midterm exam
COSC-120, Computer Hardware Fundamentals, fall 2013
Computer Science Department
Georgetown University

NAME Soln

Open books, open notes, but no electronic devices. Show and explain **all** your work. Answers w/o explanation **will not get credit**. Questions are always perfectly clear to me, and are also maybe clear to you, but sometimes with a different meaning. Therefore, partial credit is important, and I will give as much credit for whatever question you seem to be answering as far as I have evidence in the form of an explanation. This applies to questions that are over your head as well.

Section I. (write "True" or "False", or circle ANY correct answer, or fill in the blank, as appropriate; explain as needed.)

Q. Which of the following are not high-level programming languages:
C++, x86 machine language, Java, LC4 assembly language, Java byte code.

C++ and Java are high-level languages.

Q. A program can be translated into the ISA of a processor by means of:
a compiler, an assembler, an interpreter, all of these.

These all rely on execution of machine language.

Q. The value of the decimal digit 5 can be represented as which of these bit strings:
00000000000101, 000101, 00110101.

The last one is 32+16+4+1.

Q. Two values A = 1111111111110101 and B = 111110101 are 2's complement representations of integers. Let vA and vB be the values they each encode. The values vA and vB are such that:
vA < vB, vA = vB, vB < vA, we cannot tell the relationship between the values from the information provided.

They both encode the value -11 (decimal). Leading 1's are ignored for negative numbers.

Q. The value +16 encoded in 4-bit 2s-complement representation is:
0111, 1000, 1111, not possible to represent in 4-bit encoding.

Max 4-bit is 0111 = +7
Min 4-bit is 1000 = -8

Q. We have a collection of k distinct things. So that each can have its own, unique, 3-bit label:
k must be greater than 4, k must be 3, k must be less than 5, k must be at most 8, k can be any number.

There are eight 3-bit codes: {000, 001, 010, ... , 101, 110, 111}. We can label at most 8. We could label fewer.

Q. 1A is a hexadecimal representation of the same number as is represented by the unsigned binary string:
00011010, 0011010, 011010, 11010, none of these, all of these.

h1A = (0001 1010) or (00011010); unsigned leading zeroes are ignored: they all encode the same value.

Q. Let A[15:0] = 0110110010001111, as an unsigned binary number A[13:12] is:
less than A[3:2], greater than A[3:2], equal to A[3:2]

A[3:2] = unsigned(11)= +3; A[13:12] = unsigned(10) = +2, which is less.

Q. Marvin Minsky was taking cosc-120 and wanted to remove a file ("foo") from his branch ("myBranch/") in the Subversion repository. Marvin had a working copy of myBranch/. He entered the commands shown below in a terminal window on his machine ("**-->**" is the shell's prompt). Later, he found myBranch/foo was:
gone from myBranch, still there in myBranch, not in the repository under myBranch, still in the repository under myBranch.

```
-->   cd myBranch
-->   rm foo
-->   svn ci
-->   svn up
```

"rm" is a shell command: the OS deletes the file from the local file system. "svn rm" is a Subversion command, it schedules a file to be removed from the repository on the next "svn ci" and it has the OS remove the local copy. "svn up" replaces any files that are in the repository but missing locally.

Q. Grace Hopper was also taking cosc-120. Her LC4 project design was in an Electric .jelib file, lc4.jelib. Using Electric's menus, she opened the top-level cell of her design and did (followed by two terminal commands):

```
^Tool.Simulation(Verilog).WriteVerilogDeck
-->   iverilog foo.v
-->   vvp a.out
```

We can make a good **guess** that in Grace's lc4.jelib file,
foo{sch} is the design's top-level cell, foo{ic} probably does not exist, her entire LC4 circuit is in foo{sch}.

A top-level cell is an {sch} cell. Being top-level, it would be unusual to make an icon for it. We usually make hierarchical designs; so, it is a good guess Grace's is also multi-level.

