Processeur Mono-Cycle: Simulation VHDL
Project Report

Stu. KANG Jiale , XIE Shifeng
No. 19022100087 , 19022100008

April 29, 2022

Report submitted for

Electronique numérique: VHDL

at the

School of Electronic Engineering, Xidian University,
Department of Electronics and Computer Sciences, Polytech Sorbonne.

Project Area: **Digital Electronics**
Project Supervisor: **Yann DOUZE , ZONG Ru**

Abstract

The objective of this project is to design and simulate a processor. This processor will be designed from basic blocks (registers, multiplexers, memory, ALU, $\cdots$) which will be combined to produce the different blocks of the system (processing unit, instruction management unit, control unit).

The processor will be validated by simulating the execution of a simple test program. For each block, it will be described in behavioral and simulated in language VHDL with Modelsim by developing a test bench.

Key Words: ARM architecture, VHDL, simulation, processor

Contents

1 Processing Unit (Unité de Traitement) 3
1.1 Arithmetic Logic Unit (Unité Arithmétique Logique) 3
1.2 Register File (Banc de Registres) 4
1.2.1 Design Register File 4
1.2.2 Assemble ALU and Register File 6
1.3 2 to 1 Multiplexer (Multiplexeur 2 vers 1) 7
1.4 Sign Extension (Extension de Signe) 8
1.5 Data Memory (Mémoire de Données) 9
1.6 Assemble Processing Unit (Assemblage Unité de Traitement) 10

2 Instruction Management Unit (Unité de gestion des instructions) 12
2.1 Design Instruction Management Unit 12
2.2 Simulation for Instruction Management Unit 14

3 Control Unit (Unité de Contrôle) 16
3.1 32-bit Register with Load Instruction (Registre 32-bit avec Commande de Chargement) 16
3.2 Instruction Decoder (Decodeur d'Instructions) 17

4 Assembly and Validation of Processor (Assemblage et Validation du Processeur) 22
4.1 Assemble Processor 22
4.2 Simulate Processor 25

5 Test Processor Completely (Test Complet du Processeur) 27

6 Increasing the Instruction Set (Augmentation du Jeu d'Instruction) 28

A Appendices 30

1 Processing Unit (Unité de Traitement)

1.1 Arithmetic Logic Unit (Unité Arithmétique Logique)

The Arithmetic Logic Unit (ALU) performs the internal arithmetic manipulation of data in the processor. The instructions that are read and executed by the processor control the data flow between the registers and the ALU. The instructions also control the arithmetic operations performed by the ALU via the ALU's control inputs. A symbolic representation of an ALU is shown in Figure 1.

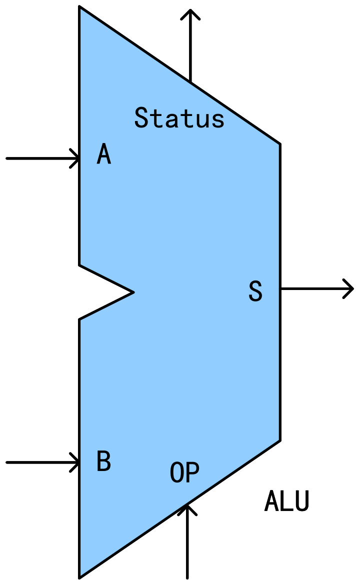


Figure 1: ALU Block Diagram

Where `A` and `B` are 32 bits input buses, `S` is a 32 bits output bus, `OP` is a 2 bits command signal, and `Status` is a Output Signal represent NVCZ (we just consider N here, so it is 1 bit).

For the operation of ALU, we can see from Table 1.

Thus, we set the inputs and outputs `ports` of the `entity` as follow:

```vhdl
entity ALU is
    port(
        op:in std_logic_vector(1 downto 0);
        a,b: in std_logic_vector(31 downto 0);
        s: out std_logic_vector(31 downto 0);
        n: out std_logic
    );
end entity;
```

Whenever instructed by the processor, the ALU performs an operation (we consider addition and subtraction, and the detailed table will be given below) on one or more values. These values, called operands , are typically obtained from two registers, or from one register and a memory location. The result of the operation is then placed back into a given destination register or memory location. The status outputs indicate any special attributes about the operation, such as whether the result was zero, negative, or if an overflow or carry occurred. [1]

Table 1: Operation Table

OP	S	Remark
00	S=A+B	ADD
01	S=B	B
10	S=A-B	SUB
11	S=A	A

Therefore, we build the `architecture` of ALU in **ALU.vhd**.

```
architecture behav of ALU is
    signal sign: std_logic_vector(31 downto 0);
begin

    process (a,b,op)
      begin
        case op is
            when"00" => sign <=std_logic_vector(signed(a)+signed(b));
            when"01" => sign <= b;
            when"10" => sign <=std_logic_vector(signed(a)-signed(b));
            when"11" => sign <= a;
            when others => sign <= a;
        end case ;
    end process;

    N <= sign(31);
    s <= sign;

end architecture;
```

1.2 Register File (Banc de Registres)

1.2.1 Design Register File

Registers are temporary storage locations inside the CPU that hold data and addresses.

The register file is the component that contains all the general purpose registers of the microprocessor. A few CPUs also place special registers such as the PC and the status register in the register file. Other CPUs keep them separate.[2]

A symbolic representation of an Register File is shown in Figure 2.

Where `Clk` is a clock Signal; `Rst` is a asynchrone reset signal which active at high level; `WE` is a enable signal of writing datas and `Rw` is a 4 bits address bus of writing register; `W`, `A` and `B` are 32 bits data buses; `Ra` and `Rb` are 4 bits address buses of reading register.

Thus, the `ports` of the `entity` we defined as follow:

```
entity Register_File is
   port (
      clk, rst, WE : in std_logic ;
      Ra, Rb, Rw : in std_logic_vector(3 downto 0);
      A, B : out std_logic_vector(31 downto 0);
      W : in std_logic_vector(31 downto 0)
   );
```

```
end entity;
```

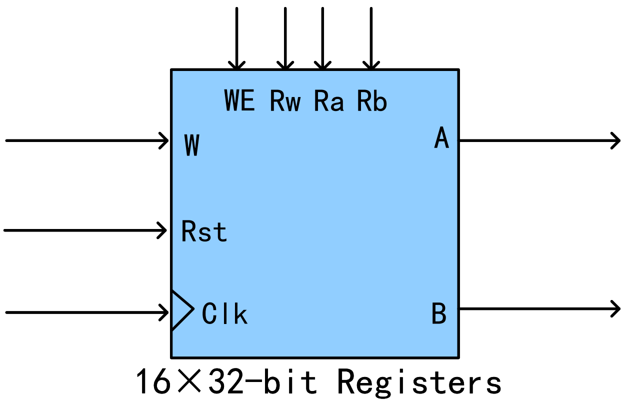


Figure 2: Register File Block Diagram

When `WE`= 1, which means *enabel to write*, so that we need to write data from bus `W` to the register on the address `Rw`; And when `WE`= 0, we will do nothing. As for reading, it will be done in a combinatorial and simultaneous way.

Therefore, we build the `architecture` of Register File in **Registre_File.vhd**.

```
architecture behav of Register_File is
    type matrix is array(15 downto 0) of std_logic_vector(31 downto 0);
    function init_banc return matrix is
        variable result : matrix;
    begin
        for i in 14 downto 0 loop
            result(i) := (others=>'0');
        end loop;
            result(15):=X"00000030";
        return result;
    end init_banc;

    signal Banc: matrix:=init_banc;
begin

    A <= Banc(to_integer(unsigned(Ra)));
    B <= Banc(to_integer(unsigned(Rb)));

    process(clk, rst)
    begin
        if rst ='1' then
            for i in 14 downto 0 loop
                Banc(i) <= (others=>'0');
            end loop;
                Banc(15)<=X"00000030";

        elsif rising_edge(clk) then
            if WE = '1' then
                Banc(to_integer(unsigned(Rw)))<=W;
            end if;
        end if;
    end process;

end architecture;
```

1.2.2 Assemble ALU and Register File

We will assemble these two component we have already finished as Figure 3.

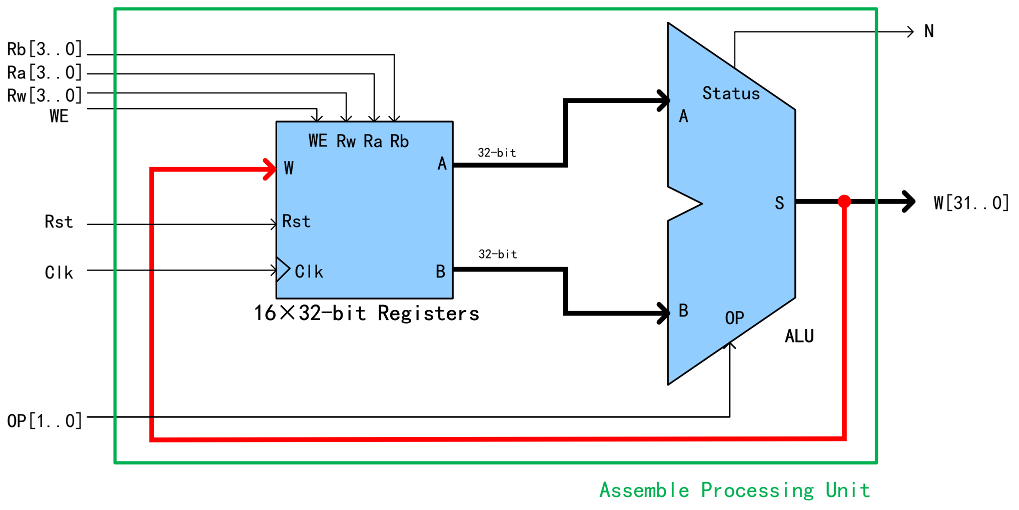


Figure 3: Assemble ALU and Register File Block Diagram

The `entity` and `architecture` of **Assemble_ALU_and_Register_File.vhd** shows below.

```
entity Assemble_ALU_and_Register_File is
    port (
        clk, rst, WE : in std_logic ;
        Ra, Rb, Rw : in std_logic_vector(3 downto 0);
        W : out std_logic_vector(31 downto 0);
        N : out std_logic;
        op : in std_logic_vector(1 downto 0)
    );
end entity;

architecture behave of Assemble_ALU_and_Register_File is
    signal busA,busB, busW : std_logic_vector(31 downto 0);
begin

    Register_File : entity work.Register_File port map(clk=>clk, rst=>rst, WE
        =>We, Ra=>Ra, Rb=>Rb, Rw=>Rw, A=>busA, B=>busB, W=>busW);

    ALU : entity work.ALU port map(a => busA, b=> busB, s => busW, op => op, n
         => n);

    W <= busw;

end architecture;
```

And based on the textbench **Assemble_ALU_and_Register_File_tb.vhd** and command file **Assemble_ALU_and_Register_File_test.do**, we test some operations:

```
   R(1) = R(15)
R(1) = R(1) + R(15)
R(2) = R(1) + R(15)
R(3) = R(1) - R(15)
R(5) = R(7) - R(15)
```

The simulation result is shown in Figure 4. Detailed waves can be found as Figure 19 in Appendices A.

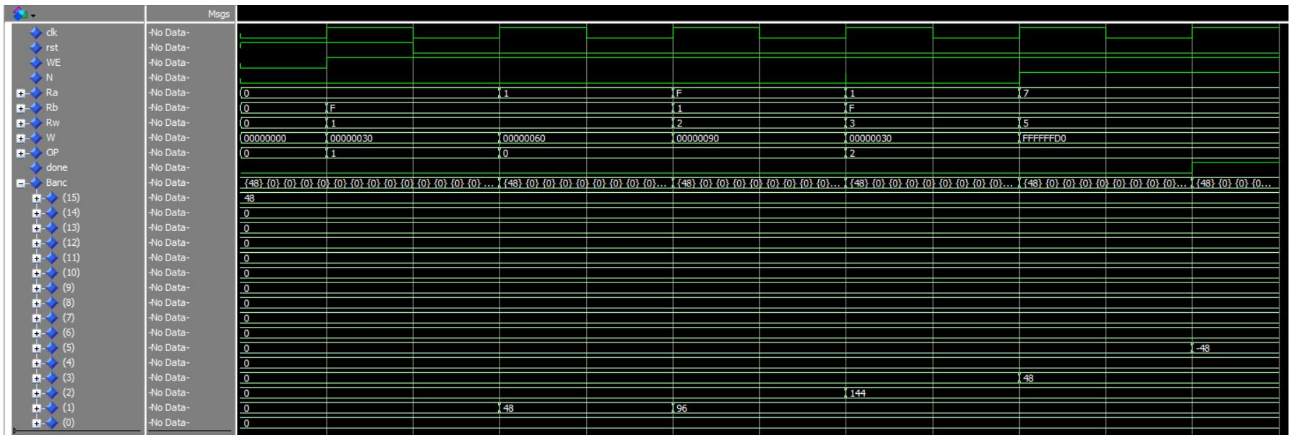


Figure 4: Simulation Waves of Assemble ALU and Regist_File

1.3 2 to 1 Multiplexer (Multiplexeur 2 vers 1)

This multiplexer has a generic parameter `N` fixing the size of the data input and output. A symbolic representation of the multiplexer is shown in Figure 5.

