Lab 1 Open Ended Portion

Group Members:
Cristian Vasiliu (SID: 3034012141)
Diether Delosreyes (SID: 3033633776)

Definition of Instruction

In this open ended portion of the lab, we implemented the common instruction of popcount(). What popcount does is that it counts every set bit in a value, so for instance, take the hexadecimal number 0xB, three bits are set (0xB == 0b1011), thus popcount returns 3.

However, in our version of popcount, to meet the specs of doing a memory operation (ld/st) and doing a microbranch that requires the zero signal from the ALU, we define popcount as follows:

popcnt rd, rs1, imm

where popcnt would take the value of a word in memory at address R[rs1] + imm, then put the number of set bits in that word in R[rd]. In summary, the operation is this:

R[rd] =: popcount(Mem[R[rs1] + imm])

In C code, it would look like this:

```
unsigned int popcount(unsigned int x) {

	unsigned int n; for (n = 0; x != 0; n++) {
		x &= x – 1;
	// Clear least significant 1 bit
	}
	return n;
}
```

In RISC-V code, it would look like this (after a few optimizations, and assuming x2 is where we want to output our result and that x1 contains the input):

```
popcnt:
	add x2, x0, x0		// clear the return register
loop:
	beq x1, x0, fetch	// check if the result is 0 if it is fetch the next instruction
	addi x6, x1, -1		// clearing the least significant bit
	and x1, x6, x1		//		“ “
	addi x2, x2, 1		// increment the counter
	jal x0, loop		// jump to the loop

fetch:
```

Microcode

We chose to encode our instruction using the custom0 encoding as follows:

```
def POPCNT          = BitPat("b???????????????????000?????0001011")
```

The microcode instructions we implemented is this (see last page for full implementation of signals):

```
A       <- R[rs1]
B       <- Sext(Imm12)
MA    <- A + B
A, B  <- MEM
R[rd] <- A - B
If A == 0: microbranch fetch
B      <- A - 1
B      <- A & B
A      <- R[rd]
R[rd] <- A + 1
A      <- B
microbranch loop
```

Testing and Simulation

Here are the test cases that we tested for:

- General random numbers as input
- All bits in the word set to 1
- All bits in the word set to 0
- Using different source and destination registers in the word
- Checking that R[rs1] doesn't change after we execute the instruction
- Checking that we access the correct address in memory with a different immediate

We defined our test for the newly implemented instruction to be:

```
#define TEST_POPCNT_A0( testnum, result, data ) \
        TEST_CASE( testnum, a0, result, \
        la a0, data; \
        .word 0x0005050B)

#define TEST_POPCNT_A0_IMMNEG4( testnum, result, data ) \
        TEST_CASE( testnum, a0, result, \
        la a0, data; \
        .word 0xFFC5050B)

#define TEST_POPCNT_A0_IMM4( testnum, result, data ) \
        TEST_CASE( testnum, a0, result, \
        la a0, data; \
        .word 0x0045050B)

#define TEST_POPCNT_A0_IMM8( testnum, result, data ) \
        TEST_CASE( testnum, a0, result, \
        la a0, data; \
        .word 0x0085050B)

#define TEST_POPCNT_A1( testnum, result, data ) \
        TEST_CASE( testnum, a0, result, \
        la a1, data; \
        .word 0x0005850B)

#define TEST_POPCNT_A1_IMMNEG4( testnum, result, data ) \
        TEST_CASE( testnum, a0, result, \
        la a1, data; \
        .word 0xFFC5850B)

#define TEST_POPCNT_A1_IMM4( testnum, result, data ) \
        TEST_CASE( testnum, a0, result, \
        la a1, data; \
        .word 0x0045850B)

#define TEST_POPCNT_A1_IMM8( testnum, result, data ) \
        TEST_CASE( testnum, a0, result, \
        la a1, data; \
        .word 0x0085850B)
```

```
RVTEST_RV64U
RVTEST_CODE_BEGIN

//FIXME: Add your tests here
//See the tests in movn.S for examples

        TEST_POPCNT_A0( 1, 4, tdat1);
        TEST_POPCNT_A0( 2, 3, tdat2);
        TEST_POPCNT_A0( 3, 0, tdat3);
        TEST_POPCNT_A0( 4, 32, tdat4);

        TEST_POPCNT_A0_IMMNEG4( 5, 4, tdat2);
        TEST_POPCNT_A0_IMMNEG4( 6, 3, tdat3);
        TEST_POPCNT_A0_IMM4( 7, 3, tdat1);
        TEST_POPCNT_A0_IMM8( 8, 32, tdat2);

        TEST_POPCNT_A1( 9, 4, tdat1);
        TEST_POPCNT_A1( 10, 3, tdat2);
        TEST_POPCNT_A1( 11, 0, tdat3);
        TEST_POPCNT_A1( 12, 32, tdat4);


        TEST_POPCNT_A1_IMMNEG4( 13, 4, tdat2);
        TEST_POPCNT_A1_IMMNEG4( 14, 3, tdat3);
        TEST_POPCNT_A1_IMM4( 15, 3, tdat1);
        TEST_POPCNT_A1_IMM8( 16, 32, tdat2);

  TEST_PASSFAIL

RVTEST_CODE_END

  .data
RVTEST_DATA_BEGIN

  TEST_DATA
tdat:
tdat1:  .word 0x000001A4
tdat2:  .word 0x00000045
tdat3:  .word 0x00000000
tdat4:  .word 0xFFFFFFFF

RVTEST_DATA_END
```

