Computer Organization and Architecture Course Design

Southeast University
School of Information Science and Technology

Author: WU YX, ZHANG YX

Student Number: 040193xx,040193xx

Project Name: Microprogrammed CPU Design

Date: May 25, 2022

Contents

1.Purpose

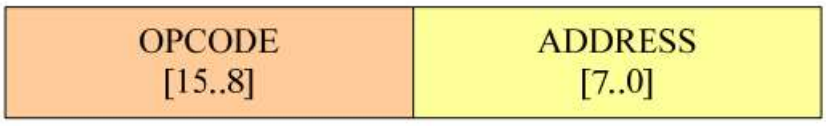
The purpose of this project is to design and verify a simple CPU (Central Processing Unit). This CPU has basic instruction set, and we will utilize its instruction set to generate a very simple program to verify its performance. For simplicity, we will only consider the relationship among the CPU, registers, 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. Tasks

The tasks of this simulation can be divided into 3 subplots from our perspective. That is:

1) instruction set design

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.

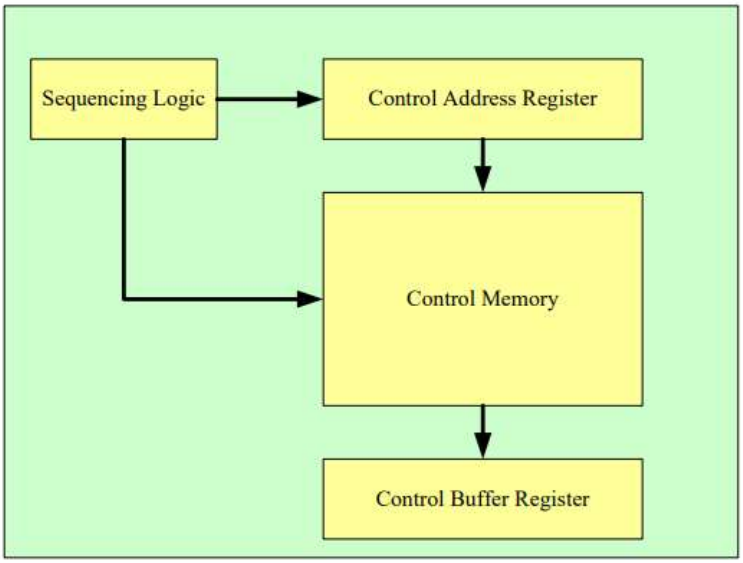


Instructions will be written into RAM, when CPU reads instructions from RAM, it then translates the instructions into operations through ALU and

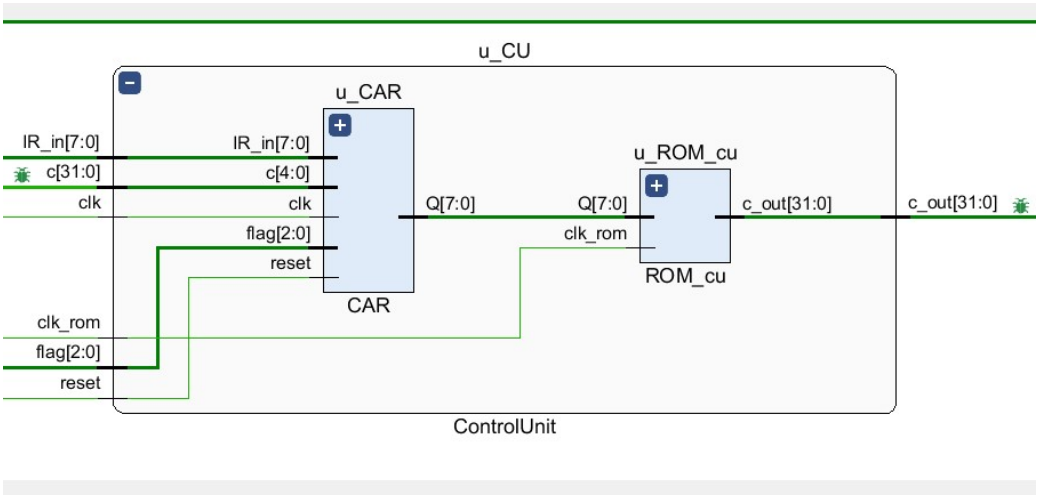
execute them.

2) Microinstruction design

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.



In our design, control unit is realized based on two modules named ROM and CAR.



CAR module is used to read instructions and then emit signals to ROM to locate the address of corresponding micro-instructions.

ROM contains all the pre-designed 32bit microinstructions, we will explain it further in section Microinstruction Design.

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.

3) Module design

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.

RAM
RAM with separate input and output ports, it works as memory which stores the instructions and data, 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.

3. Design Analysis

3.1 Overall Design

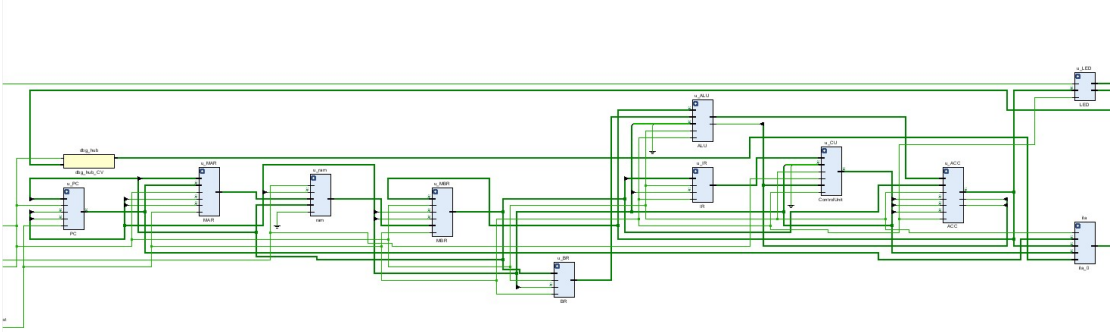
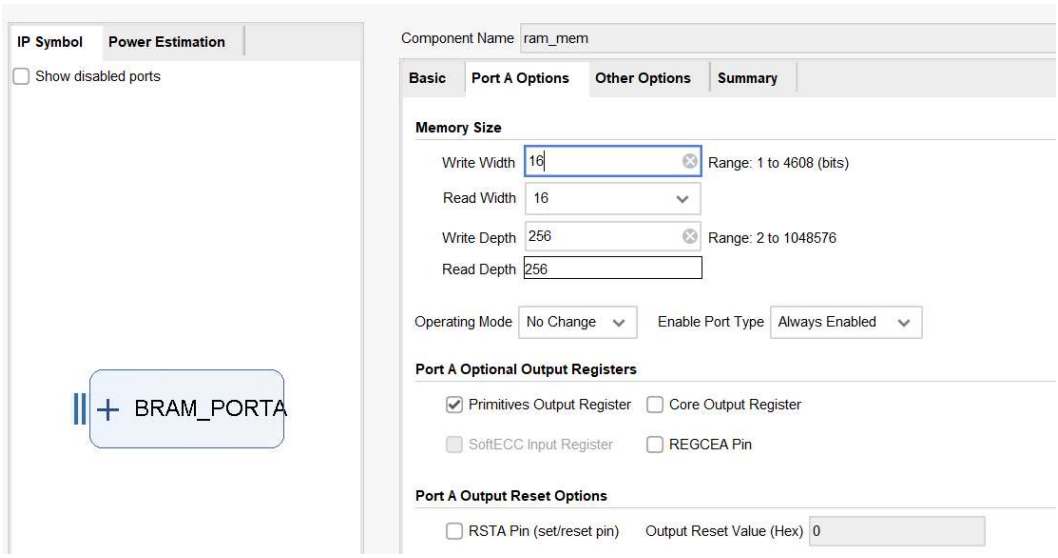


Figure Top Module schematic

In top module, we combine all the sub-modules together. Then it interacts with an outside RAM and LED digits to visualize the outcome of our design.

