Computer Organization and Architecture

Course Design

Microprogrammed CPU Design

04015XXX CCC

04015XXX NNN

School of Information Science and Engineering

Southeast University

April 2018

Content

1. Purpose

The purpose of this project is to design a simple CPU (Central Processing Unit). This CPU has basic instruction set, and we will utilize its instruction set to generate a very simple program to verify its performance. For simplicity, we will only consider the relationship among the CPU, registers, memory and instruction set. That is to say we only need consider the following items: *Read/Write Registers, Read/Write Memory and Execute the instructions.*

At least four parts constitute a simple CPU: ***the control unit, the internal registers, the ALU and instruction set***, which are the main aspects of our project design and will be studied.

2. Instruction Set

Single-address instruction format is used in our simple CPU design. The instruction word contains two sections: *the operation code* (opcode), which defines the function of instructions (addition, subtraction, logic operations, etc.); *the address part*, in most instructions, the address part contains the memory location of the datum to be operated, we called it *direct addressing*. In some instructions, the address part is the operand, which is called *immediate addressing*.

For simplicity, the size of memory is 256×16 in the computer. The instruction word has 16 bits. The opcode part has 8 bits and address part has 8 bits. The instruction word format can be expressed in **Figure .**

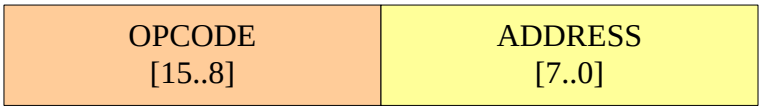


Figure 1 the instruction format

The opcode of the relevant instructions are listed in **Table 1**.

In **Table 1**, the notation [x] represents the contents of the location x in the memory. For example, the instruction word 0000001110111001_2 ($03B9_{16}$) means that the CPU adds word at location $B9_{16}$ in memory into the accumulator (ACC); the instruction word 0000010100000111_2 (0507_{16}) means if the sign bit of the ACC (ACC [15]) is 0, the CPU will use the address part of the instruction as the address of next instruction, if the sign bit is 1, the CPU will increase the program counter (PC) and use its content as the address of the next instruction.

Table 1 List of instructions and relevant opcodes

INSTRUCTION	OPCODE	COMMENTS
ADD X	01	$ACC + [X] \rightarrow ACC$
NOT X	02	NOT $[X] \rightarrow ACC$
SUB X	03	$ACC - [X] \rightarrow ACC$
LSR	04	SHIFT ACC to Right 1bit, Logic Shift
MUL X	05	$ACC \times [X] \rightarrow ACC$, MR
LSL	06	SHIFT ACC to Left 1bit, Logic Shift
AND X	07	ACC and $[X] \rightarrow ACC$
ASR	08	SHIFT ACC to Right 1bit, Arithmetic Shift
OR X	09	ACC or $[X] \rightarrow ACC$
ASL	0A	SHIFT ACC to Left 1bit, Arithmetic Shift
STORE X	0B	$ACC \rightarrow [X]$
LOAD X	0C	$[X] \rightarrow ACC$

JMPGEZ X	0D	If ACC ≥ 0 then X $\rightarrow$ PC else PC+1 $\rightarrow$ PC
JMP X	0E	X $\rightarrow$ PC
HALT	0F	Halt a program

3. Internal Registers and Memory

MAR *(Memory Address Register)*

MAR contains the memory location of the word to be read from the memory or written into the memory. Here, READ operation is denoted as the CPU reads from memory, and WRITE operation is denoted as the CPU writes to memory. In our design, MAR has 8 bits to access one of 256 addresses of the memory.

MBR *(Memory Buffer Register)*

MBR contains the value to be stored in memory or the last value read from memory. MBR is connected to the address lines of the system bus. In our design, MBR has 16 bits.

PC *(Program Counter)*

PC keeps track of the instructions to be used in the program. In our design, PC has 8 bits.

IR *(Instruction Register)*

IR contains the opcode part of an instruction. In our design, IR has 8 bits.

BR *(Buffer Register)*

BR is used as an input of ALU, it holds other operand for ALU. In our design, BR has 16 bits.

ACC *(Accumulator)*

ACC holds one operand for ALU, and generally ACC holds the calculation result of ALU. In our design, ACC has 16 bits.

MR *(Multiplier Register)*

MR is used for implementing the MPY instruction, holding the multiplier at the beginning of the instruction. When the instruction is executed, it holds part of the product.

CM

CM is a RAM with separate input and output ports, it works as memory, and its size is 256×16. Although it's not an internal register of CPU, we need it to simulate and test the performance of CPU.

All the registers are positive-edge-triggered.

All the reset signals for the registers are synchronized to the clock signal.

ALU

ALU (Arithmetic Logic Unit) is a calculation unit which accomplishes basic arithmetic and logic operations. In our design, some operations must be supported which are listed as follows

Table 2 Sample ALU Operations

Operations	Explanations
ADD	$(ACC) \Leftarrow (ACC) + (BR)$
SUB	$(ACC) \Leftarrow (ACC) - (BR)$
AND	$(ACC) \Leftarrow (ACC)$ and (BR)
OR	$(ACC) \Leftarrow (ACC)$ or (BR)
NOT	$(ACC) \Leftarrow$ Not (ACC)
SRL	$(ACC) \Leftarrow$ Shift (ACC) to Left 1 bit
SRR	$(ACC) \Leftarrow$ Shift (ACC) to Right 1 bit

4. Microprogrammed Control Unit

We have learnt the knowledge of Microprogrammed control unit. Here, we only review some terms and basic structures.