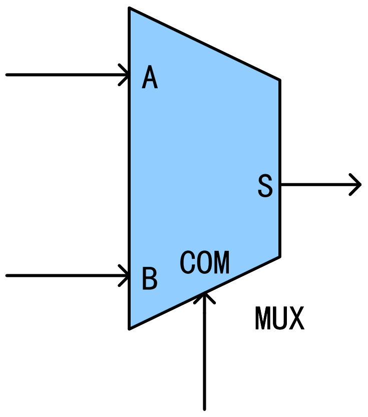


Figure 5: 2 to 1 Multiplexer

Where `A` and `B` are data inputs, `S` is data output, and they are all `N-bit`; `COM` is a choose signal which is 1 bit. The choose table is as below.

So we build the component `MUX` in **MUX.vhd**.

Table 2: MUX Choose Table

COM	S
0	S=A
1	S=B

```vhdl
entity MUX is
    generic (N : positive :=32);

    port (
        S : out std_logic_vector(N-1 downto 0);
        A, B : in std_logic_vector(N-1 downto 0);
        COM : in std_logic
    );
end entity MUX;

architecture behave of MUX is
begin

    process(A,B,COM)
    begin
        if COM = '0' then S <= A;
        elsif COM='1' then S <= B;
        end if;
    end process;

end architecture;
```

1.4 Sign Extension (Extension de Signe)

This module is used to extend the sign of an input coded on `N bits` to `32 bits`. It therefore has a generic parameter fixing the value of `N`. A symbolic representation of the multiplexer is shown in Figure 6.

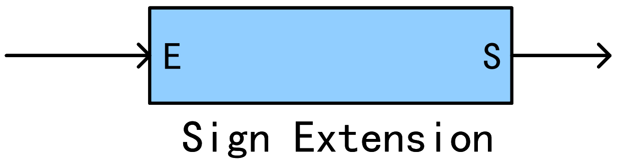


Figure 6: Sign Extension Block Diagram

Where `E` is a `N-bit` data input bus and `S` is a 32-bit output bus.
And part of its code in **Sign_Extension.vhd** shows below.

```vhdl
entity Sign_Extension is
    generic (N : positive :=8);
    port (
        S : out std_logic_vector(31 downto 0);
        E : in std_logic_vector(N-1 downto 0)
    );
end entity;

architecture behav of Sign_Extension is
begin

    process(E)
    begin
        S(N-1 downto 0) <= E;
        S(31 downto N) <= (others => E(N-1));
    end process;

end architecture;
```

1.5 Data Memory (Mémoire de Données)

This memory is used to load and store 64 32-bit words. A symbolic representation of the Memory is shown in Figure 7.

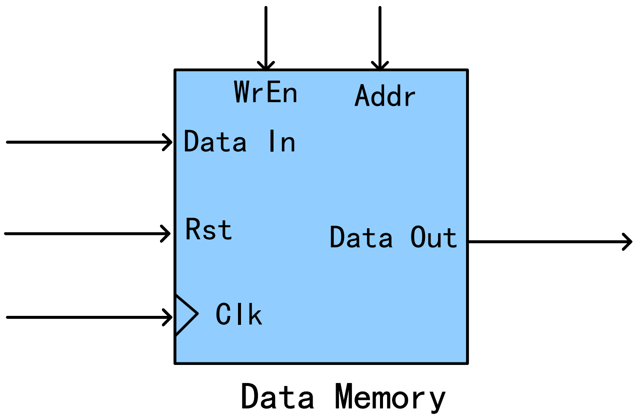


Figure 7: Data Memory Block Diagram

Where `Clk` is a clock Signal; `Rst` is a asynchrone reset signal which active at high level; `WrEn` is a enable signal of writing datas and `Addr` is a 6 bits address bus; `Data In` and `Data Out` are 32 bits data buses;

Thus, the ports of the entity we defined as follow:

```vhdl
entity Data_Memory is
    port (
        clk, rst, WE : in std_logic ;
        Addr : in std_logic_vector(5 downto 0);
        DataOut : out std_logic_vector(31 downto 0);
        DataIn : in std_logic_vector(31 downto 0)
    );
end entity;
```

Similar to Register File, when `WrEn`= 1, it will write data from `Data In` to the `Addr` register; and when `WrEn`= 0, it will do nothing. As for reading, it will be done in a combinatorial and simultaneous way.

Therefore, we build the `architecture` of Data Memory in **Data_Memory.vhd**

```
architecture behave of Data_Memory is
    type matrix is array(63 downto 0) of std_logic_vector(31 downto 0);
    signal datas: matrix;
begin

    DataOut <= datas(to_integer(unsigned(Addr)));
    process(clk, rst)
    begin
        if rst ='1' then
            for i in 63 downto 0 loop
                datas(i) <= std_logic_vector(to_unsigned(i,32));
            end loop;
        elsif rising_edge(clk) then
            if WE = '1' then
                datas(to_integer(unsigned(Addr)))<=DataIn;
            end if;
        end if;
    end process;

end architecture;
```

1.6 Assemble Processing Unit (Assemblage Unité de Traitement)

We assemble all the components we have finished before to make a Processing Unit. All the signals are readable from the bolck diagram shown as Figure 8. And we give the part of source code of **Assemble_Processing_Unit.vhd**.

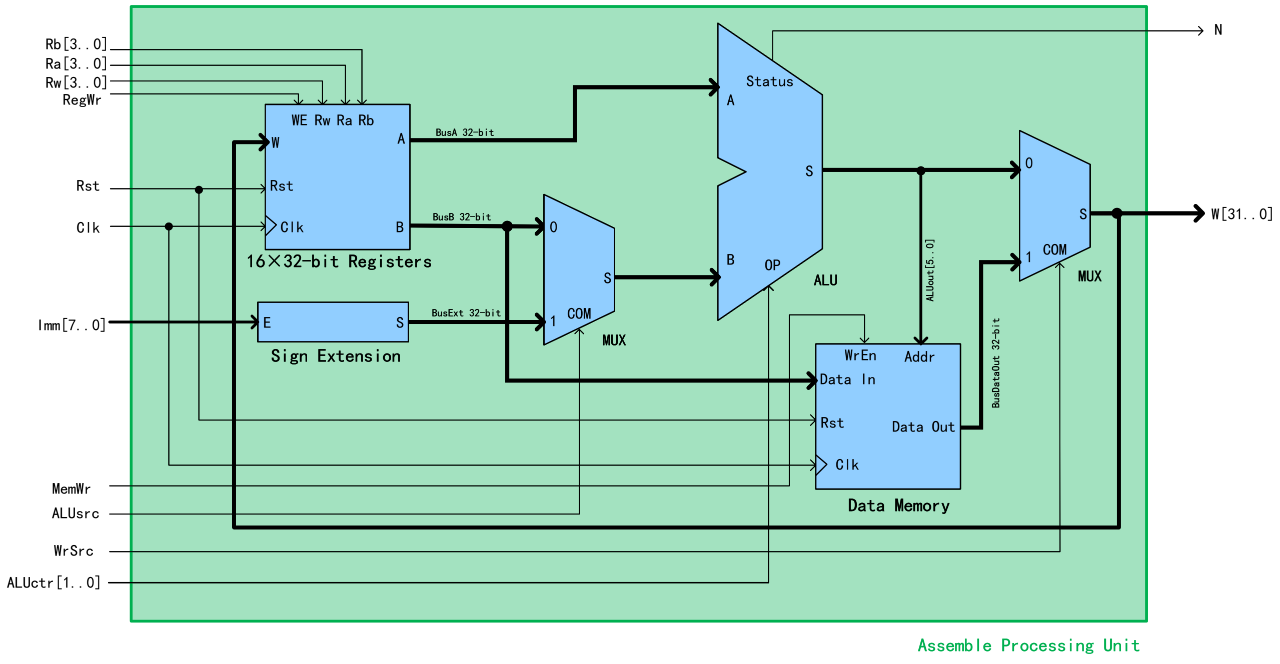


Figure 8: Assemble Processing Unit Block Diagram

```
architecture behave of Assemble_Processing_Unit is
```

```
    signal busA,busB,ALUS,busExtension,busMux, busW : std_logic_vector(31
       downto 0);
    signal DataOut: std_logic_vector(31 downto 0);
begin

    Register_File : entity work.Register_File port map(Clk => clk, rst => rst,
        WE=> RegWr, Ra => Ra, Rb => Rb, Rw => Rw, A => busA, B => busB, W=>
       busW);

    ALU : entity work.ALU port map(A => busA, B=> busMux, S => ALUS, OP =>
       ALUctr, N => N);

    Sign_Extension : entity work.Sign_Extension port map (E => Imm, S =>
       busExtension);

    MUX1 : entity work.MUX port map (A => busB, B=> busExtension, S=> busMux,
       COM => ALUSrc);

    MUX2 : entity work.MUX port map (A => ALUS, B=> DataOut, S=> busW, COM =>
       WrSrc);

    Data_Memory : entity work.Data_Memory port map (Clk => clk, rst => rst,
       Addr => ALUS(5 downto 0), WE => MemWr, DataIn => busB, DataOut =>
       DataOut);

    W <= busw;

end architecture;
```

2 Instruction Management Unit (Unité de gestion des instructions)

2.1 Design Instruction Management Unit

The 32-bit instruction management unit possesses some properties:

- An instruction memory of 64 words of 32 bits similar to that of the processing unit.
- There is no write data bus and Write Enable.
- A 32-bit register (PC register) which has a clock and a asynchronous reset (active at high level).
- An extension unit of 24 to 32 signed bits similar to the component `Extension` described previously.

A symbolic representation of an Instruction Management Unit is shown in Figure 9.

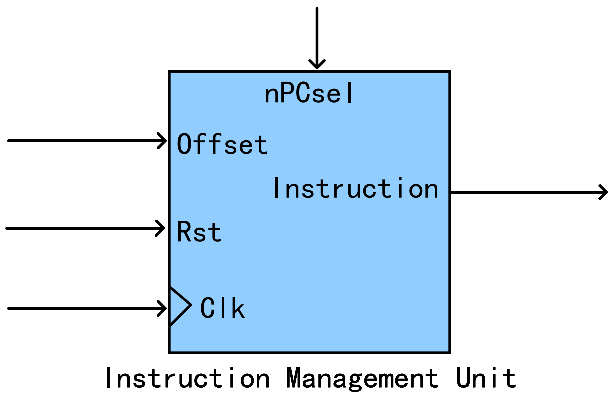


Figure 9: Instruction Management Unit Block Diagram

Abstracted by the given block diagram, the function of the instruction management unit can be discribed as follow:

$$\mathtt{PC} = \begin{cases} \mathtt{PC} + 1 & \mathtt{nPCsel} = 0 \\ \mathtt{PC} + 1 + \mathtt{offset} & \mathtt{nPCsel} = 1 \end{cases}$$

Because the instructions given are 24-bit, we need to use component `Extension` to extend them to 32-bit. After that, we store these 32-bit instructions in memory by `Datamemory`.