These two modules are separately realized here:
RAM: To build RAM, we utilize block memory IP core.
We use one port ram block with its size of 256*16 as follows:



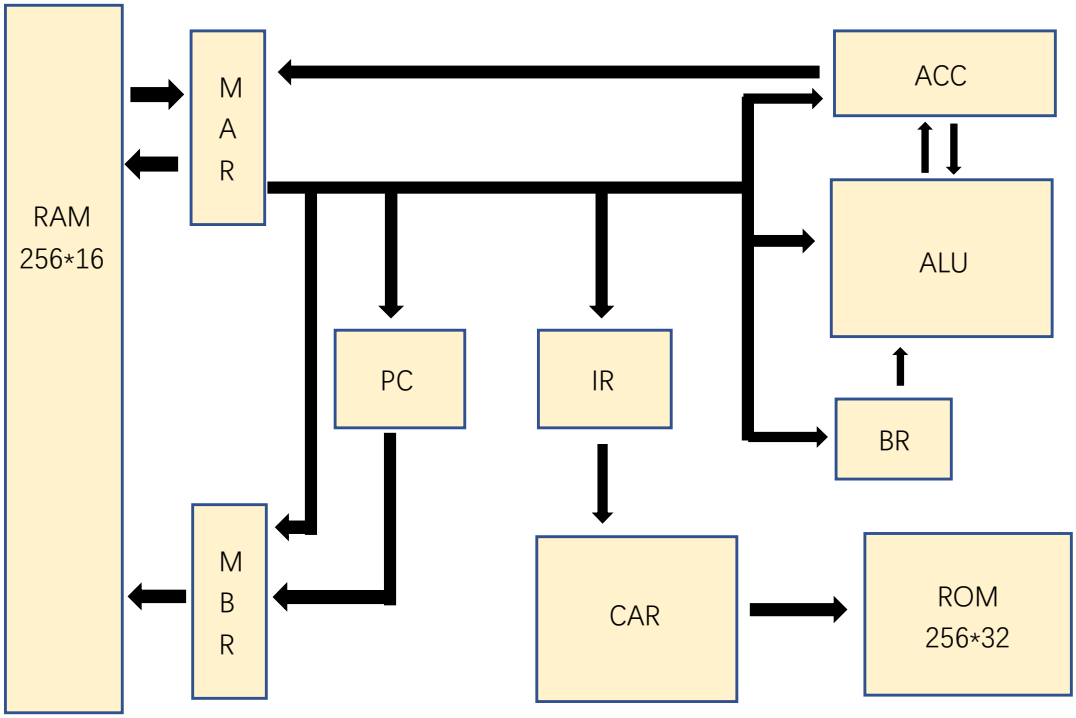

```
module ram(
    input clk,
    input reset,
    input [15:0] mbr_in,
    input [7:0] mar_in,
    input c12,
    output [15:0]ram_out
    );
```

mbr_in: receives content from MBR and write into RAM.
mar_in: point out the address where we need to access in RAM.
C12: the 12th bit of Control signal, if C12==1 it enables write operation to ram.
ram_out: enable cpu read content from ram.

There is one thing requiring our attention that is we should double the clock time for memory IP core for the read and write operation in IP core costs two time slots, which suggests that we need a doubled clock frequency.

```
always @(posedge clk_mem) begin
    count2=count2+1;
    if(count2>1)begin
        count2=0;
        clk_div=~clk_div;
    end
end
```

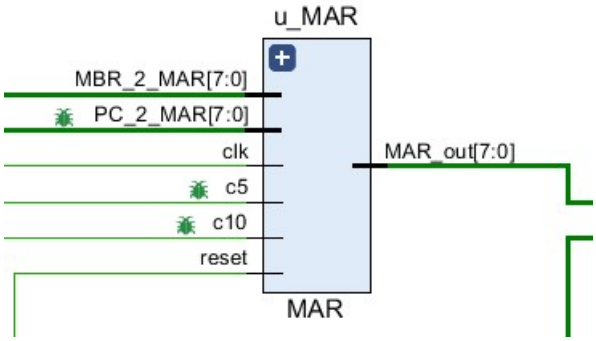
Additionally, we draw the rough schematic as our guidance for later design.



Finally, a 3-bit flag system is added to the design.
Flag bit meaning: **bit0** 1 for negative, 0 for positive
Bit1 1 for multiply finished, 0 not finished
Bit2 1 for operation finished, 0 not finished

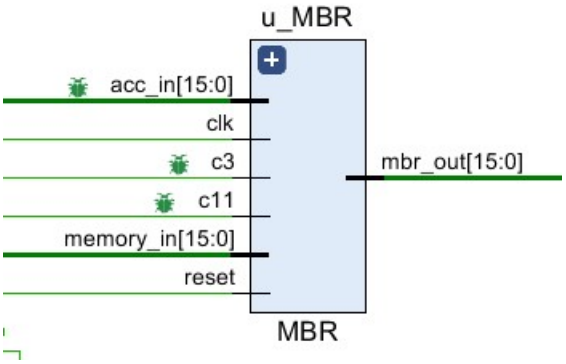
3.2 Module Design

1) MAR



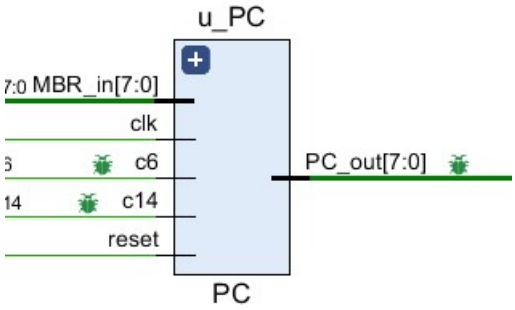
MAR gets information from MBR or PC, control bit c5 and c10 controls whether output content is from PC or from MBR.

2) MBR



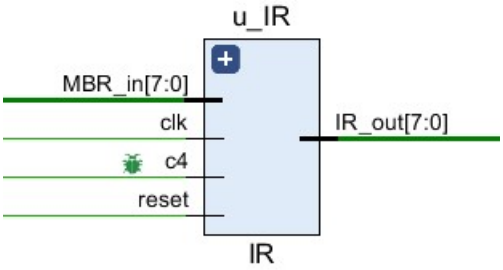
MBR gets information from ACC or Memory, control bit c3 and c11 controls whether output content is from ACC or from Memory.

3) PC



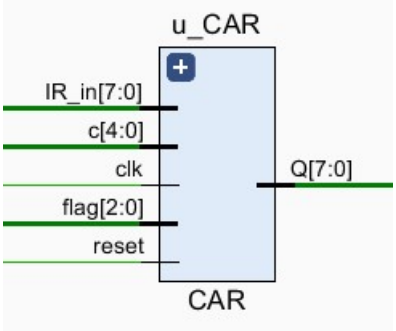
PC output points out the address of Memory which is to be directed to (read from or write into).

4) IR



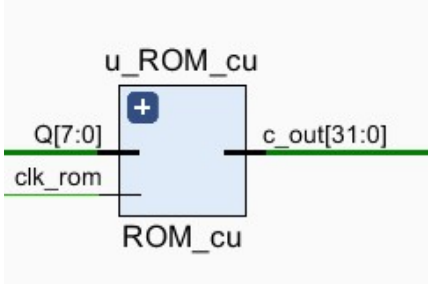
IR transmits operation code to CAR.

5) CAR



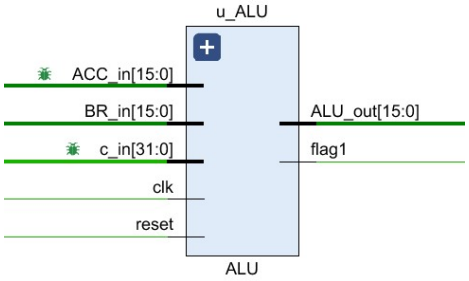
CAR translates the opcode and decides which micro-instruction is to be used in ROM.

6) ROM



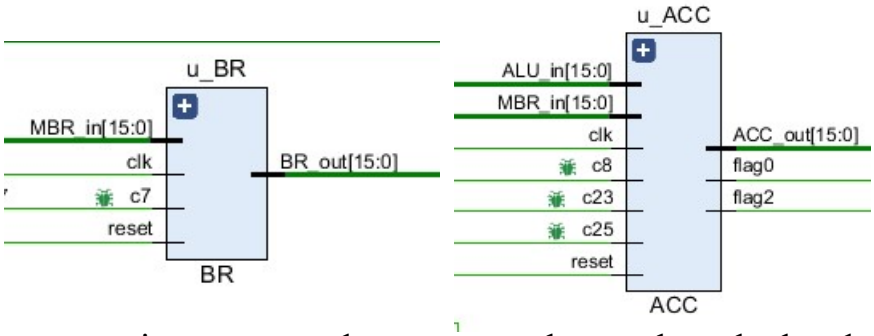
ROM receives address from CAR and issues corresponding control signal.

7) ALU



ALU executes operations carried by control signals.

8) BR and ACC



These two registers are used to store numbers to be calculated.

3.3 OperationCode Design

bit C	uOP	meaning
c0	CAR<=CAR+1	Control Address increment
c1	CAR<=****	address redirection, depends on position of uOP
c2	CAR<=0	reset car
c3	MBR<=memory	Read memory content to MBR
c4	IR<=[15:8]	copy MBR opcode
c5	MAR<=MBR[7:0]	copy MBR address
c6	PC<=PC+1	PC increment
c7	BR<=MBR	copy MBR to BR
c8	ACC<=0	reset ACC
c9	ALU<=ACC+BR	ADD BR to ACC
c10	MAR<=PC	read PC to MAR for next address
c11	MBR<=ACC	copy ACC content to MBR
c12	memory<=MBR	write MBR content to memory
c13	ALU<=ACC-BR	sub BR from ACC
c14	pc<=MBR[7:0]	copy MBR content address to PC
c15	ALU<=ACC and BR	ACC && BR
c16	ALU<=ACC or BR	ACC || BR
c17	ALU<= not ACC	!ACC
c18	ALU<=SHR ACC	logical right shift
c19	ALU<=SHL ACC	logical left shift
c20	CAR<=CAR+1+FLAG(1)	JMPGEZ
c21	ALU<=BR	input content of BR to ALU
c22	ALU<=ACC	input content of ACC to ALU
c23	ACC<=ALU	output result to ACC
c24	ALU<=RESULT[15:0] MR<=RESULT[31:16]	ALU gets the lower part of the mult result while MR gets the higher
c25	ACC<=MBR	input content of MBR to ACC
c26	ALU<=BR*ACC	multiply BR by ACC
c27	CAR<=CAR+FLAG(3)	multiply control
c28	*****	
c29	*****	
c30	*****	
c31	*****	

All the micro operations needed is stored in this table, we check this table to write instructions.

There is another table about higher class operations, including ADD, SUB, LOAD, etc. This table is listed here as well.

```
8'b00000001: add_pointer<=8'h10;//store
8'b00000010: add_pointer<=8'h08;//load
8'b00000011: add_pointer<=8'h18;//add
8'b00000100: add_pointer<=8'h20;//sub
8'b00000101: add_pointer<=8'h60;//jmpgez
8'b00000110: add_pointer<=8'h58;//jmp
8'b00000111: add_pointer<=8'h68;//halt
8'b00001000: add_pointer<=8'h50;//mpy
8'b00001010: add_pointer<=8'h28;//and
8'b00001011: add_pointer<=8'h30;//or
8'b00001100: add_pointer<=8'h38;//not
8'b00001101: add_pointer<=8'h48;//shl
8'b00001110: add_pointer<=8'h40;//shr
```

IR_in refers to the content of Opcode, which is stored in [15:8] parts of blocks in memory. We use this chart to position the relative address of each operation in pre-designed micro-instruction ROM, which is to be introduced later.

3.4 MicroInstruction Design

Instruction	CAR address	micro Instruction	Control signals	32bit Code
FETCH	00	CAR<=CAR+1 MAR<=PC	C0,C10	00000401
	01	CAR<=CAR+1 MBR<=MEMORY	C0,C3	00000009
	02	CAR<=CAR+1 IR<=MBR[15:7]	C0,C4	00000011
	03	CAR<=***	C1	00000002
LOAD	08	CAR<=CAR+1 PC<=PC+1 MAR<=MBR[7:0]	C0,C6,C5	00000061
	09	CAR<=CAR+1 MBR<=MEMORY	C0,C3	00000009
	0A	CAR<=0 ACC<=MBR	C2,C25	02000004
STORE	10	CAR<=CAR+1 PC<=PC+1 MAR<=MBR[7:0]	C0,C6,C5	00000061
	11	CAR<=CAR+1 MBR<=ACC	C0,C11	00000801

	12	CAR<=0 MEM<=MBR	C2,C12	00001004
ADD	18	CAR<=CAR+1 PC<=PC+1 MAR<=MBR[7:0]	C0,C6,C5	00000061
	19	CAR<=CAR+1 MBR<=MEMORY	C0,C3	00000009
	1A	CAR<=CAR+1 BR<=MBR	C0,C7	00000081
	1B	CAR<=CAR+1 ALU<=BR ALU<=ACC	C0,C21,C22	00600001
	1C	CAR<=CAR+1 ALU<=ACC+BR	C0,C9	00000201
	1D	CAR<=0 ACC<=ALU	C2,C23	00800004
SUB	20	CAR<=CAR+1 PC<=PC+1 MAR<=MBR[7:0]	C0,C6,C5	00000061
	21	CAR<=CAR+1 MBR<=MEMORY	C0,C3	00000009
	22	CAR<=CAR+1 BR<=MBR	C0,C7	00000081
	23	CAR<=CAR+1 ALU<=BR ALU<=ACC	C0,C21,C22	00600001
	24	CAR<=CAR+1 ALU<=ACC-BR	C0,C13	00002001
	25	CAR<=0 ACC<=ALU	C2,C23	00800004
AND	28	CAR<=CAR+1 PC<=PC+1 MAR<=MBR[7:0]	C0,C6,C5	00000061
	29	CAR<=CAR+1 MBR<=MEMORY	C0,C3	00000009
	2A	CAR<=CAR+1 BR<=MBR	C0,C7	00000081
	2B	CAR<=CAR+1 ALU<=BR ALU<=ACC	C0,C21,C22	00600001
	2C	CAR<=CAR+1 ALU<=ACC&&BR	C0,C15	00008001
	2D	CAR<=0 ACC<=ALU	C2,C23	00800004

OR	30	CAR<=CAR+1 PC<=PC+1 MAR<=MBR[7:0]	C0,C6,C5	00000061
	31	CAR<=CAR+1 MBR<=MEMORY	C0,C3	00000009
	32	CAR<=CAR+1 BR<=MBR	C0,C7	00000081
	33	CAR<=CAR+1 ALU<=BR ALU<=ACC	C0,C21,C22	00600001
	34	CAR<=CAR+1 ALU<=ACC||BR	C0,C16	00010001
	35	CAR<=0 ACC<=ALU	C2,C23	00800004
NOT	38	CAR<=CAR+1 PC<=PC+1 MAR<=MBR[7:0]	C0,C6,C5	00000061
	39	CAR<=CAR+1 MBR<=MEMORY	C0,C3	00000009
	3A	CAR<=CAR+1 ACC<=MBR	C0,C25	02000001
	3B	CAR<=CAR+1 ALU<=ACC	C0,C22	00400001
	3C	CAR<=CAR+1 ALU<=not ACC	C17,C0	00040001
	3D	CAR<=0 ACC<=ALU	C2,C23	00800004
SHR	40	CAR<=CAR+1 PC<=PC+1 ALU<=ACC	C0,C6,C22	00400041
	41	CAR<=CAR+1 ALU<=SHR ACC	C0,C19	00080001
	42	CAR<=0 ACC<=ALU	C2,C23	00800004
SHL	48	CAR<=CAR+1 PC<=PC+1 ALU<=ACC	C0,C6,C22	00400041
	49	CAR<=CAR+1 ALU<=SHL ACC	C0,C18	00040001
	4A	CAR<=0 ACC<=ALU	C2,C23	00800004
MPY	50	CAR<=CAR+1 PC<=PC+1 MAR<=MBR[7:0]	C0,C6,C5	00000061

