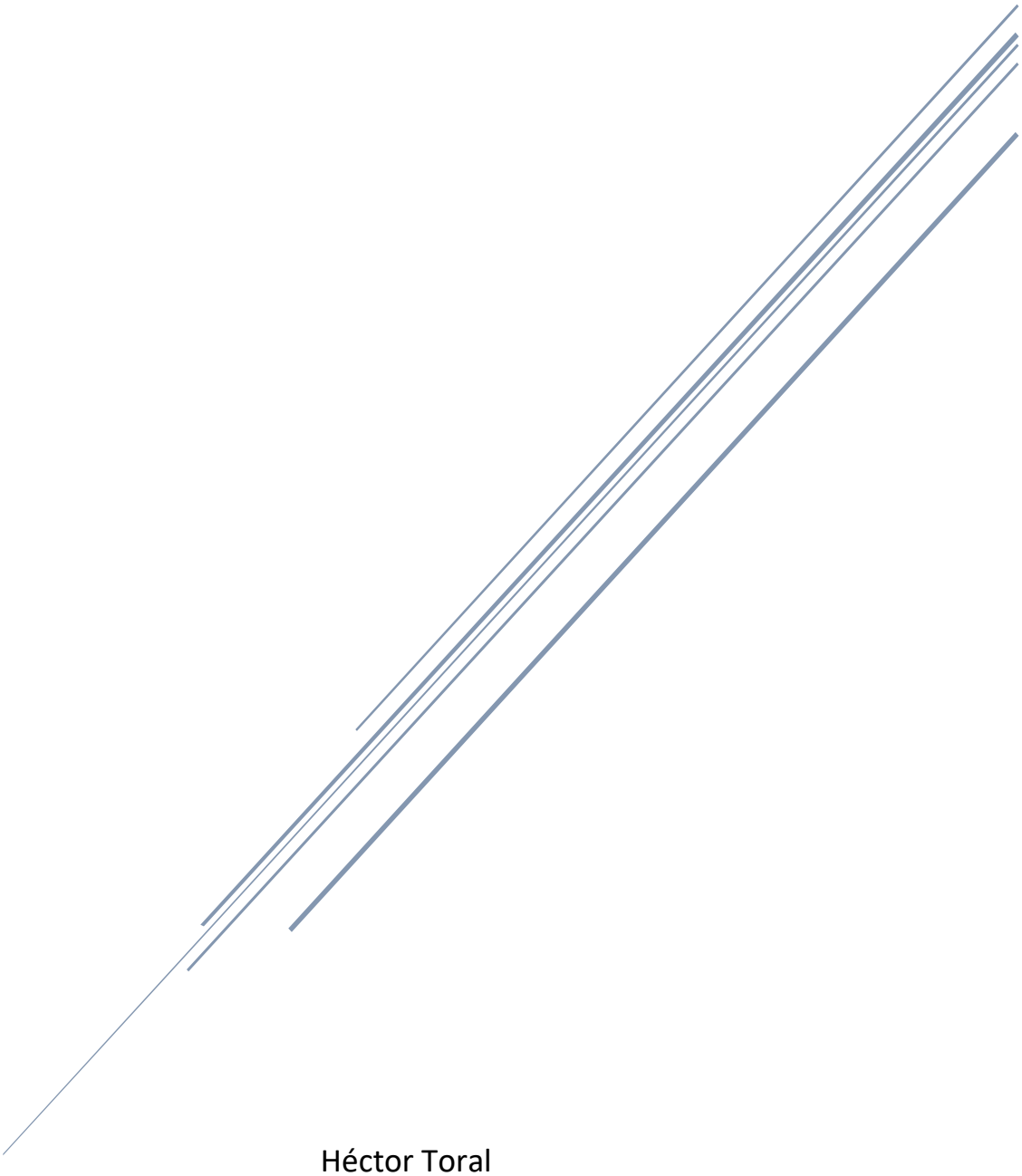
PROYECTO 1

MIPS Segmentado



Héctor Toral
Javier Pizarro

Índice

Unidad de control

```
-- nuevo ADDFP
WHEN ADDFP => Branch      <= '0'; RegDst  <= '1'; ALUSrc   <= '0';
              MemWrite    <= '0'; MemRead <= '0'; MemtoReg <= '0';
              RegWrite    <= '0'; FP_add  <= '1'; FP_mem   <= '0';
              RegWrite_FP <= '1';
-- nuevo LWFP
WHEN LWFP => Branch   <= '0'; RegDst  <= '0'; ALUSrc   <= '1';
             MemWrite <= '0'; MemRead <= '1'; MemtoReg <= '0';
             RegWrite <= '0'; FP_add  <= '0'; FP_mem   <= '1';
             RegWrite_FP <= '1';
-- nuevo SWFP
WHEN SWFP => Branch      <= '0'; RegDst  <= '0'; ALUSrc   <= '1';
             MemWrite    <= '1'; MemRead <= '0'; MemtoReg <= '0';
             RegWrite    <= '0'; FP_add  <= '0'; FP_mem   <= '1';
             RegWrite_FP <= '0';
```

Ruta: ~/UC_Mips.vhd

Rango: 69-74

Unidad de Anticipación

```
Corto_A_Mem <= '1' when RegWrite_MEM = '1' and RW_MEM = Reg_Rs_EX
                   else '0';
Corto_B_Mem <= '1' when RegWrite_MEM = '1' and RW_MEM = Reg_Rt_EX
