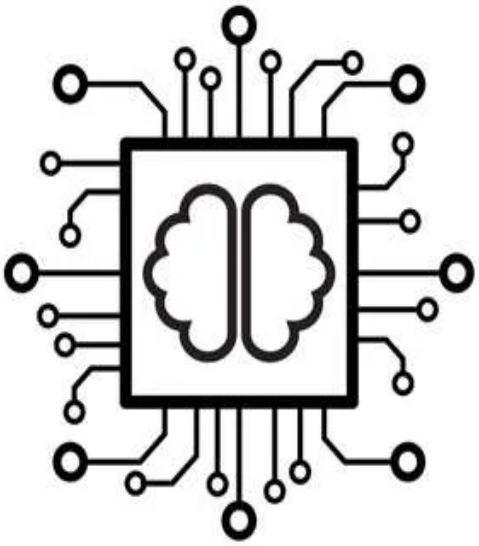
Universidad Nacional del Centro de la
Provincia de Buenos Aires

FACULTAD DE CIENCIAS EXACTAS

Ingeniería de Sistemas

Arquitectura de Computadoras I

Informe de Laboratorio
Microprocesador MIPS Segmentado



Profesor a cargo: Oscar Goñi

Martín Vazquez Arispe martin.vazquez.arispe@gmail.com

Burckhardt David burck432@gmail.com

29/06/2022

Introducción:

Durante el desarrollo de la cursada de arquitectura de computadoras de 2022 se puso en práctica el conocimiento adquirido en la teoría para implementar el procesador MIPS segmentado utilizando VHDL, un lenguaje que permite describir Hardware.

La idea principal fue reconstruirlo etapa por etapa, para luego realizar algunas pruebas de funcionamiento. Las etapas que componen el procesador son:

- IF (Fetching): Lectura de instrucción.
- ID (Decode): Decodificación de instrucción, lectura de operandos, cálculo de la dirección y condición de salto.
- EX (Execute): Ejecución de operación.
- MEM (Memory): Acceso a memoria.
- WB (WriteBack): Escritura en los registros.

La frecuencia del procesador implementado es de **20MHz**. Dato calculado a partir de conocer el tiempo de ciclo de reloj de 50 ns.

Modificaciones:

- ***Saltos condicionales:*** Una de las modificaciones que se realizó para esta implementación fue el adelanto del cálculo de la condición y dirección de salto. Se adelantaron estas operaciones a la segunda etapa con el objetivo de disminuir los ciclos perdidos ante un salto efectivo. Previamente, ante un salto, se perdían tres ciclos debido a la necesidad de hacer un flush de las tres etapas anteriores. Ahora, gracias a adelantar el cálculo, en caso de salto efectivo solo se pierde un ciclo por el flush de la primera etapa. Para lograr esta tarea fue necesario: Ver *Imagen 1*
    - Verificar que los datos de salida del banco de registros sean iguales.
    - Extender el signo del valor inmediato a 32 bits.
    - Multiplicar por 4 el dato anterior. (Concatenar dos ceros al principio)
    - Calcular el target sumando el PC+4 al valor anterior.

```
--Extension de signo
resultExtension <= x"ffff" & IFID_Instr(15 downto 0) when (IFID_Instr(15) = '1')
                   else  x"0000" & IFID_Instr(15 downto 0);

--Desplazamiento x2
resultShiftLeft <= resultExtension(29 downto 0) & "00";

--Calculo target
target <= resultShiftLeft + IFID_Pcmas4;

--Condicion salto
resultComparacion <= '1' when (data1_rd = data2_rd) else '0';
PCSrc <= resultComparacion and ID_Branch;
```

Imagen 1: *Aquí se pueden notar, señalados en rojo, los cambios mencionados que se encuentran en la etapa Decode*

- **Unidad de Hazard:** Otra de las modificaciones fue el añadido de la unidad de detección de riesgos. Este hardware extra brinda la posibilidad detectar y actuar ante los riesgos de tipo RAW insalvables producidos por las instrucciones LW seguido de

una instrucción que requiera del registro cargado. Para detectar este tipo de riesgos se requirió:

- Realizar una comparación de registros de lectura (Rs y Rt) de la etapa Decode con el registro destino de la etapa Execute.
- Verificar que la instrucción en la etapa Ex quiere leer un dato de la memoria(es de tipo LW, ID_MemRead = ‘1’)

Esta unidad también puede detectar riesgos RAW en caso de salto condicional. Para detectar este tipo de riesgos se requirió:

- Realizar una comparación de registros de lectura (Rs y Rt) de la etapa Decode con el registro destino de la etapa Execute y la etapa Memory.
- Verificar que la instrucción en la etapa Decode quiere hacer un salto condicional. (ID_Branch = ‘1’)
- Verificar que los registros a leer sean distintos de 0 para evitar detectar riesgos de dependencias con instrucciones NOP.

