Hack and *Slash* CPU Design

short presentation by: siavoosh payandeh azad

How long does it take?

- **To hack everything in, test each part and put them together you need 10-12 hours of coding...**
- **3 sessions is 12 hours...**

How long does it take?

- **To hack everything in, test each part and put them together you need 10-12 hours of coding...**
- **3 sessions is 12 hours...**

so you should be fine...

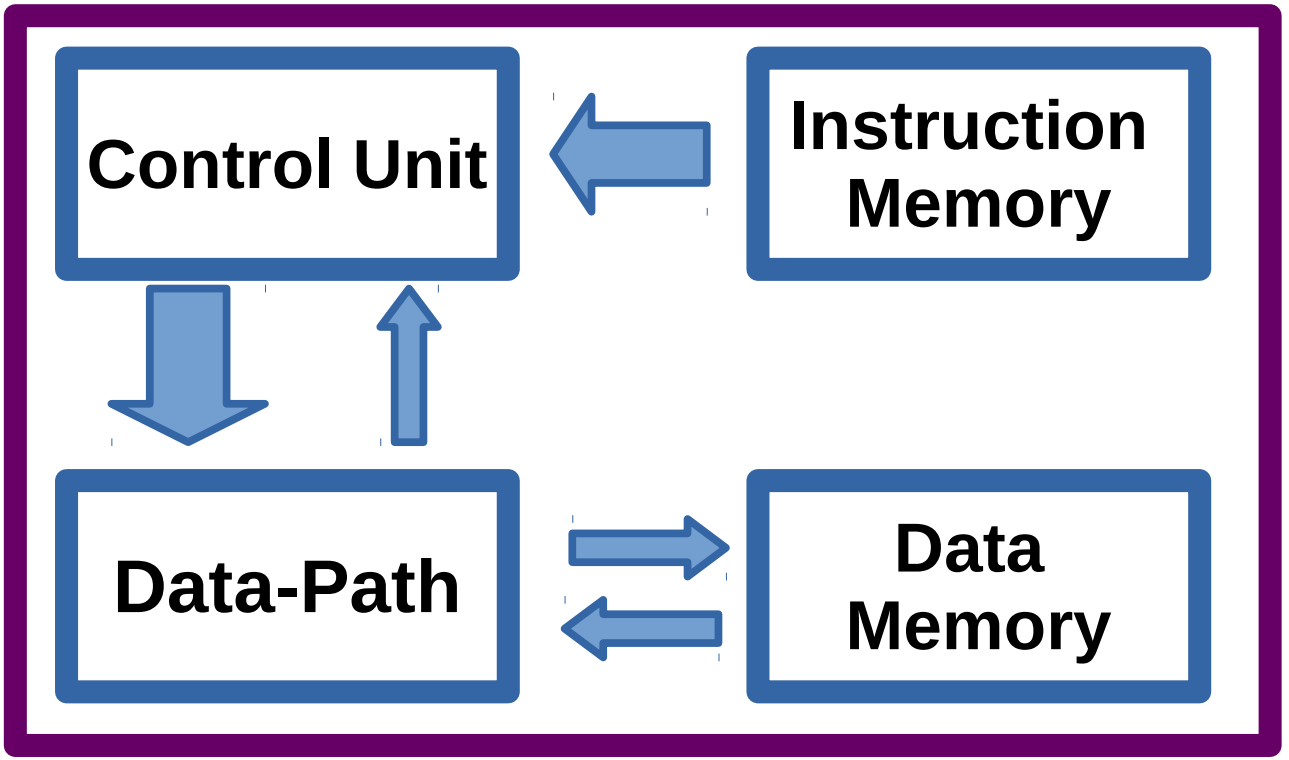
Where to start?

- Well we have to **define the problem** first
- We have to **choose our approach** to solve it

State the problem

- **We want a CPU that does these things:**
  - **Addition/Subtraction**
  - **Increment/Decrement**
  - **Arithmetic and Logical Shift**
  - **Bitwise AND, OR and XOR**
  - **Negation**
  - **Load/Store**
  - **Unconditional Branch (jump)**
  - **Branch if zero / Branch if Overflow**
  - **Set/Clear Registers/Flags**
  - **NOP/HALT**

What are the parts?



What do we have?

- 8 bits Operands
- 16 Bits of Instruction (5 bit opcode = 32 different instructions)
- 1 Accumulator
- 1 general purpose register

What do we have?

- 8 bits Operands
- 16 Bits of Instruction (5 bit opcode = 32 different instructions)
- 1 Accumulator
- 1 general purpose register
- **No fancy stuff...**

INSTRUCTION SET

Instruction	Operand
Add_A_B	----
Add_A_Mem	Memory Address
Sub_A_B	----
Sub_A_Mem	Memory Address
IncA	----
DecA	----
ShiftA_R	----
ShiftA_L	----
And_A_B	----
OR_A_B	----
XOR_A_B;	----
NegA	----
Set/ Clear Z	----
Set/ Clear OV	----

INSTRUCTION SET

Instruction	Operand
Add_A_B	----
Add_A_Mem	Memory Address
Sub_A_B	----
Sub_A_Mem	Memory Address
IncA	----
DecA	----
ShiftA_R	----
ShiftA_L	----
And_A_B	----
OR_A_B	----
XOR_A_B;	----
NegA	----
Set/ Clear Z	----
Set/ Clear OV	----

Instruction	Operand
Load_Mem_B	Memory Address
Store_A_Mem	Memory Address

INSTRUCTION SET

Instruction	Operand
Add_A_B	----
Add_A_Mem	Memory Address
Sub_A_B	----
Sub_A_Mem	Memory Address
IncA	----
DecA	----
ShiftA_R	----
ShiftA_L	----
And_A_B	----
OR_A_B	----
XOR_A_B;	----
NegA	----
Set/ Clear Z	----
Set/ Clear OV	----

Instruction	Operand
Load_Mem_B	Memory Address
Store_A_Mem	Memory Address

Instruction	Operand
Jmp	Inst memory Address
Jmp_OV	Inst memory Address
Jmp_Z	Inst memory Address

INSTRUCTION SET

Instruction	Operand
Add_A_B	----
Add_A_Mem	Memory Address
Sub_A_B	----
Sub_A_Mem	Memory Address
IncA	----
DecA	----
ShiftA_R	----
ShiftA_L	----
And_A_B	----
OR_A_B	----
XOR_A_B;	----
NegA	----
Set/ Clear Z	----
Set/ Clear OV	----

Instruction	Operand
Load_Mem_B	Memory Address
Store_A_Mem	Memory Address

Instruction	Operand
Jmp	Inst memory Address
Jmp_OV	Inst memory Address
Jmp_Z	Inst memory Address