In the Microprogrammed control, the microprogram consists of some microinstructions and the microprogram is stored in control memory that generates all the control signals required to execute the instruction set correctly. The microinstruction contains some micro-operations which are executed at the same time.

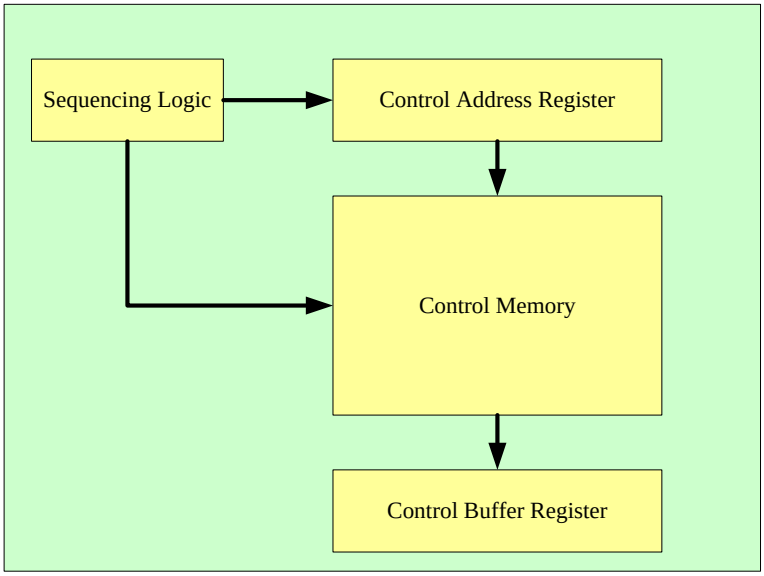


Figure 2 Control Unit Micro-architecture

Figure shows the key elements of such an implementation. The set of microinstructions is stored in the control memory. The control address register contains the address of the next microinstructions to be read. When a microinstruction is read from the control memory, it is transferred to a control buffer register. The register connects to the control lines emanating from the control unit. Thus, reading a microinstruction from the control memory is the same as executing that microinstruction. The third element shown in the figure is a sequencing unit that loads the control address register and issues a read command.

4.1. CPU Design and Control Signals

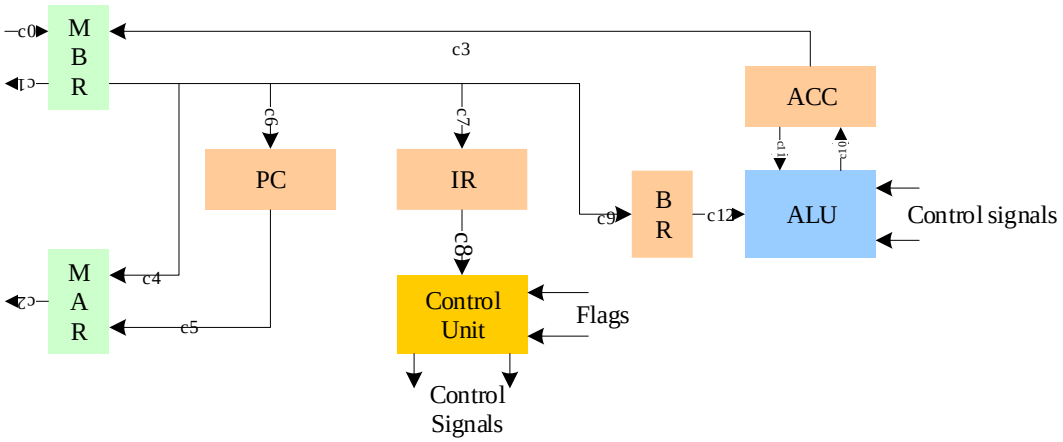


Figure 3 CPU data path and control signals

Figure indicates a simple CPU architecture and its use of a variety of internal data paths and

control signals. Our CPU design should be based on this architecture.

According to the diagram, the control signals are used to moderate CPU behavior outside the Control Unit, the control of the control unit itself is accomplished through internal signal bus. In this way, we use only 16 bits of control signal outside the CU to realize the control of the whole CPU, which is a very economical way. Our control words are 16 bits wide.

Table 3 Control Bit for the Micro-operations

Bit in Control Memory	***Micro-operation***	***Meaning***
c0	**MBR<=RAM**	RAM to MBR
c1	**RAM<=MBR**	MBR to RAM (i.e. Write)
c2		
c3	**MBR<=ACC**	Copy ACC to MBR
c4	**MAR<=MBR[7..0]**	Copy MBR[7..0] to MAR for address
c5	**MAR<=PC**	Copy PC value to MAR for next address
c6	**PC<=MBR[7..0]**	Copy MBR[7..0] to PC, for jump.
c7	**IR<=MBR[15..8]**	Copy instruction to IR
c8	**C8<=C7**	Instruction from IR to CU
c9	**BR<-MBR**	Copy MBR data to BR for buffer to ALU
c10	**ACC<=ALU**	ALU answer copied to ACC
c11	**ALU<=ACC**	**ACC copied to ALU**
c12	**ALU<=BR**	Copy BR to ALU
Calu(c12,c13,c14,c15)	**ALU control signal.**	Tell ALU what operation to be performed.

