**Universidad Nacional de Córdoba.**

**Facultad de Ciencias Exactas, Físicas y Naturales.**


Arquitectura de Computadoras

---

# Trabajo Práctico N° 3:

# *BIP - Basic Instruction-set Processor*

## Integrantes:

- Drudi, Leandro
- Goldman, Nicolás.

**Github repository:** https://github.com/nicolasgoldman07/TPs-Arquitectura

# INTRODUCCIÓN:

El trabajo práctico N° 3 de la materia, consiste en realizar los módulos correspondientes a un procesador básico, capaz de ejecutar algunas operaciones *simples* de todo procesador.
El procesador se realizó siguiendo el modelo desarrollado por algunos alumnos de la universidad de Vale do Itajaí, UNIVALI, Brasil.
**Paper: Link**

Este procesador permite darnos una idea de como funciona cualquier procesador que encontramos en la actualidad, realizando las tareas de carga, store, suma, resta, etc. para poder realizar un correcto manejo sobre los registros del mismo.

Para evitar toda la explicación realizada en el paper, se detallarán los módulos principales realizados por los alumnos. En la siguiente imagen se puede ver básicamente el esquema de nuestro BIP 1.

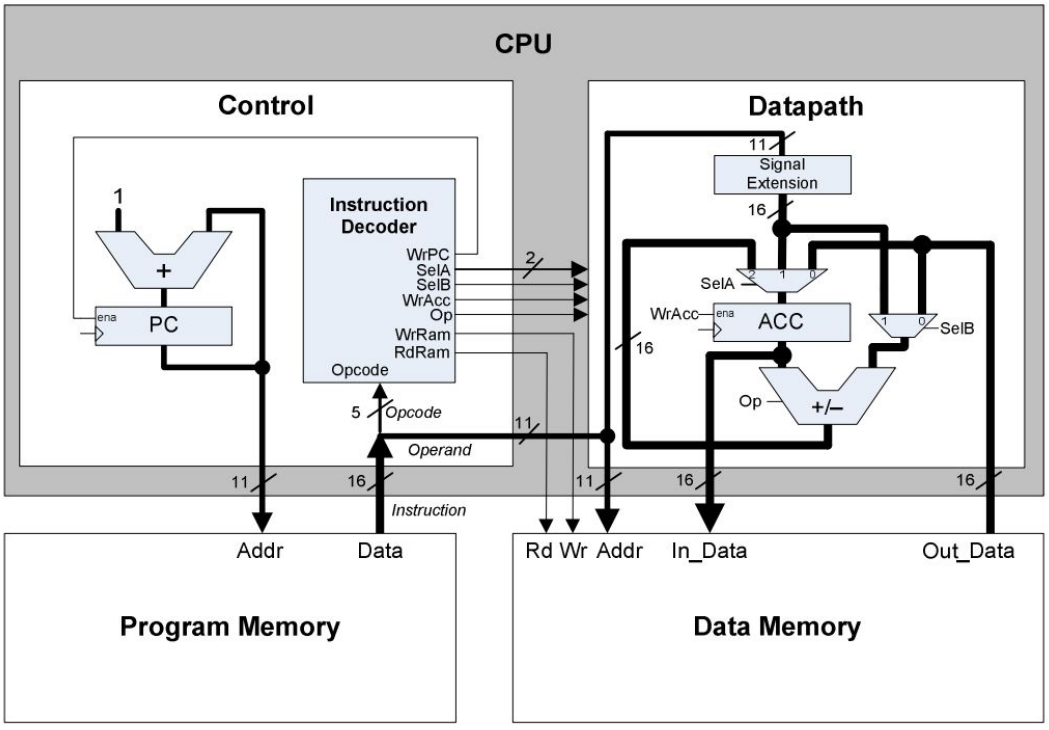



Figure 3. BIP I organization

Para llevarlo a cabo se decidió realizar un módulo para cada uno de los componentes de nuestro esquema; de esta forma se logró reducir la complejidad del sistema dividiendo en pequeños módulos todo el procesador. Luego tendremos un módulo de top_level tanto para la sección de Control como para Datapath, y por último otro nivel más arriba para la CPU.

Las memorias se implementaron haciendo uso de las herramientas dedicadas para ello en el software VIVADO.

