Notes on the interconnection among registers

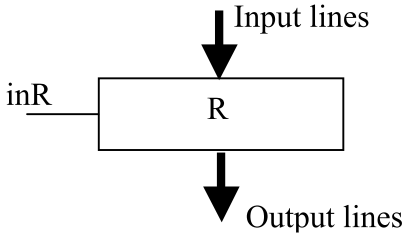
Annalisa Massini

revision by Daniele Gorla

I. Registers

A memory cell, able to store all the *k* bits of a word (that is an indivisible information unit, usually made up of 8, 16, 32 or 64 bits), is called *register*. It is made up by *k* elementary memory cells (FF), each containing one information bit. In these notes we shall work with FFs of kind SR, but everything can be easily adapted to other kinds of FFs.

Diagrammatically, we shall represent a registe ras follows:



The thick arrow denotes a set of *k* lines, one for every elementary cell (FF) of the register; hence, we are auuming here a PIPO register. Line *inR* is used to simultaneously enable all the *k* elementary cells to writing.

II. Register Interconnection

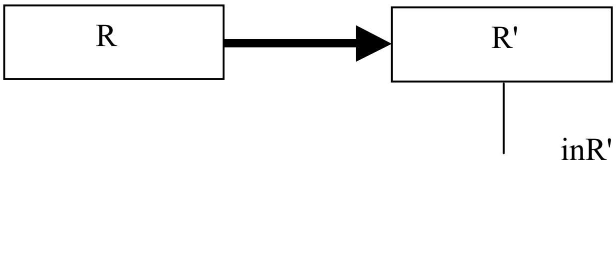
Moving info among the various registers is done through *interconnection nets* that allow to move data in the computing modules or in other regions of the memory. To be precise, it would be better to speak about copying info, and not moving, since the content of the source register remains the same and a copy is stored into the destination register.

We can distinguish 4 interconnection modalities, obtained according to whether the source and the destination are fixed or variable:

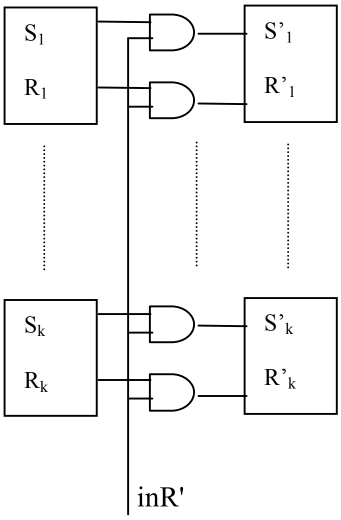
	Fixed destination	**Variable destination**
Fixed source	point-to-point (logic gates or tri-state buffers)	DECODER
Variable source	Multiplexer	mesh or bus

1. Fixed Source and Destination: one-to-one interconnection

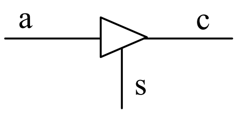
This interconnection allows to copy the content of a given source register R into a given destination register R'.



Every time we have to move info from R to R', line inR' must be set. In the following schema, inR' acts on the AND gates and enables the transfer:



Instead of logic gates, we can also use a *tri-state buffer*, that is an electronic switch drawn in the following way:



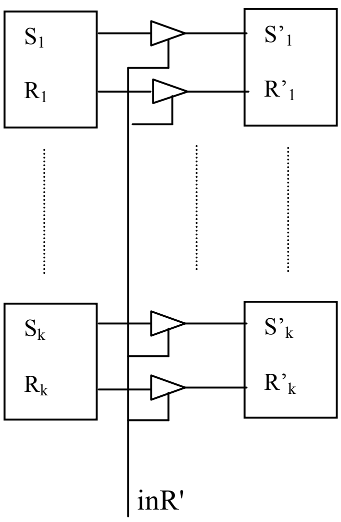
When the control signal s of the buffer is:

- 0 the impedence between input and output of the buffer is very high and so the switch is open (i.e., the link between the input and output is "cut");
- 1 the impedence is negligible, and so a and c are directly linked and the input is given in output:
  - the value of c is 0 if a is 0
  - the value of c is 1 if a is 1.

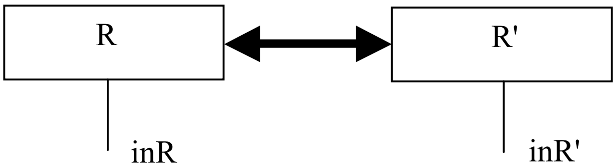
The device can hence assume three state (from here the name):

- open switch: s = 0
- closed switch and output 0: if s=1 and c=0
- closed switch and output 1: if s=1 and c=1.

In the following schema, inR' (that now plays the role of the previous signal s) is now used to control all buffers between the FFs of the two registers:



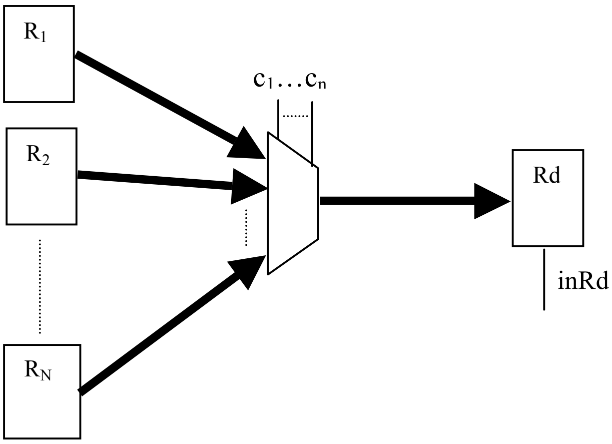
We can also design a net that allows for the bidirectional transfer of info between R and R'; in this case, we should also provide R with a control line inR:



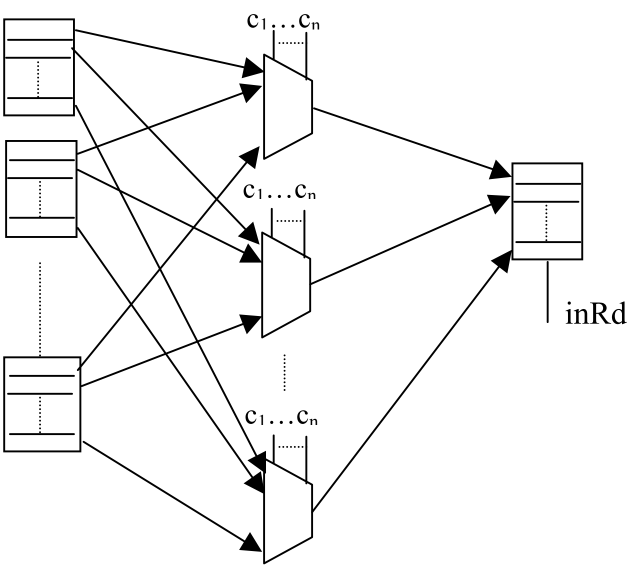
The implementation details (both with logic gates and with buffers) are an easy exercise and left to the reader.

2. Variable source and fixed destination: many-to-one interconnection with a multiplexer

The source register R_i is any of a set of N registers; the destination Rd is given:



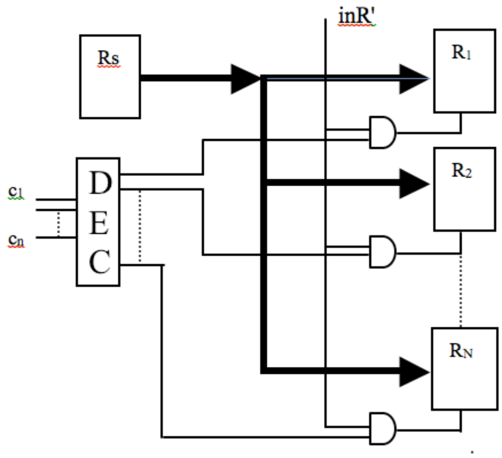
The control lines of the MUX $c_1, \ldots, c_n$ (where n is the superior integer part of $\log_2 N$) provide the binary encoding of the index i of register R_i whose content must be copied into Rd. The above MUX, with thick inputs and output, actually denotes a set of *k* MUXs, one for each of the k FFs of the registers:



The first FF of every source register is connected with the first MUX, whose output goes to the first FF of Rd; the second FF of every source register is connected with the second MUX, whose output goes to the second FF of Rd; and so on until the last FF. The selection lines $c_1, \ldots, c_n$ hold the same value for each MUX since they have to select one by one the bits of the same source register (the binary sequence $c_1 c_2 \ldots c_n$ is the index i of the selected source register).

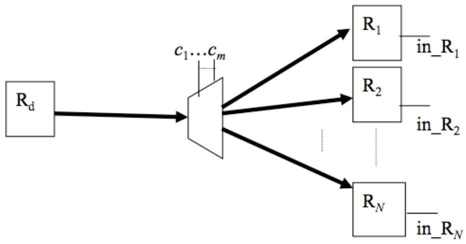
3. Fixed Source and variable destination: one-to-m interconnection with decoder

The source Rs is fixed, whereas the destination can be any R_i of a set of N registers, that is selected by a decoder when a control line inR holds 1:



The control line inR' enables writing in one of the N destinations. The actual destination (that receives the content of register Rs) is register R_j, where j is binary encoded by the n selection lines $c_1 \ldots c_n$, inputs of the decoder; only the j-th output of the decoder is set.

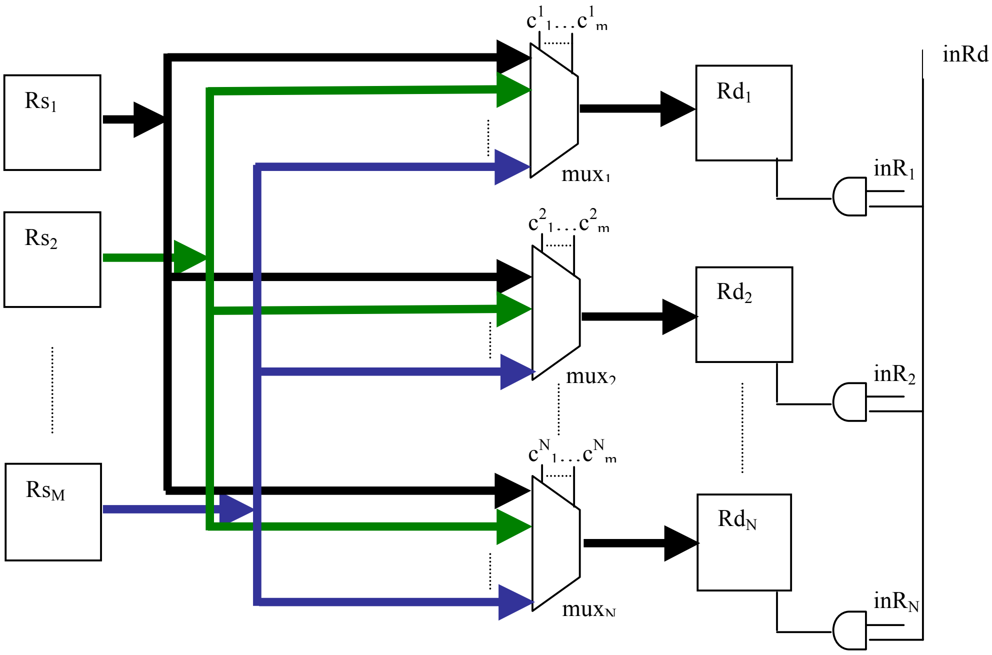
It could be tempting to realize the one-to-many interconnection with a DEMUX (dually w.r.t. the many-to-one interconnection), i.e. something like this:



However, this does NOT work well; indeed, the the non-selected lines of the DEMUX contain a 0 that, if not properly controlled, would put 0 in all non-selected destinations. Controlling the writing in the destinations requires also in this case a DEC, for properly setting lines in_R_i. Hence, the DEMUX would be useless!

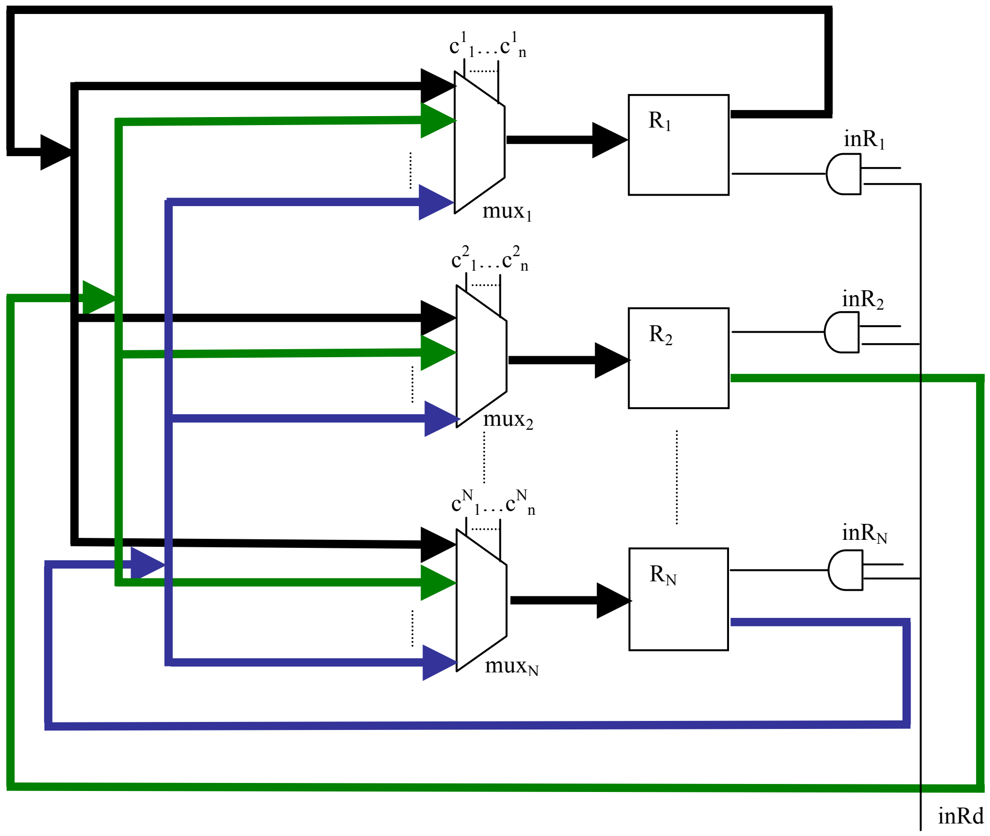
4a. Variable source and destination: many-to-many interconnection through a mesh

The most complex case is when we have to interconnect M source registers with N destinations:



To realize the net we need N multiplexer, mux_i, one for every destination (actually, each of these MUXs represents *k* one-bit MUXs, one for every FF of the registers: this is analogous to the case with variable source and fixed destination). MUX*i* is controlled by lines $c^i_1 \ldots c^i_m$ (where m is the superior integer part of $\log_2$ M) and are used to select one of the M sources Rs_j by providing the binary encoding of the index *j* of the register whose content must be copied. The destination register Rd_h that should receive the datum is enabled by the signal $inRd_h$, that is put in AND with a global signal inRd that enables all transfers.

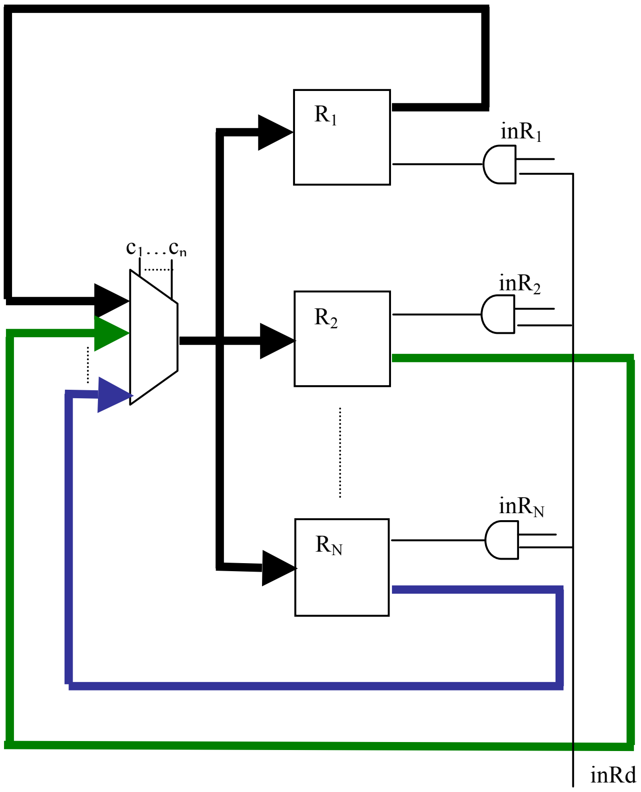
If we remove the distinction between source and destination registers, the interconnection net is becomes:



Again, the writing into a register is obtained through the AND between the global signal inRd and inR_i (the writing signal for the specific register *i*). The selection of the source register for destination R_i is done through the control lines of mux_i.

Conceptually a mesh is quite easy (it is obtained via several many-to-one interconnections). Its drawback lies in the cost: indeed, using a high number of registers would require an unaffordable number of gates. As an example, try to estimate the number of gates necessary for building a mesh like the previous one with 128 32-bits registers.

A cheaper kind of mesh. To reduce costs, we can use just one MUX (always remember that they are *k*, one for every FF of the registers):

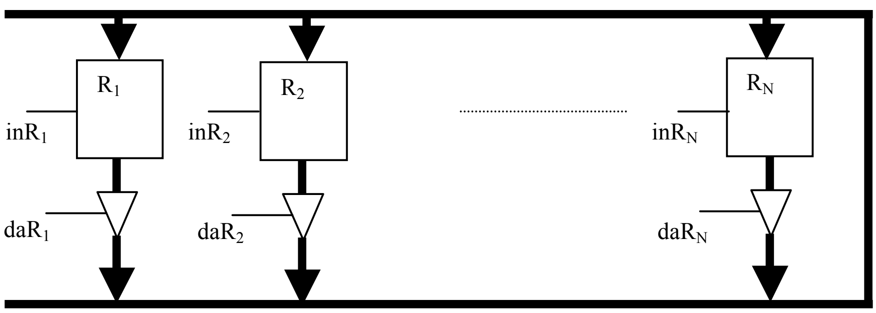


With this net, the content of R_i is output by the MUX if the control lines $c_1 \ldots c_n$ provide the binary encoding of index *i*. The content of the source register is copied into the destination R_j whenever inR_j and inRd both hold 1.

The advantage of this schema with respect to the previous one is the number of needed gates (much less MUXs here!). The price to be paid is that in this way we loose the possibility of having parallel transfers, that by contrast was enabled by the original schema (the one with a MUX for every destination).

4b. Variable source and destination: many-to-many interconnection through a bus

If we accept non-parallel transfers, we can obtain an even cheaper interconnection by using a bus (i.e. a series of bit lines) and the use of tri-state buffers:



The interconnection is realized by using *k* lines (where *k* is the number of FFs of the registers): the *bus*. Registers' inputs directly come from the bus, whereas their outputs go into the bus by oassing through a (series of *k*) tri-state buffer. To copy the content of R_i into R_j we have to set line $outR_i$ of the *i*-th tri-state buffer and line inR_j of the *j*-th register.

4c. Many-to-many: Mesh vs bus

Within the microprocessor, we have a set of few very quick registers realized with expensive technology, whereas the central memory is made up by milions of registers realized with a much cheaper technology. Microprocessor's registers are interconnected through a mesh, whereas such registers are interconnected with the central memory through a bus. Buses are largely used in a computer because they allow to use a small number of point-to-point connections for interconnectiong all registers of the machine.

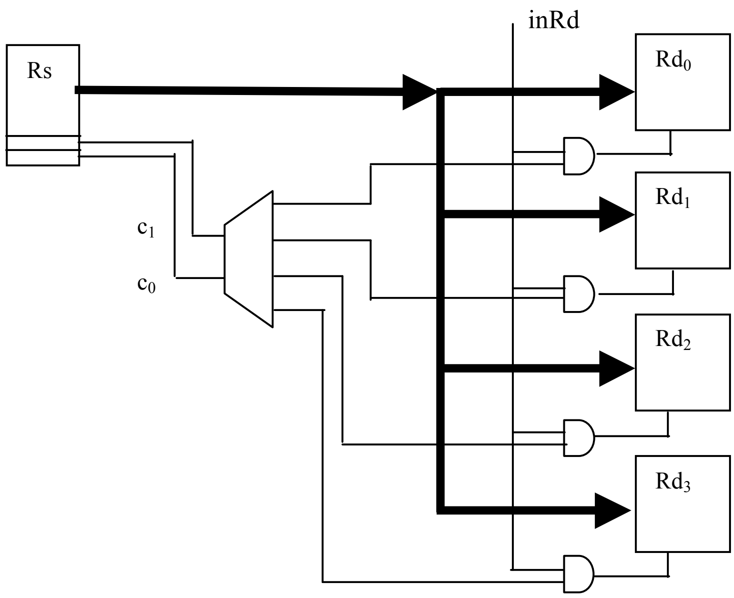
III. Design of interconnection nets

Designing an interconnection net requires solving two main issues:
- decide the needed links for the desired transfers;
- design the circuits that correctly activate the control lines, according to the specifications.

Let us see a few examples on how this is done in practice.

Example 1. Let Rs be a source register and let Rd_0, Rd_1, Rd_2 and Rd_3 be four destination registers. Design the interconnection net such that, when in_Rd holds 1, the content of Rs is moved into Rd_j, where *j* is given by the binary number resulting from the two less signifying bits of Rs.

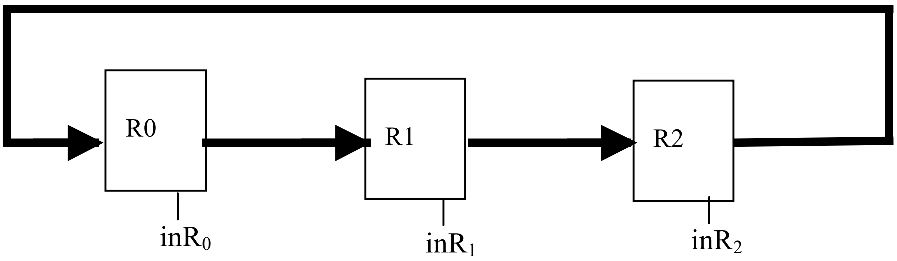
Solution
This is a one-to-many interconnection whose schema is:



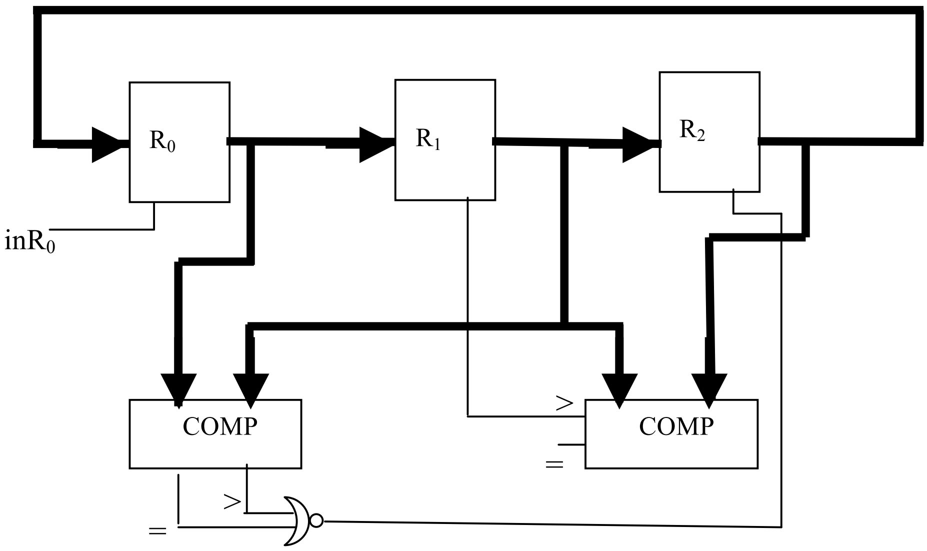
Notice that the control signals of the DEC are the two LSBs of the source register, as specified by the text.

Example 2. Design an inteconnection between R_0, R_1 and R_2 such that:
- R_0 is moved into R_1 if $R_1 > R_2$;
- R_1 is moved into R_2 if $R_0 < R_1$;
- R_2 is moved into R_0 if $R_0 = R_1 \mid R_2$ (where | denotes the bitwise OR).

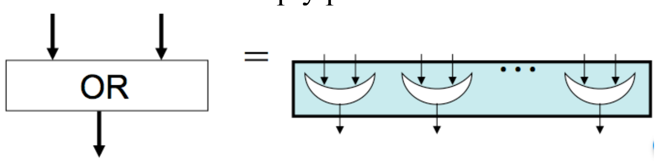
Solution
The interconnection net is obtained by joining three one-to-one transfers:



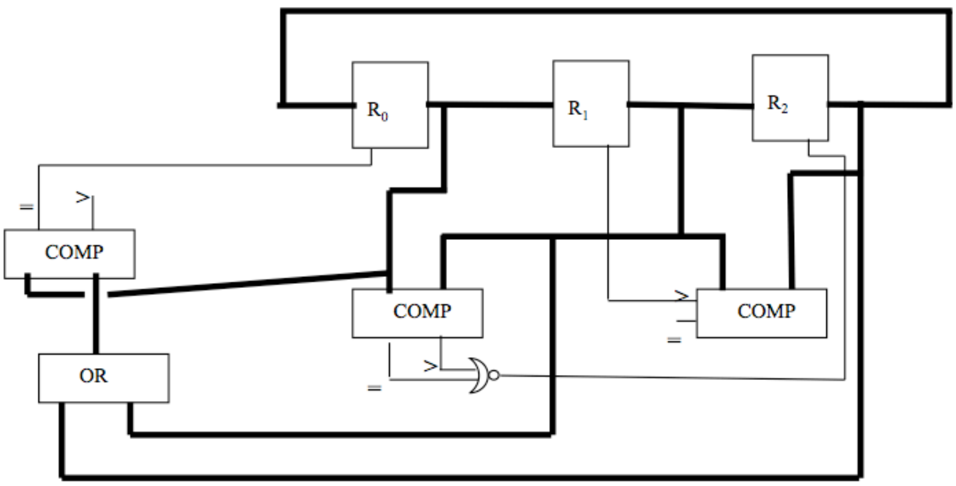
We then need to design the circuit that computes the signals inR_0, inR_1 and inR_2. The conditions for the first two interconnections suggest to use two comparators:



For in_R_0, let us observe that the bitwise OR is simply performed as follows:



By using it, we have:



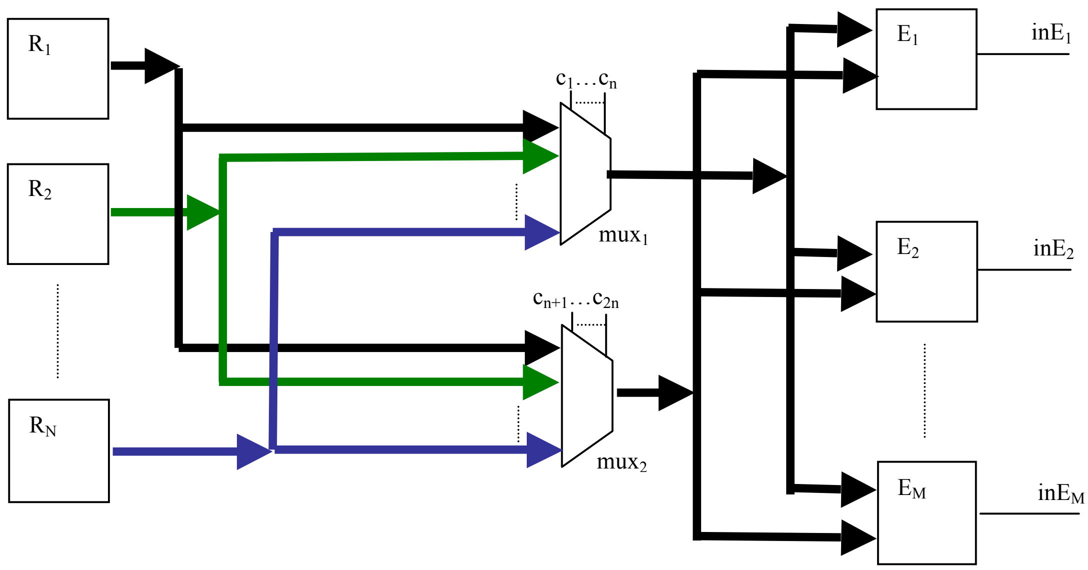
Example 3 (A simplified ALU). Design a many-to-many interconnection net that allows to move the content of two among N k-bits registers $R_1...R_N$ to one between M computing modules (with two inputs) $E_1....E_M$.
Draw the black-box schema with all control circuits necessary for:

- selecting 2 among the N source registers (i.e., the operands);
- applying to them the functionality of one of the M computing modules.

Then, draw the detailed schema (up-to the level of FFs and logic gates) when we have:

- 3 source registers that store 2 bits (N=3, k=2), with FFs of kind JK;
- 2 computing modules (M=2), one of which is an adder and the other one that computes the bitwise AND of the input operators.

Solution
The general schema is the following:

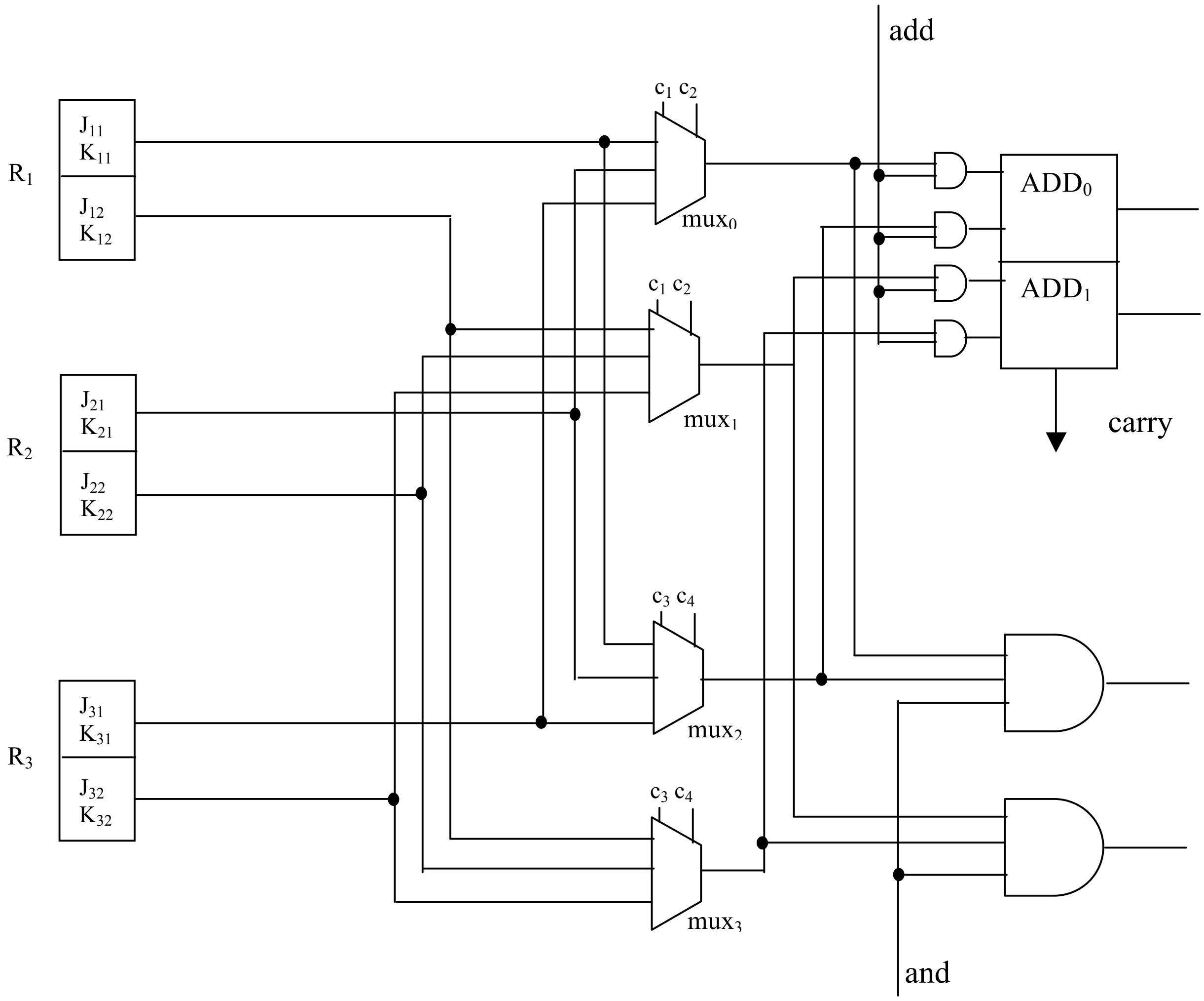


We have a (set of k) MUX for every operand in input to the computing modules: MUX1 selects the first operand among the N source registers through the control signals $c_1....c_n$; MUX2 selects the second operand through the control signals $c_{n+1}...c_{2n}$, where, as usual, n is the upper integer part of $\log_2 N$. Every computing module has a control signal in_E_j that enables reading and computing the operands previously stored in the input lines of module j.

The detailed schema for the specific case required has 4 MUXs (with single-bit lines):
- mux_0 selects the LSB of the first operand;
- mux_1 selects the MSB of the first operand;
- mux_2 selects the LSB of the second operand;
- mux_3 selects the MSB of the second operand.

Hence, mux_0 and mux_1, since they must both select the first operand, are both controlled by the same lines c_1 and c_2; similalry, mux_2 and mux_3, since they must both select the second operand, are both controlled by the same lines c_3 and c_4. The operation is chosen by setting line *add* (corresponding to inE_1) and *and* (corresponding to inE_2). Finally, *carry* is used to signal a possible final carry of the adder.

The resulting circuit is the following:


add
R1
J11
K11
J12
K12
c1 c2
mux0
c1 c2
mux1
R2
J21
K21
J22
K22
c3 c4
mux2
R3
J31
K31
J32
K32
c3 c4
mux3
ADD0
ADD1
carry
and