Note: We have 10 ALU operations in this assignment and theoretically we need at least 4 bits to distinguish them all. However, we use only 3 independent bits. This is because c12 is related to ALU operations. Every operation taking 2 operands must set c12 to '1' to use the operand in BR, thus it can act as an ALU control signal at the same time! So, operations like ADD, SUB, MUL, etc. are having their C_lau as xxx1 (15 downto 12).

Then we need to determine the relevant control signals which are given below.

4.2. Micro-operations of instructions.

Table 4 Control signals for all the instructions and their location in CR

addr	operation	Micro-ops	control signals	Control bits
0	fetch	mar<=(pc)	c5	0020
1		mbr<=memory, pc<=(pc)+I	c0	0001
2		ir<=(mbr)	c7	0080
3			c8, (0x0100)	0100
4				0000
8	add	mar<=mbr_l(addr)	c4	0010
9		mbr<=memory	c0	0001
A		br<=mbr	c9	0200
B		ac<=ac+br	c12, c11, (calu incl.)	1800
C		(wait till transferred)	c10,	0400
D			0	0000
10	not	ac<=not ac	c13, c11, (calu incl.)	2800
11		(wait till transferred)	c10,	0400
12			0	0000
18	sub	mar<=mbr_l(addr)	c4	0010
19		mbr<=memory	c0	0001
1A		br<=mbr	c9	0200
1B		ac<=ac-br	c13, c12, c11, (calu incl.)	3800
1C		(wait till transferred)	c10,	0400
1D			0	0000
20	lsr	ac<=lsr ac	c14, c11, (calu incl.)	4800
21		(wait till transferred)	c10,	0400
22			0	0000
28	mul	mar<=mbr_l(addr)	c4	0010
29		mbr<=memory	c0	0001
2A		br<=mbr	c9	0200
2B		ac<=ac*br	c14, c12, c11, (calu incl.)	5800
2C		(wait till transferred)	c10,	0400
2D			0	0000
30	lsl	ac<=lsl ac	c14, c13, c11, (calu incl.)	6800
31		(wait till transferred)	c10,	0400
32			0	0000
38	and	mar<=mbr_l(addr)	c4	0010
39		mbr<=memory	c0	0001
3A		br<=mbr	c9	0200
3B		ac<=ac and br	c14, c13, c12, c11, (calu incl.)	7800

3C		(wait till transferred)	c10,	0400
3D			0	0000
40	asr	ac<=asr ac	c15, c11, (calu incl.)	8800
41		(wait till transferred)	c10,	0400
42			0	0000
48	or	mar<=mbr_l(addr)	c4	0010
49		mbr<=memory	c0	0001
4A		br<=mbr	c9	0200
4B		ac<=ac or br	c15, c12, c11, (calu incl.)	9800
4C		(wait till transferred)	c10,	0400
4D			0	0000
50	asl	ac<=asr ac	c15, c13, c11, (calu incl.)	A800
51		(wait till transferred)	c10,	0400
52			0	0000
58	store	mar<=mbr_l(addr)	c4,	0010
59		mbr<=ac	c3,	0008
5A		memory<=mbr	c1,	0002
5B			0	0000
60	load	mar<=mbr_l(addr)	c4, c15, c14, c13	E010
61		mbr<=memory, ac<=clear alu	c0, c10,	0401
62		br<=mbr	c9,	0200
63		ac<=ac or br	c15, c12, c11, (calu incl.)	9800
64		(wait till transferred)	c10,	0400
65			0	0000
68	jmpgez	pc<=mbr_l(addr)	c6,	0040
69			0	0000
70	jmp	pc<=mbr_l(addr)	c6,	0040
71			0	0000
78	halt		c15, c14, c13, c12	F000
79			0	0000

Note:

1. **As the control of CAR and CBR are internal issues inside the CU, so regular microoperations like "CAR<=CAR+1" and "CAR<=0" are not controlled by these signals.**
2. **The instructions take up 8 bytes each, which makes it easier to get them in CM.**
3. **Special Control bits: "0000" means going to fetch; "F000" means the end of the program; "0100" means going to the address of the fetched instructions.**
4. **As can be seen from above, C15-C12 has some special usages, so it's impossible to specify 16 arithmetic/logic operations with these 4 C_ALU bits (14 operations at max under current scheme).**
5. **JMPGEZ has the same content in CM as JMP as you cannot decide whether to jump or not by merely read the control signal out. My solution to this is to decide whether to jump**

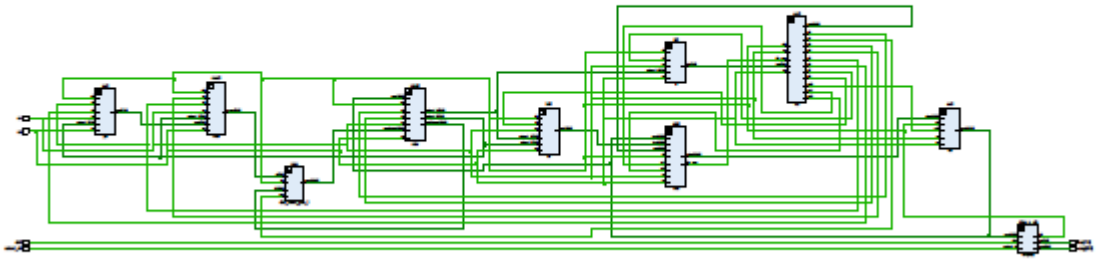
or not before issuing the control signal. If it does not jump, CAR<=CAR+2 (accomplished inside CU), which means the CPU goes to fetch the instruction right following JMPGEZ, else, it jumps to the given address.