	51	CAR<=CAR+1 MBR<=MEMORY	C0,C3	00000009
	52	CAR<=CAR+1 BR<=MBR	C0,C7	00000081
	53	CAR<=CAR+1 ALU<=BR ALU<=ACC	C0,C21,C22	00600001
	54	CAR<=CAR+FLAG(3) RESULT<=BR*ACC	C26,C27	0C000000
	55	CAR<=CAR+1 ALU<=RESULT[15:0] MR<=RESULT[31:16]	C0,C24	01000001
	56	CAR<=0 ACC<=ALU	C2,C23	00800004
JMP	58	CAR<=0 PC<=MBR[7:0]	C2,C14	00004004
JMPGEZ	60	CAR<=CAR+1+FLAG(1)	C20	00100000
	61	CAR<=0 PC<=MBR[7:0]	C2,C18	00004004
	62	CAR<=0 PC<=PC+1	C2,C6	00000044
HALT	68	CAR<=0	C2	00000004

ALL the instructions and their corresponding micro operations is recorded.

```
; depth256, width32
; block memory

memory_initialization_radix = 16;
memory_initialization_vector =
00000401, 00000009, 00000011, 00000002, 00000002, 00000002, 00000002, 00000002,
00000061, 00000009, 02000004, 00000002, 00000002, 00000002, 00000002, 00000002,
00000061, 00000801, 00001004, 00000002, 00000002, 00000002, 00000002, 00000002,
00000061, 00000009, 00000081, 00600001, 00000201, 00800004, 00000002, 00000002,
00000061, 00000009, 00000081, 00600001, 00002001, 00800004, 00000002, 00000002,
00000061, 00000009, 00000081, 00600001, 00008001, 00800004, 00000002, 00000002,
00000061, 00000009, 00000081, 00600001, 00010001, 00800004, 00000002, 00000002,
00000061, 00000009, 02000001, 00400001, 00040001, 00800004, 00000002, 00000002,
00400041, 00080001, 00800004, 00000002, 00000002, 00000002, 00000002, 00000002,
00400041, 00040001, 00800004, 00000002, 00000002, 00000002, 00000002, 00000002,
00000061, 00000009, 00000081, 00600001, 0C000000, 01000001, 00800004, 00000002,
00004001, 00000002, 00000002, 00000002, 00000002, 00000002, 00000002, 00000002,
00100000, 00004004, 00000044, 00000002, 00000002, 00000002, 00000002, 00000002,
00000004, 00000002, 00000002, 00000002, 00000002, 00000002, 00000002, 00000002;
```

These instructions are actually recorded inside ROM, which is listed above. For example, if we use JMPGEZ, the address pointer is located to 60h, which have the content of 00100000 in the second last line in this

pic. Then the pointer goes to next instruction 00004004 with singal which indicates CAR<=CAR+1+FLAG(1). This FLAG(1) means bit0, which identifies if the number is positive or not.

3.5 Top Design

In top module, we mainly instantiate all the submodules and set wires. Besides, we added an LED module to visualize our calculation result by showing the content of ACC.

```
module cpu_top(
    input clk,
    input reset,
    output [7:0] SSEG_CA,
    output [7:0] SSEG_AN
    );
```

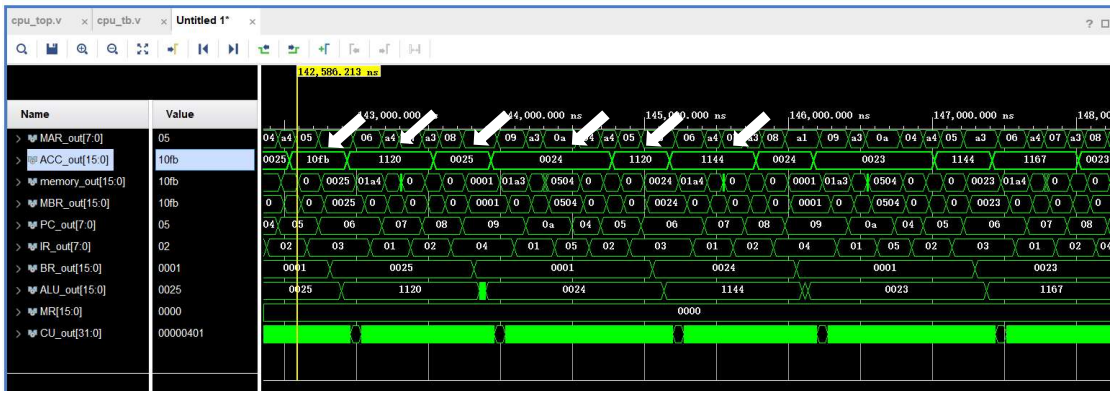
4. Simulation

4.1 Test Program Design

4.1.1 sum of 1+2+…+100

Program with C	Program with instructions	Contents of Memory (RAM) in HEX	
		Address	Contents
sum=0;	LOAD A0	00	02A0
	STORE A4	01	01A4
temp=100;	LOAD A2	02	02A2
	STORE A3	03	01A3
loop :sum=sum+temp;	LOOP:LOAD A4	04 (so LOOP=04)	02A4
	ADD A3	05	03A3
	STORE A4	06	01A4
temp=temp-1;	LOAD A3	07	02A3
	SUB A1	08	04A1
	STORE A3	09	01A3
if temp>=0 goto loop;	JMPGEZ LOOP	0A	0504
end	HALT	0B	HALT
		0C	
		…..	…..
		A0	0000
		A1	0001
		A2	0064
		A3	
		A4	
		A5	
		….	…..

Write contents into memory, then run our CPU to check the answer.

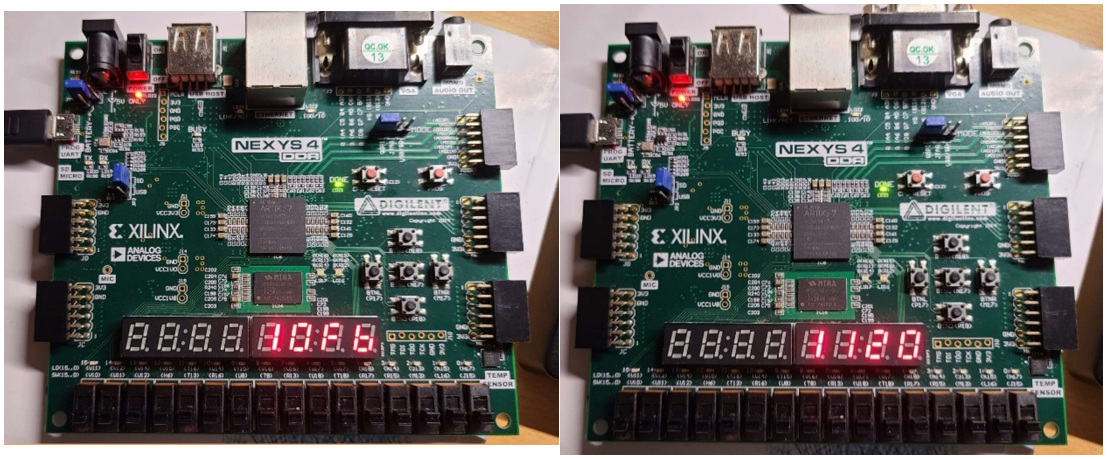


One period of the running result is shown here.

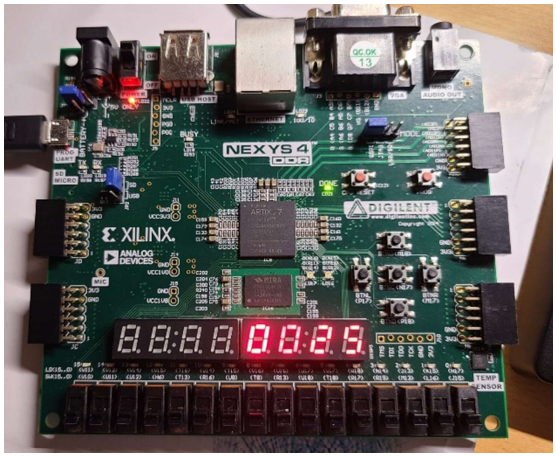
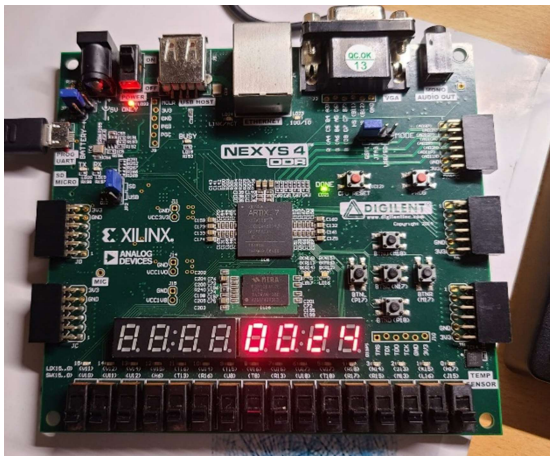
Firstly Let me describe the logic of this program. Actually it's quite simple, from 100 to 1, in every loop the counter adds itself to the result and then minus 1.