                   else '0';
Corto_A_WB  <= '1' when RegWrite_WB  = '1' and RW_WB  = Reg_Rs_EX
                   else '0';
Corto_B_WB  <= '1' when RegWrite_WB  = '1' and RW_WB  = Reg_Rt_EX
                   else '0';
-- Corto_A
MUX_ctrl_A <= "01" when Corto_A_Mem = '1' else
              "10" when Corto_A_WB  = '1' else "00";
-- Corto_B
MUX_ctrl_B <= "01" when Corto_B_Mem = '1' else
              "10" when Corto_B_WB  = '1' else "00";
```

Ruta: ~/UA.vhd

Rango de líneas: 34-44

La red de anticipación esta cableada de tal manera que al llegar un par de instrucciones de tipo productor consumidor en etapa de ejecución o memoria se pueda anticipar, permitiendo así a la instrucción consumidora obtener el dato que necesita para operar.

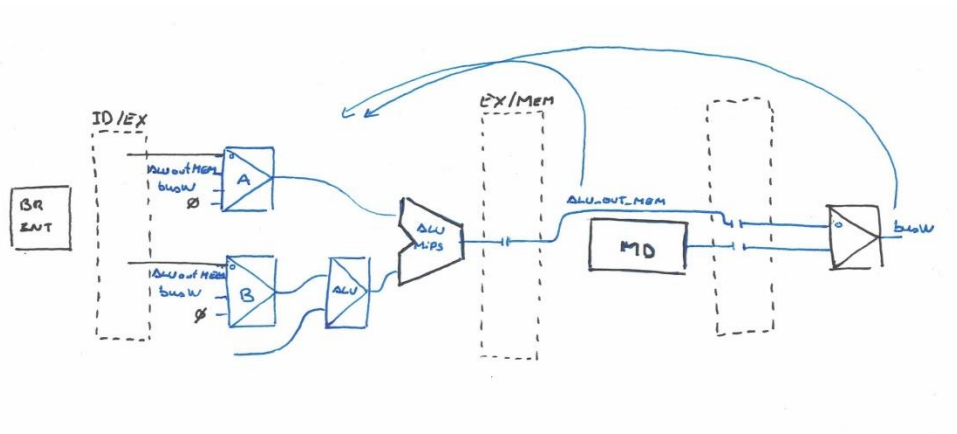


Figura 1: UA

En este primer ejemplo, la primera instrucción “add” genera un resultado que será anticipado en el siguiente ciclo de reloj para la etapa de ejecución del siguiente “add” permitiendo así ahorrar 2 ciclos de reloj evitando parar la CPU.

