# Four-Bit ALU Trainer
# Detailed Design Document

Author: Rodney Moffitt

## ABSTRACT

This document describes the detailed design for the Four-Bit ALU Trainer.

## TABLE OF CONTENTS

## 1 INTRODUCTION

This document will describe the detailed design for the ‘Four-Bit ALU Trainer Project’ which has been taken up by Rodney Moffitt as the project course ELG4905 for the semester of January - April 1995.

The ‘4-Bit ALU Trainer’ is a device that will provide students with the capability to explore a hardware implementation of a multiply and divide capable arithmetic processor. This processor uses a micro-program sequencer to generate the necessary control signals which drive logic to perform complex instructions. In the simplest sense a micro-programmed sequencer can be just an EPROM with the address as the micro-instruction ‘Op-code’ and the data as the control signals.

Such a system is required to implement complex instructions such as multiply and divide which require a number of ‘simpler’ instructions in order to execute. For example addition and subtraction requires only an ALU, yet multiply requires not only addition yet also shifting and some sort of sequencer to control these operations. Therefore the trainer shall provide for all control signals based on algorithms to be presented later in this document.

Multiply, divide and some other instructions will be the ‘instructions’ that the trainer will execute. The user needs only set-up the data and instruction registers and then toggle a clock signal. The user is notified of completion of the instruction with the ‘COMPLETE’ signal. Each of these instructions will be composed of a number of individual micro-instructions which are mapped from the address of the control memory.

## 2 ARCHITECTURE OVERVIEW

The following diagram outlines the main parts of the overall architecture of the ALU Trainer.

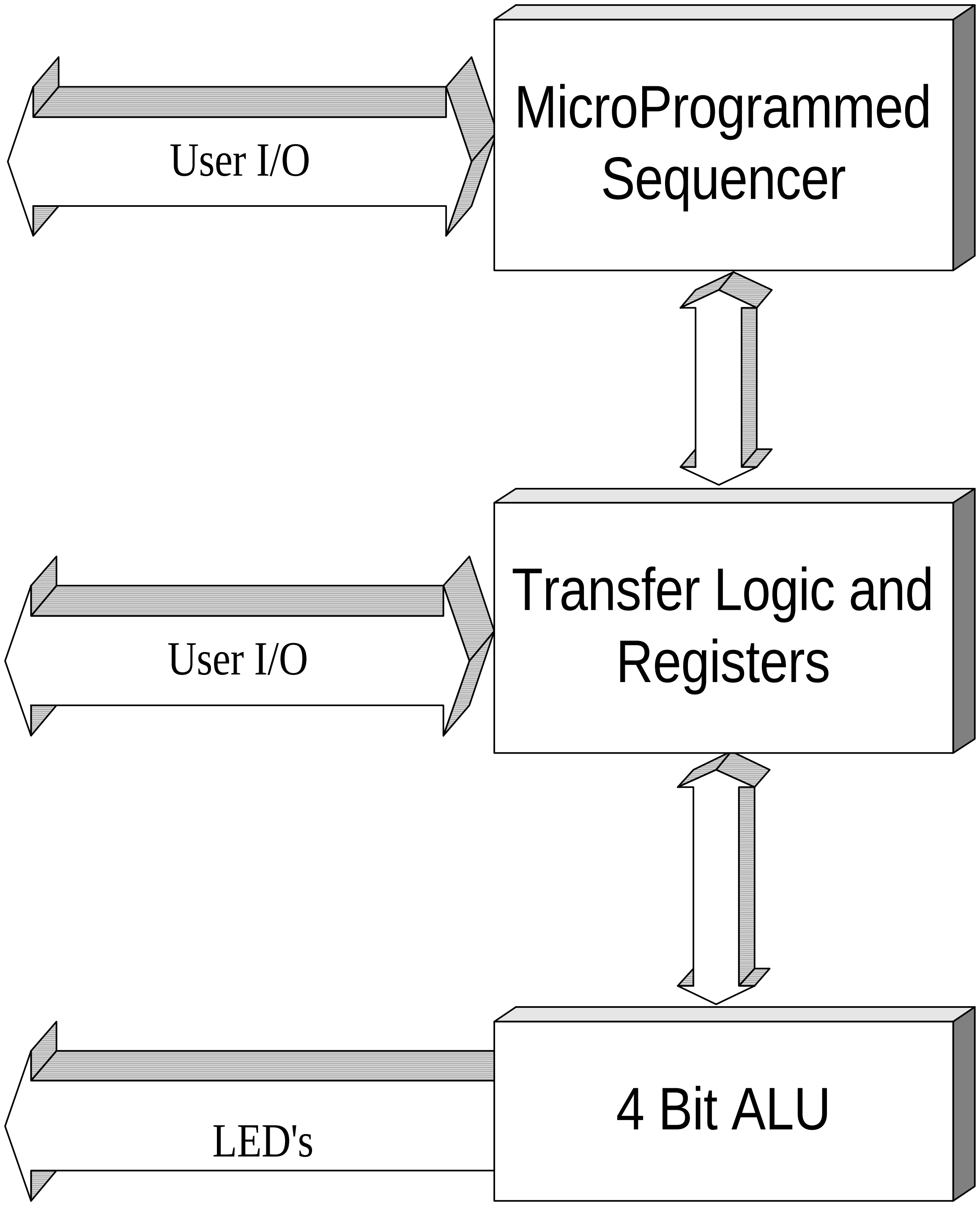

**Figure 1:** Main Parts of the Overall Architecture

Three major parts that make up the ALU Trainer are:

1. The Micro-programmed Sequencer: This mainly consists of an EPROM based micro-program sequencer and associated logic. The user can set the instruction to execute through the IR (Instruction Register).
2. Transfer Logic and Registers: The ‘linking’ logic between the ALU and the registers provides for the next main part of the trainer. This includes response drivers for the micro-program control signals and busses to connect the registers. User input via the ALPHA, GAMMA and GAMMA registers are available, plus LED outputs of the registers and control signals.
3. 4 Bit ALU: The arithmetic and logic block provides instructions based on signals from the micro-instruction take as input the value of A (Accumulator) and B (secondary input register) and channels the output back to the A register and E (carry) registers.

The remainder of this document shall provide the necessary details of the design of the ‘4 Bit ALU Trainer’ based on the above model.

## 3 MICRO-PROGRAM SEQUENCER

The micro-program sequencer can be seen in Figure 2. This part of the trainer is responsible for the following:

- Initiating a ‘instruction’ (such as divided, multiply, etc.) entered by the user in IR.
- Determining the next micro-instruction to enter into the CAR register (which in turn becomes the address for the control memory).

The following elements are involved:

1. ***RESET***: The RESET input is set high to initialize the trainer to start a instruction. When RESET is high the CAR MUX selector input S goes high and the SC PE (Parallel Enable) goes high. This causes the value of IR to be channelled to CAR, and SC to be loaded with the initial counter value. ‘R/F’ is the OR of RESET and FINISH which is used in a number of places in the trainer. More details on these signals can be found in the section ‘CLK Timing’.
2. ***COMPLETE***: The COMPLETE output indicates to the user that the current instruction (indicated by the value of IR) has completed. It goes high whenever SC has completed decrementing down to zero, in other words all of the required iterations of the instruction has completed. More details on this signal can be found in the section ‘CLK Timing’.
3. ***FINISH***: The FINISH output is similar to COMPLETE except it stays high until the next falling edge of CLK rather than the next rising edge, as COMPLETE does. This signal will be mentioned more often than COMPLETE since the discussion will have to do mostly with the actions of the combination logic rather than what the user will see when execution of a instruction is complete (which will just be an LED driven by the COMPLETE signal).
4. ***CLK***: The CLK input provides synchronization for the trainer, which is toggled by the user. The CAR is the only register that latches on the rising edge, while the remaining registers (E, A, Q and SC) latch (or in the case of SC, decrement) on the falling edge of CLK. More details on this element can be found in the section ‘CLK Timing.
5. ***IR***: The user enters an 8 bit value representing the address of the first micro-instruction of the instruction to be executed in the control memory. IR drives the input of the ‘I1’ input of the CAR MUX, therefore when this multiplexor selector input is ‘1’, the value of IR is channelled to CAR to become the new address of the control memory.
6. ***MUX***: the CAR MUX determines which of either the ‘FULL ADDER OUTPUT’ or ‘IR’ shall be used as the new value of ‘CAR’. The selector input ‘S’ determines this, if S=0 then the FULL ADDER OUTPUT is used, else the value is channelled from IR. The S input is the OR of ‘RESET’ and the control signal ‘DSC’, which means that the ‘I1’ input from IR is channelled to the CAR ONLY when RESET OR DSC are set.

The MUX will be implemented using a pair of 74157 ‘Quad 2-Input Multiplexors’.

7. ***CAR***: The Current Address Register holds the address of the micro-instruction that is currently being ‘executed’. In other words it hold the address of the control memory, which determines the control signals. The falling edge of CLK latches CAR to the output of MUX.

   The CAR will be implemented using a pair of 74175 ‘Quad D Flip-Flops’.

8. ***Control Memory***: The mapping of the address to the control signals is done by the control memory which is implemented as a pair of 8-bit EPROMs, for a total of 16 bits available for the micro-instruction control signals. More details on this element can be found in the section ‘Micro-instruction Format’.

9. ***FULL ADDER***: An 8-bit full adder serves to assist in determining the next address for the CAR. The inputs are the current value of CAR and the output of the ‘Next Address Logic’, which is a value representing the number of micro-instructions to skip based on the value of control signals and registers (see the section ‘Branching’ for more details). Basically it just takes the CAR and adds some value to it, which can be in the range of ‘+1’ to ‘+6’. The output drives the ‘I0’ input of the MUX.

   A pair of 7483 ‘4-Bit Binary Full Adders’ will be used.

10. ***SC***: The Sequence Counter keeps track of the number of iterations that have occurred of a instruction. This is needed since complex instructions such as multiply and divide are executed as loops, where after every loop SC is decremented. When SC is zero the instruction is finished and the next instruction in IR can be executed.