In this moment, for instance, we put our focus on ACC_out, now the sum result is 10fb(Hex), 10fb + 0025 = 1120, which is shown in next time slot. Then it's time for counter to minus 1, from 0025 to 0024, this process can be observed in sequence. Here comes the next loop, 1120 + 0024 = 1144.

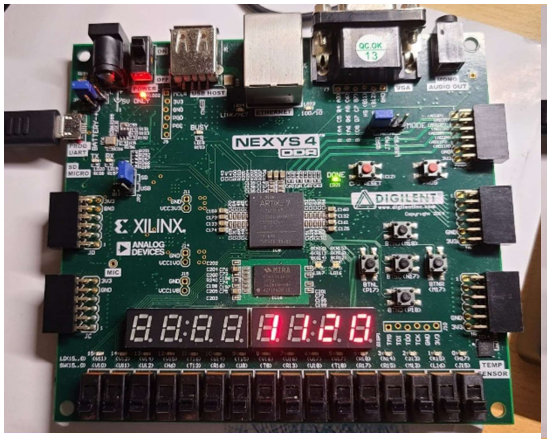
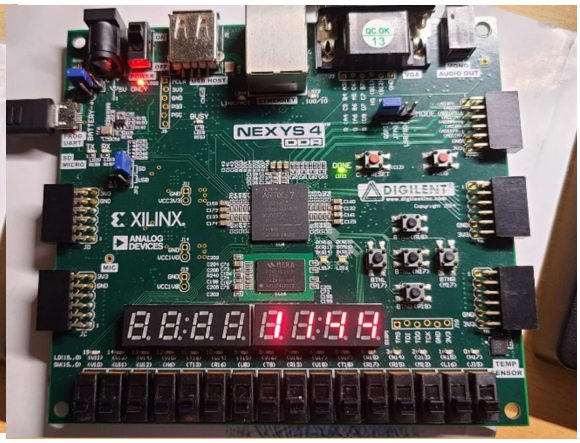
This process is also presented on board:



10FB ----------> 1120

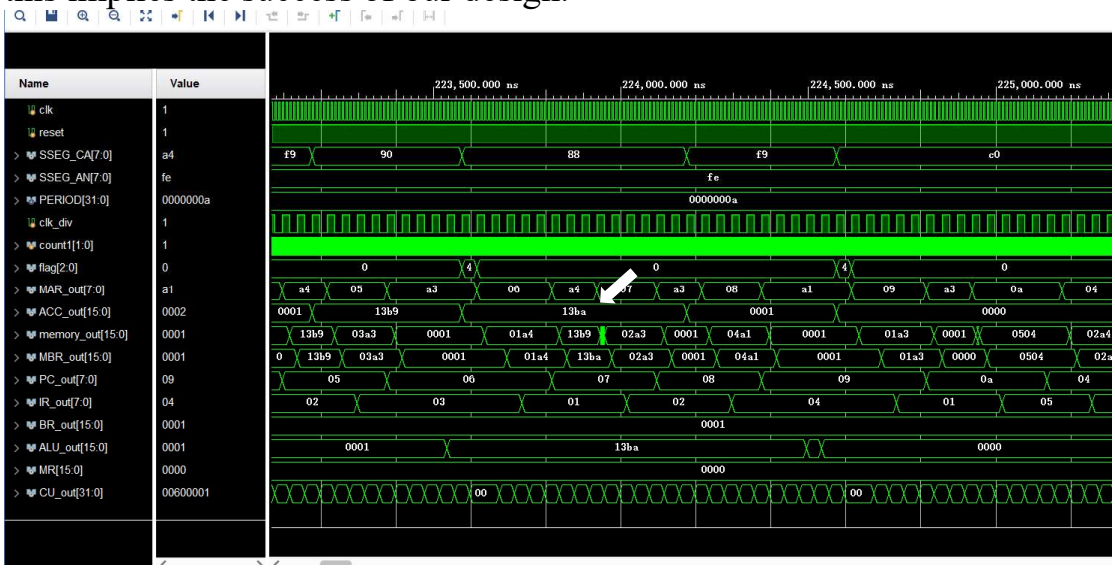



0025 ----------> 0024

1120 ----------> 1144

Finally, the result turn out to be 13ba, which represents 5050 in decimal, this implies the success of our design.



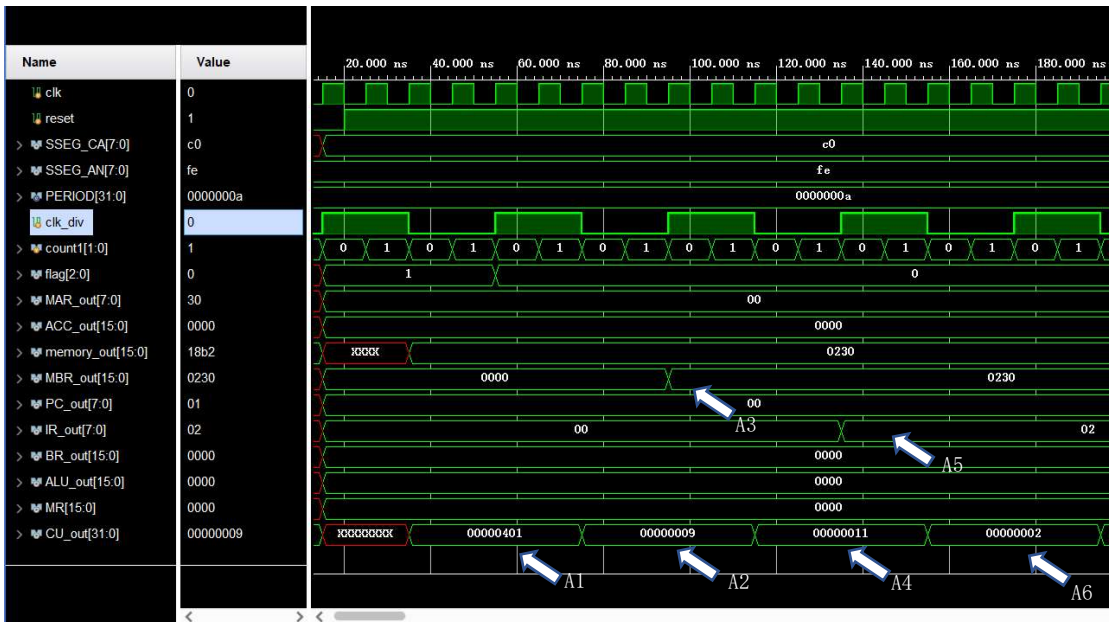
This test program proves the function of JMGEZ, LOAD, STORE, ADD, SUB, HALT and the correct function of LED display. JMP has the same logic as JMPGEZ, we don’t need to test one more time.

