Computer Organization and Architecture

Course Design

The Report of Microinstruction CPU Design

School of Information Science and Engineering

Southeast University

2014

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. Design Specification

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*.

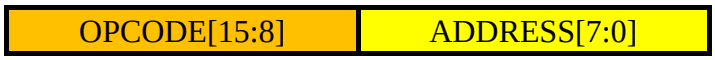

Figure 1 the instruction format

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

Table 1 List of instructions and relevant opcodes

INSTRUCTION	OPCODE	COMMENTS
STORE X	00000001	ACC → [X]
LOAD X	00000010	[X] → ACC
ADD X	00000011	ACC + [X] → ACC
SUB X	00000100	ACC − [X] → ACC
JMPGEZ X	00000101	If ACC ≥ 0 then X → PC else PC+1 → PC
JMP X	00000110	X → PC
HALT	00000111	Halt a program
MPY X	00001000	ACC × [X] → ACC, MR
DIV X	00001001	ACC ÷ [X] → ACC, DR
AND X	00001010	ACC and [X] → ACC
OR X	00001011	ACC or [X] → ACC
NOT X	00001100	NOT [X] → ACC
SHIFTR	00001101	SHIFT ACC to Right 1bit, Logic Shift
SHIFTL	00001110	SHIFT ACC to Left 1bit, Logic Shift
……	……	……

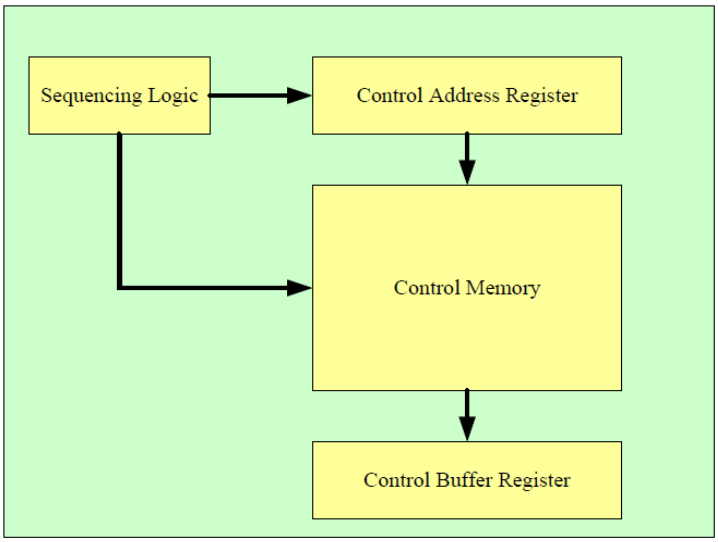

Figure 5 Control Unit Micro-architecture

Figure 5 shows the key elements of Microprogrammed Control Unit. 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.

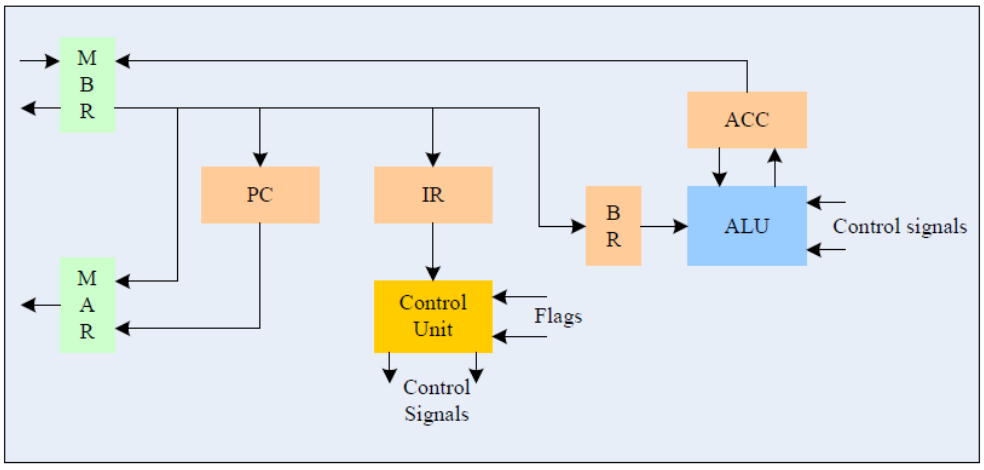

Figure 6 CPU data path and control signals

Figure 6 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.