    SC is a counter and therefore requires a clock. All instructions of this register occur upon the falling edge of the clock signal CLK.

    SC is loaded when PE (Parallel Enable) is a ‘1’. The value loaded depends on IR (since not all instructions require the same number of iterations).

    SC is decremented when ‘CEP’ is ‘1’. The ‘DSC’ control signal drives the CEP input, therefore SC is decremented only when the current micro-instruction has DSC = 1.

    Only one of these inputs can be high when CLK falls, otherwise an unknown state will result.

    When SC has decremented down to zero the TC (Terminal Count) output goes low. This drives the ‘FINISH’ output high indicating to the user that another instruction can be loaded into IR. DSC = 1 also sets the MUX to ‘point’ to IR, forcing CAR to load IR once again. TC also drives the PE input of the SC. Since all inputs are synchronous with CLK the SC value does not change until CLK falls again, which at that time a new value is loaded since PE = 1.

    A 74169 ‘Modulo 16 Binary Synchronous Bi-Directional Counter’ will be used to implement SC. SC requires values of ‘1’ through ‘5’, so the 0-15 range will be more than adequate.

11. ***Next Address Logic***: The amount to add to CAR, i.e. the ‘B’ input to the Full Adder is determined by this logic. It takes into account the values of E (the carry of the ALU), R/F, the control signal ‘CB’, whether a left-shift micro-instruction is being executed and IR. More details on this element can be found in the section ‘Branching’.

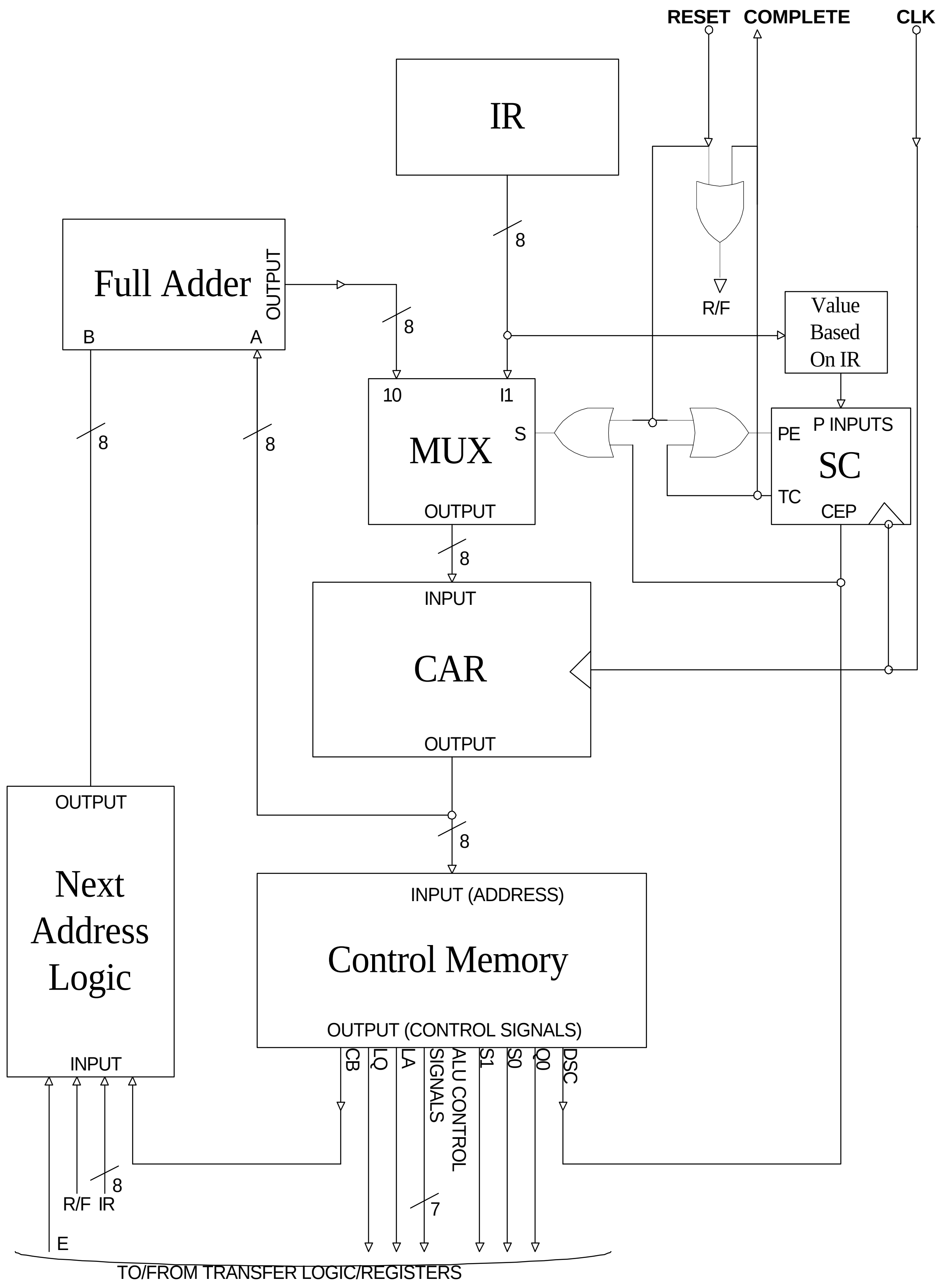

**Figure 2:** The Micro-program Sequencer

## 4 TRANSFER LOGIC AND REGISTERS

The diagram of the 'transfer logic/registers' can be seen in Figure 3. This part of the trainer is responsible for the following:

- Decoding of the micro-instruction signals.
- Provide path to and deal with the operation of the following registers
    - i. A (through the shifting network of EAQ and interface with the ALU).
    - ii. E (through the shifting network of EAQ and the output of the ALU).
    - iii. Q (through the shifting network of EAQ and provide setting of the LSB 'Q0' bit of Q).

The following elements are involved:

1. ***E***: The E register holds the 'carry' output of the ALU and the MSB of A during left-shifts. The 'D' input is latched on the falling edge of CLK. There are two separate times when E changes and this is seen by the two AND gates driving an OR gate to the input D:

    i. When left-shifting, take as input the MSB of A. When left-shifting, the S1, S0 control-signals are 1,0, respectively. Therefore when the MSB of A is '1' and left-shifting there should be a D input of a 1. This occurs when S0 = 0, S1 = 1, $A_3$ = 1 which is one of the signals that drives the OR gate to the 'D' input.

    ii. When an ALU operation takes place the output 'CO' of the ALU needs to be stored in E. S0 = 1 when A is parallel loading from the output of the ALU. Therefore when S0 = 1 and CO = 1 the D input should be a 1, which is the other signal driving the OR gate.

    A problem could occur since S0 is also 1 during a right-shift, therefore E can be loaded with the value of CO. This is of no concern since right-shifting only occurs for multiply, and the value of E is altered right after the right-shift micro-instruction during multiply, by an arithmetic-ALU based operation. So it never has a chance to have any ill-effect (see the section 'The Multiply Algorithm' for more details).

    A 7474 'D-Type Positive Edge-Triggered Flip-Flop' will be used to implement E.

2. ***ALPHA***: The user places the data intended for A in this register. A loads from this at the first micro-instruction of the instruction multiply. A simple dip switch will be used to allow the user to enter the value of ALPHA.

3. ***A***: The Accumulator register holds the result of operations using the ALU, including right and left-shifting. A loads on the falling edge of CLK. There are five modes of operation for A:

    i. Parallel loading of the output of the ALU when S1,S0 = 1,1. A takes the 'F' output of the ALU on the falling edge of CLK.

ii. Right shifting occurs when S1,S0 = 0,1. The $D_{SR}$ input takes the value of E on the falling edge of CLK.

iii. Left shifting occurs when S1,S0 = 1,0. The $D_{SL}$ input takes the value the MSB of Q on the falling edge of CLK.

Pre-loading. A must be loaded from ALPHA during the first micro-instruction of an iteration of an instruction. This occurs when LA = 1 and R/F = 1. R/F must also be 1 since each instruction does a number of iterations of the micro-instructions that make up the instruction. Since the first micro-instruction has LA = 1, the A register would keep loading ALPHA for each new iteration and nothing would be accomplished. Therefore only when R/F = 1 (i.e. when a instruction has just started) and LA = 1 should ALPHA be loaded into A. The MUX selector input S only goes to 1 when both LA and R/F are 1, which then channels ALPHA into the input of A, and forces a parallel load (by setting the S1,S0 to 1,1).

A 74194 ‘4-Bit Bi-Directional Universal Shift Register’ will be used since it has both parallel loading and shifting capabilities, and does so for 4 bits, as required.

4. ***BETA***: The Secondary Input register holds the value that is added/subtracted with A. Since it never changes there is no need for a buffer between the user input and the ALU. A simple dip switch will be used to allow the user to enter the value of BETA.

5. ***GAMMA***: The GAMMA register provides the same functionality as ALPHA does for A, yet for the Q register.

6. ***Q***: The Quotient register is used for both the multiply and divide algorithms and has the same functionality as A except that the MUX ‘I0’ input does not come from the output of the ALU, yet from the ‘SET Q0=1’ logic. The divide algorithm requires that the LSB of Q be set to 1 for certain conditions. This occurs when the control memory bit ‘Q0’ is high, then the ‘I0’ input will be a copy of Q except having the LSB forced to ‘1’. Therefore if Q0 = 1 (a control signal) and S1,S0 = 1,1 then Q will load this new value and Q will be altered as the LSB is set to a 1.

   A 74194 ‘4-Bit Bi-Directional Universal Shift Register’ will be used to implement this register.

7. ***MULTIPLY LOGIC***: The signals that control the ALU first pass through the ‘MULTIPLY LOGIC’ at the top of Figure 3. This logic is required since the multiply algorithm requires two special features that are not supplied by separate control signals (see the section ‘The Multiply Algorithm’ for further details):

   i. When CAR = the first address of multiply and R/F = 0, compliment the fifth ALU control signal.

   ii. When CAR = the first address of multiply + 1 and the LSB of Q ($Q_0$) is 0 compliment the third ALU control signal.

