**Lab 2-20**

**Group Members: Colton Watson, Benjamin Leaptrot, Christian Giauque, Nathan Hummel.**

**Regfile**

The Regfile takes in inputs from the ALUBus and needs to set up a regEnable from the Bus to read in from the ALU or to just read in from the bus. We set up a reset input because it will reset the clock and the inputs so It can read in new inputs and help with the regEnable timings, as well we set in the 15 inputs from the ALUBus.

module regfile\_2D\_memory(ALUBus, r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,r15, regEnable, clk, reset);

input clk, reset;

input [15:0] ALUBus;

input [15:0] regEnable;

accommodate for the variables that will be coming out of the regfile, with the use of the r0, r1, r2, r3….

output [15:0] r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15;

reg [15:0] r [0:15]; // 2-dimensional memory

Below is the code that we used for generating a for loop that checks for needs of reset and using an always block with posedge of the clock so that we can reset the r[i] storage to zero. Then we have it set if the regEnable[i] becomes equal we are accepting from the ALUBus but then if neither of these are the case than we can accept the normal inputs from the bus, we need to set these as nonblocking so it does not block the next variable.

genvar i;

generate

for(i=0; i<=15;i=i+1)

begin:regfile

always @(posedge clk)

begin

if (reset == 1'b1)

r[i]<= 16'd0;

else

if(regEnable[i]==1'b1)

r[i] <= ALUBus;

else

r[i] <= r[i];

end

end

endgenerate

We need to assign all the r1, r2, r3… on the input from the bus so that we assign them to be sent to the mux then the ALU, so on so forth.

assign r1 = r[1];

assign r2 = r[2];

assign r3 = r[3];

assign r4 = r[4];

assign r5 = r[5];

assign r6 = r[6];

assign r7 = r[7];

assign r8 = r[8];

assign r9 = r[9];

assign r10 = r[10];

assign r11 = r[11];

assign r12 = r[12];

assign r13 = r[13];

assign r14 = r[14];

assign r15 = r[15];

**Mux**

For the mux is a switch that we are using from our registry file so that we can spit out a variable A that will upload to the ALU. Below are the inputs that are all coming in from the registry file.

input [15:0] r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15;

For the Mux we set up a c\_in to initialize the decimal and a case for each of the r0, r1, r2….. inputs that are coming into the mux from the registry file, so we use a A variable as an output to store all the input variables. Below is the code with using the wild card always block that will change when any of these variables become changed then using a case all the decimals are defined with the inputs r0, r1, r2…. Where A the variable then becomes stored as an output register to output too the ALU.

output reg [15:0] A;

always @\*

begin

case (C\_in)

4'd0: A = r0;

4'd1: A = r1;

4'd2: A = r2;

4'd3: A = r3;

4'd4: A = r4;

4'd5: A = r5;

4'd6: A = r6;

4'd7: A = r7;

4'd8: A = r8;

4'd9: A = r9;

4'd10: A = r10;

4'd11: A = r11;

4'd12: A = r12;

4'd13: A = r13;

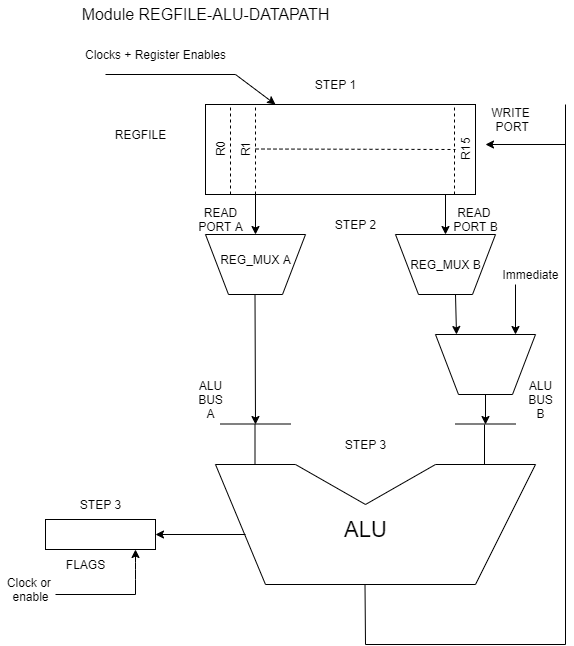
4'd14: A = r14;

