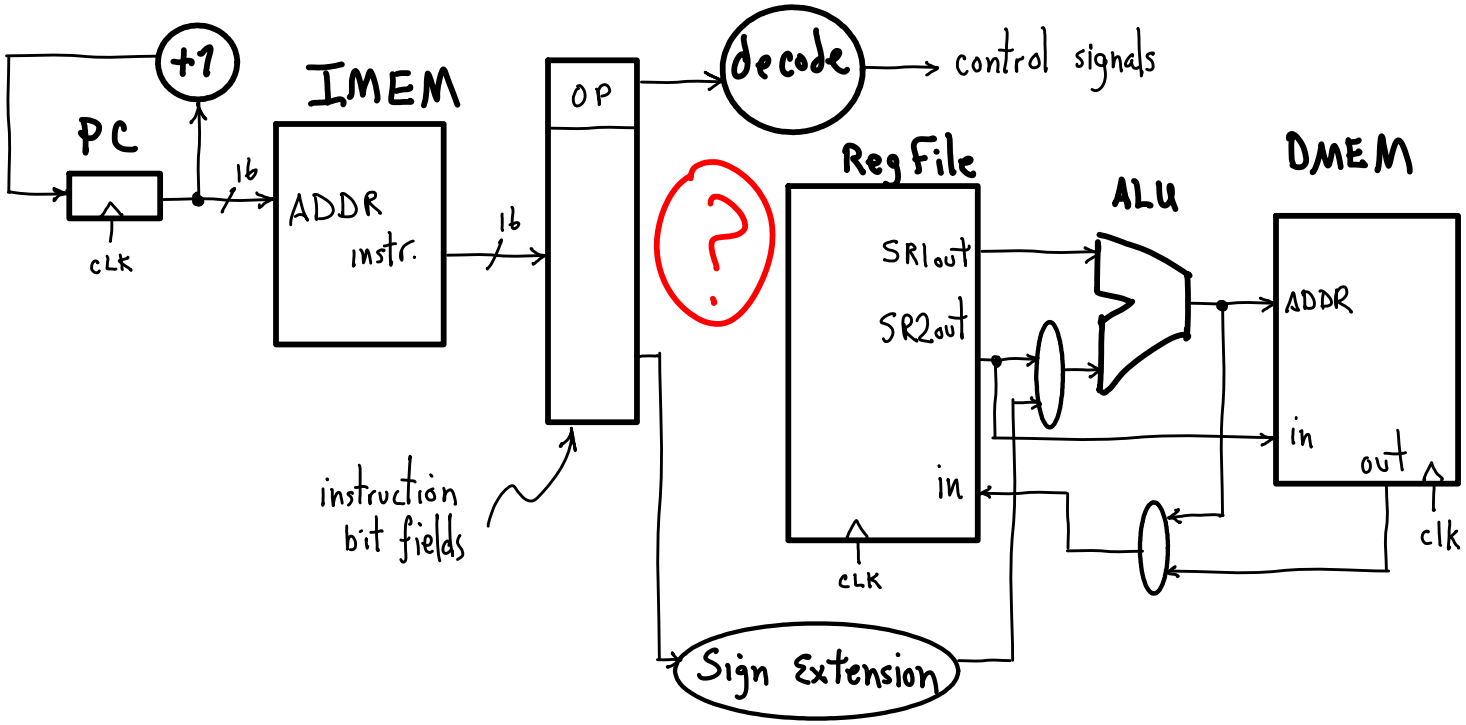
Lec-4-HW-1-LC3piped Reading: PH, Chp 4, and Lec-4a-MIPSserial.pdf

Below is preliminary rough layout of a modified LC3 suitable for pipelining. We would like to modify LC3's instruction set to make the circuit as simple as possible. After we have implemented this 1-cycle LC3 (shall we call it LC4?), we can easily convert it to a pipelined implementation and speed up the clock. In this design, the clock ticks, the PC's output changes, and all signals change accordingly, from left-to-right. When all signals have settled, the instruction's execution is finished. When the clock next ticks, the RegFile and DMEM are written into, and we start a new instruction execution. Thus, CPI = 1, but the clock runs very slowly. IMEM and DMEM are actually L1 caches.



There are no select lines into RegFile to control what is fed to SR1out and SR2out: we need to have SR1 and SR2 inputs from the instruction bits. We also need DR input to select which register is written. There are two memories: IMEM and DMEM for instructions and data, respectively.

