University of Tasmania

Design of a 32-bit Arithmetic Unit based on Composite Arithmetic and its Implementation on a Field Programmable Gate Array.

by

Tomasz Hubert Pinkiewicz, BAppComp

A dissertation submitted to the
School of Computing
in partial fulfilment of the requirements for the degree of

Bachelor of Computing with Honours

University of Tasmania
November, 1999

Declaration

I hereby state that this thesis contains no material which has been accepted for the award of any other degree or diploma in any tertiary institution, and that, to the candidate’s knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the text of the thesis.

Tomasz Pinkiewicz

Signature ______________________ Date ___________

Abstract

As we advance into the new century, computers of the future will require new techniques for arithmetic operations, which take advantage of the modern technology and yield accurate results. Floating-point arithmetic has been in use for nearly forty years, but is plagued with inaccuracies and limitations which necessitate introduction of a new concept in computer arithmetic, called Composite Arithmetic. Composite Arithmetic combines fixed-point and floating-point arithmetic into one integrated concept where numbers are automatically assigned the right form. This negates the need for differentiating between integer and real numbers in programming languages and allows for better accuracy in calculations. The concept has two main forms: exact and inexact. The exact form deals with integers and rational numbers, while inexact form deals with numbers that cannot be represented exactly. To develop and implement such concept in hardware, tools are needed that will allow for easy design and re-design process, at a low cost. A device that meets these requirements is a Field Programmable Gate Array. This electronic device provides quick and easy way of designing the system and then implementing it by downloading data to the device. It can then be tested and reprogrammed as desired, without the need for a new device. This thesis is an attempt to design and implement Simple Composite Arithmetic Machine (SCAM), which will be capable of performing operations on exact numbers (rational and integer numbers). The core of the research is Composite Arithmetic Unit, which contains operations like Multiply, Divide, Add and Subtract. It also can find Greatest Common Divisor and cast out common factors of two numbers. The CAU is controlled using Control Unit and Feedback Unit, and results are stored in the Register Unit. The SCAM is therefore a basic microarchitecture that will form a basis for further research in this field.

TABLE OF CONTENTS

LIST OF FIGURES

Acknowledgments

I would like to thank Dr. Tariq Jamil, my supervisor, who introduced me to the FPGA concept, gave me great feedback on the work, and helped me with some critical problems that were slowing me down. He also encouraged me to share my work with others by writing paper for a conference. Also special thanks go to Neville Holmes for providing a subject for my thesis. His advice on determining the scope of the project and explanations of the concept were invaluable. Also recognition goes to the Academic Staff of the School of Computing for providing me with knowledge and problem solving skills required during the project design.

On the administrative side, thanks go to Xilinx Corporation for providing me with a FPGA device as part of the company's University Program. Also appreciation goes to secretaries of the School, Raelene Couch and Yvette Kitchener and to the Technical Staff of the School, Tony Gray and Christian McGee.

Last but not least, thanks go to my family and friends for patience and understanding, and to my special friend, Donna, who was my motivation and supported me during the Honours year.

1.1 Computer Arithmetic

1.1.1 Computer Arithmetic History

Fixed-point arithmetic was first used after World War Two to perform calculations on integer values and to represent them exactly. A programmer had several lengths of representation to cope with expected range of numbers. In integer arithmetic, calculation of fractions was done using scaling and this could produce inexact results (Neville 1997a and b). The scaling required for problems to be pre-processed by the user and then they could be accommodated by fixed point-representation. However with increasing speed of computers, more complex operations had to be performed and the pre-processing became a lengthy task (Kulisch, 1999). Also large numbers introduced a new concept into computer arithmetic, called overflow. To overcome the overflow problem, floating-point arithmetic was introduced.

Floating-point arithmetic can cope with large range of numbers, but it does so only by approximating. The numbers are represented in semi-logarithmic form and they have two components, significand and exponent. The significand expresses the precision of the number while the exponent expresses the range of the values that can be represented. This floating-point representation became a standard and is used in all older and modern computers. However, this notation has many problems, which have been accepted (or ignored) over the years.

Results of floating-point arithmetic can vary. They can be satisfactory, inaccurate or completely wrong. However there is no way of telling which one of them has occurred. Although much larger range of values can be stored, comparing to fixed-point form, there is still problem of overflow. Additional to the overflow, floating-point introduces underflow, where the number is too small to be represented accurately. Different lengths of representation mean that a compromise between precision and the storage space has to be made. This can then affect accuracy of the arithmetic. