# Lab 2-20

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

## The Register File

The register file is implemented using the 2D-memory array design. It accepts a sixteen-bit input for the bus, a sixteen-bit value representing the register enable signals, as well as a basic clock and reset signal. The outputs are sixteen sixteen-bit general purpose registers. Each register is given its own register enable, where the value of the register enable is given to the register corresponding to that register enable position. For example, to set the value of r3 to be the value of the bus, set regEnable[3] to one. The benefit of this design is that setting values to registers is easy and allows us to set the value of multiple registers simultaneously to the bus with just setting every value register enable signal to one.

The convention we are using for resets is to set resets on positive edges of the clock, with high reset signals indicating a reset is to be initiate. When implemented on the board where button pushes bring the signal low, the top-level module will send an inverted signal to the lower-level modules.

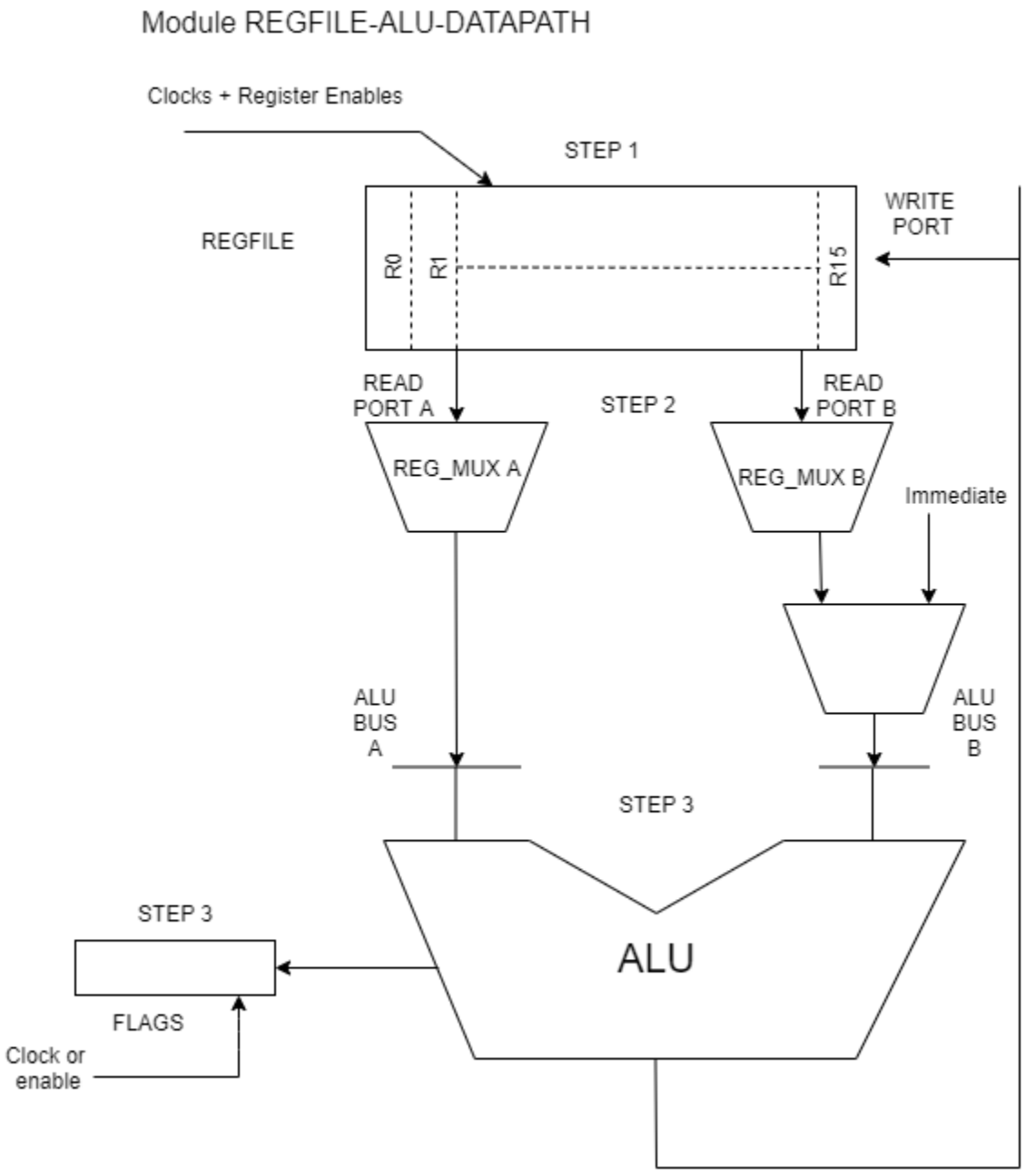
Testing of the register file:

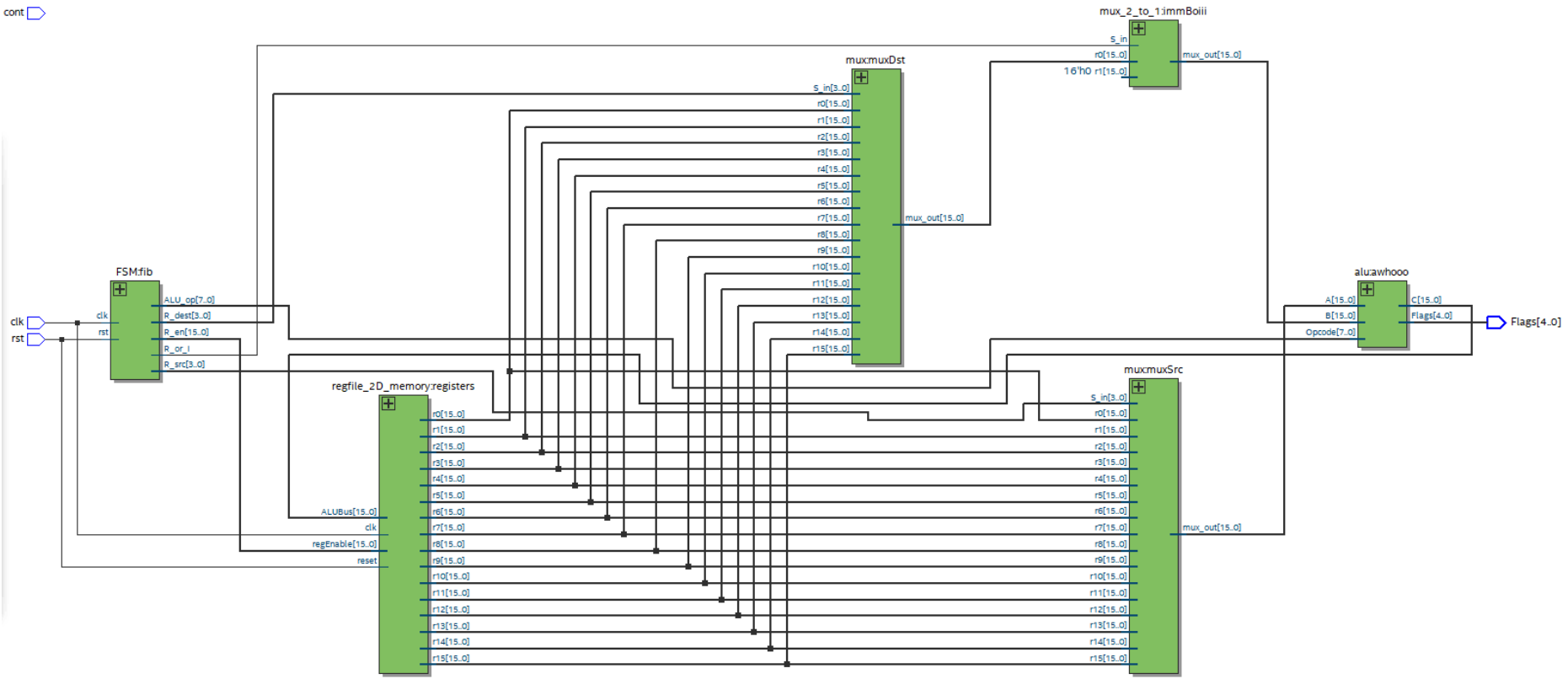
The testbench for the register file goes through the motions of setting each value from the bus, with various resets. The design is simple enough that exhaustive testing isn't necessary.

## The FSM

The FSM we have designed goes through various states of the Fibonacci sequence and stores the values into the registers. The generic assembly of the design is listed below. The register file outputs to two sixteen-to-one multiplexers which is then fed into the ALU. The output from the ALU is then wired to the bus. With this, simple programs were the operator acts as the decoder are possible, such as our Fibonacci generator. The design is then configured and looks like the second diagram.

# Register File & ALU Block Diagram




cont
clk
rst
FSM:fib
clk
rst
ALU_op[7..0]
R_dest[3..0]
R_en[15..0]
R_or_I
R_src[3..0]
mux:muxDst
S_in[3..0]
r0[15..0]
r1[15..0]
r2[15..0]
r3[15..0]
r4[15..0]
r5[15..0]
r6[15..0]
r7[15..0]
r8[15..0]
r9[15..0]
r10[15..0]
r11[15..0]
r12[15..0]
r13[15..0]
r14[15..0]
r15[15..0]
mux_out[15..0]
mux_2_to_1:immBoiii
S_in
r0[15..0]
16'h0 r1[15..0]
mux_out[15..0]
alu:awhooo
A[15..0]
B[15..0]
Opcode[7..0]
C[15..0]
Flags[4..0]
Flags[4..0]
regfile_2D_memory:registers
ALUBus[15..0]
clk
regEnable[15..0]
reset
r0[15..0]
r1[15..0]
r2[15..0]
r3[15..0]
r4[15..0]
r5[15..0]
r6[15..0]
r7[15..0]
r8[15..0]
r9[15..0]
r10[15..0]
r11[15..0]
r12[15..0]
r13[15..0]
r14[15..0]
r15[15..0]
mux:muxSrc
S_in[3..0]
r0[15..0]
r1[15..0]
r2[15..0]
r3[15..0]
r4[15..0]
r5[15..0]
r6[15..0]
r7[15..0]
r8[15..0]
r9[15..0]
r10[15..0]
r11[15..0]
r12[15..0]
r13[15..0]
r14[15..0]
r15[15..0]
mux_out[15..0]

# 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*<br>ADD r0, r1<br>r1 = r1 + r0 | Integer addition | Carry<br>Overflow |
| ADDI | 0101 xxxx | ADDI *imm*, *dest*<br>ADD $2, r1<br>r1 = r1 + 2 | Integer addition with sign-extended immediate | Carry<br>Overflow |
| ADDU | 0000 0110 | ADDU *src*, *dest*<br>ADD r0, r1<br>r1 = r1 + r0 | Unsigned integer addition | None<br>(CR16 PSR flags are not affected by ADDU instruction) |
| ADDUI | 0110 xxxx | ADDUI *imm*, *dest*<br>ADDUI $2, r0<br>r0 = r0 + 2 | Unsigned integer addition with zero-extended immediate | None |

| | | | | |
|---|---|---|---|---|
| ADDC | 0000 0111 | ADDC *src, dest*<br>ADDC r0, r1<br>r0 = r1 + r0 + C | Integer addition with carry | Carry<br>Overflow |
| ADDCI | 0111 xxxx | ADDCI *imm*, *dest*<br>ADDCI $2, r1<br>r1 = r1 + 2 + C | Integer addition with sign-extended immediate and carry | Carry<br>Overflow |
| SUB | 0000 1001 | SUB *src*, *dest*<br>SUB r0, r1<br>r1 = r1 - r0 | Integer subtraction | Carry<br>Zero<br>Overflow<br>Low<br>Negative |
| SUBI | 1001 xxxx | SUBI *imm*, *dest*<br>SUBI $2, r0<br>r0 = r0 - 2 | Integer subtraction with sign-extended immediate | Carry<br>Zero<br>Overflow<br>Low<br>Negative |
| SUBC | 0000 1010 | SUBC *src*, *dest*<br>SUBC r0, r1<br>r1 = r1 – (r0 + C) | Integer subtraction with carry | Carry<br>Zero<br>Overflow<br>Low<br>Negative |
| SUBCI | 1010 xxxx | SUBCI *imm*, *dest*<br>SUBCI $2, r1<br>r1 = r1 – (2 + C) | Integer subtraction with sign-extended immediate and carry | Carry<br>Zero<br>Overflow<br>Low<br>Negative |
| CMP | 0000 1011 | CMP *src1*, *src2*<br>CMP r0, r1 | Compare Integer.<br>PSR.Z = 1 if src1=scr2<br>PSR.N = 1 if src1<scr2 (signed)<br>PSR.L = 1 if scr1<scr2 (unsigned) | Zero<br>Negative<br>Low |
| CMPI | 1011 xxxx | CMPI *imm*, *src2*<br>CMPI $0, r0 | Compare Integer.<br>PSR.Z = 1 if imm=scr2<br>PSR.N = 1 if imm<scr2 (signed)<br>PSR.L = 1 if imm<scr2 (unsigned) | Zero<br>Negative<br>Low |
| AND | 0000 0001 | AND *src*, *dest*<br>AND r0, r1<br>r1 = r1 & r0 | Bitwise Logical AND | Zero |
| ANDI | 0001 xxxx | ANDI *imm*, *dest*<br>ANDI 0x55, r1<br>r1 = r1 & 0x55 | Bitwise Logical AND with zero-extended immediate | Zero |