The part of the source in **Instruction_Management_Unit.vhd** will be given below.

```
architecture behave of Instruction_Management_Unit is
   signal PC, S : std_logic_vector(31 downto 0);
begin

   Instruction_memory : entity work.instruction_memory port map (PC=> PC,
      Instruction=> Instruction);
   Sign_Extension : entity work.Sign_Extension generic map(N=> 24) port map (
      E => Offset, S => S);

   process(clk, rst)
```

```vhdl
    begin
        if rst ='1' then
            PC <= (others => '0');
        elsif rising_edge(clk) then
            if nPCsel = '0' then
                PC <= std_logic_vector(unsigned(PC)+1);
            else
                PC <= std_logic_vector(unsigned(PC)+1+ unsigned(S));
            end if;
        end if;
    end process;
end architecture;
```

And here is the code **instruction_memory.vhd** in Appendices A.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity instruction_memory is
    port(
        PC: in std_logic_vector(31 downto 0);
        Instruction: out std_logic_vector(31 downto 0)
    );
end entity;

architecture RTL of instruction_memory is
    type RAM64x32 is array(0 to 63) of std_logic_vector(31 downto
       0);

function init_mem return RAM64x32 is
    variable result : RAM64x32;
begin
    for i in 63 downto 0 loop
        result(i) := (others => '0');
    end loop;                       -- PC         -- INSTRUCTION
        result(0) := x"E3A01020";-- 0x0 _main -- MOV R1,#0x20
        result(1) := x"E3A02000";-- 0x1       -- MOV R2,#0x00
        result(2) := x"E6110000";-- 0x2 _loop -- LDR R0,0(R1)
        result(3) := x"E0822000";-- 0x3       -- ADD R2,R2,R0
        result(4) := x"E2811001";-- 0x4       -- ADD R1,R1,#1
        result(5) := x"E351002A";-- 0x5       -- CMP R1,0x2A
        result(6) := x"BAFFFFFB";-- 0x6       -- BLT loop
        result(7) := x"E6012000";-- 0x7       -- STR R2,0(R1)
        result(8) := x"EAFFFFF7";-- 0x8       -- BAL main
    return result;
end init_mem;

signal mem: RAM64x32 := init_mem;

begin
```

```
    Instruction <= mem(to_integer(unsigned(PC)));
end architecture;
```

2.2 Simulation for Instruction Management Unit

We build the testbench in order to test whether this unit work properly.
We mainly test the instruction such as

$$\text{PC} <= \text{PC} + 1$$
$$\text{PC} <= \text{PC} + 1 + \texttt{offset}$$

and change the value of `offset`$= \{1, -1\}$.

Given the code of the testbench **Instruction_Management_Unit_tb.vhd** as follow.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Instruction_Management_Unit_tb IS
end entity ;

architecture BENCH of Instruction_Management_Unit_tb is
    signal Instruction : std_logic_vector(31 downto 0);
    signal Clk         : std_logic := '0';
    signal rst, nPCsel : std_logic;
    signal Offset      : std_logic_vector(23 downto 0);
    signal Done        : boolean := False;
    constant Period    : time := 20 ns;
begin

    UUT : entity work.Instruction_Management_Unit port map(clk=>clk
       , rst=> rst, nPCsel =>nPCsel, Offset => Offset,Instruction
       => Instruction);

    CLK <= '0' when Done else not CLK after Period / 2;
    Rst <= '1', '0' after 5 ns;

    process
    begin

        nPCsel<= '0';
        Offset <= (others => '0'); -- PC <= PC + 1
        wait for 20 ns;

        nPCsel<= '0';
        Offset <= (others => '0'); -- PC <= PC + 1
        wait for 20 ns;

        nPCsel<= '1';
        Offset <= x"000001"; -- PC <= PC + 1 + Offset 1
        wait for 20 ns;

        nPCsel<= '0';
        Offset <= (others => '0'); -- PC <= PC + 1
        wait for 20 ns;

        nPCsel<= '1';
        Offset <= x"FFFFFF"; -- PC <= PC + 1 + Offset -1
        wait for 20 ns;

        nPCsel<= '0';
        Offset <= (others => '0'); -- PC <= PC + 1
        wait for 20 ns;

        Done <= True;
        wait;

    end process;

end architecture;
```

With the command file **Instruction_Management_Unit_test.do**, the waves of the simulation are shown as Figure 10. Detailed waves can be found as Figure 20 in Appendices A.

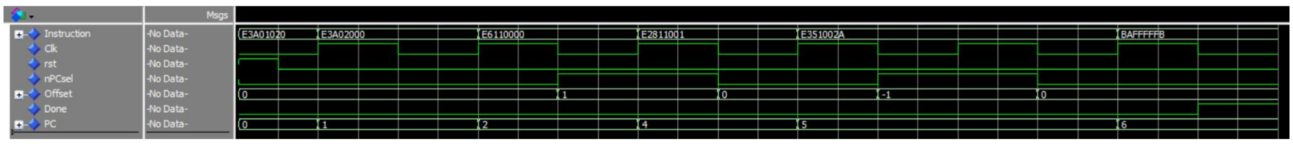


Figure 10: Simulation of Instruction Management Unit

It can be seen clearly that `PC` change itself as we expected.

3 Control Unit (Unité de Contrôle)

The control unit consists of a 32-bit register and a combinatorial decoder.

3.1 32-bit Register with Load Instruction (Registre 32-bit avec Commande de Chargement)

This register will be used to store the state of the processor (Processor State Register, PSR). In this project, we only consider the state which will be limited to the value of the `N flag` of the **ALU**. A symbolic representation of this 32-bit register is shown in Figure 11.

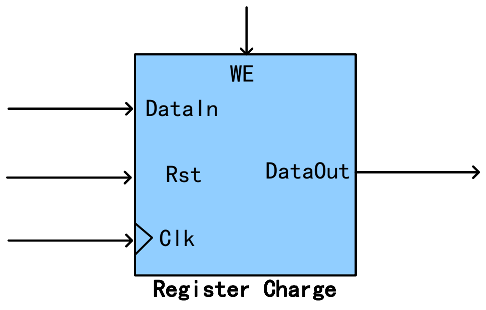


Figure 11: 32-bit Register Block Diagram

Where `DataIn` and `DataOut` are the 32-bit buses for instruction input and outpout respectively, `WE` is enable signal for charge command.

Based on the analysis above, we can draw the equation:

$$\texttt{DataOut} = \begin{cases} \texttt{DataOut} & \texttt{WE} = 0 \\ \texttt{DataIn} & \texttt{WE} = 1 \end{cases}$$

It's not hard to synthesize the code. We show the part of the code of this component in **Registre_Charge.vhd**.

```vhdl
architecture behave of Register_Charge is
begin

    process(clk, rst)
    begin
        if rst ='1' then
            DataOut <= (others => '0');
        elsif rising_edge(clk) then
            if WE = '1' then
                DataOut <= DataIn;
            end if;
        end if;
    end process;

end architecture;
```

3.2 Instruction Decoder (Decodeur d'Instructions)

This combinatorial module generates the control signals for the processing unit, the instruction management unit, as well as the PSR register(32-bit register), which all described previously.

A symbolic representation of decoder is shown in Figure 12.

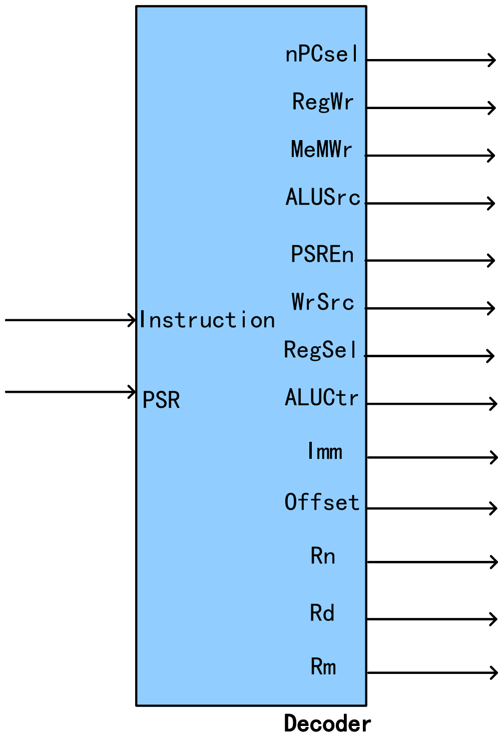


Figure 12: Decoder Block Diagram

The values of these commands depend on the statement retrieved from the instruction memory, and possibly the state of the PSR register. The structure binary of the different instructions is described in the Appendix, and we completed it as Table 3.

Table 3: Commands

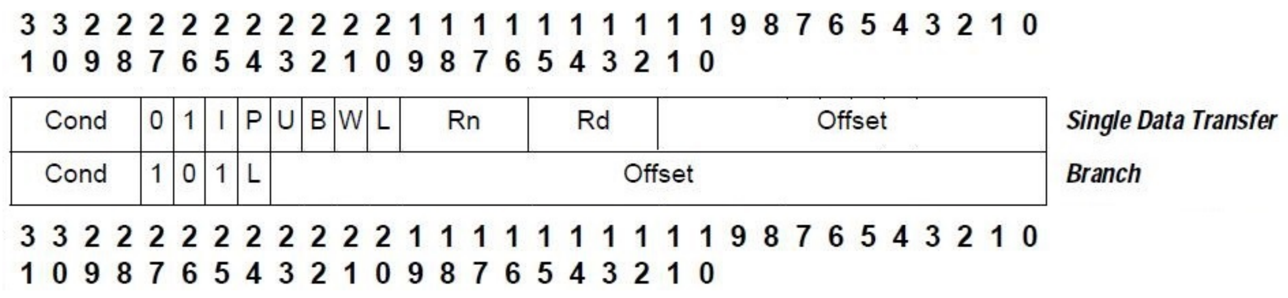
INSTRUCTION	nPCSel	RegWr	ALUSrc	ALUCtr	PSREn	MemWr	WrSrc	RegSel
ADDi	0	1	1	00	0	0	0	0
ADDr	0	1	0	00	0	0	0	0
BAL	1	0	0	00	0	0	0	0
BLT	0	0	0	00	0	0	0	0
CMP	0	0	1	10	1	0	0	0
LDR	0	1	1	00	0	0	1	0
MOV	0	1	1	10	0	0	0	0
STR	0	0	1	00	0	1	0	1

The ARM instruction set formats are shown in the Appendices A.

And we focus on the instructions such as

```
    LDR Rd, [Rn, #Offset]   @ LDR (Immediate)
    STR Rd, [Rn, #Offset]   @ STD (Immediate)
    B       label           @ B (Always)
    BLT     label           @ B (If Less Than)
```

These instruction set formats are shown as Figure 13, and more detailed information will be shown in the Appendices A.



31 … 28	27	26	25	24	23	22	21	20	19 … 16	15 … 12	11 … 0	
Cond	0	1	I	P	U	B	W	L	Rn	Rd	Offset	*Single Data Transfer*
Cond	1	0	1	L	Offset							*Branch*

Figure 13: Load, Store and Branch Instruction Set Formats

The single data transfer instructions are used to load or store single bytes or words of data. The memory address used in the transfer is calculated by adding an offset to or subtracting an offset from a base register.

The result of this calculation may be written back into the base register if auto-indexing is required.

Branch instructions contain a signed 2's complement 24 bit offset. This is shifted left two bits, sign extended to 32 bits, and added to the PC. The instruction can therefore specify a branch of +/- 32Mbytes. The branch offset must take account of the prefetch operation, which causes the PC to be 2 words (8 bytes) ahead of the current instruction. Branches beyond +/- 32Mbytes must use an offset or absolute destination which has been previously loaded into a register. In this case the PC should be manually saved in R14 if a Branch with Link type operation is required.

Thus, for these 4 instructions, bit assignments are as follow:

Table 4: LDR (Immediate) Bit Assignment

31	30	29	28	27	26	25	24	23	22	21	20	19 ... 16	15 ... 12	11 ... 0
1	1	1	0	0	1	1	0	0	0	0	1	Rn	Rd	Offset

Table 5: STR (Immediate) Bit Assignment

31	30	29	28	27	26	25	24	23	22	21	20	19 ... 16	15 ... 12	11 ... 0
1	1	1	0	0	1	1	0	0	0	0	0	Rn	Rd	Offset

Table 6: B (Always) Bit Assignment

31	30	29	28	27	26	25	24	23 0
1	1	1	0	1	0	1	0	Offset

Table 7: B (If Less Than) Bit Assignment

31	30	29	28	27	26	25	24	23 0
1	0	1	1	1	0	1	0	Offset

Based on the analysis above, we build the code of **Decoder.vhd**.

```vhdl
entity Decoder  is
    port(
        Instruction, PSRout : in std_logic_vector(31 downto 0);
        Offset : out std_logic_vector(23 downto 0);
        Immediate : out std_logic_vector(7 downto 0);
        Rn, Rm, Rd : out std_logic_vector(3 downto 0);
        ALUctr : out std_logic_vector(1 downto 0);
        nPCsel, RegWr, ALUsrc, PSRen, MemWr, WrSrc, RegSel : out
           std_logic
    );
end entity;

architecture behave of Decoder  is
    type enum_instruction is (MOV, ADDi, ADDr, CMP, LDR, STR, BAL,
       BLT, XXX);
    signal instr_courante: enum_instruction;
begin

    Immediate <= Instruction(7  downto  0);
    Offset    <= Instruction(23 downto  0);
    Rn        <= Instruction(19 downto 16);
    Rd        <= Instruction(15 downto 12);
    Rm        <= Instruction(3  downto  0);
    process(Instruction)
    begin
        case Instruction(27 downto 26) is
            when "00"   =>
                case Instruction(25 downto 23) is
                    when "001"          => instr_courante <= ADDr;
                    when "101"          =>
                        case Instruction(29) is
                            when '1'    => instr_courante <= ADDi;
                            when others => instr_courante <= XXX;
                        end case;
                    when "110" | "010"  => instr_courante <= CMP;
                    when "111"          => instr_courante <= MOV;
                    when others         => instr_courante <= XXX;
                end case;

            when "01"   =>
                case Instruction(20) is
                    when '0'    =>
                        case Instruction(29) is
                            when '1'    => instr_courante <= STR;
                            when others => instr_courante <= XXX;
                        end case;
                    when '1'    => instr_courante <= LDR;
                    when others => instr_courante <= XXX;
                end case;

            when "10"    =>
                case Instruction(29 downto 28) is
                    when "10"     => instr_courante <= BAL;
                    when "11"     => instr_courante <= BLT;
                    when others   => instr_courante <= XXX;
                end case;
            when others => instr_courante <= XXX;
        end case;
    end process;

    process(instr_courante)
    begin
        -- MemWr et RegSel
        case instr_courante is
            when STR     =>  MemWr   <= '1';
                             ALUSrc  <= '1';
                             RegSel  <= '1';
            when others =>   MemWr   <= '0';
                             RegSel  <= '0';
        end case;

        -- WrSrc
        case instr_courante is
            when LDR     =>  WrSrc  <= '1';
            when others =>   WrSrc  <= '0';
        end case;

        -- PSRen et UALctr
        case instr_courante is
            when CMP     =>  PSRen   <= '1';
                             ALUctr  <= "10";
            when MOV     =>  PSRen   <= '0';
                             ALUctr  <= "01";
            when others =>   PSRen   <= '0';
                             ALUctr  <= "00";
        end case;

        --UALsrc
        case instr_courante is
            when ADDr    =>  ALUsrc  <= '0';
            when CMP     =>  ALUsrc  <= Instruction(25);
            when others =>   ALUsrc  <= '1';
        end case;

        -- RegWr
        case instr_courante is
            when ADDi | ADDr | LDR | MOV => RegWr  <= '1';
            when others                  => RegWr  <= '0';
        end case;

        -- nPCsel
        case instr_courante is
            when BAL    => nPCsel <= '1';
            when BLT    => nPCsel <= PSRout(31);
            when others => nPCsel <= '0';
        end case;

    end process;

end architecture;
```

At this point, all the components have been constructed. In the next part, we will try to assemble the processor by using these components.