The output of this logic then becomes the signals that control the operation of the ALU.

7. ***ALU***: The ALU takes the ‘A’ input from the output of the A register and the ‘B’ input from the output of the BETA register. The ‘F’ output then drives the ‘I0’ input of the A-MUX. The CO (carry out) is used in the logic that determines the value of E. See the section ‘4-Bit ALU’ for more details.

8. ***STATUS REGISTER***: This register displays the status of A (Z: if it is zero, S: the MSB, sign-bit), E (C: the carry) and if an overflow condition occurred (OF, only valid for the divide operation). These signals are valid only when COMPLETE is high and the clock is high, and are ‘High-Z’ at all other times. This is so as in a more elaborate system they would only be needed when the ‘arithmetic-processor’ had completed it’s operation. Z and S are derived from logic based on A, C comes directly from E and OF is a special case since it is only affected by the divide operation (see the section ‘The Divide Algorithm’ for more details).

| S1 | S0 | OPERATION |
|---|---|---|
| 0 | 0 | HOLD |
| 0 | 1 | SHIFT RIGHT |
| 1 | 0 | SHIFT LEFT |
| 1 | 1 | PAR-LOAD |

Z: Accumulator = 0
S: MSB of Accumulator
C: Value of E
OF: Divide overflow

**STATUS REGISTER**

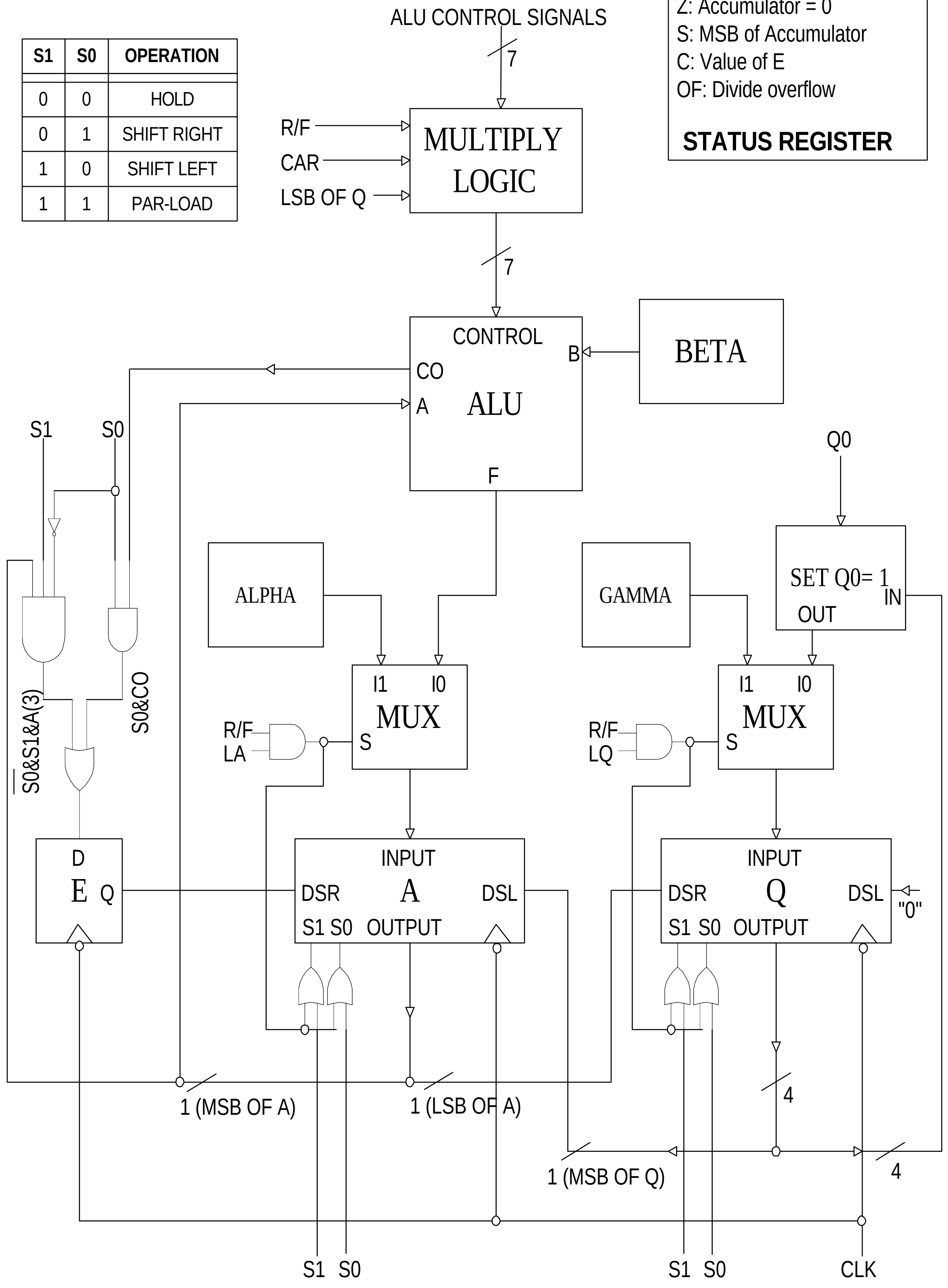

**Figure 3:** Transfer Logic/Registers

## 5 4 BIT ALU

The ALU consists of 4 ‘Single-Bit Arithmetic-Logic Circuits’ cascaded together. The logic for a single bit can be seen in Figure 4 below.

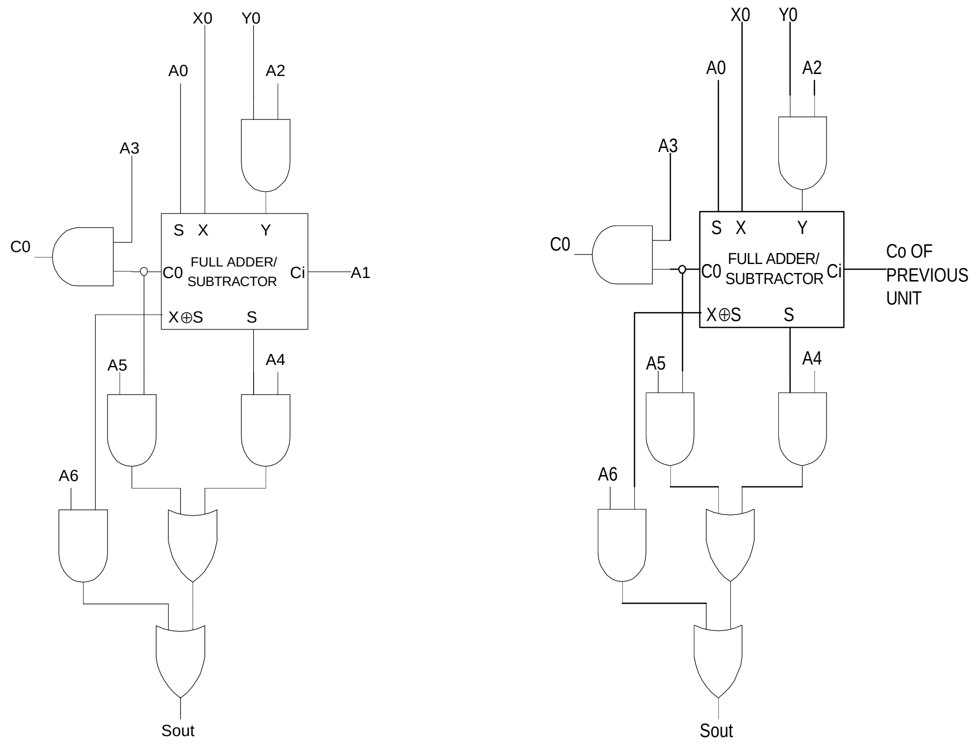

**Figure 4A and B**: Single-Bit Arithmetic-Logic Circuits

Figure 4A (left) shows the first stage of the arithmetic-logic circuit, where the remaining three stages are of the type depicted in Figure 4B (right). The only difference between the two is that the first stage has the ‘Ci’ (carry-in) set to the ALU control signal A1, where the rest have the output of the previous Co (carry-out) as the input to Ci.

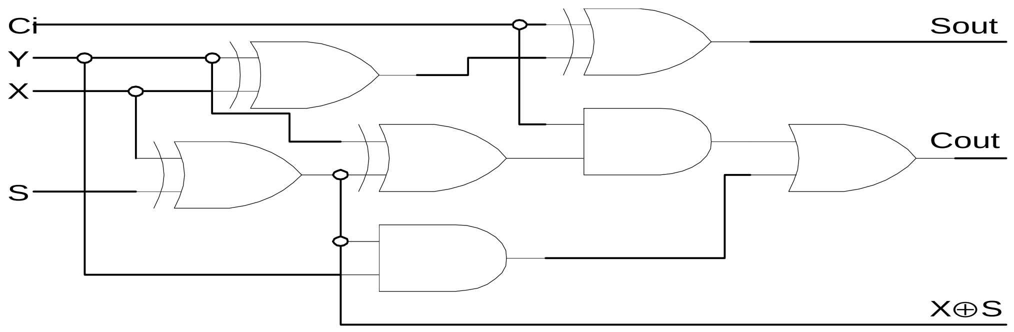

**Figure 5**: Full Adder/Subtractor Circuit

Within each of the single-bit arithmetic-logic circuits is a full adder/subtractor which can be seen in Figure 5.

The ALU control signals determine how the output and carry are generated by the full/adder subtractor. In the most simplest form, a full adder/subtractor can be set up with AND and OR gates to enable it to have more than the basic add and subtract capabilities of the full adder-subtractor.

