**Bangladesh University of Engineering and Technology**

# CSE 306
# Computer Architecture Sessional

# Assignment-1: 4-bit ALU Simulation

## Section – B1
## Group – 05

## Group Members:

1905071 – Tareq Ahmed
1905078 – Mehreen Tabassum Maliha
1905082 – Kazi Reyazul Hasan
1905084 – Wasif Jalal
1905088 – Mubasshira Musarrat

## Date:

21/12/2022

## 1) INTRODUCTION:

An arithmetic-logic unit (ALU) is the part of a central processing unit that carries out arithmetic and logic operations on the operands in computer instruction words. In some processors, the ALU is divided into two units: an arithmetic unit (AU) and a logic unit (LU). The ALU has a number of selection lines to select a particular operation in the unit. The selection lines are decoded within the ALU so that k selection variables can specify up to $2^k$ distinct operations. Here, three (k=3) selection variables (cs) are needed to perform $2^3 = 8$ distinct operations. The four data inputs from A are combined with the four inputs from B to generate an operation at the F outputs. A combination circuit is used to modify the data inputs, A and B to produce the inputs for the parallel adders to get the desired F outputs.

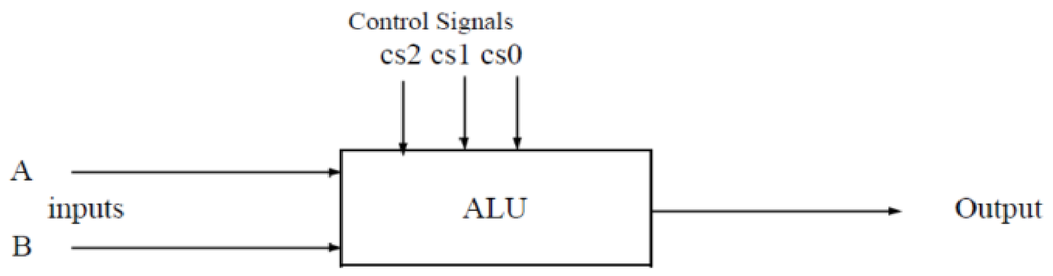
A 4-bit status register is implemented in this experiment. This status register contains 4 status bits that are denoted by C (Carry flag), V (Overflow flag), S (Sign flag), and Z (Zero flag). These status bits change during arithmetic operations. The behaviour of these status bits or 'flags' is described below:

1. **CF (Carry Flag):** Status bit C will be 1 if the output carry of the ALU is 1, otherwise it is set as 0. The carry flag becomes active when the addition of two sufficiently large n-bit numbers produces a result with (n+1) significant bits.
2. **VF (Overflow Flag):** The overflow flag is an indicator that a signed addition has yielded a result whose sign does not match what is expected from the signs of the operands, either because the capacity of the output register is exceeded, or because it is in fact an unsigned addition being interpreted as signed. Status bit V will be 1 if the X-OR operation of $C_3$ and $C_{out}$ is 1, otherwise it is set as 0
3. **SF (Sign Flag):** Status bit S is set to 1 when result is negative. When result is positive it is set to 0. Actually, this flag takes the value of the most significant bit.
4. **ZF (Zero Flag):** Status bit Z is set to 1 when result is zero. For non-zero result, this flag is set to 0. It is equivalent to the NOR of the output bits.

## 2) PROBLEM SPECIFICATION WITH ASSIGNED INSTRUCTIONS:

*Design a 4-bit ALU with three selection bits cs2, cs1 and cs0 for performing the following operations:*

| **Control Signals** | | | **Functions** | **Description** |
|---|---|---|---|---|
| **cs2** | **cs1** | **cs0** | | |
| 0 | 0 | 0 | Add with carry | A + B + 1 |
| 0 | 0 | 1 | XOR | A ⊕ B |
| X | 1 | 0 | Transfer A | Output is A |
| 0 | 1 | 1 | Add | A + B |
| 1 | 0 | X | Complement A | A’ |
| 1 | 1 | 1 | Increment A | A + 1 |