5. Testing & Simulation & Verification

5.1. The overview of CPU

Double click to open.

CPU Schematic



5.2. Testing Program

We use the task on PPT (i.e. Calculate the sum of all integers from 1 to 100.) to test our CPU.

We can translate the C language program into the instruction program as shown in **Table 1**.

Table 5 Example of a program to sum from 1 to 100

Program with C	Program with instructions	Contents of Memory (RAM) in HEX	
		Address	Contents
sum=0;	LOAD 0	00	0c10
	STORE A4	01	0b11
temp=100;	LOAD 100	02	0c12
	STORE A3	03	0b13
loop :sum=sum+temp;	LOOP:LOAD A4	04	0c11
	ADD A3	05	0113
	STORE A4	06	0b11
temp=temp-1;	LOAD A3	07	0c13

	SUB 1	08	0314
	STORE A3	09	0b13
if temp>=0 goto loop;	JMPGEZ LOOP	0a	0d04
end	HALT	0b	0f00
		10	0000
		11	0000
		12	0064
		13	0000
		14	0001

Note: A3, A4, Loop, are variable names, and 0, 1, 100 are immediate numbers. The corresponding contents are done using self-written Python based compiler.

Another test: *Calculate the result of (2+4+…+30)SHL 2bit OR(1+2+3…+15)；*

Program with instructions	Contents of Memory (RAM) in HEX	
Of	Address	Contents
LOAD 0	00	0c20
STORE sum1	01	0b21
LOAD 30	02	0c22
STORE A3	03	0b23
LOOP:LOAD sum1	04	0c21
ADD A3	05	0123
STORE sum1	06	0b21
LOAD A3	07	0c23
SUB 2	08	0324
STORE A3	09	0b23
JMPGEZ LOOP	0a	0d04
load sum1	0b	0c21
asl	0c	0a00
asl	0d	0a00
store sum1	0e	0b21
LOAD 0	0f	0c20
STORE sum2	10	0b25
LOAD 15	11	0c26
STORE A4	12	0b27
LOOP2:LOAD sum2	13	0c25
ADD A4	14	0127

STORE sum2	15	0b25
LOAD A4	16	0c27
SUB 1	17	0328
STORE A4	18	0b27
JMPGEZ LOOP2	19	0d13
load sum2	1a	0c25
or sum1	1b	0921
store res	1c	0b29
halt	1d	0f00
	20	0000
	21	0000
	22	001e
	23	0000
	24	0002
	25	0000
	26	000f
	27	0000
	28	0001
	29	0000

