**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;

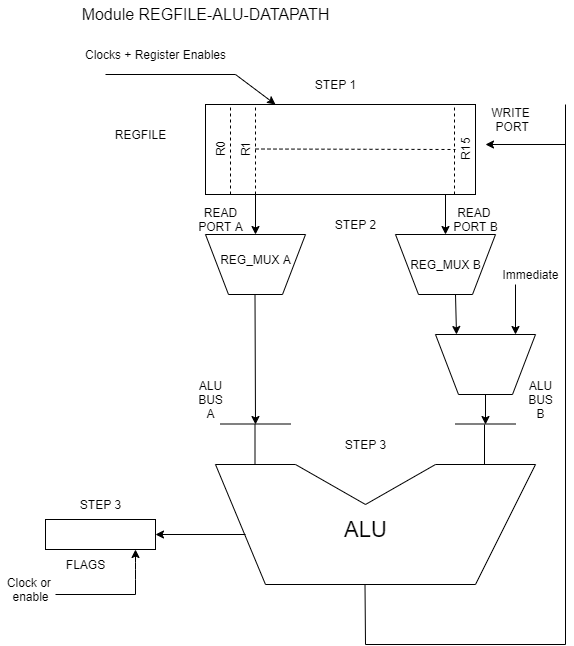
4'd14: A = r14;

4'd15: A = r15;

endcase

end

**Block Diagram**



**The ALU**

The arithmetic-logic unit currently supports most operations listed in the ISA document. MUL(I) have not been implemented since it is believed that the extra logic required for multiply instructions is not worth having when a combination of shift and addition operations also perform the operation in most situations. The ALU expects an eight-bit opcode, two sixteen-bit values for the inputs, and a sixteen-bit value for the output. For instructions using immediate values, such as ADDI, ADDUI, SUBI, etc., the low order bits of the opcode, the opcode extension, is instead interpreted as the high bits of the immediate, with the immediate value following the control flow of the diagram above.

The Flags register also is outputted as a five-bit value where the order from the most-significant bit to the least significant bit is the Carry Flag (C), the Low Flag (L), the Overflow Flag (F), the Zero Flag (Z), and the Negative Flag (N). This ordering was based on the order that was given in the Lab 1 instruction packet. The order of the bits can be rearranged if needed. Different Operations affect different flags, but the ISA requirements have been met. In addition to that, a few operations have been given control of extra flags. Instructions with modified flag control:

* ADDU(I) – The ISA document and the CR16 PSR mentions no flags being set.
* SUB(I/C/CI) – All subtraction operations adjust all flags they normally adjust and the flags CMP instructions sets since compare operations are subtractions with no write-back.
* AND – The AND operation affects the Zero flag if the operation of A&B is zero.

The table below lists all implemented instructions, their Opcode, an example in assembly with mathematical representation, a simple description of the operation, and the flags that the instruction modifies.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Instruction** | **Opcode / Opcode Extension** | **Example** | **Description** | **Flags that are affected** |
| 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, shift left. If count is negative, shift right | None |
| LSHI | 1000 000s | LSHI *imm*, *dest*  LSHI $1, r1  r1 = r1 << 1  LSHI $-1, r1  r1 = r1 >> 1 | Logical Shift Integer Immediate. Immediate range: [0,15]  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  If count is positive, shift left. If count is negative, shift right | None |
| ASHUI | 1000 001s | ASHUI *imm*, *dest*  ASHUI $1, r1  r1 = r1 <<< 1  LSH $-1, r1  r1 = r1 >>> 1 | Arithmetic Shift with immediate. Immediate range: [0,15]  s->0 right shift  s->1 left shift | None |
| LUI | 1111 xxxx | LUI *imm*, *dest*  LUI 0x55, r0  r0 = 0x5500 | Load upper immediate | None |

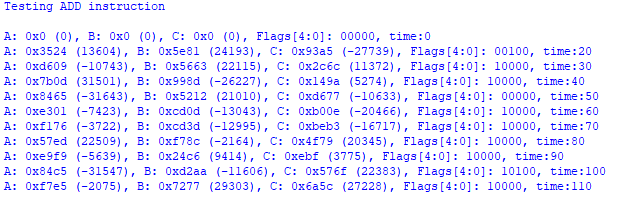
The following table is a list of currently postponed instructions until further documentation and modules have been supplied and implemented.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 |  |

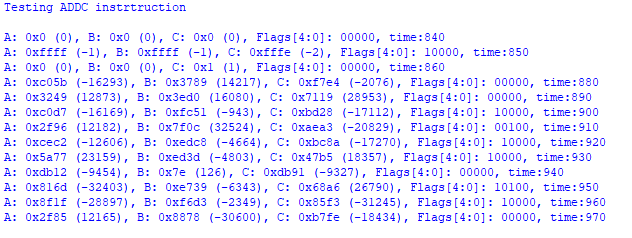
Testing of the ALU:

The testbench is structured to go through every operation, test a few simple examples, test a few operations to set flags, and perform random input operations. The testbench writes to the console the operation that is being tested, the value inside the A-input, the value inside the B-input, the value of the C-output, and the value of the Flags Register. The values for A, B, and C are given first in hexadecimal value and then its equivalent as a signed integer.

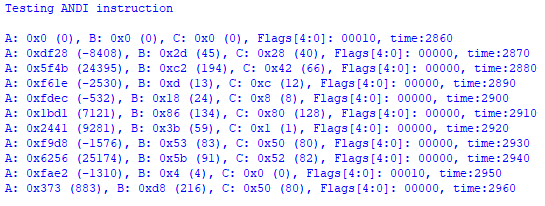
For the first example, a simple ADD operation. The first operation is adding zero and zero, the second operation sets the overflow flag, with the third setting the carry flag with a variety of random stimulus. Observe the second to last operation where it sets both the overflow flag and the carry flag.

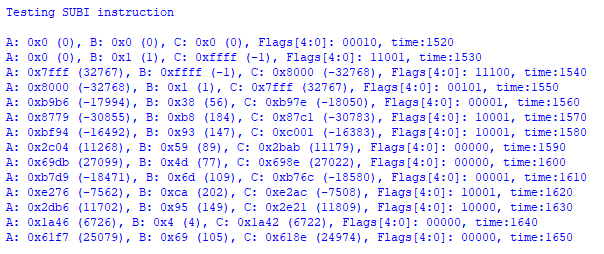


The second example demonstrates the ADDC operation which does addition with the Carry Flag from the previous operation. The second operation sets the carry flag, and then the third operation adds zero with zero and the carry bit, which results in an output of 1. This operation could be used to capture the value of the carry flag into one of the general-purpose registers.



The third example is of the ANDI instruction to demonstrate how immediate values work. Notice on the operations where the return value is zero, the zero flag is set.



The fourth example is of the SUBI instruction to show the additional flags being set. The first operation set the Zero Flag. The second operation set the Carry Flag, the Low Flag, and the Negative Flag. The third operation sets the Carry Flag, the Low Flag, and the Overflow Flag.

The fifth example is of the ASHUI instruction since it behaves differently from other operations where the least significant bit of the opcode determines the direction of the shift. A value of one is an arithmetic right shift, a value of zero is an arithmetic left shift. Something to be aware of, a negative value shifted to the right enough times will return a value of -1 (0xFFFF) instead of converging to zero.