With the addition of the ALU control signals the arithmetic-logic circuit can now have additional instructions as detailed in Table 1. Note that this is not the most efficient method for designing an ALU, yet required due to the complexity of the multiply instruction (see the section 'The Multiply Algorithm' for more details). The 'x' values in the A6 through A0 columns represent 'don't care' inputs.

| Instruction | A6 | A5 | A4 | A3 | A2 | A1 | A0 |
|---|---|---|---|---|---|---|---|
| X + Y | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| X - Y | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| X + 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| X - 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| X \|\| Y | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| X & Y | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| X ⊕ Y | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| -X | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | x | x | x | 0 |
| X | x | 0 | 1 | x | 0 | 0 | 0 |

**Table 1**: ALU Instructions

Table 1 summarizes the ALU control signals and how they affect the arithmetic-logic circuits to enable the simple full-adder/subtractors to behave as different arithmetic-logic circuits.

Table 2 shows how each of the ALU control signals affects the arithmetic-logic circuit of the ALU. Each signal is used for one particular instruction, yet the combination of a number of them produces more complex instructions. The value in brackets '( )' details when it is 'active'.

| **ALU Control Signal** | **Use** |
|---|---|
| **A0** | Controls whether the full adder/subtractor acts as a full adder (0) or a full subtractor (1). |
| **A1** | Input Ci of first stage only. |
| **A2** | Blocks the Y input (0), passes the Y input (1). |
| **A3** | Blocks Cout(0), passes Cout (1). |
| **A4** | Blocks S to Sout (0), passes S to Sout (1). |
| **A5** | Blocks Co to Cout of stage (0), passes Co to Cout of stage (1). |
| **A6** | Blocks X ⊕ Y to output (0), passes X ⊕ Y to output (1). |

**Table 2**: ALU Control Signal Descriptions

## 6 MICRO-INSTRUCTION FORMAT

The following table details the format of the micro-instruction:

| 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OF | CB | LQ | LA | A6 | A5 | A4 | A3 | A2 | A1 | A0 | S1 | S0 | Q0 | DSC |

**Table 3** Micro-instruction Format

Fifteen bits are used therefore requiring 2 8-bit EPROMs to provide the control memory.

There are seven ‘fields’ that make up the micro-instruction with each having between one and seven bits (individual control signals):

1) ***OF***: When a micro-instruction has the ‘Overflow’ bit set, the overflow bit of the STATUS REGISTER is latched to the compliment of E on the falling edge of CLK. This is used by the divide instruction to indicate to the user an overflow condition has occurred. Since the STATUS REGISTER only displays after the instruction is complete, a temporary holding register is used to keep the value until the instruction is complete.

2) ***CB***: The CB determines the type of branching through the Full Adder - CAR MUX logic (Figure 2). See the section ‘Branching’ for more details.

3) ***LQ, LA***: These flags force the Q and A registers to load from the GAMMA and ALPHA registers, respectively.

   To demonstrate, the A register loading will be explained. If LA is set (and R/F is also set, indicating the first micro-instruction of an instruction) then the selector bit for the A multiplexor (Figure 2) is set. The AND of LA and R/F also forces the S1,S0 to 1,1, of A which means that at the next falling edge of CLK, A will load from ALPHA.

   The same procedure applies to Q. They are loaded separately since some instructions don’t require both, and specifically the multiply instruction only loads Q.

4) ***A6 through A0***: The ALU control signals A6-A0 determine the type of operation that the ALU performs. These control signals first pass through the ‘Multiply Logic’ (Figure 2) since some special conditions apply to these control signals when the multiply instruction is being executed (see the sections ‘4 Bit ALU’ and ‘The Multiply Algorithm’ for more details).

5) ***S1, S0***: The A and Q mode selector flags S1, S0 control the type of latching that occurs for A and Q. These registers are falling edge triggered latches yet latch in various modes, based on the table at the top of Figure 3. For example if S1,S0 = 0,1, then A and Q are right-shifted on the falling edge of CLK. All arithmetic operations involving A loading the result must have S1,S0 = 1,1 (i.e. parallel load) which causes

A to load the result from the 'F' output of the ALU. Since LQ and LA force S1,S0 to 1,1, (for Q and A respectively) therefore the loading of A and Q from ALPHA or GAMMA and a shifting operation (which require S1, S0 to be other than 1,1) can not occur simultaneously. Yet a pre-loading of Q and an ALU operation can occur at the same time (which is done in the multiply algorithm) since S1,S0 = 1,1 for either of these to occur.

6) ***Q0***: When this bit is set the MUX input 'I0' for Q (Figure 2) has the value of Q with the LSB ORed with 1, in other words the same value as Q yet the LSB is forced to a 1. This operation is required in the divide algorithm.

   In order for Q to load this new value, S1,S0 must be 1,1 (to force a parallel load) then the ALU must have the logic operation A $\Leftarrow$ A since S1,S0 = 1,1 will cause A to load from the 'F' output of ALU at the same time. LQ must also be 0 (otherwise the MUX selector bit for Q will 'point' to GAMMA and not the new Q value). Therefore a number of limitations are imposed on the use of this flag, except if A is to also load from the ALU in a single micro-instruction.

7) ***DSC***: This bit is set in the last micro-instruction of an iteration of an instruction. As mentioned before the complex instructions multiply and divide require a number of iterations/loops of the algorithm to complete the instruction, so the counter SC is used. DSC = 1 causes SC to be decremented by one on the falling edge of CLK. When SC = 0 then FINISH and COMPLETE goes high and the instruction is complete. For non-complex instructions SC is just 1 initially, and therefore a single iteration is executed.

# 7 MICRO-INSTRUCTION CALLING CONVENTION

## 7.1 EXECUTING AN INSTRUCTION

IR is loaded with the address of the first micro-instruction of the instruction that is to be executed. On the falling edge of CLK CAR latches the value of IR and SC is loaded with a value which represents the number of iterations to execute of the instruction pointed to by IR. For all but the two complex instructions, multiply and divide, SC is loaded with 1 (since only one iteration is required). An example is ‘ADD’ which only requires one addition. The complex instructions must execute a number of similar additions including shifting. This requires a counter which the SC register acts as.

After a single iteration of any instruction SC is decremented, via the ‘DSC’ control signal. DSC = 1 also causes CAR to reload IR causing another loop/iteration to be executed. When SC = 0 FINISH and COMPLETE goes high causing IR to be loaded again into CAR on the next rising edge, starting a new instruction.

An external circuit can be used to load IR, ALPHA, BETA and GAMMA from some ‘expression stacks’ when COMPLETE rises, since this signal indicates the completion of the arithmetic instruction.

Figure 6 shows the sequence of events to execute an instruction. A number of ‘iterations’ are executed, based on the instruction.

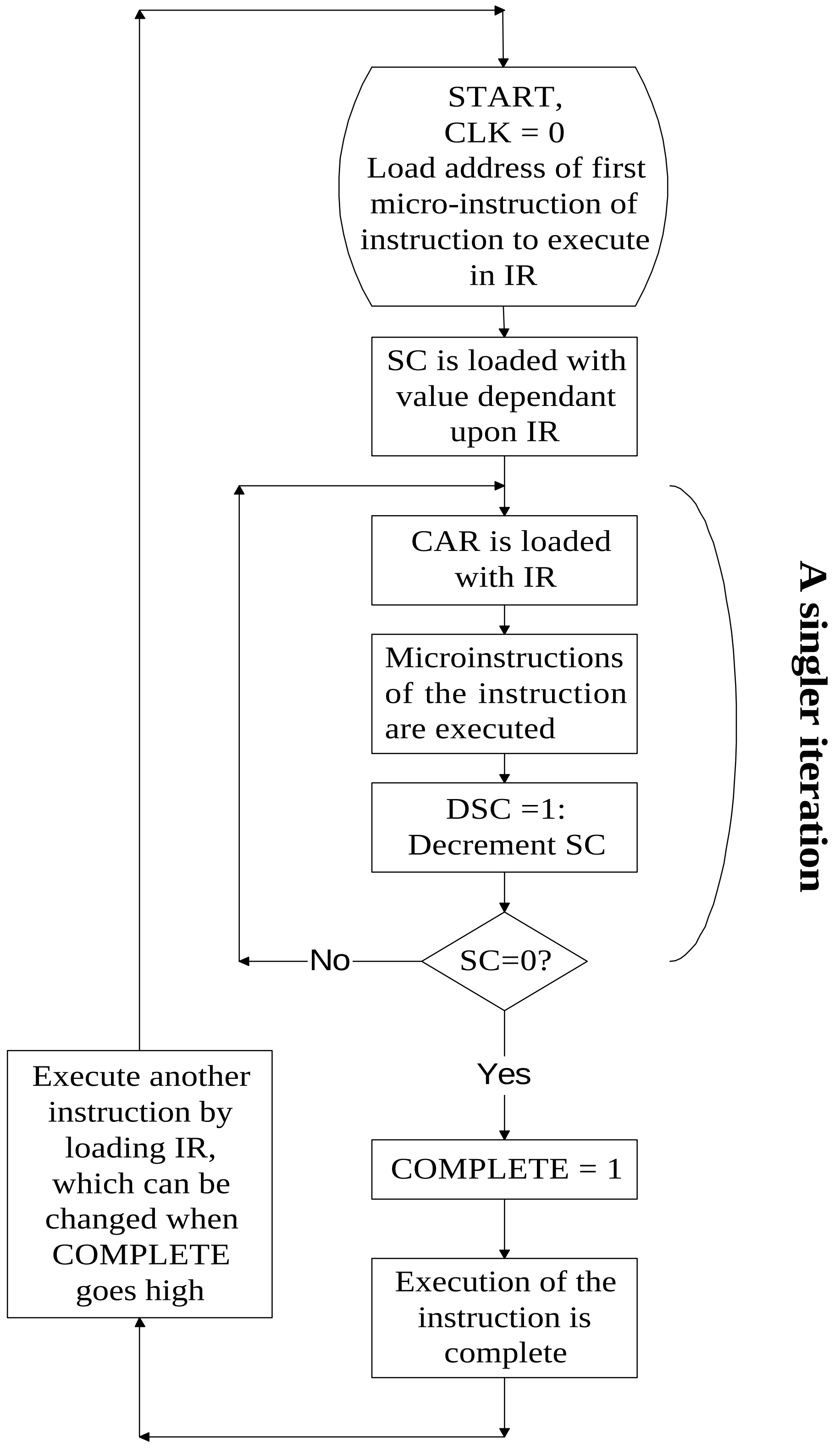
START,
CLK = 0
Load address of first micro-instruction of instruction to execute in IR
SC is loaded with value dependant upon IR
CAR is loaded with IR
Microinstructions of the instruction are executed
DSC =1: Decrement SC
SC=0?
No
Yes
COMPLETE = 1
Execution of the instruction is complete
Execute another instruction by loading IR, which can be changed when COMPLETE goes high
A singler iteration

**Figure 6**: Sequence For Executing An Instruction

## 7.2 BOOTING THE TRAINER

When the trainer is first turned on the CAR holds an unknown values and therefore subsequent CLK pulses will cause unexpected micro-instructions to be executed. To guard against this the trainer must be 'booted' or started in the following manner:

1. Turn the trainer OFF.
2. Set CLK = 0.
3. Turn the machine on.
4. Set RESET = 1.
5. Enter the address of the instruction to execute in IR.
6. Set CLK = 1.
7. Load ALPHA, GAMMA and BETA as needed.
8. Set CLK = 0.
9. Set RESET = 0.
10. Subsequent CLK pulses will execute the remainder of the instruction.