3. Design Description

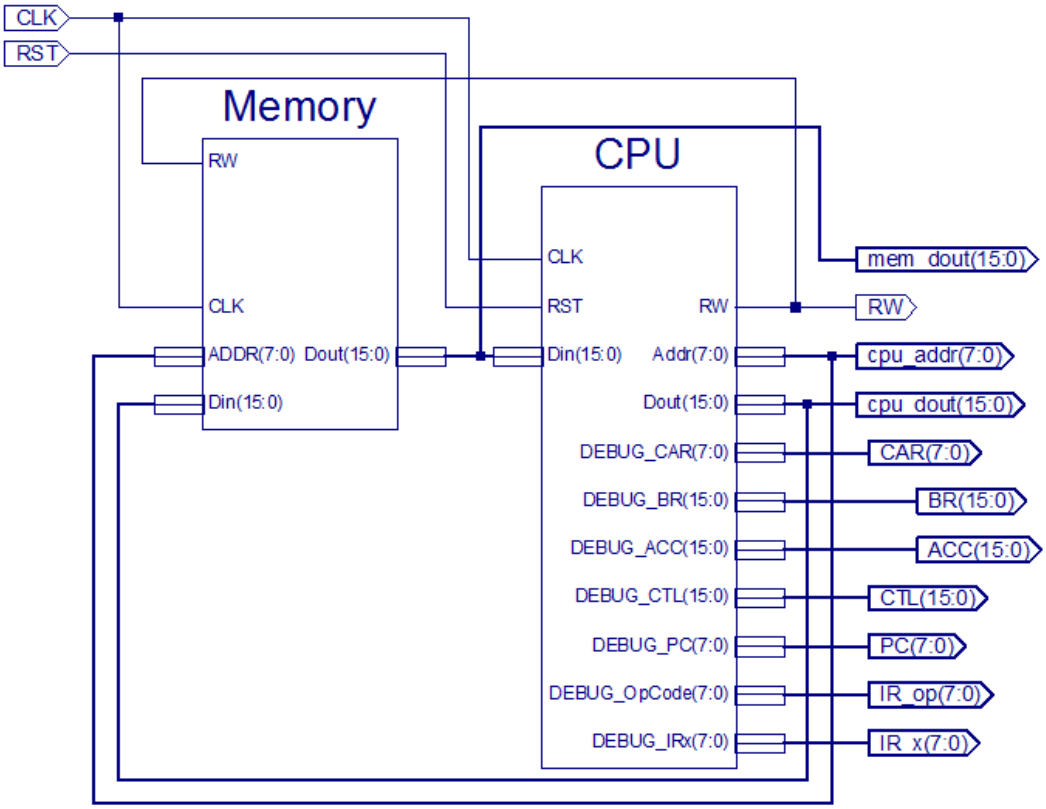

Figure 3-1 The Top Schematic of the simulated CPU

Memory is simulated as a general storage device, storing instructions as well as data; Specifically, Instruction starts from the address @0x0000.

CPU is designed as a simple micro programmed center control unit with a few standard ports and some *DEBUG* ports for observation. The interior Hierarchy is shown as Figure 3-2.