Instruction	Operand
NOP	-----
HALT	-----

INSTRUCTION SET

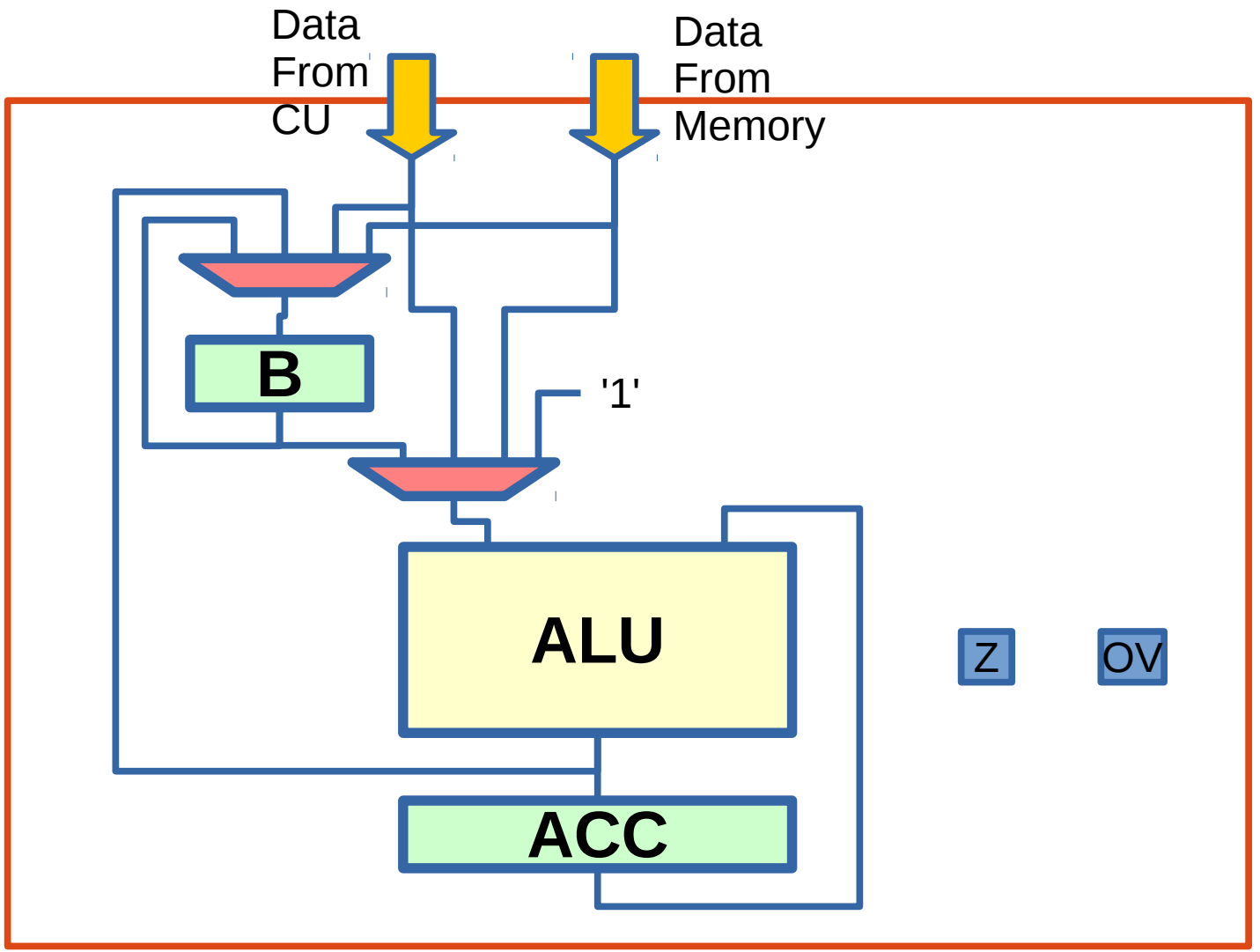
Instruction	Operand
Add_A_B	----
Add_A_Mem	Memory Address
Sub_A_B	----
Sub_A_Me	
IncA	
DecA	
ShiftA_R	
ShiftA_L	
And_A_B	
OR_A_B	
XOR_A_B;	----
NegA	----
Set/ Clear Z	----
Set/ Clear OV	----

Instruction	Operand
Load_Mem_B	Memory Address
Store_A_Mem	Memory Address
NOP	-----
HALT	-----