5.3. Simulation of the CPU

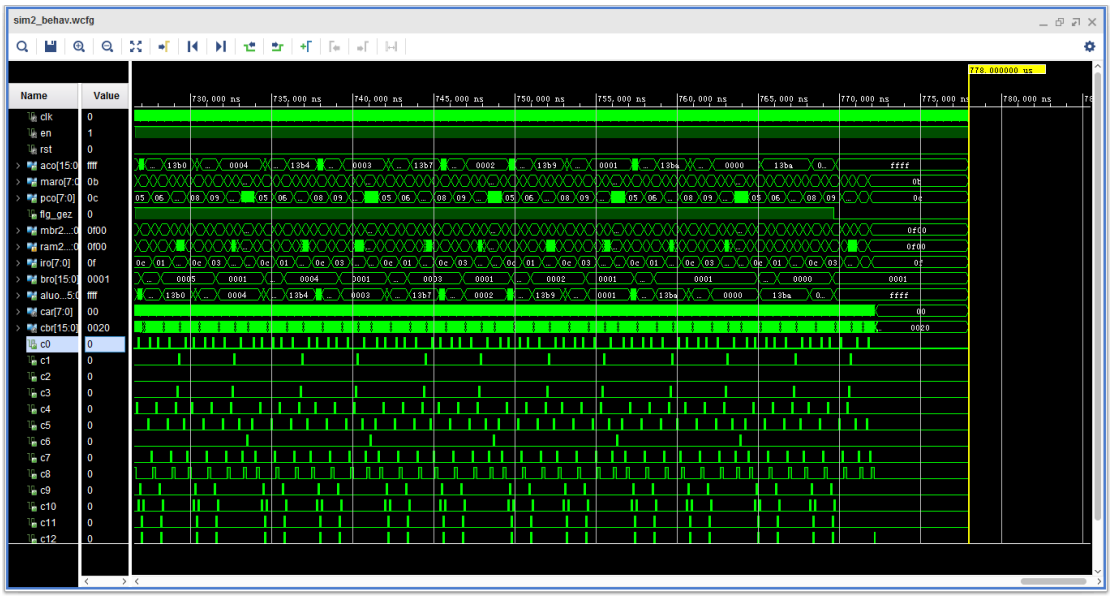


Figure 1 1+2+…+100 (end part). Result=5050=0x13ba

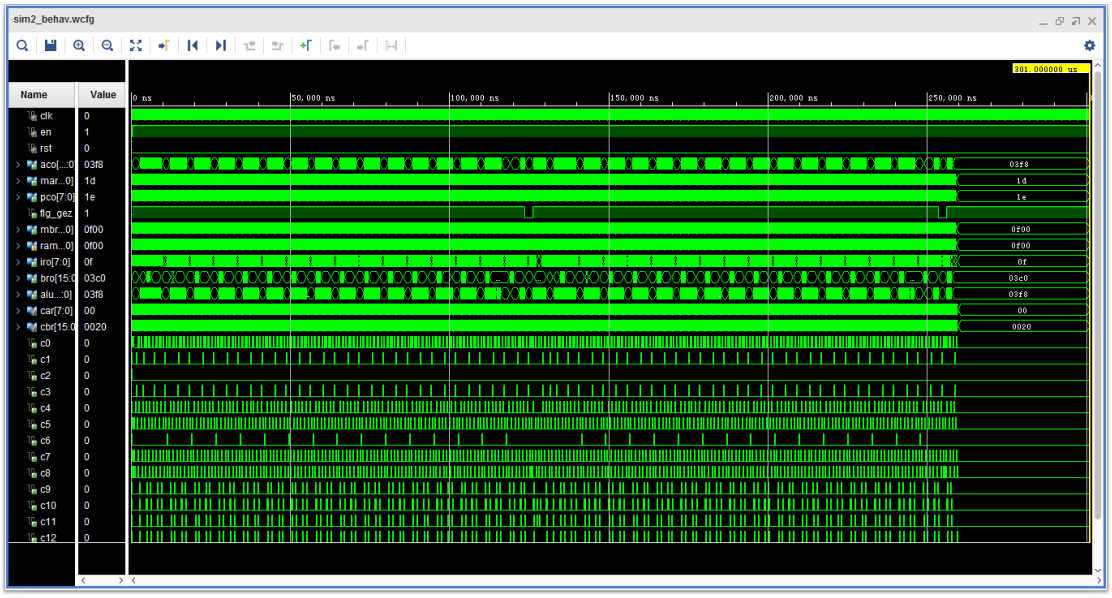


Figure 2 (2+4+…+30)SHL 2bit OR(1+2+3…+15). Result = (240)*4 or 120=960 or 120 = 1016 = 0x03f8

5.4. Verification

It is done on the FPGA experiment board.

6. BONUS: A Simple Compiler Based on Python 3.6

When testing our CPU, we found it dreary and erroneous to translate the ASM into machine code and assign the memory for constants and variables manually. So, using the powerful regex functions of Python, I wrote a simple compiler. As it is only a bonus part, we only assume that developers of our CPU write the correct ASM code, as there are numerous exceptions for a real complier to handle and we don't have so much time to deal with them all.

Example: sum from 1 to 100

1. The raw code using valid ASM (operation names may differ):

```
LOAD 0
STORE res              ; storage for result
 LOAD    0x64          ; i=100
STORE A3
 L1 :        LOAD res
   ADD A3
   STORE res
   LOAD A3
   SUB 1
   STORE A3
JMPGEZ  l1

HALT
```

As can be seen it has tabs, spaces, comments, and it doesn't assign memory spaces, only

variable names and constants are used.

2. The standardized code:

```
0       load 16
1       store 17
2       load 18
3       store 19
4       load 17
5       add 19
6       store 17
7       load 19
8       sub 20
9       store 19
10      jmpgez 4
11      halt
12
13
14
15
16      0
17
18      100
19
20      1
21
```

As can be seen, the constants and variable names are replaced by memory addresses (in decimal), and the constants are pre-assigned into particular places, and comments and redundant spaces are stripped. Specially, if you are using the same constant in several places, only one copy will be reserved.

3. The machine code:

```
00      0c10
01      0b11
02      0c12
03      0b13
04      0c11
05      0113
06      0b11
07      0c13
08      0314
09      0b13
0a      0d04
0b      0f00
0c      0000
0d      0000
0e      0000
0f      0000
10      0000
11      0000
12      0064
13      0000
14      0001
```

According to the method, the standardized code is transformed into machine code.

Python compiler code is attached to the appendix.