Si un riesgo es detectado entonces se procede a:

- Introducir una instrucción NOP en el pipeline mediante este nuevo hardware. Esta instrucción NOP asigna ceros en las señales de control del registro de segmentación ID/EX.
- Bloquear la escritura del PC
- Bloquear la escritura del registro de segmentación IF/ID.

De tal manera se pierde un ciclo de reloj pero con la ayuda de la unidad de forwarding se logra evadir el riesgo de dependencia . Ver *imagen 2*

```
process (readMem, branch, regDstEx, regDstMem, readRs, readRt) begin
    if (readMem = '1' and (regDstEx = readRs or regDstEx = readRt)) then
        PcWrite <= '0';
        ifIdWrite <= '0';
        nop <= '1';
    else if (branch = '1' and (regDstEx = readRs or regDstEx = readRt or regDstMem = readRs
or regDstMem = readRt) and readRt /= "00000" and readRs /= "00000") then
        PcWrite <= '0';
        ifIdWrite <= '0';
        nop <= '1';
    else
        PcWrite <= '1';
        ifIdWrite <= '1';
        nop <= '0';
    end if;
    end if;
end process;
```

Imagen 2: *Señalados con rojo se encuentran las condiciones a cumplir para detectar el riesgo RAW insalvable mientras que en verde se denotan las medidas que tomará la unidad de Hazard frente a este. Por otro lado en amarillo se marcaron las condiciones para detectar un salto condicional que tiene riesgo/s RAW con instrucciones anteriores, las medidas a tomar son las mencionadas anteriormente.*

- **Unidad de Forwarding:** Con el objetivo de eliminar los riesgos por dependencias de datos se incorporó la unidad de adelantamiento. Este hardware adicional funciona a partir de las siguientes condiciones: Ver *imagen 3*
  - Comparar los registros de escritura de las instrucciones más avanzadas (Ex y Mem) con los registros de lectura de la etapa Decode.
  - Verificar que la instrucción más adelantada necesita escribir en un registro. (regWriteMem = ‘1’ o regWriteWb =’1’)
  - Verificar que los registros de lectura no sean el registro $zero.

Para enviar una respuesta que no interfiera con el flujo de instrucciones las salidas del adelantamiento se conectaron como señales de dos multiplexores adicionales(ForwardA y ForwardB). Estos permiten decidir entre: Ver *imagen 4*

- Ingresar el dato del banco de registros.
- Ingresar el dato que se encuentra en el registro EX/MEM.
- Ingresar el dato que se encuentra en el registro MEM/WB.

Se tuvo en cuenta, además, que el dato a escribir en memoria en caso de una instrucción SW, puede ser un dato que necesite ser adelantado por lo que sale del Mux ForwardB. Frente a esta consideración se debió agregar un nuevo multiplexor que decide que ingresar en la entrada b de la ALU, ya sea, la salida del Mux ForwardB o el valor inmediato, éste es controlado por la señal ALUSrc . Ver *imagen 4*

```
-- Adelantar desde Etapa Mem
condMemRs <= '1' when (regWriteMem = '1' and readRs /= "00000" and readRs = regDstMem) else '0';
condMemRt <= '1' when (regWriteMem = '1' and readRt /= "00000" and readRt = regDstMem) else '0';

-- Adelantar desde Etapa Wb
condWbRs <= '1' when (regWriteWb = '1' and readRs /= "00000" and not(regWriteMem = '1' and readRs = regDstMem) and readRs = regDstWb) else '0';
condWbRt <= '1' when (regWriteWb = '1' and readRt /= "00000" and not(regWriteMem = '1' and readRt = regDstMem) and readRt = regDstWb) else '0';
```