add r1 r2 r2	**F**	**D**	**E**	**M**	**W**	
add r3 r1 r2		**F**	**D**	**E**	**M**	**W**

En este segundo ejemplo, la primera instrucción “add” anticipa el resultado a la siguiente instrucción “add” con la que tiene 1 instrucción de separación. Así, al anticipar desde escritura en banco de registros a ejecución permite que la segunda instrucción “add” no se detenga en “deco”.

add r1 r2 r2	**F**	**D**	**E**	**M**	**W**		
nop		**F**	**D**	**E**	**M**	**W**	
add r3 r1 r2			**F**	**D**	**E**	**M**	**W**

Unidad de detención

```
...
parar_EX_FP <= parar_EX_FP_internal;
Parar_ID    <= '1' when parar_ID_FP = '1' or
                    ld_uso_rs = '1' or ld_uso_rt = '1' or
                    BEQ_rs = '1' or BEQ_rt = '1' else '0';
Kill_IF     <= '1' when IR_op_code = BEQ and PCSrc = '1' and
                    BEQ_rs = '0' and BEQ_rt = '0' else '0';
```

Ruta: UD.vhd

Rango 37-70

La unidad de detención se encarga de controlar 3 tipos de eventos:

- **Riesgo estructural**, gestionado por la señal: "parar_EX_FP" que detiene el sistema en el momento en que se está utilizando el sumador de racionales.
- **Riesgos de datos**, detectado mediante la señal: "Parar_ID", esta señal es la encargada de hacer esperar a aquellas instrucciones cuyo valor de registro todavía no se encuentre debidamente actualizado.
- **Riesgo de control**, gestionado por: "Kill_IF", responsable de eliminar la instrucción saliente de MI (Memoria de instrucciones) si el salto no ha sido tomado.

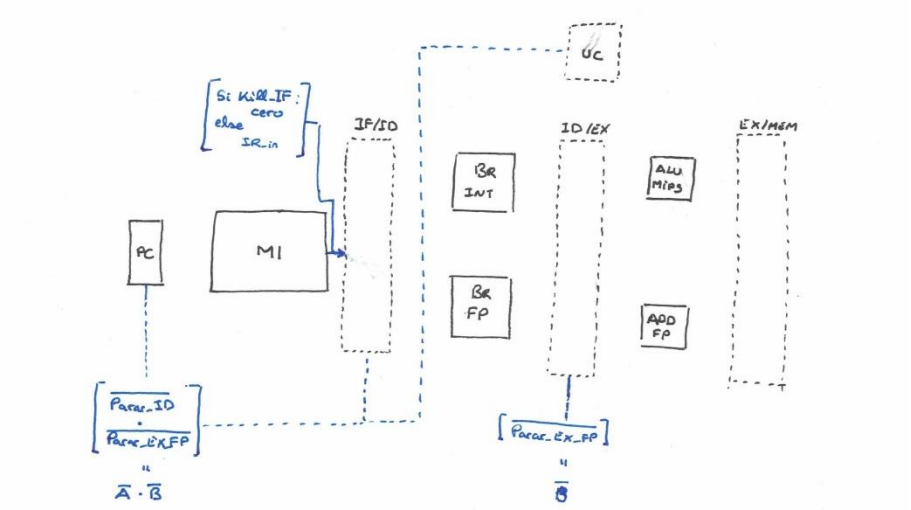


Figura 2: UD

Mips Segmentado

UD – Etapa IF

En esta etapa, se realiza parada si y solo si las señales de "Parar_ID" o "Parar_EX_FP" están activas a 1 y se elimina la instrucción que se va a escribir en IR si el salto no ha sido tomado.

```
pc: reg32 port map (Din => PC_in, clk => clk, reset => reset,
                    load => load_PC, Dout => PC_out );
load_PC    <= '0' when Parar_ID = '1' or Parar_EX_FP = '1' else '1';
load_IF_ID <= '0' when Parar_ID = '1' or Parar_EX_FP = '1' else '1';
IR_KILL    <= cero when Kill_IF = '1' else IR_in;
Banco_IF_ID: Banco_ID port map (IR_in => IR_KILL, PC4_in => PC4,
             clk => clk, reset => reset, load => load_IF_ID,
             IR_ID => IR_ID, PC4_ID => PC4_ID);
```