7. Appendix

7.1. CPU project file

```
----------------------------------------------------------------------------------
-- IR

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;

entity ir is
    Port(clk: in std_logic;
    rst: in std_logic;
    en: in std_logic;
    c7: in std_logic;
--    c8: in std_logic;

    mbro1_h:in std_logic_vector(7 downto 0);        --mbr's 1st output[15:8]
    iro:out std_logic_vector(7 downto 0)
    );
end ir;

architecture Behavioral of ir is

begin

process(rst,clk,en)
variable reg:std_logic_vector(7 downto 0):=x"00";
begin
    if rst='1' then
        reg:=x"00";
    elsif rising_edge(clk) and en='1' then
        if c7='1' then reg:=mbro1_h;
        end if;
    end if;
    iro<=reg;
end process;

end Behavioral;

----------------------------------------------------------------------------------
-- mar
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;

entity mar is
  Port(clk: in std_logic;
    rst: in std_logic;
    en: in std_logic;
```

```
--    c2: in std_logic;
    c4: in std_logic;
    c5: in std_logic;
    mbro1_l:in std_logic_vector(7 downto 0);        --mbr's 1st output[7:0]
    pco: in std_logic_vector(7 downto 0);
    maro: out std_logic_vector(7 downto 0)          --addr to RAM
    );
end mar;

architecture Behavioral of mar is

begin

p1:process(rst,clk,en)
begin
if rst='1' then
    maro<=x"00";
elsif rising_edge(clk) and en='1' then
    if c4='1' then maro<=mbro1_l;
    elsif c5='1' then maro<=pco;
    end if;
end if;

end process;
end Behavioral;


----------------------------------------------------------------------------------
-- mbr
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;

entity mbr is
    Port(clk: in std_logic;
    rst: in std_logic;
    en: in std_logic;
    c0: in std_logic;
--    c1: in std_logic;
    c3: in std_logic;

    aco: in std_logic_vector(15 downto 0);
    ram2mbr:in std_logic_vector(15 downto 0);
    mbr2ram:out std_logic_vector(15 downto 0);          --data to RAM
    mbro1_l:out std_logic_vector(7 downto 0);        --mbr's 1st output[7:0]
    mbro1_h:out std_logic_vector(7 downto 0)        --mbr's 1st output[15:8]
    );
end mbr;

architecture Behavioral of mbr is

begin

process(rst,clk,en)
```

```
variable reg:std_logic_vector(15 downto 0):=x"0000";
begin
    if rst='1' then
        reg:=x"0000";
    elsif rising_edge(clk) and en='1' then
        if c0='1' then reg:=ram2mbr;
        elsif c3='1' then reg:=aco;
        end if;
    end if;
    mbr2ram<=reg;
    mbro1_l<=reg(7 downto 0);
    mbro1_h<=reg(15 downto 8);
end process;
end Behavioral;


--------------------------------------------------------------------------------
-- pc
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;


entity pc is
    Port(clk: in std_logic;
    rst: in std_logic;
    en: in std_logic;
    c6: in std_logic;
    c5: in std_logic;

    mbro1_l:in std_logic_vector(7 downto 0);        --mbr's 1st output[15:8]
    pco:out std_logic_vector(7 downto 0)
    );
end pc;

architecture Behavioral of pc is

begin
process(rst,clk,en)
variable reg:std_logic_vector(7 downto 0):=x"00";
variable delay:std_logic:='0';
begin
    if rst='1' then
        reg:=x"00";
    elsif rising_edge(clk) and en='1' then
        if c6='1' then reg:=mbro1_l;
        elsif c5='1' and delay='0' then delay:='1';           --可能需要人工延
        elsif delay='1' then reg:=reg+1;delay:='0';
        end if;
    end if;
    pco<=reg;
end process;

end Behavioral;
```

```
156
157
158 ----------------------------------------------------------------------------
159 -- br
160 library IEEE;
161 use IEEE.STD_LOGIC_1164.ALL;
162
163 entity br is
164     Port(clk: in std_logic;
165     rst: in std_logic;
166     en: in std_logic;
167 --    c12: in std_logic;
168     c9: in std_logic;
169
170     mbro1_l:in std_logic_vector(7 downto 0);        --mbr's 1st output[7:0]
171     mbro1_h:in std_logic_vector(7 downto 0);        --mbr's 1st output[15:8]
172     bro:out std_logic_vector(15 downto 0)
173     );
174 end br;
175
176 architecture Behavioral of br is
177
178 begin
179
180 process(rst,clk,en)
181 variable reg:std_logic_vector(15 downto 0):=x"0000";
182 begin
183     if rst='1' then
184         reg:=x"0000";
185     elsif rising_edge(clk) and en='1' then
186         if c9='1' then
187             reg(15 downto 8):= mbro1_h;
188             reg(7 downto 0):= mbro1_l;
189         end if;
190     end if;
191     bro<=reg;
192 end process;
193
194 end Behavioral;
195
196
197 ----------------------------------------------------------------------------
198 -- alu
199 library IEEE;
200 use IEEE.STD_LOGIC_1164.ALL;
201 use ieee.std_logic_signed.all;
202
203
204 entity alu is
205     Port(clk: in std_logic;
206     rst: in std_logic;
207     en: in std_logic;
208     c11: in std_logic;
209     --    c10: in std_logic;
```

```
    c12: in std_logic;
    calu:in std_logic_vector(3 downto 0);

    -- we can add some flags.
    flg_gez: out std_logic;
    --
    aluo:out std_logic_vector(15 downto 0):=x"0000";
    aco:in std_logic_vector(15 downto 0);
    bro:in std_logic_vector(15 downto 0)
    );
end alu;

architecture Behavioral of alu is

begin
p0:process(clk)
variable reg32:std_logic_vector(31 downto 0):=x"00000000";
begin
    if rst='1' then
        reg32:=x"00000000";
    elsif rising_edge(clk) and en='1' then
        if calu=x"E" then reg32:=x"00000000";       -- clear alu when sent
