Lab 1
CPU Architecture, ALU, Register Files

Fall 2008

低功率高效能VLSI設計實驗室

NCKU, Low power, high performance VLSI design lab

Lab Overview

- Target Design
  - A datapath consisting of ALU and register file
- This ppt first reviews the architecture for SMILE CPU. An data_flow.pps file is given for you to observe the behavior of the architecture.
- Instruction set is then defined for SMILE
- Major components like ALU and register file are explained and described.
- Detailed description for each step are in Lab1_handout.

Outline

- CPU Architecture
- Instruction Format
- Arithmetic Logic Unit
- Register File
- Reference Codes

CPU Architecture (1/4)

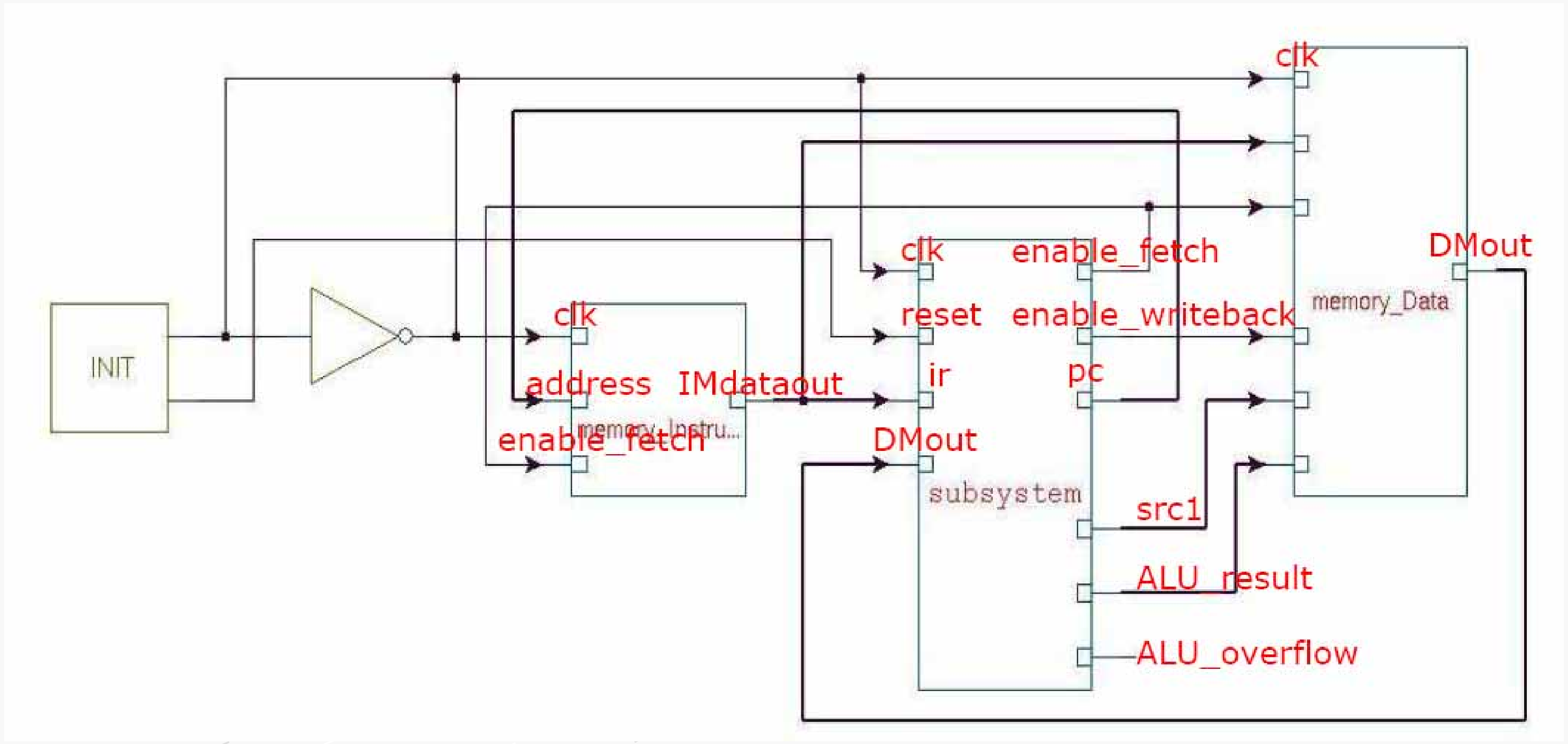
- Central Processor Unit (CPU) – An active part in the computer which adds numbers, tests numbers, signal I/O devices to activate, and so on….
- CPU architecture is a high level description of the hardware to implement the CPU
- 3 different architectures:
  - Single cycle CPU
    - – An instruction is executed in one clock cycle
  - Multicycle CPU
    - – An instruction is executed in multiple clock cycles
  - Superscalar CPU
    - – An advance pipelining CPU architecture that execute more than one instruction per clock cycle