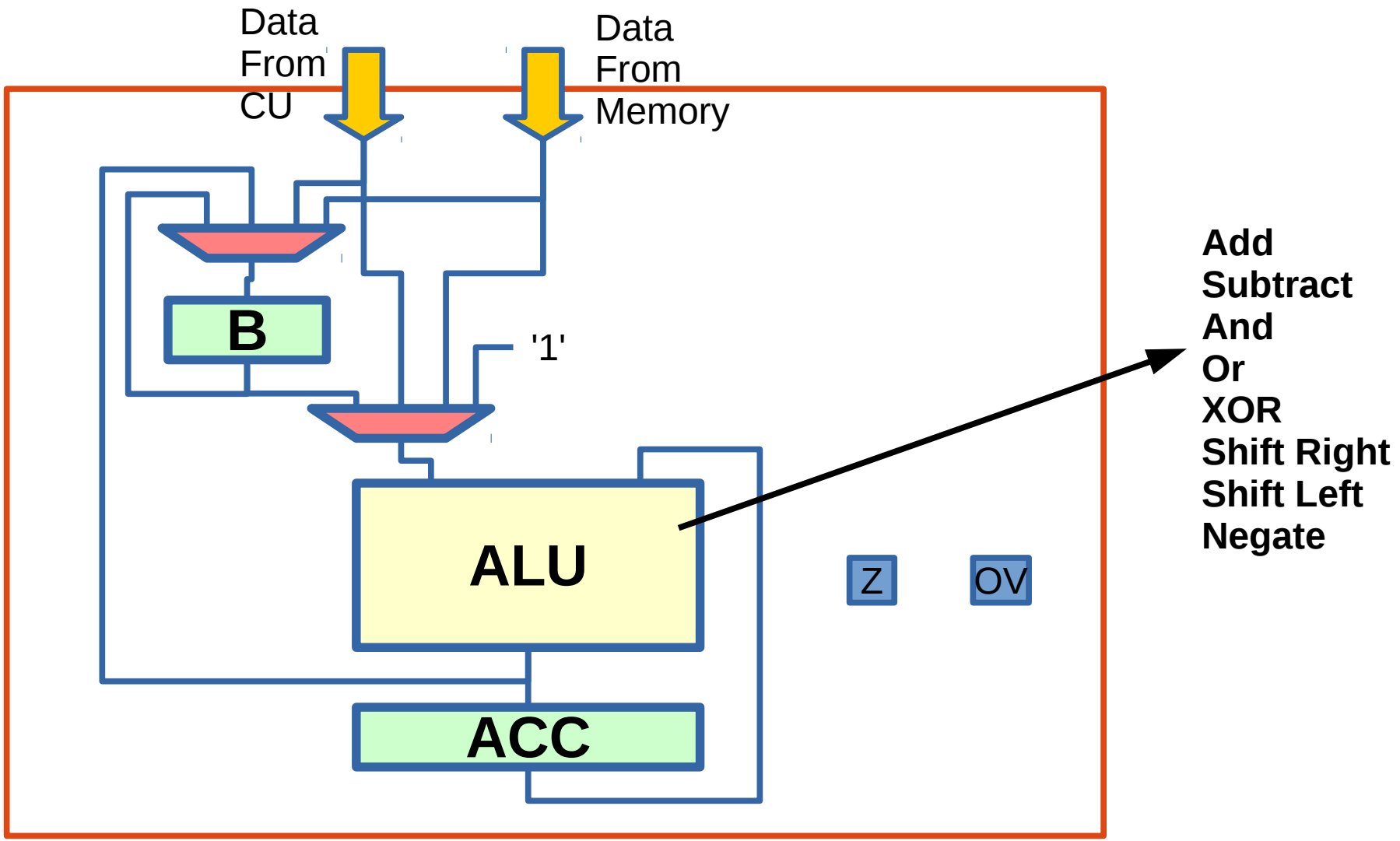
21 Instructions

INSTRUCTION SET

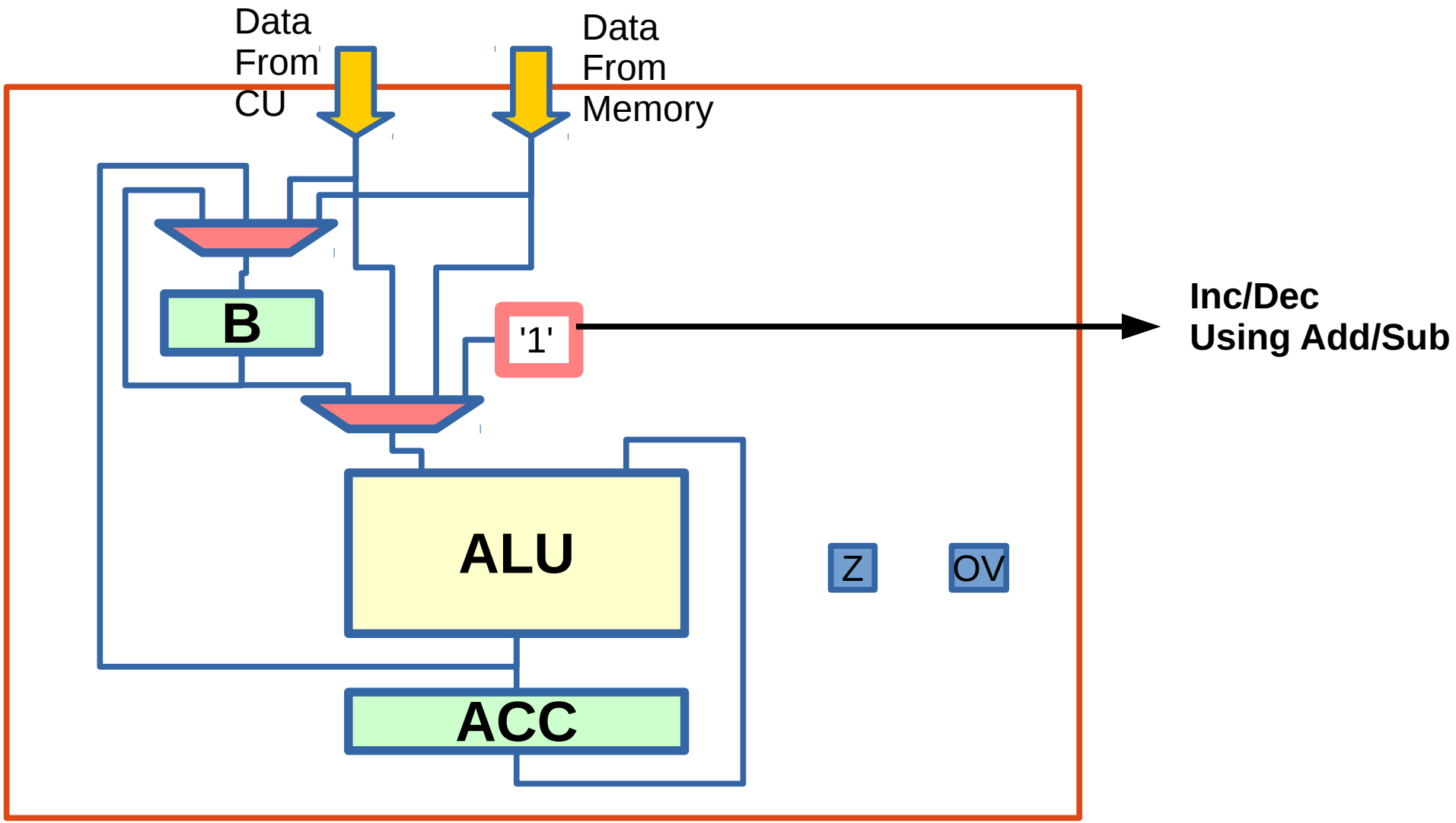
Starting from DPU



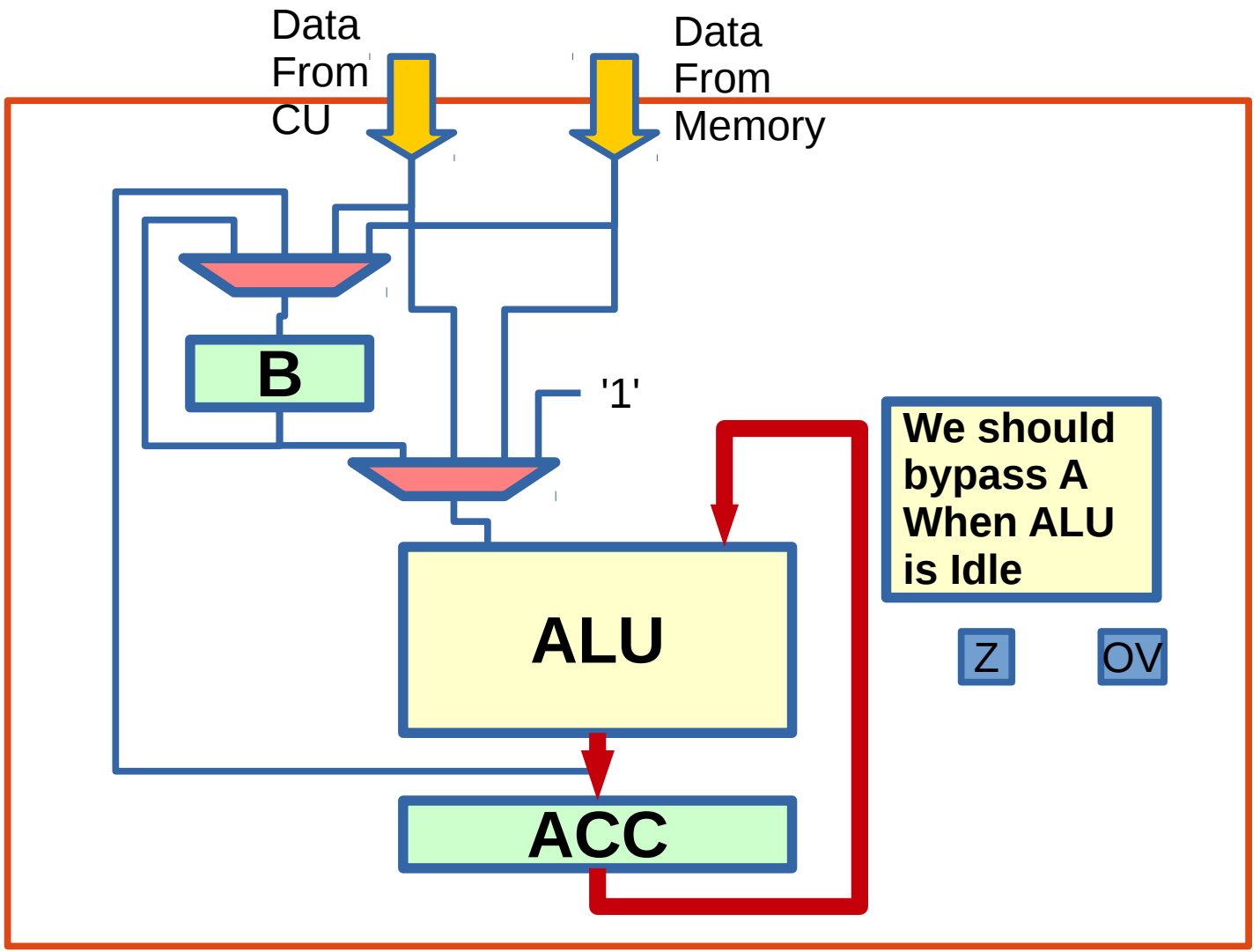
Starting from DPU



Starting from DPU



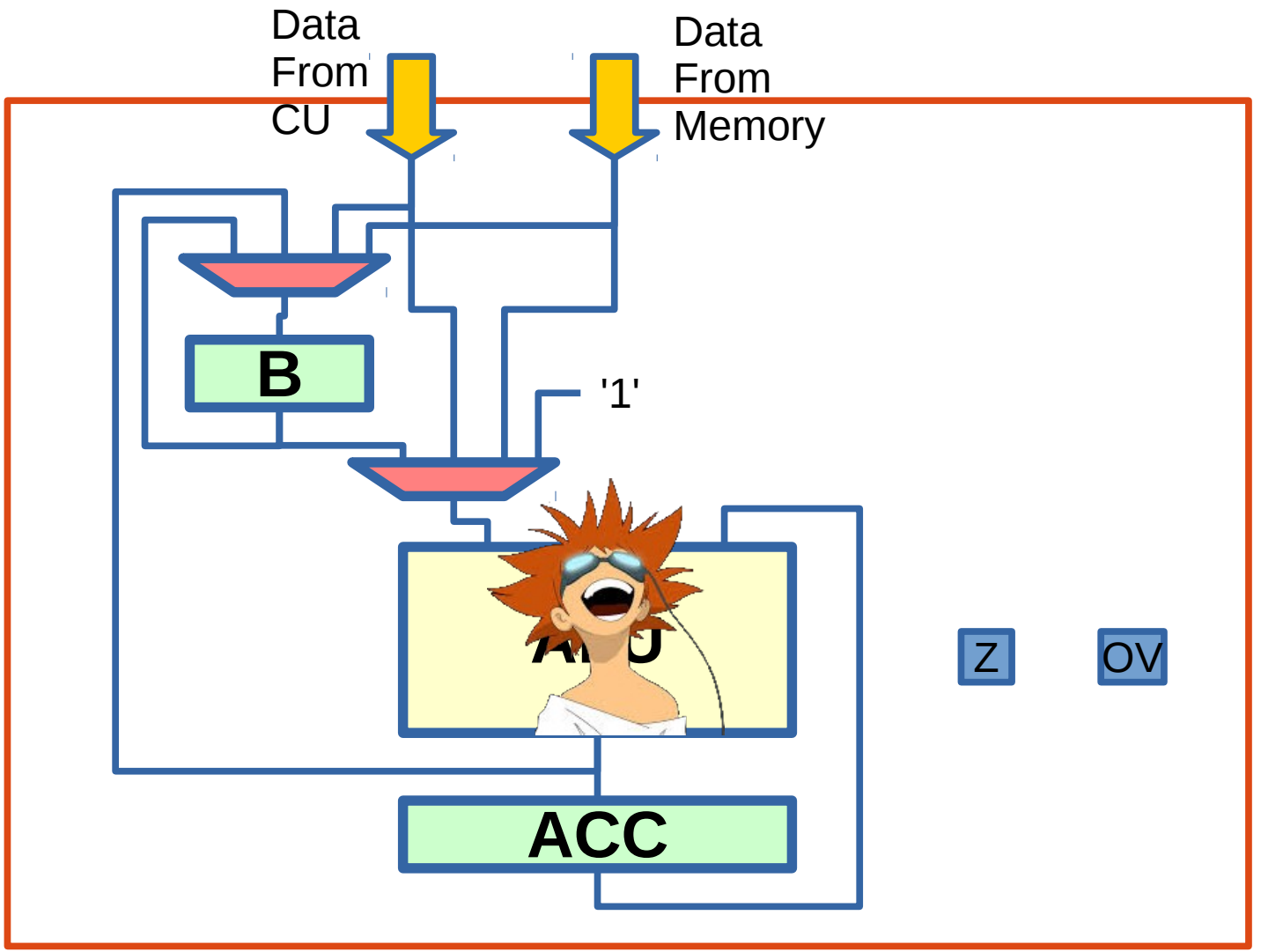
Starting from DPU



Lets break the ALU

```
PROC_ALU: process(Command,A,B)
  begin

    case Command is
        WHEN some condition =>      --add
        WHEN some condition =>      -- Subtract
                                    --Bypass A
                                    --Bypass B
      ....                          --And
                                    --or