Q. Grace's foo.v included the following verilog code:

```
reg clk;
always @( posedge clk ) begin
    #10 clk = 0;
end
always @( negedge clk ) begin
    #10 clk = 1;
end
always @( clk ) begin
    $write(" t=%d, clk=%b, ", $time, clk);
end
initial begin
       clk = 0;
    #1 clk = 1;
    #1 clk = 0;
end
```

The "@(posedge clk)" means wait until the clock makes a 0-to-1 transistion; "negedge clk" is a 1-to-0 transition. Because all the "always" and "initial" statments run in parallel, Grace thinks it might be possible to see this output:

```
t=0 clk=X, t=0 clk=_0_, t=1 clk=1, t=2 clk=0, t=11 clk=0, t=12 clk=_1_
```

because the 1st "always" is triggered by the 2nd change in the "initial", and the 2nd "always" is triggered by the 3rd change in the "initial". **True** **False**

These events are generated (time, what): (0, clk=X), (0, clk=0), (1, clk=1), (2, clk=0). A rising edge occurs at (1, clk=1); a falling edge at (2, clk=0). These last two events cause the creation of events (11, clk=0) and (12, clk=1) because the always statements were triggered. Output occurs whenever clk changes.

Section I (continued).

Q. A programming language is Turing complete if it provides the language elements to describe any computational machine. Most machines we call computers have an ISA that is Turing complete. Joe argues that while a high-level language such as C is complete, the LC4's machine language is too restrictive to be complete. Jane points out that every LC3 instruction can be simulated by one or more LC4 instructions, and that there is a C compiler for the LC3. Jane says, "From that we can conclude:

every C program can be rewritten in LC4 ISA, LC4 requires a C compiler to be complete, LC4's ISA is complete.

It is safe to assume C must be a complete language. Any machine that can execute C programs must also be universal (able to simulate any machine.) C programs can be compiled to run on LC3. LC3 machine instructions can be simulated by LC4; therefore, a translator could convert any C program to LC4 machine instructions. Therefore, LC4's machine language must be complete.

Q. Because a universal Turing machine can be simulated by a typical computer, we can define an effective procedure, or algorithm, as a computer program that is guaranteed to complete its computation and give the correct answer. So, any program we write that has those features is an algorithm. Whether or not a Java program is an algorithm can be determined by knowing that it:

always halts, is bug free, is designed to loop forever (such as an OS or shell), has no branches.

A program that is guaranteed to exit (halt) might be buggy, a bug-free program might loop forever, a program w/o branches might be incorrect. A program that is designed to loop forever either: 1) does loop forever, or 2) incorrectly halts. In either case, it cannot be an algorithm.

Q. Suppose the LC5 defines a new ISA: it extends the LC4's ISA by adding one new instruction, MUL, for multiply. Multiplying two n-bit numbers produces a 2n-bit result. Consequently, the semantics of "MUL SR1 SR2 DR3 DR4" is that it multiplies the content of the two source registers, R1 and R2, and stores the result's most-significant 16 bits in R3 and the least-significant 16 bits in R4. We conclude that,

LC4 machine language programs will run on the LC5, no LC5 program can run on the LC4, LC5 instruction format is not 16-

Because LC5 includes all LC4 instructions, LC4 programs will run on an LC5. Any LC5 program that does not include MUL has only LC4 instructions. Nothing was mentioned about changing instruction formats.

Q. The LC3 does not have a subtract instruction. However, for doing arithmetic with 2s-complement encoded values, we can do the following to produce the encoding of the negative of the value held in a register,

R3 <=== NOT(R3)
R3 <=== R3 + 1

E.g., if R3 contained the encoding for +1, it will then have the encoding for -1, and vice versa. This is done using LC3's NOT and ADD instructions. Subtraction consists of the above operations, then adding:

R2 <=== R2 + R3

The LC4's "ALU SR2 SR3 DR2 SUB" also does subtraction, but without changing the value in R3. Suppose we have a large LC4 program that does nothing but a lot of subtraction. An equivalent LC3 program **might**,

have 5 times the instructions, take 5 times as long, have 3 times the instructions, have the same number of

Changing R3 back requires two more instructions: each LC3 subtract might be five instructions. Five LC3 instructions might take 5X longer than one LC4 instruction. If we never restore R3, it's only 3X more instructions. There cannot be the same number of instructions doing the same number of subtractions.