4 Assembly and Validation of Processor (Assemblage et Validation du Processeur)

4.1 Assemble Processor

We modifying the modeling of the processor by assembling its three main units

- The instruction management unit
- The processing unit
- The control unit

Complete the previously designed processing unit by adding a 2-input 4-bit multiplexer controlled by the RegSel control signal generated by the control unit. This multiplexer will be placed at the input of the address Rb of the register file, as shown in the Figure 14.

According to this block diagram, we modify the code for **Processing_Unit.vhd**.

```vhdl
entity Processing_Unit is
   port (
       clk, rst, MemWr, RegWr, ALUsrc, WrSrc ,RegSel, PSREn: in std_logic ;
       Rn, Rm, Rd : in std_logic_vector(3 downto 0);
       busout : out std_logic_vector(31 downto 0);--PSR[31..0]
       Imm : in std_logic_vector(7 downto 0);
       ALUctr : in std_logic_vector(1 downto 0)
   );
end entity;

architecture behave of Processing_Unit is
   signal busA,busB,ALUS,busExtension,busMux, busW : std_logic_vector(31
      downto 0);
   signal DataOut ,fl: std_logic_vector(31 downto 0);
   signal Rb : std_logic_vector(3 downto 0);
   signal N: std_logic;
begin

   Register_File : entity work.Register_File port map(Clk => clk, rst => rst,
       WE=> RegWr, Ra => Rn, Rb => Rb, Rw => Rd, A => busA, B => busB, W=>
      busW);

   ALU : entity work.ALU port map(A => busA, B=> busMux, S => ALUS, OP =>
      ALUctr, N => N);

   Sign_Extension : entity work.Sign_Extension port map ( E => Imm, S =>
      busExtension);

   MUX1 : entity work.MUX port map (A => busB, B=> busExtension, S=> busMux,
      COM => ALUSrc);

   MUX2 : entity work.MUX port map (A => ALUS, B=> DataOut, S=> busW, COM =>
      WrSrc);

```

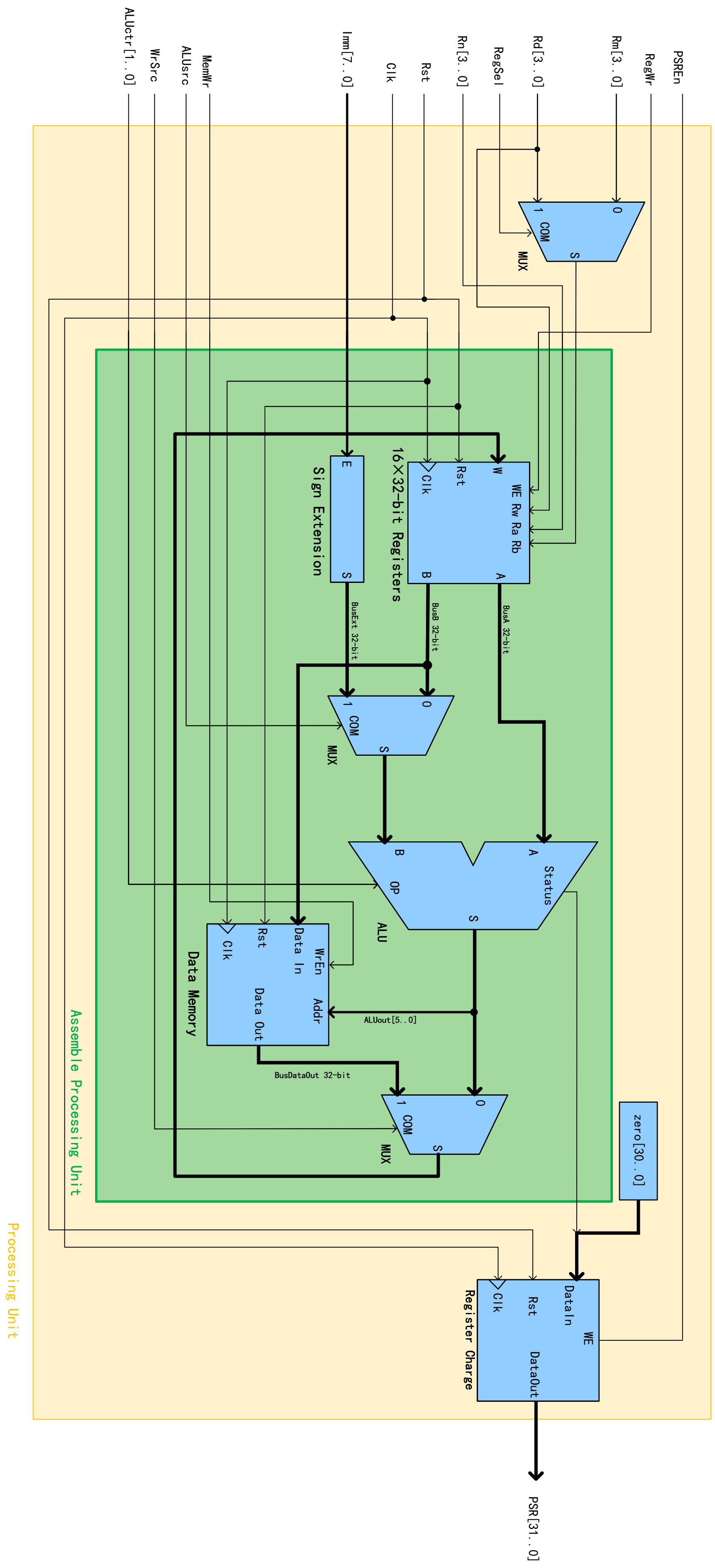


Figure 14: Processing Unit Block Diagram

```vhdl
   Data_Memory : entity work.Data_Memory port map (Clk => clk, rst => rst,
      Addr => ALUS(5 downto 0), WE => MemWr, DataIn => busB, DataOut =>
      DataOut);

   MUX3 : entity work.MUX generic map( N=> 4) port map (A => Rm, B => Rd, COM
       => RegSel, S=> Rb);

   Register_Charge : entity work.Register_Charge port map (clk => clk, rst =>
       rst, WE => PSREn, DataIn => fl, DataOut => busout);

   fl <= N&"000"&X"0000000";

end architecture;
```

With addition of **Processing_Unit**, **Instruction_Management_Unit** and **Decoder**, the **Processor** can be assembled as Figure 15. The detailed block diagram will be given in the Appendices A.

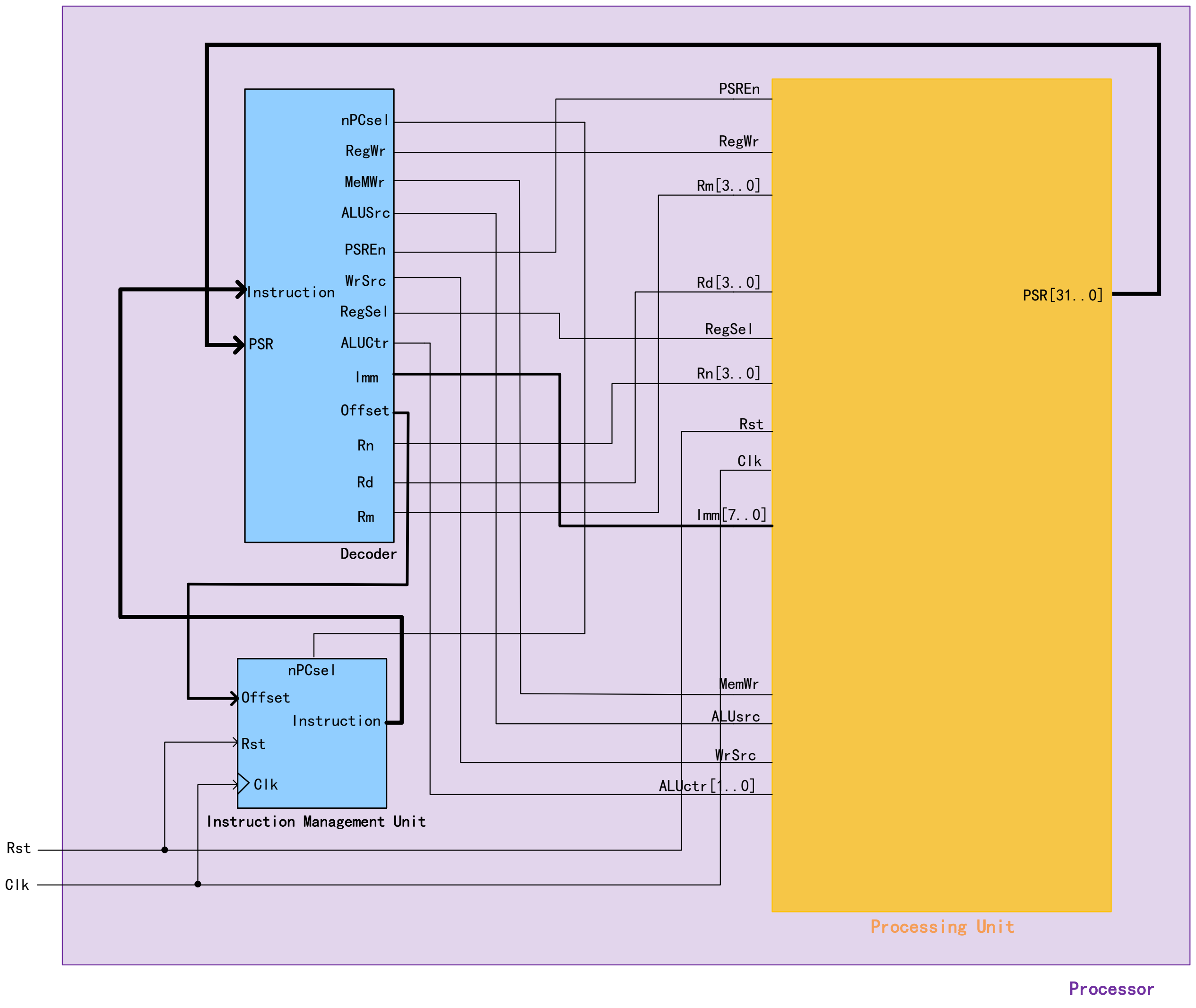


Figure 15: ProcessorBlock Diagram

Thus, the core code for **Processor.vhd** is shown below.

```vhdl
architecture behave of Processor is
    signal nPCSel, MemWr, RegWr, ALUsrc, WrSrc, RegSel, PSRen : std_logic ;
    signal ALUctr : std_logic_vector(1 downto 0);
    signal offset : std_logic_vector(23 downto 0);
    signal Immediate: std_logic_vector(7 downto 0);
    signal Instruction, busout : std_logic_vector(31 downto 0);--busout -> PSR
    signal Rn, Rd, Rm : std_logic_vector(3 downto 0);
begin

    Processing_Unit : entity work.Processing_Unit port map(clk => clk, rst =>
        rst, RegWr=> RegWr, Rn => Rn, Rd => Rd, Rm=> Rm, busout => busout, Imm
        => Immediate, ALUctr=> ALUctr, MemWr=> MemWr, ALUSrc=> ALUSrc,WrSrc=>
        WrSrc, RegSel=> RegSel, PSRen => PSRen);

    Instruction_Management_Unit : entity work.Instruction_Management_Unit port
        map(Clk => clk, rst => rst, nPCSel=> nPCSel, Instruction=> Instruction
        , offset => offset);

    Decoder : entity work.Decoder port map(RegWr=> RegWr, Rn => Rn, Rd => Rd,
        Rm=> Rm, psrout => busout, Immediate =>Immediate, ALUctr=> ALUctr,
        MemWr=> MemWr, ALUSrc=> ALUSrc, WrSrc=> WrSrc, RegSel=> RegSel, PSRen
        => PSRen, nPCSel=> nPCSel, Instruction=> Instruction, offset=> offset);

end architecture;
```

4.2 Simulate Processor

According to the testbench shown below, we run the simulation with command file **Processor_test.do** and obtaine the waves as Figure 16. Detailed waves can be found as Figure 25 in Appendices A.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Processor_tb is
end entity;

architecture test_bench of Processor_tb is
    signal clk, rst: std_logic;
    signal done : std_logic := '0';
    constant clk_period : time:= 10 ns;
begin

    UUT : entity work.Processor(behave)port map(clk => clk,rst => rst);

    rst <= '1', '0' after clk_period;

    clock : process is
    begin
```

```
        if done = '0' then
            clk <= '0';
            wait for clk_period/2;
            clk <= '1';
            wait for clk_period/2;
        else
            wait;
        end if;
    end process;

    signal_gen : process is
    begin
        done <= '0';
        wait for clk_period*180;
        done <= '1';
        wait;
    end process;

end architecture;
```

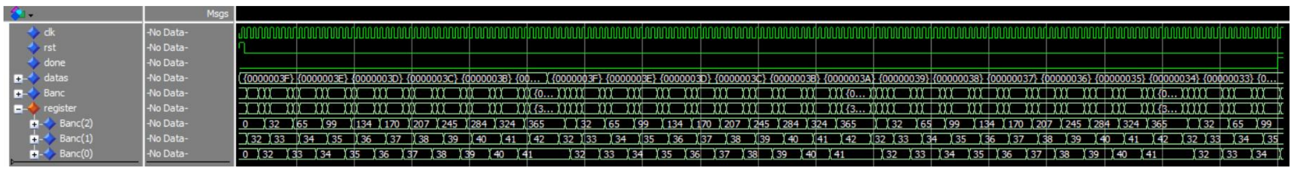


Figure 16: Simulation of Processing Unit

5 Test Processor Completely (Test Complet du Processeur)

In this part, we will test the processor complement.

We first change the code Assembly given to the instruction code in **instruction_memory2.vhd** according to Figure 21.

```
result (0):=x"E3A00010";  -- 0x0 _main   -- MOV R0,#0x10
result (1):=x"E3A01001";  -- 0x1         -- MOV R1,#0x01
result (2):=x"E6103000";  -- 0x2 _for    -- LDR R3,[R0]
result (3):=x"E6104001";  -- 0x3         -- LDR R4,[R0,#1]
result (4):=x"E6004000";  -- 0x4         -- STR R4,[R0]
result (5):=x"E6003001";  -- 0x5         -- STR R3,[R0,#1]
result (6):=x"E2811001";  -- 0x6         -- ADD R1,R1,#1
result (7):=x"E2800001";  -- 0x7         -- ADD R0,R0,#1
result (8):=x"E351000A";  -- 0x8         -- CMP R1,0xA
result (9):=x"BAFFFFF8";  -- 0x9         -- BLT loop
result (10):=x"EAFFFFFF"; -- 0xA _wait   -- BAL wait
```

After runing the simulation with command file **Processor_test.do**, we obtaine waves as Figure 17. Detailed waves can be found as Figure 26 in Appendices A.

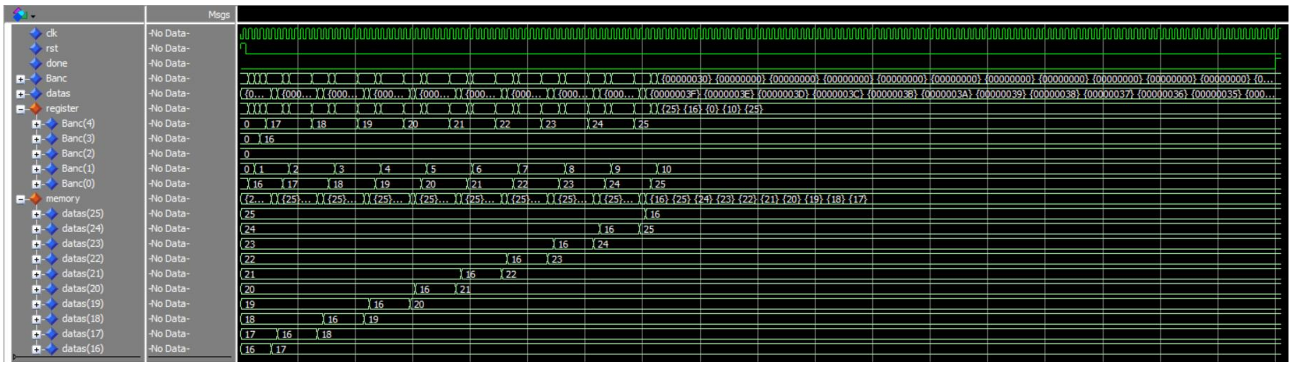


Figure 17: Simulation Waves of Processing Unit Completely

6 Increasing the Instruction Set (Augmentation du Jeu d'Instruction)

In this section, we have added additional command suffix support, including `EQ, NE, LT, GT`. The detailed information is given in Table 8.

Table 8: Condtion Code

Code	Suffix	Flag	Meaning
0000	EQ	Z=1	equal
0001	NE	Z=0	not equal
1011	LT	N=1	less than
1100	GT	N=0	greater than

In order to match these changes, we need to modify the codes we have done before such as **ALU.vhd**, **Decoder.vhd** and **Data_memory.vhd** and rewrite the instructions in **instruction_memory3.vhd**.

For **ALU.vhd**, we add an additional output port `Z` which indicated ZERO in Status.

For **Decoder.vhd**, we add the branch of `case` to make it decoder `EQ, NE, LT, GT` successfully according to Table 8.

For **Data_memory.vhd**, we add the initialization for datas in memory as Table 9.

Table 9: Datas in Memory

Address	Data
0x20	3
0x21	107
0x22	27
0x23	12
0x24	322
0x25	155
0x27	63

Finally, we created the command in **instruction_memory3.vhd** as below.