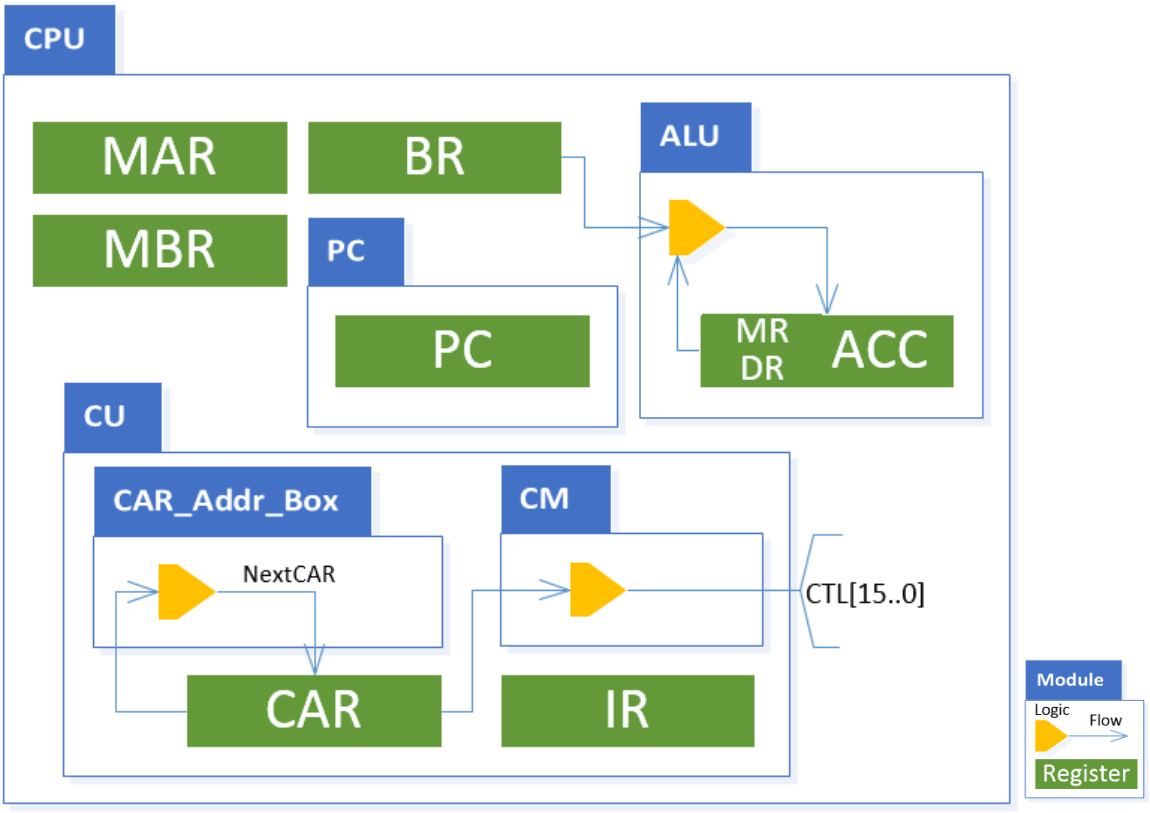

Figure 3-2 CPU Design Hierarchy

Figure 3-2 illustrates the location of the registers and the data flow. The logic unit in *CAR_Addr_Box* and *CM* module is actually small RAM.

4. Design of Important Modules

4.1 ALU

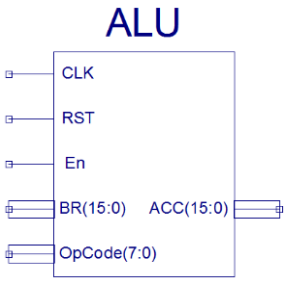

Figure 4-1 The Schematic Symbol of ALU

The math and logic operation is simply implemented by compiler itself with "$signed" macro, thus free from the details of mathematical operations.

Specially, CU informs ALU which operation is required by Opcode in IR, not some particular control signals.

Module Input：

Pin	Function
CLK	Clock
RST	Reset, 1 Valid

En	Execute one ALU operation at positive edge of CLK
BR(15:0)	Buffer Register
OpCode(7:0)	Refer to IR(15:8)

Module Output：

Pin	Function
ACC	ALU result register

4.2 CU

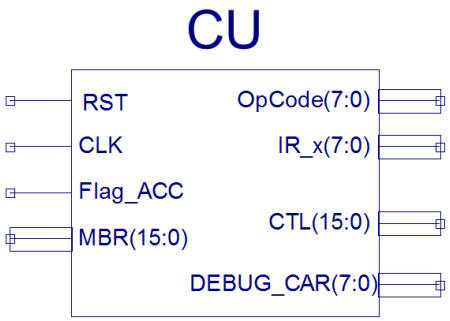

Figure 4-2 The Schematic Symbol of ALU

Module Input：

Pin	Function
CLK	Clock
RST	Reset, 1 Valid
Flag_ACC	ACC[15], the sign bit of ACC
MBR(15:0)	Get Instruction from MBR

Module Output：

Pin	Function
OpCode(7:0)	Refer to IR(15:8)
IR_x(7:0)	Refer to IR(7:0)
CTL(15:0)	Control Signal
DEBUG_CAR(7:0)	For observation