The trainer must first be turned off as CLK must not fall while it is on, otherwise the A, E and Q registers may load (depending on the values of CAR when turned on, which will be random). With CLK = 0 when the trainer is turned ON, then RESET is set to 1. This forces the CAR MUX selector input to channel the value from IR as the value to load into CAR when CLK rises. CLK is then set to 1 causing CAR to load IR. The input registers (ALPHA, GAMMA and BETA) can be loaded anytime before the first falling edge of CLK. CLK is then set to 0 enabling the registers (A, E, Q and SC) to change. RESET is now set to 0 since it is no longer needed as CAR has been successfully loaded at the rising edge of CLK, and A and Q have loaded (if LA, LQ are set, respectively) on the rising edge of CLK. Subsequent pulses of CLK will execute the instruction and when FINISH goes high (indicating a completed instruction) it will act exactly as if RESET was set high, as above, and cause a loading of CAR from IR. Therefore as long as the trainer is not turned off, RESET never has to be set to 1 again.

## 8 THE MULTIPLY ALGORITHM

The first complex instruction is 'Multiply'. Figure 7 details the hardware algorithm that will provide unsigned multiplication:

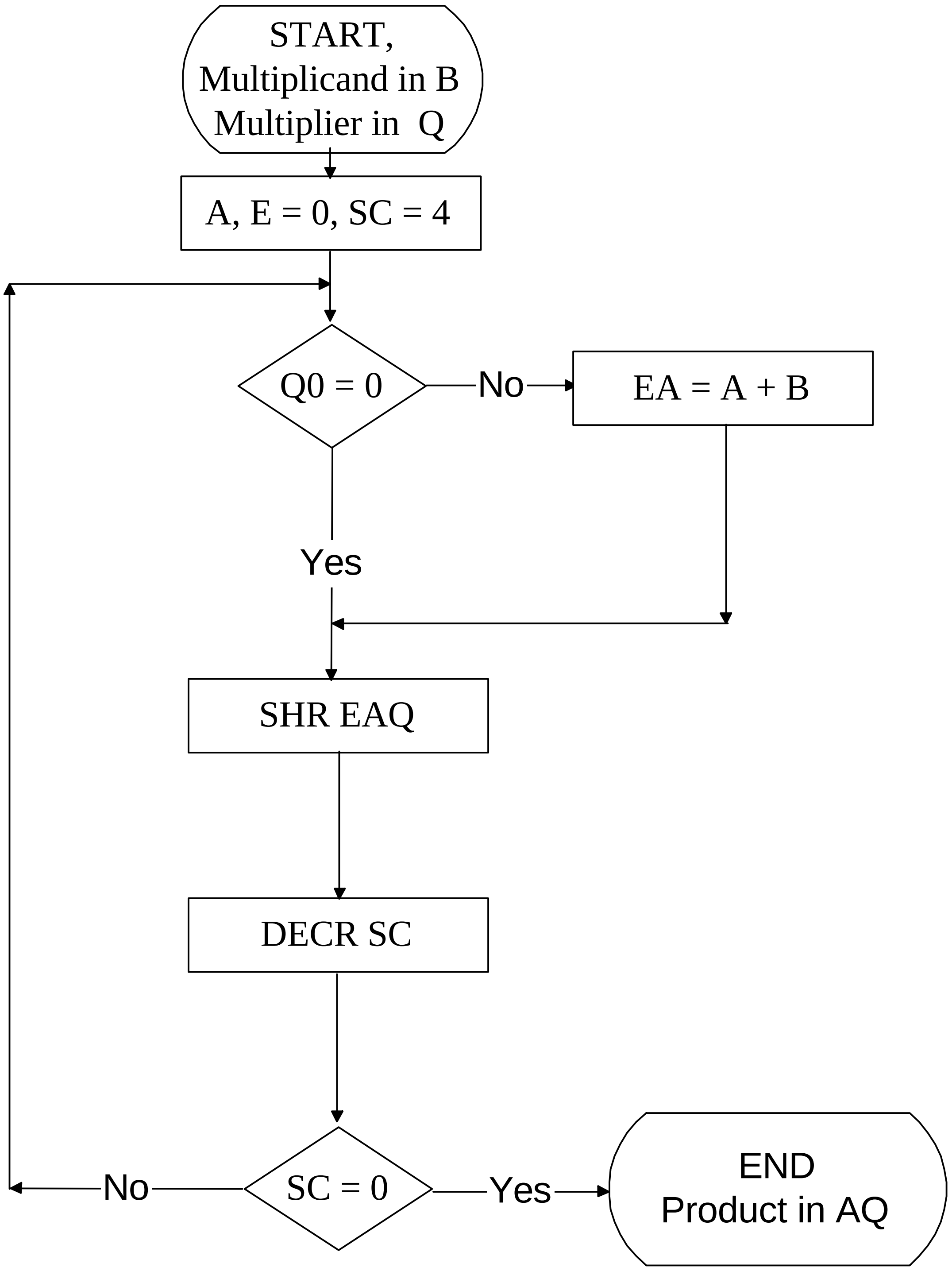

**Figure 7: The Multiply Hardware Algorithm**

The algorithm starts by clearing A and E. SC is loaded with 4 since the loop will be executed 4 times, once for every bit in the multiplier. The multiplicand is in B and the multiplier is in Q.

The low-order bit of the multiplier in Q is tested. If it is a ‘1’, the multiplicand in B is added to the present partial product stored in A, else nothing is done. The combined register EAQ is then right-shifted by one bit. SC is then decremented since one ‘iteration’ of the instruction is complete. When SC = 0 the multiplication is complete.

The partial product that is stored in A is shifted into Q a single bit at a time and eventually replaces the multiplier. The product of ‘B x Q’ is stored in AQ, where the most significant bits are in A.

The LSB of Q is tested, if it is ‘0’ then the ALU instruction ‘EA = A + B’ is executed otherwise it is skipped. EAQ is then right-shifted by one bit. Since no bit testing is provided for in the trainer, special logic will have to impede ‘EA = A + B’ from occurring if the LSB of Q is 0 (see below).

The micro-instruction implementation of the algorithm is shown in Table 4 below. Each micro-instruction will now be described:

| **Micro-instruction** | **OF** | **CB** | **LQ** | **LA** | **A6** | **A5** | **A4** | **A3** | **A2** | **A1** | **A0** | **S1** | **S0** | **$Q_0$** | **DSC** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CLR A, Q ⇐ γ | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| EA⇐A + B, | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| if Qo=1 | | | | | | | | | | | | | | | |
| SHR EAQ, DSC | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

**Table 4** Multiply Algorithm

CLR A, Q ⇐ γ

This is the first micro-instruction to be executed for the multiplication algorithm. As with all instructions, the first micro-instruction loads the registers, yet only for the first iteration of the instruction. Therefore when R/F = 1 the Q register loads from GAMMA (γ).

A is cleared. This is a special case since A is not actually loaded from ALPHA, rather a logic operation using the ALU is performed and the A register is cleared. Special logic must be implemented to only allow this operation to be performed for the first iteration as well. Taking a look at Table 1 the only difference between A ⇐ 0 and A ⇐ A is that A4 is complimented. Therefore when CAR = ‘first address of multiply’ (since this only applies to the first micro-instruction of the multiply algorithm) and R/F = 1 (which occurs for the first micro-instruction of an iteration), A4 is complimented to turn the ALU command A ⇐ 0 into A ⇐ A command. This is implemented in Figure 3 in the ‘Multiply Logic’.

Now both the clearing of A and the loading of Q only occur for the first micro-instruction of the first iteration.

$EA \Leftarrow A + B$,
if Qo=1

IF Q0 = 1 (the LSB of Q) then the arithmetic operation $EA \Leftarrow A + B$ should be performed, otherwise no arithmetic operation should occur. Since no conditional arithmetic operations are available then additional ‘multiply-only’ logic is required in order implement this conditional operation. Table 1 also shows the only difference between $EA \Leftarrow A + B$ and $A \Leftarrow A$ is that A2 is complimented. This should only occur for the second micro-instruction in the multiply algorithm. Therefore when CAR = ‘second address of multiply’, A2 is complimented to turn the ALU command $EA \Leftarrow EA$ into $A \Leftarrow A$ command. Again in Figure 3 the ‘Multiply Logic’ is used to implement this. The value of E for $A \Leftarrow A$ is not important because the next micro-instruction forces E to load a 0.

SHR EAQ, DSC

The last micro-instruction causes a right shift of EAQ, with E loading 0 in the process. SC is also decremented, therefore during the final iteration of the multiply algorithm (when SC is decremented to zero) FINISHED will go high.

