Lecture 26 – Processor design 1

The Hamming code

- A 0 check bit designates even parity over the checked bits and a 1 designates odd parity
- Since the bits were stored with even parity, the result:

$$C = C_8C_4C_2C_1 = 0000$$

indicates that no error has occurred

- Here is some magic: *However, if $C \neq 0$, then the 4-bit binary number formed by the check bits gives the position of the erroneous bit!*

While storing:

Bit position:	1	2	3	4	5	6	7	8	9	10	11	12
	P_1	P_2	1	P_4	1	0	0	P_8	0	1	0	0

$$P_1 = \text{XOR of bits } (3, 5, 7, 9, 11) = 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 = 0$$

$$P_2 = \text{XOR of bits } (3, 6, 7, 10, 11) = 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 = 0$$

$$P_4 = \text{XOR of bits } (5, 6, 7, 12) = 1 \oplus 0 \oplus 0 \oplus 0 = 1$$

$$P_8 = \text{XOR of bits } (9, 10, 11, 12) = 0 \oplus 1 \oplus 0 \oplus 0 = 1$$

While reading:

	0	0	1	1	1	0	0	1	0	1	0	0
Bit position:	1	2	3	4	5	6	7	8	9	10	11	12

$$C_1 = \text{XOR of bits } (1, 3, 5, 7, 9, 11)$$

$$C_2 = \text{XOR of bits } (2, 3, 6, 7, 10, 11)$$

$$C_4 = \text{XOR of bits } (4, 5, 6, 7, 12)$$

$$C_8 = \text{XOR of bits } (8, 9, 10, 11, 12)$$

The Hamming code

Detecting the position of erroneous bit

Bit position:	1	2	3	4	5	6	7	8	9	10	11	12	
	0	0	1	1	1	0	0	1	0	1	0	0	No error
	1	0	1	1	1	0	0	1	0	1	0	0	Error in bit 1
	0	0	1	1	0	0	0	1	0	1	0	0	Error in bit 5

	C_8	C_4	C_2	C_1
For no error:	0	0	0	0
With error in bit 1:	0	0	0	1
With error in bit 5:	0	1	0	1

C_1 = XOR of bits (1, 3, 5, 7, 9, 11)

C_2 = XOR of bits (2, 3, 6, 7, 10, 11)

C_4 = XOR of bits (4, 5, 6, 7, 12)

C_8 = XOR of bits (8, 9, 10, 11, 12)

Hamming code

- The Hamming code can be used for data words of any length.
- Hamming code consists of *k check bits* and *n data bits*, for a total of n + k bits.
- The syndrome value C consists of *k* bits and has a range of 2^k values between 0 and $2^k - 1$.
- One of these values, usually zero, is used to indicate that no error was detected, leaving $2^k - 1$ values to indicate which of the n + k bits was in error.
- Each of these $2^k - 1$ values can be used to uniquely describe a bit in error.
- Therefore, $2^k - 1 >= n + k$
- Solving for n in terms of k, we obtain $2^k - 1 - k >= n$
  - *Eg: k=4 check bits ==> n <= 11 data bits*

