cosc-120, midterm review

Things to know

A little bit of unix:
ls (list dirctory contnets), cd (change current working directory),
cp (copy file), mv (rename file), rm (delete file), mkdir (create sub-directory),
shell environment variables (in particultar, PATH).

A little bit of Subversion:
svn rm (schedule a file to be removed from repository),
svn add (schedule a file/directory to be added to repository),
svn ci (send changes to repository),
svn update (get changes from repository).

A little bit of Electric:
Export (port) connections, wires versus busses, WriteVerilogDeck.

A little bit of Verilog:
The connection between an Electric Export and a Verilog argument list,
"wire" versus "reg", "initial" and "always" statements, delays.

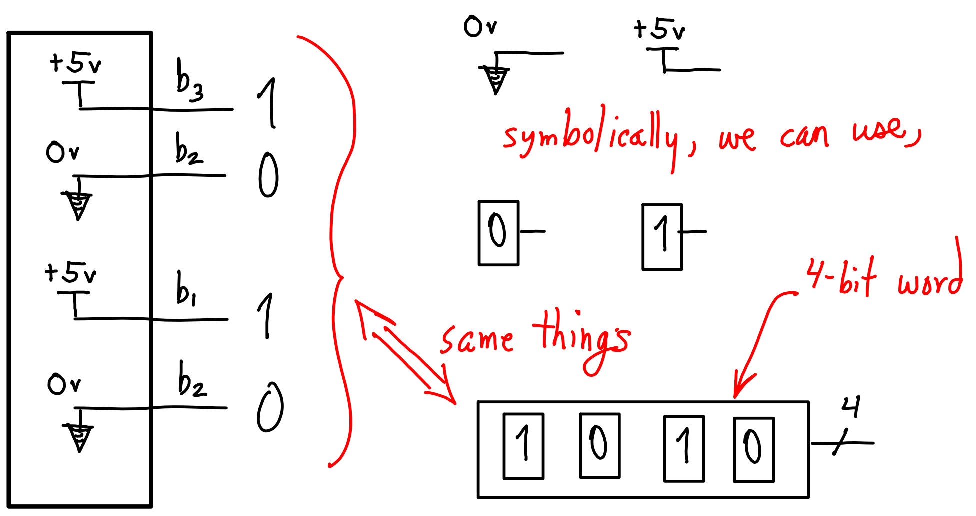
Von Neumann Architecture:
datapath, control, PC, Memory [address, data_in, data_out, write_enable],
Turing Competeness. LC3 has a finite-state machine controller w/ multiple states to execute a single instruction, LC4 has a controller w/ one state per instruction, in both cases control signals write-enable datapath registers and set muxes.

Memory:
Word size (number of bits of data read or written in one access),
bit order, byte order (position of bits/bytes within data (where is Least-Significant-bit, LSb)),
MAR (or address input),
MDR (or data-input or data-input),
addressability (smallest granularity of addressing),
address space (range of possible addresses).

The difference between an ISA and a micro-architecture:
ISA, an instruction set and abstract hardware features are accessible to programs;
micro-architecture, the basic hardware construction implementing the ISA's abstractions.

Logic:
AND, NAND, OR, NOR, NOT, and bit-wise logic;
binary addition (unsigned), simple 2s-complement representation, sign-extension

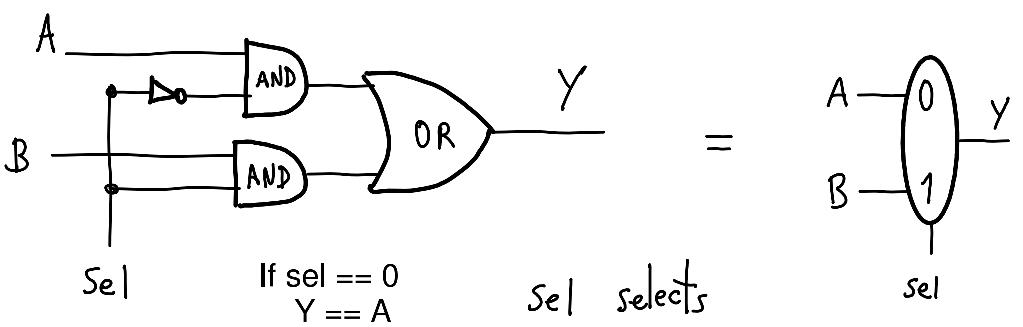
a 4-bit, read-only register (or word), Containing the value 1010.



a MUX

1-bit

2-input X 1-output



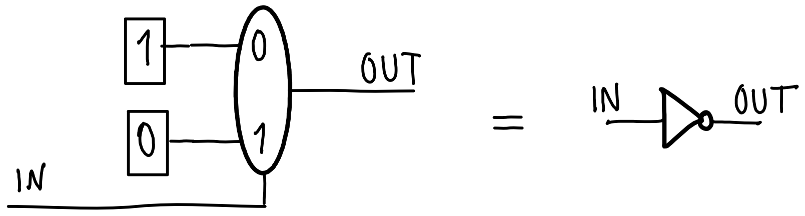
If sel == 0
  Y == A
If sel == 1
  Y == B

Sel selects which signal, A or B, has its value at Y.

Setting A = 1 and B = 0,

We can think of this as a logic gate:

If IN == 0
  OUT == 1
If IN == 1
  OUT == 0

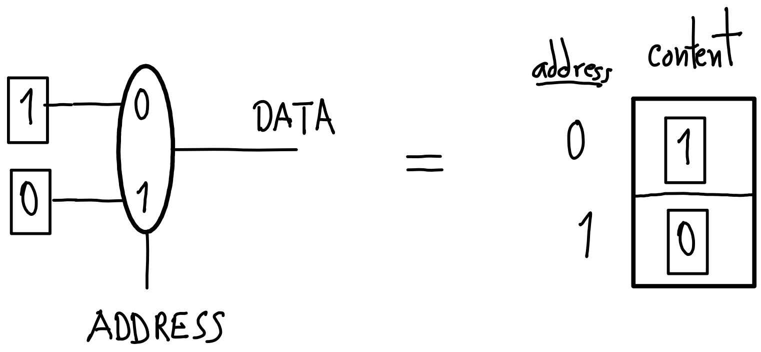


We can also think of this as a Read-Only-Memory:

if ADDRESS = 0
  DATA = content of cell 0 (== 1)

If ADDRESS = 1
  DATA = content of cell 1 (== 0)

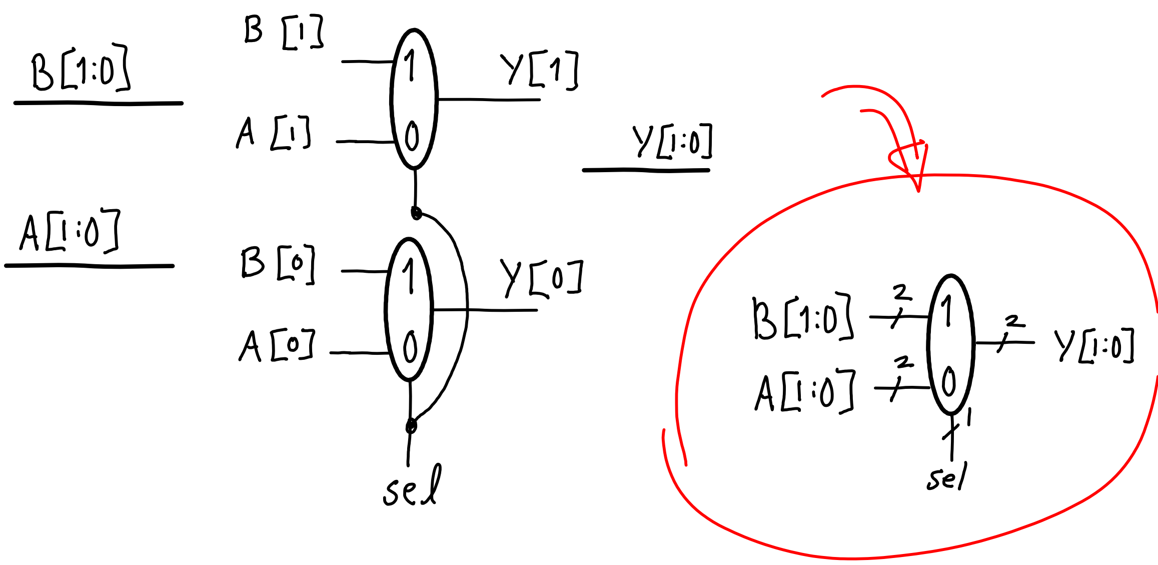
Memory words are 1-bit, addressability is 1-bit, address space is 2 (address range is 0 to 1), total size of memory is (2 words X 1bit per word) == 2 bits.



multi-bit MUX

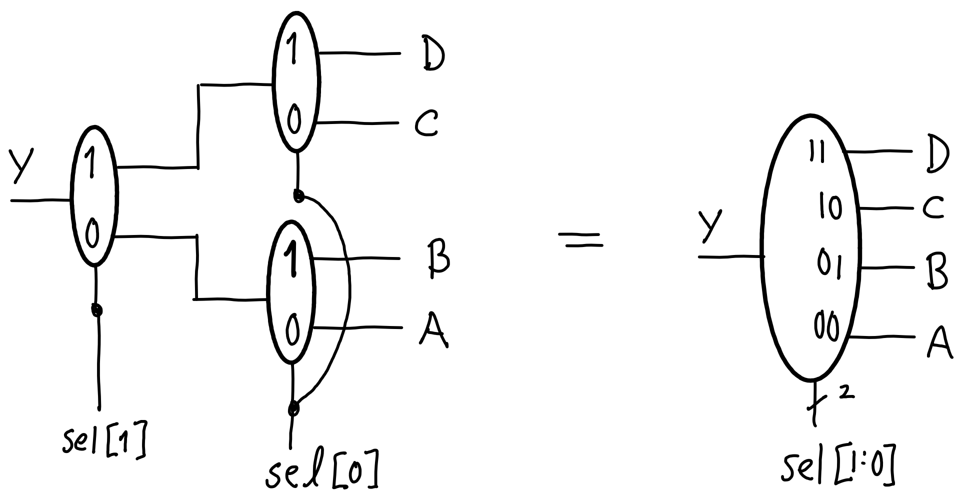
2-bit,

2-input X 1-output



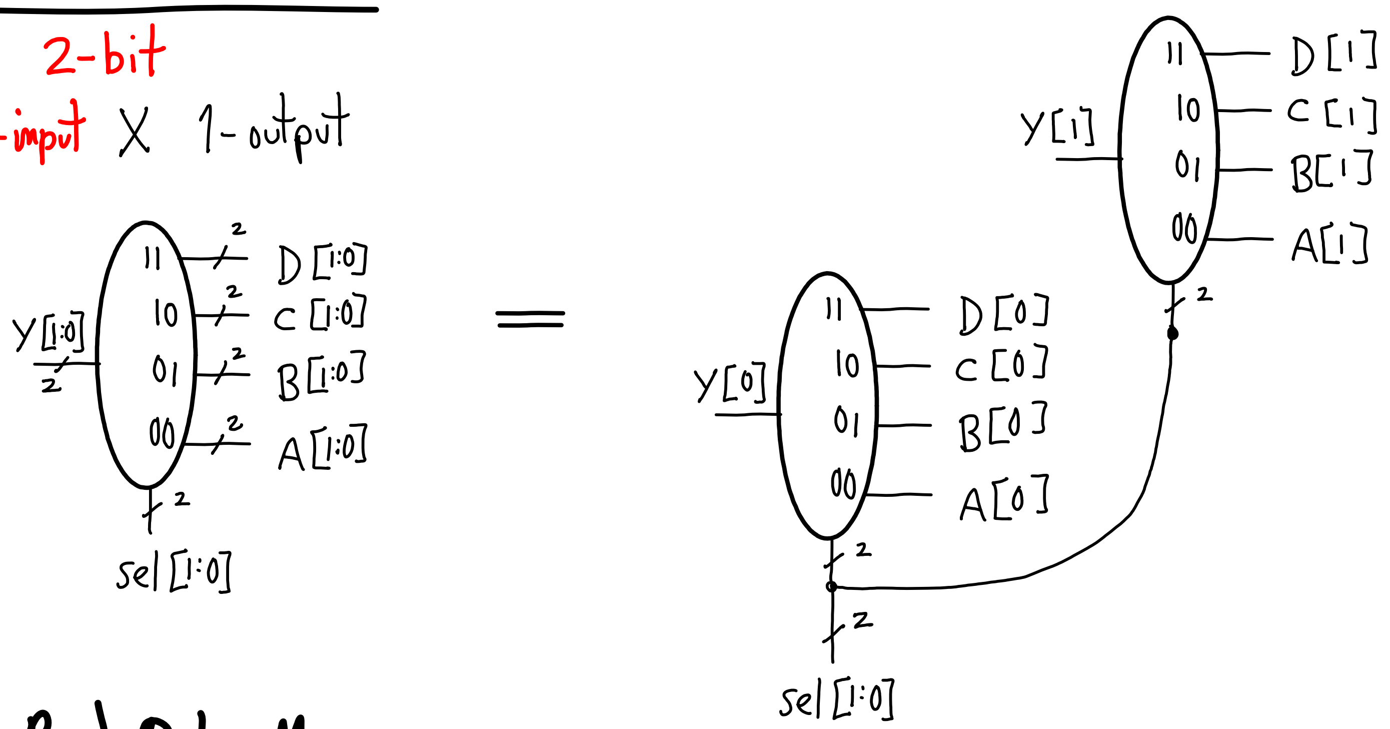
Multi-Way MUX

1-bit
4-input X 1-output

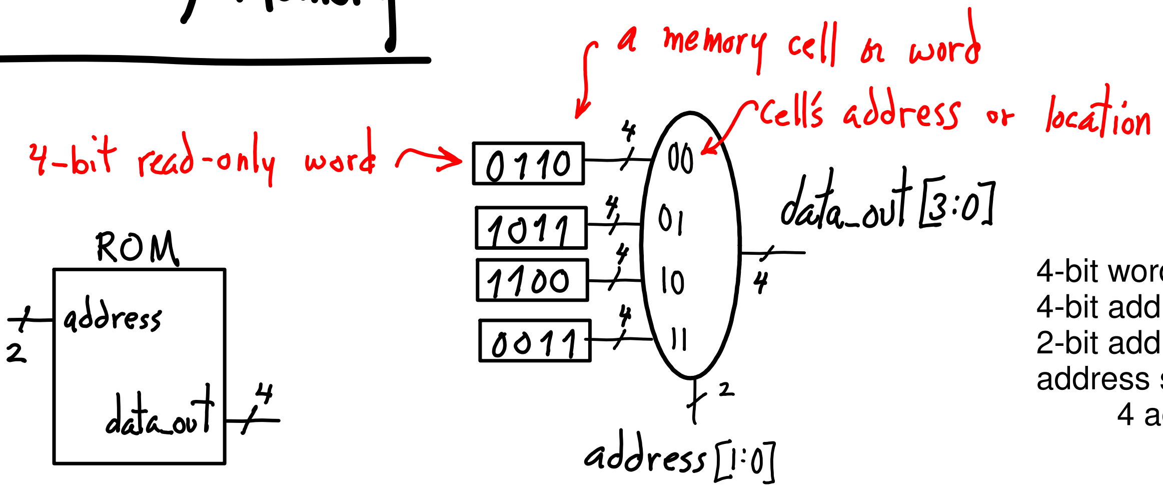


Multi-bit, Multi-Way MUX

2-bit
4-input X 1-output

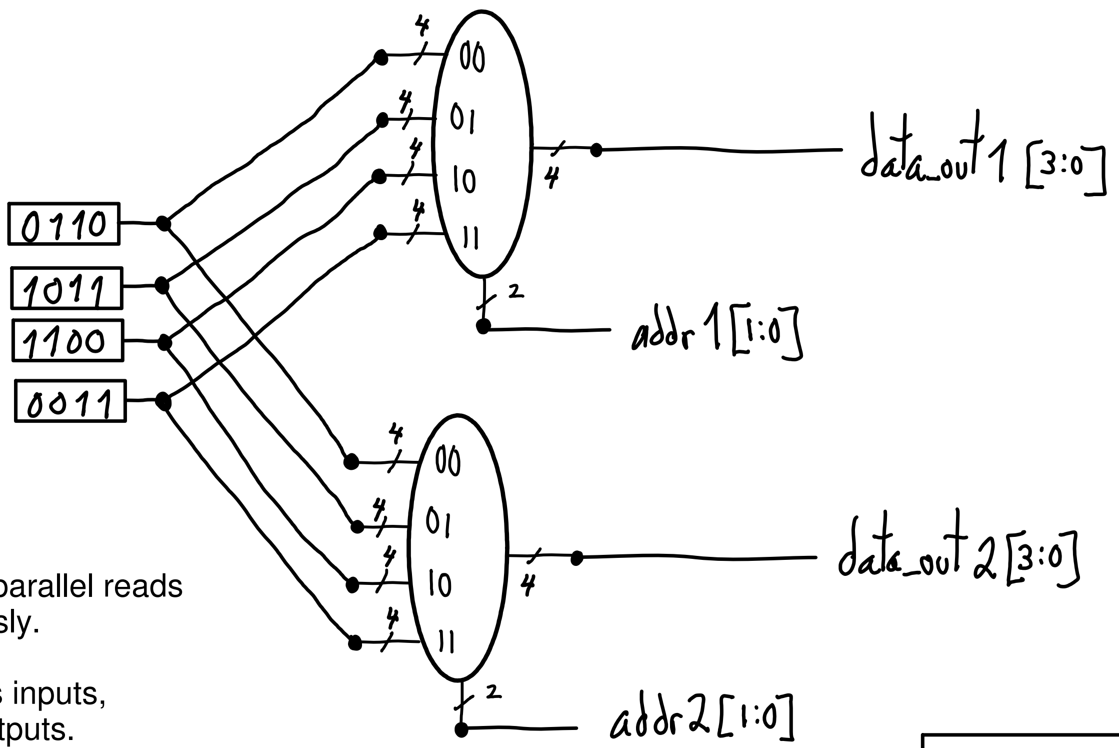


A Read-Only-Memory



4-bit words
4-bit addressability
2-bit addresses
address space: 00 to 11,
4 addresses

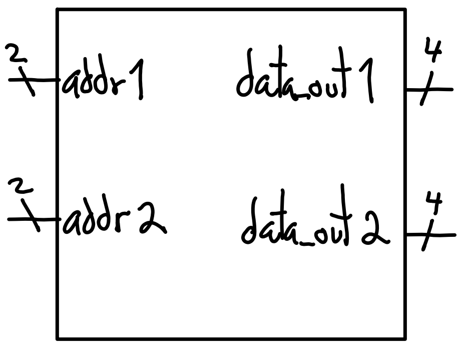
ROM w/ 2 read ports



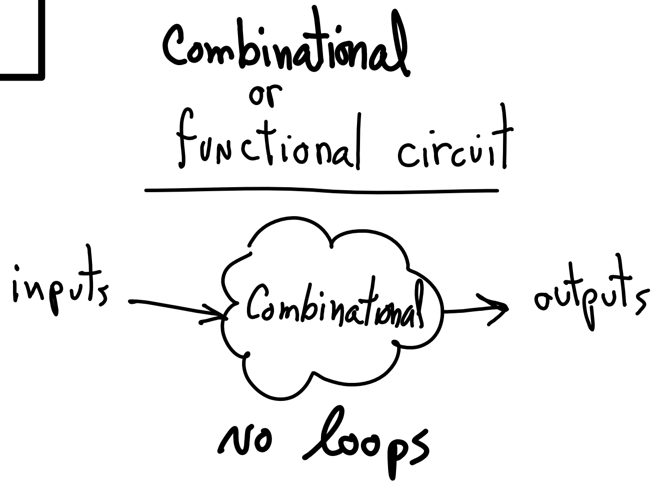
Can do two parallel reads simultaneously.

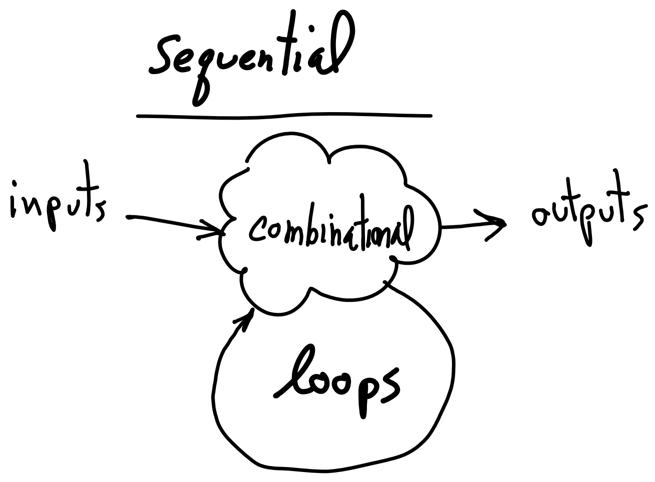
Two address inputs,
Two data outputs.

Reads are independent of each other.



State

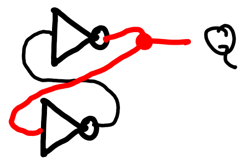




has state

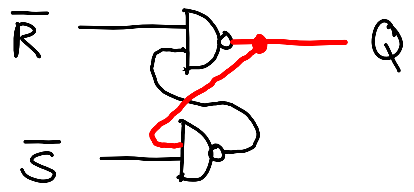
e.g. mux

e.g.



state 0 : Q=0
state 1 : Q=1

$\overline{R}$-$\overline{S}$ latch



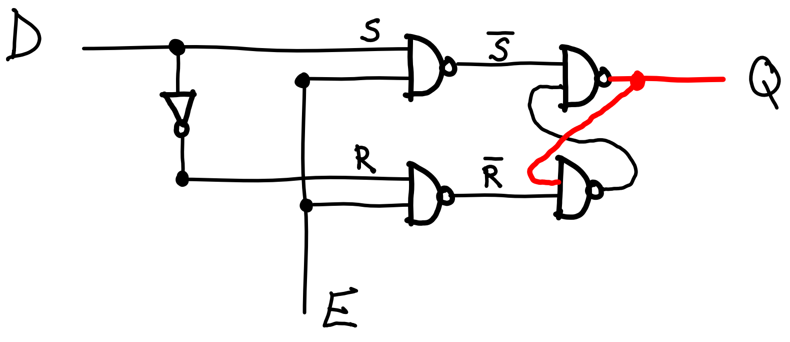
controllable state

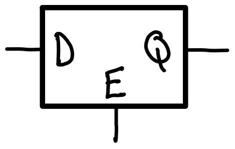
normally $\overline{R} = \overline{S} = 1$, holds state

set: $\overline{S}=0$, $\overline{R}=1$, $Q=1$

reset: $\overline{S}=1$, $\overline{R}=0$, $Q=0$

D-latch w/ enable





IF E == 0
  The S and R NANDS output 1:
   NOT(S) == NOT(R) == 1 , stable state

IF E == 1

  IF D == 1
    NOT(S) == 0
    NOT(R) == 1
    Q == 1 == D

  IF D == 0
    NOT(S) == 1
    NOT(R) == 0
    Q == 0 == D

Q follows D

like a wire, "transparent"

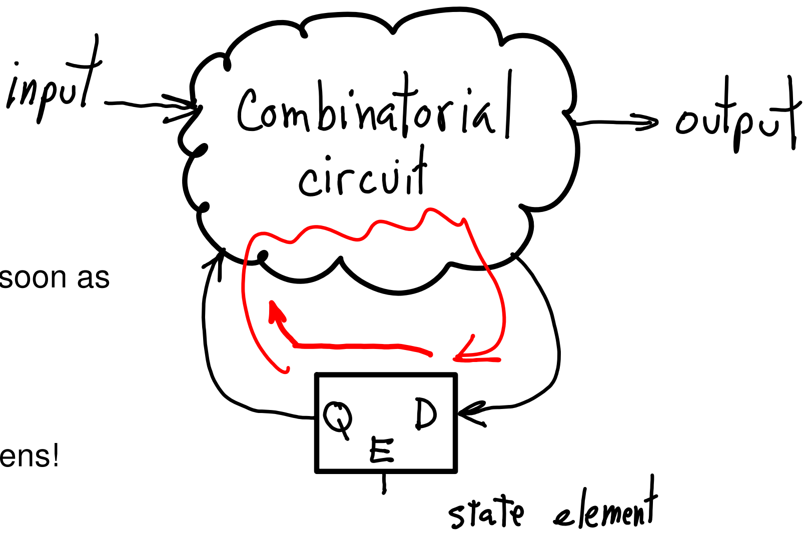
if $E = 0$ no changes

if $E = 1$ $Q = D$

This a basic 1-bit writeable data register

feedback problem

Finite State Machine



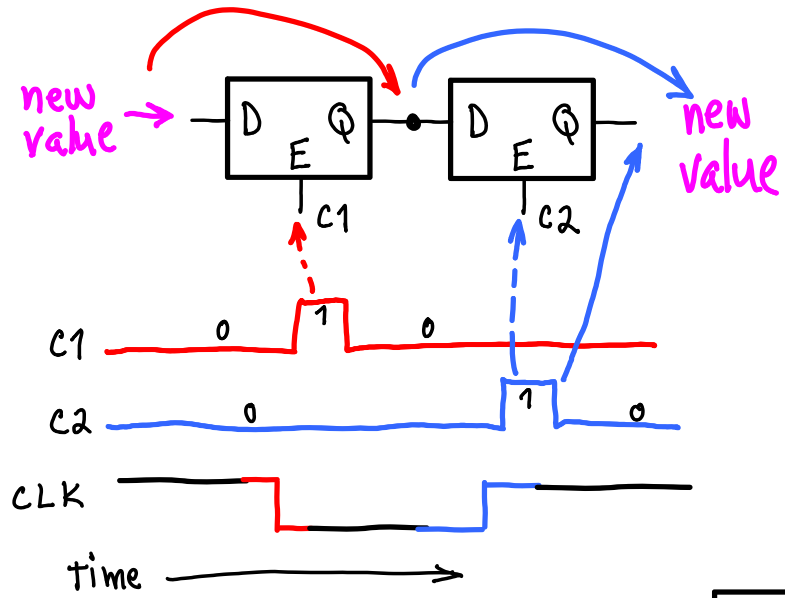
If we use a D-latch for the state element, as soon as we set E=1, uncontrolled feedback occurs!

State continually changes!

We need to control when state change happens!

D - flip flop

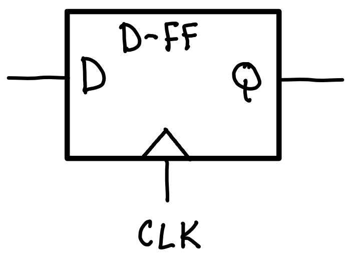
Solution: use 2 D-latches,
enable only one at a time



CLK generates two signals:
C1 controls left latch
C2 controls right latch

CLK goes **1-to-0**: **new data** moves in,
negative clock-edge captures data;

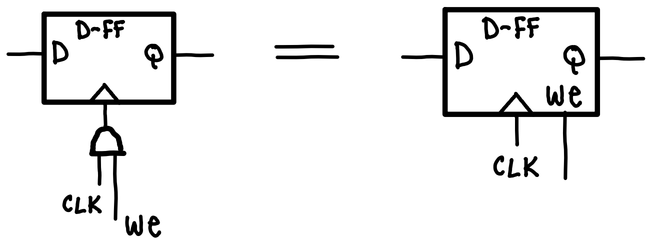
CLK goes **0-to-1**: **new data** at Q,
positive clock-edge changes state.



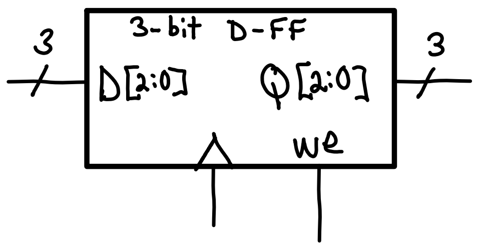
a positive edge-triggered D-flip flop

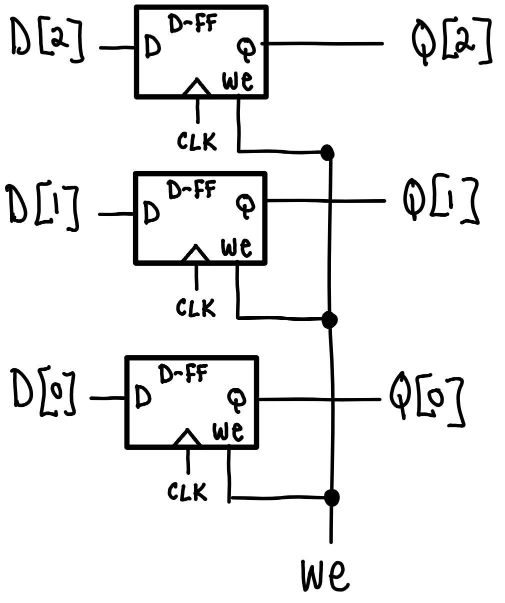
This is our read/write data element,
our **1-bit register**.

One more detail:
We need to **control if it will be written**, it needs a **write-enable**



3-bit R/W register





All our memory words for R/W memory are D-FFs.
All our R/W registers are D-FFs.
All the registers in our RegFile are D-FFs.

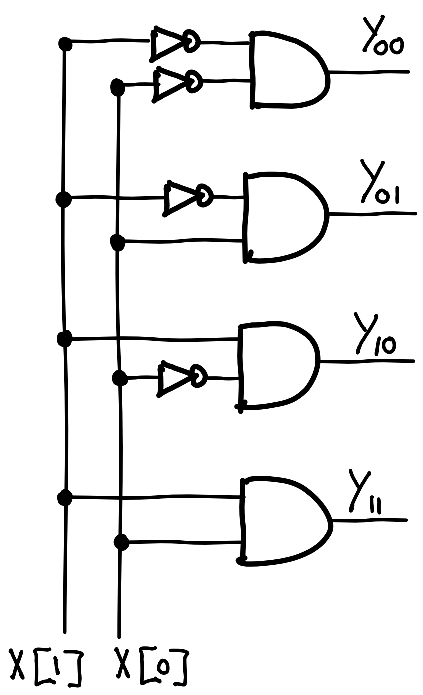
Memory or RegFile writes

Cannot build a "backward MUX".

Instead, control which register is written.

Use write-enables, choose which is 1, (if any).

a decoder, 2-input X 4-output



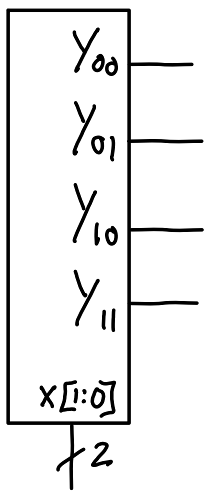
IF **X[1:0] == 00**
Y_00 == 1
Y_01 == 0
Y_10 == 0
Y_11 == 0

IF **X[1:0] == 01**
Y_00 == 0
Y_01 == 1
Y_10 == 0
Y_11 == 0

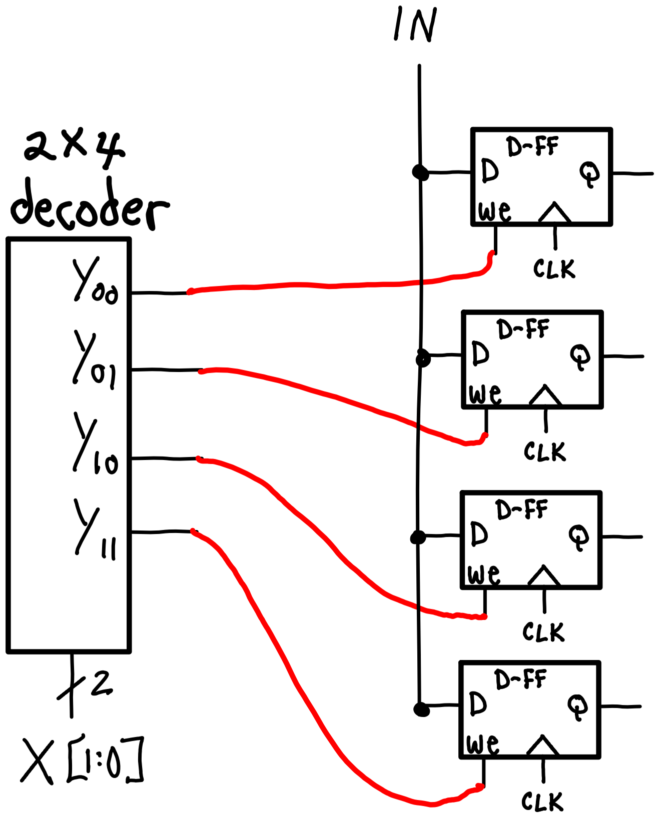
IF **X[1:0] == 10**
Y_00 == 0
Y_01 == 0
Y_10 == 1
Y_11 == 0

IF **X[1:0] == 11**
Y_00 == 0
Y_01 == 0
Y_10 == 0
Y_11 == 1

exactly one output is 1



X[1:0] selects which output is 1.



A 4-word memory:

Data to be written is IN.

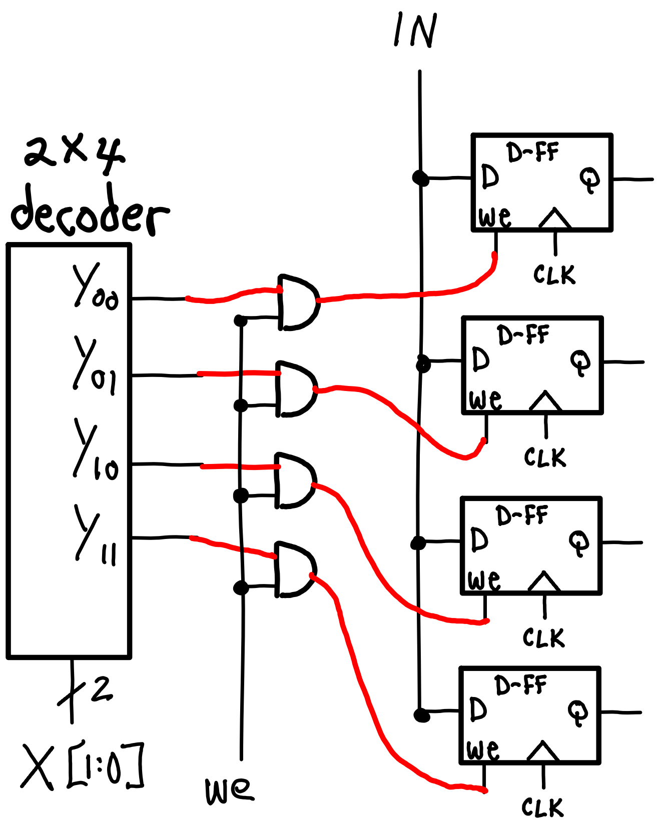
Address to write to is X[1:0].

A 4-register RegFile:

Data to be written is IN.

Destination Register to write is X[1:0].

Oops, a write *always* occurs. What if we don't want to write at all?

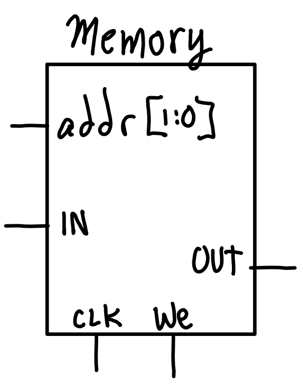


Now the write only occurs if we ==1.

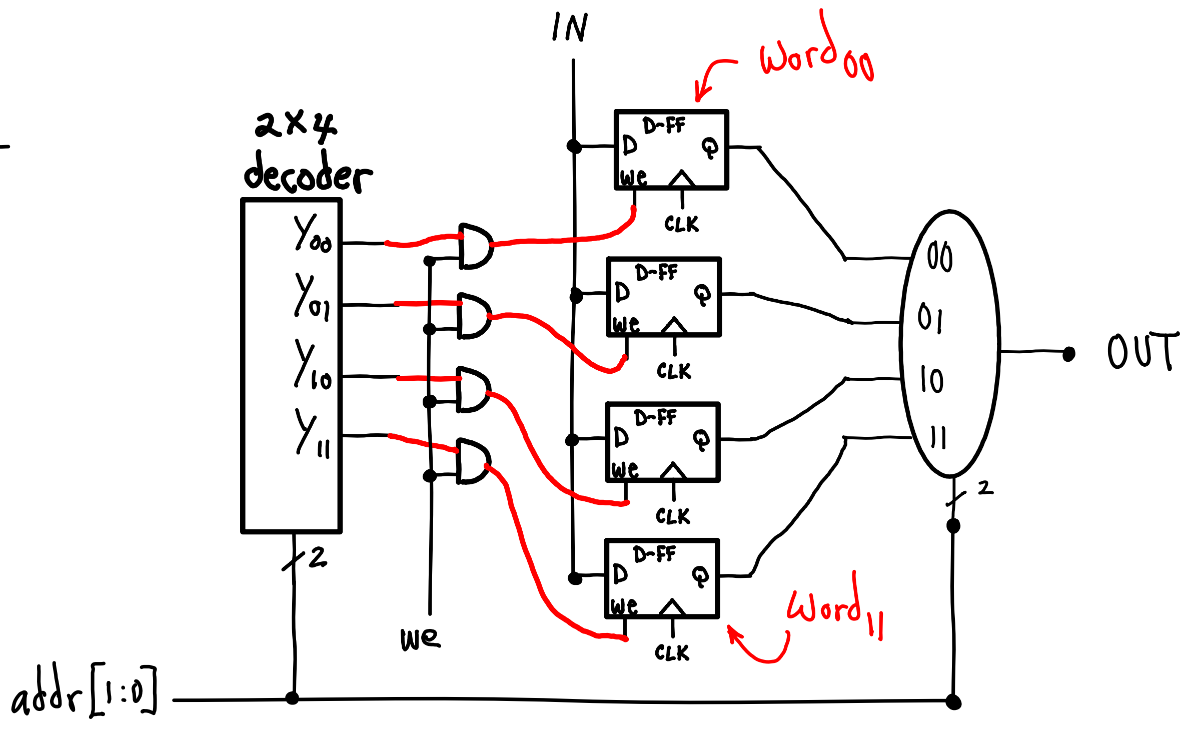
The RegFile has a DR input, which selects the destination register. If the circuit at left had a 3-bit select, DR[2:0], in place of X, we would have 8 registers, as both the LC3 and LC4 RegFiles have.

For LC3 and LC4 main memory, we have a 16-bit select, addr[15:0], in place of X. That give us 64k memory words.

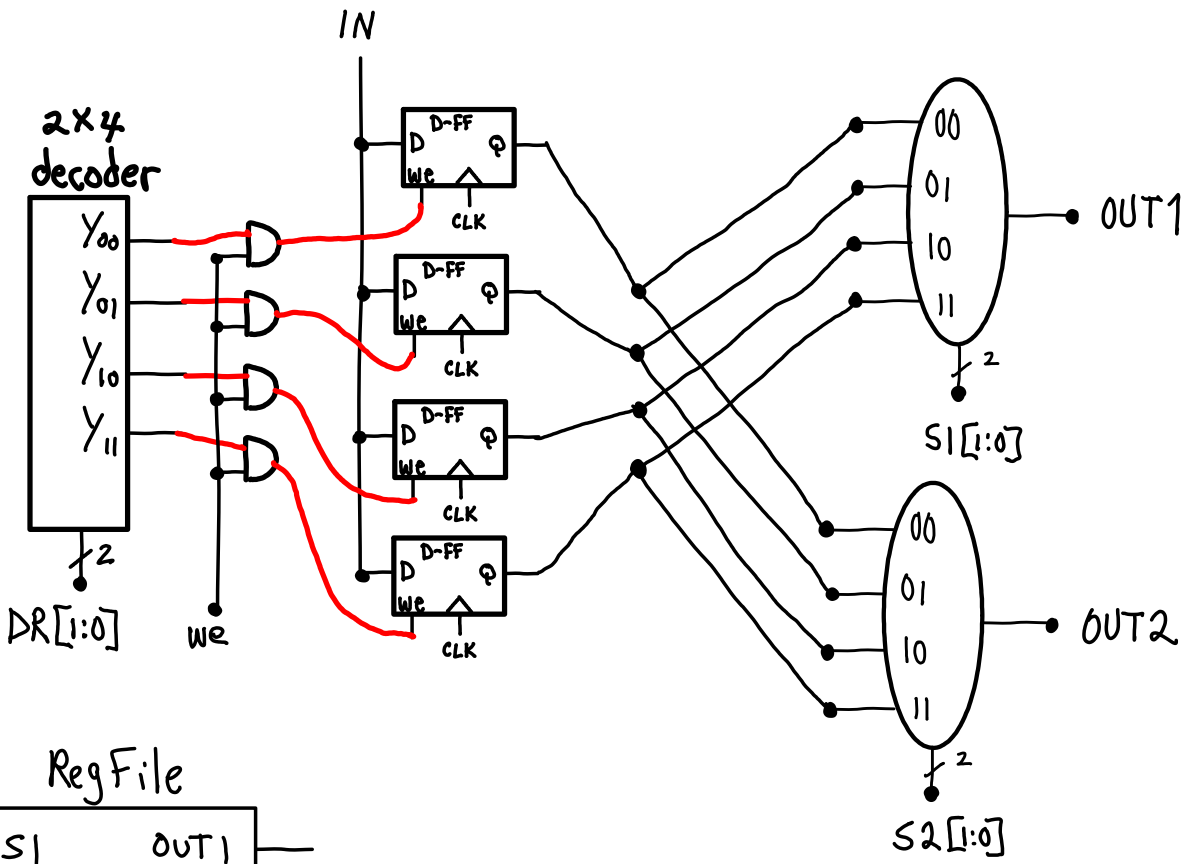
Here's a R/W 4-word memory. One address controls both read select and the write select.

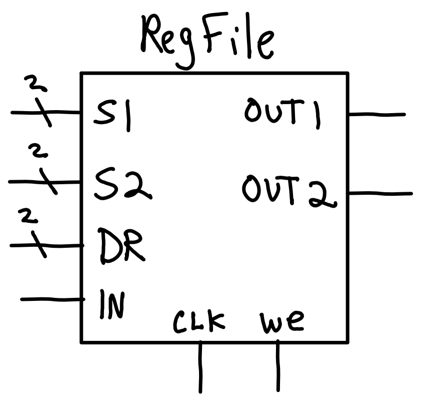


addr	content
00	$Word_{00}$
01	$Word_{01}$
10	$Word_{10}$
11	$Word_{11}$



Here's a 4-register RegFile. It has three independent select inputs (DR, S1, S2), on for writing and two for reading. All three are done simultaneously (unless we == 0, then just two register reads).



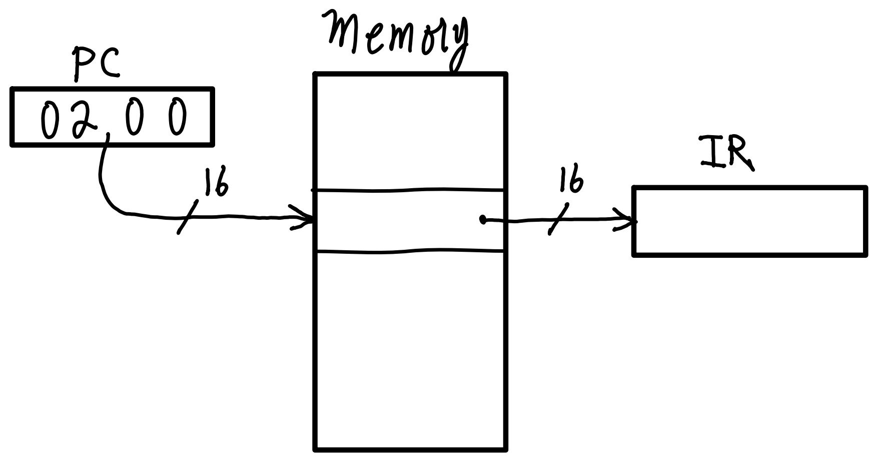


For both the memory and RegFile above, words or registers were 1-bit. But we could instead use k-bit flipflops and muxes. Data inputs and outputs would then be k bits each. We would have k-bit memory words or k-bit registers. LC3 and LC4 have k=16. Other machines have k equal to 4, 8, 16, 32, 64, 128, or other sizes.

instruction processing

Now that we are clear on what registers, memory, and register files are, we can discuss instruction processing. The LC3 and LC4 differ in that the LC3 must devote a series of control states to manage the steps of processing an instruction while LC4 does all the steps in one pass. However, the basic phases of instruction processing are the same, and use essentially the same hardware. The difference in hardware is mostly due to the fact that the LC3 has to use intermediate registers, such as the MAR, MDR, and IR.

instruction fetch (all numbers shown are in hex notation)

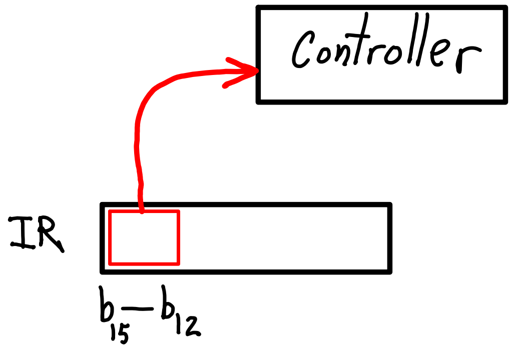


Instruction fetch is simply the process whereby the content of the PC is used as an address to access memory. We say the PC points to a particular instruction. In this case, the memory word at address 0200.

The content of the memory word at that address is passed from the memory to the IR (the IR is a bus on the LC4, it is a register on the LC3).

Addresses and memory words are both 16 bits. Some machines do not use the same number of bits for both.

instruction decode

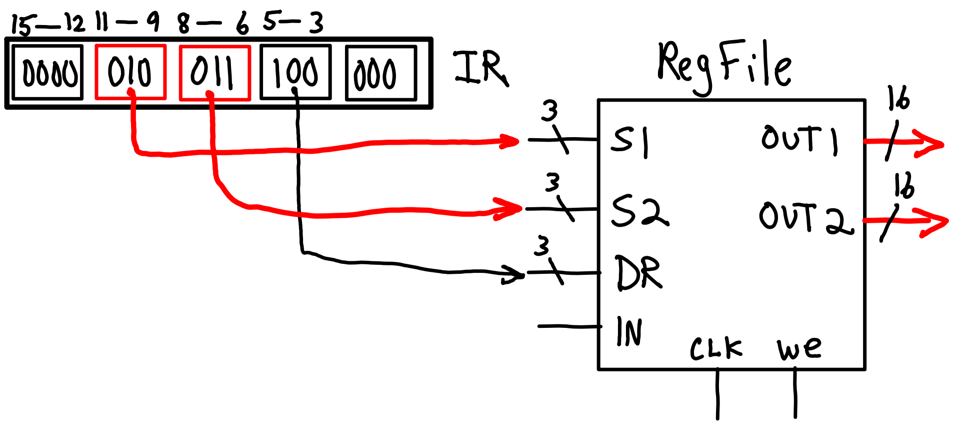


Instruction decode amounts to telling the controller what instruction has been fetched. This is done by sending the opcode of the instruction to the controller's inputs. For the LC3 and LC4 the opcode is the most-significant 4 bits.

The signals that control the datapath are then generated by the controller to cause the appropriate actions to occur.

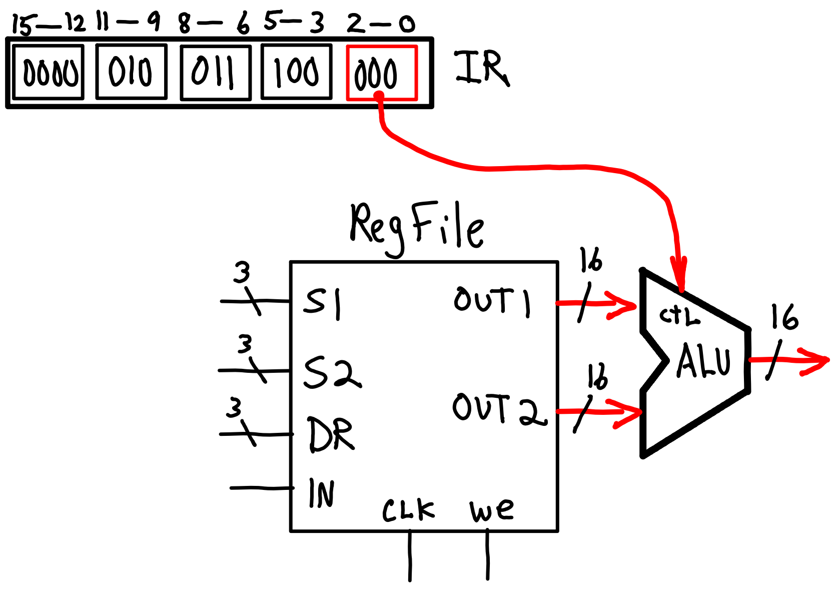
The LC4 generates all the signals at once. The LC3 controller goes through a series of states, generating a portion of the signals in each state.

operand fetch



Operand fetch amounts to selecting which registers' contents will appear on the outputs OUT1 and OUT2. In this case, the instruction is LC4's
ALU SR2 SR3 DR4 ADD
In this case R2's content is at OUT1 and R3's content is at OUT2. Which bits the controller routes to the source selects (S1 and S2) is determined by the instruction's format. Control signals control muxes for that in the LC3. The LC4 always uses the same bit fields for this; so, muxes are not needed for this in the LC4.

execute phase

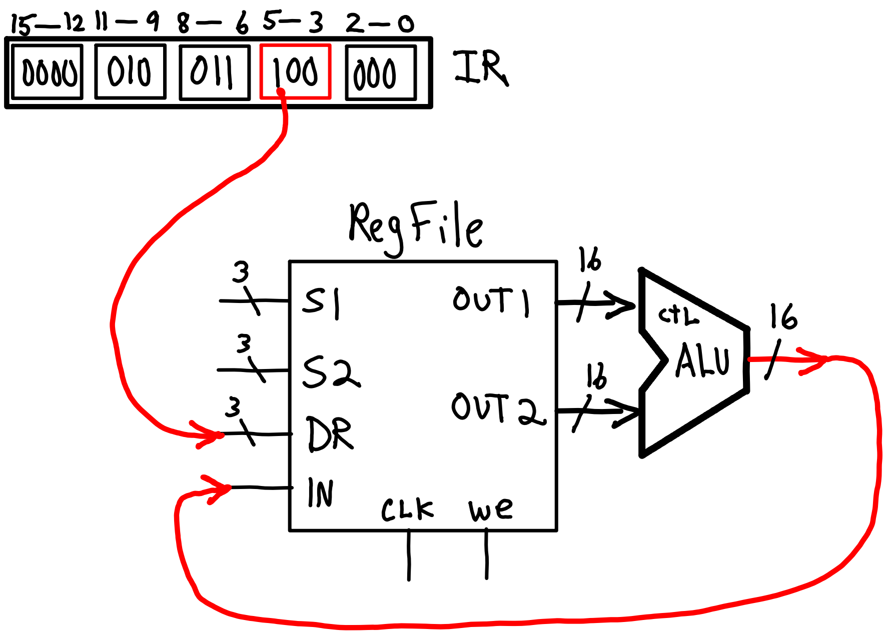


Execute phase for LC4's ALU instruction consists of controlling the ALU using the bits IR[2:0]. In this case, IR[2:0] selects the ALU operation to be ADD. The three-bit select allows for eight ALU operations.