## 9 THE DIVIDE ALGORITHM

The second complex instruction is 'Divide'. Figure 8 details the hardware algorithm that will provide unsigned division:

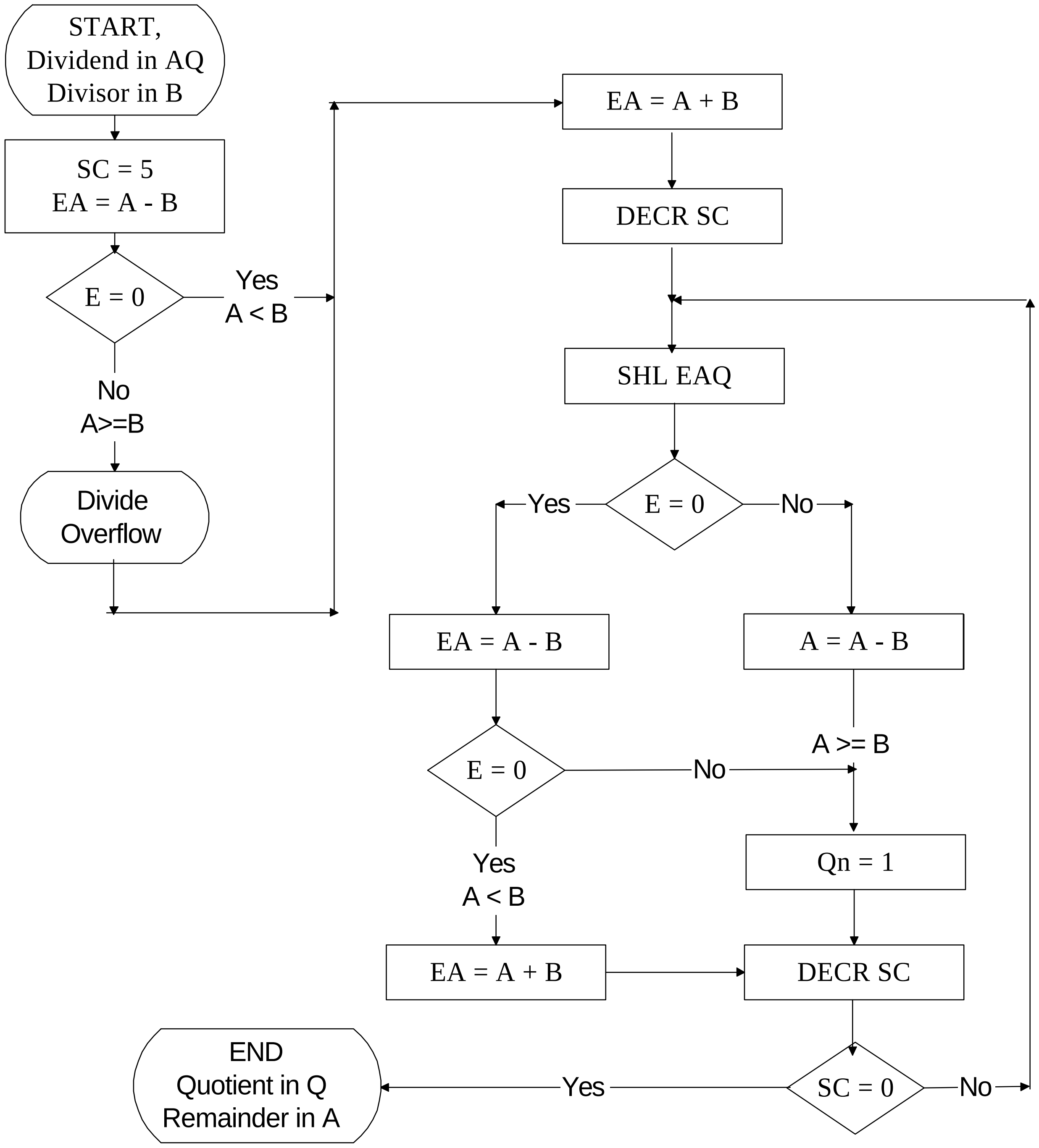

**Figure 8: The Restoring-Divide Hardware Algorithm**

SC is loaded with 5 since the loop will be executed 4 times, once for every bit in the multiplier plus an additional iteration (explained below). The multiplicand is in B and the multiplier is in Q.

B is subtracted from A to determine if an overflow condition has occurred, which is based on the value of E (see below). A is restored by adding B and SC is decremented (the reason for 5 instead of 4 iterations).

The dividend is left-shifted into AQ with the MSB being loaded into E. A test is performed on E, if it is 1 then EA > B since EA consisted of a 1 followed by 4 bits, while B has only 4 bits. Because E was 1, B is subtracted from EA and the LSB of Q is set. Now A holds the value:

$$EA - B$$

Otherwise if the left-shifted value into E was a zero, then the divisor B is subtracted. The carry is held in E, and if it is 1 then the LSB of Q is set since A ≥ B. If E is 0 then A < B and the original value is restored by adding B and the LSB of Q must be 0 (which was loaded during the left-shift so nothing else must be done).

The micro-instruction implementation of the algorithm is shown in Table 5 below. Each micro-instruction will now be described:

| Micro-instruction | OF | CB | LQ | LA | A6 | A5 | A4 | A3 | A2 | A1 | A0 | S1 | S0 | $Q_0$ | DSC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $A \Leftarrow \alpha$, $Q \Leftarrow \gamma$ | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SHL, skip 3 if (E = 1) | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| EA ⇐ A-B, skip3 if (E = 0) | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| EA⇐A+B, DSC | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| EA⇐A-B | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| Q0⇐1,DSC,A=A | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| *EA⇐A-B, OF | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| EA⇐A+B, DSC | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |

**Table 5** Divide Algorithm

$A \Leftarrow \alpha$, $Q \Leftarrow \gamma$

This is the first micro-instruction to be called for the divide algorithm. As with all instructions, the first micro-instruction loads the registers A and Q from ALPHA and GAMMA, respectively (if R/F = 1, i.e. if this is the first iteration of the divide instruction).

A special situation arises if R/F = 1 and IR = Divide, which occurs when the Divide instruction is first called. When this happens a special skip of six micro-instructions must occur, rather than executing the next micro-instruction ‘SHL, skip 3(E=1)’. The next instruction then becomes ‘EA⇐A-B, OF’, the second last micro-instruction of the divide instruction (indicted by ‘*’). This micro-instruction is ONLY executed when divide is first started (that is why R/F = 1 is a condition). It’s purpose is to test for the over-flow condition (see the description of the micro-instruction below). Therefore the ‘NEXT ADDRESS LOGIC’ must apply a value of 6 to the CAR FULL ADDERS if R/F = 1, otherwise it is not the first iteration of the divide instruction and a 1 will be used.

SHL,
skip 3 if (E = 1)

The combined register EAQ is left shifted, with the MSB of A being loaded into E, the LSB of A being loaded from the MSB of Q and the LSB of Q loads a 0. A skip of three micro-instructions occurs if E is set (1) after the shift. The CB bit of the micro-instruction being set indicates a skip, yet the values of S1, S0 and E determine the condition which causes the skip (see the section ‘Branching’ for more details). In this case when a SHL occurs and CB is set, the condition for a skip = 3 is that E is set, if E is cleared then the next micro-instruction is executed.

If the skip = 3 occurs then the micro-instruction ‘EA⇐A-B’ is the next to be executed. This bypasses ‘EA ⇐ A-B’ and ‘EA ⇐ A+B’.

EA ⇐ A-B,
skip3 if (E = 0)

This micro-instruction causes A to load the result from the arithmetic operation ‘EA ⇐ A-B’. Again CB is set so a branching may occur. In this case S1,S0 = 1,1 so the effect is that a skip = 3 occurs for a cleared E, otherwise skip = 1 (execute the next micro-instruction).

If the skip = 3 occurs the micro-instruction ‘Q0⇐1,DSC,A=A’ is the next to be executed. This bypasses ‘EA ⇐ A+B’ and ‘EA ⇐ A-B’.

EA⇐A+B, DSC

The A register loads from the output of the arithmetic micro-instruction ‘EA⇐A+B’ and SC is decremented, indicating the end of an iteration.

EA⇐A-B

The A register loads from the output of the arithmetic micro-instruction ‘EA⇐A-B’.

Q0⇐1,DSC,A=A

The LSB of Q is set, A loads itself and SC is decremented, indicating the end of an iteration. A loads itself is redundant yet is required since in order to set the LSB of Q it loads copy of itself with the LSB forced to a ‘1’. Since both Q and A have common latch control signals S1,S0 then a parallel loading of Q must occur at the same time as a parallel loading of A. Therefore S1,S0 = 1,1 forces both A and Q to load with A loading from the output ‘F of the ALU, and Q loads the altered version of Q (with Q0 = 1). Therefore the ALU operation must be for A = A otherwise A will become altered.

*EA⇐A-B, OF

As stated previously, this micro-instruction is ONLY executed after the first micro-instruction of divide is executed. It is never executed again during any of the other four remaining iterations.

The arithmetic operation EA⇐A-B occurs, OF being set causes the compliment of E to be latched into a temporary register which will later be latched into the overflow flag of the STATUS register at the end of the divide instruction. If E is cleared due to the arithmetic operation then it indicates that A ≥ B, which is the case when an overflow will occur. Even if a overflow condition occurs, the remainder of the ‘Divide’ instruction is executed, the values that should have been the quotient in Q and remainder in A will be useless then.

EA⇐A+B, DSC

The A register loads from the output of the arithmetic micro-instruction ‘EA⇐A+B’ and SC is decremented, indicating the end of an iteration. This restores A to the value before this ‘special’ once-executed few micro-instructions.

This is why SC is loaded with a ‘5’ since these last two micro-instruction are executed only at the beginning of the instruction ‘Divide’, and DSC is set on the last of the micro-instructions. Without overflow checking SC would have been ‘4’ as is the case for multiply, since one iteration must occur for each bit in the divisor. Since the only way to execute another iteration of a instruction is to have DSC set.