Single correct, double detect

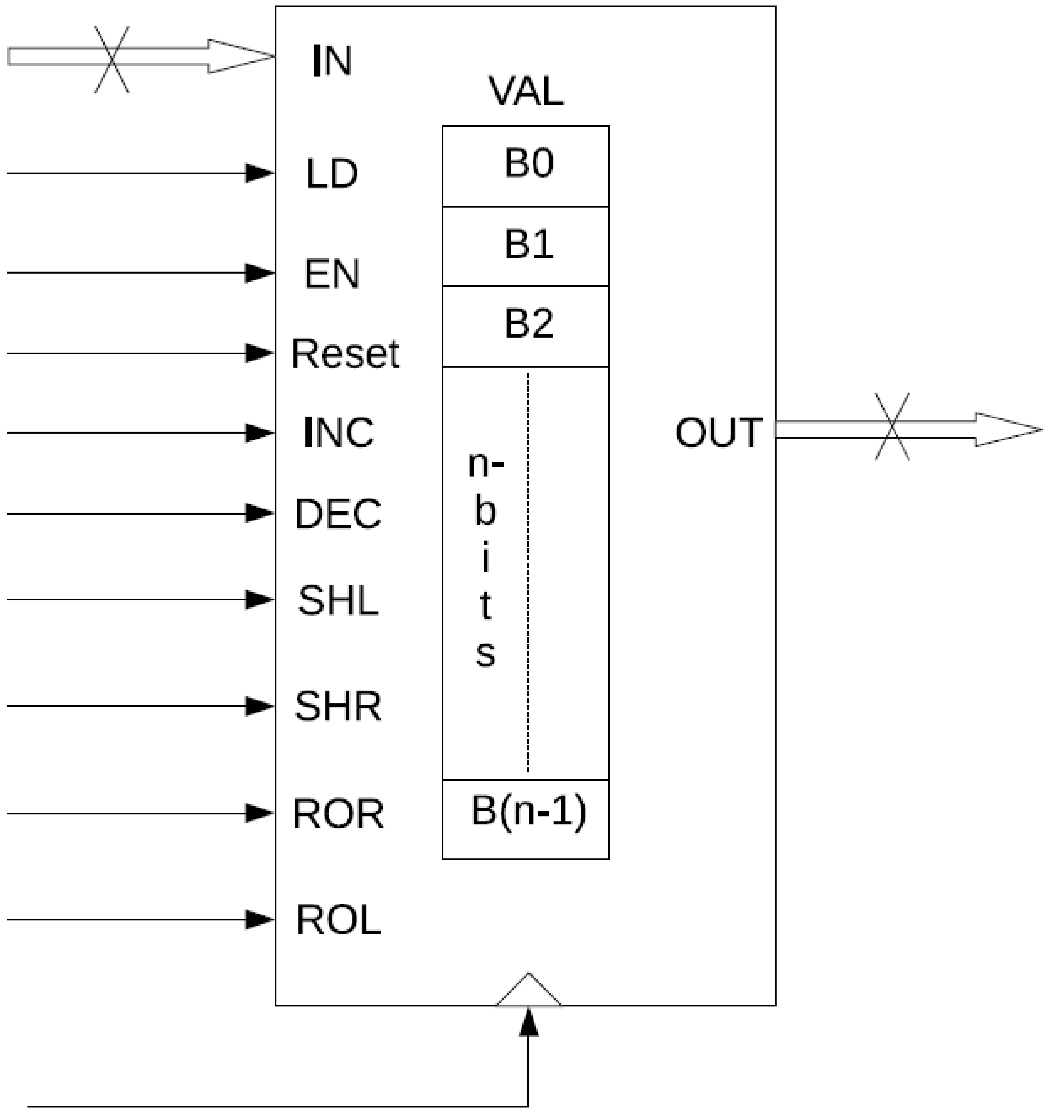
- The Hamming code can detect and correct only a single error
- By adding another parity bit to the coded word, the Hamming code can be used to correct a single error and detect double errors
- If we include this additional parity bit, then the previous 12-bit coded word becomes 001110010100P_{13}, where P_{13} is evaluated from the exclusive-OR of the other 12 bits
- This produces the 13-bit word 0011100101001 (even parity)
- When the 13-bit word is read from memory, the check bits are evaluated, as is the parity P over the entire 13 bits
- The following four cases can arise:

1. If $C = 0$ and $P = 0$, no error occurred
2. If $C = 0$ and $P = 1$, an error occurred in the P_{13} bit!
3. If $C \neq 0$ and $P = 1$, a single error occurred that can be corrected
4. If $C \neq 0$ and $P = 0$, a double error occurred, but that cannot be corrected

Processor design 1

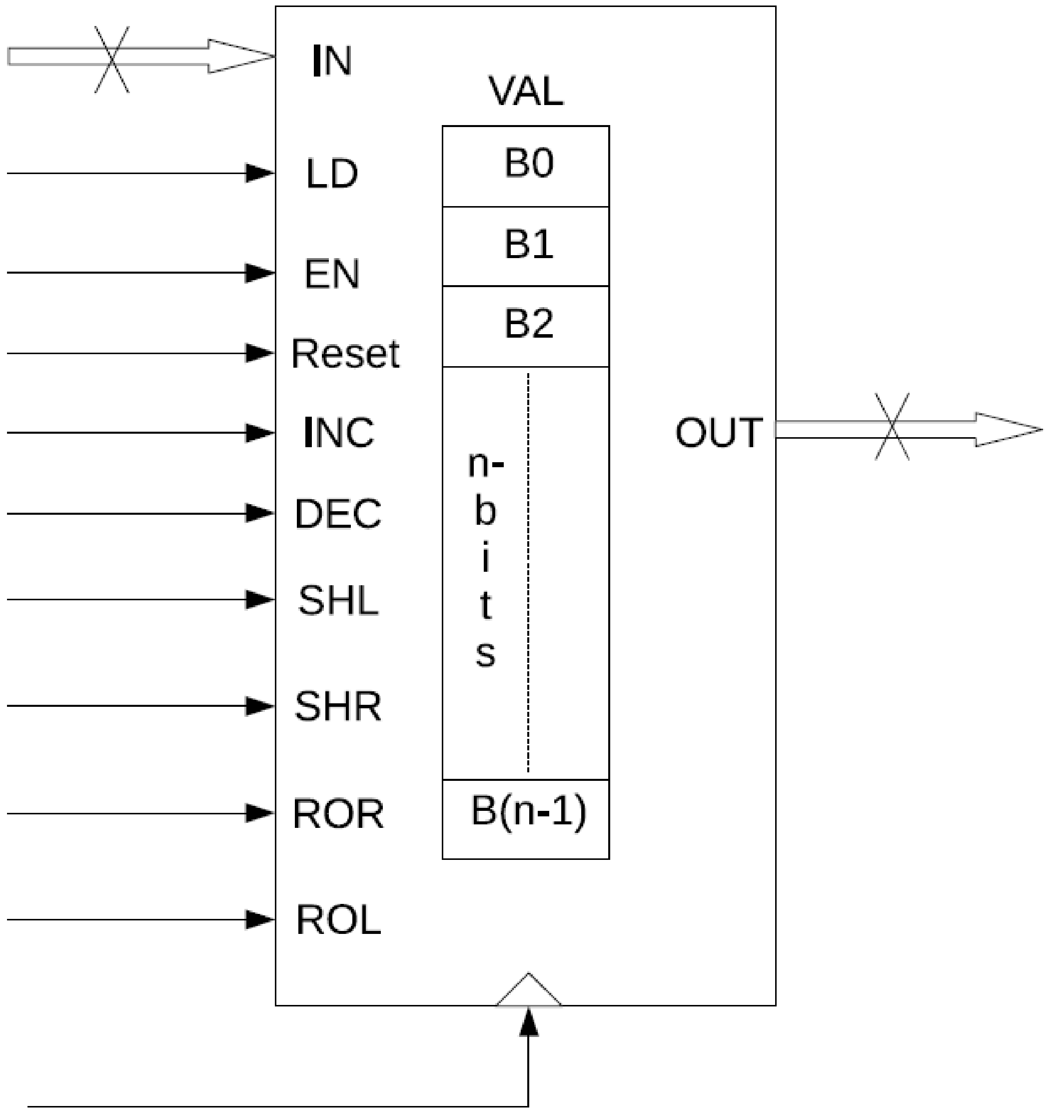
Multipurpose register

- A multipurpose/universal register is a very useful entity in processor design
- It can store value as well as have some basic arithmetic functions such as increment/decrement etc.
- The register has two input/output lines (sometimes coupled into one IO bus)
- Input lines IN are logically connected internally to the inputs of the n flip-flops inside the register
- Output lines OUT are the lines that hold the value stored in the register when the register is accessed for read. The OUT lines are tristate-capable so that they can be connected to a common data bus



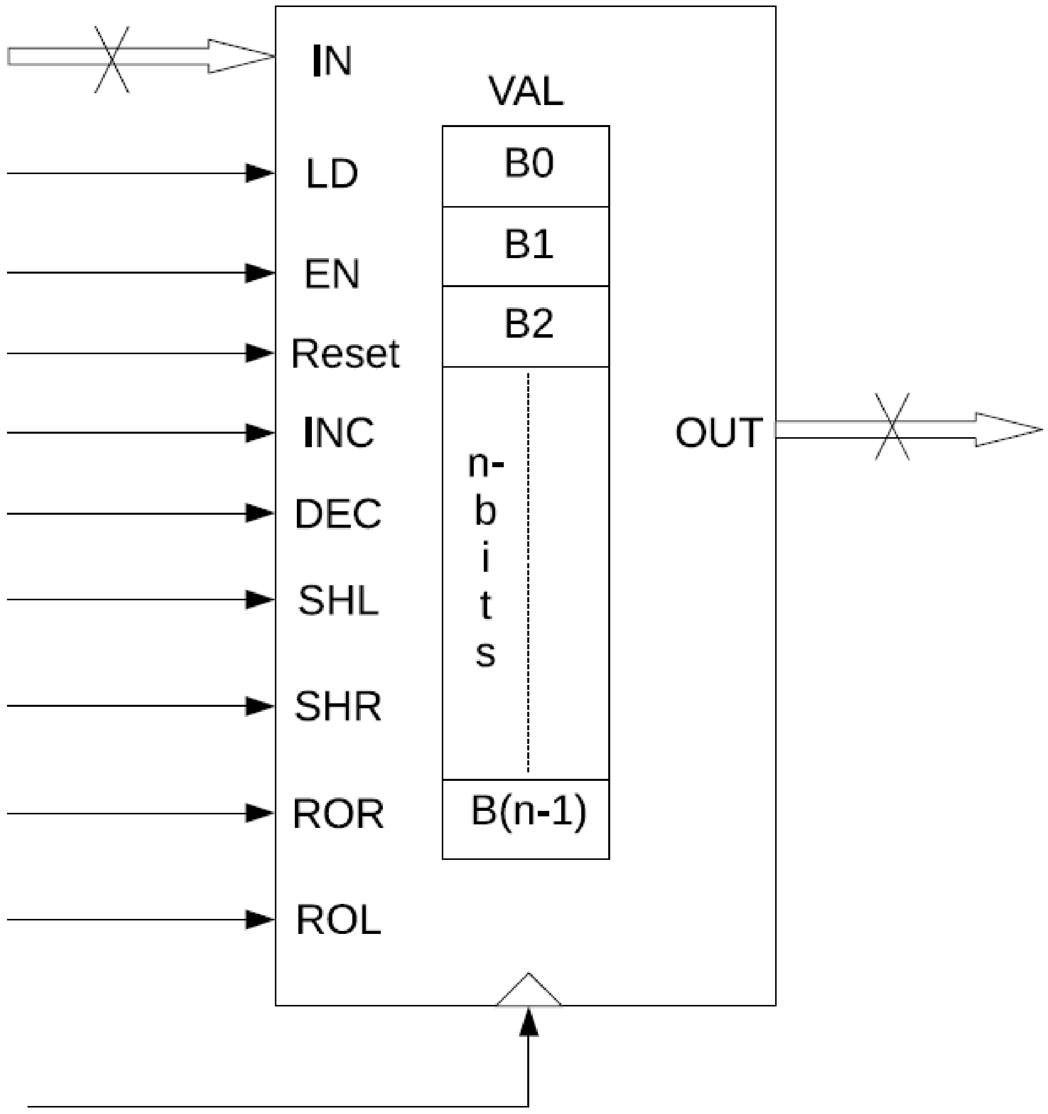
Multipurpose register

- The control bits are as follows:
  - The input EN controls the high-impedence state of the output lines OUT. When EN is 0, the OUT lines will be in the high-impedence state. *When EN is 1, the OUT lines will be driven to the electric levels corresponding to the value stored in the register*
  - The input LD loads the register from the input, which changes the value stored in it. *When LD is 1, the value indicated by the electric levels of the IN lines will replace the previous value stored in the register*
  - The input RESET controls resetting of the register. *When RESET is 1, a 0 value will be written to all bits stored in the register. We will assume a synchronous reset.*



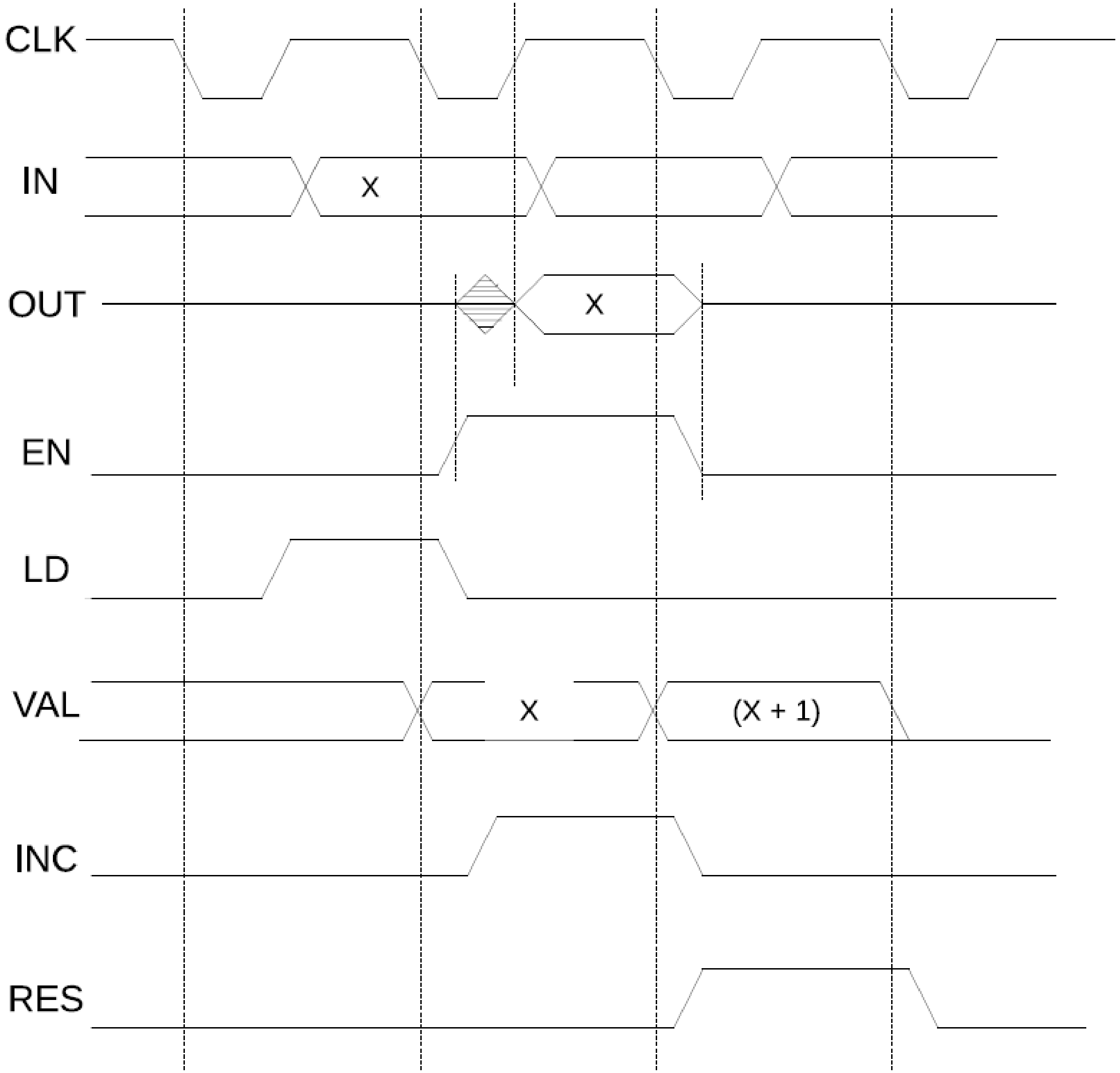
Multipurpose register

- The control bits are as follows:
  - Inputs INC and DEC control the incrementing or decrementing the value stored in the register, interpreted as a number.
  - Inputs ROL, ROR control the left or right rotation of the register contents. This is shifting such that the last bit is feedback to the first bit
  - Inputs SHL, SHR control the left or right shift of the register contents. We assume a 0 will fill the void during the shift



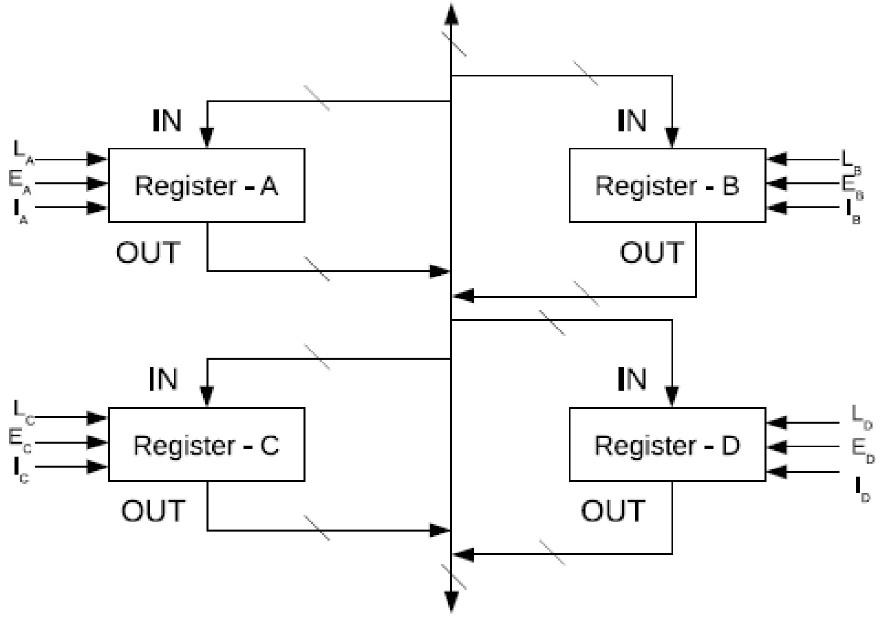
Multipurpose register

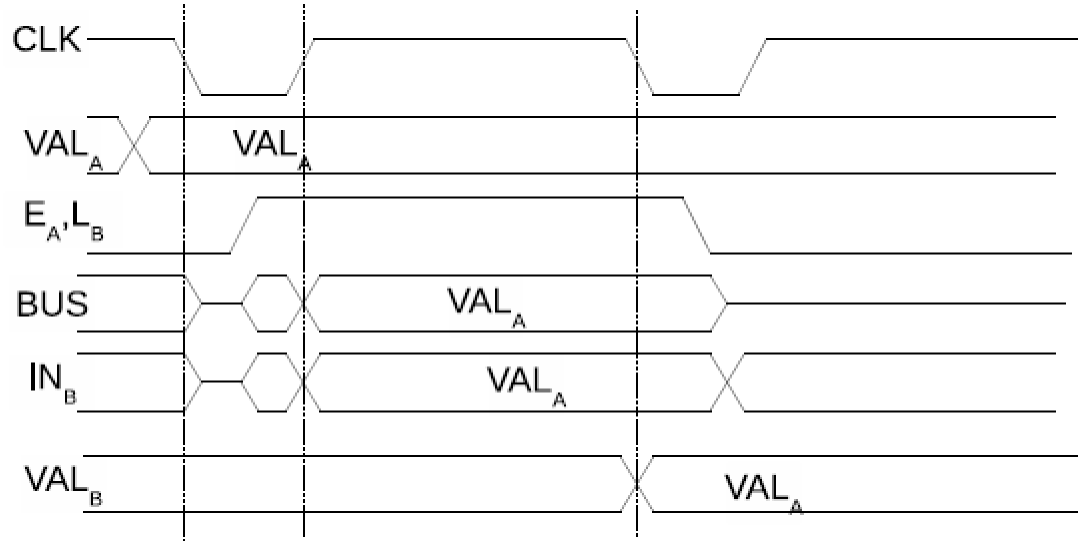
- The timing diagram depends on how we design the register
- In this case, we have designed such that all instructions (actions) happen to the stored value at the negative edge of the clock
- While the EN puts the data on the output bus at the positive edge of the clock



Creating a common bus

- Using tristate buffers, we can connect multiple input/outputs of multiple registers on the same bus
- Consider registers A and B
- The enable, load, and increment control signals of register A are respectively E_A, L_A, and I_A and for register B are labelled E_B, L_B, and I_B
- Let us say we need to transfer from A to B
- We enable E_A and L_B and in one clock cycle, the contents get transferred





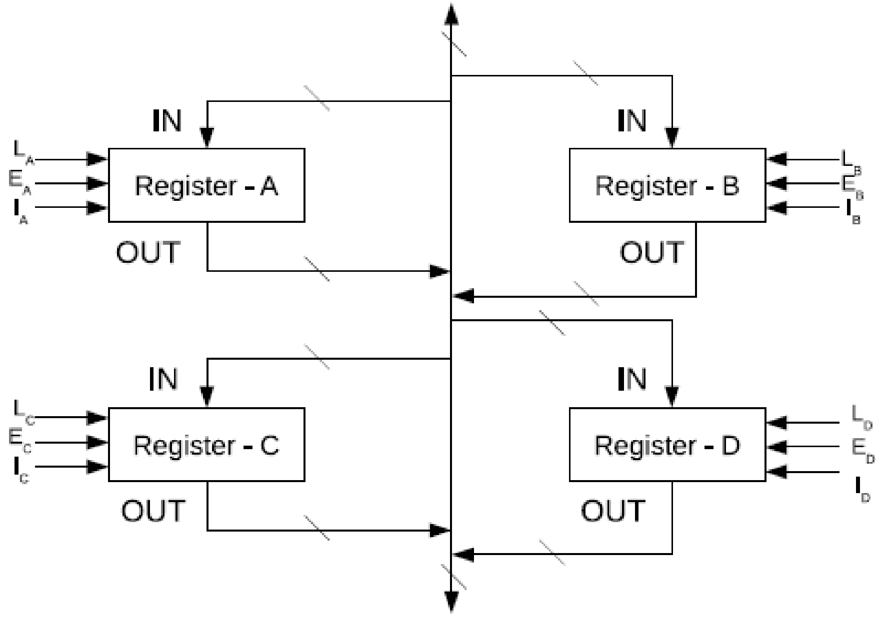
Creating a common bus

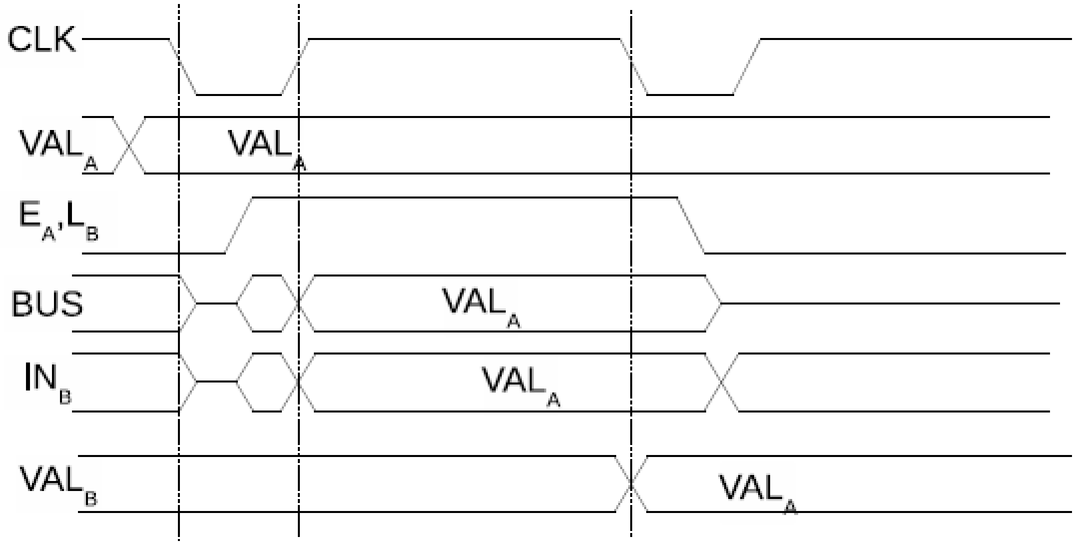
- In effect, we have moved the data to register B from register A, like a assignment instruction!

  A=B;

  MOV B, A

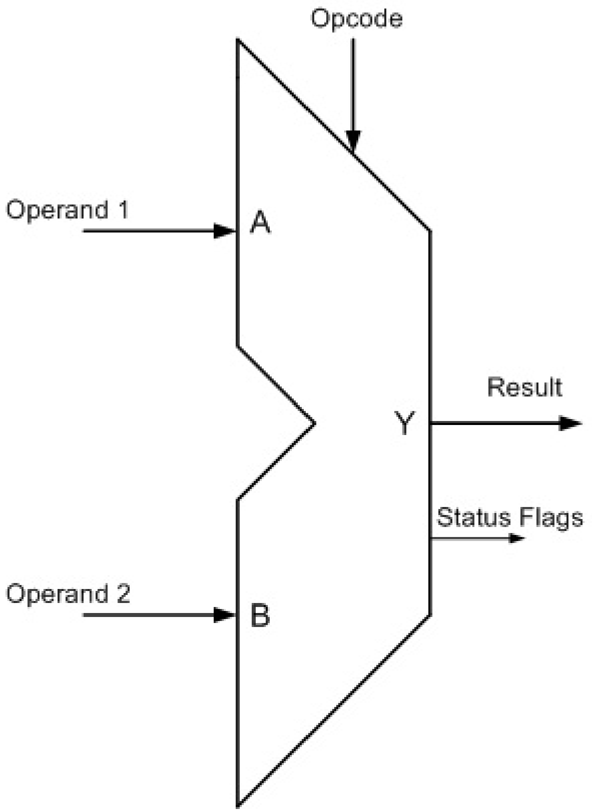
- At the level of the hardware, activating 2 signals simultaneously was all that was done
- The rest happened due to the bus connection, the clock, and the design of the register
- What if we activate L_B, E_D, L_C, L_A together?
- What if we activate E_B, L_A, I_B





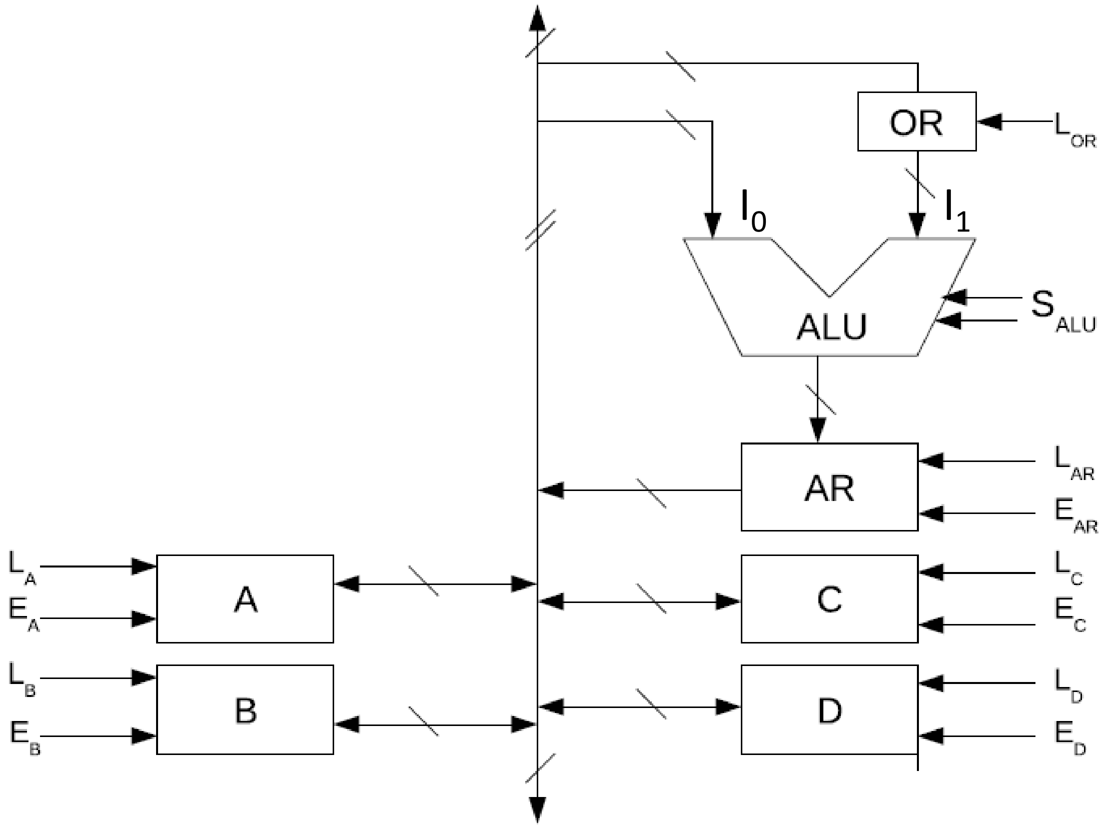
ALU design

- Enter, The ALU!
- We generally consider an ALU as a "simple" combinational circuit capable of performing several basic functions on a set of two binary numbers – add, sub, multiply, increment, AND, OR, XOR, and many other operations based on a select input
- This is generally implemented using a combination of MUXes within the ALU



ALU on a bus

- Consider the configuration which has a few registers and an ALU
- Assume all the registers are made using the multipurpose registers
- The ALU is a combinational circuit with 2 inputs and two lines to select the function to be performed
- The two select lines can be in one of 4 combinations
    - We refer to them symbolically as ADD, SUB, AND, PASS0



ALU on a bus

- The ALU has its left input connected to the bus and the right input to a register called OR or operand register, whose inputs are connected to the bus
- The output of the ALU is connected to the input of a register called AR or accumulator register, whose output is connected to the bus
- Every register has all the control signals of our generic multipurpose register