Imagen 3: *Señalado con verde se observa las condiciones a cumplir para que el adelantamiento de datos sea efectivo*

```
--Mux ForwardA
MuxEntradaA: process (IDEX_data1_rd, MemToRegMux, EXMEM_ALUResult, ForwardA) begin
    case ForwardA is
        when "00" => EntradaA <= IDEX_data1_rd;
        when "01" => EntradaA <= MemToRegMux;
        when others => EntradaA <= EXMEM_ALUResult;
    end case;
end process;

--Mux ForwardB
MuxEntradaB: process (IDEX_data2_rd, MemToRegMux, EXMEM_ALUResult, ForwardB) begin
    case ForwardB is
        when "00" => salidaForwB <= IDEX_data2_rd;
        when "01" => salidaForwB <= MemToRegMux;
        when others => salidaForwB <= EXMEM_ALUResult;
    end case;
end process;

--MuxImediato
EntradaB <= salidaForwB when (IDEX_aluSrc = '0') else IDEX_immExt;
```

Imagen 4: *Señalado con rojo se observa las posibles datos de entrada "a" de la ALU que dependen de la condición que verifica la unidad de forwarding junto con los datos en amarillo que referencian la salida del Mux ForwardB. Por otro lado, en color verde se observa la entrada "b" de la ALU que selecciona entre la salida del Mux ForwardB o el valor inmediato.*

- **Salto incondicional:** Una nueva modificación fue la incorporación de la instrucción Jump. La cual es un salto incondicional, es decir, siempre se va a efectuar. El hardware extra funciona de la siguiente manera:
  - La unidad de control es capaz de reconocer un jump.
  - En la etapa Decode se realiza el cálculo de la dirección de salto. Para ello se efectúan las siguientes operaciones : Ver *Imagen 5*
    - Tomar los bits del 26 al 0 de la instrucción (addr).
    - Multiplicar por 4 (ShiftLeft x2)

- Concatenar el valor anterior a los bits 31 a 28 del PC+4.

```
--Desplazamiento_Jump x2
resultShiftLeft_Jump <= IFID_Instr(25 downto 0) & "00";

--Calculo targetJump
targetJump <= IFID_Pcmas4(31 downto 28) & resultShiftLeft_Jump;
```

Imagen 5: *Señalado en rojo la multiplicación por 4 del campo addr de la instrucción jump, por otra parte, en verde la concatenación de los 4 bits de más peso del PC+4 y el resultado del producto para así obtener el targetJump.*

- Se lleva la dirección a la entrada del PC para así efectuar el salto, esto se realiza de la siguiente forma: Ver *Imagen 6*
  - Un nuevo Mux manejado por la señal ID_Jump decide si pasa la señal del Mux original(Manejado por la señal PCSrc que transmite la señal del PC+4 o la dirección del salto del Beq) o la dirección de salto del Jump.
  - Para dar paso a la dirección dada por la instrucción Jump la señal ID_Jump debe ser igual a 1

```
MuxPC4_Target <= PcMas4 when (PCSrc = '0') else targetBEQ;

IF_PcIn <= MuxPC4_Target when (ID_Jump = '0') else targetJump;
```

Imagen 6: *Se señala en rojo el Mux original que transmite la señal del PC+4 si PCSrc es igual a 0, de lo contrario, transmite la dirección del target dada por el Beq. Por otro lado, en verde, se ve el nuevo Mux que transmite a la entrada del PC, la salida del Mux anterior si ID_Jump es igual a 0 o el target de la instrucción Jump en caso contrario.*

- **Addi:** Para incorporar las sumas de valores inmediatos se decidió incorporar la instrucción ADDI. Esta, es de Tipo I, y se encarga de guardar en un registro dado, la suma del valor inmediato extendido más el valor de otro registro del banco de registros. No fue necesario hardware adicional pero se complejizó un poco el funcionamiento de la unidad de control: Ver *imagen 7*
  - La unidad de control necesita detectar un nuevo opcode = “001000” para indicar que se trata de esta nueva instrucción y actuar al respecto.
  - La ALU Control ya conoce el código de AluOp “00” que indica la suma por lo que trabaja normalmente. Lo mismo sucede con la ALU.

```
when "001000" => -- ADDI Type
    ID_RegWrite <= '1';
    ID_MemToReg <= '0';
    ID_Branch <= '0';
    ID_MemRead <= '0';
    ID_MemWrite <= '0';
    ID_RegDst <= '0';
    ID_AluOp <= "00";
    ID_AluSrc <= '1';
    ID_Jump <= '0';
```

Imagen 7: *Aquí se puede observar el agregado de la instrucción ADDI en la unidad de control.*

