Implementation and Simulation of Simplified Necessary Components of a Classic CPU*

*Note: Project of the course ICE3405P-Computer Organization

Nan Lin
SPEIT
Shanghai Jiao Tong University
Shanghai, China
lns_brandon@sjtu.edu.cn

Yanxu Meng
SPEIT
Shanghai Jiao Tong University
Shanghai, China
meng-mou-xu@sjtu.edu.cn

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—component, formatting, style, styling, insert**

I. Question 1-5: Implementation of a Two's Complement Calculation

A. Question 1

The two's complement of a binary number is obtained by following the rule:

- If the sign bit (highest bit) is 0, it indicates that the sign-magnitude representation is a positive number. In this case, the binary number is unchanged.
- If the sign bit is 1, we should invert its digits and add one to the least significant bit.

This method allows for binary arithmetic and simplifies the design of digital circuits for arithmetic operations. A straightforward implementation is to adopt different operation modes depending on the sign bit using a multiplexer. A typical multiplexer is shown in Figure 1. For an 8-bit binary number, suppose that it could be represented as:

$$\overline{s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0} \tag{1}$$

s_7 is the sign bit, therefore playing the role of Select for all other bits (s_0 to s_6) here.

In order to add one to the least significant bit, an adder is required. A half-adder is a fundamental component in digital arithmetic. It takes two single-bit binary inputs and produces a sum and carry output. The truth table for a half-adder is shown below:

A	B	Sum (Result)	CarryOut
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

TABLE I
Truth table for a half-adder

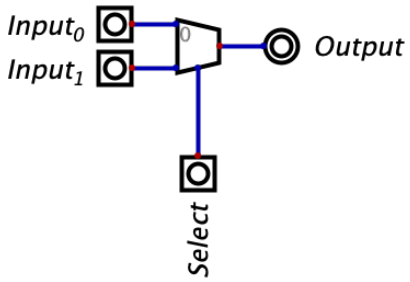

Fig. 1. A Multiplexer. When Select is 0, the output corresponds to the value of Input_0, neglecting the condition of Input_1. Inversely, when Select is 1, the output corresponds to the value of Input_1.

The corresponding half-adder circuit diagram is illustrated in Figure 2.

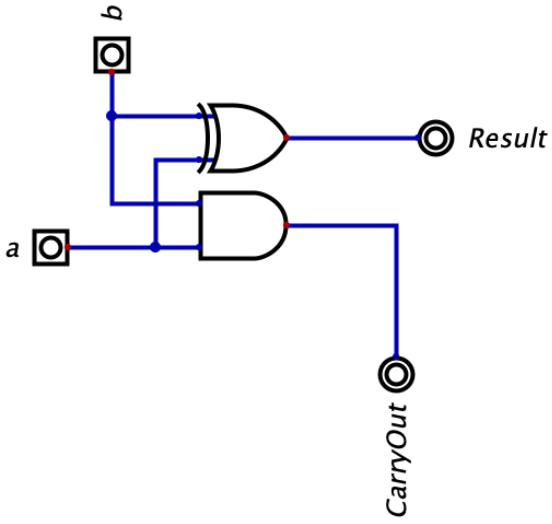

Fig. 2. Half-Adder Circuit

We use a half-adder rather than a full adder for multiple reasons:

- A half-adder has a relatively simpler structure. Most

importantly, its depth is one less than a full adder and it contains fewer gates which provides the advantage of lower energy consumption and less time consumption.
- A half-adder is proved to be effective because to implement a two's complement circuit, we only need to calculate $\overline{s_6 \dots s_0} + \overline{0 \dots 01}$ (taking 8-bit binary numbers as an example). Since all the bits are 0, we can put the CarryOut onto the place of another adder.

To implement an 8-bit two's complement circuit, we combine multiple half-adders; each half-adder is used to calculate the corresponding bit of the binary number. The process involves inverting each bit of the input number and then adding one using a series of adders. The detailed circuit diagram is shown in Figure 3.

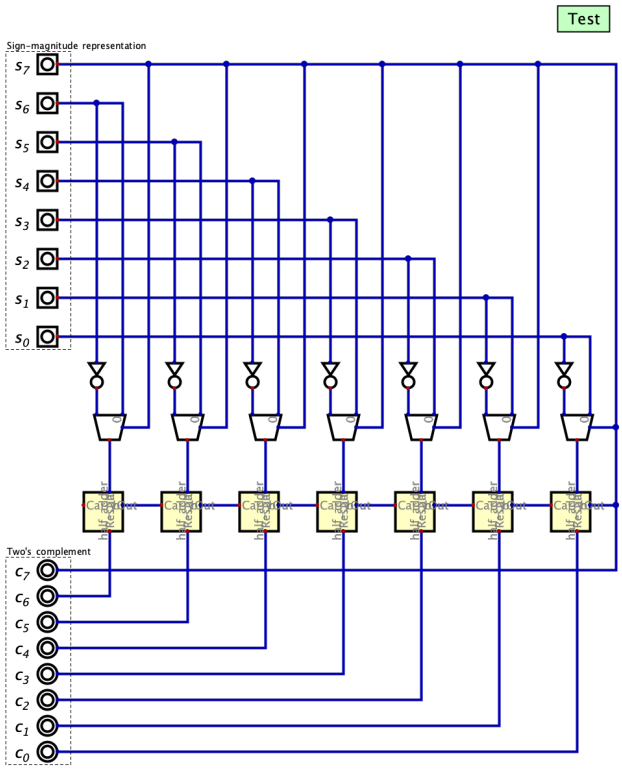

Fig. 3. 8-bit Two's Complement Circuit. The yellow box is a half-adder implemented in Figure 2

B. Question 2

To analyze the depth and complexity of the two's complement circuit for 8 bits, we divide the process into two stages.

The first stage is the **inversion stage**. Each bit requires a NOT gate, thus in the first stage:

- Depth: 1 (because all of the NOT gates operate simultaneously)
- Complexity: 7 (NOT gates)

The second stage is the **addition stage**. For each bit, a half-adder is applied. The depth of each half-adder is 1 and the complexity is 2. Since there are 7 bits that need to be calculated one by one, in the second stage:

- Depth: $7 \times 1 = 7$
- Complexity: $7 \times 2 = 14$

Summing up the two stages, the depth of calculation of the two's complement of an 8-bit binary number is $1+7=8$ and the complexity is $7+14=21$.

We expand the results to any 2^p-bit ($p \in \mathbb{N}$) two's complement circuit.

1) In the **inversion stage**, the depth is 1 due to parallelization, and the complexity is $2^p - 1$.
2) In the **addition stage**, the depth is $2^p - 1$ and the number of gates required is $2 \times (2^p - 1)$.

Conclusion: For a 2^p-bit two's complement circuit, the depth is 2^p, and the complexity is $3(2^p - 1)$.

C. Question 3

In order to reduce the depth of the circuit, having a closer look at the circuit implemented in Section I-A, the main problem is that the CarryOut is passed on in series. In other words, to calculate the two's representation of s_k of a 2^p-bit binary number ($0 < k \leq 2^p - 2$), we have to wait for the values of $k-1$ CarryOuts one by one.

To optimize this process, we first propose a new version of the two's complement circuit for an 8-bit machine by employing a method that uses two 4-bit lookahead carry adders in series. This approach allows for the calculation of four bits and their carry at once, then passing the carry to the next set of four bits. This reflects the divide and conquer strategy.

1) Two 4-bit Lookahead Adders in Series: The implementation involves the following steps:

1) **Inversion Stage**: This stage is unchanged, where each of the 8 bits is inverted using NOT gates.
2) **Addition Stage**: The 8-bit addition is divided into two 4-bit additions using lookahead carry adders.
   - **First 4-bit Adder**: Computes the result and the carry for the first four bits.
   - **Second 4-bit Adder**: Computes the result for the next three (but not four!) bits.

We then implement a classical 4-bit lookahead carry adder which can be found in textbooks. First, we introduce two fundamental components.