```
result (0) :=x"E3A00020";-- 0x0 _start     -- MOV R0,#0x20
result (1) :=x"E3A02001";-- 0x1            -- MOV R2,#1
result (2) :=x"E3A02000";-- 0x2 _while     -- MOV R2,#0
result (3) :=x"E3A01001";-- 0x3            -- MOV R1,#1
result (4) :=x"E6103000";-- 0x4 _for       -- LDR R3,[R0]
result (5) :=x"E6104001";-- 0x5            -- LDR R4,R0,#1
result (6) :=x"E1530004";-- 0x6            -- CMP R3, R4
result (7) :=x"C6004000";-- 0x7            -- STRGT R4,[R0]
result (8) :=x"C6003001";-- 0x8            -- STRGT R3,[R0,#1]
result (9) :=x"C2822001";-- 0x9            -- ADDGT R2,R2,#1
result (10):=x"E2800001";-- 0xA            -- ADD R0, R0, #1
result (11):=x"E2811001";-- 0xB            -- ADD R1, R1, #1
result (12):=x"E3510007";-- 0xC            -- CMP R1, #0x07
result (13):=x"BAFFFFF6";-- 0xD            -- BLT FOR
result (14):=x"E3520000";-- 0xE            -- CMP R2, #0
result (15):=x"E3A00020";-- 0xF          -- MOV R0, #0x20
result (16):=x"1AFFFFF1";-- 0x10         -- BNE WHILE
result (17):=x"EAFFFFFF";-- 0x11 _wait   -- BAL wait
```

After running the testbench with command file **Processor_test.do**, we obtain the waves as Figure 18.

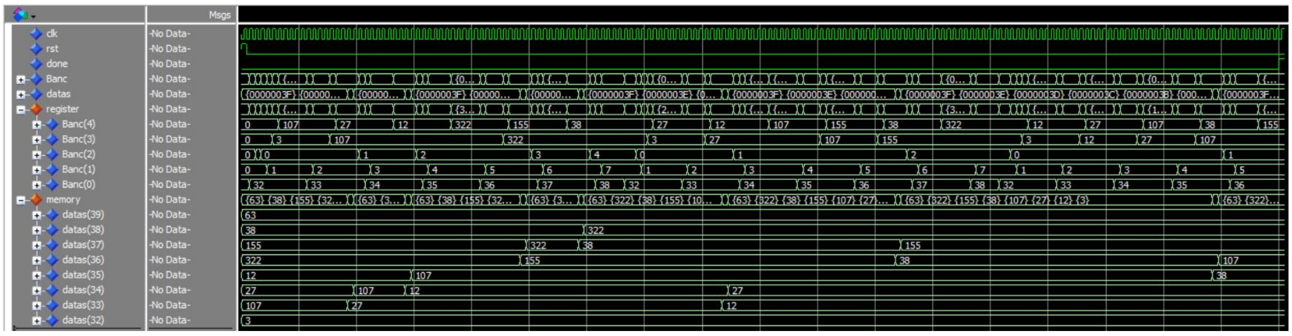


Figure 18: Simulation Waves of Processing Unit with IS Increasing

A Appendices

In Section 1.2.2, the full view of the waves in simulation **Assemble_ALU_and_Regist_File_tb.v** is shown as Figure 19.

In Section 2.2, the full view of the waves in simulation **Instruction_Management_Unit_tb.vhd** is shown as Figure 20.

In Section 3.2, the ARM instruction set formats are shown as Figure 21.

And in the same sectoin (Section 3.2), the detailed information about `Single Data Transfer` and `Branch` are shown as Figure 22 and Figure 23.

In Section 4.1, the Block Diagram of Processor as Figure 24.

In Section 4.2, the full view of the waves in simulation **Processor_tb.vhd** in **part 4** is shown as Figure 25.

In Section 5, the full view of the waves in simulation in **part 5** is shown as Figure 26.