                                    --xor
                                    --shift right
                                    --shift left
        WHEN some condition =>      --negation

     WHEN OTHERS => Result <= "00000000";
   END CASE;

end process PROC_ALU;
```

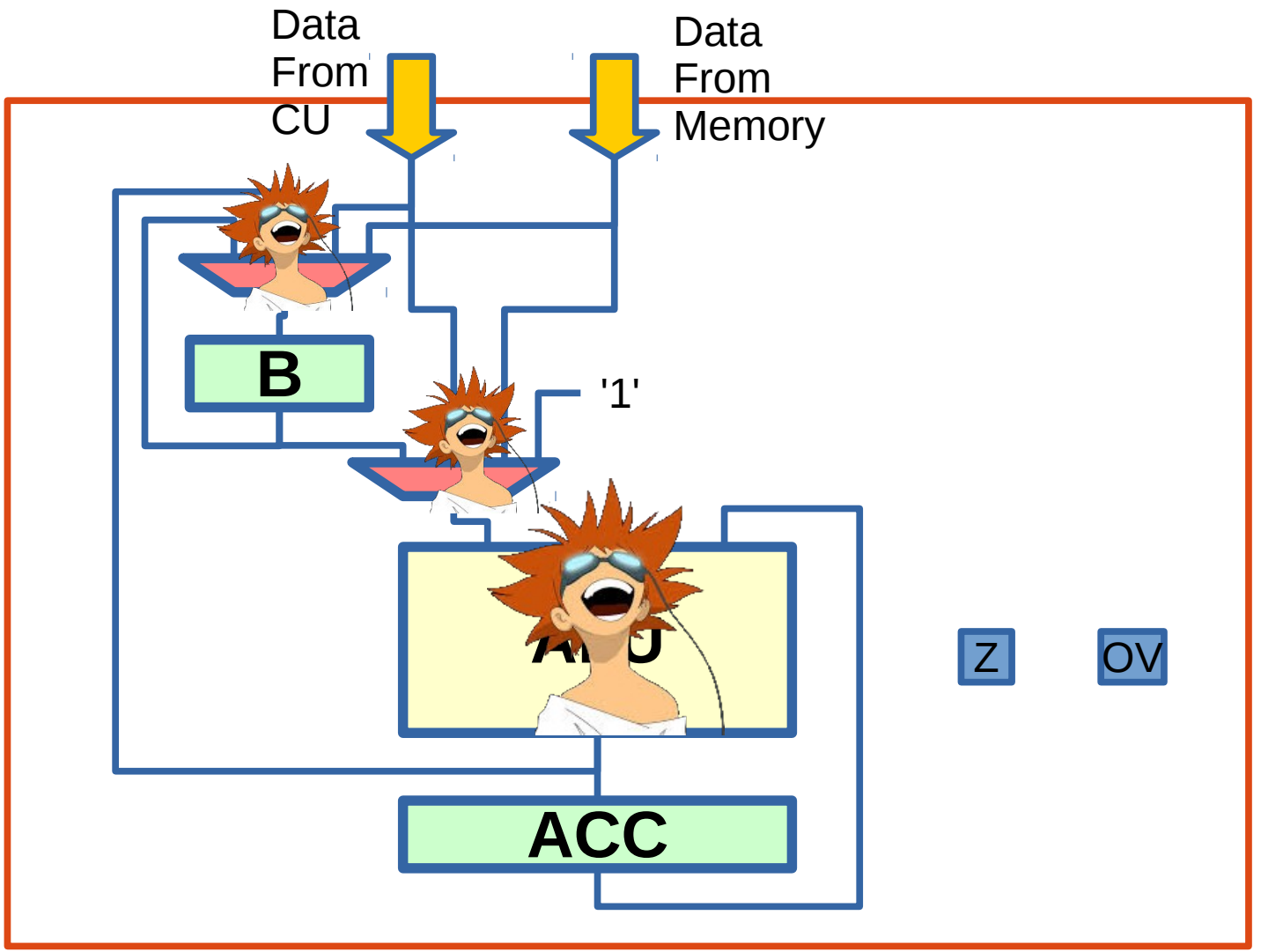
Starting from DPU



Multiplexers

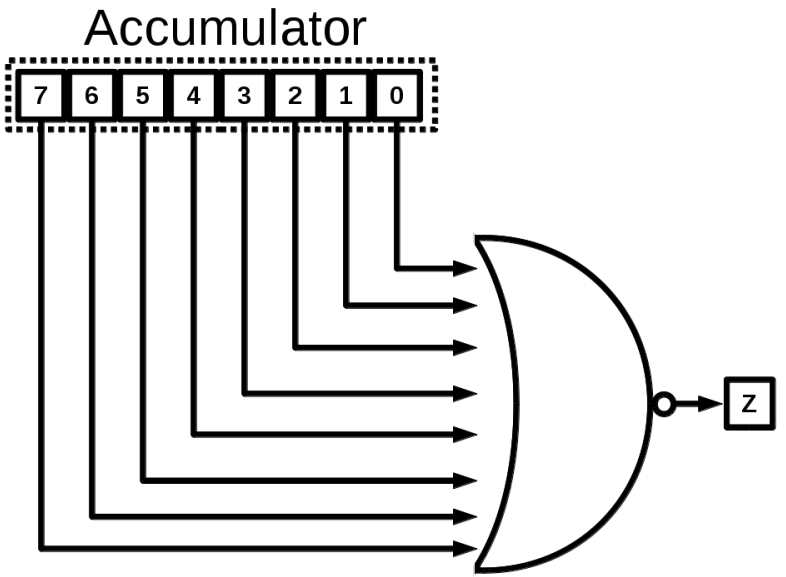
```
  process
(Data_in_mem,Data_in,B_out,Mux_Cont)
    begin
      case Mux_Cont is
        When "00" => ……
        when "01" => ……
        when "10" => ……
        when "11" => ……
      when others => ……
      end case;
  end process;
```

