IBCM 2.2
Principles of Operation

The Itty Bitty Computing Machine 2.2 is a very, very simple computer. In fact, it's so simple that no one in their right mind would build it using today's technology. Nevertheless, except for limits on problem size, any computation that can be performed on the most modern, sophisticated computer can also be performed on the IBCM 2.2. Its main virtue is that it can be taught quickly and will provide context for talking about more recent architectures.

CPU Characteristics

- single accumulator,
- 16-bit, fixed point, 2's complement arithmetic
- the program counter, PC, generally points to the next instruction

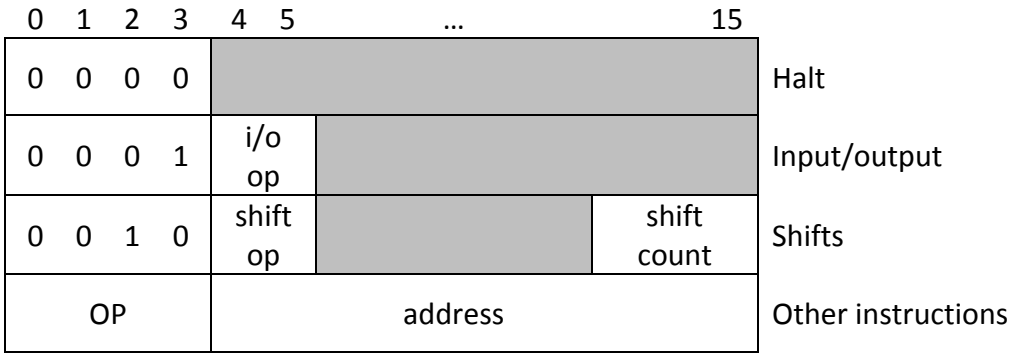
Memory Characteristics

- 4096 16-bit "words" in big-endian format

I/O Characteristics

- move numbers or characters from the accumulator to the screen, or from the keyboard to the accumulator.

Instruction Format



The IBCM equivalent of "statements" in a higher level language are very simple instructions. Each instruction is encoded in a 16-bit word in one of the formats shown in above (shaded areas in the figure represent portions of the word whose value doesn't matter). So, for example, a word whose leftmost four bits are zero is an instruction to halt the computer.

Words with leftmos bits being 0001 and 0010 are input/output instructions and shift instructions respectively; we'll come back to those in a moment. All other bit combinations in the left-most four bits either specify arithmetic or control – sort of like assignment statements and goto's in a high level language. These four bits are called the "op" field of the instruction; the value of this field is often called the "opcode". The "address" portion of the instruction generally specifies an address in memory where an operand (variable) will be found.

There are 13 IBCM operations of this form – eight that manipulate data and five that perform control. The data manipulation operations all involve the "accumulator" and data from a memory location specified by the address portion of the instruction. The result of most data manipulation operations is recorded in the accumulator. Thus, the "add" instruction forms the arithmetic sum of the present contents of the accumulator with the contents of the memory location specified by "address" and puts the result back into the accumulator. Thus, it's like the very primitive assignment statement "`accumulator = accumulator + memory [ address ]`".

The control instructions determine the next instruction to be executed. The "jump" instruction, for example, causes the next instruction executed to be the one at the location contained in its address field. If you think of the address of a memory cell like a label in a high level language, then jump is just "goto address".

Two of the control instructions are conditional; they either cause a change in the control flow or not, depending on the nature of the contents of the accumulator. The simplest of the control instructions is "nop"; it does nothing.

The following table describes the function of each of the 16 IBCM instructions. Where possible, both English and programming language-like explanations are given for each instruction. In the latter, "a" is the accumulator, "addr" is the values of the address portion of the instruction, and "mem[]" is memory.

op	name	HLL-like meaning	English explanation
3_{16}	load	a:= mem[addr]	load accumulator from memory
4_{16}	store	mem[addr] := a	store accumulator into memory
5_{16}	add	a:= a + mem[addr]	add memory to accumulator
6_{16}	sub	a:= a - mem[addr]	subtract memory from accumulator
7_{16}	and	a:= a & mem[addr]	logical 'and' memory into accumulator
8_{16}	or	a:= a | mem[addr]	logical 'or' memory into accumulator
9_{16}	xor	a:= a xor mem[addr]	logical 'xor' memory into accumulator
A_{16}	not	a := ~a	logical complement of accumulator
B_{16}	nop		do nothing (no operation)
C_{16}	jmp	goto addr	jump to 'addr'
D_{16}	jmpe	if a = 0 goto addr	jump to 'addr' if accumulator equals zero
E_{16}	jmpl	if a < 0 goto addr	jump to 'addr' if accumulator less than zero
F_{16}	brl	a:= PC; goto addr	jump (branch) to 'addr'; set accumulator to the value of the PC just before the jump (i.e., to the address following the brl). This instruction is often called "branch and link".