***Some exceptions:***

**A. For NOT Operation:**

1. After the NOT operation, which makes the result 0000, if Z becomes 0 from 1, it will not be accepted. However, if Z becomes 1 or if the Z flag remains unchanged, both will be accepted.
2. After the NOT operation, if S remains unchanged or it reflects the highest order bit of the result, both will be accepted. But if the S flag is changed and it is changed to a wrong value, it will not be accepted.
3. The C and V flags after NOT operation are not to be checked

**B. For AND/OR/XOR Operation:**

1. C and V should be cleared (0) after the operation.
2. S and Z should be changed according to the output.

## 3) DETAILED DESIGN STEPS:

- **Control Logic:** The ALU performs in total 6 types of operations directed by 8 different opcodes. A 3x8 line decoder with inverting outputs is chosen as the most efficient way to map each opcode (a specific assignment to the 3 control bits) to each operation's pathway in the circuit. The decoder's outputs are combined with some minimal combinational logic (just AND gates) to produce five Boolean expressions (that we will refer to as *Control Variables*) that are used as selector bits for multiplexers, or in certain cases, data inputs of operations that in turn derive other operations due to the specific assignment of data inputs, but produce outputs from a common path, thus reducing the amount of circuitry required. The mathematical derivation of the expressions which define the Control Variables will be done from the truth table in the next section.

- **Operation Path Selection:** The unit has two almost distinct parts for two types of operations. The selection of the path for a specific operation is done by multiplexing, with a quad 2-to-1 multiplexer which is placed right before the **final output F** and is labelled as **$MUX_1$**. The decision between two paths is made by the **Control Variable $P_1$**. The operations Add, Add with Carry, Increment and Transfer are done through the Arithmetic Path; the operations XOR and Complement are done through the Boolean Logic Path.

  - **Arithmetic Path:** At the heart of the Arithmetic Path there sits a 4-bit parallel adder which does the operations A+B, A+B+1, A+1 straightforwardly as expected (A+B+0, A+B+1, A+0000+1 respectively; where the last operand is $C_{in}$ of the adder). Transfer A is also implemented as an arithmetic operation, but as (A + 1111 +1) which produces an output whose lowest 4 bits are the same as A. The **adder has two 4-bit inputs which we label as X and Y, and an input carry bit $C_{in}$**. The value of X is tied directly to our ALU input A, but the value of Y is variable. For addition with or without carry, it's value is set to B, but for the other two operations its value is chosen between 0000 and 1111 by a **Control Variable $P_3$.** This selection between B and a specific constant is done by another quad 2-to-1 multiplexer (labelled **$MUX_2$)**

whose selector bit is decided by another **Control Variable $P_2$.** The value of $C_{in}$ is decided between 0 and 1 by yet another **Control Variable $P_0$.**

- **Boolean Logic Path:** The two logical operations Complement and XOR can actually be done minimally with just one quad XOR-gate because $A \oplus 1 = \overline{A}$. In this case the multiplexing done to determine the required value of the adder input Y can be utilised again. The adder inputs X and Y will be sent to the XOR-gate as well, but just the definition of Control Variables $P_3$ and $P_2$ will be adjusted to set Y = 1111 for the opcode of Complement A.

❖ **Implementation of Flags:** The Sign Flag (SF) is just an extension of the MSB of the 4-bit ALU output F. So, we have the following expression for it, that does not require any additional logic circuitry.

$$\boldsymbol{S = F_3}$$

The Carry Flag (CF) is produced directly by the parallel adder as its output carry $C_{out}$**.** The Overflow Flag is defined as $V = C_3 \oplus C_{out}$. Although it is not possible to obtain the intermediate carry directly from the parallel adder's IC, it can be worked around with simple combinational logic: The sum bits of the parallel adder are defined as

$S_i = A_i \oplus B_i \oplus C_i$

=> $S_i \oplus (A_i \oplus B_i) = (A_i \oplus B_i) \oplus (A_i \oplus B_i) \oplus C_i$

=> $S_i \oplus (A_i \oplus B_i) = 0 \oplus C_i$ => $C_i = S_i \oplus (A_i \oplus B_i)$

Therefore, we have: $V = S_3 \oplus X_3 \oplus Y_3 \oplus C_{out}$

However, there is a requirement to clear C and V in case of logical operation XOR. The current definition of Transfer A also produces a carry of 1. (It was defined in this way after concluding through trial and error that it requires the least number of AND-gates when building the expressions of the Control Variables).

To satisfy that requirement another **Control Variable $P_4$**, which is to be inactive only when the opcodes for XOR and Transfer A are triggered, is ANDed with both the results of C and V. So, the final expressions we implement in the circuit are –

$$V = (S_3 \oplus X_3 \oplus Y_3 \oplus C_{out}) \,.\, P_4$$

$$C = C_{out} \,.\, P_4$$

A quad XOR-gate will be used to build the expression for V, which contains three XOR operations. Now, the Zero flag Z is defined as the NOR of the 4 output bits. NOR is not associative, so a quad OR-gate will be used to OR the $F_i$ bits first. Then instead of bringing in a new NOT-gate, the one leftover gate from the quad XOR used in building the V flag is used to complement the expression because $X \oplus 1 = \overline{X}$.
So, the final expression for the Z-flag is –

$$Z = (F_3 + F_2 + F_1 + F_0) \oplus 1$$

Now to summarise the flow of logic in the circuit –

1) The control signals cs2, cs1 and cs0 (referred to as $s_2$, $s_1$, $s_0$ henceforth) are fed to a inverting 3x8 decoder. Five Control Variables are produced by applying some basic combinational logic on the decoder outputs.
2) A multiplexer (driven by $P_2$) choses between B and $P_3$ to output Y. The adder and XOR-gate receive X = A and Y. The adder also receives $P_0$ as input carry.
3) A multiplexer (driven by $P_1$) choses between the adder's and XOR-gate's outputs and produces the ALU's final output.
4) Flags are outputted by applying some basic combinational logic on the ALU outputs and $P_4$.

Thus, to make the unit completely functional now, all we require is to find the expressions for the Control Variables in a way that accurately maps the control signals to the desired flow path in the circuit. The decoder abstracts the bulk of the combinational logic in such an efficient way that there is no requirement for gate-level minimization using K-maps, as we are primarily targeting to minimise the count of ICs required. The derivation of expressions (consisting of decoder outputs, i.e., inverted minterms only) can be done directly from the following truth table.

## 4) TRUTH TABLE:

| Control signal | | | F (Output) | | $P_k$ (conditional expressions produced from decoder outputs) | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $s_2$ | $s_1$ | $s_0$ | | Equation (Detail) | $P_0$ ($C_{in}$ of adder) | $P_1$ ($MUX_1$ selector: 0 = logical path output, 1 = arith. path output) | $P_2$ ($MUX_2$ sel.: 0 = pass $B_i$ ; 1 = pass $P_3$ ) | $P_3$ (value of $Y_i$ of adder if $MUX_2$ not passing $B_i$ ) | $P_4$ (condition: 0 = clear C,V; 1 = get C,V from adder |
| 0 | 0 | 0 | A+B+1 | A+B+1 | 1 | 1 | 0 | X | 1 |
| 0 | 0 | 1 | A ⊕ B | A ⊕ B | X | 0 | 0 | X | 0 |
| 0 | 1 | 0 | A | A + 1111 +1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | A+B | A+B+0 | 0 | 1 | 0 | X | 1 |
| 1 | 0 | 0 | A’ | A ⊕ 1 | X | 0 | 1 | 1 | X |
| 1 | 0 | 1 | A’ | A ⊕ 1 | X | 0 | 1 | 1 | X |
| 1 | 1 | 0 | A | A + 1111 +1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | A+1 | A + 0 + 1 | 1 | 1 | 1 | 0 | 1 |

From the truth table, taking POS forms of the Control Variables as functions of the control bits $s_2$, $s_1$, $s_0$, it is evident that:

$$P_0 = (s_2 + \overline{s_1} + \overline{s_0}) = \overline{\overline{s_2} s_1 s_0} = \overline{D_3}$$

$$P_1 = (s_2 + s_1 + \overline{s_0})(\overline{s_2} + s_1 + s_0)(\overline{s_2} + s_1 + \overline{s_0})$$

$$= \overline{\overline{s_2}\,\overline{s_1} s_0} \cdot \overline{s_2 \overline{s_1}\,\overline{s_0}} \cdot \overline{s_2 \overline{s_1} s_0} = \overline{D_1}\,\overline{D_4}\,\overline{D_5}$$

$$P_2 = (s_2 + s_1 + s_0)(s_2 + s_1 + \overline{s_0})(s_2 + \overline{s_1} + \overline{s_0})$$

$$= \overline{\overline{s_2}\,\overline{s_1}\,\overline{s_0}} \cdot \overline{\overline{s_2}\,\overline{s_1} s_0} \cdot \overline{\overline{s_2} s_1 s_0} = \overline{D_0}\,\overline{D_1}\,\overline{D_3}$$