Starting from DPU

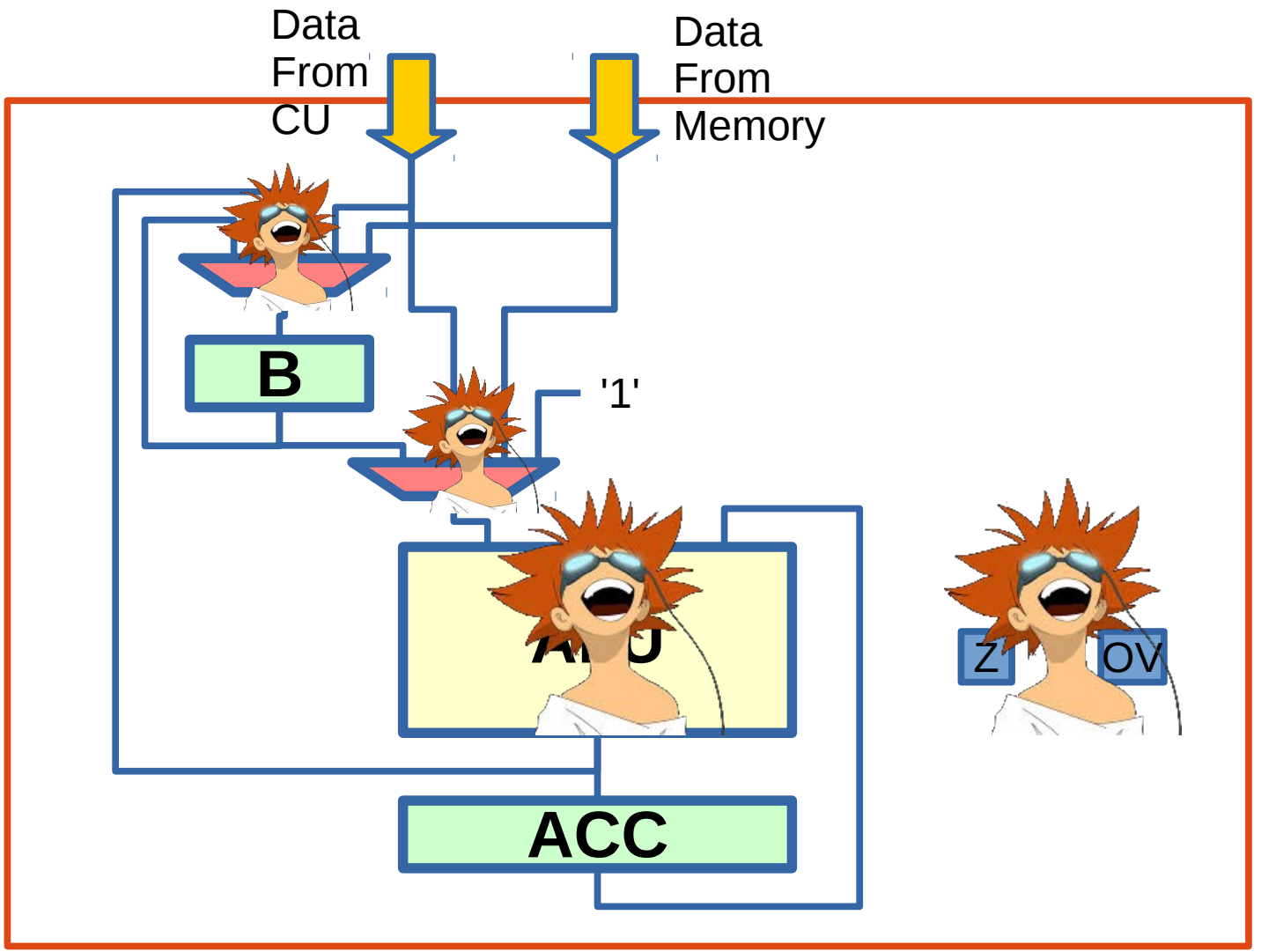


Flags

$$\mathrm{OV} = (\mathrm{A}[8] \;.\; \mathrm{B}[8] \;.\; \overline{R[8]}) \mid (\overline{A[8]} \;.\; \overline{B[8]} \;.\; \mathrm{R}[8])$$

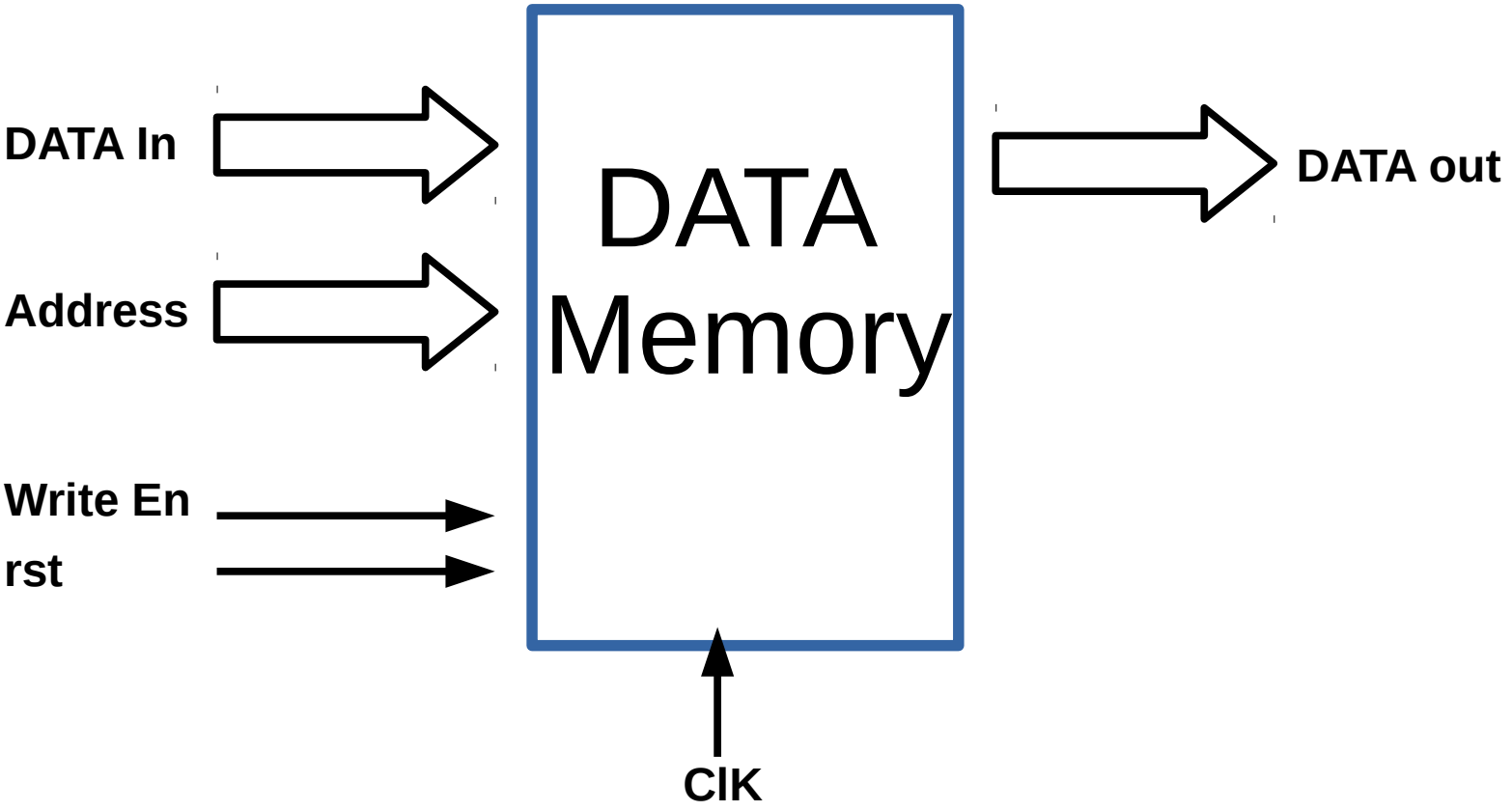


Starting from DPU



Test
it...

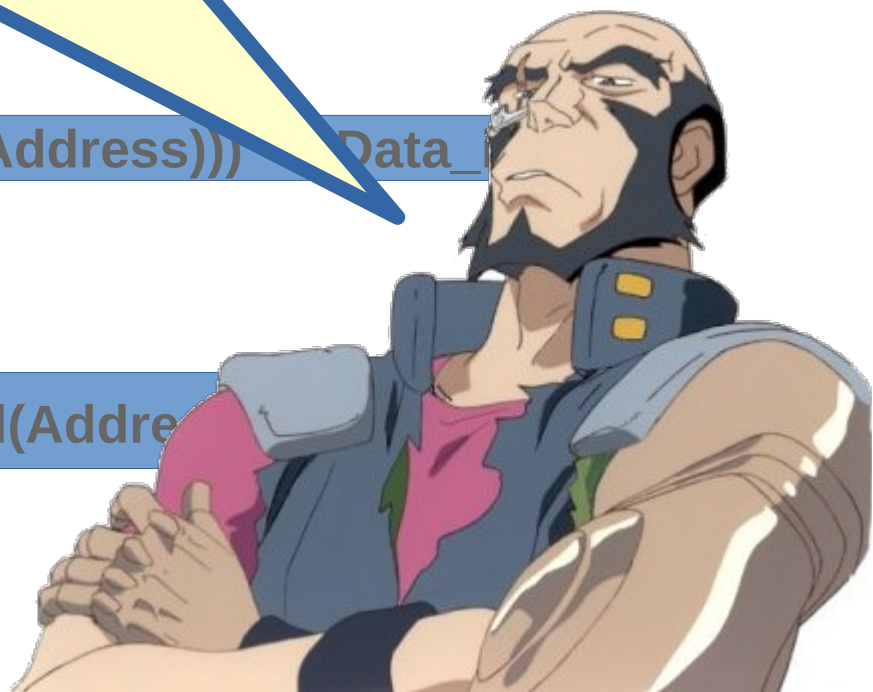
DATA Memory



```
architecture beh of Mem is

type Mem_type is array (0 to 255) of std_logic_vector(7 downto 0);
signal Mem : Mem_type;

begin
 MemProcess: process(clk,rst) is
  begin
        if rst = '1' then
              Mem <= ((others=> (others=>'0')));
        elsif rising_edge(clk) then
              if RW = '1' then
                    Mem(to_integer(unsigned(Address))) <= Data_in;
              end if;
        end if;
  end process MemProcess;

 Data_Out <= Mem(to_integer(unsigned(Address)));

end beh;
```

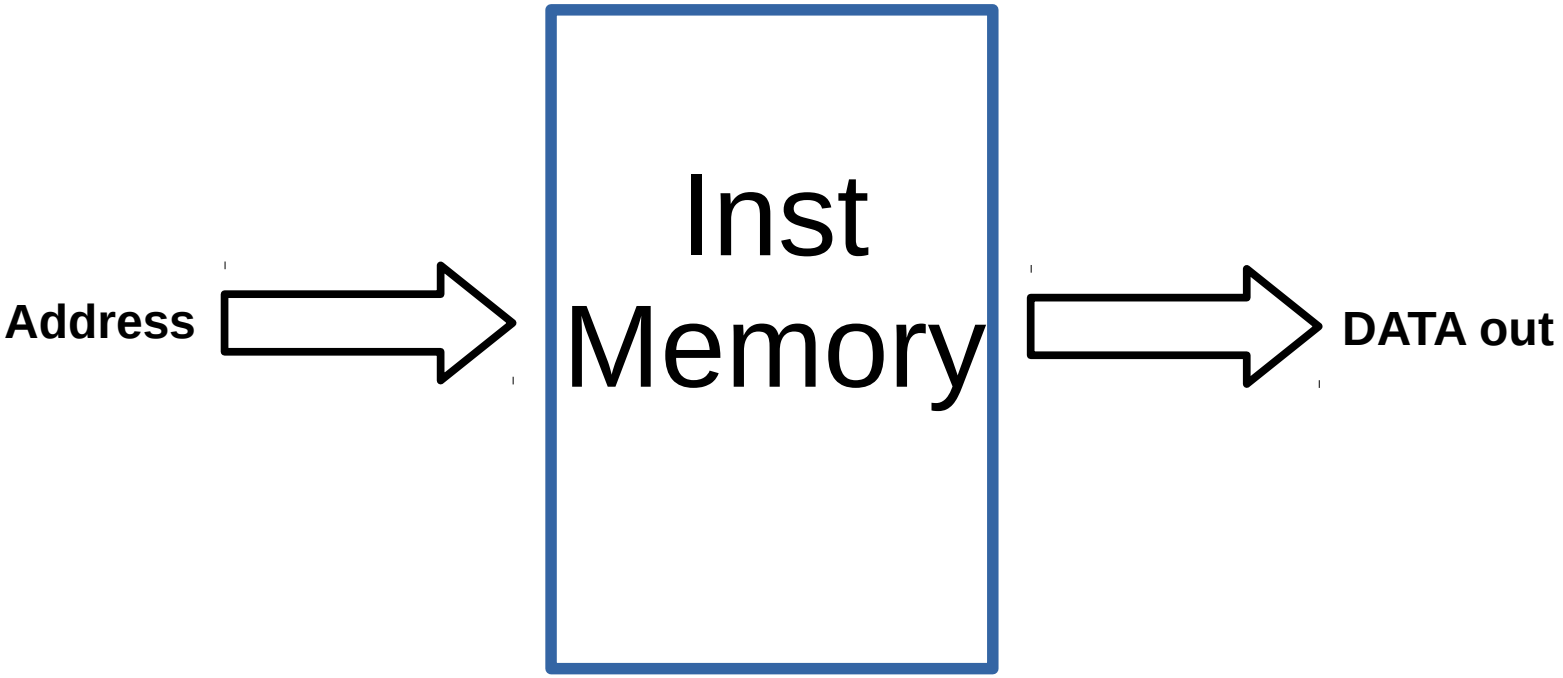
```
architecture beh of Mem is
  type Mem
  signal Me                                          (7 downto 0);

begin
 MemProces
  begin
        if rst = '1
              Mem <= ((others=> (others=> '0' )));
        elsif rising_edge(clk) then
              if RW = '1' then
                    Mem(to_integer(unsigned(Address)))      Data_
              end if;
        end if;
  end process MemProcess;

  Data_Out <= Mem(to_integer(unsigned(Addre

end beh;
```

Piece of cake



Test
it...

Instruction Memory

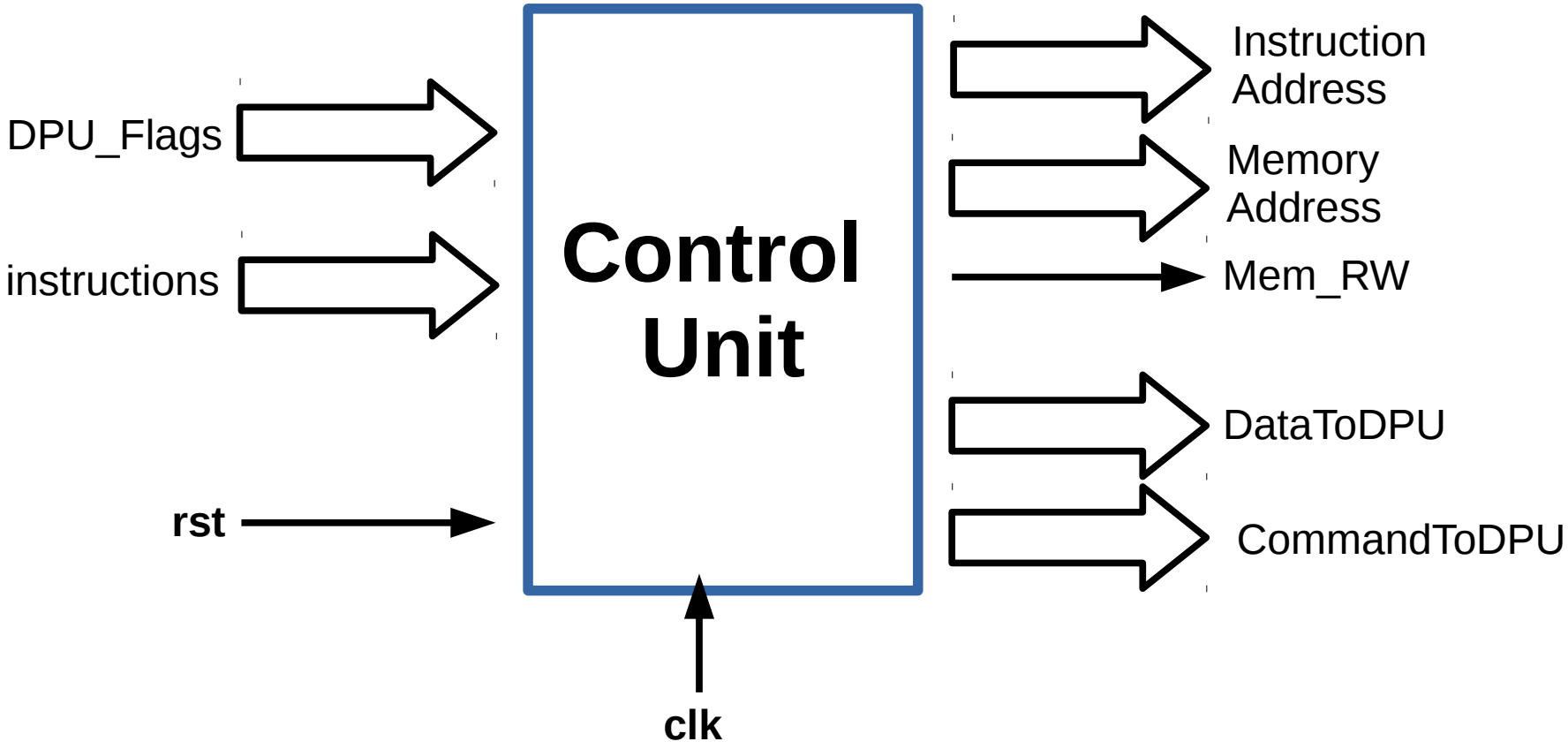


```
entity InstMem is
  port ( address : in std_logic_vector(7 downto 0);
         data : out std_logic_vector(15 downto 0) );
end entity InstMem;

architecture behavioral of InstMem is
      type mem is array ( 0 to 255) of std_logic_vector(15 downto 0);
          constant my_InstMem : mem := (
                              0  => instr 0
                              1  => instr 1
                              2  => instr 2
                              .......
                              254  => instr 254
                              255  => Halt
      );
begin
  data <= my_InstMem(to_integer(unsigned(address)));
end architecture behavioral;
```