We want to implement as few instructions as possible and still have a reasonable machine. Here is a selection of instructions that cover all the capabilities of LC3:

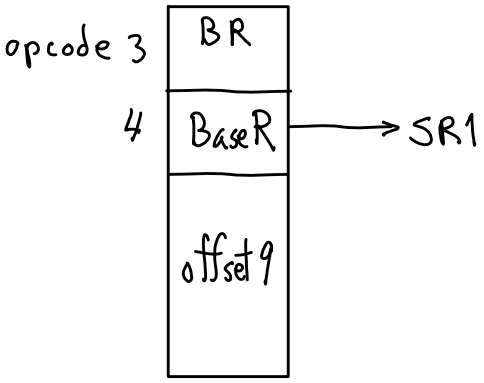
LDR, STR, LEA, BR, OP*, SHL, LIM.

*OP would be the opcode for ALU operations. A second bit-field in the instruction, as in MIPS, could be fed to the ALU to select the actual operation.

If we choose our instruction formats carefully, we can make the LC4's design and control signals very simple, and still pipeline it. That's our task in this exercise. In particular, feeding the instruction bits into the RegFile inputs can get complicated, requiring several MUXes and complicating control signals.

We want uniform instruction formats. We use base-register + offset addressing for almost all addressing needs.

Here are some ideas for instructions. You build on this and see what you can come up with. Figure out what new connections are needed that are missing above, or that need to be modified.

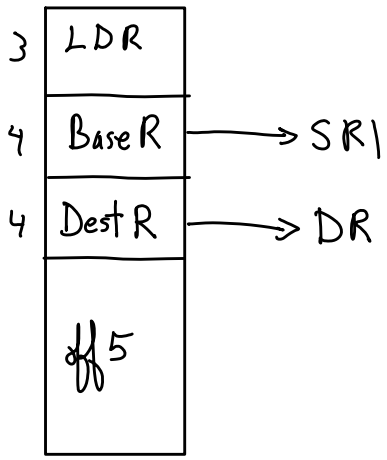


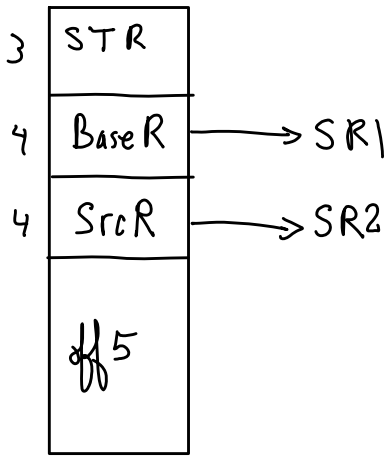
All instructions have bits instruction[15:13] as the opcode. Instruction[12:9] designate the Base Register for addressing. This allows for 16 registers, which would help us a lot. SR1 always selects the Base Register for address arithmetic in every instruction.

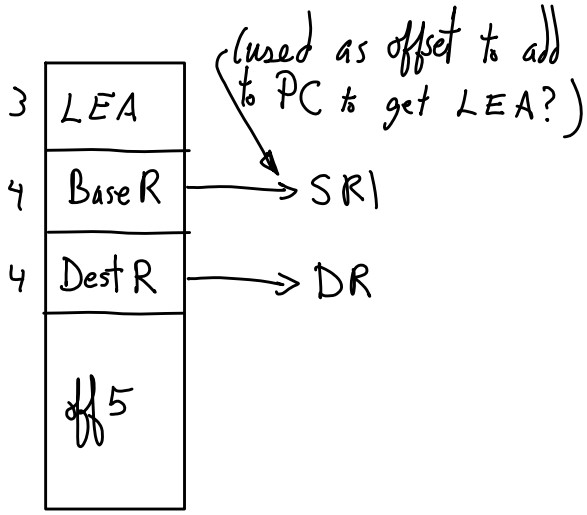
BR can be used to implement Jump-via-register, Jump-to-subroutine, and Trap. For regular branches, BaseR holds the branch target address.