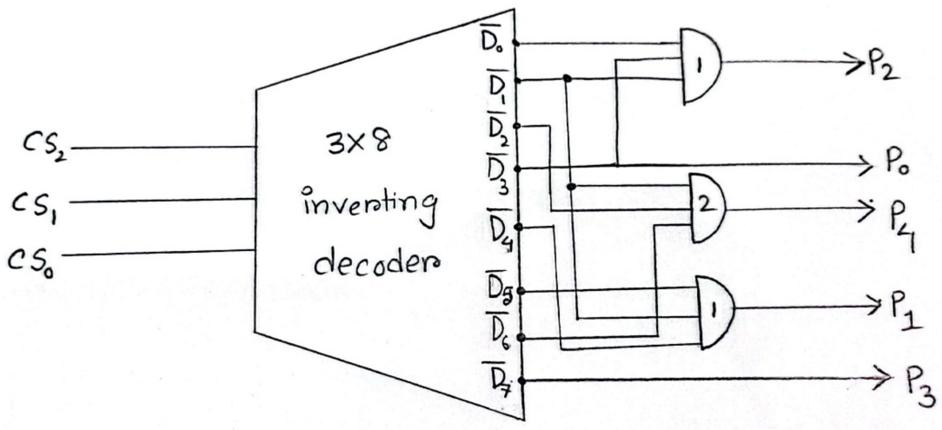
$$P_3 = (\overline{s_2} + \overline{s_1} + \overline{s_0}) = s_2 s_1 s_0 = \overline{D_7}$$

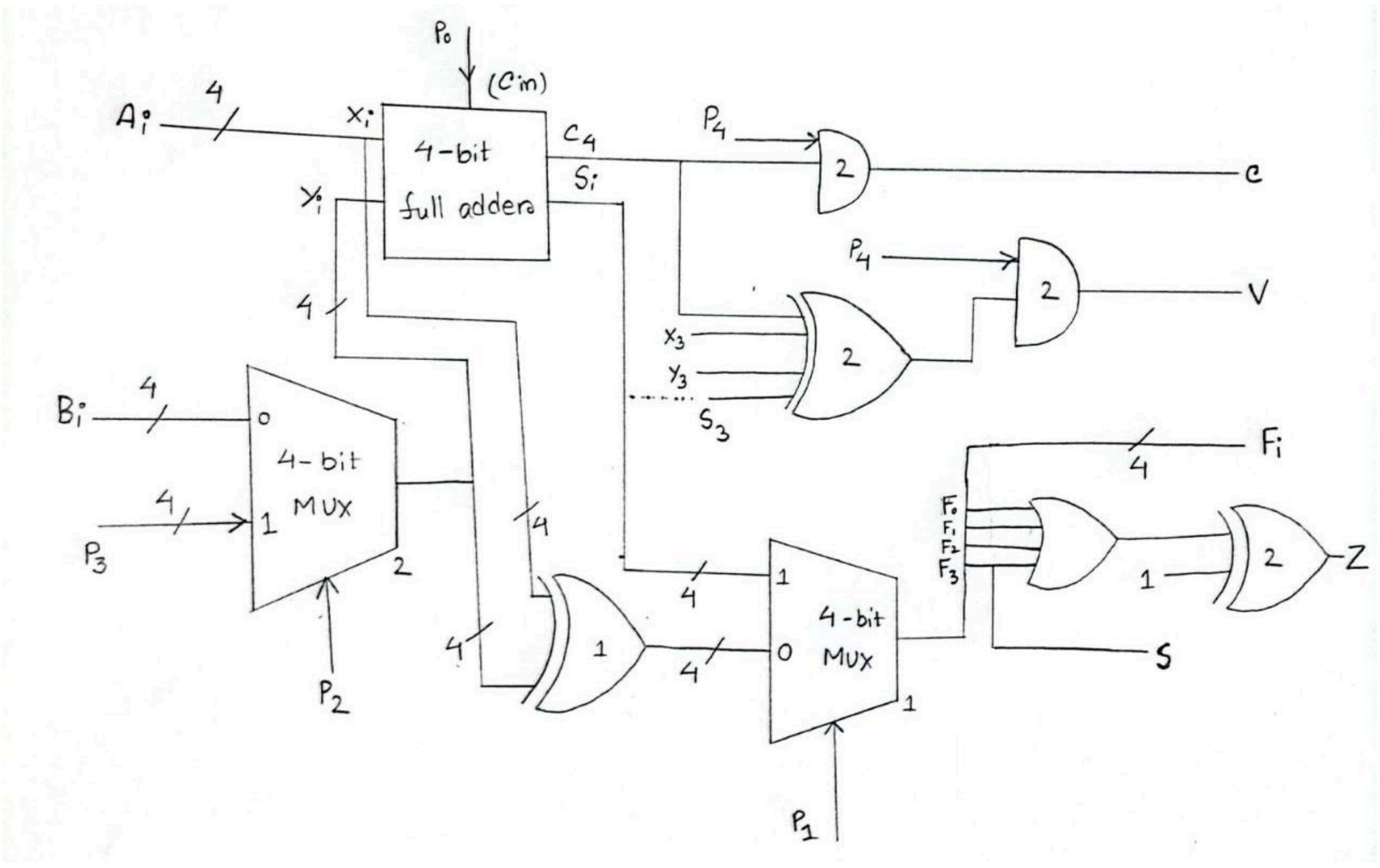
$$P_4 = (s_2 + s_1 + \overline{s_0})(s_2 + \overline{s_1} + s_0)(\overline{s_2} + s_1 + s_0)$$