In digital circuits, the Generate-Propagate (GP) logic is used to speed up the carry calculation in adders. For two binary inputs A_i and B_i, the generate (G_i) and propagate (P_i) signals are defined as:

$$g_i = a_i \cdot b_i \tag{2}$$

$$p_i = a_i + b_i \tag{3}$$

The GP generator is used to generate respectively the AND and OR results of two variables, this result will be exploited later on. The implementation of the GP generator is displayed in Figure 4. For any bit, the value is

$$\forall k \in [\![1, 2^p - 2]\!], \quad s_k = a_k + b_k + c_{k-1} \tag{4}$$

where c_{k-1} is the carry bit into position k.

The carry bits can be calculated as follows:

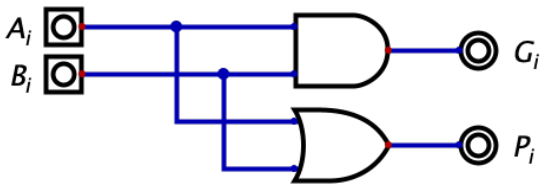

Fig. 4. GP Generator Circuit

$$c_{i+1} = a_i b_i + (a_i + b_i)c_i \tag{5}$$

due to the reason that the carry bit is 1 if

- Both of the two bits are 1
- Any of the two bits is 1 and the carry bit from the former bit is 1.

Here we replace $a_i b_i$ and $a_i + b_i$ with g_i and p_i. The Equation 5 turns into:

$$c_{i+1} = g_i + p_i c_i \tag{6}$$

We therefore deduce all the carry bits based on Equation 6:

$$c_1 = g_0 + p_0 c_0 \tag{7}$$

$$c_2 = g_1 + p_1 c_1 = g_1 + p_1 g_0 + p_1 p_0 c_0 \tag{8}$$

$$c_3 = g_2 + p_2 c_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \tag{9}$$

$$c_4 = g_3 + p_3 c_3 = g_3 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0 \tag{10}$$

A full adder also takes the CarryIn of the former bit compared to a half adder. It is displayed in Figure 5.

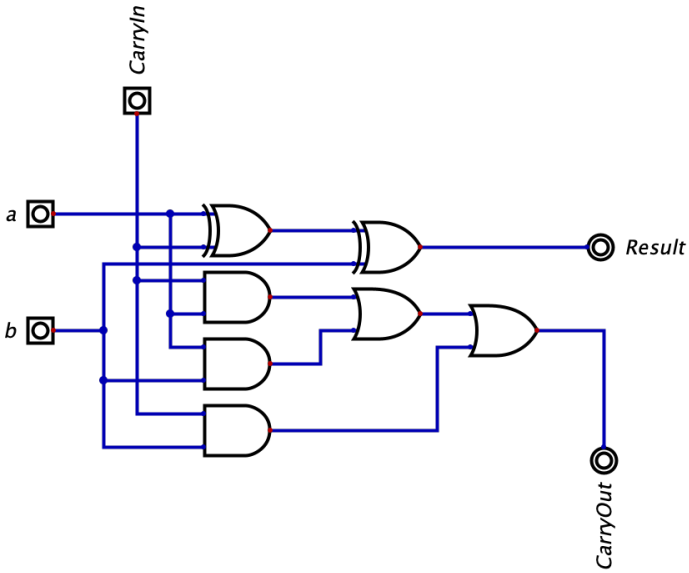

Fig. 5. Full-Adder Circuit

A classic 4-bit lookahead carry adder based on full adders is shown in Figure 6. We then combine the two 4-bit adders in series. The final two's complement circuit is shown in Figure 7. When the sign bit is 1, it implies that we should invert its digits and add one to the least significant bit. In this case, the sign bit is passed to the first CarryIn input, and the operation of the full adder is identical to:

$$\overline{a_3 a_2 a_1 a_0} + \overline{0000} \quad \text{with CarryIn 1} \tag{11}$$

$$\overline{0 a_6 a_5 a_4} + \overline{0000} \quad \text{with CarryIn from the former 4-bit adder} \tag{12}$$

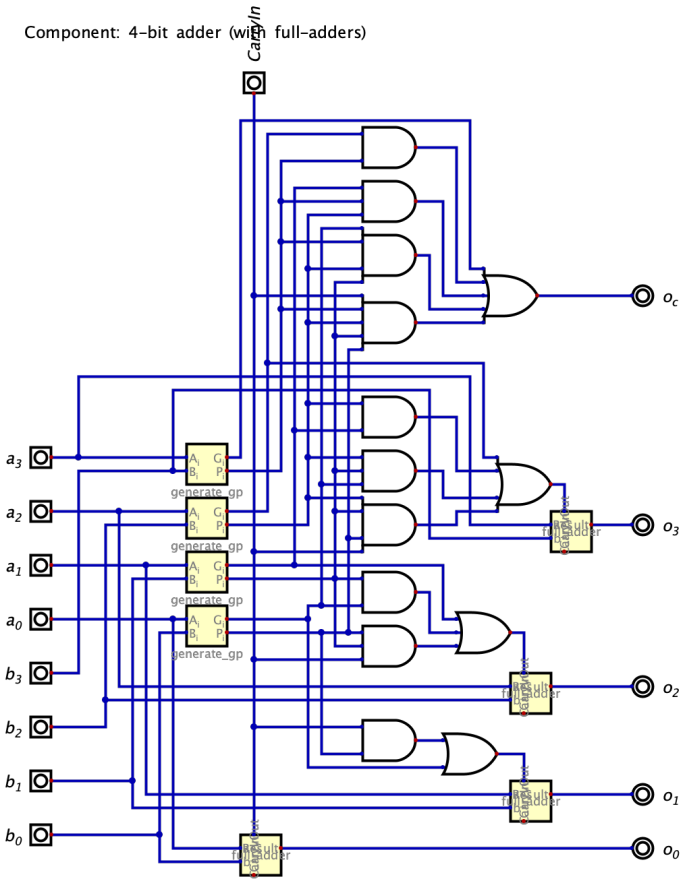

Fig. 6. 4-bit Lookahead Carry Adder based on Full-Adders

The reason why we put in series of 4-bit Lookahead Carry Adder instead of series of 8 (or even 16, 32, ...)-bit Lookahead Carry Adder is to reduce the complexity of designing the circuit. As shown in Figure fig:lca, to implement a 4-bit Lookahead Carry Adder, we need an AND gate with 5 inputs. This is already hard to fabricate in reality and it consumes much more time and energy if we separate the AND gates.

However, there are still spaces of optimization. This indeed is the implementation of a standard adder between two 8-bit binary numbers. Here, we're only required to perform addition on binary numbers with 1. We could throw out 0s and full adders and adopt half adders.

Furthermore, since the other adder is always 0 in Equation 11, the G(enerate) signal is always 0. This implies for any valid value of g_i, they should be 0. Equations 7 to 10

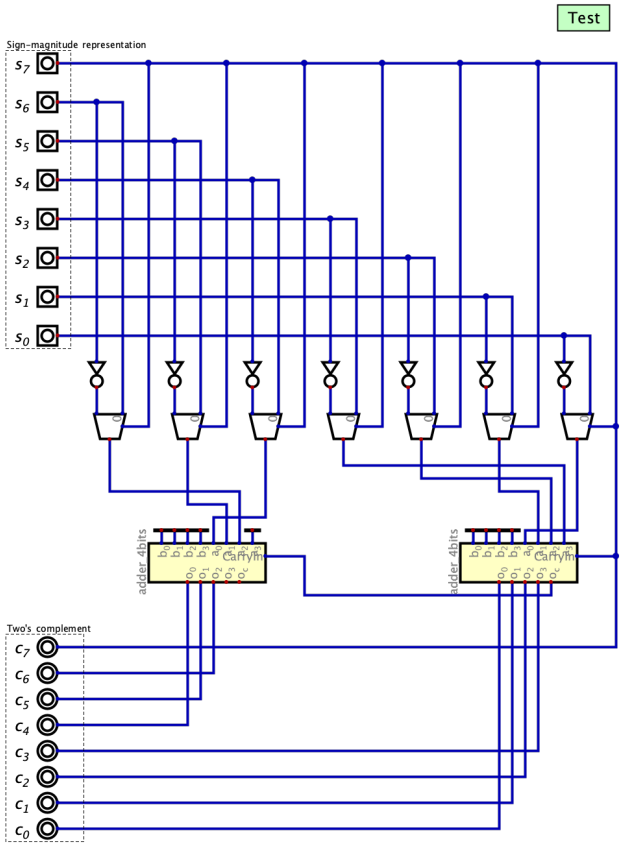

Fig. 7. 8-bit Two's Complement Circuit (Divide and Conquer Method) Based on Full-Adders

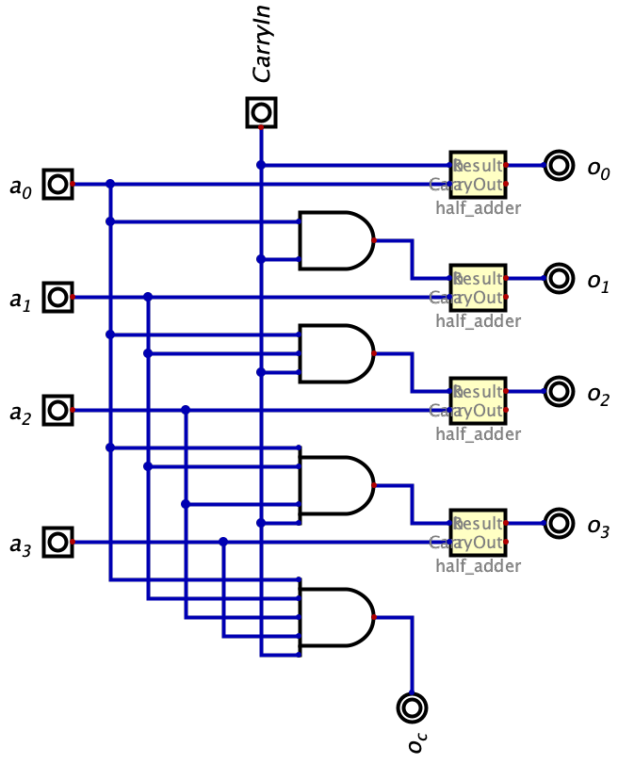

Fig. 8. Optimized 4-bit Lookahead Carry Adder based on Half-Adders and simplified gates

could be transformed into:

$$c_1 = p_0 c_0 \tag{13}$$
$$c_2 = p_1 p_0 c_0 \tag{14}$$
$$c_3 = p_2 p_1 p_0 c_0 \tag{15}$$
$$c_4 = p_3 p_2 p_1 p_0 c_0 \tag{16}$$

Based on these two points, the 4-bit Lookahead Carry Adder could be optimized as shown in Figure 8.

The corresponding 8-bit two's complement circuit is shown in Figure 9.

It seems feasible for p relatively small. However, if p is much greater (e.g., to sum up two 64-bit binary numbers), the depth is still comparatively high. Calculation of depth and complexity of this circuit is shown in Section I-D1. The divide and conquer method is not fully implemented since we're only grouping up 4 bits together and then calculating them one by one. There are still spaces for improvement.

2) Solution for binary numbers with much more bits: A real divide and conquer solution is like this: we separate the bits into groups of 2 bits, calculate the values separately, then combine them into groups of 4 bits, then calculate the values separately, etc.

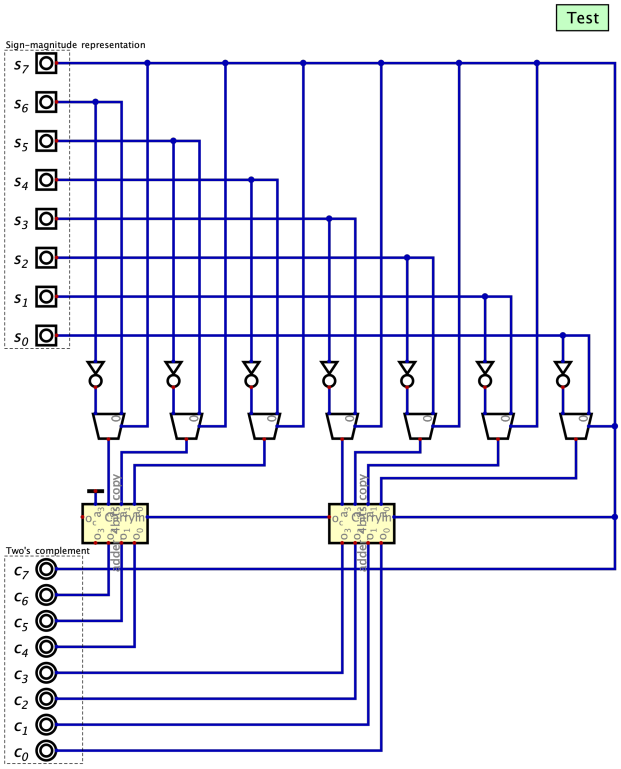

Fig. 9. Optimized 8-bit Two's Complement Circuit (Divide and Conquer Method) Based on 4-bit Lookahead Carry Adder Proposed in Figure 8

We analyze as follows (with some notations): A 1-bit adder is:

$$a_0 + b_0 = \overline{r_0 s_0} \quad \text{with} \quad \begin{cases} r_0 = a_0 \wedge b_0 \\ s_0 = a_0 \oplus b_0 \end{cases} \tag{17}$$

If we consider the carry from the former bit, knowing that

$$1 + \overline{r_0 s_0} = \overline{R_0 t_0} \quad \text{with} \quad \begin{cases} R_0 = (a_0 \wedge 1) \vee (b_0 \wedge 1) = a_0 \vee b_0 \\ t_0 = s_0 + 1 = a_0 \oplus b_0 + 1 \end{cases} \tag{18}$$

(It is worthy to know that, since $t_0 = s_0 + 1$, we could add a NOT gate to calculate the value of t_0.) To execute the ADD operation, four values (r, s, R, t) should be output so as to be passed on. R could be understood as a carry could be generated if it is excited by former bits. The implementation is shown in Figure 10.

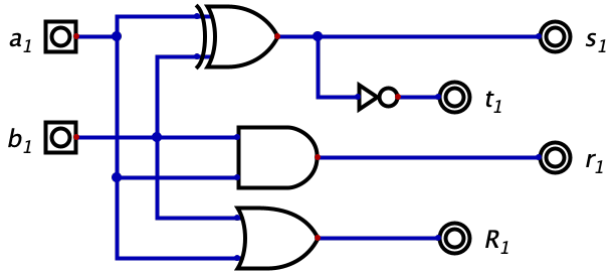

Fig. 10. Implementation of a 1-bit Adder with four values output

To add two 2-bit binary numbers, we first calculate the four values of each corresponding bit. We name them $r_{1,0}, s_{1,0}, R_{1,0}, t_{1,0}$ and $r_{1,1}, s_{1,1}, R_{1,1}, t_{1,1}$ accordingly. Thus, a 2-bit adder is:

$$\overline{a_1 a_0} + \overline{b_1 b_0} = \overline{r_1 s_1 s_0} \quad \text{with} \quad \begin{cases} r_1 = r_{1,1} \vee (r_{1,0} \wedge R_{1,1}) \\ s_1 = \begin{cases} s_{1,1} \text{ if } r_{1,0} = 0 \\ t_{1,1} \text{ if } r_{1,0} = 1 \end{cases} \\ s_0 \text{ unchanged} \end{cases} \tag{19}$$

Similarly, using the notation $1 + \overline{r_1 s_1 s_0} = \overline{R_1 t_1 t_0}$, the 2-bit Adder should calculate the value of R_1, t_1 and t_0 with expressions:

$$1 + \overline{r_1 s_1 s_0} = \overline{R_1 t_1 t_0} \quad \text{with} \quad \begin{cases} R_1 = r_{1,1} \vee (R_{1,1} \wedge R_{1,0}) \\ t_1 = \begin{cases} s_{1,1} \text{ if } R_{1,0} = 0 \\ t_{1,1} \text{ if } R_{1,0} = 1 \end{cases} \\ t_0 \text{ unchanged} \end{cases} \tag{20}$$

Corresponding circuit implementation is shown in Figure 11.