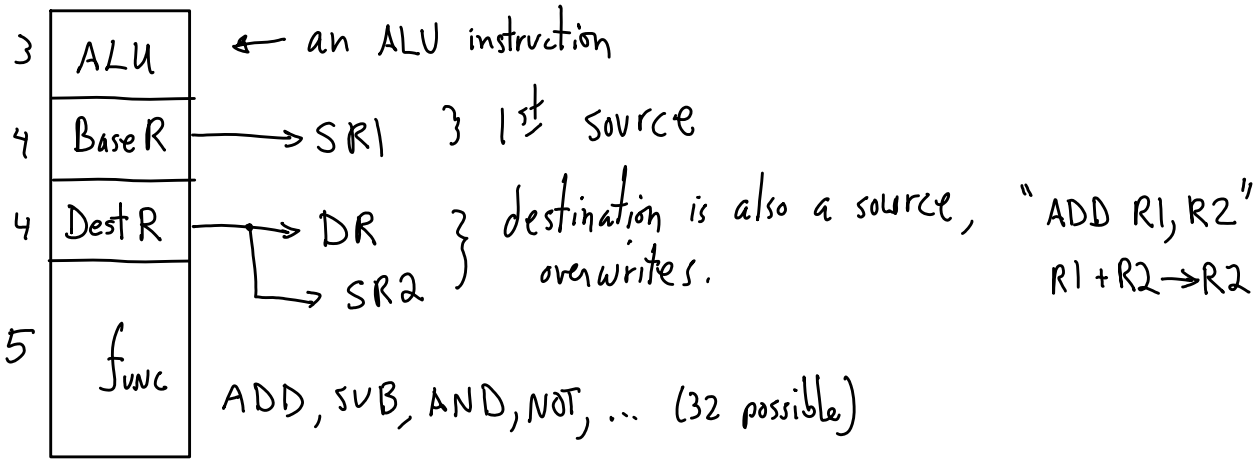
BR is conditioned on a prior zero output from the ALU, which is stored in a 1-bit register (not shown). This makes BR a BRz instruction. Any branch (e.g., BRn, BRp, BRnp, ...) can be implemented by doing enough logic and BRz. To do branching, do SUB, for instance, and then BR will branch if the result was zero.

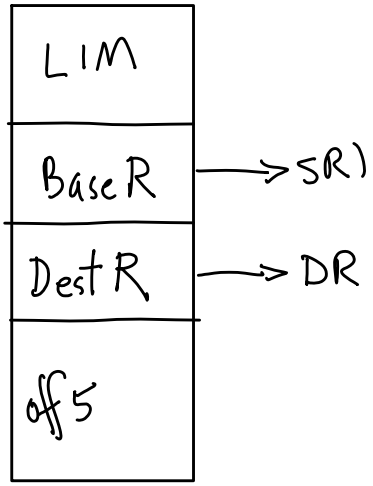
It might be worth it to use the offset bits for some other purpose here.



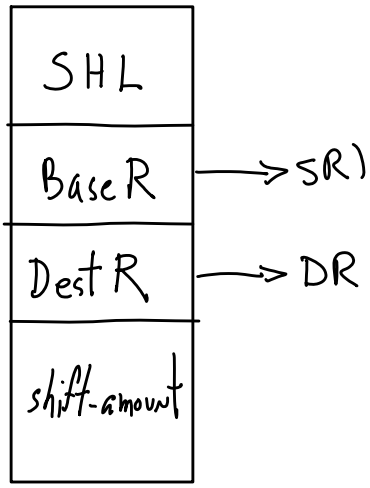








We have limited immediate values we can use. LIM gets us 5 bits, unsigned, loaded into the low bits of DestR. Using this with SHL (see below), will allow us to get 16 bits into a destination register using 4 LIM instructions with 3 SHL instructions. BaseR could be used as a shift left amount?



SHL shifts bits left in DR by the number in the "shift-amount" field. I don't know what BaseR would be used for. It could be used as the source so DestR could be a different register.

Maybe LIM can be used instead, and we can eliminate SHL?

Well, this has been my thinking so far. We can do things MIPS style (no RTI stack push/pop type instructions, special registers reserved for the OS so it can save user's state without clobbering user registers. This is why we will need plenty of registers. Exceptions and interrupts can use reserved registers to hold "cause" and the PC value when an interrupt or exeception occurs. We can also simplify by having all exceptions and interrupts go to a single address, and the code there can figure out what happened and what to do next. For instance, a reserved register can hold a trap number, and we use BR to jump to the specific address. The code there jumps to a trap routine according to what is in the reserved register. We shift complexity from hardware to software this way. But, who cares? Well, good luck. It will be fun to see what you come up with.