Arithmetic & Logic Unit (ALU)

Hengyi Shi CS 147
Computer Science, San Jose State University

***Abstract*—This report description of the implementation of a behavioral model of a computer system with a 32-bit processor and 256MB memory which support the CS147DVg using verilog .**

I. Introduction *(Heading 1)*

Processor is design to execute instructions, therefore to finish the instruction set which supports CS147DV language became the first step to do.

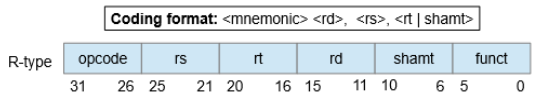
The fully contents of CS147DV operations is listed below:

For R type:

'CS147DV' Instruction Set

Name	Mnemonic	Format	Operation	OpCode /funct
Addition	add	R	R[rd] = R[rs] + R[rt]	0x00 / 0x20
Subtraction	sub	R	R[rd] = R[rs] - R[rt]	0x00 / 0x22
Multiplication	mul	R	R[rd] = R[rs] * R[rt]	0x00 / 0x2c
Logical AND	and	R	R[rd] = R[rs] & R[rt]	0x00 / 0x24
Logical OR	or	R	R[rd] = R[rs] | R[rt]	0x00 / 0x25
Logical NOR	nor	R	R[rd] = ~(R[rs] | R[rt])	0x00 / 0x27
Set less than	slt	R	R[rd] = (R[rs] < R[rt])?1:0	0x00 / 0x2a
Shift left logical	sll	R	R[rd] = R[rs] << shamt	0x00 / 0x01
Shift right logical	srl	R	R[rd] = R[rs] >> shamt	0x00 / 0x02
Jump Register	jr	R	PC = R[rs]	0x00 / 0x08

Coding format: <mnemonic> <rd>, <rs>, <rt | shamt>



12

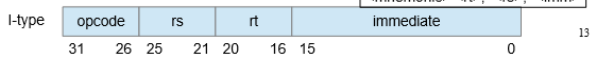
For I type:

'CS147DV' Instruction Set

Name	Mnemonic	Format	Operation	OpCode
Addition immediate	addi	I	R[rt] = R[rs] + SignExtImm	0x08
Multiplication immediate	muli	I	R[rt] = R[rs] * SignExtImm	0x1d
Logical AND immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	0x0c
Logical OR immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	0x0d
Load upper immediate	lui	I	R[rt] = {imm, 16'b0}	0x0f
Set less than immediate	slti	I	R[rt] = (R[rs] < SignExtImm)?1:0	0x0a
Branch on equal	beq	I	If (R[rs] == R[rt]) PC = PC + 1 + BranchAddress	0x04
Branch on not equal	bne	I	If (R[rs] != R[rt]) PC = PC + 1 + BranchAddress	0x05
Load word	lw	I	R[rt] = M[R[rs]+SignExtImm]	0x23
Store word	sw	I	M[R[rs]+SignExtImm] = R[rt]	0x2b

BranchAddress = {16{Imm[15]}, immediate }

Coding format: <mnemonic> <rt>, <rs>, <imm>



13

For J type:

'CS147DV' Instruction Set

Name	Mnemonic	Format	Operation	OpCode
Jump to address	jmp	J	PC = JumpAddress	0x02
Jump and Link	jal	J	R[31] = PC + 1; PC = JumpAddress	0x03
Push to Stack	push	J	M[$sp] = R[0] $sp = $sp - 1	0x1b
Pop from Stack	pop	J	$sp = $sp + 1 R[0] = M[$sp]	0x1c

JumpAddress = { 6'b0, address } // zero extend for 6 bit

Coding format: <mnemonic> <address>

14

Therefore by reusing the code from project 1, I firstly developed the ALU file. And getting the correct testing out put:

```
[TEST] 15 + 3 = 18 , got 18 ... [PASSED]
[TEST] 15 - 5 = 10 , got 10 ... [PASSED]
[TEST] 15 + 5 = 20 , got 20 ... [PASSED]
[TEST] 46 - 12 = 34 , got 34 ... [PASSED]
[TEST] 147 - 110 = 37 , got 37 ... [PASSED]
[TEST] 10 * 40 = 400 , got 400 ... [PASSED]
[TEST] 18 * 10 = 180 , got 180 ... [PASSED]
[TEST] 19 >> 2 = 4 , got 4 ... [PASSED]
[TEST] 1991 >> 3 = 248 , got 248 ... [PASSED]
[TEST] 19 << 2 = 76 , got 76 ... [PASSED]
[TEST] 1926 << 4 = 30816 , got 30816 ... [PASSED]
[TEST] 0 & 1 = 0 , got 0 ... [PASSED]
[TEST] 13 & 6 = 4 , got 4 ... [PASSED]
[TEST] 1 & 1 = 1 , got 1 ... [PASSED]
[TEST] 0 | 1 = 1 , got 1 ... [PASSED]
[TEST] 0 | 0 = 0 , got 0 ... [PASSED]
[TEST] 3 | 2 = 3 , got 3 ... [PASSED]
[TEST] 0 ~| 1 = 4294967294 , got 4294967294 ... [PASSED]
[TEST] 0 ~| 0 = 4294967295 , got 4294967295 ... [PASSED]
[TEST] 1 ~| 1 = 4294967294 , got 4294967294 ... [PASSED]
[TEST] 5 < 10 = 1 , got 1 ... [PASSED]
[TEST] 15 < 10 = 0 , got 0 ... [PASSED]
[TEST] 5 < 5 = 0 , got 0 ... [PASSED]

      Total number of tests          23
      Total number of pass           23
```

II. Design and Implement Memory

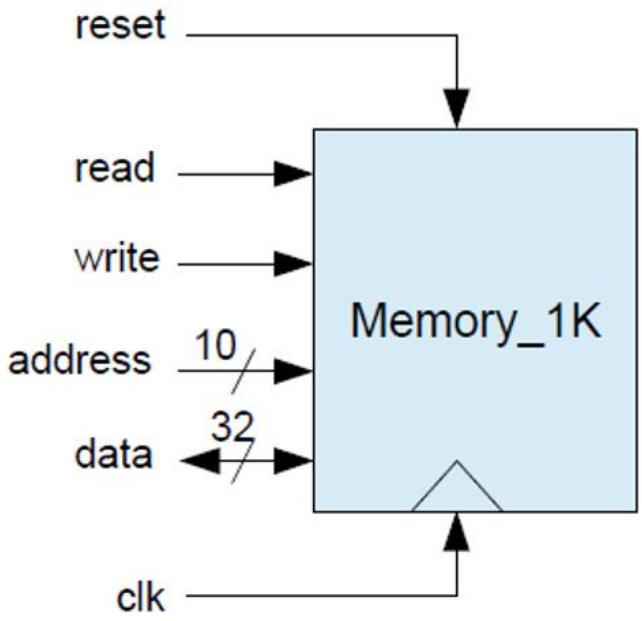
The Verilog provides a simple way to implement a memory model which can use ModelSim to simulate the low level of simulation such as RTL level and Gate level implementation.

Design Description:

For each memory unite, set it as a module while noting the I/O ports.

```
module memory (R, W, rst, I, D)
    input Read;
    input Write;
    input rst;
    input addr[0:9];
    inout Data[0:31];
    reg [0:31] data_str [0:63]
//operation code
endmodule
```

The inputs of read write and reset control are all single bit. The address input is 10 bits. The data come in/out memory is 32 bits.

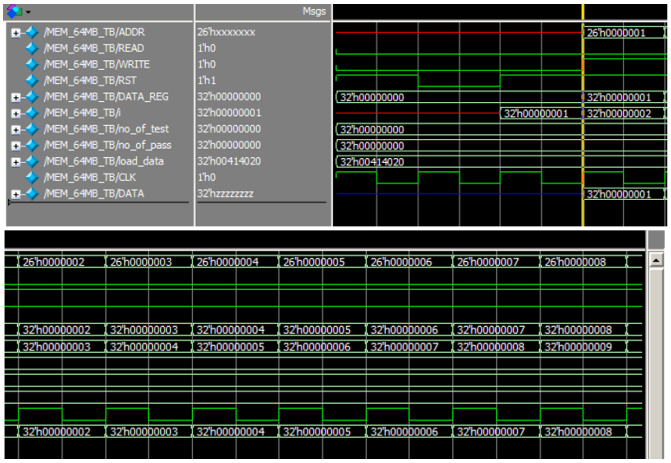


Also, 32x64 two-dimensional array can make sure to fit the specifications.

The read write reset operation is implemented by changing the machine state signal, Read, write, and Address. Describing as:

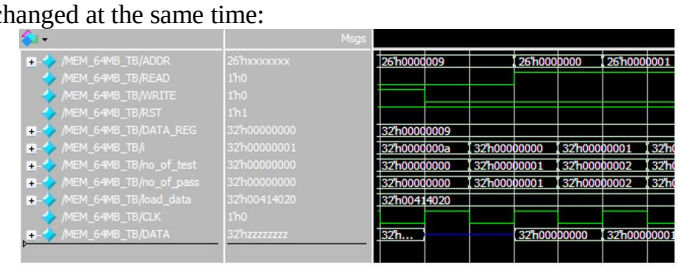
```
always @ (negedge RST or posedge CLK) begin
        if (RST === 1'b0) begin
            for(i=0;i<=`MEM_INDEX_LIMIT; i = i +1) begin
                sram_32x64m[i] = { `DATA_WIDTH{1'b0} };
            end
        $readmemh(mem_init_file, sram_32x64m);
        end
        else begin
            if ((READ===1'b1)&&(WRITE===1'b0))/*read*/ begin
                data_ret =  sram_32x64m[ADDR];
            end
            else if ((READ===1'b0)&&(WRITE===1'b1))/*write*/ begin
                sram_32x64m[ADDR] = DATA;
            end
        end
    end
```

Testing Result:

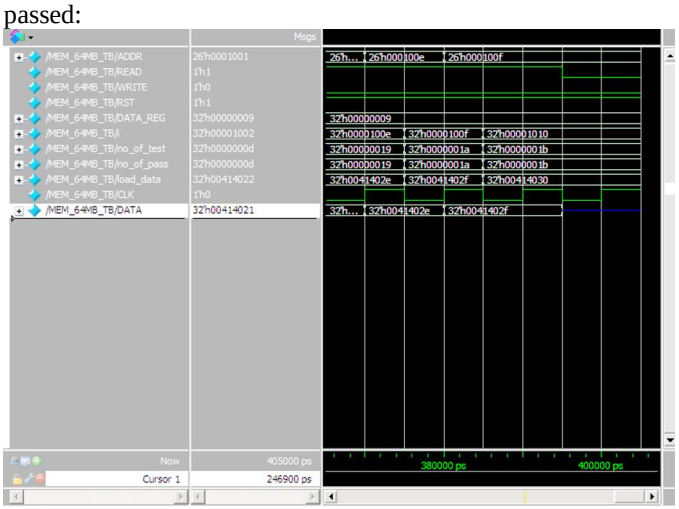


From the result-diagram which shows that the initialisation of the memory, reset signal,is setting the opera tion to write and the beginning of writing file data into the memory which shows the previous implementation is working.

As the simulations change from write to read, the signal also changed at the same time:



The end of test from /DATA part, it shows the tset result passed:



III. Design and Implementation of Processor

The processor is implemented as a summation of its compon ents. Our processor contains a Control Unit, a 32x32 Register a nd an ALU.

Treat the processor as a module and note the i/o ports.

There are out ports for ADDR, READ and WRITE.

There is also an In/Out port required for DATA.

Outline:

```
module memory (R, W, rst, I, D)      input CLK;      input RST;      output READ;
    output WRITE;
    output ADDR[0:9];      inout DATA[0:31];      Control_Unit cu_init(...);
    Regester_File rf_init(...);      ALU alu_init(...);  endmodule
```

The inputs of read write and reset control are all single bit. The address input is 10 bits. The data come in/out memory is 32 bits.

For control unit design, I completed the file by following these instructions:

NOTE: The Register file operation is similar to the memory operation. Only it does not handle READ=WRITE states, allo wing it to retain previous data.

NOTE: ALU operation has been described in other docume ntation. Refer to ALU documentation for implementation and t esting.