4.3 CM

CM stores action the CPU can perform. With specific CM, this CU module has following features:

1. If no control signals related to CAR, CAR will increase itself automatically.

2. Redundant CM space is implemented to store *CAR_RST*, in prevent CAR runaway.

Table 2 Control Memory Content

Address	Microinstruction	Control Signal
[Get Instruction：----------]		
0x00	MAR ← PC	C6
0x01	PC++ MBR ← memory	C0 C1
0x02	IR ← MBR	C3
0x03	CAR jmp to *OPERATION** if no operand	C12
[Get Operand：----------]		
0x04	MAR ← IR(x)	C7
0x05	MBR← memory	C1
0x06	BR ← MBR CAR jmp to *OPERATION**	C4 C13
[STORE：----------]		
0x08	MBR ← ACC MAR ← IR(x)	C5 C7
0x09	memory ← MBR CAR jmp to @0x00	C2 C15
[ALU OPERATION：----------]		
0x0D	ALU Enable CAR jmp to @0x00	C10 C15
[JMPGEZ：----------]		
0x0F	if AAC⩾0 then CAR jmp to *[JMP] @0x12*	C11
0x10	CAR jmp to @0x00	C15
[JMP：----------]		
0x12	PC ← IR(x) CAR jmp to @0x00	C9 C15
[HALT：----------]		
0x14	CAR Disable	C13
[Default：----------]		
Default	CAR jmp to @0x00	C15

* *OPERATION*: Different Instruction jumps to different address. For example, STORE leads to @0x08, JMP leads to @0x12, but ADD, SUB and other ALU instructions lead to @0x0D, etc.

Table 3 Control Signal Definition

CTL[15:0]	Signal Name	Microinstruction
CTL[0]	PC_inc	PC++
CTL[1]	Memory_Read	MBR← memory
CTL[2]	Memory_Write	memory ← MBR
CTL[3]	IR_Write_fMBR	IR ← MBR
CTL[4]	BR_Write_fMBR	BR ← MBR
CTL[5]	MBR_Write_fACC	MBR ← ACC
CTL[6]	MAR_Write_fPC	MAR ← PC

CTL[7]	MAR_Write_fIRx	MAR ← IR(x)
CTL[8]	**RESERVED	**NULL
CTL[9]	PC_Write_fIRx	PC ← IR(x)
CTL[10]	ALU_Operation	ALU Enable
CTL[11]	CAR_JMPGEZ	if AAC⩾0 then CAR jmp to *[JMP]*
CTL[12]	CAR_Judge_GetX	if no operand CAR jmp to *OPERATION*
CTL[13]	Jmp_to_Operation	CAR jmp to *OPERATION*
CTL[14]	**RESERVED	**NULL
CTL[15]	CAR_RST	CAR jmp to @0x00

* There is no control signal for "CAR Increase", because CAR is designed to automatically increase itself if there is no control signal related to CAR(e.g. CTL[11..15]).

5. Simulation

5.1 Test Program

The design is required to test:

1. (-5) × (-8)
2. 100 ÷ (-3)
3. Sum of 1+3+5+...+97+99

```
LOAD #-5
MPY #-8       //(-5) × (-8)
LOAD #100
DIV #-3       //100 ÷ (-3)
//------Additional Test-- ----
LOAD #99
ADD #-8       //99+(-8)
LOAD #-25
SUB #1        //(-25)-(1)
LOAD #51
AND #15       //51 & 15(0x0033 & 0x000F)
LOAD #5
SHR           //(0x05)>> 1
//--------  Sum of 1+3+5+...+97+99 -----
LOAD #0
STORE count //count=0;
LOAD #1       //i = 1;
LOOP:
   STORE i
   SUB #100
   JGEZ FINISH //while (i<100)
```

```
    LOAD count
    ADD i       //count += i;
    STORE count

    LOAD i
    ADD #2      //i=i+2;
    JMP LOOP

FINISH:         //Program halt
    HALT
```