The LC3 uses the opcode itself to determine the ALU's operation. This is one reason the LC3 has only three ALU operations.

Most instructions do not include the execute phase. E.g., loads and stores do not, unless you want to call accessing memory for data read or write "execution phase". We prefer to call that "memory access phase."

STORE—RESULT PHASE



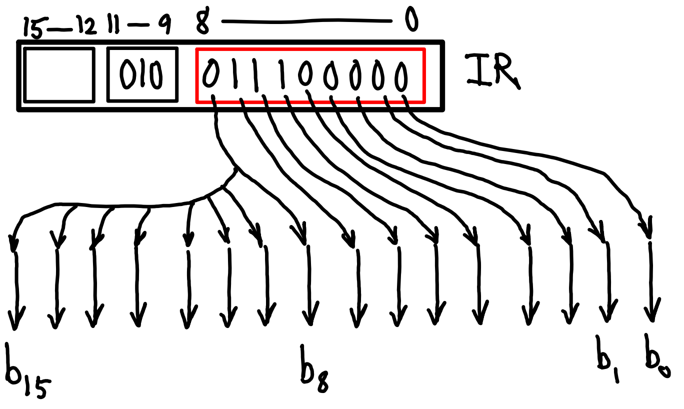
Store-result phase for LC4's ALU instruction routes the ALU's output to the RegFile's IN via a mux (mux not shown). Which register will be written into is controlled by the destination register bit-field in the IR, IR[5:3] in this case.

Other LC4 instructions use IR[11:9] to select the destination register; e.g., LDR.

LC3 also has a mux on its RegFile DR select input, which steers the appropriate IR bits to select which register is written.

Of course, the controller must set the RegFile.we = 1. The next 0-to-1 transition of the CLK signal (positive edge) will cause the register to change its state to be value coming into the IN input.

immediate data



Some instructions use bits within the instruction itself as data. Called "immediate data", these bits are sign-extended to a full word by copying the MSb. The LC4 LIM (load immediate) sign-extends IR[8:0] by copying IR[8] seven times. The result is steered to RegFile.IN and written to a register. The data is encoded in 2s-complement form.

Sign-extending preserves the value of the immediate data. Copying leading zeroes does not change the value of a positive 2s-complement encoded number. It turns out that copying leading ones does not change a negative 2s-complement encoded number either.

2s-complement encoding

1-bit encoding:

code	value (in decimal)
0	0
1	-1

2-bit encoding:

code	value (decimal)
01	+1
00	0
11	-1
10	-2

3-bit encoding:

code	value (decimal)
011	+3
010	+2
001	+1
000	0
111	-1
110	-2
101	-3
100	-4

4-bit encoding:

code	value (decimal)
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

The pattern shown above continues for 2s-complement encodings with more than four bits. It turns out that doing unsigned binary addition with these codes yields codes that represent the correctly encoded result value. E.g. 001 + 111 = 000, if you ignore the carry, encodes (+1) + (-1) == (0). Of course, not every result can be correct, in which case we say the operation "overflowed". Note that all negative values have encodings with MSb == 1, all non-negatives have MSb == 0.

A simple rule for converting a positive value encoded in 2s-complement form to its negative as a 2s-complete code is as follows:

Invert all the bits, then add one.

E.g., 001 ==> 110+1 == 111, i.e., (+1) ==> (-1).

That will also do the reverse, turn a negative to a positive.

branching and jumping

"Branching" and "jumping" are both instruction operations that load the PC with a new value. By default, the PC is loaded with whatever it had previously, plus 1. Branching loads it with a "target address" so that the next instruction fetch is not from the next word in memory but rather some other place. For the LC4, the PC is loaded with the target address if OUT1[15] = 1.

On the LC4, the target address is held in a register whose content is on OUT2, which is steered to the PC in the case that the branch is "taken", i.e. OUT1[15] = 1 and the instruction is BRR. E.g., BRR CR1 AR2 will load the PC with the content of R2 if bit R1[15] == 1.

On the LC3, the target address is formed by sign-extending IR[8:0] and adding it to the PC content. There are eight LC3 branch instructions BR_000, BR_001, BR_010, BR_011, BR_100, BR_101, BR_110, and BR_111 (aka, NOP, BRp, BRz, BRzp, BRn, BRnp, BRnz, and BRnzp). They branch depending on the value last written into a register. If the last value was negative BRn, BRnz, and BRnp will all take the branch; if zero, BRnz, BRz, BRzp will; if positive, BRnp, BRzp, and BRp will. BRnzp always branches, NOP never branches.

control states

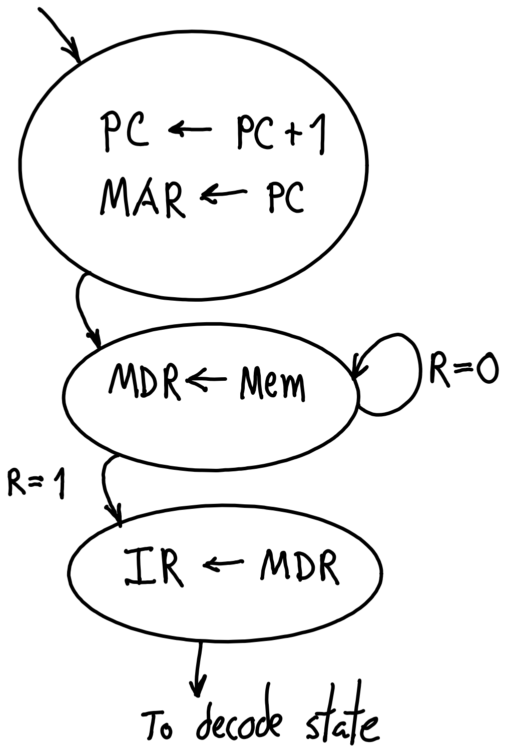
Each state determines what step of the operation is being done and produces differing control signals to cause the appropriate actions to occur. LC4 has 6 states, one for each opcode, LC3 has 59. LC3 requires several states to execute a single instruction; LC4 uses one state per instruction.

Each state can be described by a Register-Transfer-Language statement. E.g.,

MDR <=== MEM

states that the LC3's memory output is written into the LC3's MDR register. All RTL operations within a state happen simultaneously.

LC3's instruction fetch states



Each transition from one state to the next happens at the positive clock edge.

The register writes occur at the positive clock edge, also.

When the LC3 is in the 2nd state of instruction fetch, memory has to send a ready signal, R ==1, before the control changes to the third state. As long as R == 0, it continues to transition back to the 2nd state.

After LC3's decode state, some chain of states is transitioned through to effect the instructions execution. Some instructions require only one state for execution while others may require as many as seven states.