# DESARROLLO:

### sum1.v y pc.v MODULES:

El módulo del **program counter** es extremadamente simple. Cada vez que nos llega un pulso de clock, de encontrarse el enable en 1, nos llevará la salida del contador a la entrada del siguiente módulo encargado de aumentarnos dicho PC en 1.

```
always @(negedge i_clk)
  begin
    if (i_reset)
      begin
        RegR <= 11'b00000000000;
      end
    else if (i_WrPC)
      begin
        RegR <= i_sum1;
      end
  end

assign o_PC = RegR;
```

Lógica del PC

```
  //    ENTRADAS:
  input       [N_BUS_IN-1:0]    i_pc,

  //    SALIDAS:
  output      [N_BUS_IN-1:0]    o_sum1
);

  reg         [N_BUS_IN-1:0]    RegR;

  always @(*)
    begin
          RegR <= i_pc + 1;
    end

  assign o_sum1 = RegR;
```

Lógica del sumador 1

Podemos ver también la imágen de la lógica que implementa el módulo encargado de realizar la suma para movernos a la próxima instrucción. Es muy simple ya que podemos ver que cuenta con la entrada del Program Counter Module, **i_pc**, y apenas obtenemos un valor de dicha entrada, le sumamos 1 y lo sacamos por la única salida del módulo, **o_sum1**.

### instr_deco.v MODULE:

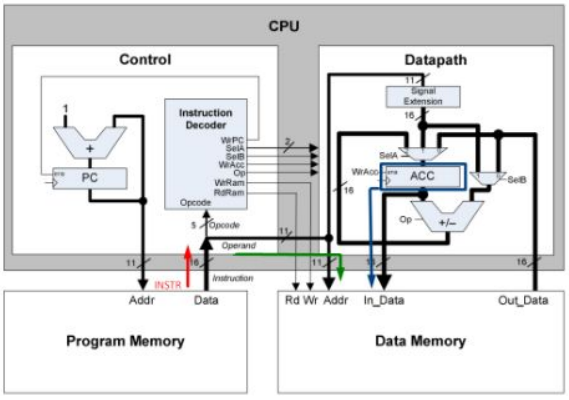
Este es el módulo *más complejo* de la sección de control. Es el encargado de controlar las banderas de los diferentes módulos del procesador para activarlos o desactivarlos cuando corresponda.
Para implementarlo, la lógica es muy simple. Mediante un ***case*** analizamos la variable de entrada cuyo valor es el código de operación correspondiente a los 5 MSB de la instrucción proveniente de

la memoria de instrucciones. Dependiendo del valor de dicho case, pondremos valores de 1 y 0 en las variables que manejan el resto de los módulos.

A modo de ejemplificar lo explicado arriba, se mostrarán únicamente 2 casos de instrucciones (STO y ADDI). El resto de instrucciones se comportan de la misma manera con cambios en los valores seteados de estas banderas.

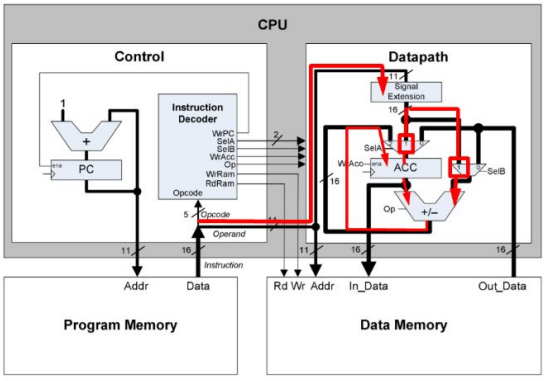
**Instruccion STO:**

Cuando nos llega una instrucción de store, debemos guardar lo que tenemos en nuestro ACC en la memoria de datos. Para ello debemos activar la bandera de la escritura en la memoria de datos y, como en todas las instrucciones excepto en HALT, activar la bandera del contador de programa para que el mismo aumente en 1 al finalizar.




```
//STO: WrPC=1,WrRam=1,RdRam=0,
5'b00001:
begin
                    WrPC=1;
                    SelA=2'b00;
                    SelB=0;
                    WrAcc=0;
                    Op=0;
                    WrRam=1;
                    RdRam=0;
end
```

**Instruccion ADDI:**




```
//ADDI: WrPC=1,WrRam=0,RdRam=0,SelA=2'b10,SelB=1,Op=1,WrAcc=1
5'b00101:
begin
                    WrPC=1;
                    SelA=2'b10;
                    SelB=1;
                    WrAcc=1;
                    Op=1;
                    WrRam=0;
                    RdRam=0;
end
```

En este caso podemos ver que la bandera de escritura en la memoria se desactiva ya que no se requiere guardar nada en ella. Tenemos el selector B en 1 para tomar el dato que llega del signal extension y no de memoria, la escritura en ACC activada para guardar el nuevo resultado, y el selector A en 2 para poder realizar también esto último.