CPU Architecture (2/4)

- CPU consists of three main blocks:
  - Datapath – The component that performs arithmetic and logical operations, e.g. ALU
  - Controller – The component that commands the datapath, memory, and I/O devices according to the instructions of the program
  - Memory – A storage area in which programs and data are stored, e.g. Register Files, Instruction Memory…
- **In this homework1, a Multicycle CPU will be designed and synthesized**

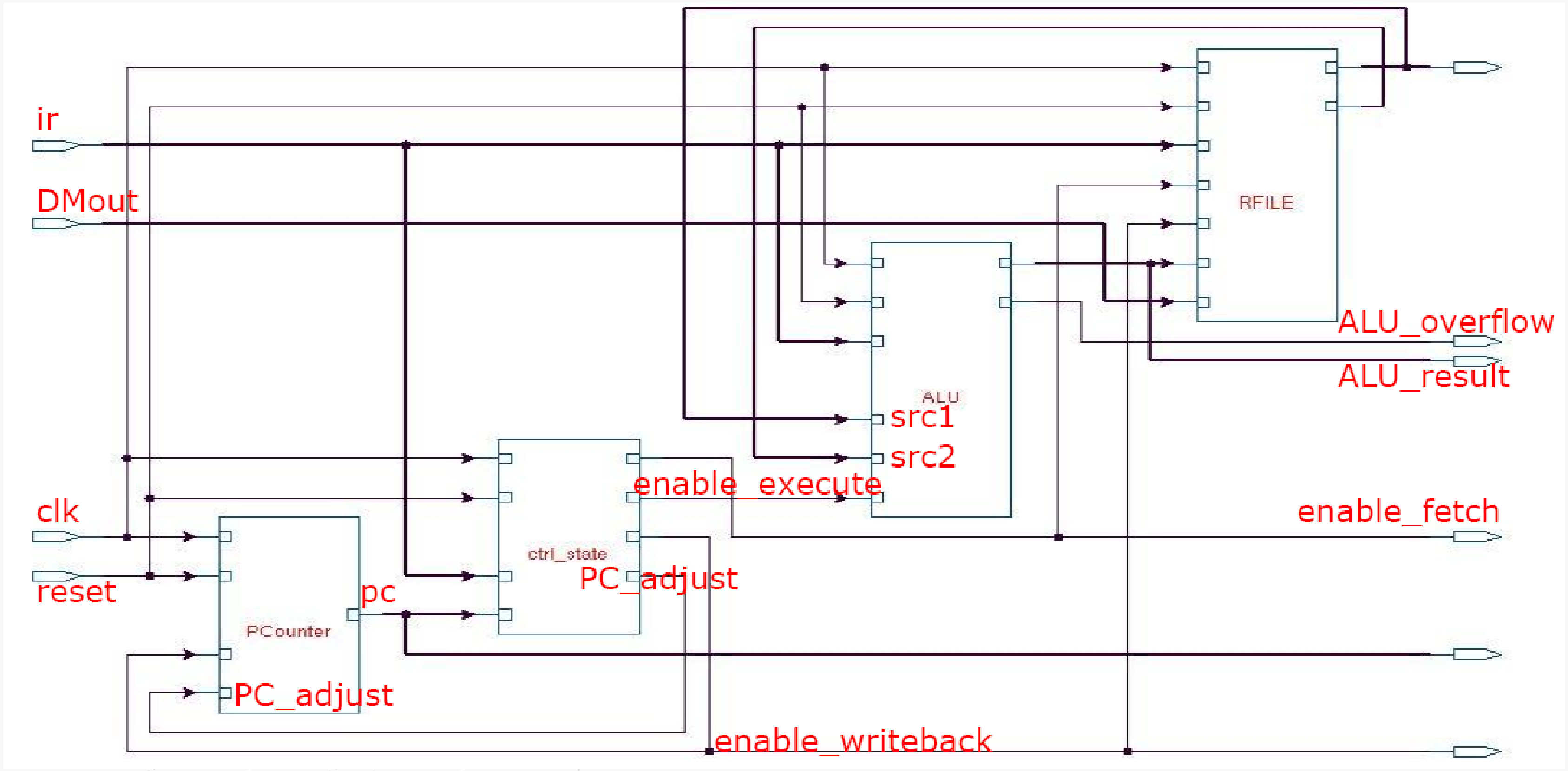
CPU Architecture (3/4)

- Overall CPU Architecture



CPU Architecture (4/4)

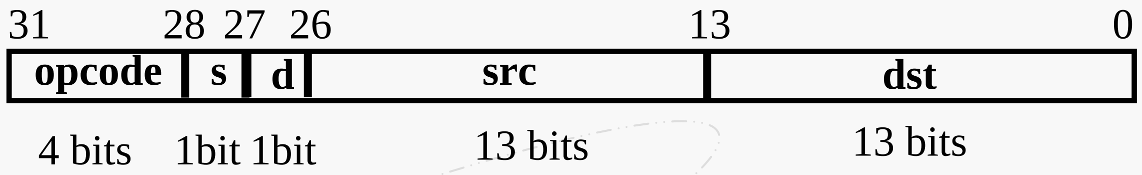
- Subsystem Block Diagram



For animation of SMILE CPU, please open data_flow.pps.

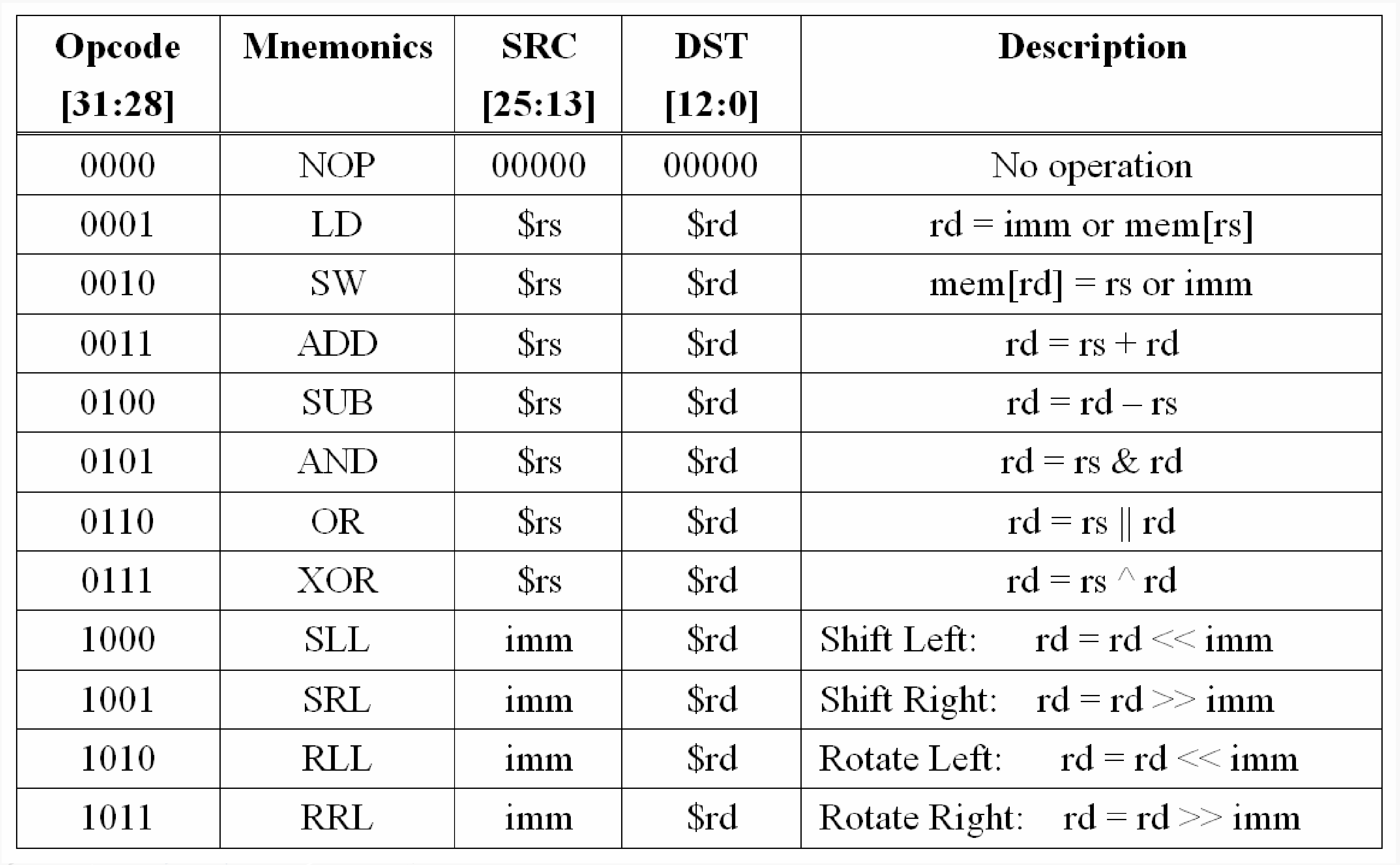
Instruction Format (1/5)