In Section 6, the full view of the waves in simulation in **part 6** is shown as Figure 27.

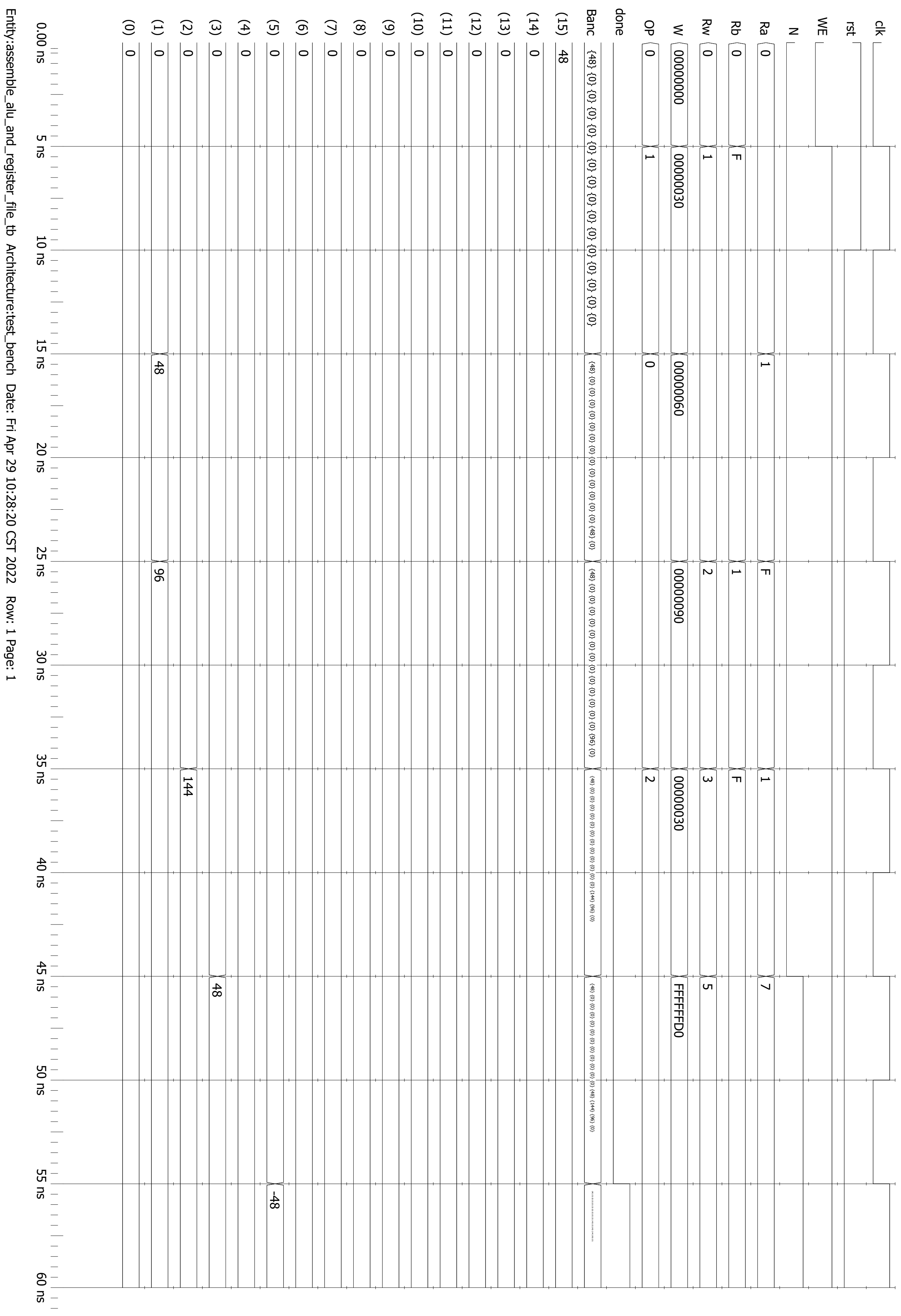


Figure 19: Simulation Waves of Assemble ALU and Regist_File

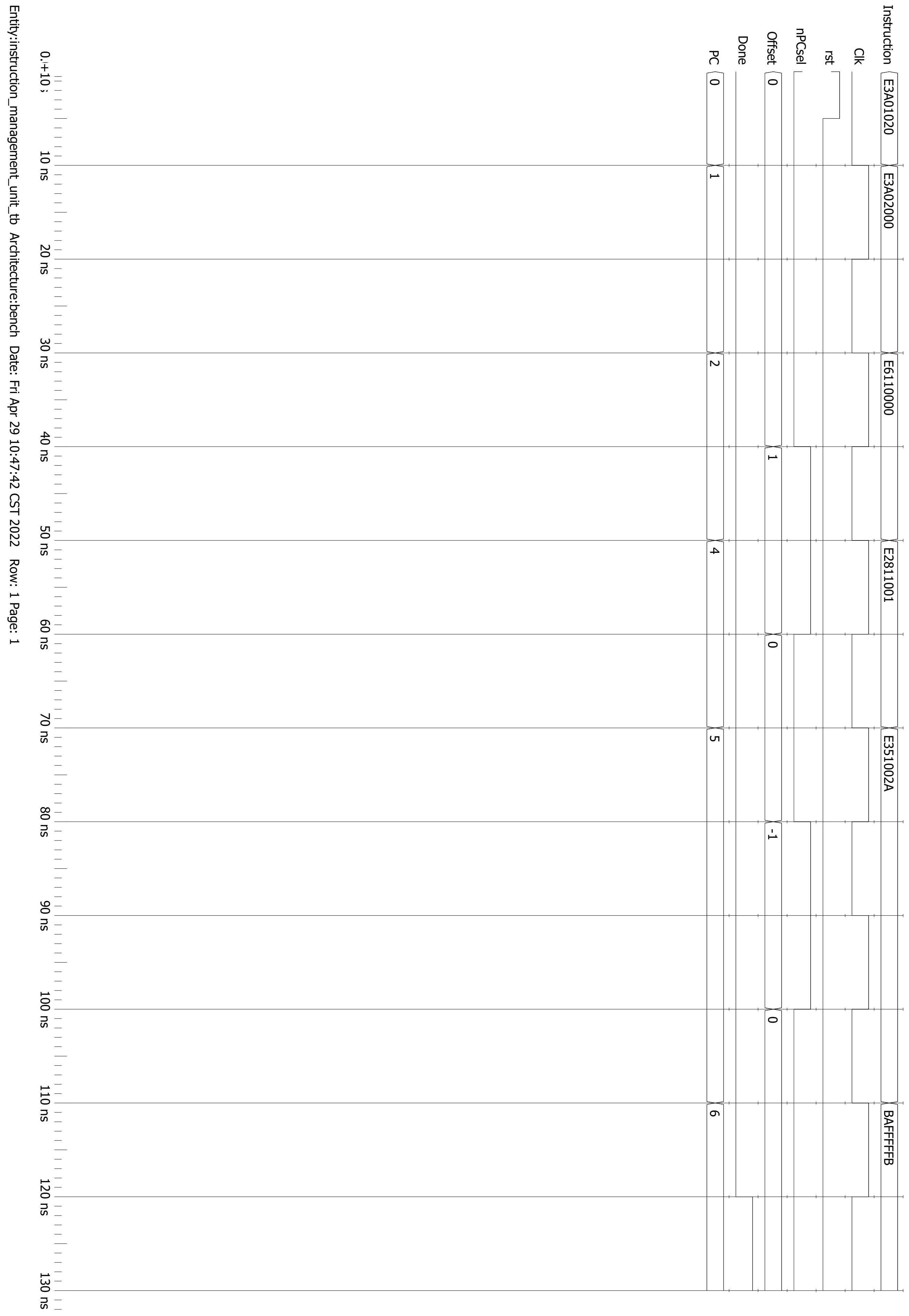


Figure 20: Simulation of Instruction Management Unit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Cond				0	0	I	Opcode				S	Rn				Rd				Operand 2												***Data Processing / PSR Transfer***
Cond				0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm				***Multiply***
Cond				0	0	0	0	1	U	A	S	RdHi				RdLo				Rn				1	0	0	1	Rm				***Multiply Long***
Cond				0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm				***Single Data Swap***
Cond				0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn				***Branch and Exchange***
Cond				0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm				***Halfword Data Transfer: register offset***
Cond				0	0	0	P	U	1	W	L	Rn				Rd				Offset				1	S	H	1	Offset				***Halfword Data Transfer: immediate offset***
Cond				0	1	I	P	U	B	W	L	Rn				Rd				Offset												***Single Data Transfer***
Cond				0	1	1																					1					***Undefined***
Cond				1	0	0	P	U	S	W	L	Rn				Register List																***Block Data Transfer***
Cond				1	0	1	L	Offset																								***Branch***
Cond				1	1	0	P	U	N	W	L	Rn				CRd				CP#				Offset								***Coprocessor Data Transfer***
Cond				1	1	1	0	CP Opc				CRn				CRd				CP#				CP			0	CRm				***Coprocessor Data Operation***
Cond				1	1	1	0	CP Opc			L	CRn				Rd				CP#				CP			1	CRm				***Coprocessor Register Transfer***
Cond				1	1	1	1	Ignored by processor																								***Software Interrupt***
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

Figure 21: ARM Instruction Set Formats

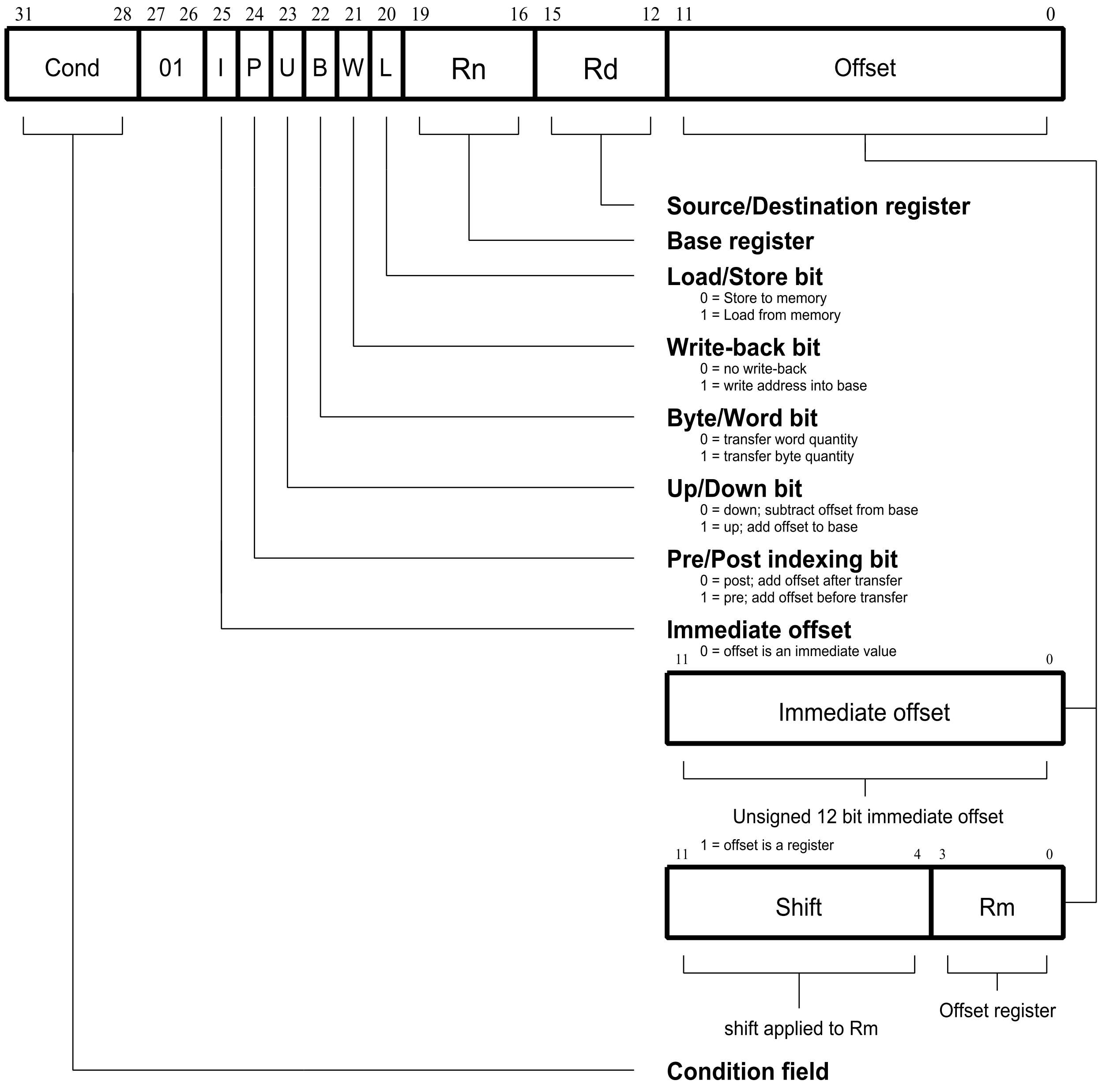


Figure 22: Single Data Transfer Instructions

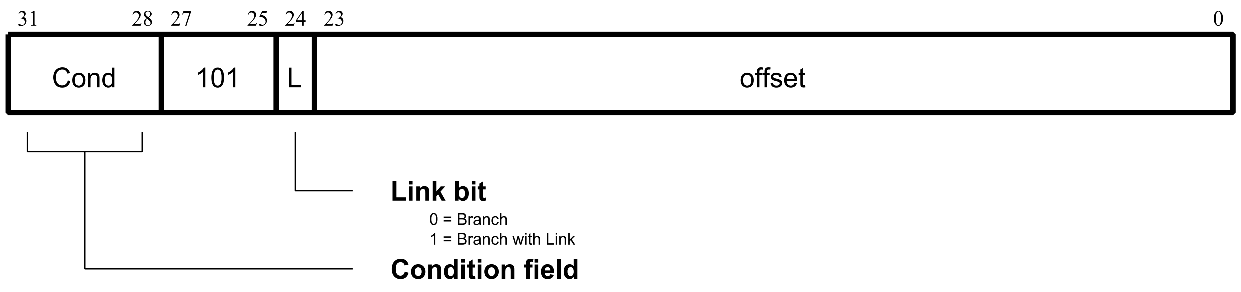


Figure 23: Branch Instructions

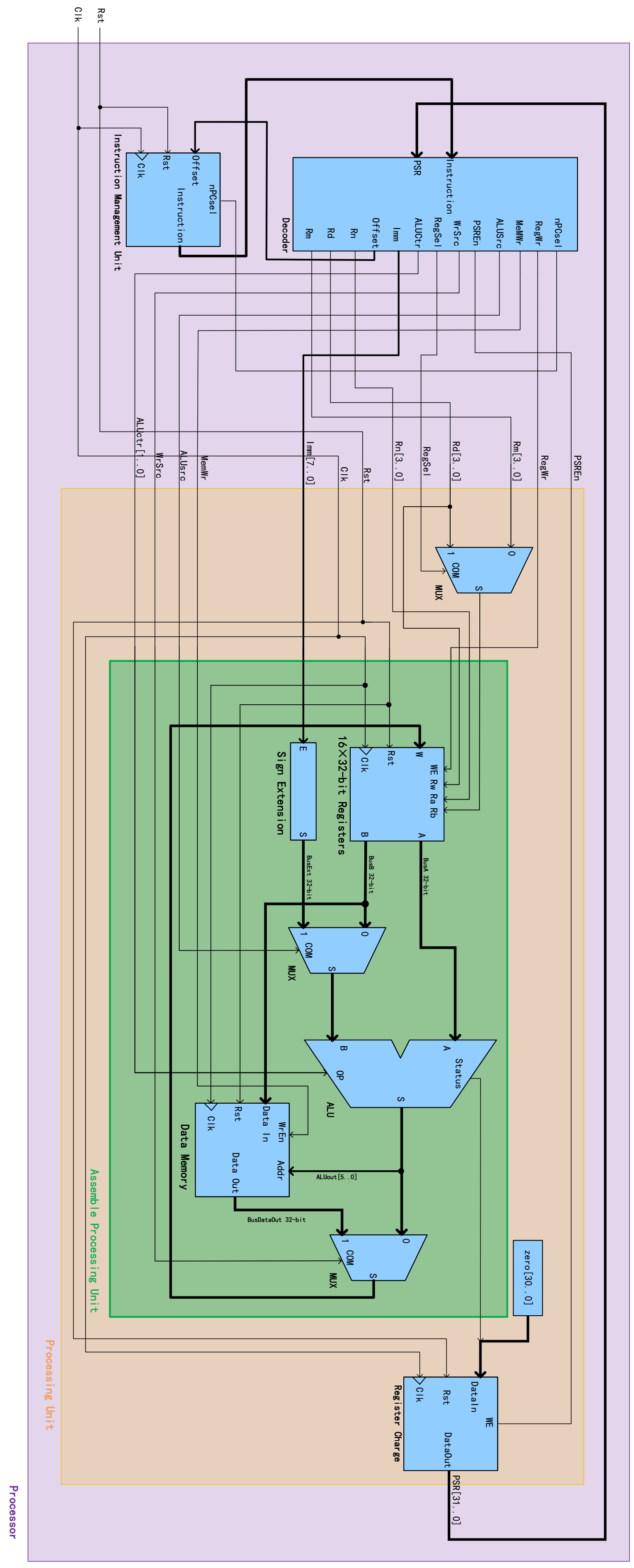