UD – Etapa ID

En esta fase, se modifican las salidas de las señales de control de escritura y/o lectura en BR y MemD para no mandar una acción incorrecta.

```
Unidad_det: UD port map (Reg_Rs_ID => Reg_Rs_ID,
  Reg_Rt_ID    => Reg_Rt_ID,    MemRead_EX => MemRead_EX,
  RegWrite_EX  => RegWrite_EX,  RW_EX   => RW_EX,
  RegWrite_Mem => RegWrite_Mem, RW_Mem  => RW_Mem,
  IR_op_code   => IR_op_code,   PCSrc   => PCSrc,
  FP_add_EX    => FP_add_EX,    FP_done => FP_done,
  RegWrite_FP_EX  => RegWrite_FP_EX, RW_FP_EX => RW_FP_EX,
  RegWrite_FP_MEM => RegWrite_FP_MEM, RW_FP_MEM => RW_FP_MEM,
  Kill_IF => Kill_IF, Parar_ID => Parar_ID,
  Parar_EX_FP => Parar_EX_FP);

RegWrite_FP_ID <= '0' when Parar_ID = '1' or Parar_EX_FP = '1'
                      else RegWrite_FP;
RegWrite_ID    <= '0' when Parar_ID = '1' or Parar_EX_FP = '1'
                      else RegWrite;
FP_add_ID      <= '0' when Parar_ID = '1' or Parar_EX_FP = '1'
                      else FP_add;
MemWrite_ID    <= '0' when Parar_ID = '1' or Parar_EX_FP = '1'
                      else MemWrite;
MemRead_ID     <= '0' when Parar_ID = '1' or Parar_EX_FP = '1'
                      else MemRead;
```

UD – Etapa EX

En esta etapa se mantiene a 0 la entrada del load si y solo si se ha detectado un riego estructural.

```
load_EX_FP <= '0' when Parar_EX_FP = '1' else '1';
```

Contadores

```
component counter is
  Port (clk    : in STD_LOGIC;
        reset : in STD_LOGIC;
        count_enable : in STD_LOGIC;
        load  : in  STD_LOGIC;
        D_in  : in  STD_LOGIC_VECTOR (7 downto 0);
        Count : out STD_LOGIC_VECTOR (7 downto 0));
end component;

...
KIF     <= '1' when Kill_IF = '1' and parar_ID = '0' and
              parar_EX_FP = '0' else '0';
PID     <= '1' when parar_ID = '1' and parar_EX_FP = '0' else '0';
PEFP    <= '1' when parar_EX_FP = '1' else '0';

count_paradas: counter port map (clk => clk, reset => reset,
  count_enable => KIF, load => '0', D_in => "00000000",
  count => paradas_control);
count_datos: counter port map (clk => clk, reset => reset,
  count_enable => PID, load => '0', D_in => "00000000",
  count => paradas_datos);
count_fp: counter port map (clk => clk, reset => reset,
  count_enable => PEFP, load => '0', D_in => "00000000",
  count => paradas_fp);
```

Banco de Pruebas

Para comprobar el correcto funcionamiento del procesador, se ha creado un banco de pruebas en el que se han probado cado uno de las posibles paradas y anticipaciones, además de algún que otro caso para comprobar el riesgo estructural y la parada de control.