4.1.2 Shift Test

Our program is shown here:

Instruction	Address	Contents
LOAD 30	30	02
SHL	00	0D
STORE 40	40	01
LOAD 31	31	02
SHR	00	0E
AND 40	40	0A

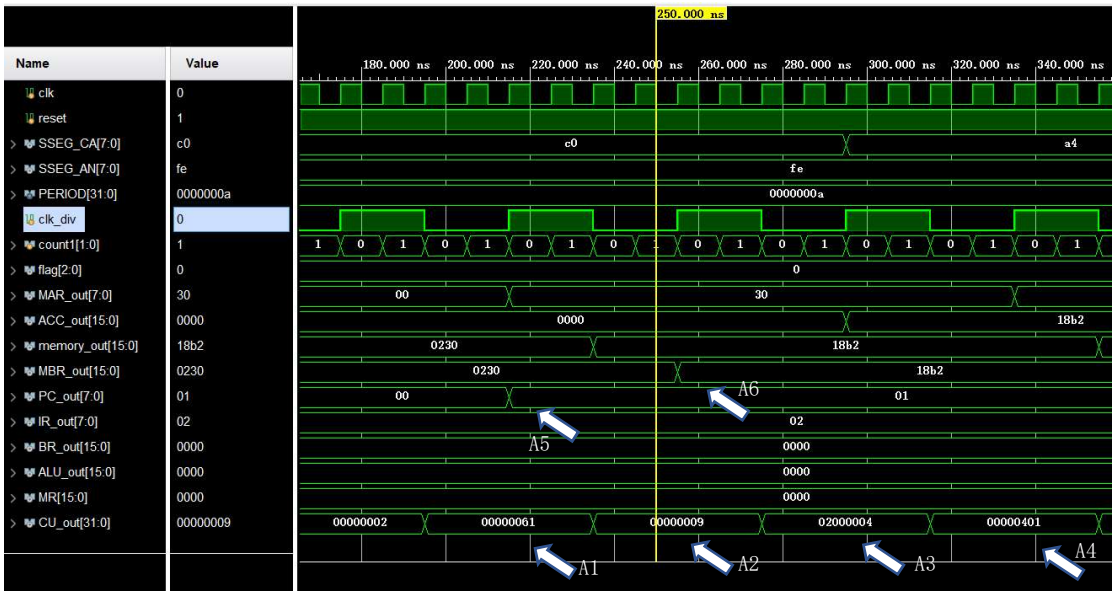
30H: 18B2 31H:FE9C



A1,A2,A4,A6 :Fetch instruction is executed,00000002 waits the next instruction.

A3: when 00000009 is executed, mbr<= memory_out, it becomes 0230.

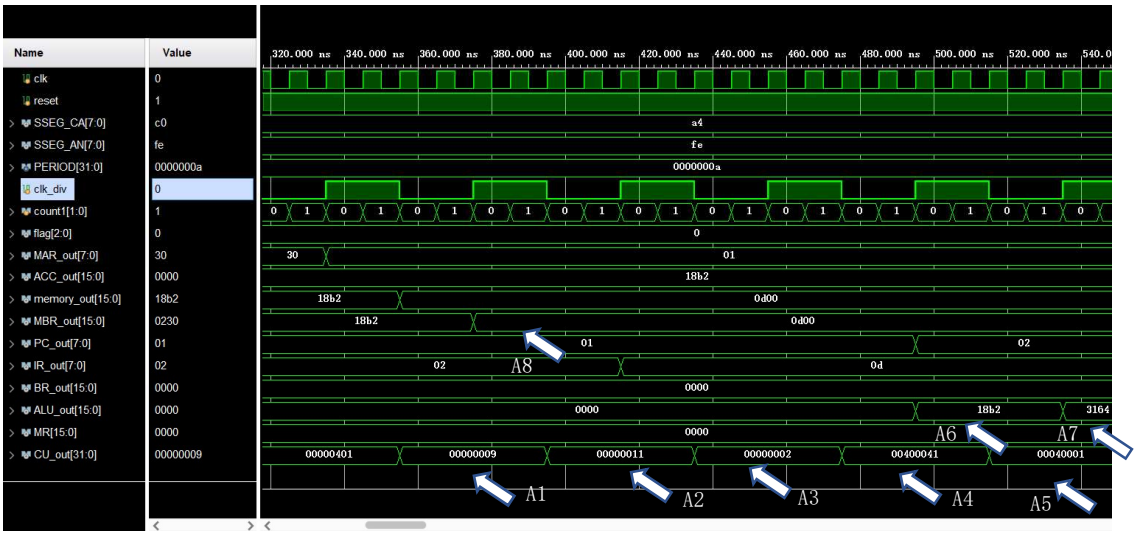
A5: when 00000011 is executed, IR gets mbr[15:7], becomes 02.



A1,A2,A4,A6 :Load instruction is executed, from A1 to A3 executes Load, then a Fetch is executed implicitly from A4.

A5: after 00000061, PC = PC+1, begin reading the next block in memory.

A6: after 00000009, mbr<=memory, which is form Mar pointed(30h) position. We have 18b2.



A1, A2, A3: implicit Fetch, gets the next instruction.

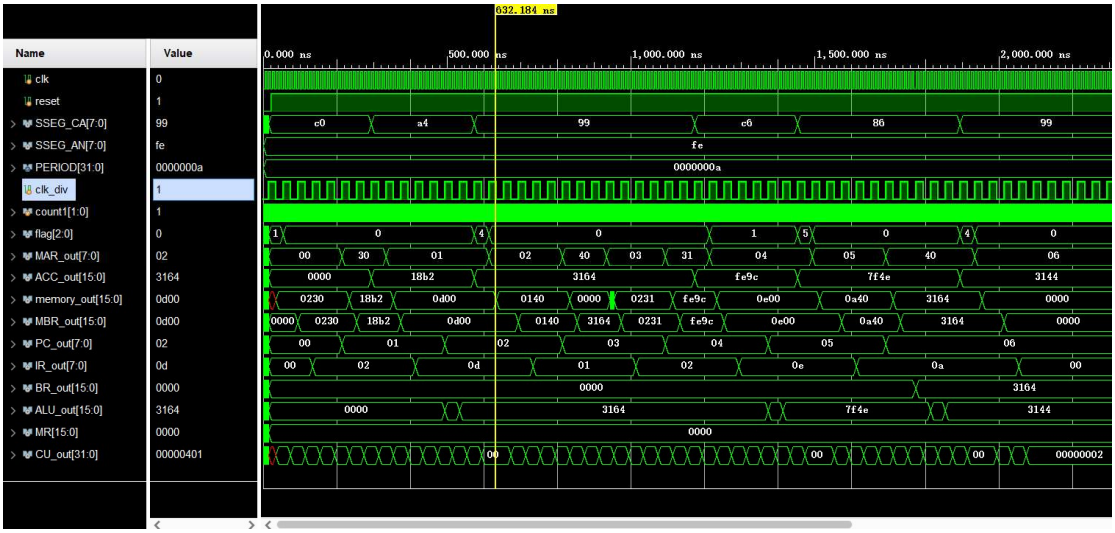
A4, A5: SHL, logical left shift.

A8: after 00000009, mbr get next instruction.

A6: after 00400041, ALU loads ACC content 18b2.

A7: after 00040001, ALU left shifts its content and get 3164.

Until this moment, basic micro-operation execution has been fully presented, let's have a full review of this program.



ACC: 0000->(load)18b2->(shl)3164->(load)fe9c->(shr)7f4e->(and)3144
This program testifies SHL, SHR, AND. Additionally, OR and NOT have the same logic as AND, we don't test them one more time.

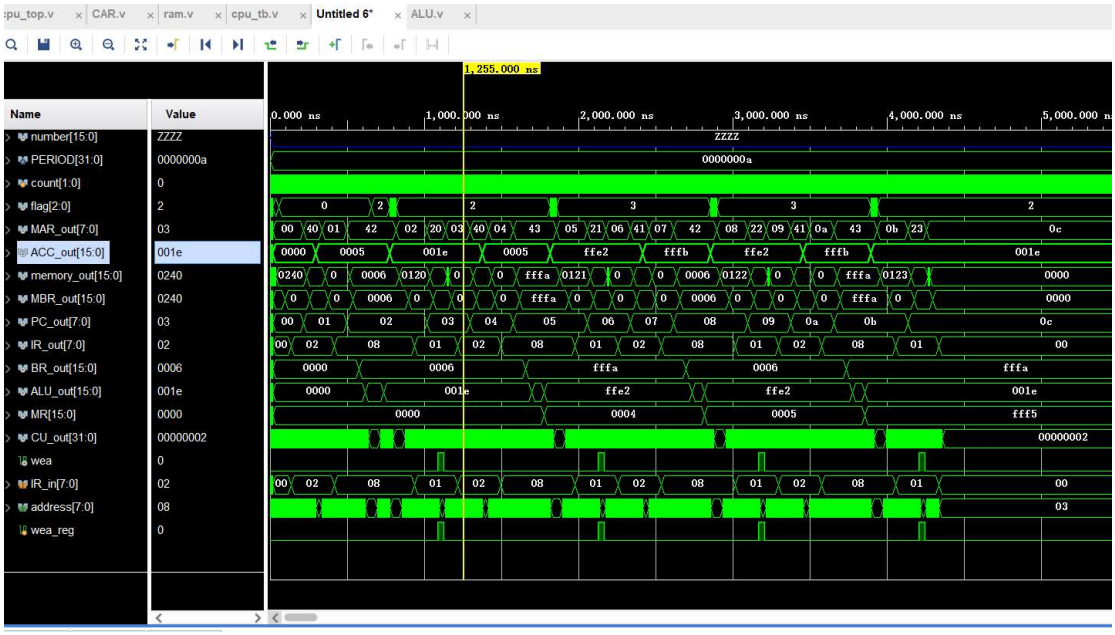
4.1.2 Multiply Test

program to multiply 6 by 5, -6 by 5 and -6 by -5 using MPY instruction

Instruction	Address	Contents
LOAD 40	40	02
MPY 42	42	08
STORE 20	20	01
LOAD 40	40	02
MPY 43	43	08
STROE 21	21	01
LOAD 41	41	02
MPY 42	42	08
STORE 22	22	01