- Instruction range:
  - `define OPCODE ir[31:28]
  - `define SRCTYPE ir[27]
  - `define DSTTYPE ir[26]
  - `define SRC ir[25:13]
  - `define DST ir[12:0]



  - SRC Types: 1: Immediate value, 0: Register File type
  - DST Types: 1: Data Memory type, 0: Register File type,

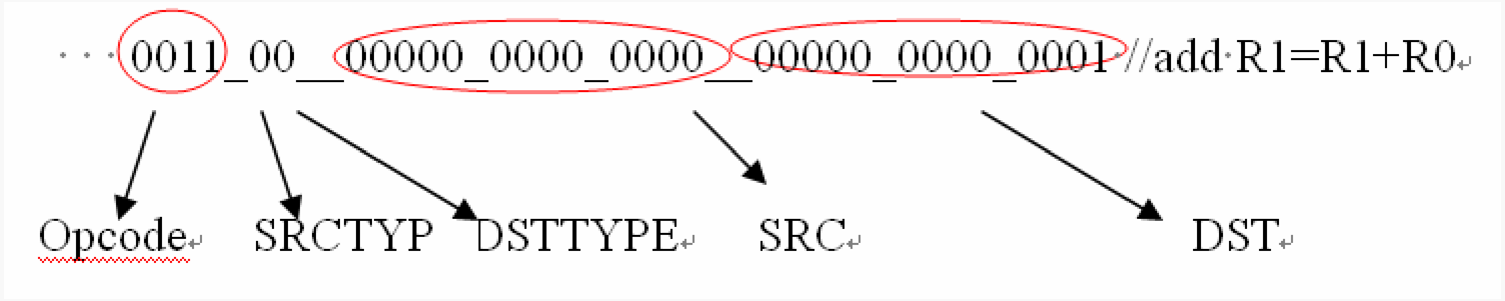
Instruction Format (2/5)



Opcode [31:28]	Mnemonics	SRC [25:13]	DST [12:0]	Description
0000	NOP	00000	00000	No operation
0001	LD	$rs	$rd	rd = imm or mem[rs]
0010	SW	$rs	$rd	mem[rd] = rs or imm
0011	ADD	$rs	$rd	rd = rs + rd
0100	SUB	$rs	$rd	rd = rd – rs
0101	AND	$rs	$rd	rd = rs & rd
0110	OR	$rs	$rd	rd = rs || rd
0111	XOR	$rs	$rd	rd = rs ^ rd
1000	SLL	imm	$rd	Shift Left: rd = rd << imm
1001	SRL	imm	$rd	Shift Right: rd = rd >> imm
1010	RLL	imm	$rd	Rotate Left: rd = rd << imm
1011	RRL	imm	$rd	Rotate Right: rd = rd >> imm

Instruction Format (3/5)

- Example



- 0011_10__00000_0000_1101__00000_0000_0000 //add R0=R0+13
- 0001_10__00000_0000_0111__00000_0000_0000 //LD R0<=7
- 0001_00__00000_0000_0000__00000_0000_0010 //LD R2<=M0
- 0010_00__00000_0000_0000__00000_0000_0000 //SW M0<=R0
- 0010_10__00000_1111_1111__00000_0000_0001 //SW M1<=2'hff

Instruction Format (4/5)

- Instruction Decoder – A datapath unit that will decode the an instruction by generating the corresponding fields to be used by the controller to carry out the corresponding operations

```
`define OPCODE ir[31:28]
`define SRCTYPE ir[27]
`define DSTTYPE ir[26]
`define SRC ir[25:13]
`define DST ir[12:0]
```

- Shift Operation
  – data "0" will replace the shifted position
- Rotate Operation
  – data is rotated from one end to the other end

Instruction Format (5/5)

- LD – Loading data to Register File
  - `DSTTYPE = 0

```
0001_10__00000_0001_0101__00000_0000_0000 //LD  R0 21
0001_00__00000_0000_0001__00000_0000_0001 //LD  R1 <= M1
```

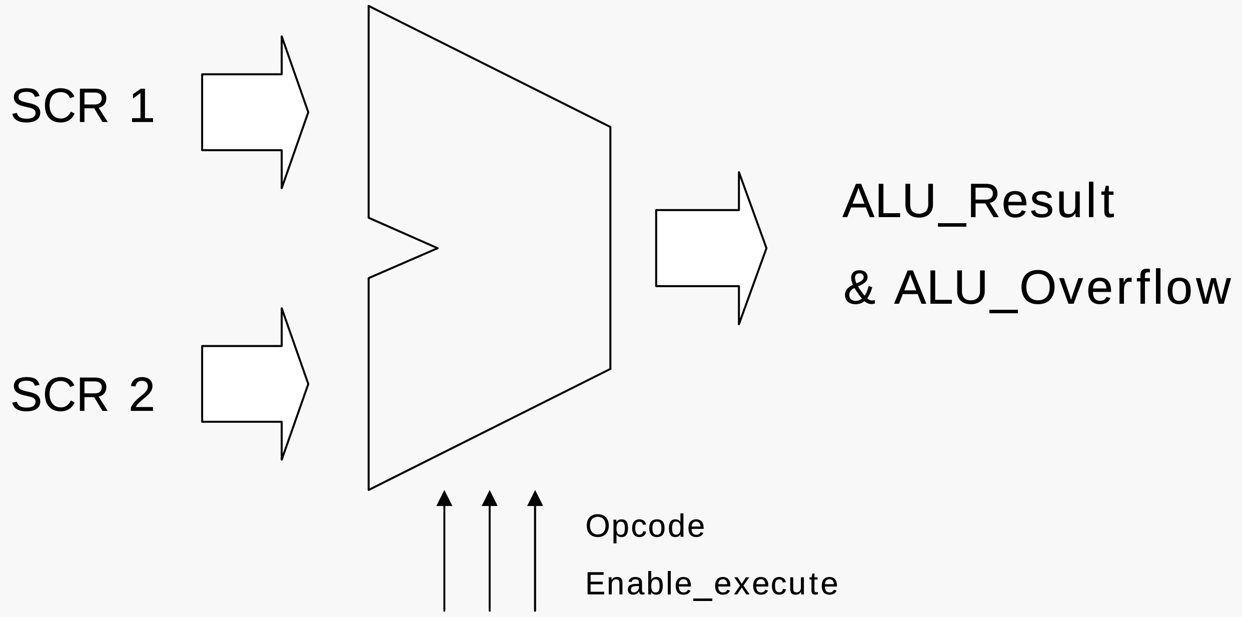
```
If(`SRCTYPE==1) RFILE_array[`DST]=`SRC;
else RFILE_array[`DST]=the value from Data Memory
```

- SW – Store a word to Data Memory
  - `DSTTYPE = 1

```
0010_01__00000_0000_0000__00000_0000_0000 //SW  M0=R0
0010_11__00000_1111_1111__00000_0000_0001 //SW  M1=2'hff
```

```
If(`SRCTYPE==1) mem_data[`DST]=`SRC;
else mem_data[`DST]=the value from RFILEIn
```

Arithmetic Logic Unit (1/3)



- Combinational circuit performs a set of basic operation using two inputs to generate the result

Arithmetic Logic Unit (2/3)

- ALU unit will read the data from SCR1 & SCR2 and perform the designated operation
- Output contains the ALU_Result and ALU_Overflow
- *opcode* will be used to select which operation to be executed
- Control line signals to control ALU operation
- Consists of three different combinational circuit building blocks:
  - Add/Sub to perform addition and subtraction arithmetic operations
  - Logic block to perform logical operations
  - Barrel Shifter to perform shift and rotate operations