A few words of additional explanation are necessary for some of these operations, all of which are fairly standard for most computers.

Arithmetic operations may "overflow" or "underflow"; that is, the magnitude of the result may be larger than can be represented in 16 bits. The programmer is responsible for ensuring that this doesn't happen. The result of such an overflow or underflow is undefined (meaning the simulator may not work properly or predictably).

The "logical operations", AND, OR, XOR and NOT perform bit-wise operations on the operands. The operations are defined by the following tables:

and	0	1
0	0	0
1	0	1

or	0	1
0	0	1
1	1	1

xor	0	1
0	0	1
1	1	0

not	
0	1
1	0

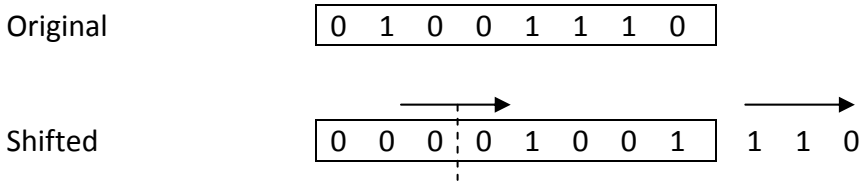
The branch and link instruction is used for subroutine calls, as will be discussed in class. Note that the address stored in the accumulator is the address of the instruction *after* the brl instruction. Thus, if the brl is at the last location of memory (i.e. memory address 4095), then the address inserted into the accumulator (4096) is an invalid memory address.

Now let's return to the two classes of instructions that we skipped earlier, input/output and shifts.

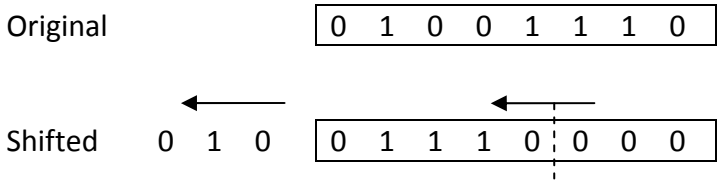
The input/output (or 'io') instructions move data between the accumulator and the computer ‘devices’ – the keyboard and screen. Data can be moved either as hexadecimal numbers or as a ascii character (in the later case, only the bottom 8 bits of the accumulator are involved). The four possibilities (in/out, hex/ascii) are specified by the bits 5 and 6 of the instruction word as follows:

bit 4	bit 5	operation
0	0	read a hexadecimal word (four digits) into the accumulator
0	1	read an ascii character into the accumulator bits 8-15
1	0	write a hexadecimal word (four digits) from the accumulator
1	1	write an ascii character from the accumulator bits 8-15

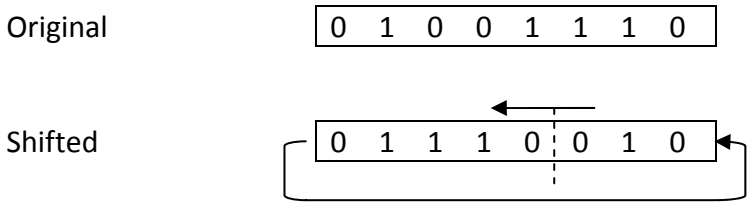
Shifting and rotating are common in computers. Shifting means moving data to the right or left; rotating is much the same thing except that bits that "fall off" one end are reinserted at the other. Diagrammatically, for example, a "right shift" of "n positions" looks like the following when *n* (the sift count) is 3:



Each bit is moved *n* positions to the right (three in this case). Alternatively, it is moved one bit to the right *n* times. This causes the original *n* rightmost bits to fall off the right end and *n* new (zero) bits to be inserted at the left. A "left shift" is much the same, except that bits move to the left:



As noted above, rotates are like shifts except that the bits don't fall off the end but are reinserted at the other end. Below, for example is a left rotation (we'll leave right rotations to your imagination).



Note that the shift instructions uses bits 5-6 to specify the shift operation (left/right, shift/rotate) as specified by the following table:

bit 4	bit 5	operation
0	0	shift left
0	1	shift right
1	0	rotate left
1	1	rotate right

In addition, bits 12-15 of the shift instructions specify the "shift count" - that is, the number of bits positions that the data is to be shifted (or rotated).

A Note on Using Hexadecimal Notation

IBCM is a binary computer. Internally all operations are performed on 16-bit binary values. However, because it's so tedious and error-prone for humans to write or read 16-bit quantities, all of IBCM's input/output is done either as ascii characters or 4-digit hexadecimal numbers. Remember that these are just external shorthands for the internal 16-bit values!

Running the Simulator

Currently, there are two simulators available to program in IBCM. The original simulator, used prior to the fall 2008 semester, is an application written in Visual Basic, and only runs on Windows computers. The new simulator is a PHP/Javascript application that runs in any modern web browser. Documentation for how to use each simulator is available at the same website where the simulators are found.