LOAD 41	41	02
MPY 43	43	08
STORE 23	23	01

Here is the waveform:



We can see ACC_out contents are :

0005 001e 0005 ffe2 fffb ffe2 fffb 001e

(+5) (30) (+5) (-30) (-5) (-30) (-5) (30)

This sequence fits the condition correctly.

Besides, when computing, we can see the flag is changing based on results.

This program testifies MPY function.

5. Conclusion

1. From the this Design, We've reviewed the entire work flow of CPU, especially about its micro-program ideas. When designing the control signals, at first we were bemused about how these control bits can cooperate with other modules to complete different instructions, but after a long time of thinking, an idea suddenly hit me, and let me understand that all the operations are break into micro instructions that are controlled

by each gate, and all the gates are controlled by control signal. If we want certain instruction, we just combine the gate signals we need.

2. Pipeline structure in CPU is of vital significance. In CPU simulation design, we must arrange different modules and instructions with proper time sequence, or instructions may arrive too early to destroy previous instructions. To satisfy the timing requirement, we adopt flags to identify if one operation is over, thus CAR won't allow the next one to exceed. Additionally, when accessing RAM and ROM, as a result their accessing rules, we must double their clock so that their output can be updated on time.

3. In the whole design, we adopt hexadecimal 2s-complementary code. This coding method has several merits. First, this method avoids configuring if our content is minus or not, for the differences has been embedded into coding rules. Second, hexadecimal coding saves space for visualizing, thus enabling us to pay more attention on data structure and data flow.

6.Discussion

1. Compared to the vast modern computer design strategies, this CPU is just a toy level experiment. But still, we have learned a lot from all the procedure. This design can be improved if we further design a auto-translate program that could turn assembly language into machine languages. By adopting this program, the system will look more like a modern CPU.

2. The efficiency of our CPU can be improved by optimizing the pipeline. In our design, we doubled all the clocks only to serve RAM and ROM. However, we can insert time gaps for any procedure related to read and

write about memory blocks, and still keep other procedure a fast speed to run. This trick is pragmatic, but requires more skills to debug, so we didn't choose to adopt it.

3. Thanks for reading

6.Appendix

All of our source are zipped along with this document, readers can refer to our **xpr.project** to see the details of our design.

TOP

```
module cpu_top(
    input clk,
    input reset,
    output [7:0] SSEG_CA,
    output [7:0] SSEG_AN
    );

//wire clk_50M;
//wire clk_100M;
//wire locked;

//clk_divide u_clk
//   (
//    // Clock out ports
//    .clk_100M(clk_100M),        // output clk_100M
//    .clk_50M(clk_50M),        // output clk_50M
//    // Status and control signals
//    .locked(locked),            // output locked
//   // Clock in ports
//    .clk_in1(clk)
//    );          // input clk_in1
reg clk_div=0;
reg [1:0] count1=0;
//分频
always @(posedge clk) begin
    count1=count1+1;
    if(count1>1)begin
        count1=0;
        clk_div=~clk_div;
    end
end

// reg [31:0] c_bit=0;//操作信号
// reg [2:0] flag_reg=3'b0;//flag信号
wire [2:0]flag;
//assign flag=flag_reg;
//这里所有out并不区分端口位置
//每个module抽象化为多输入，单输出端口连接彼此

wire [7:0]MAR_out;
wire [15:0]ACC_out;
wire [15:0]memory_out;
wire [15:0]MBR_out;
wire [7:0]PC_out;
wire [7:0]IR_out;
wire [15:0]BR_out;
wire [15:0]ALU_out;
wire [15:0]MR;
wire [31:0]CU_out;
//sseg

//ram
ila_0 ila(
    .clk(clk_div), // input wire clk
    .probe0(PC_out), // input wire [7:0]
addr
    .probe1(ACC_out), // input wire [15:0]
acc
    .probe2(CU_out) // input wire [31:0]
Cbit
);

ram u_ram(
    .clk(clk),
    .reset(reset),
    .mbr_in(MBR_out),
    .mar_in(MAR_out),
    .c12(CU_out[12]),
    .ram_out(memory_out)
    );

MAR u_MAR(
    .clk(clk_div),
    .c5(CU_out[5]),//控制信号
    .c10(CU_out[10]),
    .reset(reset),
    .PC_2_MAR(PC_out),//PC导入MAR
```

```
    .MBR_2_MAR(MBR_out[7:0]),//MBR输出地址
    .MAR_out(MAR_out)
);

MBR u_MBR(
.acc_in(ACC_out),
.memory_in(memory_out),
.reset(reset),
.c3(CU_out[3]),
.c11(CU_out[11]),
.clk(clk_div),
.mbr_out(MBR_out)
);

PC u_PC(
.clk(clk_div),
.reset(reset),
.c6(CU_out[6]),
.c14(CU_out[14]),
.MBR_in(MBR_out[7:0]),
.PC_out(PC_out)
    );

IR u_IR(
.clk(clk_div),
.reset(reset),
.c4(CU_out[4]),
.MBR_in(MBR_out[15:8]),
.IR_out(IR_out)
    );

//ACC
ACC u_ACC(
.clk(clk_div),
.c8(CU_out[8]),
.c23(CU_out[23]),
.c25(CU_out[25]),
.reset(reset),
.ALU_in(ALU_out),
.MBR_in(MBR_out),
.ACC_out(ACC_out),
.flag0(flag[0]),
.flag2(flag[2])
);

ControlUnit u_CU(
    .clk(clk_div),
    .clk_rom(clk),
    .reset(reset),
    .IR_in(IR_out),
    .c(CU_out),
    .flag(flag),//flag0:正负1乘法完成2操作完成
    .c_out(CU_out)
);

BR u_BR(
    .clk(clk_div),
    .reset(reset),
    .c7(CU_out[7]),
    .MBR_in(MBR_out),
    .BR_out(BR_out)
);

ALU u_ALU(
    .clk(clk_div),
    .reset(reset),
    .c_in(CU_out),
    .ACC_in(ACC_out),
    .BR_in(BR_out),
    .flag1(flag[1]),
    .MR(MR),
    .ALU_out(ALU_out)
);

LED u_LED(
    .clk(clk),
    .rst_n(reset),
    .data(ACC_out),
    .SSEG_AN(SSEG_AN),//位选
    .SSEG_CA(SSEG_CA)//段选
);

endmodule
```

MAR

```
module MAR (
    input clk,
    input c5,//控制信号
    input c10,
    input reset,
    input [7:0]PC_2_MAR,//PC导入MAR
    input [7:0]MBR_2_MAR,//PC导入MBR
    output [7:0]MAR_out
);
reg [7:0]MAR_out_reg;
always @(posedge clk )
begin
    if(!reset)
        MAR_out_reg<=8'b00000000;
    else if(c10==1)
        MAR_out_reg<=PC_2_MAR;
    else if(c5==1)
        MAR_out_reg<=MBR_2_MAR;
end
assign MAR_out=MAR_out_reg;
```

```
endmodule
```