Anticipacion EX/MEM	
inst1	inst2
add rd rs rt	add rd rs rt
add rd rs rt	lw rt inm rs
add rd rs rt	sw rt inm rs
add rd rs rt	beq rs rt inm
add rd rs rt	addfp rd rs rt
add rd rs rt	lwfp rt inm rs
add rd rs rt	swfp rt inm rs

Anticipacion MEM/WB	
inst1	inst2
add rd rs rt	add rd rs rt
add rd rs rt	lw rt inm rs
add rd rs rt	sw rt inm rs
add rd rs rt	beq rs rt inm
add rd rs rt	addfp rd rs rt
add rd rs rt	lwfp rt inm rs
add rd rs rt	swfp rt inm rs

Anticipacion MEM/WB	
inst1	inst2
lw rt inm rs	add rd rs rt
lw rt inm rs	lw rt inm rs
lw rt inm rs	sw rt inm rs
lw rt inm rs	beq rs rt inm
lw rt inm rs	addfp rd rs rt
lw rt inm rs	lwfp rt inm rs
lw rt inm rs	swfp rt inm rs

parar_id	
inst1	inst2
lw rt inm rs	add rd rs rt
lw rt inm rs	lw rt inm rs
lw rt inm rs	sw rt inm rs
lw rt inm rs	beq rs rt inm
lw rt inm rs	addfp rd rs rt
lw rt inm rs	lwfp rt inm rs
lw rt inm rs	swfp rt inm rs
addfp rd rs rt	addfp rd rs rt
addfp rd rs rt	lwfp rt inm rs
addfp rd rs rt	swfp rt inm rs
lwfp rt inm rs	addfp rd rs rt
lwfp rt inm rs	lwfp rt inm rs
lwfp rt inm rs	swfp rt inm rs

Figura 3: Tabla de pruebas

Tras ejecutar el fichero de pruebas, se ha podido comprobar como se han ido detectando y ejecutando los distintos eventos programados, en la UA y la UD.

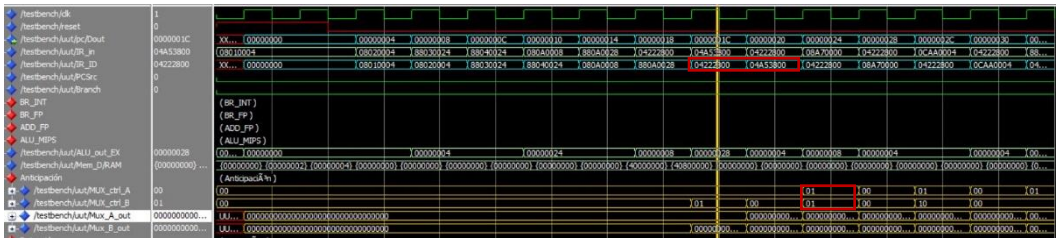


Figura 4: Anticipación en rs y rt

En la figura de la parte superior se muestra el valor que toman los muxes A y B (01) colocados para la anticipación de operandos.

El código que provoca esa anticipación es:

```
add  r5 r1 r2;
add  r7 r5;
```

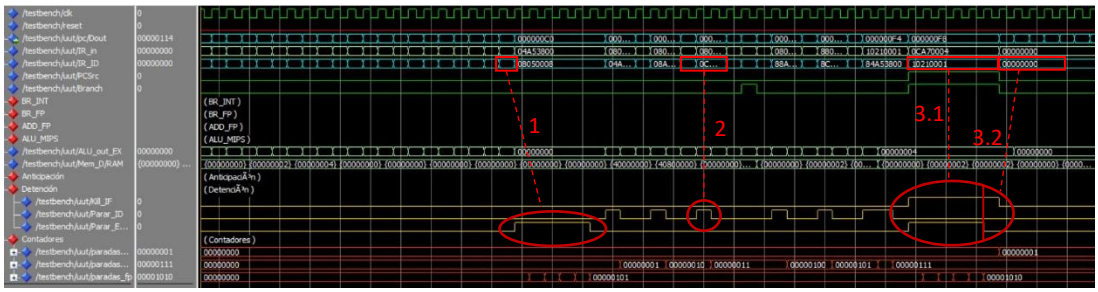


Figura 5: UD en acción

En la figura 5, se pueden observar los 3 tipos de eventos que están programados en la UD.