### mult_A.v y mult_B.v MODULE:

Los módulos mult_A y mult_B corresponden a los multiplexores ubicados en la sección encargada de procesar los datos enviados y comandados por el sector de control.

La función de estos multiplexores es muy simple, ya que solo consiste en asignar a su salida correspondiente, alguna de las entradas que recibe. La decisión de cuál entrada redireccionar se tomará en base a otra señal que proviene del sector de control, la cual, de acuerdo al valor que tenga, será la entrada que el multiplexor deberá mostrar en su output.

```
always @(*)
    begin
        case (i_selA)
            2'b00   :   RegSelected = i_DATA;
            2'b01   :   RegSelected = i_SIGNAL;
            2'b10   :   RegSelected = i_RES_ARIT;
        endcase
    end

assign o_MUL_A = RegSelected;
```

```
always @(*)
    begin
        if (i_selB) begin
            RegSelected = i_SIGNAL;
        end else begin
            RegSelected = i_DATA;
        end
    end

assign o_MUL_B = RegSelected;
```

### arit_unit.v MODULE:

Por otro lado, nos encontramos con el módulo de la unidad aritmética propiamente dicha. Es la encargada de realizar las operaciones correspondientes entre los operandos que recibe en sus entradas. En este caso sólo recibe dos operandos de 16 bits y una señal de operación; ésta última indicará si la operación será una suma o una resta. Luego guarda el resultado en un registro que está asignado a la salida del módulo.

```
reg   signed  [N_BUS-1:0] RegR;

always @(*)
    begin
      if (i_OP)
        begin
          RegR              <=          i_ACC - i_DATA;
        end
      else
        begin
          RegR              <=          i_ACC + i_DATA;
        end
    end

assign o_RES = RegR;
```

### signal_ext.v MODULE:

Este es uno de los módulos más simples debido a que su única funcionalidad es tomar la señal de entrada, es decir el dato u operando que envía explícitamente el módulo de Control, y convertirlo de 11 bits a 16 bits. Como se puede ver en el código, solo concatenamos 5 ceros a los bits más significativos.

```
reg       signed    [N_BUS-1:0]     RegR;

always @(*)
    begin
        RegR = {5'b00000, i_signal};
    end

assign o_signal = RegR;
```

### ACC.v MODULE:

```
reg       signed      [N_BUS-1:0] RegR;

always @(posedge i_clk)
  begin
    if (i_WrAcc)
      begin
        RegR <= i_MUL_A;
      end
  end

assign o_ACC = RegR;
```

El acumulador es el único bloque que responde a cambios del clock. Éste sólo muestra su entrada a su salida cuando se cumplen las siguientes dos condiciones: Por un lado el clock del sistema debe encontrarse en un flanco positivo; y a su vez, el Control debe habilitar una señal “WrAcc” que será la que indica si el acumulador debe actualizar su valor o no. Es decir que su valor a la salida cambia, solo si varía su entrada, ya que no depende de la misma.

### TOP_datapath.v MODULE:

Luego de definir todos estos pequeños módulos, nos encargamos de hacerlos funcionar en conjunto como un *gran módulo*. La función principal del TOP_datapath es instanciar todos estos módulos y hacer la conexiones entre los mismos.