In general, to add between 2^p-bit binary numbers where $p \in \mathbb{N}^*$ and $p \geq 1$, suppose that the calculation of two 2^{p-1}-bit Adders are resp. $r_{1,0}, s_{2^{p-1},0}, \ldots, s_{1,0}, t_{2^{p-1},0}, \ldots, t_{1,0}, R_{1,0}$ and $r_{1,1}, s_{2^{p-1},1}, \ldots, s_{1,1}, t_{2^{p-1},1}, \ldots, t_{1,1}, R_{1,1}$, depending on values of $r_{1,0}$ and $R_{1,0}$:

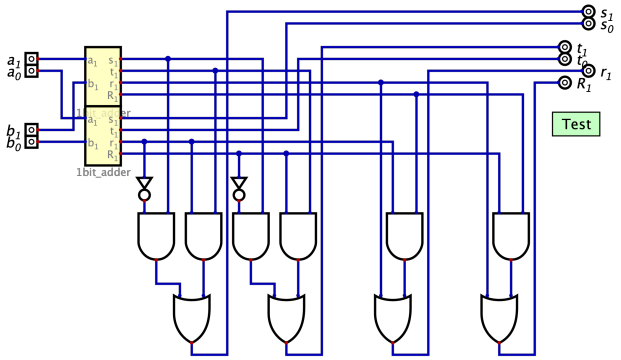

Fig. 11. Implementation of a 2-bit Adder with six values output. The 1-bit Adder displayed in Figure 10 is utilized.

- If $r_{1,0} = 0$ and $R_{1,0} = 0$, $r_1 = r_{1,1}$, $R_1 = r_{1,1}$,

$$s_k = \begin{cases} s_{k,1} & \text{if} \quad 1 \leq k \leq 2^{p-1} \\ s_{k,2} & \text{others} \end{cases} \tag{21}$$

and

$$t_k = \begin{cases} t_{k,1} & \text{if} \quad 1 \leq k \leq 2^{p-1} \\ s_{k,2} & \text{others} \end{cases} \tag{22}$$

(Note: the higher half of the t values is equal to s since nothing would be changed even if the binary number is added by one)

- If $r_{1,0} = 0$ and $R_{1,0} = 1$, we concatenate the result, that is to say, $r_1 = r_{1,1}$, $R_1 = R_{1,1}$,

$$s_k = \begin{cases} s_{k,1} & \text{if} \quad 1 \leq k \leq 2^{p-1} \\ s_{k,2} & \text{others} \end{cases} \tag{23}$$

and

$$t_k = \begin{cases} t_{k,1} & \text{if} \quad 1 \leq k \leq 2^{p-1} \\ t_{k,2} & \text{others} \end{cases} \tag{24}$$

- If $r_{1,0} = 1$ and $R_{1,0} = 1$, we concatenate the result, that is to say, $r_1 = R_{1,1}$, $R_1 = R_{1,1}$,

$$s_k = \begin{cases} s_{k,1} & \text{if} \quad 1 \leq k \leq 2^{p-1} \\ t_{k,2} & \text{others} \end{cases} \tag{25}$$

and

$$t_k = \begin{cases} t_{k,1} & \text{if} \quad 1 \leq k \leq 2^{p-1} \\ t_{k,2} & \text{others} \end{cases} \tag{26}$$

Based on this analysis, a 4-bit Adder with ten values output is shown in Figure 12, and an 8-bit Adder with eighteen values output is shown in Figure 13.

Using the 8-bit Adder implemented above, we design our final edition of the two's complement circuit: Using Multiplexer, the circuit is displayed in Figure 14.

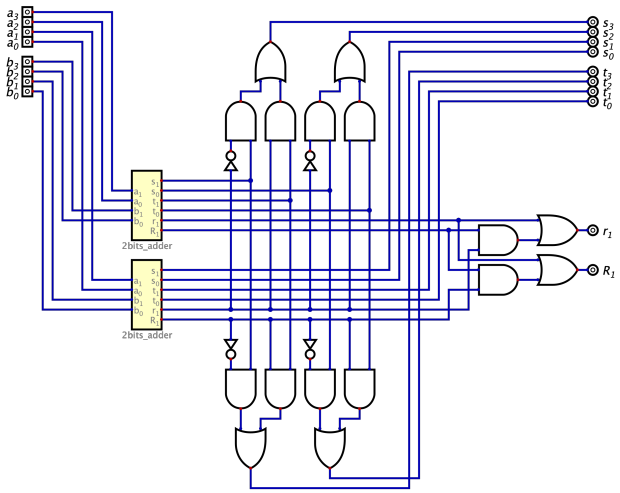

Fig. 12. Implementation of a 4-bit Adder with eight values output. The 2-bit Adder displayed in Figure 11 is utilized.

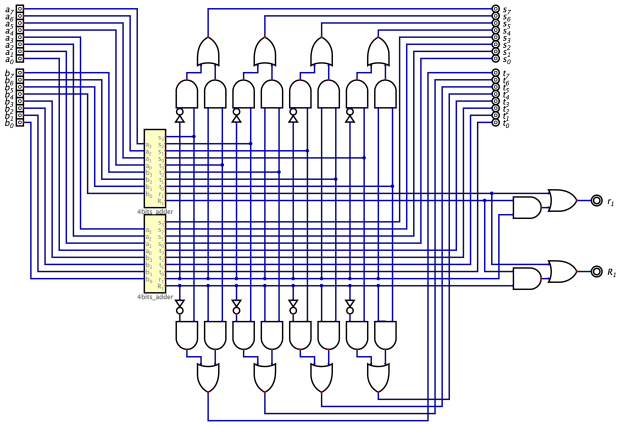

Fig. 13. Implementation of an 8-bit Adder with eighteen values output. The 4-bit Adder displayed in Figure 12 is utilized.

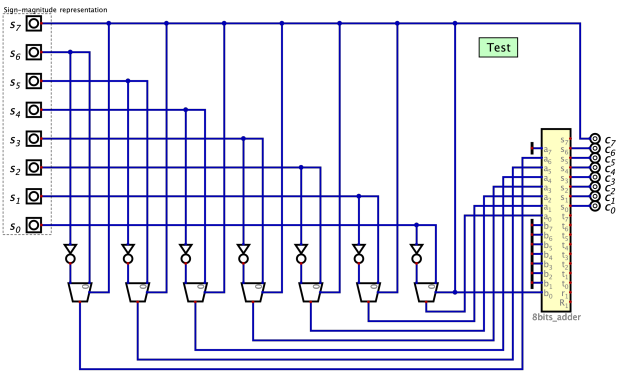

Fig. 14. Implementation of an 8-bit Adder with eighteen values output. The 8-bit Adder displayed in Figure 13 is utilized.

D. Question 4

1) Calculation for the first implementation: For an 8-bit two's complement circuit implemented in Section I-C1, to calculate the depth and complexity of the circuit, we divide the circuit into two parts:

1) In the **inversion stage**, all components are inverted simultaneously, thus depth is 1 and the number of NOT gates required is 7.
2) In the **addition stage**, the two 4-bit lookahead carry adders are connected in series. For a single 4-bit lookahead adder, its depth is 2 since one for the AND gates and one for the Half-Adders. The number of gates utilized is:

$$4 + 4 \times 2 = 12 \tag{27}$$

Since there are two lookahead carry adders, therefore, its depth is 4 and its complexity is 24.

Therefore, for an 8-bit circuit, its depth is $1+4=5$ and its complexity is $7+24=31$.

Now, for any 2^p-bit two's complement circuit, the inversion stage is unchanged: the depth is 1 and the complexity is 2^p-1 in this stage. We need to count how many 4-bit Lookahead adders are needed.