Bugs

Some bugs that these tests caught include:

- Address not being calculated right
- Wrong value being given by memory

Microcode implementation in instructions.scala:

```
/* POPCNT          */
/* A  <- R[rs1]    */,Label("POPCNT"),Signals(Cat(CSR.N, LDIR_0, RS_RS1, RWR_0, REN_1, LDA_1, LDB_0, ALU_X             , AEN_0, LDMA_0, MWR_0, MEN_0, MT_X , IS_X , IEN_0, UBR_N) , "X")
/* B  <- Sext(Imm12)*/,                Signals(Cat(CSR.N, LDIR_0, RS_X  , RWR_0, REN_0, LDA_0, LDB_1, ALU_X    , AEN_0, LDMA_0, MWR_0, MEN_0, MT_X , IS_I , IEN_1, UBR_N) , "X")
/* MA <- A + B     */,                Signals(Cat(CSR.N, LDIR_0, RS_X  , RWR_0, REN_0, LDA_0, LDB_0, ALU_ADD          , AEN_1, LDMA_1, MWR_0, MEN_0, MT_X , IS_X , IEN_0, UBR_N) , "X")
/* A, B <- MEM     */,                Signals(Cat(CSR.N, LDIR_0, RS_X  , RWR_0, REN_0, LDA_1, LDB_1, ALU_X    , AEN_0, LDMA_0, MWR_0, MEN_1, MT_W , IS_X , IEN_0, UBR_S) , "X")
/* R[rd] <- A - B  */,                Signals(Cat(CSR.N, LDIR_0, RS_RD , RWR_1, REN_0, LDA_0, LDB_0, ALU_SUB          , AEN_1, LDMA_0, MWR_0, MEN_0, MT_X , IS_X , IEN_0, UBR_N) , "X")
/* if A = 0: jmp fetch  */,Label("LOOP"), Signals(Cat(CSR.N,  LDIR_0, RS_X  , RWR_0, REN_0, LDA_0, LDB_0, ALU_COPY_A , AEN_1, LDMA_0, MWR_0, MEN_0, MT_X , IS_X , IEN_0, UBR_EZ), "FETCH")
/* B <- A - 1      */,                Signals(Cat(CSR.N, LDIR_0, RS_X  , RWR_0, REN_0, LDA_0, LDB_1, ALU_DEC_A_1, AEN_1, LDMA_0, MWR_0, MEN_0, MT_X , IS_X , IEN_0, UBR_N) , "X")
/* B <- B & A      */,                Signals(Cat(CSR.N, LDIR_0, RS_X  , RWR_0, REN_0, LDA_0, LDB_1, ALU_AND          , AEN_1, LDMA_0, MWR_0, MEN_0, MT_X , IS_X , IEN_0, UBR_N) , "X")
/* A <- R[rd]      */,                Signals(Cat(CSR.N, LDIR_0, RS_RD , RWR_0, REN_1, LDA_1, LDB_0, ALU_X            , AEN_0, LDMA_0, MWR_0, MEN_0, MT_X , IS_X , IEN_0, UBR_S) , "X")
/* R[rd] <- A + 1  */,                Signals(Cat(CSR.N, LDIR_0, RS_RD , RWR_1, REN_0, LDA_0, LDB_0, ALU_INC_A_1, AEN_1, LDMA_0, MWR_0, MEN_0, MT_X , IS_X , IEN_0, UBR_N) , "X")
/* A <- B          */,                Signals(Cat(CSR.N, LDIR_0, RS_X  , RWR_0, REN_0, LDA_1, LDB_0, ALU_COPY_B , AEN_1, LDMA_0, MWR_0, MEN_0, MT_X , IS_X , IEN_0, UBR_N) , "X")
/* jmp loop        */,                Signals(Cat(CSR.N, LDIR_0, RS_X  , RWR_0, REN_0, LDA_0, LDB_0, ALU_X    , AEN_0, LDMA_0, MWR_0, MEN_0, MT_X , IS_X , IEN_0, UBR_J) , "LOOP")
```