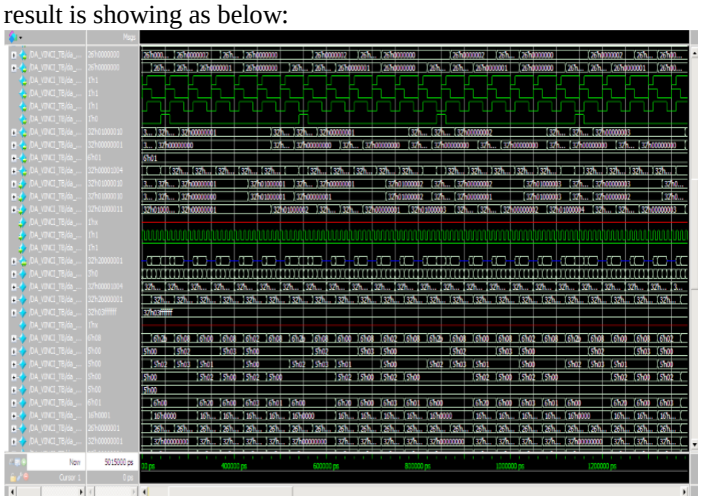
```
module Control_Unit(...)
   output [31:0] RF_DATA_W, ALU_OP1, ALU_OPT2;
   output [25:0] RF_ADDR_W, RF_ADDR_R1, RF_ADDR_R2, MEM_ADDR;
   output RF_READ, RF_WRITE, MEM_READ, MEM WRITE;
   input [32:0] RF_DATA, RF_DATA_R1, RF_DATA_R2, ALU_RESULT;
   input CLK, RST, ZERO;
   inout MEM_DATA;
   reg [25:0] PC_REG, INST_REG, SP_REG;
   assign MEM_DATA = ((MEM_READ==0)&&(MEM_WRITE==1))?mem_data_ret:{z};
   initial begin
      PC_REG = 000001000;
      SP_REG = 03ffffff;
   end
   //operation code
endmodule
```

Main module contains 5 parts fetch, decode, exe, mem, write back.

```
case(proc_state)
`FETCH
   MEM_ADDR=PC_REG;
`DECODE
   {opcode, rs, rt, immediate} = INST
   {opcode, address} = INST
   {opcode, rs, rt, rd, shmat, funct} = INST
//various padding assignments
`EXE
   case(opcode)
   r-type:
      case(funct)
      //various r-type tests and operations
      default: $write("error");
  //individual J and I type operations
  default: $write("error");
`MEM
   MEM_READ = MEM_WRITE = 0;
   case(opcode)
   Load Word:
      //mem opperations
   Store Word:
      //mem opperations
   Push:
      //mem opperations
   Pop:
      //mem opperations
   default: //not used
`WB
   RF_READ=0;
```

IV. Test Da Vinci

A Test Bench can now be built to test the entire system, the result is showing as below:



Expected output matches the actual output for the fibonacci sequence:

```
// memory data file (do not edit the following line - required for mem load use)
//
instance=/DA_VINCI_TB/da_vinci_inst/memory_inst/sram_32x64m
// format=hex addressradix=h dataradix=h version=1.0 wordsperline=1 noaddress
00000000
00000001
00000001
00000002
00000003
00000005
00000008
0000000d
00000015
00000022
00000037
00000059
00000090
000000e9
00000179
```

```
// memory data file (do not edit the following line - required for mem load use)
//
instance=/DA_VINCI_TB/da_vinci_inst/memory_inst/sram_32x64m
// format=hex addressradix=h dataradix=h version=1.0 wordsperline=1 noaddress
00000000
00000001
00000001
00000002
00000003
00000005
00000008
0000000d
00000015
00000022
00000037
00000059
00000090
000000e9
00000179
```

As well as the remaining test:

```
//
instance=/DA_VINCI_TB/da_vinci_inst/memory_inst/sram_32x64m
// format=hex addressradix=h dataradix=h version=1.0 wordsperline=1 noaddress
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
0000000f
00000002
```

```
//
instance=/DA_VINCI_TB/da_vinci_inst/memory_inst/sram_32x64m
// format=hex addressradix=h dataradix=h version=1.0 wordsperline=1 noaddress
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
0000000f
00000002
```

The project is done.

V. Conclusion

From project 2 I got the much better understanding about the Verilog language, the ModelSim IDE, the operation of a computer. From this project I also get to know the relation to the clock cycles and the/division of operation for a instruction execution.