4'd15: A = r15;

endcase

end

**Block Diagram**



Implemented Instructions:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Instruction | Opcode / Opcode Extension | Example | Description | Flags |
| ADD | 0000 0101 | ADD *src*, *dest*  ADD r0, r1  r1 = r1 + r0 | Integer addition | Carry  Overflow |
| ADDI | 0101 xxxx | ADDI *imm*, *dest*  ADD $2, r1  r1 = r1 + 2 | Integer addition with sign-extended immediate | Carry  Overflow |
| ADDU | 0000 0110 | ADDU *src*, *dest*  ADD r0, r1  r1 = r1 + r0 | Unsigned integer addition | None  (CR16 PSR flags are not affected by ADDU instruction) |
| ADDUI | 0110 xxxx | ADDUI *imm*, *dest*  ADDUI $2, r0  r0 = r0 + 2 | Unsigned integer addition with zero-extended immediate | None |
| ADDC | 0000 0111 | ADDC *src, dest*  ADDC r0, r1  r0 = r1 + r0 + C | Integer addition with carry | Carry  Overflow |
| ADDCI | 0111 xxxx | ADDCI *imm*, *dest*  ADDCI $2, r1  r1 = r1 + 2 + C | Integer addition with sign-extended immediate and carry | Carry  Overflow |
| SUB | 0000 1001 | SUB *src*, *dest*  SUB r0, r1  r1 = r1 - r0 | Integer subtraction | Carry  Zero  Overflow  Low  Negative |
| SUBI | 1001 xxxx | SUBI *imm*, *dest*  SUBI $2, r0  r0 = r0 - 2 | Integer subtraction with sign-extended immediate | Carry  Zero  Overflow  Low  Negative |
| SUBC | 0000 1010 | SUBC *src*, *dest*  SUBC r0, r1  r1 = r1 – (r0 + C) | Integer subtraction with carry | Carry  Zero  Overflow  Low  Negative |
| SUBCI | 1010 xxxx | SUBCI *imm*, *dest*  SUBCI $2, r1  r1 = r1 – (2 + C) | Integer subtraction with sign-extended immediate and carry | Carry  Zero  Overflow  Low  Negative |
| CMP | 0000 1011 | CMP *src1*, *src2*  CMP r0, r1 | Compare Integer.  PSR.Z = 1 if src1=scr2  PSR.N = 1 if src1<scr2 (signed)  PSR.L = 1 if scr1<scr2 (unsigned) | Zero  Negative  Low |
| CMPI | 1011 xxxx | CMPI *imm*, *src2*  CMPI $0, r0 | Compare Integer.  PSR.Z = 1 if imm=scr2  PSR.N = 1 if imm<scr2 (signed)  PSR.L = 1 if imm<scr2 (unsigned) | Zero  Negative  Low |
| AND | 0000 0001 | AND *src*, *dest*  AND r0, r1  r1 = r1 & r0 | Bitwise Logical AND | Zero |
| ANDI | 0001 xxxx | ANDI *imm*, *dest*  ANDI 0x55, r1  r1 = r1 & 0x55 | Bitwise Logical AND with zero-extended immediate | Zero |
| OR | 0000 0010 | OR *src*, *dest*  OR r0, r1  r1 = r1 | r0 | Bitwise Logical OR | None |
| ORI | 0010 xxxx | ORI *imm*, *dest*  ORI 0x55, r1  r1 = r1 | 0x55 | Bitwise Logical OR with zero-extended immediate | None |
| XOR | 0000 0011 | XOR *src*, *dest*  XOR r0, r1  r1 = r1 ^ r0 | Bitwise Logical XOR | None |
| XORI | 0011 xxxx | XORI *imm*, dest  AND 0x55, r1  r1 = r1 ^ 0x55 | Bitwise Logical XOR with zero-extended immediate | None |
| MOV | 0000 1101 | MOV *src*, *dest*  MOV r0, r1  r1 = r0 | Move | None |
| MOVI | 1101 xxxx | MOV *imm*, *dest*  MOV $7, r0  r0 = $7 | Move with zero-extended immediate | None |
| LSH | 1000 0100 | LSH *count*, *dest*  LSH r0, r1  r1 = r1 << r0 (r0 > 0)  r1 = r1 >> r0 (r0 < 0) | Logical Shift Integer  If count is positive, left shift, if count is negative, right shift | None |
| LSHI | 1000 000s | LSHI *imm*, *dest*  LSHI $1, r1  r1 = r1 << 1  LSHI $-1, r1  r1 = r1 >> 1 | Logical Shift Integer Immediate  s->0 right shift  s->1 left shift | None |
| ASHU | 1000 0110 | ASHU *count*, *dest*  ASHU r0, r1  r1 = r1 <<< r0 (r0 > 0)  r1 = r1 >>> r0 (r0 < 0) | Arithmetic Shift | None |
| ASHUI | 1000 001s | ASHUI *imm*, *dest*  ASHUI $1, r1  r1 = r1 <<< 1  LSH $-1, r1  r1 = r1 >>> 1 | Arithmetic Shift with immediate  s->0 right shift  s-> left shift | None |
| LUI | 1111 xxxx | LUI *imm*, *dest*  LUI 0x55, r0  r0 = 0x5500 | Load upper immediate | None |

Postponed Instructions

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Instruction | Opcode / Opcode Extension | Example | Description | Flags |
| LOAD | 0100 0000 |  | Load | None |
| STOR | 0100 0100 |  | Store | None |
| Bcond | 1100 xxxx |  | Conditional Branch | None |
| Jcond | 0100 1100 |  | Conditional Jump | None |
| JAL | 0100 1000 |  | Jump and Link |  |