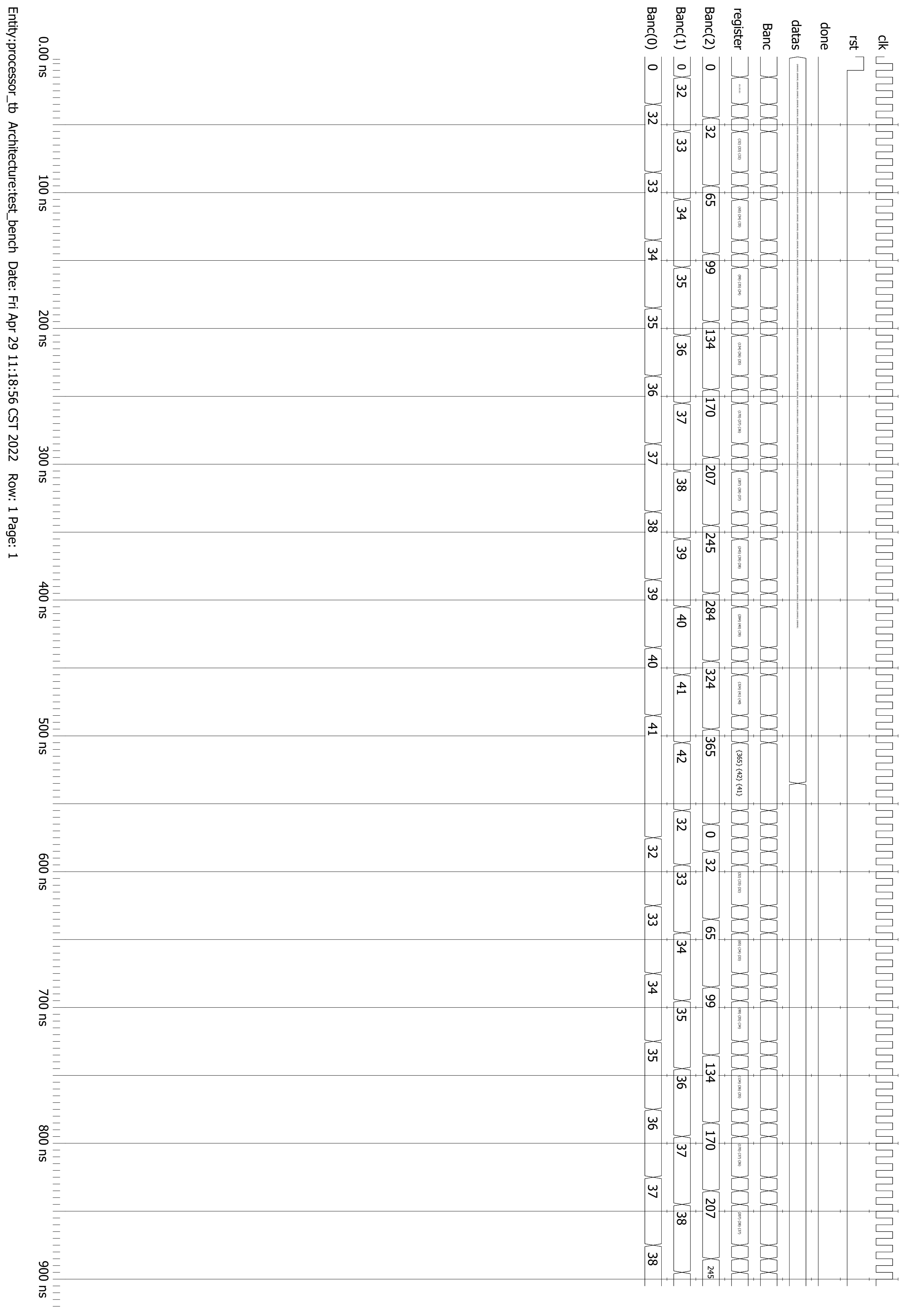


Figure 24: Processor Block Diagram


clk
rst
done
datas
Banc
register
Banc(2)
Banc(1)
Banc(0)
0.00 ns
100 ns
200 ns
300 ns
400 ns
500 ns
600 ns
700 ns
800 ns
900 ns
0 32 65 99 134 170 207 245 284 324 365 0 32 65 99 134 170 207 245
0 32 33 34 35 36 37 38 39 40 41 42 32 33 34 35 36 37 38
0 32 33 34 35 36 37 38 39 40 41 32 33 34 35 36 37 38
{365} {42} {41}
Entity:processor_tb Architecture:test_bench Date: Fri Apr 29 11:18:56 CST 2022 Row: 1 Page: 1

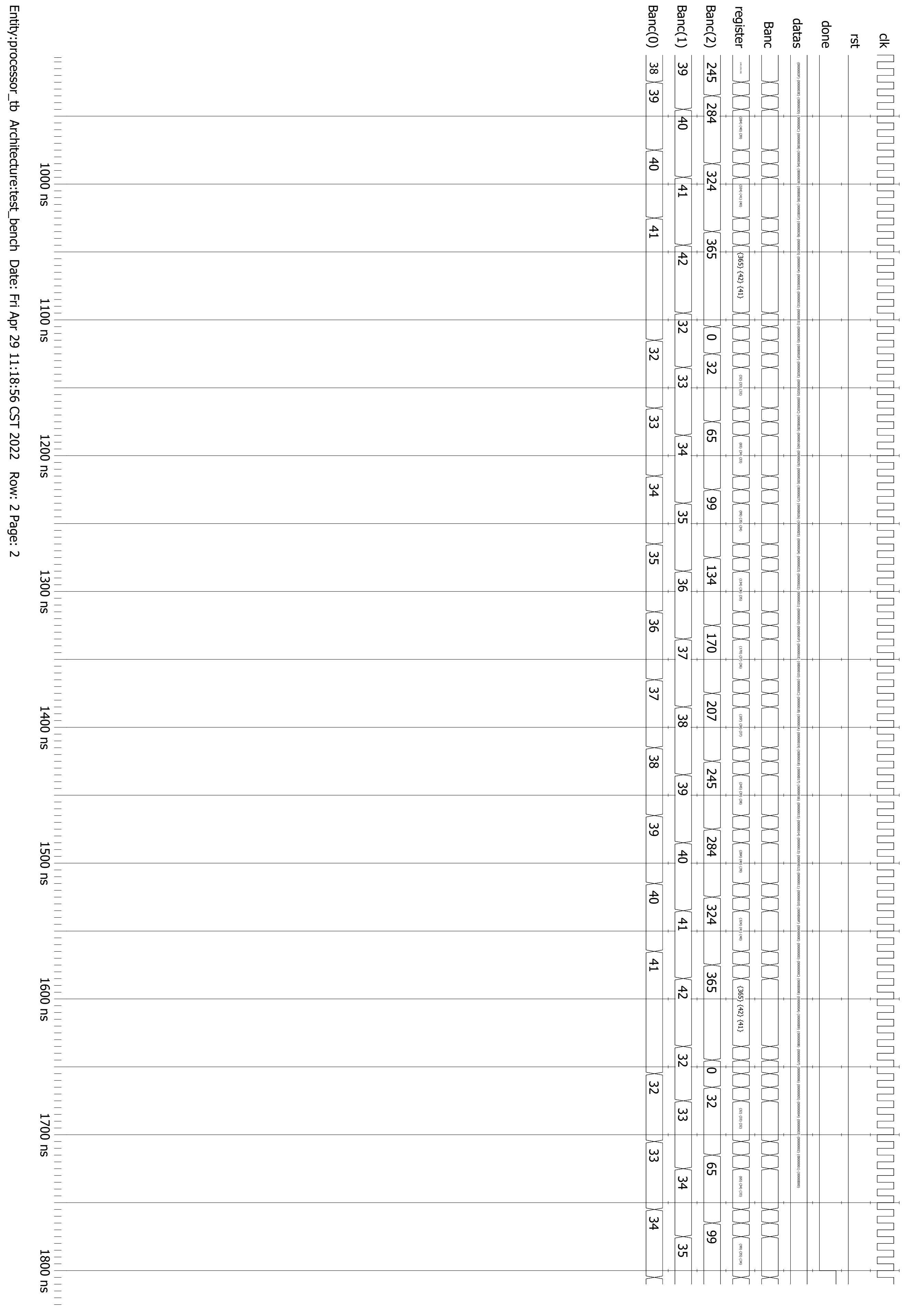
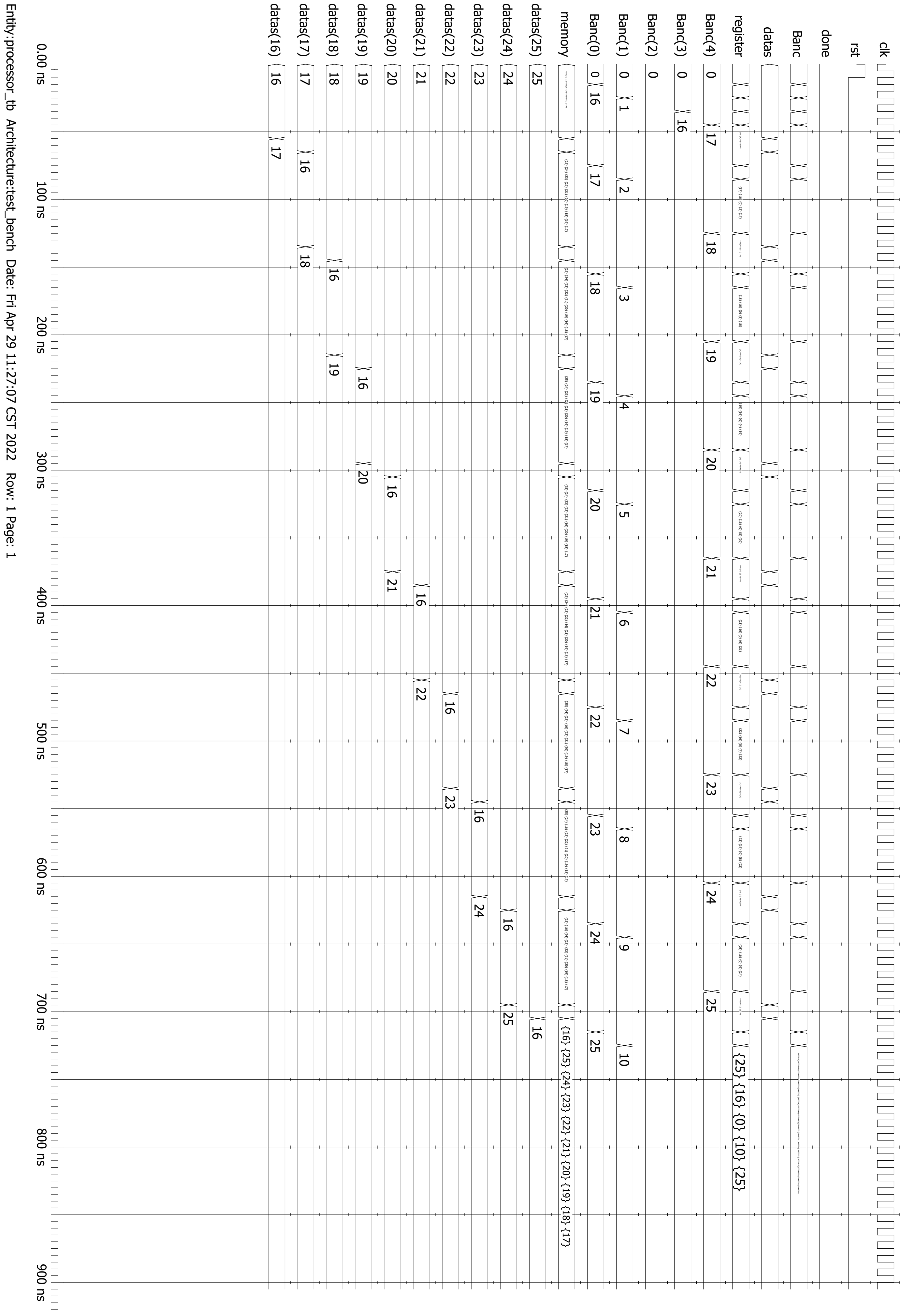


Figure 25: Simulation Waves of Processor in Part 4


clk
rst
done
Banc
datas
register
Banc(4)
Banc(3)
Banc(2)
Banc(1)
Banc(0)
memory
datas(25)
datas(24)
datas(23)
datas(22)
datas(21)
datas(20)
datas(19)
datas(18)
datas(17)
datas(16)
{25} {16} {0} {10} {25}
{16} {25} {24} {23} {22} {21} {20} {19} {18} {17}
0.00 ns
100 ns
200 ns
300 ns
400 ns
500 ns
600 ns
700 ns
800 ns
900 ns
Entity:processor_tb Architecture:test_bench Date: Fri Apr 29 11:27:07 CST 2022 Row: 1 Page: 1

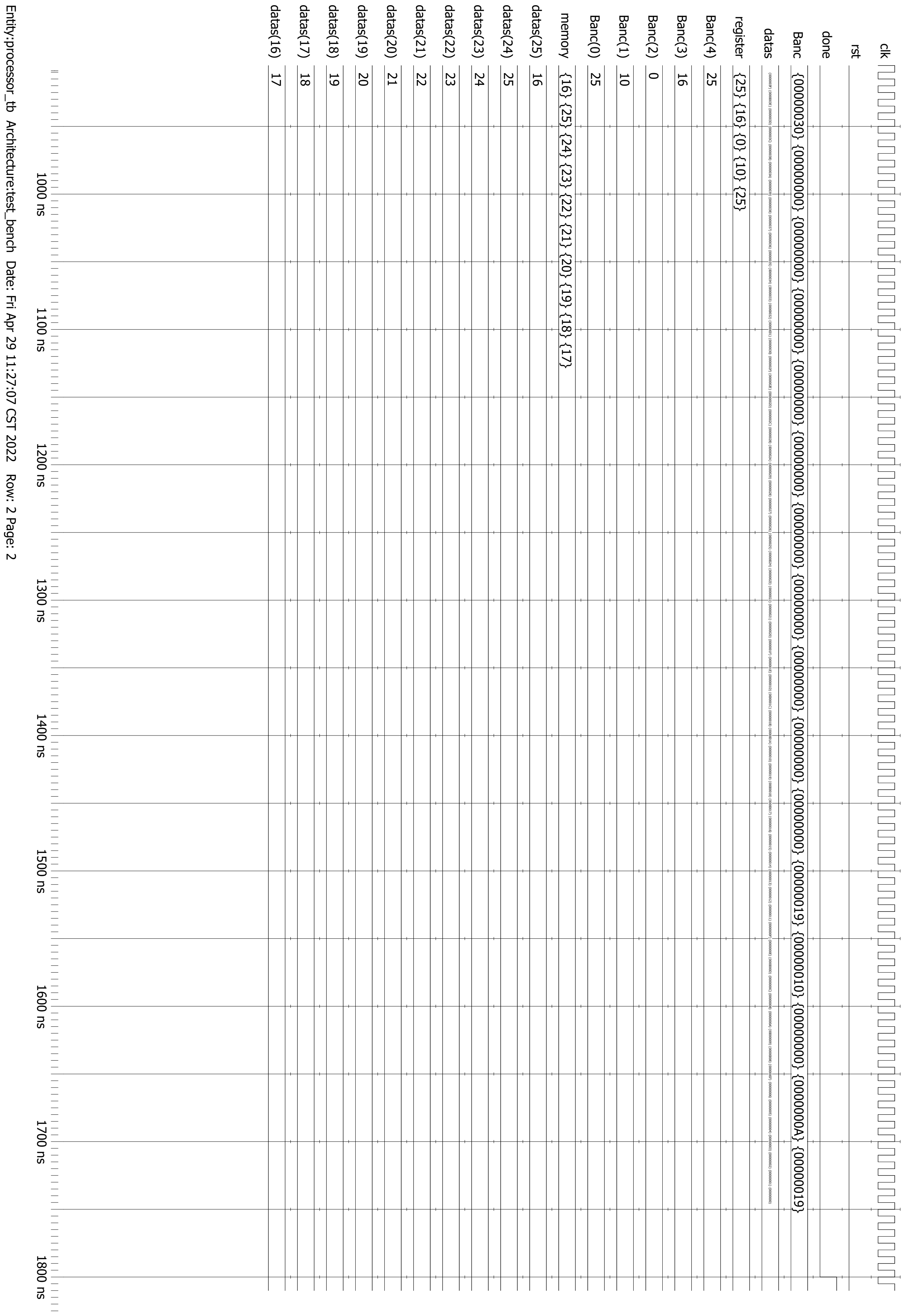
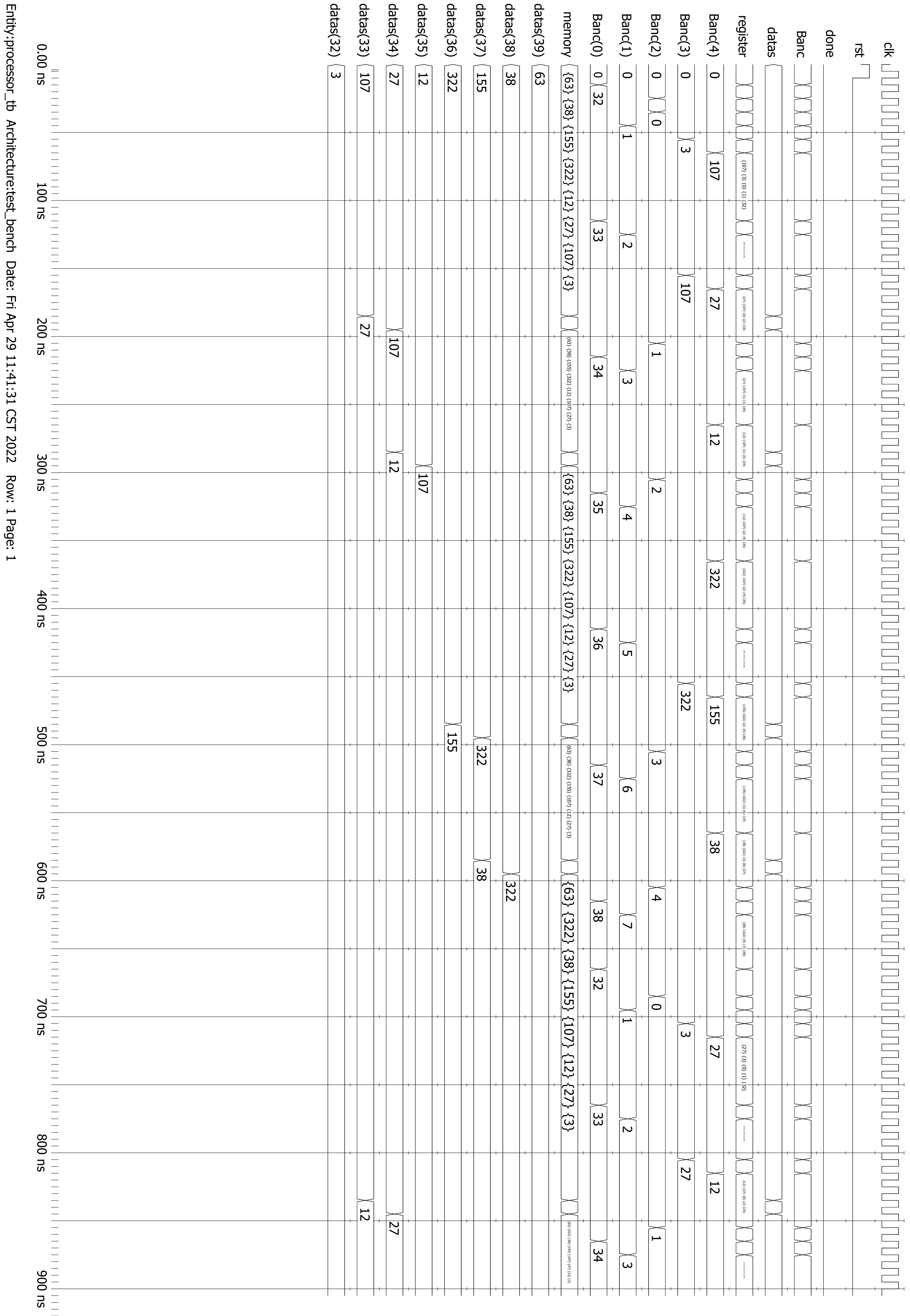


Figure 26: Simulation Waves of Processor Completely in Part 5


clk
rst
done
Banc
datas
register
Banc(4)
Banc(3)
Banc(2)
Banc(1)
Banc(0)
memory
datas(39)
datas(38)
datas(37)
datas(36)
datas(35)
datas(34)
datas(33)
datas(32)
{63} {38} {155} {322} {12} {27} {107} {3}
{63} {38} {155} {322} {107} {12} {27} {3}
{63} {322} {38} {155} {107} {12} {27} {3}
0.00 ns
100 ns
200 ns
300 ns
400 ns
500 ns
600 ns
700 ns
800 ns
900 ns
Entity:processor_tb Architecture:test_bench Date: Fri Apr 29 11:41:31 CST 2022 Row: 1 Page: 1

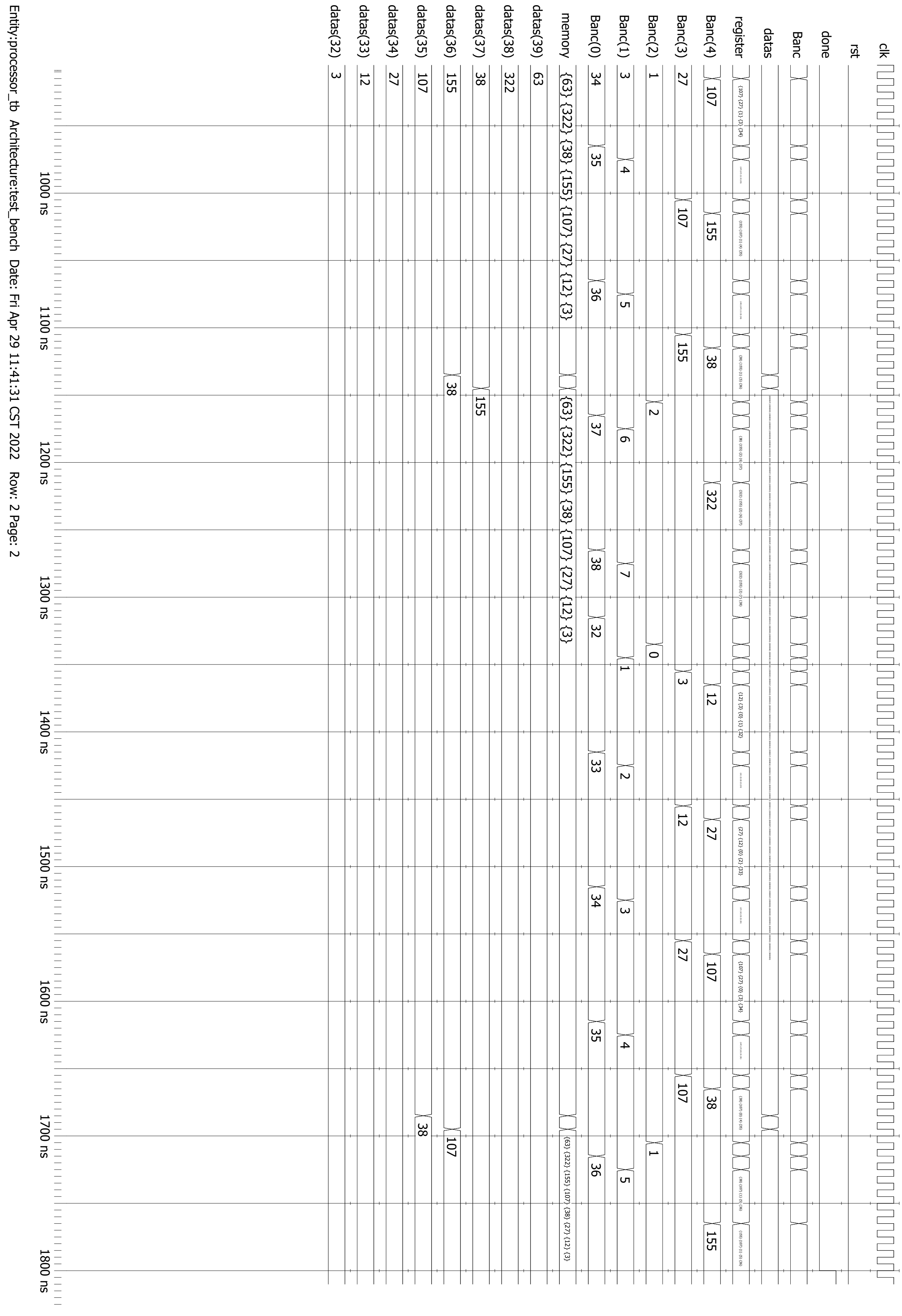


Figure 27: Simulation Waves of Processor with IS Increasing in Part 6

References

[1] John Catsoulis. *Designing Embedded Hardware: Create New Computers and Devices.* O'Reilly Media, Inc., 2005.

[2] J.D. Dumas. *Computer Architecture: Fundamentals and Principles of Computer Design.* Taylor & Francis, 2005.