Q. LC3/LC4 logic such as NOT, AND, OR, and so on, are implemented as 16 bit-wise operations. E.g., the LC4's "ALU SR1 SR2 DR3 AND" does the following operation on the i-th bits, for i = 0 to 15:

R3[i] <=== R1[i] AND R2[i]

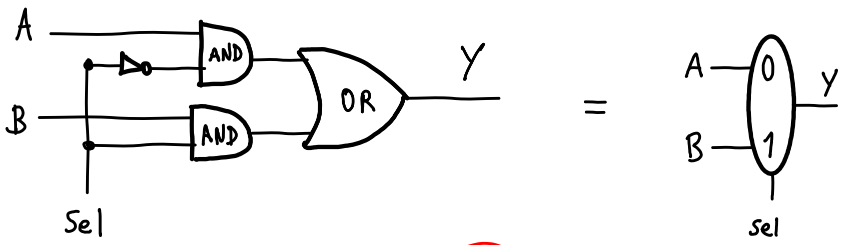
However, C's "if (A == B)" evalutates a 1-bit expressions, (A==B), that has the value TRUE or FALSE. Implementing "if" in LC3/LC4 machine code **might** have to implement TRUE/FALSE by ("h" indicates hex notation),

using a bit mask, e.g., h8000, using 16-bit values, e.g., h0000 and h1111, detecting if bit_15 is 0 or 1, it is

The machine code that implements "if" has to branch based on creating a value from the contents of whatever registers hold values A and B. What that value is, as a boolean, must be detected by branching. These values shown could be used for this branching process.

Section II.

Q. At right is shown a circuit for a 1-bit, 2X1 mux and its circuit symbol. Complete the tables below for that device. Hint: notice the patterns each signals changes.



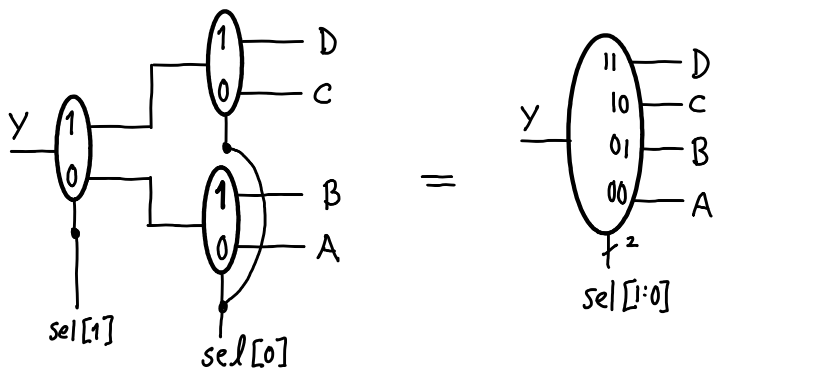
	sel	A	B	Y
t=0:	0	0	0	0
t=1:	0	0	1	0
t=2:	0	1	0	1
t=3:	0	0	0	0
t=4:	0	0	1	0
t=5:	0	1	0	1
t=6:	0	1	1	1

	sel	A	B	Y
t=0:	1	0	0	0
t=1:	1	0	1	1
t=2:	1	1	0	0
t=3:	1	0	0	0
t=4:	1	0	1	1
t=5:	1	1	0	0
t=6:	1	1	1	1

	sel	A	B	Y
t=0:	0	1	0	1
t=1:	0	1	0	1
t=2:	1	1	0	0
t=3:	1	1	0	0
t=4:	0	1	0	1
t=5:	1	1	0	0
t=6:	0	1	0	1

	sel	A	B	Y
t=0:	0	0	1	0
t=1:	1	0	1	1
t=2:	0	0	1	0
t=3:	1	0	1	1
t=4:	0	0	1	0
t=5:	1	0	1	1
t=6:	0	0	1	0

Q. At right is shown a circuit for a 1-bit, 4X1 mux and its circuit symbol. Complete the table below for that device. Hint: first notice how sel[1:0] changes its value.



	sel[1:0]	A	B	C	D	Y
t=0:	00	0	0	1	0	0
t=1:	00	1	0	1	0	1
t=2:	00	0	0	1	0	0
t=3:	00	1	0	1	0	1
t=4:	01	0	1	1	0	1
t=5:	01	0	1	1	0	1
t=6:	01	0	0	1	0	0
t=6:	01	0	0	1	0	0
t=6:	11	0	1	1	0	0
t=6:	11	0	1	1	1	1
t=6:	11	0	1	1	1	1
t=6:	11	0	1	1	0	0

Q. Suppose A=B=C=1 and D=0. For which values of sel[1:0] will the output be 1? 0? Treating sel[1:0] as the argument to a function and Y as the output of the function, what is the name of the logic function the circuit implements?

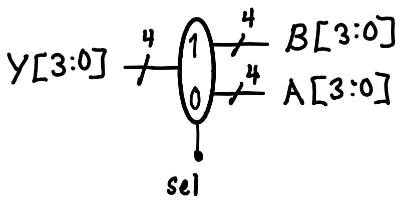
1 for every value except 11

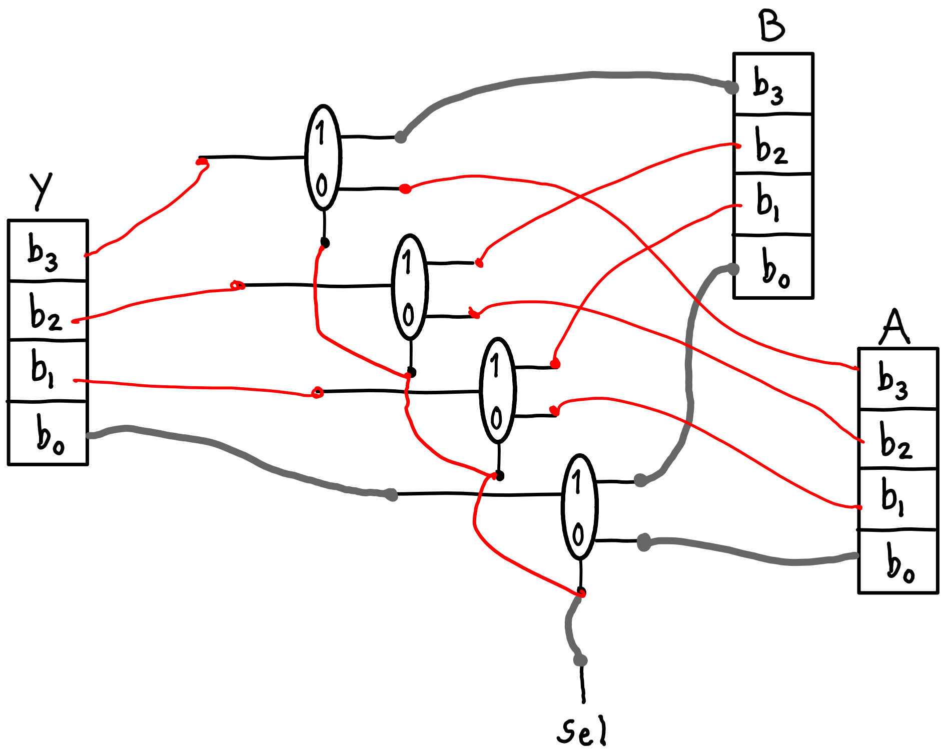
Sel	
00	1
01	1
10	1
11	0

$\rightarrow$ NAND(sel)

Q. At right is an incomplete circuit for a 4-bit, 2X1 mux. Complete the circuit by drawing in appropriate connections, or naming connection points to indicate connections ("connect-by-name").

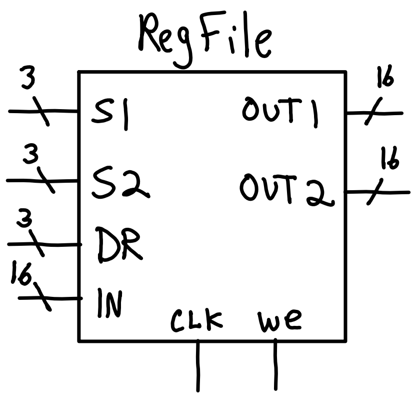
The circuit symbol for this device is shown below.





The same register file (RegFile) is used in both the LC3 and LC4. It consists of eight 16-bit D-FF registers, and two 16-bit 8X1 muxes. Each mux has its own 3-bit select input. This allows the contents of two registers to be read simultaneously. The select inputs to the muxes are named "S1" and "S2", and their respective 16-bit data outputs are named "OUT1" and "OUT2", as shown at right.