Arithmetic Logic Unit (3/3)

- Add/Sub block:

```
assign result=(control == 0)?(operand2+operand1):(operand2-operand1);
```

- Logic block

```
and_result=src2&src1;
or_result=src2|src1;
xor_result=src2^src1;
```

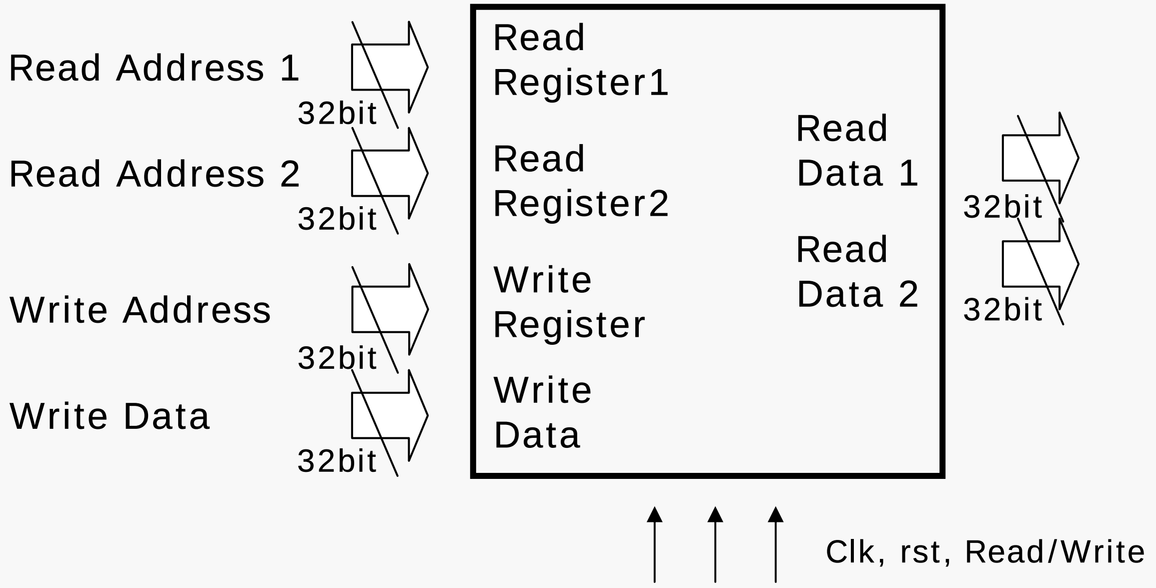
- Shift block
  - Rotate Operation:

```
output [31:0] result; // result operand
reg    [95:0] rotate_reg;
rotate_reg=(direction)?rotate_reg>>in1:rotate_reg<<in1;
result=rotate_reg[63:32];
```

  - Shift Operation:

```
result=(direction)?in2>>in1:in2<<in1;
```

Register File (1/3)