        elsif c12='1' and c11='1' then     --double operand operations
            case calu is
                when x"1"=>             --add
                    reg32(15 downto 0):=aco+bro;      --signed or unsigned??
                when x"3"=>             --sub
                    reg32(15 downto 0):=aco-bro;
                when x"5"=>             --mul
                    reg32:=aco*bro;
                    if reg32(30 downto 16)="000000000000000" then
                        reg32(15):=reg32(31);
                    end if;
                when x"7"=>
                    reg32(15 downto 0):=aco and bro;
                when x"9"=>
                    reg32(15 downto 0):=aco or bro;
                when others=> null;
                end case;
        elsif c11='1' then              -- single operand operations
            case calu is
                when x"2"=>
                    reg32(15 downto 0):=not aco;
                when x"4"=>
                    reg32(15 downto 0):='0'&aco(15 downto 1);
                when x"6"=>
                    reg32(15 downto 0):=aco(14 downto 0)&'0';
                when x"8"=>
                    reg32(15 downto 0):=aco(15)&aco(15 downto 1);
                when x"a"=>
                    reg32(15 downto 0):=aco(15)&aco(13 downto 0)&'0';
                when others=>null;
                end case;
        end if;
```

```
264
265         if calu/=x"0" then
266             if reg32(15 downto 0)>=0 then flg_gez<='1';     -- 负数怎么表示?
267             else flg_gez<='0'; end if;
268         end if;
269 --        if reg32>"FFFF" then          --flags not set yet
270 --        if reg32=x"00000000" then
271     end if;
272
273     aluo<=reg32(15 downto 0);
274 end process p0;
275
276 end Behavioral;
277
278
279 ----------------------------------------------------------------------------------
280 -- cu
281 library IEEE;
282 use IEEE.STD_LOGIC_1164.ALL;
283 use ieee.std_logic_unsigned.all;
284
285 entity cu is
286     Port(
287     clk,en,rst: in std_logic;
288     --flgs
289     flg_gez: in std_logic;
290     ----
291     c0,c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12: out std_logic;
292     calu:out std_logic_vector(3 downto 0);
293     iro:in std_logic_vector(7 downto 0)
294     );
295 end cu;
296
297 architecture Behavioral of cu is
298 component dist_mem_gen_0 is
299     port(
300     a: in std_logic_vector(7 downto 0);
301     spo:out std_logic_vector(15 downto 0)
302     );
303 end component;
304 signal car: std_logic_vector(7 downto 0):=x"00";
305 signal cbr: std_logic_vector(15 downto 0):=x"0000";
306 begin
307
308 rom:dist_mem_gen_0 port map(a=>car,spo=>cbr);
309
310 sequencing:process(clk)
311
312 variable halted:std_logic:='0';
313 variable busy:std_logic :='0';
314 variable md:std_logic :='0';
315 begin
316     if rst='1' then
317         halted:='0';
```

```
318         car<=x"00";
319 --        cbr<=x"0000";       --FIXME
320         c0<='0';
321         c1<='0';
322         c2<='0';
323         c3<='0';
324         c4<='0';
325         c5<='0';
326         c6<='0';
327         c7<='0';
328         c8<='0';
329         c9<='0';
330         c10<='0';
331         c11<='0';
332         c12<='0';
333     elsif halted='1' then
334         car<=x"00";
335         c12<='0';
336     elsif rising_edge(clk) and en='1' then
337         if busy='0' and( cbr=x"0020" or cbr=x"0000")
338             car<=x"00";
339             busy:='1';
340         elsif  busy='1' then         --fetch 指令的微程序未结束前，一直执行
341             if cbr=x"0100" and md='0' then     --微操控制符，只有 c8 为 1，用在
342                 md:='1';
343             elsif cbr=x"0100" and md='1' then   --人工延时
344                 if iro(3 downto 0)=x"d" and flg_gez='0'then   --如果不需要条件
345                     car<='0'&iro(3 downto 0)&"001";
346                 else
347                     car<='0'&iro(3 downto 0)&"000";           --寻找 fetch 来的
348                 end if;
349                     md:='0';
350             elsif cbr/=x"0000" then
351                 car<=car+1;
352             end if;
353 --            if flg_gez='0' and car=x"68" then c6<='0';  --确定跳转时使用，for
354 --            else c6<=cbr(6); end if;
355             c0<=cbr(0);c1<=cbr(1);c2<=cbr(2);c3<=cbr(3);c4<=cbr(4);c5<=cbr(5);
356             c7<=cbr(7);c8<=cbr(8);c9<=cbr(9);c10<=cbr(10);c11<=cbr(11);c12<=cb
357             calu<=cbr(15 downto 12);
358 
359             if cbr=x"0000" then busy:='0';      --用于一般程序结尾
360             elsif cbr=x"f000" then halted:='1';
361             end if;
362         end if;
363     end if;
364 end process sequencing;
365 
366 
367 end Behavioral;
368 
369 ----------------------------------------------------------------------------
370 -- acc
371 library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;


entity ac is
    Port(clk: in std_logic;
    rst: in std_logic;
    en: in std_logic;
    --    c11: in std_logic;
    --    c3: in std_logic;
    c10: in std_logic;

    aluo:in std_logic_vector(15 downto 0);
    aco:out std_logic_vector(15 downto 0)
    );
end ac;

architecture Behavioral of ac is

begin

process(clk,rst,en)
variable reg:std_logic_vector(15 downto 0):=x"0000";
begin
    if rst='1' then
        reg:=x"0000";
    elsif rising_edge(clk) and en='1' then
        if c10='1' then reg:=aluo;
        end if;
    end if;
    aco<=reg;
end process;

end Behavioral;