There is also a 3-bit destination register select "DR", that selects one register to be written. The data to be written comes from the 16-bit input "IN". The write is done at the next positive edge of the "clk" input, but only if the write-enable "we" is 1.



Q. Suppose we wanted to have an instruction that reads two registers and writes a register, all at the same time. How many bits would the instruction need to have connected to the RegFile's two read and one write select inputs to specify all three registers? As opcodes are an instruction's most-significant four bits and cannot be used to select registers, are there enough bits in a 16-bit instruction to accomodate such an instruction?

need 3 selects @ 3 bits $\rightarrow$ 9 bits for selecting

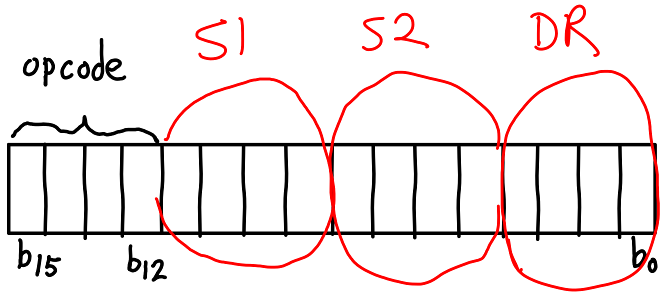
16 − (4 opcode bits) = 12 bits left, plenty for our selects.

Q. We decide eight registers are not enough, we want 16 instead. How many bits do we now need for the read and write selects each? For the two-reads-one-write instruction mentioned above, would it be possible to implement it in a 16-bit instruction? If possible, show an instruction format for that instruction below: show which instruction bits connect to the S1, S2, and DR select inputs; if not, explain why not.

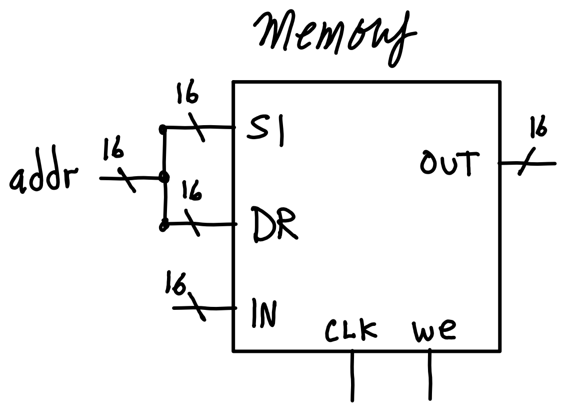
Need 4 bits for a select.

3 selects @ 4 bits $\rightarrow$ 12 bits,

we just have enough



The device at right is the same as the RegFile above, but with some changes: 1) there is a single read select S1 and a write select DR, which are tied together and named "addr"; 2) as there is only one read select, there is only one OUT; and 3) the read and write selects are now 16 bits. The registers are now called "memory words", and a bit pattern that selects a particular word is called its "address".

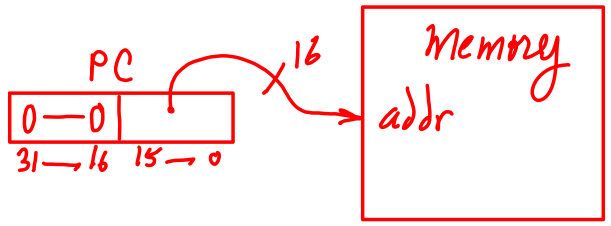


In the LC3, addr comes from the 16-bit register MAR, IN comes from the output of a 16-bit register MDR, and OUT goes to the input of MDR. In the LC4, if the memory is the instruction memory, addr comes directly from the PC, IN is ignored, and OUT goes to the IR bus.

As there are 16-bit selects, there are 2^16 words, or 64k 16-bit words. We also call a word a cell and its address its location.

Q. Suppose we doubled the size of the ALU, all the datapath registers (R0-R7, PC), and busses in the LC4 to 32 bits. Could we use the same memory as shown above? Ignoring other considerations, given that the PC would now be 32 bits, how would we get address information to the instruction memory, which only has 16-bit addresses? Suggest how this might be done. Hint, some of the PC's output bits could be ignored and permanently 0.

The address input is still 16-bit. We could use only the Least-Significant (LS) 16 bits of the PC, and route them to the memory's addr[15:0] input. MS 16 bits of the PC could be all zeroes. The programs written for this machine must conform to executing in a 64k word memory.

