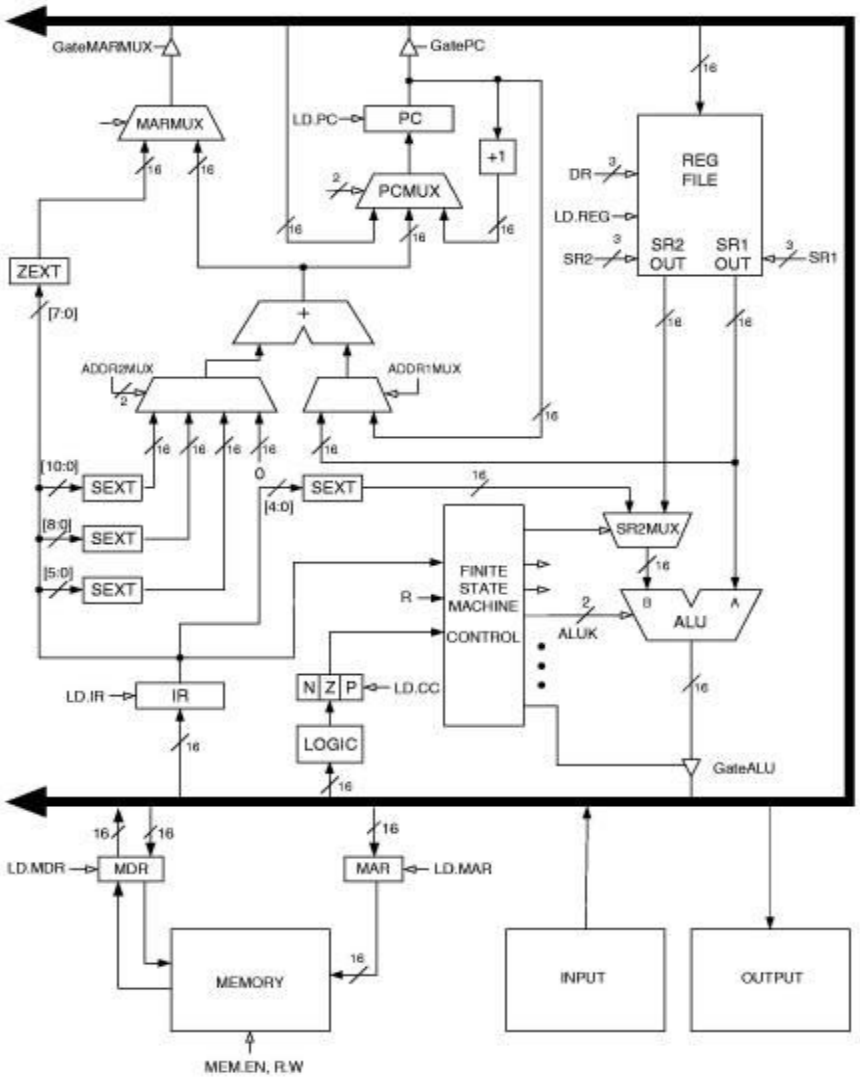
Effective address calculations, etc.



Eleven of the LC3's 15 instructions involve address calculation *(see the instruction table at the end)*:

- all the Load/Store instructions (and LEA) need to come up with a full 16-bit address to read or write data to/from memory; this address has to be written to the MAR (via the MARMUX & the bus).
- All the control instructions - BR, JMP, JSR/JSRR & TRAP - need to come up with the address of an instruction; this address has to be written to the PC (via the PCMUX).

So it should be no surprise that about half of the data path is dedicated to calculating addresses!

First question: why do we need to "calculate" an address?

Because our instructions are only 16 bits wide, and they need 4 bits for the opcode, plus at least 3 bits for other parameters such as a destination register, or the nzp condition code specs, etc.
But we need a 16-bit address for the PC or the MAR!
So we're going to need some "tricks" to use, say, a 9-bit field in the instruction to "point" to the desired address *(aka Effective or Target address)*.
These "tricks" are called "Memory Addressing Modes".

Second question: How will we do the calculation?

All address calculations start with an available 16-bit address, and then add an offset to it *(except for the TRAP instruction, which we will deal with later)*.
There are only 2 ways to get hold of an existing 16-bit address:
either the **PC**, or the **Source Register 1 bus**.

(Note that the PC always holds the address of the next instruction in line - i.e. the incremented address of the instruction currently being executed).

The address 1 mux (ADDR1MUX) allows us to choose one of these for our starting point.

Then there are 4 possible offsets we can choose, using the address 2 mux (ADDR2MUX):
offset6, PCoffset9, PCoffset11, or hard-coded 0 *(we will see shortly which instructions use which offsets)*. These offsets are all 2's complement - i.e. they must be *sign-extended* to 16 bits before we can add them to the chosen 16-bit starting address.
So all instructions that need an effective address (except TRAP) will:

1. select either the PC or SR1 bus as the starting value (ADDR1MUX)
2. select one of the 4 possible offsets (ADDR2MUX)
3. take the sum of these inputs, and either write it to the MAR *(via the MARMUX & the bus)*; or write it to the PC *(via the PCMUX)*.