```

7.2. Python compiler code

```
import re

with open("ram.txt", 'r') as f1:
    contents = f1.read()
contents = contents.lower()

ins_set = {"add": 1, "not": 2, "sub": 3, "lsr": 4, "mul": 5, "lsl": 6,
        "and": 7, "asr": 8,"or": 9, "asl": 10, "store": 11, "load": 12,
        "jmpgez": 13, "jmp": 14, "halt": 15}

size = 64
maccode = [0 for i in range(size)]
standard_code = ["" for j in range(size)]
lines = contents.splitlines()

```

```
linenum = 0  # line number in raw code
compiled_linenum = 0  # line number in standard code
storage_num = 0  # the number of used storage

jump_2 = {}  # jump-to L1:addr
jump_from = {}  # jump-from jmp L1,=> L1:addr(jmp)
store_addr = {}  # addr_var:addr
store_addr_start = size // 2

for i in lines:
    linenum += 1
    to_write = 0  # machine code to write
    spltcomm = i.split(';')
    if len(spltcomm) > 0:
        # Non comments: (;comments...)
        before_comm = spltcomm[0]
        before_comm_re = re.search(r"^\s*(\w+.*\w+)\s*$", before_comm)  # possible
        if before_comm_re is not None:
            before_comm = before_comm_re.groups()[0]  # NON-comment part
            ins = re.split(r"\s*:\s*", before_comm)  # list after split by " : "
            op_opr = ""
            if len(ins) == 2:  # incl. xxx:xxx

                jump_2[ins[0]] = compiled_linenum  # {"l1":10} 10 is ln in standard
                op_opr = ins[1]
            elif len(ins) == 1:
                op_opr = ins[0]

            if len(op_opr) > 0:  # PURE INSTRUCTION ,e.g. load a1, slr, ...
                op_opr_lst = re.split(r"\s+", op_opr)
                if len(op_opr_lst) == 1 and op_opr_lst[0] in ins_set:  # single operand
                    to_write = 256 * ins_set[op_opr_lst[0]]
                    standard_code[compiled_linenum] = op_opr_lst[0]

                elif len(op_opr_lst) == 2 and op_opr_lst[0] in ins_set:  # double
                    to_write = 256 * ins_set[op_opr_lst[0]]
                    standard_code[compiled_linenum] = op_opr_lst[0] + " "

                    operand = op_opr_lst[1]  # divide to opcode+operand
                    opcode = op_opr_lst[0]
                    if opcode not in ("jmp", "jmpgez"):
                        if opcode in ("store",):  # OPERAND of store must be an addr
                            if operand in store_addr:
                                store_addr[operand].append(compiled_linenum)
                            else:
                                store_addr[operand] = [compiled_linenum]  #

                        else:  # add, sub, mul, and, or, load
                            if re.fullmatch(r"[a-zA-Z]+\w*", operand):  # when is an
                                if operand in store_addr:
                                    store_addr[operand].append(compiled_linenum)
                                else:
                                    store_addr[operand] = [compiled_linenum]

```

```
                            else:  # when is an immediate num, validity check required
                                if eval(operand) in store_addr:
                                    store_addr[eval(operand)].append(compiled_linenum)
                                else:
                                    store_addr[eval(operand)] = [compiled_linenum]

                    else:
                        jump_from[operand] = compiled_linenum

                maccode[compiled_linenum] = to_write
                compiled_linenum += 1

for i in jump_from:
    jmpfrom = jump_from[i]
    standard_code[jmpfrom] += str(jump_2[i])
    maccode[jmpfrom] += jump_2[i]

for i in store_addr:
    if isinstance(i, int):
        standard_code[storage_num + store_addr_start] = str(i)
        maccode[storage_num + store_addr_start] = i
    for j in store_addr[i]:
        standard_code[j] += str(storage_num + store_addr_start)
        maccode[j] += storage_num + store_addr_start
    storage_num += 1

j = 0
for i in standard_code:
    print(j, '\t', i)
    j += 1

print()
j = 0
for i in maccode:
    print("%02x"%j,'\t',"%04x"%i)  #
    j += 1

lines = maccode

with open("ramdata.coe", 'w') as f2:
    f2.write("MEMORY_INITIALIZATION_RADIX=10;\n")
    f2.write("MEMORY_INITIALIZATION_VECTOR=\n")
    for i in lines:
        f2.write(str(i) + ',\n')
    f2.write('0;')

with open("ramdata.txt", 'w') as f2:
    f2.write("MEMORY_INITIALIZATION_RADIX=10;\n")
    f2.write("MEMORY_INITIALIZATION_VECTOR=\n")
    for i in lines:
        f2.write(str(i) + ',\n')
    f2.write('0;')

```