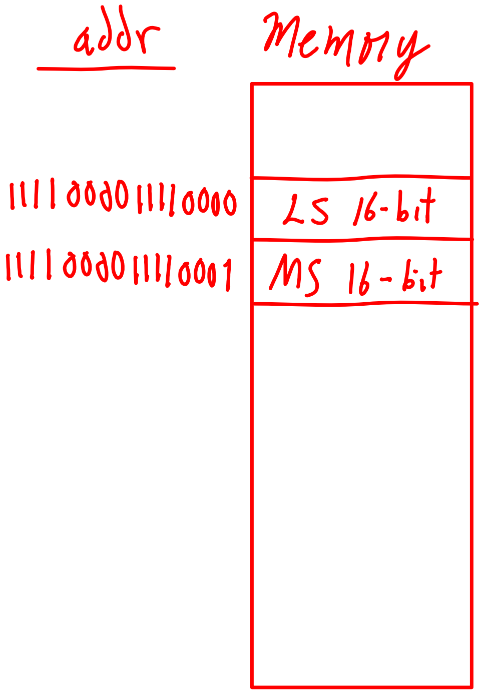


Q. Let's consider the next issue: would instructions be 16 or 32 bits? Would data be 16 or 32 bits? In the LC4, which has an instruction memory and a data memory, the two memories would be the same as the above memory. If instructions and data are 32 bits, how do we get them from memory? Hint, we can use two clock cycles to access memory twice. In that case, at what addresses are the least-significant 16 bits of the 32-bit words (Hint, either the least-significant or the most significant come first in address order)?

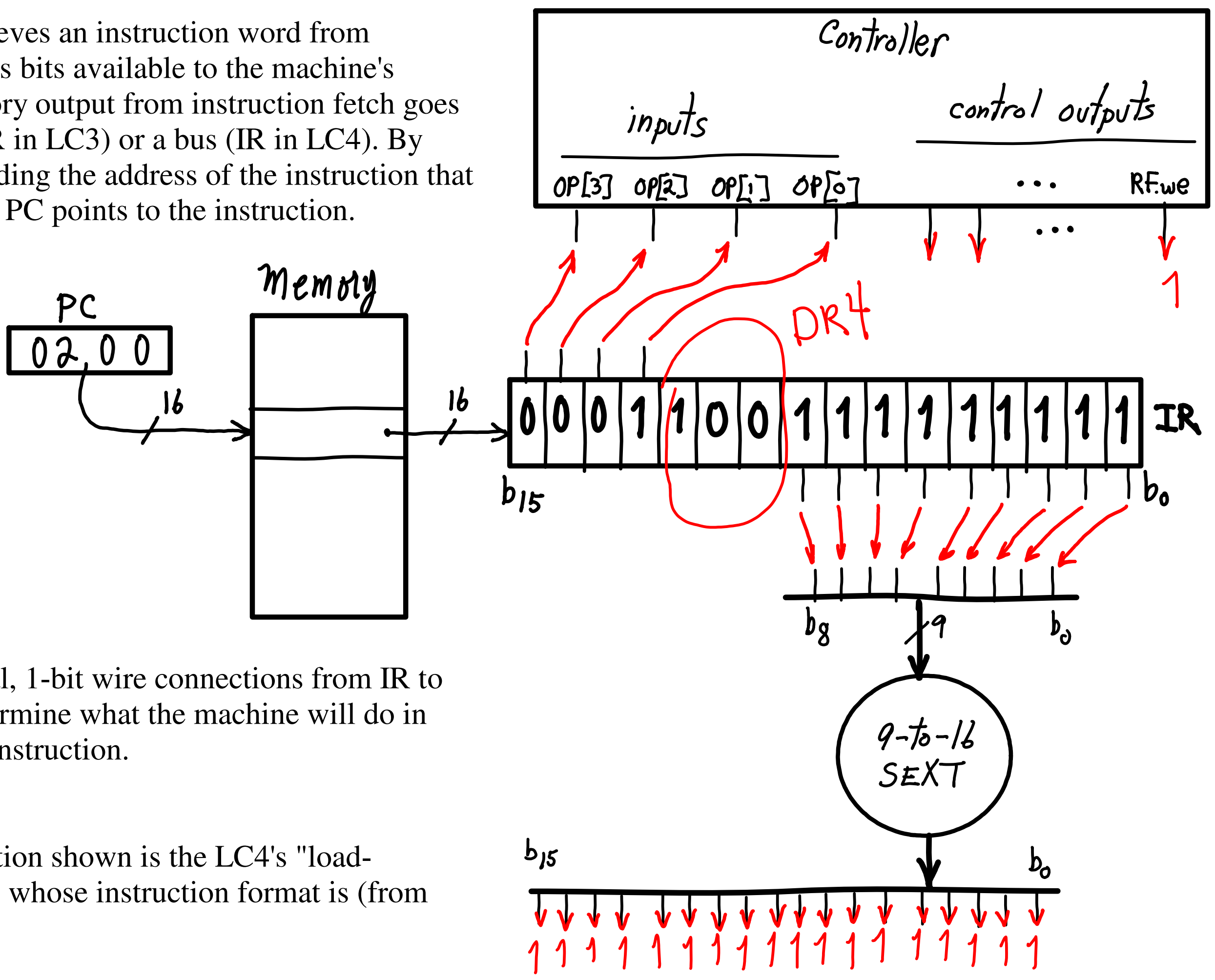
Memory words are still 16-bit. The new datapath is 32-bits; therefore, data words are 32-bits. It wasn't mentioned whether or not the IR is now 32-bit. If so, instructions are now 32-bit. To fetch a 32-bit instruction or 32-bit data word requires two memory accesses. So, we do one 16-bit read followed by another 16-bit read to complete reading a 32-bit data item or instruction.