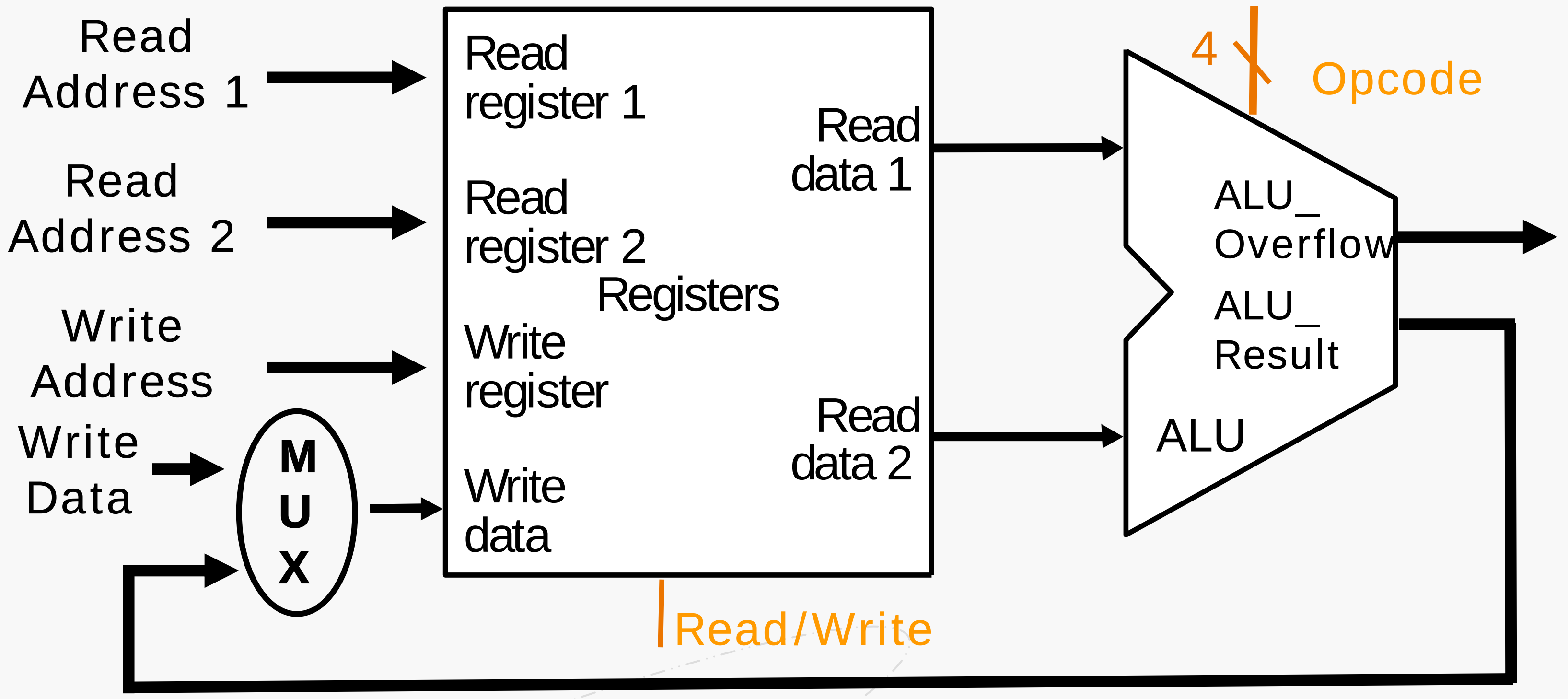


- Special type of fast memory that permits one or more words to be read and one or more words to be written simultaneously
- For this CPU design, consists of 32 registers:
  - Two Read Ports and two data output ports (for source register and destination register)
  - One Write Port and one Write Data Port

Register File (2/3)

- Working at positive edge of clock
- When reset, data values in all registers are reset to zero
- When enable is high and write is asserted, the write_data is stored into the register as pointed by write_address
- When enable is high and read is asserted, the data that are stored in registers as pointed by read_address1 & read_address2 are written to src1 & src2

Register File (3/3)



- Combining a Register File and an ALU unit to form a subsystem that could carry out the basic operation of a CPU

Reference Code 1 (ALU)

```
module ALU (opcode, src1, src2, enable_execute, alu_result, alu_overflow);
input   enable_execute;
input   [31:0] src1, src2;
Input [3:0] opcode;
output  [31:0] alu_result;
output  alu_overflow;
reg     alu_overflow;
reg     [31:0] alu_result;
wire    [32:0] addsub_result;
wire    [31:0] shf_result;
wire    [31:0] and_result=src2&src1;
wire    [31:0] or_result=src2|src1;
wire    [31:0] xor_result=src2^src1;
wire    add_control=(`OPCODE==`ADD)?1'b0:1'b1;
wire    shf_type=(`OPCODE==`SLL || `OPCODE==`SRL)?1'b0:1'b1;
wire    shf_dir= (`OPCODE==`SLL || `OPCODE==`RLL)?1'b0:1'b1;
add_sub ADDSUB(src1, src2, addsub_result, add_control);
barrel_shifter SHT_ROT(src1, src2, shf_dir, shf_type, shf_result);
…
always @(opcode or enable_execute)
begin
       if(enable_execute)
       begin
       case(`OPCODE)
       `ADD: begin {alu_overflow, alu_result}=addsub_result;
          end
       `SUB:  begin {alu_overflow, alu_result}=addsub_result;
          end
       … (Please add-in your own codes)
        default:{alu_overflow, alu_result}=0;
       endcase
     end
```

```
//adder and substator
module add_sub (operand1,operand2,result,control);
input  [31:0] operand1,operand2;
input  control;
output [32:0] result;
wire   [32:0] result;

assign result=(control == 0)?(operand2+operand1):(operand2-operand1);
endmodule

//Barrel_shifter
module barrel_shifter (in1,in2,direction,type,result);
input  [31:0] in1,in2;// input operand
input  direction;     // left or right
input  type;          // shift or rotate
output [31:0] result; // result operand

reg    [31:0] result; // shift/rotate result
reg    [95:0] rotate_reg;

always @(in1 or in2 or direction or type)
begin
  rotate_reg={in2,in2,in2};
  if(type) begin
    rotate_reg=(direction)?rotate_reg>>in1:rotate_reg<<in1;
    result=rotate_reg[63:32];
  end
  else begin
    result=(direction)?in2>>in1:in2<<in1;
  end
end
endmoduleArithmetic Logic Unit
```