| | | | | |
|---|---|---|---|---|
| OR | 0000 0010 | OR *src*, *dest*<br>OR r0, r1<br>r1 = r1 \| r0 | Bitwise Logical OR | None |
| ORI | 0010 xxxx | ORI *imm*, *dest*<br>ORI 0x55, r1<br>r1 = r1 \| 0x55 | Bitwise Logical OR with zero-extended immediate | None |
| XOR | 0000 0011 | XOR *src*, *dest*<br>XOR r0, r1<br>r1 = r1 ^ r0 | Bitwise Logical XOR | None |
| XORI | 0011 xxxx | XORI *imm*, dest<br>AND 0x55, r1<br>r1 = r1 ^ 0x55 | Bitwise Logical XOR with zero-extended immediate | None |
| MOV | 0000 1101 | MOV *src*, *dest*<br>MOV r0, r1<br>r1 = r0 | Move | None |
| MOVI | 1101 xxxx | MOV *imm*, *dest*<br>MOV $7, r0<br>r0 = $7 | Move with zero-extended immediate | None |
| LSH | 1000 0100 | LSH *count*, *dest*<br>LSH r0, r1<br>r1 = r1 << r0 (r0 > 0)<br>r1 = r1 >> r0 (r0 < 0) | Logical Shift Integer<br><br>If count is positive, shift left. If count is negative, shift right | None |
| LSHI | 1000 000s | LSHI *imm*, *dest*<br><br>LSHI $1, r1<br>r1 = r1 << 1<br><br>LSHI $-1, r1<br>r1 = r1 >> 1 | Logical Shift Integer Immediate. Immediate range: [0,15]<br><br>s->0 right shift<br>s->1 left shift | None |
| ASHU | 1000 0110 | ASHU *count*, *dest*<br>ASHU r0, r1<br>r1 = r1 <<< r0 (r0 > 0)<br>r1 = r1 >>> r0 (r0 < 0) | Arithmetic Shift<br><br>If count is positive, shift left. If count is negative, shift right | None |
| ASHUI | 1000 001s | ASHUI *imm*, *dest*<br>ASHUI $1, r1<br>r1 = r1 <<< 1<br><br>LSH $-1, r1<br>r1 = r1 >>> 1 | Arithmetic Shift with immediate. Immediate range: [0,15]<br><br>s->0 right shift<br>s->1 left shift | None |

