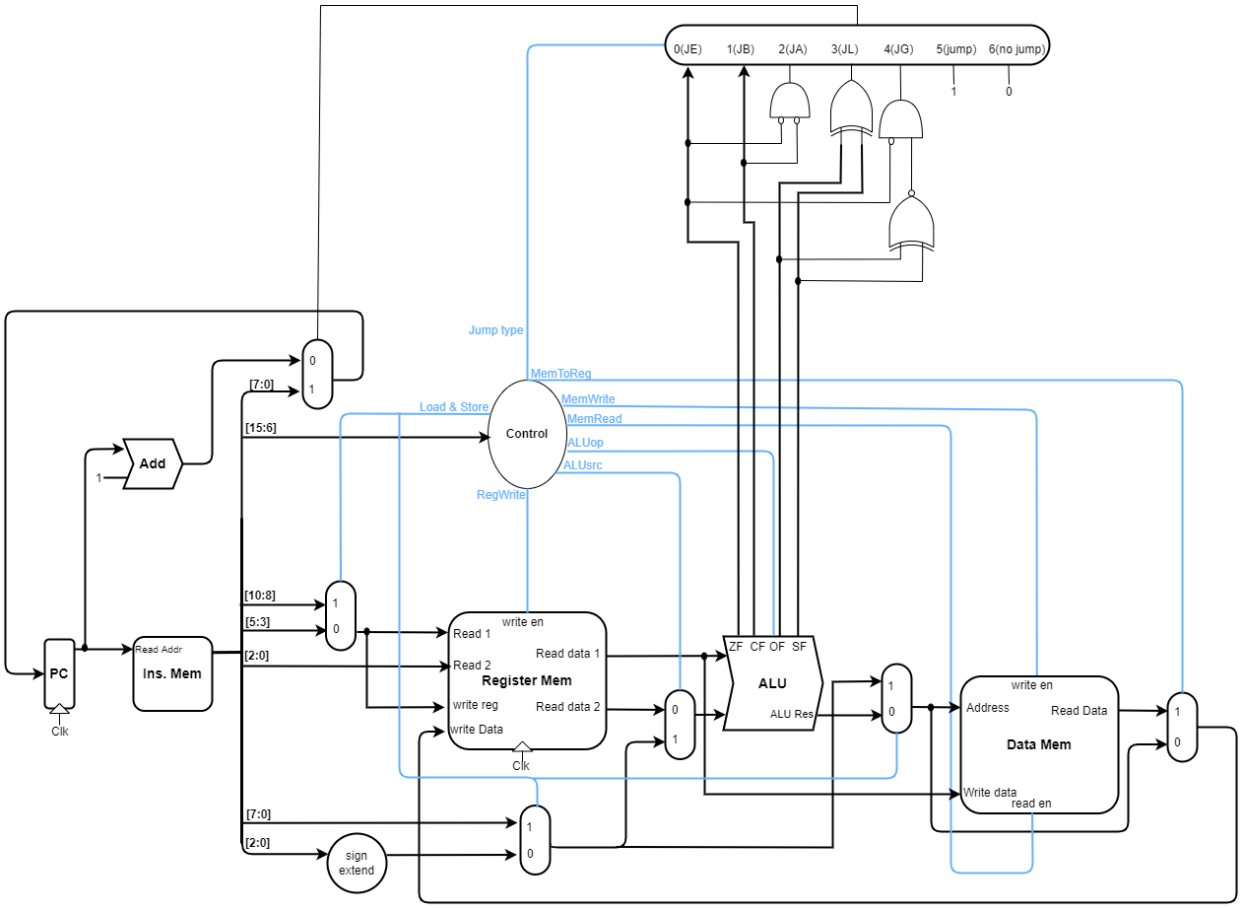
بسمه تعالی

گروه ۲ :
شمیسا آشکار ۹۹۲۴۳۰۱۲
نازنین نوراللهی ۹۹۲۴۳۰۷۲
محمدرضا عسگری ۹۹۲۴۳۱۰۸

DL-CA Lab-2
Project 07
Single cycle Processor
۱۴۰۱/۰۳/۱۶

طراحی Datapath پردازنده تک سیکل:



در ابتدا program counter قرار دارد که در لبه های بالارونده کلاک آدرس مکانی که باید دستور بعدی از آن خوانده شود را روی خروجی خود قرار میدهد که در ابتدا به خانه صفر حافظه im اشاره میکند. در دفعه های بعدی در هر کلاک یک واحد زیاد شده یا طبق دستورات وارد شده پردازنده در صورت وجود jump و همچنین taken بودن آن،آدرس جدیدی در pc قرار خواهد گرفت.

