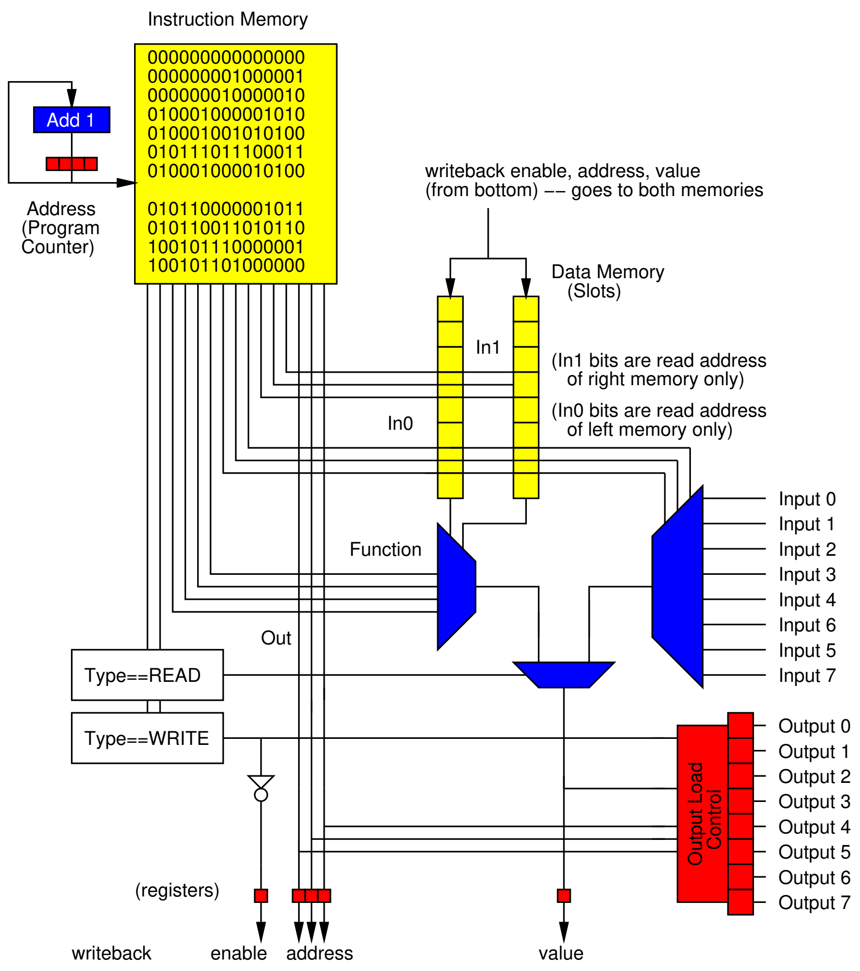
1. Continuing with our single-gate processor from Lecture 15:



Continue to simulate the processor on the next 4 instructions, by completing the memory contents after each instruction executes.
Memory is updated at the beginning of the cycle following the operation, so reflect the memory update for line i (e.g. 4) in line $i+1$ (e.g. 5) operating from the memory contents shown for line i (e.g. 4). (Because of this, you are not executing instruction 8, but you should show what value the memory holds when instruction 8 begins to executes in the final row.)

Inst..	Instruction Fields					Memory Contents							
Addr	Type	Function	In0	In1	Out	0	1	2	3	4	5	6	7
4	GATE	AND	1	2	4	1	1	0	1	0	0	0	0
5	GATE	OR	3	4	3								
6	GATE	AND	0	2	4								
7	GATE	OR	3	4	5								
8													

(continued on back)

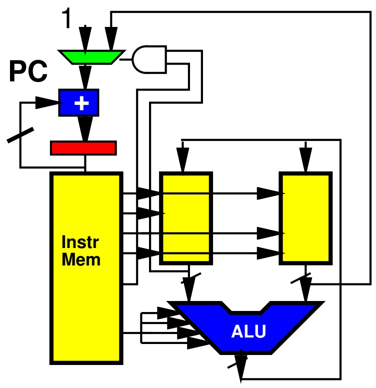
2. Consider word-wide operators working on an 8b-wide word:

   - An 8b add behaves as in C or Java (adding the two 8b numbers, giving an 8b result – modulo 256 if it overflows)
   - bit-wise logical operators (like AND, OR) perform the associated logical operators on the pair of bits in the same bit position
   - INV/Invert (bitwise-not) only operates on the Operand 1 input
   - `a^b` is XOR – equivalently `((a&!b)|(!a&b));`

   Complete table:

Operation	Operand 1	Operand 2	Result
ADD	00011000	00010100	
SUB	00011000	00010100	
INV	00011000	XXXXXXXX	
XOR	00011000	00010100	
SRL	00011000	XXXXXXXX	00001100

3. This will come up in lecture. Consider the datapath shown that supports data-dependent branching. This datapath also differs from the one from Lecture 15 (and in Problem 1) because it stores data in multi-bit words and operates on multi-bit words using operations like those in Problem 2.



   The basic instruction for branching is: BR SRC1 SRC2
   which behaves as follows: If (SRC1[0]==0) then PC=PC+SRC2, else PC=PC+1.
   SRC2 is signed, so can be a positive or negative value.
   SRC1[0] says we are looking only at the low bit of the data value read from SRC1.

   For each of the following consider an instruction sequence to accomplish the task (for simplicity assume you can preload memory slots with offset values as needed):

   (a) How would you **unconditionally** branch to the top of a loop that is 12 instructions before the branch instruction?

   (b) How would you **conditionally** branch to the top of a loop based on the value in memory slot 0 that is 12 instructions before the branch instruction?

   (c) How would you implement an if-then-else construct?