- **SLL y SRL:** Para casos donde se necesite hacer un corrimiento(lógico) a izquierda/derecha, se incorporó al procesador la capacidad de poder realizarlos. El

corrimiento se logra desplazando todos los bits hacia la izquierda/derecha y concatenando la cantidad de ceros al final/principio dada por el campo Shamt de la instrucción. Para que el procesador pueda llevar a cabo esta instrucción fueron necesarios algunos cambios: Ver *imagen 8*

- En primer lugar estas instrucciones son de Tipo R, por lo tanto, en la unidad de control no fue necesario agregar hardware adicional.
- La ALUControl decide qué operación realizar con el campo funct de la instrucción (funct = 000000 para corrimiento a izquierda y funct = 000010 para corrimiento a derecha)
- La ALU es informada para que efectúe la instrucción SLL o SRL (codigoAlu = 0110 para SLL o codigoALU = 1000 para SRL) y luego ese resultado es almacenado en el registro Rd.

- **XOR y NOR:** Se agregaron dos instrucciones de Tipo R. Ver i*magen 8*

```
Operaciones: process(control,a,b) begin
      case control is
        when"0000" => result <= (a and b);
        when"0001" => result <= (a or b);
        when"0010" => result <= (a + b);
        when"0011" => result <= (a - b);
        when"0100" => result <= b(15 downto 0) & x"0000";
        when"0110" => result <= b sll to_integer(shamt); -- Corrimiento a izquierda
        when"1000" => result <= b srl to_integer(shamt); -- Corrimiento a derecha
        when"1001" => result <= (a xor b);
        when"1010" => result <= (a nor b);
        when"0111" => if (a<b) then
                        result <= x"00000001";
                      else
                        result <= x"00000000";
                      end if;

        when others => result <= x"00000000";
      end case;
end process;
```

Imagen 8: *Se puede observar en rojo la operación Shift Left Logic (SLL) en la ALU, con su respectivo código de operación (0110). En verde se ve la operación Shift Right Logic (SRL) en la ALU, con su respectivo código de operación (1000). Por ultimo, en amarillo, se marcan las dos nuevas operaciones XOR y NOR de la ALU con códigos de operacion (1001) y (1010) respectivamente.*

- **ANDI,ORI,XORI:** Se agregaron tres instrucciones de Tipo I. Fue necesario agregar un bit extra a la entrada *"Aluop"* de la ALU Control y la salida *"ID_AluOp"* de la Unidad de Control. En la ALU no hubo cambios ya que las operaciones se implementaron previamente. Ver *imagen 9*

```
LogicaAluControl: process (funct, Aluop)
begin
    case(Aluop) is
        when "010" => --Tipo R
            case funct is
                when "100000" => signalAlu <= "0010"; --ADD
                when "100010" => signalAlu <= "0011"; --SUB
                when "100100" => signalAlu <= "0000"; --AND
                when "100101" => signalAlu <= "0001"; --OR
                when "101010" => signalAlu <= "0111"; --SLT
                when "000000" => signalAlu <= "0110"; --SLL
                when "000010" => signalAlu <= "1000"; --SRL
                when "100110" => signalAlu <= "1001"; --XOR
                when "100111" => signalAlu <= "1010"; --NOR
                when others => signalAlu <= "0101";
            end case;
        when "000" => signalAlu <= "0010"; --LW/SW/Addi
        when "001" => signalAlu <= "0100"; --LUI
        when "011" => signalAlu <= "0000"; --ANDI
        when "100" => signalAlu <= "0001"; --ORI
        when "101" => signalAlu <= "1001"; --XORI
        when others => signalAlu <= "0101";
    end case;
end process;
```

Imagen 9: *En rojo se subrayan las nuevas operaciones que puede decodificar la ALU Control.*

Riesgos RAW en el programa:

1) Durante la prueba del procesador a partir del programa dado por la cátedra se identificó el riesgo RAW presente entre las instrucciones SLT y BEQ.

slt **$s7**, $t1, $t2 // s7 = 1
slt $t8, $s0, $t2 // t8 = 0
beq $t1, **$s7**, salto1

Se desarrollaron dos soluciones posibles a este problema:

- Agregar una instrucción NOP en el programa:
  slt **$s7**, $t1, $t2 // s7 = 1
  slt $t8, $s0, $t2 // t8 = 0
  NOP (x00000000)
  beq $t1, **$s7**, salto1
- Utilizar la Unidad de Hazard. El hardware se encarga de detectar el riesgo RAW y el salto para poder actuar al respecto introduciendo una instrucción NOP, bloqueando la escritura del PC y del registro de segmentación IF/ID.