Reference Code 2 (ALU)

```
`timescale 1ns/10ps
module ALU( OverFlow, OUT, in_a, in_b, OP, enable);
parameter ISize = 32;
parameter OSize = 4;  //2^3>6
input       enable;
input       [ISize-1:0] in_a;
input       [ISize-1:0] in_b;
input       [OSize-1:0] OP;
output      [ISize-1:0] OUT;                        //33bits
output      OverFlow;
reg         [ISize-1:0] OUT;
reg         [2*ISize-1:0] temp;
reg         OverFlow;

always@(OP or enable)begin
 OverFlow = 0;
  if(enable)begin
   case(OP)
    4'b0000 :       OUT = OUT;                      //NOP
    4'b0001 :       OUT = in_a;                     //load
    4'b0010 :       OUT = in_a;                     //store
    4'b0011 :       begin
                    OUT = in_a + in_b;              //ADD
                    if( -(2^31-1)<= OUT <=2^31-1)
                        OverFlow =0;
                    else
                         OverFlow =1;
                    end
```

```
   4'b0100 :  begin
                    OUT = in_a - in_b;                          //SUB
                    if( -(2^31-1)<= OUT <=2^31-1)
                        OverFlow =0;
                    else
                         OverFlow =1;
                    end
    4'b0101 :       OUT = in_a & in_b;                          //AND
    4'b0110 :       OUT = in_a | in_b;                          //OR
    4'b0111 :       OUT = in_a ^ in_b;                          //XOR
    4'b1000 :       OUT = in_b<<in_a;                           //left  shift
    4'b1001 :       OUT = in_b>>in_a;                           //right shift

    4'b1011 :   begin
                    //<<??rotate right 1 bit
                    temp = {in_b,in_b};
                    temp = temp>>in_a;
                    OUT = {temp[31:0]};
                    end


    4'b1010 :   begin
                    //<<??rotate left  1 bit
                    temp = {in_b,in_b};
                    temp = temp<<in_a;
                    OUT = {temp[63:32]};
                    end


   endcase
  end
 end
 endmodule
```

Reference Code 3 (Register File)

```verilog
`timescale 1ns/10ps
module Mreg_4for32(OUT_1, OUT_2, Write, Read, Read_ADDR_1,
Read_ADDR_2, Write_ADDR, DIN, enable, clk, rst);

parameter ADSize = 13;                          //13 bits
parameter REGSize = 32;                         //32
parameter DASize = 32;                          //32bits

input clk;
input rst;
input enable;

input [ADSize-1:0]Read_ADDR_1;
input [ADSize-1:0]Read_ADDR_2;
input [ADSize-1:0]Write_ADDR;

input [DASize-1:0]DIN;

input       Write;
input       Read;

output [DASize-1:0]OUT_1;
output [DASize-1:0]OUT_2;

reg   [DASize-1:0]OUT_1;
reg   [DASize-1:0]OUT_2;

reg   [DASize-1:0]  Mreg[REGSize-1:0];

integer i;
```

```verilog
always@(posedge clk)begin
            if( rst )begin
            for(i=0 ; i<REGSize ; i=i+1)
            Mreg[i] = 0;
            end

            else
            begin
             if(enable)begin
               if( Write )
               Mreg[Write_ADDR] = DIN;
               else if( Read )begin
               OUT_1 = Mreg[Read_ADDR_1];
               OUT_2 = Mreg[Read_ADDR_2];
               end
             end
            end
end

endmodule
```