Sus entradas son las señales que provienen del sector de Control, es decir que su funcionamiento dependerá de éste. También posee otra entrada correspondiente a los registros de los valores almacenados en la memoria de datos. Cada valor se obtiene a partir de la lectura del registro indicado por una dirección de memoria.

```
wire      [N_BUS-1:0]         large_signal;
wire      [N_BUS-1:0]         output_mul_b;
wire      [N_BUS-1:0]         output_mul_a;
wire      [N_BUS-1:0]         output_ACC;
wire      [N_BUS-1:0]         output_res_arit;

reg       [N_BUS_IN-1:0]      operand;
reg       [N_BUS-1:0]         to_memory;

always @(*)
begin
    operand     <=  i_signal;
    to_memory   <=  output_ACC;
end


assign o_Addr = operand;
assign o_In_Data = to_memory;
```

Sus salidas pueden ser la dirección de memoria de la que hablamos anteriormente, o puede ser también el valor presente en el acumulador. Esto quiere decir que para guardar un dato en la memoria, lo deberemos ubicar en el acumulador, y luego se procede a almacenarlo.

### TOP_cpu.v MODULE:

Este módulo orquesta los principales sectores del procesador, el de control y el de procesamiento de datos; es decir el procesador en sí mismo. Definiendo sus salidas y entradas para luego ser utilizado con las memorias que correspondan o con otros periféricos ajenos al procesador en sí mismo.

```
wire        [N_BUS_IN-1:0]      signal;
wire        [TAM-1:0]           selA;
wire                            selB;
wire                            WrAcc;
wire                            OP;


TOP_control#(.N_BUS(N_BUS), .N_BUS_OUT(N_BUS_IN),
    .N_OP(N_OP), .TAM(TAM))
    TOP_control_top(.i_clk(i_clk), .i_reset(i_reset),
        .i_instr(i_instr),
        .o_PC(o_PC),
        .o_signal(signal),
        .o_selA(selA),
        .o_selB(selB),
        .o_WrAcc(WrAcc),
        .o_OP(OP),
        .o_WrRam(o_WrRam),
        .o_RdRam(o_RdRam));
```

```
TOP_datapath#(.N_BUS(N_BUS), .N_BUS_IN(N_BUS_IN),
    .TAM(TAM))
    TOP_datapath_top(.i_clk(i_clk),
        .i_signal(signal),
        .i_Out_Data(i_Out_Data),
        .i_selA(selA),
        .i_selB(selB),
        .i_WrAcc(WrAcc),
        .i_OP(OP),
        .o_Addr(o_Addr), .o_In_Data(o_In_Data));
```

### Proc + Memory MODULE:

Por último, tenemos el encargado de modelar el procesador completo. Como vemos en el código, unifica las dos memorias con el núcleo del BIP asignando las salidas y entradas como corresponden.

Las entradas a este gran módulo son el Clock del sistema y el botón de Reset. Su salida fue configurada como un arreglo de led's en los cuales exhibimos la salida del acumulador.

```
TOP_cpu TOP_cpu_top(.i_clk(i_clk),
    .i_reset(i_reset),
    .i_instr(instr),
    .i_Out_Data(Out_Data),
    .o_PC(PC),
    .o_Addr(Addr),
    .o_In_Data(In_Data),
    .o_WrRam(WrRam)
    );

data_memory data_memory_top(.clka(~i_clk),
    .wea(WrRam),
    .addra(Addr),
    .dina(In_Data),
    .douta(Out_Data)
    );

program_memory program_memory_top(.clka(i_clk),
    .wea(),
    .addra(PC),
    .dina(),
    .douta(instr)
    );

assign o_led = In_Data;
```

### Testbench MODULES:

A medida que fuimos creando cada módulo, también realizamos sus correspondientes módulos de prueba. Ésto nos ahorró muchísimo trabajo y contratiempos porque encontramos pequeños errores que si no los hubiéramos solucionado; luego, en el momento de instanciar el procesador completo, llevaría el doble de tiempo encontrar y solucionar los mismos.

## CONCLUSIÓN:

Para terminar nuestro ensayo, podemos afirmar que fue uno de los trabajos más desafiantes hasta el momento; tuvimos que poner en práctica los conocimientos obtenidos en la materia, y también buscar información extra en los casos que no sabíamos resolver.

En cuanto al proyecto en sí, nos permitió entender o apreciar la forma de trabajo de un procesador simple y básico pero de una manera muy educativa y provechosa. La forma de implementarlo y explicarlo está muy bien lograda, con conceptos claves y entendibles; así como también esquemas completos y explicativos en sí mismos.

Haciendo referencia a nuestros conocimientos del tema, sentimos que avanzamos y afianzamos los mismos adquiridos en la teoría, que sin ésta práctica, no podrían haber sido alcanzados.