حافظه im آدرس ورودی را گرفته و بیت های دستور را روی بیت لاین قرار میدهد. سپس آن بخش های بیت ها که مربوط به بخش های مختلف نظیر read register ها در rm و همچنین write register و قسمت imm و آدرس jump ، است تقسیم بندی میشود.

بخش imm برای دستورات load & store با imm دستورات دیگر تفاوت دارد برای همین در بخش انتخاب imm یک mux قرار دارد که از میان آنها انتخاب کند.

این imm در صورت load یا store بودن دستور لازم است تا روی آدرس ورودی dm قرار گیرد تا از حافظه اصلی داده خوانده شود یا نوشته شود.

واحد alu دو ورودی خود را پردازش کرده و علاوه بر نمایش حاصل آنها ، flag ها را نیز تحت تاثیر خود قرار میدهد. این flag ها برای بررسی taken بودن jump های شرطی مورد بررسی قرار میگیرد. سیگنال های این flag ها بر اساس شرط هر jump خاص، وارد یک mux ۷ به ۱ میشوند. شرط های دستورات بدین صورت است:

Je => if (zf==1)

Jb => if (cf==1)

Ja => if (cf==zf==0)

Jl => if (sf != of)

Jg => if (sf == of)&&(zf == 0)

Jmp => no condition (always taken)

که این شرط ها با گیت های منطقی پیاده سازی شده اند.(در صورت غیر jump بودن دستور ورودی، واحد کنترلی مقدار سلکتور این مالتی پلکسر را ۶ قرار داده که مقدار صفر در آن قرار دارد و پرش رخ نمیدهد. همچنین در صورت jmp بودن دستور، همیشه ۱ روی خروجی میرود و پرش همیشه رخ میدهد.)

سلکتور این ماکس از واحد کنترلی گرفته میشود که واحد کنترلی با opcode هر دستور بررسی میکند که کدام jump روی ورودی قرار گرفته است. سپس سیگنال taken بودن این jump از mux انتخاب شده و به mux انتخاب کننده pc قدیمی یا ادرس جدید حافظه دستورات که از jump گرفته شده است، میرود.

سیگنال های واحد کنترلی:

op	9	8	7	6	5	4	3	2	1	0	Alusrc	mmtoreg	regwri	memred	memwrite	jum[2:0]	l_s	aluop[3:0]
add	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	6	0	1
and	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	6	0	2
sub	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	6	0	3
or	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	6	0	4
xor	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	6	0	5
move	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	6	0	6
not	0	0	0	0	0	0	1	0	0	0	X	0	1	0	0	6	0	8
sar	0	0	0	0	0	0	1	0	0	1	1	0	1	0	0	6	0	9
slr	0	0	0	0	0	0	1	0	1	0	1	0	1	0	0	6	0	10
sal	0	0	0	0	0	0	1	0	1	1	1	0	1	0	0	6	0	11
sll	0	0	0	0	0	0	1	1	0	0	1	0	1	0	0	6	0	12
rol	0	0	0	0	0	0	1	1	0	1	1	0	1	0	0	6	0	13
ror	0	0	0	0	0	0	1	1	1	0	1	0	1	0	0	6	0	14
inc	0	0	0	0	0	0	1	1	1	1	X	0	1	0	0	6	0	15
dec	0	0	0	0	0	1	0	0	0	0	X	0	1	0	0	6	0	7
nop	0	0	0	0	0	0	0	0	0	0	X	0	0	0	0	6	0	0
comp	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	6	0	3
je	1	0	0	0	0	X	X	X	-	-	X	0	0	0	0	8:5	0	0
jb	1	0	0	0	1	X	X	X	-	-	X	0	0	0	0	8:5	0	0
ja	1	0	0	1	0	X	X	X	-	-	X	0	0	0	0	8:5	0	0
jl	1	0	0	1	1	X	X	X	-	-	X	0	0	0	0	8:5	0	0
jg	1	0	1	0	0	X	X	X	-	-	X	0	0	0	0	8:5	0	0
jmp	1	0	1	0	1	X	X	X	-	-	X	0	0	0	0	8:5	0	0
li	1	1	0	0	0	X	X	X	-	-	X	0	1	0	0	6	1	6
lm	1	1	0	0	1	X	X	X	-	-	X	1	1	1	0	6	1	0
sm	1	1	0	1	0	X	X	X	-	-	X	0	0	0	1	6	1	0