5.2 Simulation Result

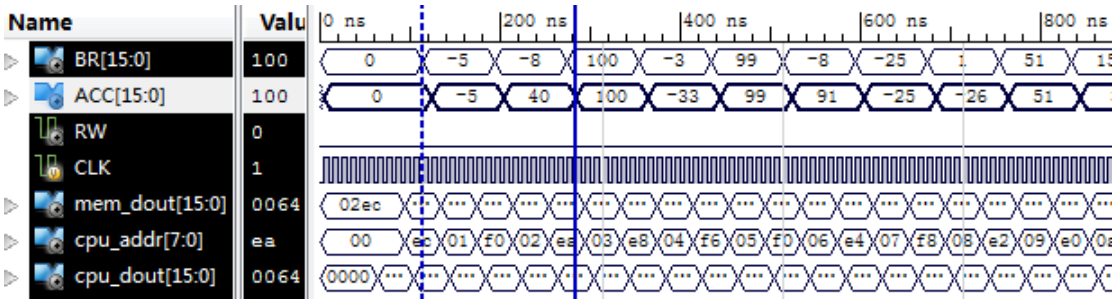

Figure 5-1： $-5\times-8=40$

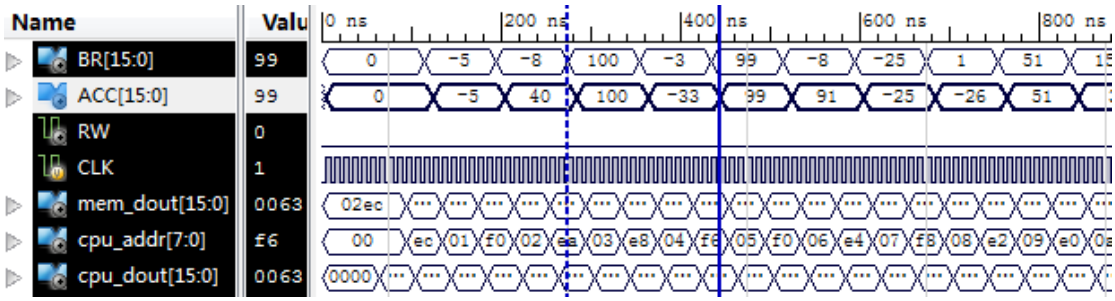

Figure 5-2： $100\div-3=-33$

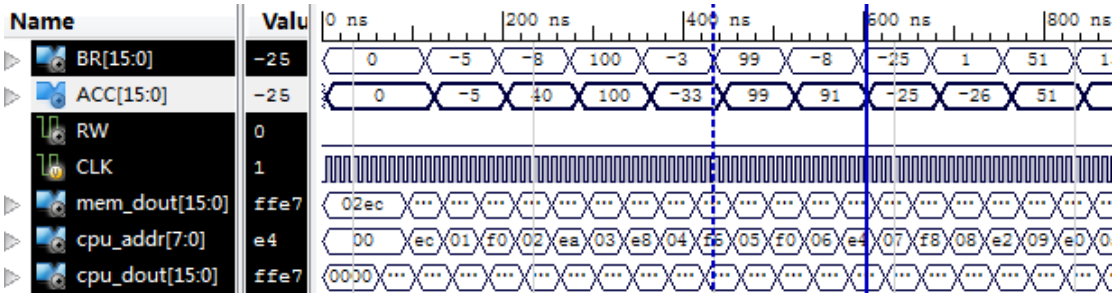

Figure 5-3： $99+(-8)=91$

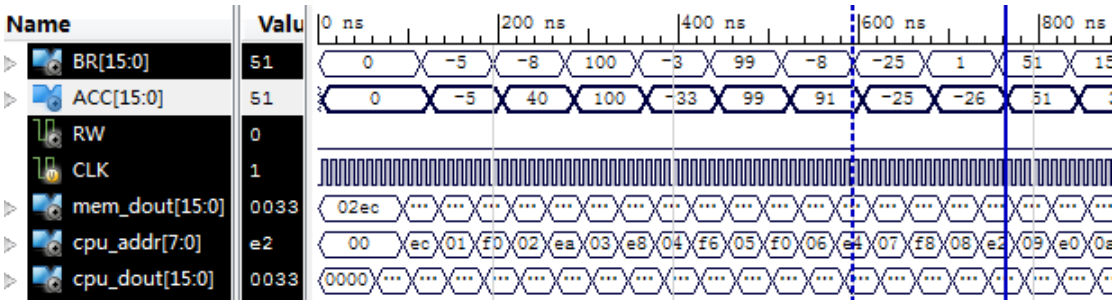

Figure 5-4 : $-25-1=-26$

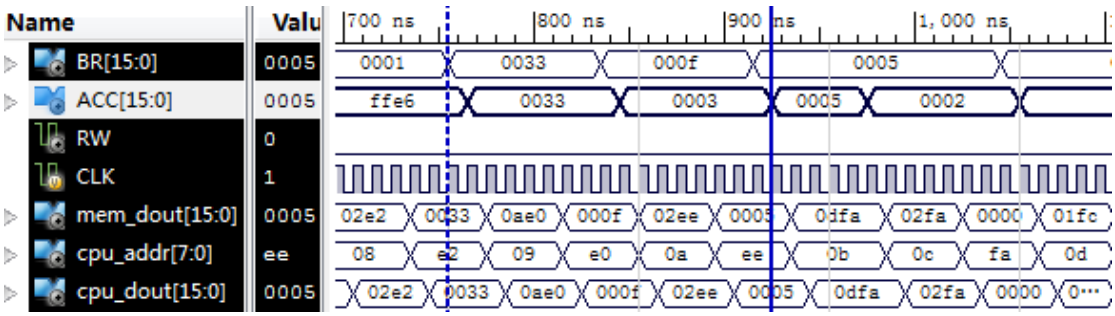

Figure 5-4 : 0x0033 & 0x000F = 0x0003

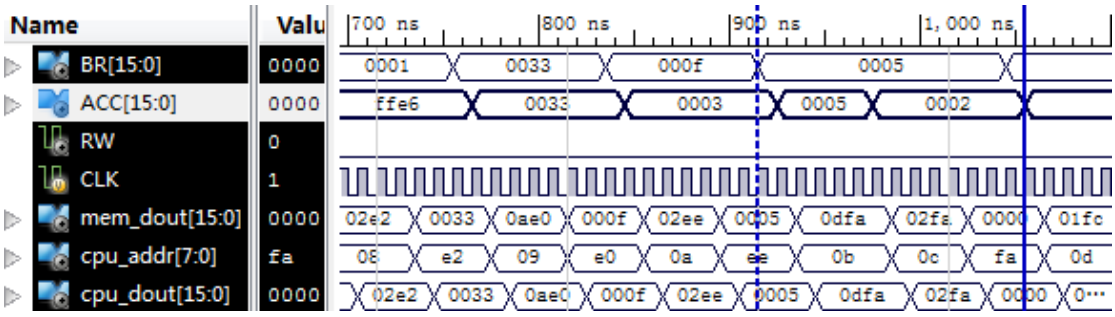

Figure 5-4 : 0x0005 >> 1= 0x0002

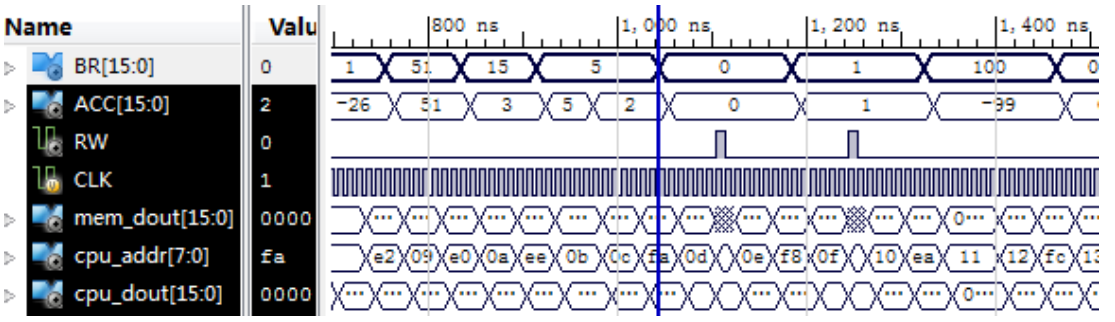

Figure 5-5 : Start calculating 1+3+5+..+97+99 at 1020ns

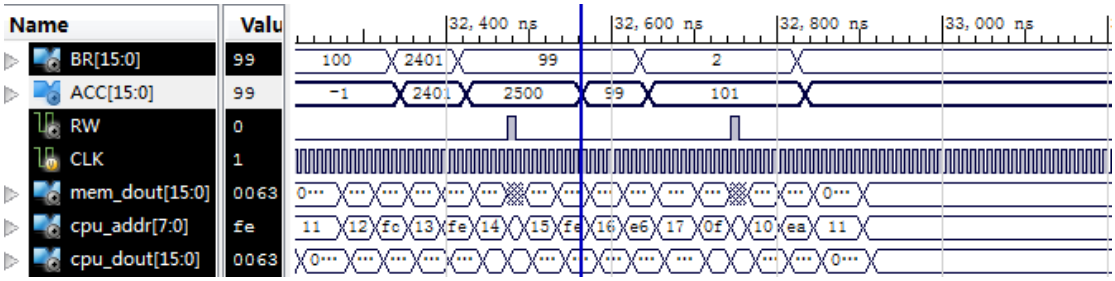

Figure 5-6 : Finish calculating 1+3+5+...+97+99=2500 at 32580ns, consuming

3156 CPU Clock Cycles.

--- Appendix:

/********** CommonMacro.v **********/

```
`define bit4 [3:0]
`define bit8 [7:0]
`define bit16 [15:0]

`define reg4   reg[3:0]
`define reg8   reg[7:0]
`define reg16  reg[15:0]

`define wire4  wire[3:0]
`define wire8  wire[7:0]
`define wire16 wire[15:0]
```

/********** End of CommonMacro.v **********/

/********** InstrucionSet.v **********/

```
`define   STORE  8'h01    //Opcode and Instruction