$$= \overline{\overline{s_2}\,\overline{s_1} s_0} \cdot \overline{\overline{s_2} s_1 \overline{s_0}} \cdot \overline{s_2 s_1 \overline{s_0}} = \overline{D_1}\,\overline{D_2}\,\overline{D_6}$$

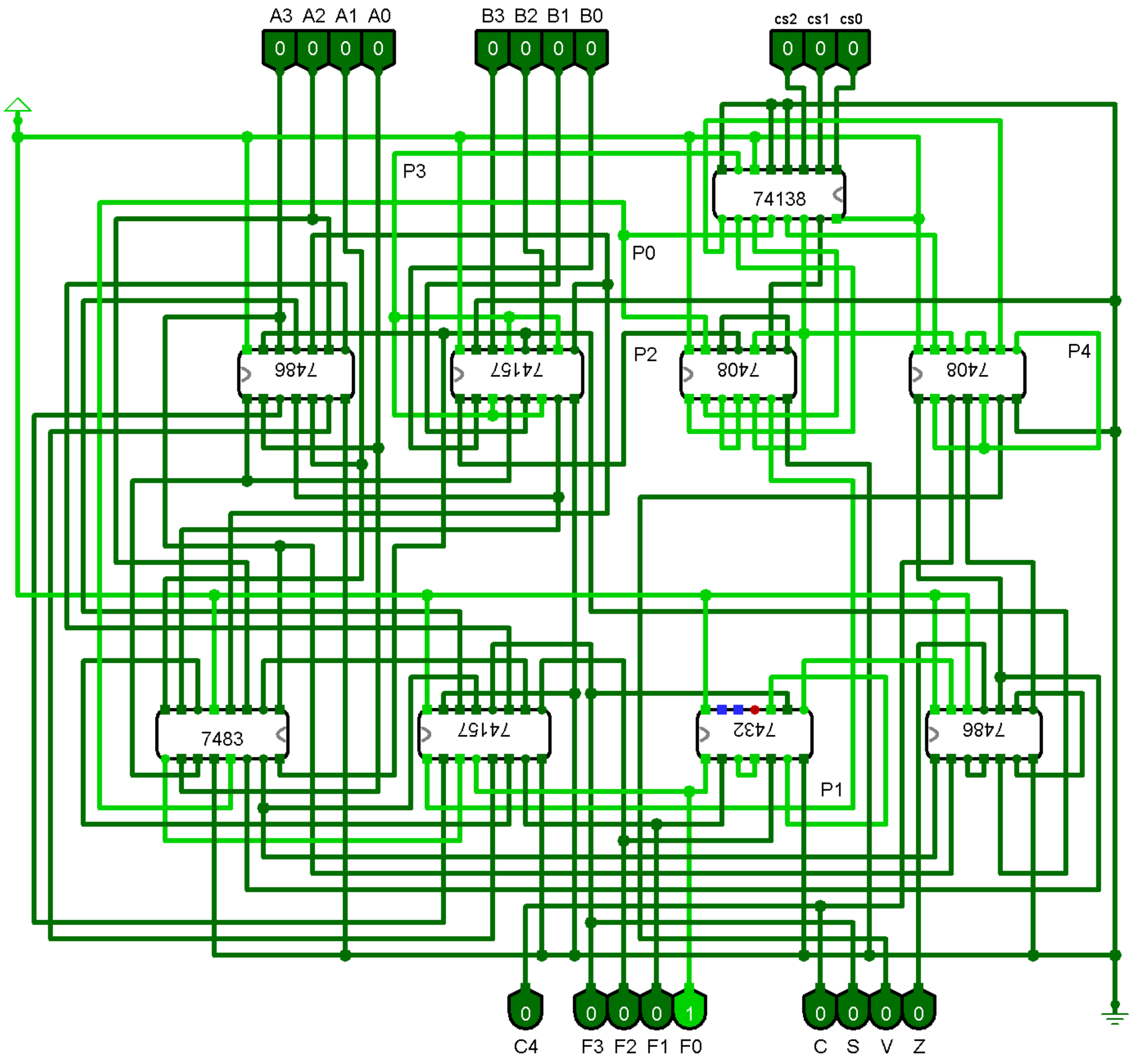
– where $\overline{D_0}$ to $\overline{D_7}$ are complements of the minterms formed of $s_2$, $s_1$, $s_0$, produced from a 3 to 8 line decoder with inverting outputs.

## 5) BLOCK DIAGRAM:





## 6) COMPLETE CIRCUIT DIAGRAM:



## 7) REQUIRED ICs:

| IC Name | Description | Count | |
|---|---|---|---|
| IC 74138 | 3-to-8 line decoder/demultiplexer, inverting outputs | 1 | |
| IC 7486 | Quad 2-input XOR gate | 2 | |
| IC 74157 | Quad 2-line to 1-line data selector/multiplexer, non-inverting outputs | 2 | |
| IC 7408 | Quad 2-input AND gate | 2 | |
| IC 7483 | 4-bit binary full adder | 1 | |
| IC 7432 | Quad 2-input OR gate | 1 | |
| | | **Total** | **9** |

## 8) SIMULATOR NAME:

Logisim-2.7.1 / Logisim-Ita-2.16.1.4 (cross-compatible)

## 9) DISCUSSIONS:

In the design of this ALU, the circuit could be greatly minimized and simplified by making the effective use multiplexers and decoders. The final design was settled upon after analysing several other designs on a trial-and-error basis.

A straightforward combinational design channelling all outputs through the parallel adder used at first required 12 ICs. Multiplexing separate pathways did not help in bringing down the IC count initially, buy using a decoder to abstract the combinational logic of the control signals brought this count down to 10 ICs. Afterwards, branching the possible paths of logic with multiplexers, coupled together with the decoder enabled us to complete the design very tightly within 9 ICs, with only one gate left unused. Multiplexing also made the circuit much more debuggable and easy to understand compared to a random combination of basic gates. The carry and overflow flags were left as don’t cares for the Complement operation because handling them effectively would require a single not gate and thus an entire mostly unused IC.

In the hardware implementation, care was taken to ensure that all the connections in the circuit were properly attached, through repeated continuity tests with a multimeter. Extensive random value tests were run along with result verification to ensure the functionality of the unit. During construction and testing phases, the instruments used were fortunately mostly functional, thus enabling us to build a functional ALU with minimal debugging.