کد Control Unit:

```
module Control(op, AluSrc, MemToReg, RegWrite, MemRead, MemWrite, Jump, LS, Aluop);
    input[9:0] op;
    output reg AluSrc;
    output reg MemToReg;
    output reg RegWrite;
    output reg MemRead;
    output reg MemWrite;
    output reg [2:0] Jump;
    output reg LS;
    output reg [3:0] Aluop;
always@(op)begin
    if(op[9]==0)begin
        AluSrc =(op[3]==1)? 1 : 0;
        MemToReg = 0;
        RegWrite =(op==0)? 0 :
                    (op[4:2]==3'b101)? 0:1;
        MemRead = 0;
        MemWrite    = 0;
        Jump = 6;
        LS = 0;
        if(op[4]==0)
            Aluop=op[3:0];
        else if(op[4]==1)
            Aluop=(op[3:0]==0)? 7:3;
    end
    else if(op[9]==1)begin
        AluSrc = 0;
        MemToReg =(op[8:5]==4'b1001)? 1:0;
        RegWrite =(op[8:5]==4'b1001)? 1 :
                    (op[8:5]==4'b1000)? 1:0;
        MemRead =(op[8:5]==4'b1001)? 1:0;
        MemWrite    =(op[8:5]==4'b1010)? 1:0;
        Jump = (op[8]==1)? 6: op[8:5];
        LS =(op[8]==1)? 1:0;
        Aluop =(op[8:5]==4'b1000)? 6:0;
    end
end
endmodule
```

کد Instruction Memory :

```
`timescale 1ns / 1ps

module Instruction_Memory(Address, Read_data);
    input[7:0] Address;
    output[15:0] Read_data;
    reg[15:0] Mem[255:0];

    assign Read_data = Mem[Address];

endmodule
```

کد Register File :

```
`timescale 1ns / 1ps

module Register_Bank(clk, RegWrite, ReadReg1, ReadReg2, WriteReg, Write_data, Read_data1, Read_data2);
    input clk;
    input RegWrite;
    input[2:0] ReadReg1;
    input[2:0] ReadReg2;
    input[2:0] WriteReg;
    input[7:0] Write_data;
    output[7:0] Read_data1;
    output[7:0] Read_data2;

    reg[7:0] Registers[7:0];

    assign Read_data1=Registers[ReadReg1];
    assign Read_data2=Registers[ReadReg2];

    always@(posedge clk)begin
        if(RegWrite==1)
            Registers[WriteReg] = Write_data;
    end

endmodule
```