`define   LOAD   8'h02
`define   ADD    8'h03
`define   SUB    8'h04
`define   JMPGEZ 8'h05
`define   JMP    8'h06
`define   HALT   8'h07
`define   MPY    8'h08
`define   DIV    8'h09
`define   AND    8'h0A
`define   OR     8'h0B
`define   NOT    8'h0C
`define   SHIFTR 8'h0D
`define   SHIFTL 8'h0E
////////////////////////////////////////
`define   PC_inc             CTL[0]
`define   Memory_Read        CTL[1]
`define   Memory_Write       CTL[2]
`define   IR_Write_fMBR      CTL[3]
`define   BR_Write_fMBR      CTL[4]
`define   MBR_Write_fACC     CTL[5]
`define   MAR_Write_fPC      CTL[6]
`define   MAR_Write_fIRx     CTL[7]
```

```
//`define NULL                CTL[8]  //Reserved
`define   PC_Write_fIRx       CTL[9]
`define   ALU_Operation       CTL[10]
`define   CAR_JMPGEZ          CTL[11]
`define   CAR_Judge_GetX      CTL[12]
`define   Jmp_to_Operation    CTL[13]
//`define NULL                CTL[14]  // Reserved
`define   CAR_RST             CTL[15]
```

/********** End Of InstrucionSet.v **********/

/********** CPU.v **********/

```
module CPU(

);
```