MBR

```
module MBR(
input [15:0] acc_in,memory_in,
input c3,
input c11,
input clk,
input reset,
output [15:0] mbr_out
);
reg [15:0]mbr_out_reg;
always @(posedge clk)
      if(!reset)
            mbr_out_reg<=0;
      else if(clk==1) begin
         if(c3==1) begin
             mbr_out_reg<=memory_in;
         end
         else if (c11==1) begin
            mbr_out_reg<=acc_in;
          end
      end
assign mbr_out=mbr_out_reg;
endmodule
```

PC

```
module PC(
input clk,reset,c6,c14,
input [7:0] MBR_in,
output [7:0] PC_out
      );
reg [7:0] PC_out1;
always @(posedge clk) begin
      if(reset==0) begin
      PC_out1<=8'b0;
      end
      else if(c6==1) begin
      PC_out1<=PC_out1+1;
      end
      else if(c14==1) begin
      PC_out1<=MBR_in;
      end
end
assign PC_out=PC_out1;
endmodule
```

IR

```
module IR(
input clk,reset,c4,
input [7:0] MBR_in,
output [7:0] IR_out
      );
reg [7:0] IR_out_reg;
always @(posedge clk)
begin
      if(reset==0)
      begin
            IR_out_reg<=8'h00;
      end
      else if(c4==1)
      begin
            IR_out_reg<=MBR_in;
      end
end
assign IR_out=IR_out_reg;
endmodule
```

ACC

```
module ACC(
input clk,c8,c23,c25,reset,
input [15:0] ALU_in,MBR_in,
output [15:0] ACC_out,
output flag0,flag2
      );
      reg [15:0] ACC_out1;
      reg flag2_reg;
      reg flag0_reg;
always @(posedge clk) begin
      if (!reset) begin
         ACC_out1<=16'h0;
         flag2_reg<=0;
```

```
        flag0_reg<=0;
    end
    else if(c8==1)begin
        ACC_out1<=16'h0;
    end
    if (c23==1) begin
    ACC_out1<=ALU_in;
    flag2_reg<=1;
    end
    else begin
    flag2_reg<=0;
    end
    if (c25==1)begin
    ACC_out1<=MBR_in;
    end
    if (ACC_out1[15]==0) begin
    flag0_reg<=0;
    end
    else begin
    flag0_reg<=1;
    end
end
assign flag2=flag2_reg;
assign flag0=flag0_reg;
assign ACC_out=ACC_out1;
endmodule
```

ControlUnit

```
module ControlUnit (
    input clk,
    input clk_rom,
    input reset,
    input [7:0]IR_in,
    input [31:0]c,
    input [2:0]flag,//flag0:正负1乘法完成2
操作完成
    output [31:0]c_out
);
wire [7:0]address;

CAR u_CAR(
    .clk(clk),
    .reset(reset),
    .IR_in(IR_in),
    .c(c),
    .flag(flag),//flag0:正负1乘法完成2操作
完成
    .address(address)
);

ROM_cu u_ROM_cu(
    .address(address),
    .clk(clk_rom),
    .c_out(c_out)
    );
Endmodule
```

CAR

```
module CAR (
    input clk,
    input reset,
    input [7:0]IR_in,
    input [31:0]c,
    input [2:0]flag,//flag0:正负1乘法完成2
操作完成
    output [7:0]address
);
//reg [7:0]opcode;
reg [7:0]add_pointer;

always @(posedge clk )
begin
    //opcode<=IR_in;
    if(c[0]==1)//步增
        add_pointer<=add_pointer+1;
    if(c[2]==1)//指针清零
        add_pointer<=0;
    if(flag[2])//当前操作结束
        add_pointer<=0;
    if(c[27]==1)//flag=1乘法结束继续
        add_pointer<=add_pointer+flag[1];
    if(c[20]==1)//flag0=1对应结果<0循环结
束
```

```
add_pointer<=add_pointer+1+flag[0];
      if (c[1]==1&&IR_in!=0)
      begin
            case (IR_in)
                  8'b00000001:
add_pointer<=8'h10;//store
                  8'b00000010:
add_pointer<=8'h08;//load
                  8'b00000011:
add_pointer<=8'h18;//add
                  8'b00000100:
add_pointer<=8'h20;//sub
                  8'b00000101:
add_pointer<=8'h60;//jmpgez
                  8'b00000110:
add_pointer<=8'h58;//jmp
                  8'b00000111:
add_pointer<=8'h68;//halt
                  8'b00001000:
add_pointer<=8'h50;//mpy
                  8'b00001010:
add_pointer<=8'h28;//and
                  8'b00001011:
add_pointer<=8'h30;//or
                  8'b00001100:
add_pointer<=8'h38;//not
                  8'b00001101:
add_pointer<=8'h48;//shl
                  8'b00001110:
add_pointer<=8'h40;//shr

                  default:
add_pointer<=8'h00;//default
            endcase
      end
      if(!reset)
            add_pointer<=0;
end
assign address=add_pointer;

endmodule
```

ROM_cu

```
module ROM_cu(
      input [7:0]address,
      input clk,
      output [31:0]c_out
      );
rom_cm u_rom(
   .clka(clk),       // input wire clka
   .ena(1),          // input wire ena
   .addra(address),  // input wire [7 : 0] addra
   .douta(c_out)     // output wire [31 : 0] douta
);

endmodule
```

BR

```
module BR (
      input clk,
      input reset,
      input c7,
      input [15:0] MBR_in,
      output [15:0] BR_out
);
reg [15:0] BR_out1;
always@(posedge clk)
begin
      if (!reset)
            BR_out1 <= 16'h0000;
      else if(c7==1)
            BR_out1 <= MBR_in;
end
assign BR_out = BR_out1;
endmodule
```

ALU

```
module ALU (
      input clk,
      input reset,
      input [31:0]c_in,
      input [15:0]ACC_in,
      input [15:0]BR_in,
      output flag1,
```

```
    output [15:0]MR,
    output [15:0]ALU_out
);
reg [15:0]ALU_out_reg;

reg [31:0]mult;
//reg [15:0]alu;
reg flag1_reg;
reg [15:0]MR_reg;

always@(posedge clk)
begin
    //alu<=ALU_out_reg;
    if(!reset)
    begin
        ALU_out_reg<=16'h0000;
        MR_reg<=0;
        flag1_reg<=0;
    end
    else if(c_in[21]==1)
        ALU_out_reg<=BR_in;
    else if(c_in[22]==1)
        ALU_out_reg<=ACC_in;
    else if(c_in[9]==1)//加法
        ALU_out_reg<=ACC_in+BR_in;
    else if(c_in[13]==1)//减法
        ALU_out_reg<=ACC_in-BR_in;
    else if(c_in[15]==1)//和
        ALU_out_reg<=ACC_in&BR_in;
    else if(c_in[16]==1)//或
        ALU_out_reg<=ACC_in|BR_in;
    else if(c_in[17]==1)//取反
        ALU_out_reg<=~BR_in;
    else if(c_in[19]==1)//逻辑右移

ALU_out_reg<={1'b0,ALU_out_reg[15:1]};
    else if(c_in[18]==1)//逻辑左移

ALU_out_reg<={ALU_out_reg[14:0],1'b0};
    else if(c_in[24]==1)
    begin
        ALU_out_reg<=mult[15:0];
        MR_reg<=mult[31:16];
    end
    else if(c_in[26]==1)//乘法
    begin
        mult<=ACC_in*BR_in;
        flag1_reg<=1;//乘法完成
    end
end
assign flag1=flag1_reg;
assign ALU_out=ALU_out_reg;
assign MR=MR_reg;
endmodule
```

RAM

```
module ram(
    input clk,
    input reset,
    input [15:0] mbr_in,
    input [7:0] mar_in,
    input c12,
    output [15:0]ram_out
    );
reg wea_reg=0;
ram_mem u_ram (
  .clka(clk),       // input wire clka
  .wea(wea),          // input wire [0 : 0] wea
  .addra(mar_in),   // input wire [7 : 0] addra
  .dina(mbr_in),      // input wire [15 : 0]
dina
  .douta(ram_out)   // output wire [15 : 0]
douta
);
always@(posedge clk)
begin
    if(c12==1)//控制mbr写入mem不是无限
制就可以
        wea_reg<=1;//读取功能在mbr管理
    else
        wea_reg<=0;
end
assign wea=wea_reg;
endmodule
```