Parar_EX_FP

Este se encuentra identificado por el número 1 en la imagen. Como se puede observar en la simulación se ve como detiene a la CPU durante 5 ciclos cada vez que se realiza una operación addfp.

Parar_ID

Identificado mediante el número 2 en la ilustración se puede ver como este aparece más de una vez. En concreto, el causante de esta parada señala es el siguiente par de instrucciones:

```
lw r5 #8 r0;
sw r7 #4 r5;
```

Kill_IF

Representado por los números 3.1 y 3.2 en la imagen se puede ver cómo tras una ejecución de "addfp" que detiene la CPU durante 5 ciclos (ver 3.1), la señal "Kill_IF" ha saltado, pero esta no es contada hasta que no se resuelve en 3.2 donde además se puede apreciar como borra la operación "sw" que iba a continuación.

Esta ejecución es debida a este trio de instrucciones ensamblador:

```
addfp r7 r5; -- causante del parar_ex_fp
beq r1 #1;   -- causante del kill_if
sw r7 #4 r5;    -- eliminada por el kill_if
```

Rendimiento

Para el cálculo del rendimiento entre el MIPS con anticipación de operandos vs el MIPS sin anticipación, se ha decidido utilizar el propio código del banco de pruebas. Tras haber realizado la ejecución del programa para ambas versiones han salido los siguientes resultados:

MIPS con anticipación: 829574ps | 80 ciclos
Mips sin anticipación: 19884751ps | 202 ciclos

Speed Up = (Núm. ciclos con anticipación) / (Núm. Ciclos sin anticipación) = 0.39

Tras ver este resultado podemos decir que la unidad de anticipación es un 61% más rápida que la que no tiene unidad de anticipación. Esto se debe principalmente a que esta permite a la CPU seguir ejecutándose, evitando así paradas innecesarias.

Gestión del trabajo

Fecha	Tiempo	Autor/es	Tarea
¿?	¿?	Javier Pizarro	Lectura del enunciado
¿?	3-4h	Héctor Toral	Lectura del enunciado y del código para ver cómo está estructurado y conectado por dentro
¿?	1h	Javier Pizarro, Héctor Toral	UC
15/04/22	30m	Héctor Toral	UA
17/04/22	2h	Héctor Toral	UD
18/04/22	2h	Héctor Toral	UA <- debug
19/04/22	2h	Javier Pizarro, Héctor Toral	UA <- debug, UD<- debug
21/04/22	2h	Javier Pizarro, Héctor Toral	UD <-debug
24/04/22	4h	Javier Pizarro, Héctor Toral	UD-Mips_Segmentado <- debug
25/05/22	5h	Javier Pizarro, Héctor Toral	Memoria + UD
26/05/22	7h	Héctor Toral	UD-Mips_Segmentado <- debug + testing
27/05/22	7h	Héctor Toral	UD-Mips_Segmentado <- debug + testing
28/05/22	3h	Javier Pizarro, Héctor Toral	Memoria
29/05/22	7h	Hector Toral	Javier Pizarro

Conclusiones y autoevaluación

Como conclusión del trabajo realizado, hemos podido plasmar en la práctica aquellos conceptos que se nos contaron en clase de teoría pudiendo implementar una unidad de anticipación, y una unidad de detención. A su vez, tras largas sesiones de revisar errores que iban apareciendo durante la implementación de la unidad de detención, han ido surgiendo nuevas ideas sobre como implementar alguna de las partes, ampliando así nuestra capacidad para poder encarar un repertorio nuevo de problemas.

Además de todo esto, hemos podido aprender cómo realizar bancos de pruebas para comprobar el correcto funcionamiento del código mediante técnicas como el “smoke testing”, pruebas que verifican la funcionalidad básica de una aplicación de forma rápida, estas pruebas al final fueron unificadas en un único fichero que ha sido el entregado

Por último, hemos aprendido que la gestión del tiempo es importante y que contabilizar el tiempo es más importante de lo que parece, para futuros proyectos se tendrá más presente.

Como conclusión y autoevaluación, creemos que el trabajo realizado ha sido bastante bueno.