We don't know if this is done in hardware in a single instruction execution, or done by fetching and executing two instructions. In either case, it would make sense to store the 16 LS bits next to the 16 MS bits. The LS bits could be in locations w/ even addresses, and the MS bits in locations w/ odd addresses (or vice versa).

An example layout is shown at right. This is little-endian.



Instruction fetch retrieves an instruction word from memory and makes its bits available to the machine's controller. The memory output from instruction fetch goes either to a register (IR in LC3) or a bus (IR in LC4). By virtue of the PC's holding the address of the instruction that is fetched, we say the PC points to the instruction.



Q. Show the individual, 1-bit wire connections from IR to the controller that determine what the machine will do in executing the current instruction.

Q. The current instruction shown is the LC4's "load-immediate data", LIM, whose instruction format is (from MSb to LSb),

[opcode, DR select, 9-bit data]

For LIM, the controller steers the sign-extended data to the RegFile's IN input. It also sets its control outputs so that the RegFile will be written: the signal shown above "RF.we" is connected to RegFile.we. Show the current bit value on that control signal, and indicate which register is being selected to be written.

Controller's RF.we = 1 for a register write to occur.
The register written will be R4.

Q. In executing LIM, the 9-bit data is sent to a sign-extender, SEXT. Show the individual wire connections from IR to the sign-extender's input. Also show the individual bit values exiting SEXT.