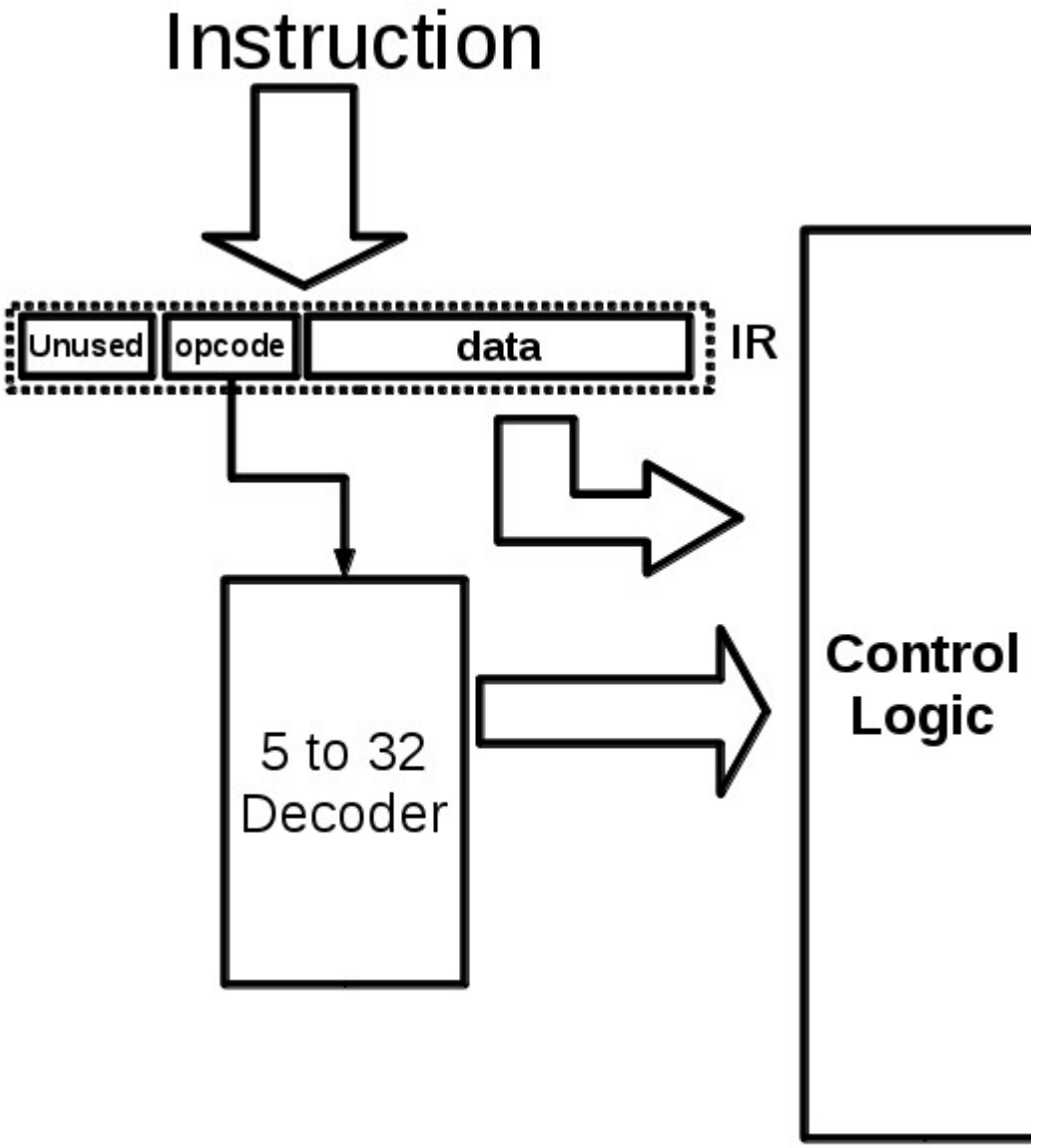
```
entity InstMem is
  port ( address : in std_
         data : out std_
end entity InstMe

architecture b
     type mem
          const
     );
begin
  data <= my_InstMem(to_integer(unsigned(address)));
end architecture behavioral;
```

Test
it...

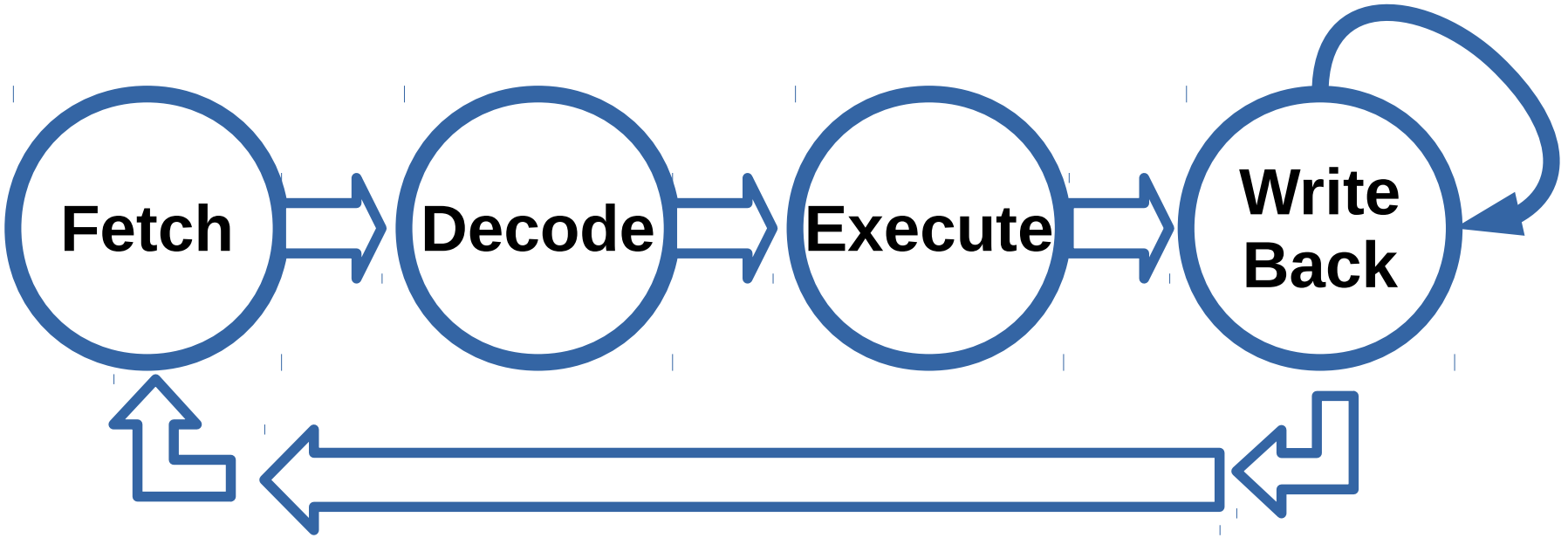
Control Unit



Block diagram



THE FSM



```
TYPE STATE_TYPE IS (Fetch, Decode, Execution,WriteBack);
Signal State_in, State_out :STATE_TYPE;
```

```
CASE State_out IS
        WHEN Fetch =>
            State_in <= Decode;
        WHEN Decode =>
            State_in <= Execution;
        WHEN Execution =>
            State_in <= WriteBack;
        WHEN WriteBack =>
            if Instr = HALT then
                State_in <= WriteBack;
            Else
                State_in <= Fetch;
             end if;
 END case;
```