Let's look at the actual "Memory Addressing Modes" used by the LC3.

- Direct (LEA, LD, ST, BR, and JSR): here, the Effective Address is indicated by a label, which is just an alias for an actual address. Our starting point for encoding this address will be the PC - i.e. the *incremented address of the current instruction*.
  When you assemble your code, the assembler does this calculation:
  `offset = (target address) - (incremented instruction address)`
  (when an instruction is being executed, the PC will of course be pointing to the next instruction).
  If the offset is > 255, or < (-256), an assemble error is thrown, since it cannot be represented as a 9-bit, 2's complement number.
  Otherwise, the assembler reduces the result to 9 bits, and embeds it in the instruction field PCoffset9, bits[8:0]
  For instance, the instruction `LD R3, label_x ; given label_x => offset #120 = x78` would assemble to LD = 0010; R3 = 011; 9-bit offset to label_x = 0 0111 1000.
  Putting it all together, we get: `0010 011 0 0111 1000 => x2678`
  When the instruction is being executed, the full target address (label_x) is reconstituted by the calculation **`EA = (PC) + SEXT(PCoffset9)`**
  Note that the JSR instruction uses an 11-bit offset, PCoffset11.

- Indirect (LDI, STI): this is structurally identical to the Direct Memory Addressing Mode, except that the value stored at the calculated address is not the target data, but the *address of* the target data: so we perform one memory access to obtain the EA, and a second memory access to perform the Load or Store at that EA.
  This means that after doing the above calculation, the result *(the full address of label_x)* must be written to the MAR; the memory triggered for Read; the resulting data copied from the MDR *back to the MAR*. In other words **`EA = Mem[(PC) + SEXT(PCoffset9)]`**
  The final step is then to read data from (LDI) or write data to (STI) the EA.

- Relative (LDR, STR, JSRR, JMP): here, a starting address has already been stored in a register (called the BaseReg), and the effective address is calculated by adding an offset to it - either offset6 (LDR & STR), or x0000 (JMP & JSRR)
  In other words, **`EA = (BaseReg) + SEXT(offset)`** *(where "offset" is either offset6 or 0)*

STORE vs LOAD Instructions

LD is `DR <- Mem[(PC) + SEXT(PCoffset9)]`
and assembles as `0010 DR PCoffset9`

ST is `Mem[(PC) + SEXT(PCoffset9)] <- SR`
and assembles as `1010 SR PCoffset9`

Notice that Load's DR is specified by the same bit-field as ST's SR (Instruction[11:9])

As our data path is constructed so far, that's a real problem - bits [11:9] are the input for the DR decoder, not the SR decoder (which is bits [8:6])!

The way we handle this is with some extra hardware - specifically a mux!
The SR1MUX has two 3-bit inputs:
- for the operational instructions, and for any instruction phases using a BaseReg, SR1MUX selects the default IR[8:6] *(input 0)* as the source register address.
- for the data phase of all Store instructions, SR1MUX selects IR[11:9] *(input 1)* as the source register address.

A second issue for the data phase of the Store instructions is: how do we get the data on the SR1bus into the MDR *(from where it will be written to the Memory address in the MAR)*?
We use a fourth ALUK option, "PassThru", which simply passes the data input at arm A of the ALU straight through to the output unchanged. The ALU Gate then writes it to the bus.

So for instance the Register Transfer steps of the ST instruction are:

- MAR <- (PC) + SEXT(PCoffset9) *(via the MARMUX & the bus)*
- MDR <- (SR) *(via SR1bus & the ALU & the bus)*
- MEM[(MAR)] <- (MDR)

The complete sequence of control signals that implement these steps:

MAR <- (PC) + SEXT(PCoffset9)

- ADDR1MUX selects PC
- ADDR2MUX selects SEXT(IR[8:0])
- MARMUX selects address adder
- Gate.Marmux
- LD.MAR

MDR <- (SR)

- SR1MUX selects IR[11:9]
- ALUK selects PassThru
- Gate.ALU
- Ld. MDR

MEM[(MAR)] <- (MDR)

- Mem.En/W

Instruction opcode & parameter table

	15 14 13 12	11 10 9	8 7 6	5	4 3	2 1 0
ADD+	0001	DR	SR1	0	00	SR2
ADD+	0001	DR	SR1	1	imm5	
AND+	0101	DR	SR1	0	00	SR2
AND+	0101	DR	SR1	1	imm5	
BR	0000	n z p	PCoffset9			
JMP	1100	000	BaseR	000000		
JSR	0100	1	PCoffset11			
JSRR	0100	0 00	BaseR	000000		
LD+	0010	DR	PCoffset9			
LDI+	1010	DR	PCoffset9			
LDR+	0110	DR	BaseR	offset6		
LEA+	1110	DR	PCoffset9			
NOT+	1001	DR	SR	111111		
RET	1100	000	111	000000		
RTI	1000	000000000000				
ST	0011	SR	PCoffset9			
STI	1011	SR	PCoffset9			
STR	0111	SR	BaseR	offset6		
TRAP	1111	0000	trapvect8			
reserved	1101					

Figure A.2 Format of the entire LC-3 instruction set. Note: + indicates instructions that modify condition codes