/********** End Of CPU.v **********/

/********** ALU.v **********/

```
module ALU(

);
```

/********** End Of ALU.v **********/

/********** CU.v **********/

```
module CU(

);
```

/********** End Of CU.v **********/

/********** CAR_Addr_Box.v **********/

```
module CAR_Addr_Box(
);
```

/********** End Of CAR_Addr_Box.v **********/

/********** CM.v **********/

```
module CM(
    input  `bit8  ADDR,     output `bit16 CTL
);
    `reg16 cm[0:2**8-1];
```

```
    Initial begin
        $readmemh("D:\\CPU\\mem\\cm.txt", cm);
    end

    assign CTL = cm[ADDR];
endmodule
```

/********** End Of CM.v **********/

/********** PC.v **********/

```
module PC(

);
```

/********** End Of PC.v **********/

/********** Memory.v **********/

```
module Memory(
    input         RW,
    input  `bit8  ADDR,
    input  `bit16 Din,
    output `bit16 Dout,

    input CLK
);
`reg16 mem[0:2**8-1];
Initial begin
    $readmemh("D:\\CPU\\mem\\mem.txt", mem);
end

always @(posedge CLK)
begin
    if (RW) mem[ADDR] <= Din;
end

assign Dout = mem[ADDR];
endmodule
```

/********** End Of Memory.v **********/