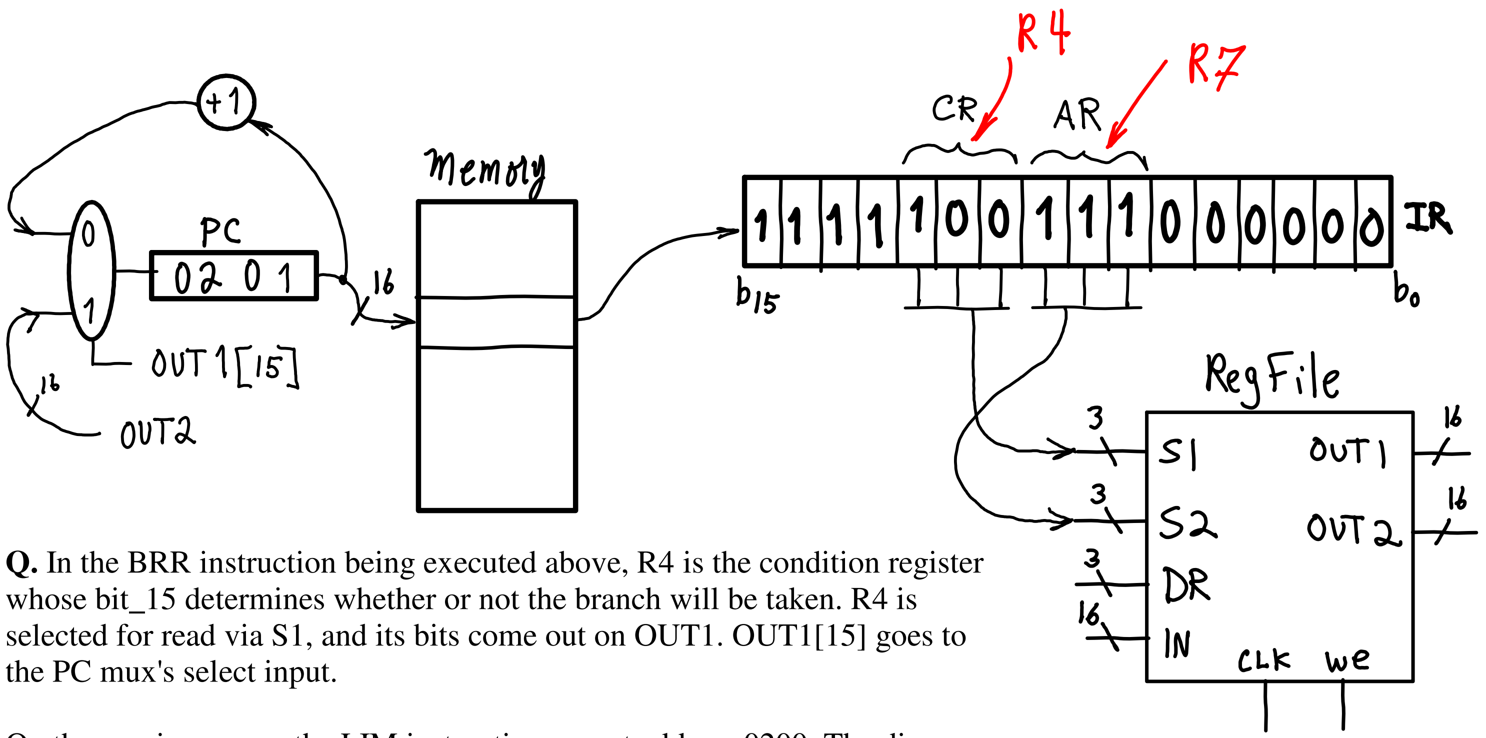
The value coming out of SEXT is a 16-bit, 2s-complement encoding of what value?

Value is -1, all 15 bits coming out of SEXT are 1s.
n 1s for n-bit, 2s-complement encoding encodes the value -1.

Q. The next instruction fetched after LIM will come from what location in memory, given the PC content as shown above?

PC gets 0200 + 1 = 0201 will be the address used for the next instruction fetch.

Q. In the LC4, all branching and jumping, of any kind, is done with the single instruction, BRR. A taken branch loads the PC with a value other than PC+1: If the select=1 on the 16-bit 2X1 mux feeding the PC's input, the branch target address goes to the PC's input.



Q. In the BRR instruction being executed above, R4 is the condition register whose bit_15 determines whether or not the branch will be taken. R4 is selected for read via S1, and its bits come out on OUT1. OUT1[15] goes to the PC mux's select input.

On the previous page, the LIM instruction was at address 0200. The diagram here shows the BRR instruction immediately follows in location 0201. Can you determine whether or not the branch will be taken? Why or why not.

This BRR instruction specifies R4 as the condition register, which means the branch will be taken if R4[15] == 1. According to the PC value shown above, the BRR instruction is in location h0201, just after the LIM instruction in location h0200. LIM used R4 as its destination register, and stored -1 (16'hFFFF) into R4. So, all R4's bits are 1, and the branch will be taken.

Q. Suppose R7 contains 0200 when the above BRR instruction was fetched. Given what you know about R4, can you determine whether or not this program ever halts? Explain.

This BRR instruction also specifies R7 as the branch target address register, which means that if the branch is taken, PC <=== R7. The assumption is made that R7 == h0200; so, the jump will be back to the memory location containing the LIM instruction above. Because, LIM always loads R4 with -1, the branch will always be taken. This is an infinite loop.