کد ALU :

```
module alu_8bit(a, b, func, zf, of, cf, sf, res);
input[7:0] a,b;
input[3:0] func;
output reg zf,of,cf,sf;
output reg[7:0] res;

always @(func or a or b) begin
    case(func)
     //no_op
    0:res=0;

     //add
    1:begin
        {cf,res} = a + b;
        sf= res[7];
        zf=(res==0)? 1 : 0;
        of=(res[7]!=(a[7]==b[7]))? 1 : 0;
     end

     //and
     2:begin
        res = a & b;
        sf= res[7];
        zf=(res==0)? 1 : 0;
        of = 0;
        cf = 0;
     end

     //sub - comp
    3:begin
        {cf,res} = a - b;
        sf= res[7];
        zf=(res==0)? 1 : 0;
        of=(res[7]!=(a[7]==b[7]))? 1 : 0;
     end

     //or
    4:begin
        res = a | b;
        sf= res[7];
        zf=(res==0)? 1 : 0;
        of = 0;
        cf = 0;
     end

     //xor
    5:begin
        res = a ^ b;
        sf= res[7];
        zf=(res==0)? 1 : 0;
        of = 0;
        cf = 0;
     end

     //move - li
    6: res=b;

     //dec
    7:begin
        {cf,res} = a - 1;
        sf= res[7];
        zf=(res==0)? 1 : 0;
        of=(res[7]!=(a[7]==b[7]))? 1 : 0;
     end

     //not
     8: res=~a;
```

```
70
71      //sar
72      9:begin
73         {res[6:0], cf} = a >> b;
74         res[7] = a[7];
75         of=0;
76         sf = res[7];
77         zf=(res==0)? 1 : 0;
78      end
79
80      //slr
81      10:begin
82         {res, cf} = a >> b;
83         of = (res[7] != a[7])? 1 : 0;
84         sf = res[7];
85         zf=(res==0)? 1 : 0;
86      end
87
88      //sal
89      11:begin
90         {cf, res} = a << b;
91         of = (res[7] != a[7])? 1 : 0;
92         sf = res[7];
93         zf=(res==0)? 1 : 0;
94      end
95
96      //sll
97      12:begin
98         {cf, res} = a << b;
99         of = (res[7] != a[7])? 1 : 0;
100        sf = res[7];
101        zf=(res==0)? 1 : 0;
102     end
103
```

```
104     //rol
105     13:begin
106        res= {a, a} >> b[2:0];
107        of = (res[7] != a[7])? 1 : 0;
108        cf = res[7];
109        sf = res[7];
110        zf =(res==0)? 1 : 0;
111     end
112
113     //ror
114     14:begin
115        res= {a, a} << b[2:0];
116        of = (res[7] != a[7])? 1 : 0;
117        cf = res[7];
118        sf = res[7];
119        zf =(res==0)? 1 : 0;
120     end
121
122     //inc
123     15:begin
124        {cf,res} = a + 1;
125        sf= res[7];
126        zf=(res==0)? 1 : 0;
127        of=(res[7]!=(a[7]==b[7]))? 1 : 0;
128     end
129  endcase
130  end
131
132  endmodule
133
```

کد Data Memory :

```
`timescale 1ns / 1ps

module Data_Mem(MemRead, MemWrite, Address, Write_data, Read_data);
    input MemRead;
    input MemWrite;
    input[7:0] Address;
    input[7:0] Write_data;
    output reg[7:0] Read_data;

    reg[7:0] Mem[255:0];

    always@(MemWrite or MemRead) begin
        if(MemWrite==1)
            Mem[Address]=Write_data;
        else if(MemRead==1)
            Read_data = Mem[Address];
    end

endmodule
```

کد Program Counter :

```
`timescale 1ns / 1ps

module Program_counter(clk, in, out);
input clk;
input [7:0] in;
output reg[7:0] out;

    integer i;
    initial i=0;

    reg[7:0] pc;
    always@(posedge clk)begin
        if(i==0)begin
            out = 0;
            i = 1;
            end
        else begin
            pc = in;
            out = pc;
            end
end
endmodule
```

کد Mux2to1 : در بعضی قسمت های datapath ماکس مورد نیاز ما ۳ بیت یا ۸ بیت بوده است که برای نمونه کد یکی از آنها را قرار میدهیم.

```
`timescale 1ns / 1ps

module Mux2to1_8bit(a,b,sel,out);
input[7:0] a,b;
input sel;
output reg[7:0] out;

always @(sel or a or b) begin
    case(sel)
    0:   out=a;
    1:   out=b;
    endcase
end

endmodule
```

کد Mux7to1 :

```
`timescale 1ns / 1ps

module Mux7to1_1bit(a,b,c,d,e,sel,out);
input a,b,c,d,e;
input[2:0] sel;
output reg out;

always @(sel or a or b or c or d or e) begin
    case(sel)
    0:   out=a;
    1:   out=b;
    2:   out=c;
    3:   out=d;
    4:   out=e;
    5:   out=1;
    6:   out=0;
    endcase
end
endmodule
```

کد Adder pc :

```
`timescale 1ns / 1ps

module Add(a, out);
input[7:0] a;
output [7:0] out;

assign out=a+1;

endmodule
```