The main concept is to divide the large binary number into groups of 4-bit binary numbers. After that, we put them into series. The number of groups:

$$\begin{cases} \frac{2^p}{4} = 2^{p-2} & \text{if} \quad p \geq 2 \\ 1 & \text{if} \quad p = 0,\ 1 \end{cases} \tag{28}$$

Therefore, The depth is $2 \times 2^{p-2} = 2^{p-1}$ and the complexity is $12 \times 2^{p-2} = 3 \times 2^p$ in the addition stage.

Conclusion: For $p \geq 1$, the depth is $2^{p-1}+1$ and the complexity is $2^p - 1 + 3 \times 2^p = 2^{p+2} - 1$.

2) Calculation for the second implementation: For the circuit implemented in Section I-C2, the depth could be calculated as follows:

- In the **inversion stage**, the depth is 1 and the number of NOT gates required is 7.
- In the **addition stage**, suppose that to implement a 2^{p-1}-bit adder where $p \in \mathbb{N}^*$ and $p \geq 1$, the depth in this stage (Note: it is worthy to limit the range only in the addition stage) is d_{p-1} and the complexity is c_{p-1}. Therefore, the depth of a 2^p-bit adder in this stage is

$$d_p = d_{p-1} + 3 \tag{29}$$

since the longest route is NOT gate-AND gate-OR gate. Moreover, the complexity in this stage is

$$c_p = 2 \times c_{p-1} + 4 \times 2^p + 2 \times 2 \tag{30}$$

We then calculate the expression for d_p and c_p based on their recurrent expressions.

$$d_p = \begin{cases} 2 & \text{if} \quad p = 0 \\ 3p + d_0 = 3p + 2 & \text{others} \end{cases} \tag{31}$$

The solution for the homogeneous part of Equation 30 is $c_p^{(h)} = A \times 2^p$. Suppose the particular solution $c_p^{(p)} = k_p \times 2^p$ where the expression of k_p to be determined.

$$k_p \times 2^p = 2 \times k_{p-1} \times 2^{p-1} + 4 \times (2^p + 1)$$
$$k_p - k_{p-1} = 4 + 4 \times \frac{1}{2^p}$$

Therefore the expression of k_p is

$$k_p = k_0 + 4p + 4 \times \sum_{i=1}^{p} \frac{1}{2^i} = k_0 + 4p + 4 - \frac{4}{2^p} \tag{32}$$

Finally, from Figure 10, $c_0 = 4$ therefore

$$k_0 \times 2^0 = 4 \tag{33}$$

$k_0 = 4$ thus the expression of c_p is

$$c_p = (8 + 4p - \frac{4}{2^p})2^p = (2 + p) \times 2^{p+2} - 4 \tag{34}$$

We could validate this result by calculating the amount of gates utilized from Figure 10 to 12, where $c_0 = 4$, $c_1 = 20$, $c_2 = 60$.

Conclusion: Therefore, after considering all of the two stages, as for an 8-bit two's complement circuit, the depth is $3p + 3$ and the complexity is $(2 + p)2^{p+2} + 3$.

E. Question 5

In this section, we use the circuit of Figure 9.

(Explanation: Since we provide multiple solutions in our project, we consider selecting one of them as feasible since it is not specified in the question. To export the circuit of Figure 3 needs to change all the output labels R_1 into some other alphabets, as we have tested that it would lead to errors, saying that the name r and R collide with each other.)

We export the VHDL code based on the circuit designed in the Digital software and generate the code using gtkwave. Reminding that the input 8-bit binary number and the output 8-bit binary number are represented by:

$$\overline{s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0} \longrightarrow \overline{c_7 c_6 c_5 c_4 c_3 c_2 c_1 c_0} \tag{35}$$

The text data is listed in Figure 15.

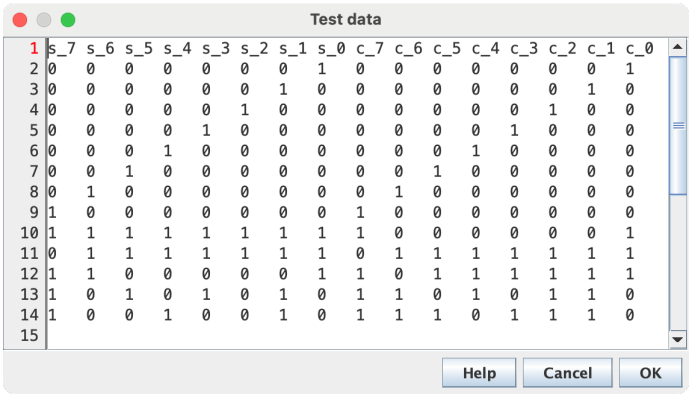

Fig. 15. Test Benchmark

As is required, all the ports/input-output have a delay of 1 unit time and all the multiplexer have a delay of 2 unit time. The corresponding simulation of signals is shown in Figure 19.

Analysis of the result:

- Setting the delay of the multiplexer as 2 and the delay of the NOT gate as 1 allows us to directly exploit the result of the NOT gate after the result is calculated. (Figure 17)
- The calculation of the two 4-bit Lookahead Carry Adder proved the fact the two adders are connected in series. One would begin to operate only after the other has finished its calculation. (Figure 18)

(An equivalent expression is shown in Figure 16. It is worthy to note that this code cannot generate VHDL code for the reason that the delay component is not implemented in VHDL (at least not in Digital software))

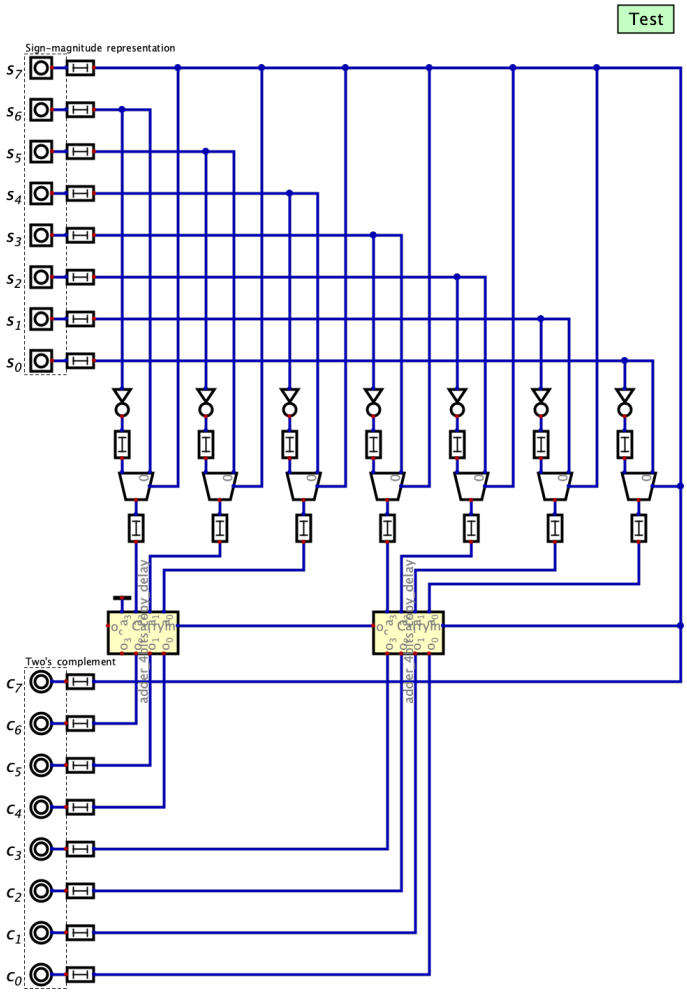

Fig. 16. Equivalent Representation of the circuit combined with delayed I/O, Gates and Multiplexers. This Representation is also useful to understand the Analysis 1. Part

II. Question 6-10

A. Question 6

To execute the operation, it is essential to:

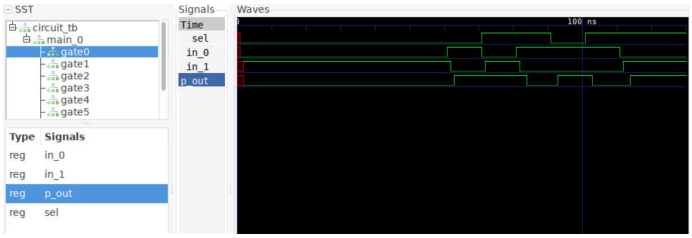

Fig. 17. Analysis 1: Observation of the time delay of the component NOT Gate + Multiplexer. After the signal is passed to the NOT Gate, the NOT Gate generates the result after 1 ns, and the multiplexer generates the result after 2 ns, which is time-efficient

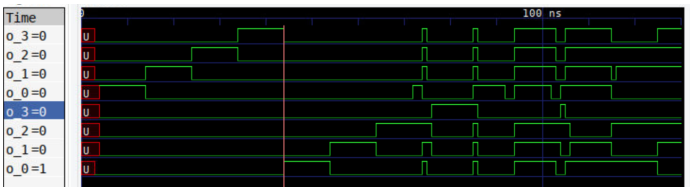

Fig. 18. Analysis 2: Observation of the time delay of the two 4-bit Lookahead Carry Adders. The upper four signals correspond to the lower 4 bits of the output. (Note that here, o_3 has no actual meaning since the highest bit should be the sign bit)

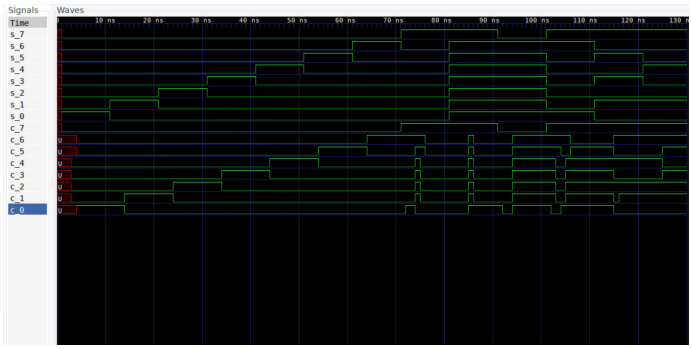

Fig. 19. Result with delay with test data in Figure 15

1) Fetch the number from the Memory
2) Load them in the Register
3) The ALU read the numbers in the Registers, and output
4) We load the output result in the Memory.

Consequently, with M_1, M_2, M_3 Memories and R_1, R_2 Registers, the commands are:

LOAD M1, R1, 0
LOAD M2, R2, 0
SUB R1, R2, R1
STORE R1, M3, 0

B. Question 7

The question proposed that four instruction should execute in series, which we will note them:

- FI: Fetch instruction
- PO: Process Operand
- EI: Execute Instruction
- WO: Write Operand

Undoubtly, the substraction command should wait before the binary numbers have been successfully stored into the corresponding Registers.

```
Cycle   1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
-------------------------------------------------------
LOAD R1 FI PO EI WO
LOAD R2    FI PO EI WO
SUB R1        FI       PO EI WO
STORE R1         FI            PO EI WO
```

Fig. 20. Pipeline to Execute the Insturctions

C. Question 8

TBD

D. Question 9

First, we will introduce several elements embedded in our implementation in the sections later on.

1) Trigger: To synchronize all the signals in the system. We need a clock signal to control all the elements. The term trigger is used to synchronize the signals with the global status of the whole system. Figure 21 compares three types of triggers based on different excitation modes. The uppermost, which is the simplest trigger, could be excited if the clock signal is high. The trigger in the middle could be excited if two consecutive high clock signals are detected, and the third trigger could be excited if two consecutive low clock signals are detected. Corresponding output is shown in Figure 22.

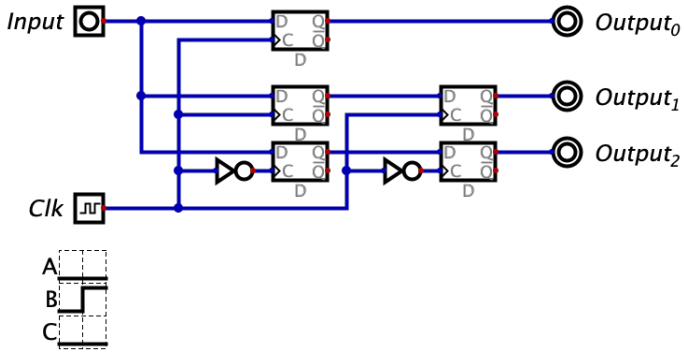

Fig. 21. Triggers of different modes

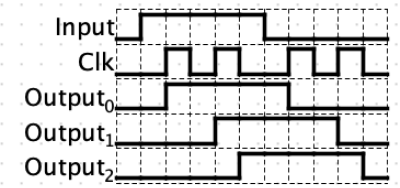

Fig. 22. Triggered output figure of different triggers

In our implementation, an **impulse signal** is essential so as to enable the operations. For example, it is utile when we wish to execute the read command to a certain Memory block

and to isolate it or switch if off in order for saving energy consumption.

Using the property of a D Flip-Flop Trigger, we output 1 and then 0 in series. Detailed implementation could be found in Figure 23.

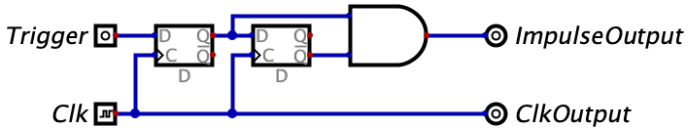

Fig. 23. Impulse Signal Trigger

2) Memory: A real memory should consist of several small blocks of storage, each used to store its own numbers. In our implementation, a large memory contains three smaller memory blocks, each capable of storing an 8-bit number.

We proceed step by step.

An 1-bit Memory is implemented in Figure 24. To store a single bit, a D Filp-Flop Trigger is utilized. First of all, this 1-bit Memory should be Selected (the input of Operate input should be 1) to function. When the Read/Write input is 1, the write mode is initiated, and the input is written into the memory. Correspondingly, when the Read/Write input is 0, the read mode is initiated, and the value stored in the memory is output. The Operate signal should be an impulse signal (which would be implemented later on) in order to automatically switch off the operation mode of the memory after the operation has been executed.

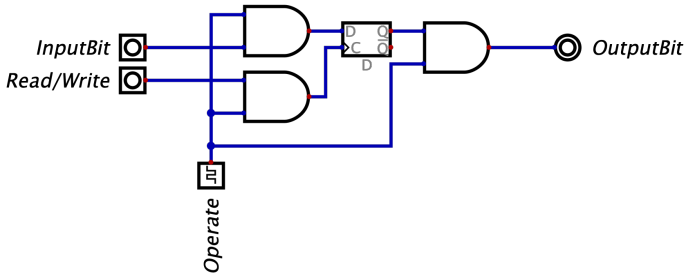

Fig. 24. Implementation of 1-bit Memory

To store a 8-bits Memory, we parallize the process of storing 8 1-bit binary numbers, which is shown in Figure 25.

The final structure of the 3-blocks Memory should have a clock cycle signal input to facilitate synchronization when connected to a circuit and to enable sequential execution of instructions. In our implementation, we need the clock to indicate when internal operations are complete. Since the memory contains three smaller memory blocks, the input needs to specify which block should store or retrieve the number. Traditionally, this should be done by reading the address of the memory block, but we will simplify the process here by directingly reading the indice. After type-in the number to be stored and the memory block to be written in or to be read, when the value of the clock input is 1, it begin to read to the memory block or export the stored value to the output. The implementation is shown in Figure 26.