Make Registers

```
-------------------------------------------
process (clk,rst)
  begin
  if rst = '1' then

     State_out <= Fetch;
     PC_out <= "00000000";
     InstrReg_out <= (others => '0');

  elsif clk'event and clk='1' then

     State_out <= State_in;
     PC_out <= PC_in;
     InstrReg_out <= InstrReg_in;

end if;
end process;
-------------------------------------------
```

Fetch

```
WHEN Fetch =>
        DataToDPU <= "00000000";
        MemAddress <= "00000000";
        CommandToDPU <=   --stay idle
        Instr_Add <= PC_out;
        Mem_RW <= '0';
        State_in <= Decode;
```

Decode

```
WHEN Decode =>
        DataToDPU <= "00000000";
        MemAddress <= "00000000";
        CommandToDPU <= --do not do anything
        State_in <= Execution;
```

Wait for instruction decoder to finish

Decoder...

```
TYPE Instruction IS ( Add_A_B,
Add_A_Mem, Sub_A_B, Sub_A_Mem,
IncA,DecA, ShiftA_R, ShiftA_L, And_A_B,
OR_A_B, XOR_A_B, Load_Mem_B,
LoadBControl, Store_A_Mem, Jmp, JmpZ,
JmpOV, SetZ, ClearZ,  SetOV, ClearOV,
NOP, HALT, NegA);

Signal Instr:Instruction;
```

```
-----------------------------------------
 process (opcode) -- Instr decoder
 begin
   case opcode is
     when "00000" => Instr <= Add_A_B;
     when "00001" => Instr <= Add_A_Mem;
     when "00010" => Instr <= Sub_A_B;
     when "00011" => Instr <= Sub_A_Mem;
     when "00100" => Instr <= IncA;
     when "00101" => Instr <= DecA;
    ……
     when "11000" => Instr <= NOP;
     when "11001" => Instr <= HALT;
     when "11010" => Instr <= NegA;
     when "11011" => Instr <= LoadBControl;
     when others =>  Instr <= NOP;
     end case;
       end process;
-----------------------------------------
```

Off with their heads (Execute)

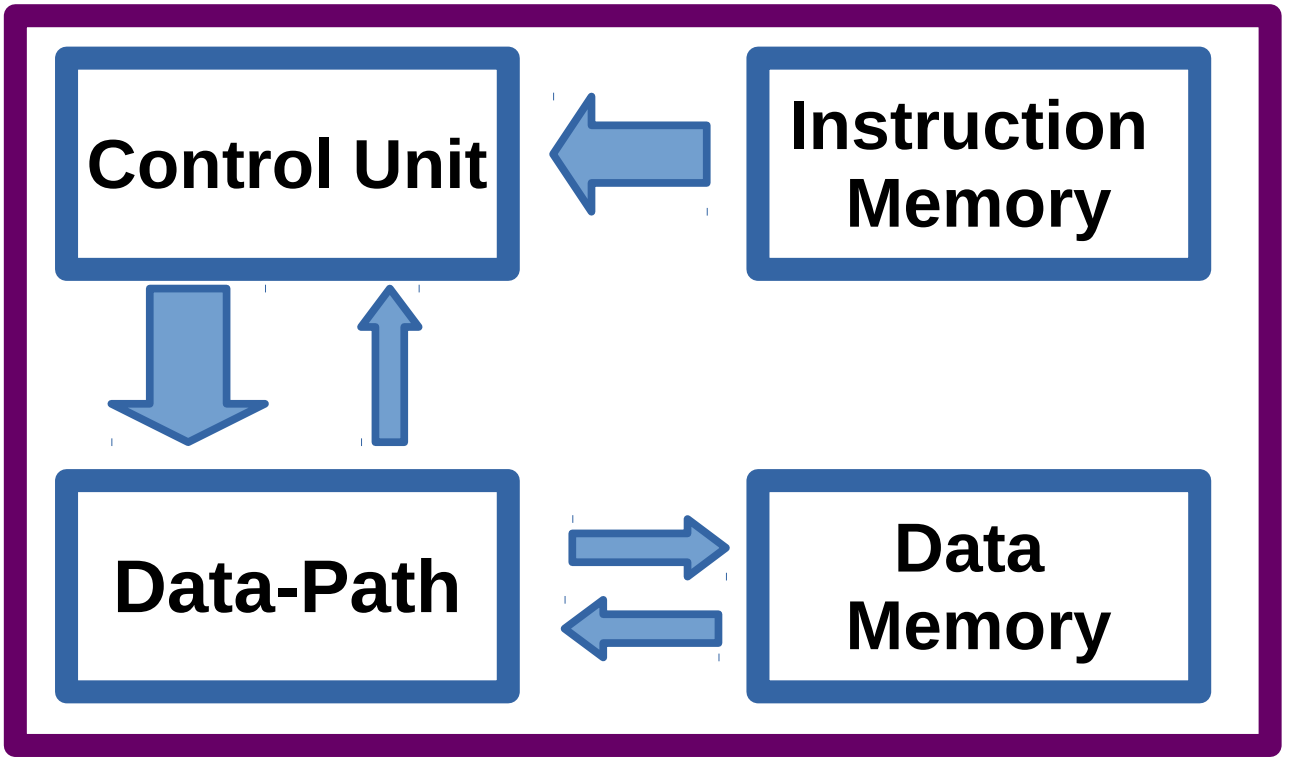
```
WHEN Execution =>
            DataToDPU <= "00000000";
            MemAddress <= "00000000";
            Mem_RW <= '0';

             if Instr = Add_A_B then
                  CommandToDPU <= – Add command
             elsif Instr = Add_A_Mem then
                  MemAddress <= – proper memory address
                  CommandToDPU <= – memory load command

                  …..

              elsif Instr = HALT then
                  CommandToDPU <=  --Stay Idle
              else
                  CommandToDPU <=  --Stay Idle
            end if;

            State_in <= WriteBack;
```

Write Back

```
WHEN WriteBack =>
        DataToDPU <= "00000000";
        MemAddress <= "00000000";
        CommandToDPU <=  --Stay Idle
    if Instr = Store_A_Mem then
        MemAddress <= – proper memory address
        Mem_RW <= '1';
        PC_in <= PC_out+1;
        State_in <= Fetch;
    elsif Instr = HALT then
        PC_in <= PC_out;
        State_in <= WriteBack;
    elsif Instr = Jmp then
        PC_in <= – proper memory address
        State_in <= Fetch;
    elsif Instr = JmpZ and DPU_Flags(2) = '1' then
        PC_in <= – proper memory address
        State_in <= Fetch;
    elsif Instr = JmpOV and DPU_Flags(0) = '1' then
        PC_in <= – proper memory address
        State_in <= Fetch;
    else
        PC_in <= PC_out+1;
        State_in <= Fetch;
    end if;
```

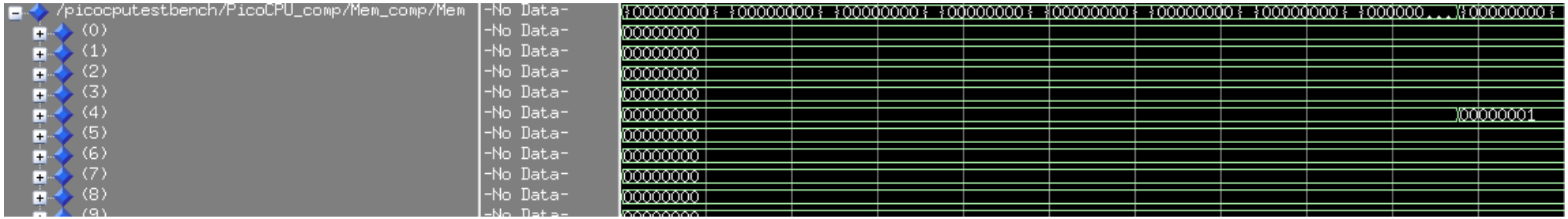
Test
it...

Now... lets put everything back together...



And run some program... :D

```
constant my_InstMem : mem := (
   0  => "000" & "11011" & "00011000", --load B with
   1  => "000" & "01001" & "00000000", --Or a b
   2  => "000" & "00100" & "00000000", --inc a
   3  => "000" & "00010" & "00000000", --sub a b
   4  => "000" & "01101" & "00000100", --Store A Mem
   5  => "000" & "11001" & "00000000"); --Halt
egin
```



And
Hacking
Begins...