## 10 OTHER INSTRUCTIONS

Although the complexity of a micro-programmed arithmetic processor is required for multi-iteration instructions such as divide and multiply, other more simpler instructions can still be executed.

The following is a list of nine other instructions that take advantage of the capabilities of the ALU, four arithmetic and five logical. Other instructions can be derived such as basic shifting and be programmed into the control memory as required.

Since these are ‘simple’ instructions which do not require multiple iterations, they require SC to load ‘1’. Therefore the second micro-instruction of these double micro-instruction instructions have DSC = 1, ending the instruction after one iteration.

All the arithmetic operations are signed, with negative ‘twos compliment’ representation.

### ADD

The basic signed addition for four bit numbers. The first micro-instruction loads A. The next executes the arithmetic operation, and ends the instruction.

| Micro-instruction | OF | CB | LQ | LA | A6 | A5 | A4 | A3 | A2 | A1 | A0 | S1 | S0 | $Q_0$ | DSC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $A \Leftarrow \alpha$ | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $EA \Leftarrow A+B$, DSC | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |

### SUB

The basic signed subtraction for four bit numbers. The first micro-instruction loads A. The next executes the arithmetic operation, and ends the instruction.

Note the only difference between ADD and SUB is that SUB has the ALU control signal A0 = 1, which causes the ALU to act as a full subtractor, where A0 = 0 emulates a full adder.

| Micro-instruction | OF | CB | LQ | LA | A6 | A5 | A4 | A3 | A2 | A1 | A0 | S1 | S0 | $Q_0$ | DSC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $A \Leftarrow \alpha$ | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $EA \Leftarrow A-B$, DSC | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |

### INC

The value of ALPHA is incremented (A = ALPHA + 1). The first micro-instruction loads A. The next executes the arithmetic operation, and ends the instruction.

Note the only difference between ADD and INC is that INC has the ALU control signals A1 = 1, which causes the ALU to force a 1 into the first carry in, and A2 = 0 which blocks the input from the B register.

| Micro-instruction | OF | CB | LQ | LA | A6 | A5 | A4 | A3 | A2 | A1 | A0 | S1 | S0 | $Q_0$ | DSC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A ⇐ α | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| EA⇐A+1, DSC | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |

## DEC

The value of ALPHA is decremented (A = ALPHA - 1). The first micro-instruction loads A. The next executes the arithmetic operation, and ends the instruction.

Note the only difference between INC and DEC is that DEC has the ALU control signal A0 = 1, which causes the ALU to act as a full subtractor, where A0 = 0 emulates a full adder.

| Micro-instruction | OF | CB | LQ | LA | A6 | A5 | A4 | A3 | A2 | A1 | A0 | S1 | S0 | $Q_0$ | DSC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A ⇐ α | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A⇐A-1, DSC | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |

## OR

The value of ALPHA is ORed with BETA (A = ALPHA \|\| BETA). The first micro-instruction loads A. The next executes the logic operation with E loading a 0, and ends the instruction.

| Micro-instruction | OF | CB | LQ | LA | A6 | A5 | A4 | A3 | A2 | A1 | A0 | S1 | S0 | $Q_0$ | DSC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A ⇐ α | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A⇐A \|\| B, DSC | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |

## AND

The value of ALPHA is ANDed with BETA (A = ALPHA & BETA). The first micro-instruction loads A. The next executes the logic operation with E loading a 0, and ends the instruction.

| Micro-instruction | OF | CB | LQ | LA | A6 | A5 | A4 | A3 | A2 | A1 | A0 | S1 | S0 | $Q_0$ | DSC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A ⇐ α | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A⇐ A&B, DSC | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |

## XOR

The value of ALPHA is XORed with BETA (A = ALPHA ⊕ BETA). The first micro-instruction loads A. The next executes the logic operation with E loading a 0, and ends the instruction.

| Micro-instruction | OF | CB | LQ | LA | A6 | A5 | A4 | A3 | A2 | A1 | A0 | S1 | S0 | $Q_0$ | DSC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A ⇐ α | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A⇐ A⊕B, DSC | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |

## NOT

The value of ALPHA is complimented (A = -ALPHA). The first micro-instruction loads A. The next executes the logic operation with E loading a 0, and ends the instruction.

| Micro-instruction | OF | CB | LQ | LA | A6 | A5 | A4 | A3 | A2 | A1 | A0 | S1 | S0 | $Q_0$ | DSC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A ⇐ α | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A⇐ -A, DSC | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |

## CLR

A is loaded with the value 0. Although no operation is performed on any register, since no register is loaded, the instruction requires two micro-instructions. This is due to the fact that SC is loaded (with ‘1’ for simple instructions), yet at the falling edge of CLK, which is the same position in the CLK signal where SC may be decremented. Since both a loading and decrementing of SC at the same time is not possible, therefore they must execute in two separate micro-instructions.

The first micro-instruction does nothing (also known as a ‘NOP’), where the next executes the logic operation with E loading a 0, and ends the instruction.

| Micro-instruction | OF | CB | LQ | LA | A6 | A5 | A4 | A3 | A2 | A1 | A0 | S1 | S0 | $Q_0$ | DSC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NOP | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A⇐ 0, DSC | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |

## 11 USER INPUT/OUTPUT

In order for the user to be able to interface with the trainer, logic switches for input and LEDs (Light Emitting Diodes) for output will be used.

### 11.1 INPUT

Since the CLK input provides synchronization for the trainer it must be made from 'debounced' logic. The debouncing removes any ringing in the input switch. A simple SR latch will provide the debouncing, as seen in the following figure:

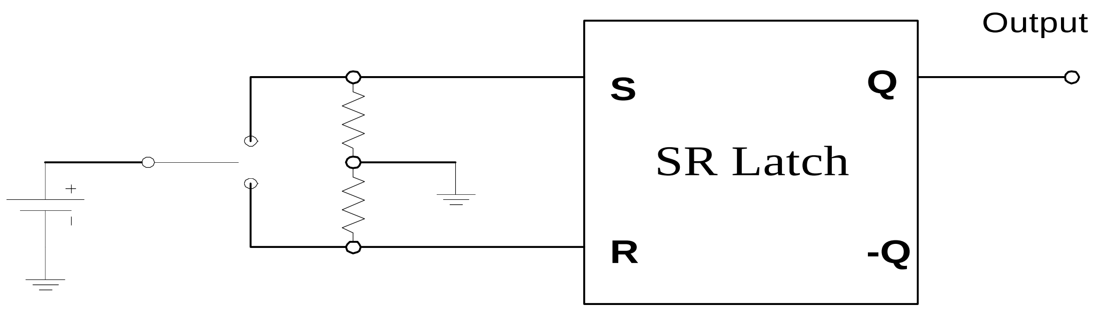

**Figure 9: The Debounce Logic Switch**

The input registers ALPHA, BETA, GAMMA and IR do not require debouncing since they do not affect any synchronous logic in the trainer. A general 'logic pull-up' input therefore can be used for these registers, which can be seen in Figure 10 below.

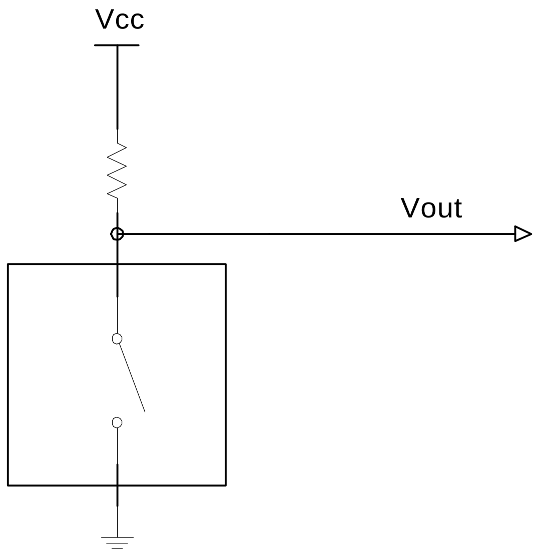

**Figure 10: Regular Logic Input Switch**

## 11.2 OUTPUT

All the registers and control signals will be displayed by LEDs. The following figure shows the circuit to drive the LEDs.

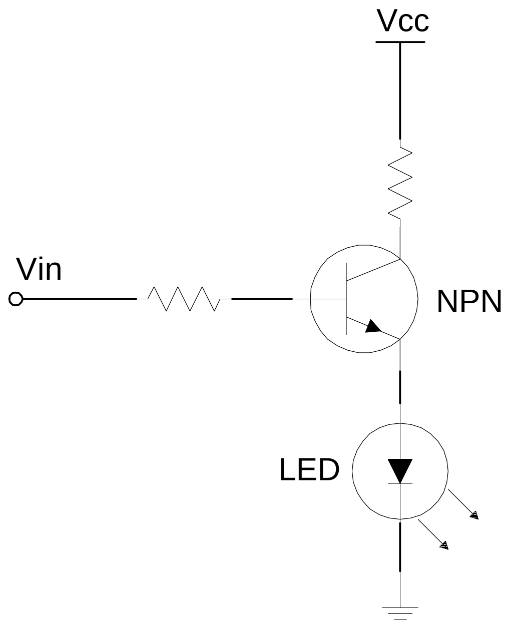

**Figure 11: LED Driver Circuit**

## 12 CLK TIMING

To provide synchronization for the trainer the signal ‘CLK’ drives all the sequential logic such as the registers and the D flip-flops. The following diagram details the general use of CLK and the other synchronization related signals:

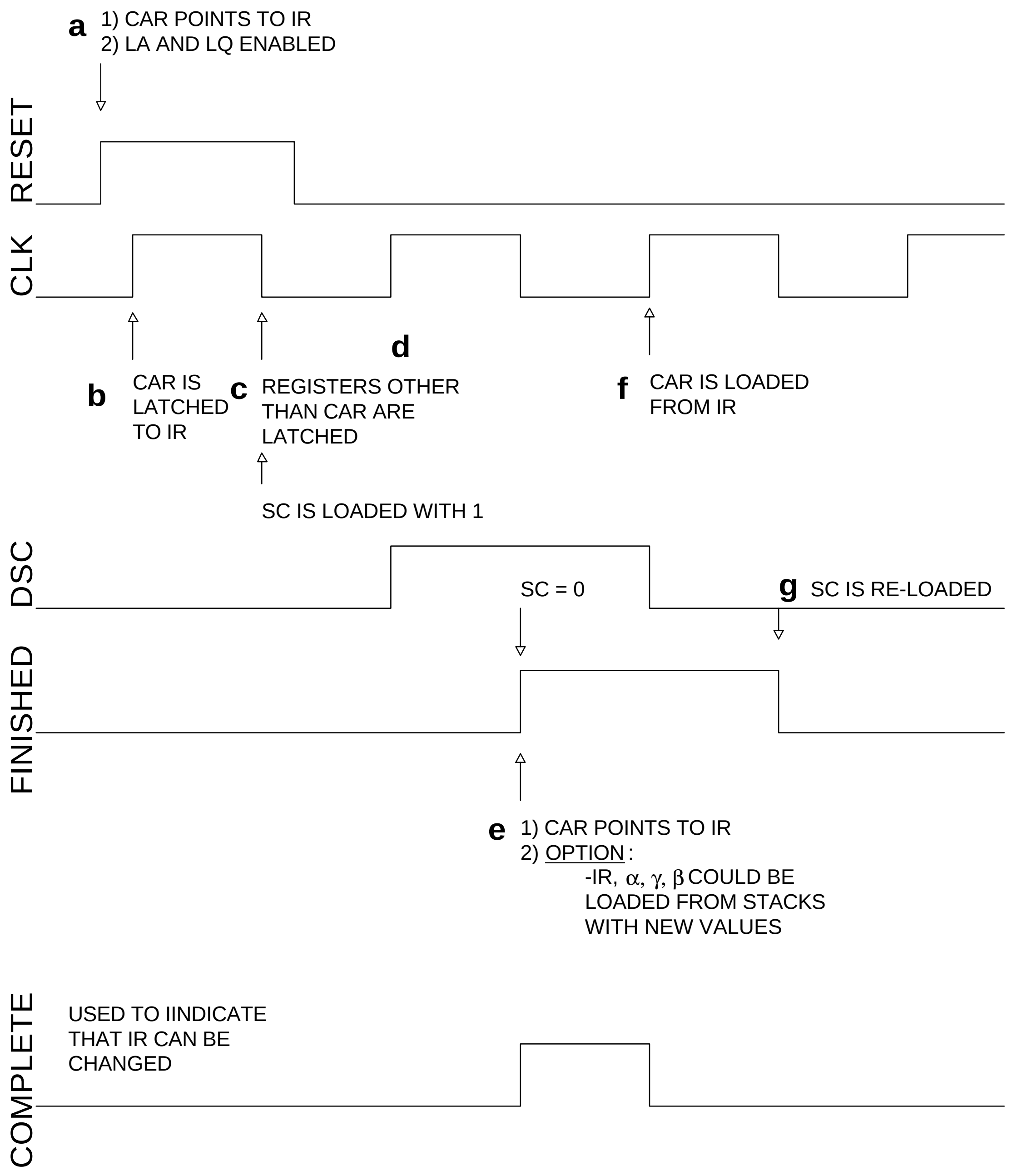

**Figure 12: Synchronization Timing Diagram**

When the trainer is first turned on (a) RESET is high and CLK is low (see section 'Booting the Trainer' for more details). This forces the CAR MUX to 'point' to IR and enables LA and LQ to load A and Q, respectively.

Upon the rising edge of CLK (b) CAR is latched to IR, and the micro-instruction control signals change, and propagate causing the output of the ALU to change according to the new micro-instruction.

The falling edge of CLK (c) causes the E, A, Q and SC registers to latch. SC is loaded with the value based on the instruction 'pointed' by IR ('4' for multiply, '5' for divide, etc.) when R/F = 1. In this example SC is loaded with '1'.

The next rising edge of CLK (d) then loads the next micro-instruction address into CAR, based on the 'NEXT ADDRESS LOGIC'. In this example the new micro-instruction has DSC = 1. Therefore the next falling edge of CLK causes SC to be decremented, and a short delay afterwards COMPLETE and FINISH goes high (e) as SC = 0, indicating the instruction has completed execution.

If a stack was used to load A, Q, B and IR then the COMPLETE or FINISHED signals going high could be used to indicate that a new set of these registers should be loaded in order to allow a new instruction (and set of data to act on) to be executed. Otherwise the user can now reload these registers for a new instruction to be executed.

With R/F = 1 (because FINISHED = 1) the CAR MUX again points to IR (as was the case when RESET was high). Therefore the next rising edge of CLK (f) causes CAR to load IR, and a new micro-instruction is loaded. The net falling edge of CLK (g) causes a new value for SC to be loaded (again based on the value in IR), and FINISHED falls.

This sequence is the same for all instructions, with the only difference in the value of SC loaded, and thus the number of iterations that are executed (and the number of CLK cycles to complete execution of the instruction).

After CAR is loaded with a new address (upon a falling edge of CLK), the arithmetic/logic operation takes. On the falling edge of CLK parallel loading or shifting occurs for all the registers other than CAR. Since this occurs after the arithmetic/logic operations take place, both types of operations can occur in a single micro-instruction. Therefore a complete four-operation micro-instruction can be executed in one CLK cycle:

1. Arithmetic/logic instructions (ALU command).
2. Shifting/parallel-load operations (E, A and Q shifting, OR A and Q loading.
3. A 'skip' of a number of micro-instructions (a branch) based on E (and the value of S1,S0, see the section 'Branching' for more details).
4. The SC register can be decremented (DSC = 1), and the overflow flag of the status register can be latched (OF = 1).

The following example instruction demonstrates the complete four-operation capability:

| Micro-instruction | OF | CB | LQ | LA | A6 | A5 | A4 | A3 | A2 | A1 | A0 | S1 | S0 | $Q_0$ | DSC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| EA⇐A+B, OF, Q0⇐1, DSC, skip3 (E = 0) | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |

Decoded:

1. EA⇐A+B (S1,S0 = 1,1 for A to load the result). (Type 1,2)
2. OF flag of the STATUS register is latched with the compliment of E (after the arithmetic operation). (Type 4)
3. Q0⇐1, causes the LSB of Q to be set (again, S1,S0 = 1,1 for Q to load). (Type 2)
4. DSC, causes SC to be decremented (indicating ‘end of an iteration’). (Type 4)
5. Skip 3 if E = 0 (after the arithmetic operation). (Type 3)

Result:

Therefore both A and Q are loaded with new values since S1,S0 = 1,1. A loads the result of the ALU operation and Q loads a copy of itself with the LSB set (since Q0 = 1). The over-flow flag is set E = 0. The amount to ‘skip’ for the determination of the new CAR is based on the value of E (both of these occur at the next rising edge of CLK). The SC register is decremented (at the falling edge of CLK).

Therefore the multi-capable micro-instruction provides for a powerful, yet simple to code program.

## 13 BRANCHING

Although branching is not exactly what takes place in the micro-program for the trainer there is the capability for altering the flow of execution by a single flag ‘CB’.

A combination of CB, E and S1,S0 determine the number of micro-instructions to ‘skip’ or bypass. This logic feeds a value, either 1, 3 or 6 to the CAR FULL ADDER where the output of drives one of the inputs to the CAR MUX. The other input to the FULL ADDER is the value of CAR.

How the branching works is as follows. The Full Adder takes as input ‘A’ the value of CAR and input ‘B’ as the value from the ‘Next Address Logic’ (which is described here). The output of the Full Adder is just the addition of these two inputs. Therefor the value of CAR plus an offset is provided at the output of the Full Adder, allowing a ‘jump’ or skip by the amount of this offset in the control memory.

There is one special case which allows for a ‘6’ to be the output of the logic. This occurs for the first micro-instruction executed in the divide instruction (not discussed here any further, see the section ‘The Divide Algorithm’ for more details).

If the CAR MUX ‘points’ to the output of the CAR FULL ADDER (which it does as long as RESET = 0 AND DSC = 0) then CAR will be loaded from the output of the FULL ADDER on the next rising edge of CLK.

The following table is used for determining this input that is added to CAR. The SHL column is a 1 when S1,S0 = 1,0, in other words for a left-shift micro-instruction it is a 1, else it is a 0, and X represents ‘don’t care’.

| **CB** | **SHL** | **E** | **FULL ADDER INPUT** |
|---|---|---|---|
| 0 | X | X | 1 |
| 1 | 0 | 0 | 3 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 3 |

Since branching is only used in the divide instruction, the above table was designed specifically to be efficient for use in that instruction. An example of it’s use shall be presented now:

The second micro-instruction of the divide instruction (see the section ‘The Divide Algorithm’) first causes a left shift through EAQ. Since CB = 1 and a left shift (SHL = 1) then The ‘FULL ADDER INPUT’ is a 3 if E = 1 else it is a 1. Therefore the micro-instruction causes a ‘jump’ by 3 micro-instructions if E is set otherwise the next micro-instruction is executed (a jump of 1).

The ‘3’ was chosen since a jump/skip of three micro-instructions was required for the divide instruction. This instruction has two different jump  micro-instructions, one active on E = 1 the other active when E = 0. Some sort of differentiation had to be determined since only a single bit control signal CB is used in the control memory, which is not enough to indicate two different kinds of branching. Another micro-instruction control signal could have been used, yet it was obvious that the difference was that a left-shift occurred for the E = 1 and a parallel-load occurred for the E = 0. Therefore the S1,S0 control signals were just made part of the logic that determines the amount to skip.

## REFERENCES

1 Moffitt, Rod S., “Four-Bit ALU Trainer Instructional Specification”, 1995

2 Mano, M. Morris, “Computer System Architecture”, Prentice-Hall, N.J., 1982