در نهایت لازم است تا از بخش های مختلف ساخته شده، نمونه سازی کرد و در یک فایل نهایی wire های لازم را وصل کرد تا cpu نهایی ساخته شود. تنها ورودی آن کلاک است.

```
module CPU(input clk);

wire [7:0] newpc_in;
wire [7:0] newpc_out;
Program_counter pc(clk, newpc_in, newpc_out);

wire [15:0] instruction;
Instruction_Memory im(newpc_out, instruction);

wire alusrc,memtoreg,regwrite,memread,memwrite,load_store;
wire[2:0] jump_type_wire;
wire[3:0] aluop;
Control control_unit(instruction[15:6], alusrc, memtoreg, regwrite, memread, memwrite, jump_type_wire, load_store, aluop);

wire[7:0] added_pc;
Add adder(newpc_out, added_pc);

wire jump_taken;
Mux2to1_8bit mux_jump(added_pc, instruction[7:0], jump_taken, newpc_in);

wire[2:0] reg_read1;
Mux2to1_3bit mux_reg_read1(instruction[5:3], instruction[10:8], load_store, reg_read1);

wire[7:0] write_backto_reg, read_reg1, read_reg2;
Register_Bank rm(clk, regwrite, reg_read1, instruction[2:0], reg_read1, write_backto_reg, read_reg1, read_reg2);

wire[7:0] immediate_wire;
Mux2to1_8bit mux_immediate( {{5{instruction[2]}},instruction[2:0]}, instruction[7:0], load_store, immediate_wire);

wire[7:0] alu_operand2;
Mux2to1_8bit mux_srcALU(read_reg2, immediate_wire, alusrc, alu_operand2);

wire of,zf,sf,cf;
wire[7:0] alu_res;
alu_8bit ALU(read_reg1, alu_operand2, aluop, zf, of, cf, sf, alu_res);

 wire[7:0] address;
 Mux2to1_8bit mux_mem_address(alu_res, immediate_wire, load_store, address);

 Mux7to1_1bit jump_set_mux(zf, cf, (~zf)&(~cf), of^sf, (~zf)&(~(of^sf)), jump_type_wire, jump_taken);

 wire[7:0] data_out_mem;
 Data_Mem dm(memread, memwrite, address, read_reg1, data_out_mem);

 Mux2to1_8bit mux_write_back(address, data_out_mem, memtoreg, write_backto_reg);

 endmodule
```

برای نوشتن برنامه لازم است تا درون instruction memory دستورات خود را به صورت باینری ذخیره نماییم. ابتدا دستورات دلخواه را مینویسیم:

```
[0]    li $r0,0     # i=0
[1]    li $r1,4     # loop from 0 to 4 => 5 times
[2]    li $r3,10
loop:
[3]    je end_Loop      #jump if alu flag zero is 1
[4]    add $r3,$r3      #every time add r3 to itself
[5]    inc $r0          #i=i+1
[6]    comp $r1,$r0     #r1-r0 and set flages
[7]    jump loop        #jump with no condition
end_loop:
[8]    sm $r3,200       #store result in mem[200]
```

معادل باینری آنها را در خانه های حافظه ذخیره میکنیم:

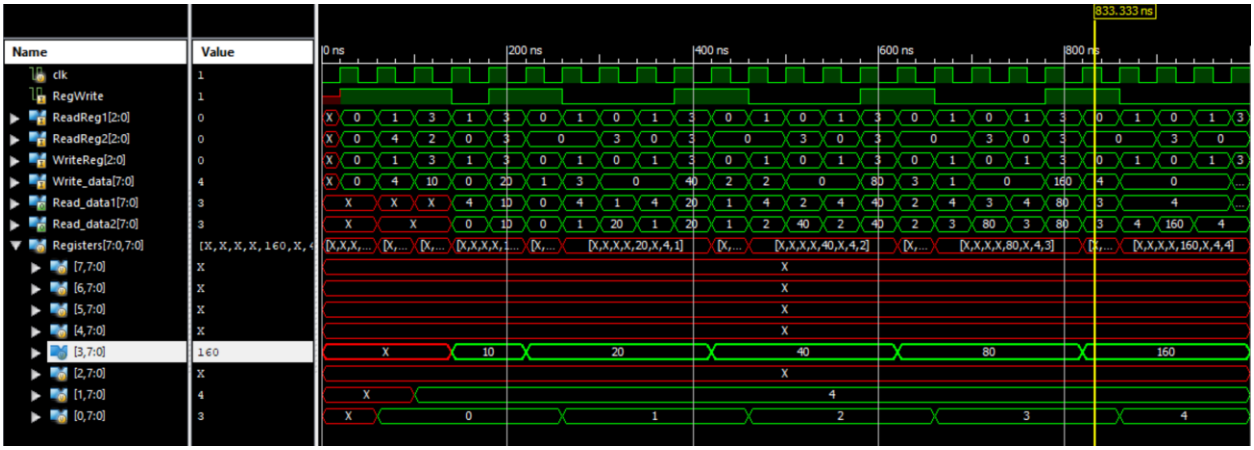
```
`timescale 1ns / 1ps

module Instruction_Memory(Address, Read_data);
    input[7:0] Address;
    output[15:0] Read_data;
    reg[15:0] Mem[255:0];

    assign Read_data = Mem[Address];
    initial Mem[0]=16'b1100000000000000;    //li $r0,0
    initial Mem[1]=16'b1100000100000100;    //li $r1,4
    initial Mem[2]=16'b1100001100001010;    //li $r3,10
    initial Mem[3]=16'b1000000000001000;    //je end_Loop
    initial Mem[4]=16'b0000000001011011;    //add $r3,$r3
    initial Mem[5]=16'b0000001111000000;    //inc $r0
    initial Mem[6]=16'b0000010100001000;    //comp $r1,$r0
    initial Mem[7]=16'b1010100000000011;    //jump loop
    initial Mem[8]=16'b1101001111001000;    //sm $r3,200
endmodule
```

در تست بنچ تنها لازم است تا

کلاک ورودی را تنظیم نماییم.

```
`timescale 1ns / 1ps


module test_bench;
    reg clk;

    CPU uut (
        .clk(clk)
    );

    initial begin
        clk = 0;
        forever #20 clk=~clk;
    end

endmodule
```

شرایط رجیستر ها و حافظه پس از اجرای دستورات:

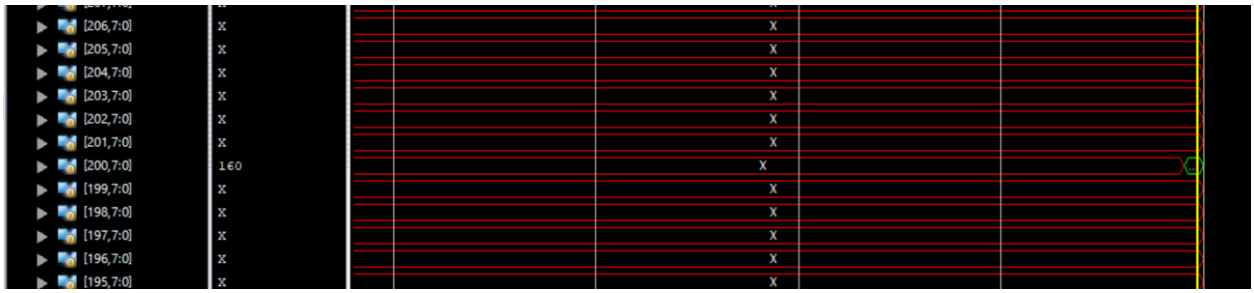


در این دستورات، ابتدا رجیستر شماره صفر به عنوان شمارنده به عدد صفر تعیین میشود.

سپس رجیستر شماره ۲ به عنوان سقفی برای اجرای حلقه به عدد ۴ تعیین میشود.

سپس در رجیستر شماره ۳ عدد ۱۰ ذخیره شده و در هر دور حلقه، رجیستر ۳ با خودش جمع شده و در خودش ذخیره میشود.

همچنین شمارنده یک واحد افزایش میابد. تا وقتی که به ۴ میرسد و دستور پرش که در هنگام ۱ شدن سیگنال ZF انجام میشود اتفاق میوفتد، و به دستور ذخیره رجیستر شماره ۳ در خانه ۲۰۰ حافظه میرویم.



میبینیم که عدد ۱۶۰ در خانه ۲۰۰ حافظه ذخیره شده است.