| | | | | |
|---|---|---|---|---|
| LUI | 1111 xxxx | LUI *imm*, *dest*<br>LUI 0x55, r0<br>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.

```
Testing ADD instruction

A: 0x0 (0), B: 0x0 (0), C: 0x0 (0), Flags[4:0]: 00000, time:0
A: 0x3524 (13604), B: 0x5e81 (24193), C: 0x93a5 (-27739), Flags[4:0]: 00100, time:20
A: 0xd609 (-10743), B: 0x5663 (22115), C: 0x2c6c (11372), Flags[4:0]: 10000, time:30
A: 0x7b0d (31501), B: 0x998d (-26227), C: 0x149a (5274), Flags[4:0]: 10000, time:40
A: 0x8465 (-31643), B: 0x5212 (21010), C: 0xd677 (-10633), Flags[4:0]: 00000, time:50
A: 0xe301 (-7423), B: 0xcd0d (-13043), C: 0xb00e (-20466), Flags[4:0]: 10000, time:60
A: 0xf176 (-3722), B: 0xcd3d (-12995), C: 0xbeb3 (-16717), Flags[4:0]: 10000, time:70
A: 0x57ed (22509), B: 0xf78c (-2164), C: 0x4f79 (20345), Flags[4:0]: 10000, time:80
A: 0xe9f9 (-5639), B: 0x24c6 (9414), C: 0xebf (3775), Flags[4:0]: 10000, time:90
A: 0x84c5 (-31547), B: 0xd2aa (-11606), C: 0x576f (22383), Flags[4:0]: 10100, time:100
A: 0xf7e5 (-2075), B: 0x7277 (29303), C: 0x6a5c (27228), Flags[4:0]: 10000, time:110
```

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.

```
Testing ADDC instrtruction

A: 0x0 (0), B: 0x0 (0), C: 0x0 (0), Flags[4:0]: 00000, time:840
A: 0xffff (-1), B: 0xffff (-1), C: 0xfffe (-2), Flags[4:0]: 10000, time:850
A: 0x0 (0), B: 0x0 (0), C: 0x1 (1), Flags[4:0]: 00000, time:860
A: 0xc05b (-16293), B: 0x3789 (14217), C: 0xf7e4 (-2076), Flags[4:0]: 00000, time:880
A: 0x3249 (12873), B: 0x3ed0 (16080), C: 0x7119 (28953), Flags[4:0]: 00000, time:890
A: 0xc0d7 (-16169), B: 0xfc51 (-943), C: 0xbd28 (-17112), Flags[4:0]: 10000, time:900
A: 0x2f96 (12182), B: 0x7f0c (32524), C: 0xaea3 (-20829), Flags[4:0]: 00100, time:910
A: 0xcec2 (-12606), B: 0xedc8 (-4664), C: 0xbc8a (-17270), Flags[4:0]: 10000, time:920
A: 0x5a77 (23159), B: 0xed3d (-4803), C: 0x47b5 (18357), Flags[4:0]: 10000, time:930
A: 0xdb12 (-9454), B: 0x7e (126), C: 0xdb91 (-9327), Flags[4:0]: 00000, time:940
A: 0x816d (-32403), B: 0xe739 (-6343), C: 0x68a6 (26790), Flags[4:0]: 10100, time:950
A: 0x8f1f (-28897), B: 0xf6d3 (-2349), C: 0x85f3 (-31245), Flags[4:0]: 10000, time:960
A: 0x2f85 (12165), B: 0x8878 (-30600), C: 0xb7fe (-18434), Flags[4:0]: 00000, time:970
```

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.

```
Testing ANDI instruction

A: 0x0 (0), B: 0x0 (0), C: 0x0 (0), Flags[4:0]: 00010, time:2860
A: 0xdf28 (-8408), B: 0x2d (45), C: 0x28 (40), Flags[4:0]: 00000, time:2870
A: 0x5f4b (24395), B: 0xc2 (194), C: 0x42 (66), Flags[4:0]: 00000, time:2880
A: 0xf61e (-2530), B: 0xd (13), C: 0xc (12), Flags[4:0]: 00000, time:2890
A: 0xfdec (-532), B: 0x18 (24), C: 0x8 (8), Flags[4:0]: 00000, time:2900
A: 0x1bd1 (7121), B: 0x86 (134), C: 0x80 (128), Flags[4:0]: 00000, time:2910
A: 0x2441 (9281), B: 0x3b (59), C: 0x1 (1), Flags[4:0]: 00000, time:2920
A: 0xf9d8 (-1576), B: 0x53 (83), C: 0x50 (80), Flags[4:0]: 00000, time:2930
A: 0x6256 (25174), B: 0x5b (91), C: 0x52 (82), Flags[4:0]: 00000, time:2940
A: 0xfae2 (-1310), B: 0x4 (4), C: 0x0 (0), Flags[4:0]: 00010, time:2950
A: 0x373 (883), B: 0xd8 (216), C: 0x50 (80), Flags[4:0]: 00000, time:2960
```

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.

```
Testing SUBI instruction

A: 0x0 (0), B: 0x0 (0), C: 0x0 (0), Flags[4:0]: 00010, time:1520
A: 0x0 (0), B: 0x1 (1), C: 0xffff (-1), Flags[4:0]: 11001, time:1530
A: 0x7fff (32767), B: 0xffff (-1), C: 0x8000 (-32768), Flags[4:0]: 11100, time:1540
A: 0x8000 (-32768), B: 0x1 (1), C: 0x7fff (32767), Flags[4:0]: 00101, time:1550
A: 0xb9b6 (-17994), B: 0x38 (56), C: 0xb97e (-18050), Flags[4:0]: 00001, time:1560
A: 0x8779 (-30855), B: 0xb8 (184), C: 0x87c1 (-30783), Flags[4:0]: 10001, time:1570
A: 0xbf94 (-16492), B: 0x93 (147), C: 0xc001 (-16383), Flags[4:0]: 10001, time:1580
A: 0x2c04 (11268), B: 0x59 (89), C: 0x2bab (11179), Flags[4:0]: 00000, time:1590
A: 0x69db (27099), B: 0x4d (77), C: 0x698e (27022), Flags[4:0]: 00000, time:1600
A: 0xb7d9 (-18471), B: 0x6d (109), C: 0xb76c (-18580), Flags[4:0]: 00001, time:1610
A: 0xe276 (-7562), B: 0xca (202), C: 0xe2ac (-7508), Flags[4:0]: 10001, time:1620
A: 0x2db6 (11702), B: 0x95 (149), C: 0x2e21 (11809), Flags[4:0]: 10000, time:1630
A: 0x1a46 (6726), B: 0x4 (4), C: 0x1a42 (6722), Flags[4:0]: 00000, time:1640
A: 0x61f7 (25079), B: 0x69 (105), C: 0x618e (24974), Flags[4:0]: 00000, time:1650
```

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.

```
Testing ASHUI instruction

A: 0x0 (0), B: 0x0 (0), C: 0x0 (0), Flags[4:0]: 00000, time:5360
A: 0x8000 (-32768), B: 0x3 (3), C: 0x0 (0), Flags[4:0]: 00000, time:5370
A: 0x8000 (-32768), B: 0x3 (3), C: 0xf000 (-4096), Flags[4:0]: 00000, time:5380
A: 0xee0e (-4594), B: 0x1 (1), C: 0xf707 (-2297), Flags[4:0]: 00000, time:5390
Shift: 1
A: 0xd1af (-11857), B: 0x1 (1), C: 0xe8d7 (-5929), Flags[4:0]: 00000, time:5400
Shift: 1
A: 0x9f70 (-24720), B: 0x1 (1), C: 0xcfb8 (-12360), Flags[4:0]: 00000, time:5410
Shift: 0
A: 0x3dbd (15805), B: 0x1 (1), C: 0x7b7a (31610), Flags[4:0]: 00000, time:5420
Shift: 0
A: 0x5e50 (24144), B: 0x1 (1), C: 0xbca0 (-17248), Flags[4:0]: 00000, time:5430
Shift: 1
A: 0x8478 (-31624), B: 0x3 (3), C: 0xf08f (-3953), Flags[4:0]: 00000, time:5440
Shift: 0
A: 0x9b23 (-25821), B: 0x1 (1), C: 0x3646 (13894), Flags[4:0]: 00000, time:5450
Shift: 0
A: 0x46bb (18107), B: 0x1 (1), C: 0x8d76 (-29322), Flags[4:0]: 00000, time:5460
Shift: 1
A: 0x30c (780), B: 0x2 (2), C: 0xc3 (195), Flags[4:0]: 00000, time:5470
Shift: 0
A: 0xdc59 (-9127), B: 0x2 (2), C: 0x7164 (29028), Flags[4:0]: 00000, time:5480
Shift: 1
A: 0xbff4 (-16396), B: 0x3 (3), C: 0xf7fe (-2050), Flags[4:0]: 00000, time:5490
Shift: 0
A: 0x7a39 (31289), B: 0x2 (2), C: 0xe8e4 (-5916), Flags[4:0]: 00000, time:5500
```