Se probaron ambas soluciones y funcionaron exitosamente.

2) Otro riesgo RAW encontrados en el programa “program1” fue el siguiente:

*add **$t1**, $t1, $t1*
*salto1: beq **$t1**, $zero, salto2*

Durante la ejecución no hubo problema ya que debido a que el salto 1 siempre es efectivo la instrucción que hace la operación add nunca se efectúa, aun así,es de utilidad identificarlo.
En caso de querer solucionarlo se podrían agregar dos instrucciones NOP o dejar que la unidad de Hazard y Forwarding lo resuelvan en ejecución.
Optando por la primer opción el programa quedaría:

*add **$t1**, $t1, $t1*
NOP (x00000000)
NOP (x00000000)
*salto1: beq **$t1**, $zero, salto2*

Programas de prueba:

A continuación se adjuntan algunos programas de prueba utilizados para comprobar el funcionamiento de los agregados adicionales:

Referencias → *Registro RS* *Registro RT* *Registro RD*

Prueba de detección de riesgo insalvable:

8c090000 → 100011 **00000** ***01001*** 0000000000000000 // lw **$t1**, 0($zero) // t1 = 1
01205020 → 000000 ***01001*** **00000** **01010** 00000100000 // add $t2, **$t1**, $zero // t2 =1

Prueba de Jump:

8c09000c → 100011 **00000** **01001** 0000000000001100 // lw $t1, 12($zero) // t1 = 8
08000004 → 000010 00000 00000 0000000000000100 // j salto
8c090000 → 100011 **00000** **01001** 0000000000000000 // lw $t1, 0($zero) // t1 = 1
ac090004 → 101011 **00000** **01001** 0000000000000100 // sw $t1, 2($zero) // M(2) = 1
01208020 → 000000 **01001** **00000** **10000** 00000 100000 // add $s0, $t1, $zero // s0 = 1

Prueba de Unidad de Forwarding e instrucciones SLL y SRL :

8c090000 → 100011 **00000** **01001** 0000000000000000 / /lw $t1, 0($zero) // t1 = 1
00094940 → 000000 **00000** **01001** **01001** 00101 000000 // sll $t1, $t1, 5 // t1 << 5
00094902 → 000000 **00000** **01001** **01001** 00100 000010 // srl $t1, $t1, 4 // t1 >> 4

Prueba de instrucción ADDI y programa con Loop:

20090008 → 001000 **00000** **01001** 0000000000001000 // addi $t1, $zero, 8 // t1 = 8
00005020 → 000000 **00000** **00000** **01010** 00000100000 // add $t2, $zero, $zero // t2 = 0
00005820 → 000000 **00000** **00000** **01011** 00000100000 // add $t3, $zero, $zero // t3 = 0
112a0003 → 000100 **01001** **01010** 0000000000000011 // loop: beq $t1, $t2, done // t1 = t2?
216b0002 → 001000 **01011** **01011** 0000000000000010 // addi $t3, $t3, 2 // t3 = t3 + 2
214a0001 → 001000 **01010** **01010** 0000000000000001 // addi $t2, $t2, 1 // t2 = t2 + 1
08000003 → 000010 00000 00000 0000000000000011 // j loop
ac0b0000 → 101011 **00000** **01011** 0000000000000000 // done: sw $t3, 0($zero) // M(0) = t3
8c0c0000 → 100011 **00000** **01100** 0000000000000000 // lw $t4, 0($zero) // t4 = M(0)

Programa de instrucción ANDI, ORI, XORI:

8c090000 → 100011 **00000** **01001** 0000000000000000 // lw $t1, 0($zero) // t1 = 1
312a0001 → 001100 **01001** **01010** 0000000000000001 // andi $t2, $t1, 1 // t2 = 1
352b0002 → 001101 **01001** **01011** 0000000000000010 // ori $t3, $t1, 2 // t3 = 3
392c0003 → 001110 **01001** 01100 0000000000000011 // xori $t4, $t1, 3 // t4 = 2

Conclusión:

Como conclusión se puede remarcar la importancia de este trabajo para el completo entendimiento del procesador MIPS segmentado. El poder probarlo con programas conllevó a enfrentarse a diferentes problemas que complementaron la comprensión del funcionamiento del hardware. Además, la puesta en práctica de la teoría y la construcción etapa por etapa del mismo permitió un análisis óptimo de la arquitectura. Por último, fue de gran valor el aprender a utilizar una nueva herramienta como es VHDL para describir este tipo de hardware.