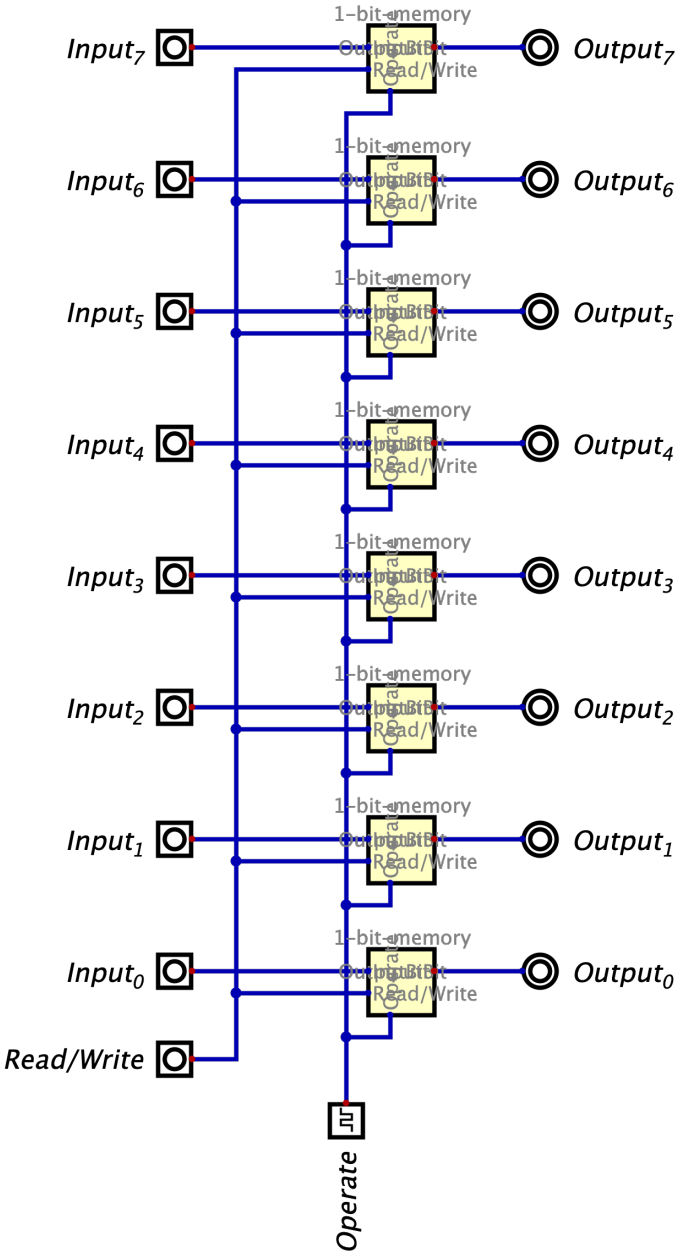

Fig. 25. Implementation of a single 8-bit Memory Block, the 1-bit Memory shown in Figure 24 is utilized.

3) Register: A 8-bits register is simply bringing together 8 blocks of register, controlled by a clock signal and an enable signal. The register is implemented in Figure 27.

4) Store Instruction: To store some binary numbers, we need the infomation of:

- The 8-bit binary number to be input
- Which block of the memory should it be stored into

We design the circuit by adding an D Filp-Flop Trigger to synchronize the operation with the global clock. Consider the case of storing data into any one of the three memories. The instruction circuit is shown in Figure 29.

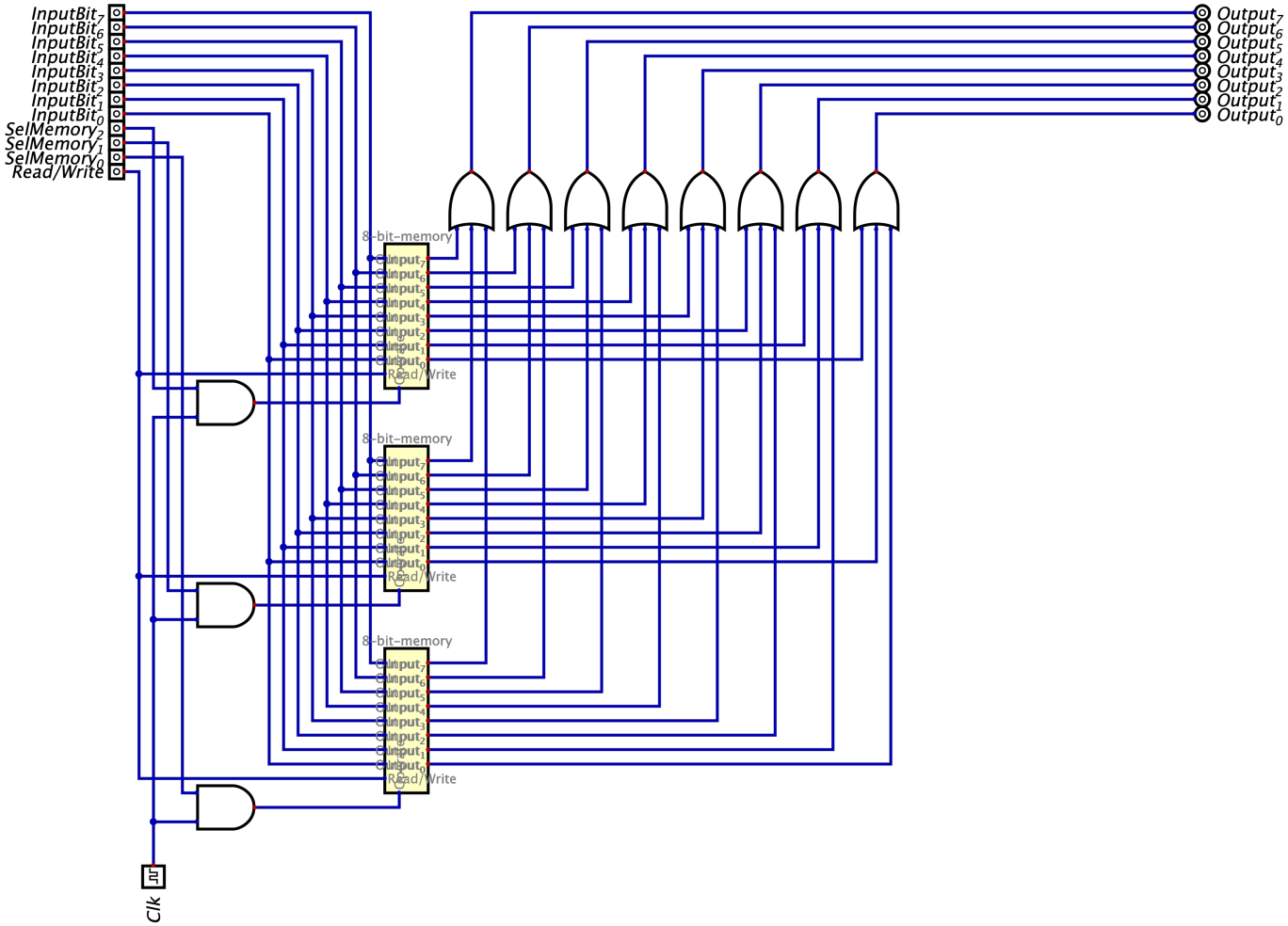

Fig. 26. Implementation of a 8-bit Memory, three blocks of a 8-bit Memory shown in Figure 25 is utilized.

5) Load Instruction: First, we implement the LOAD instruction from one single memory to one single register, ignoring all the unused parts.

To make the circuit work, we should first STORE some valid binary numbers then LOAD the numbers from the memory to a certain register.

Suppose that we have prepared the STORE and LOAD instructions in separate places. Ensuring the sequential order by applying a D Flip-Flop trigger, and connect to the memory block using a multiplexer to determine which block of memory should be visited depend on different instructions.

The implementation of the STORE and LOAD instructions to all of the memories and all of the registers is implemented in Figure 30.

E. Question 10

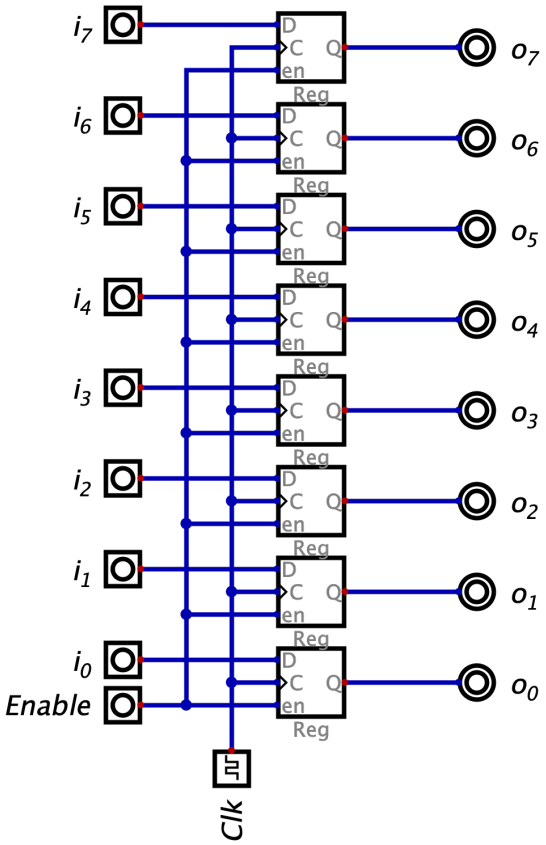

Fig. 27. Implementation of a 8-bits Register.

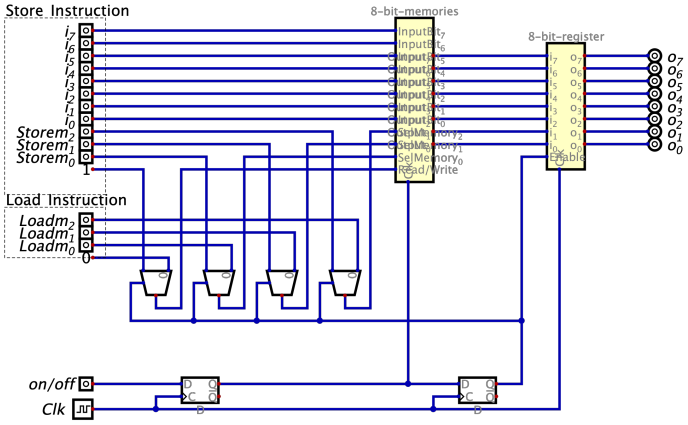

Fig. 28. Implementation of the LOAD instruction from a single memory to a single register.

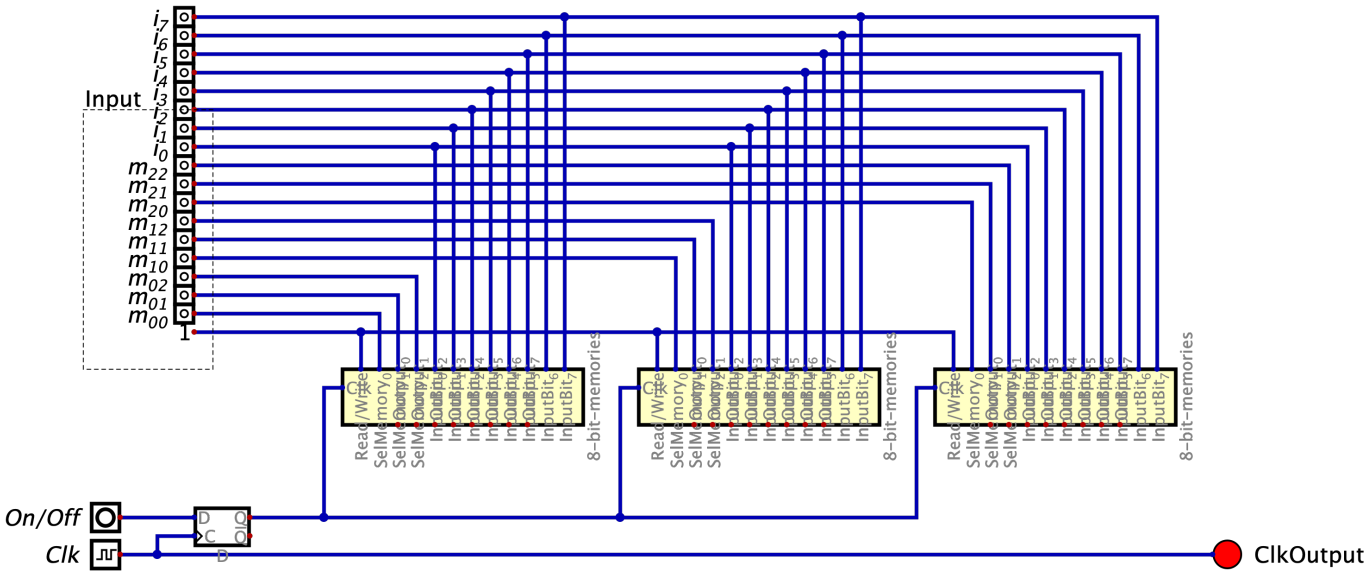

Fig. 29. Implementation of the STORE instruction to all of the memories.

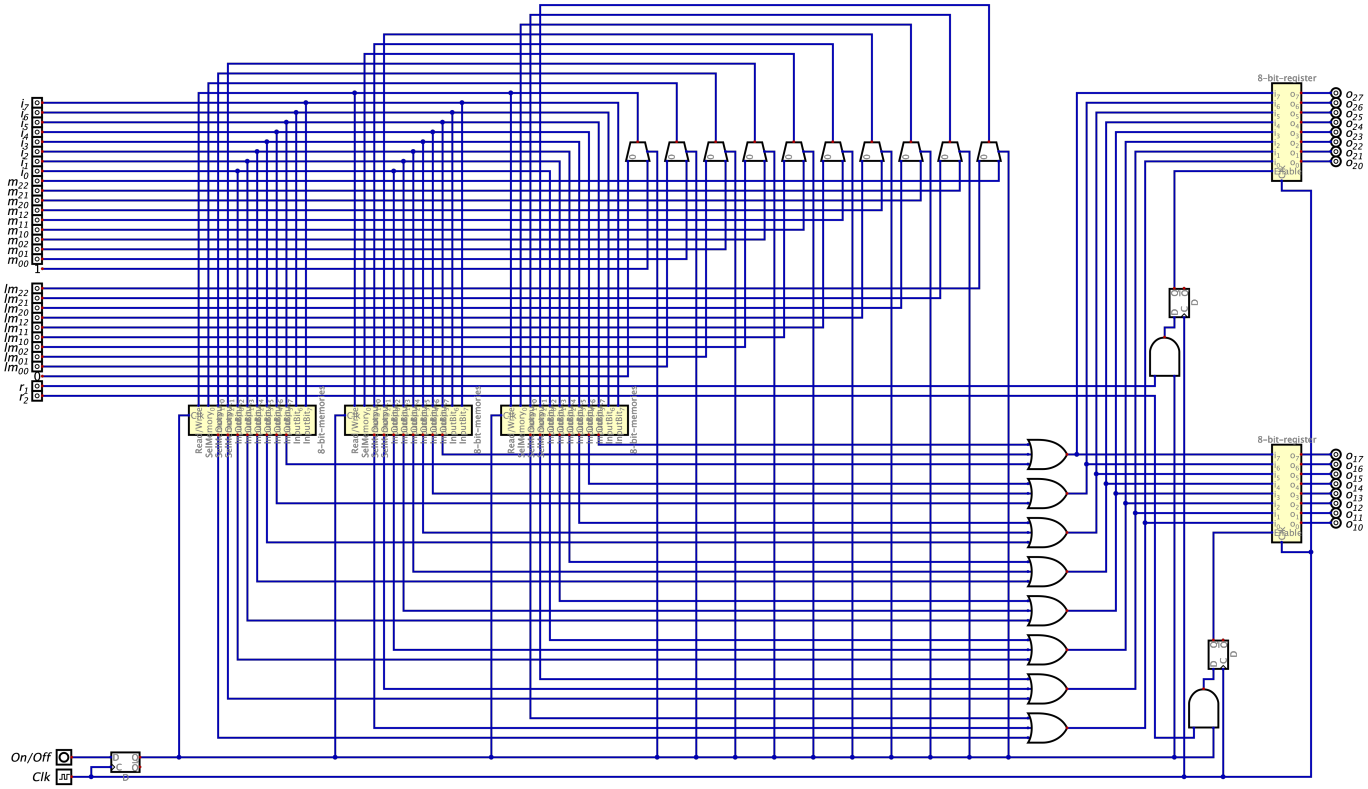

Fig. 30. Implementation of the LOAD instruction to all of the memories and all of the registers.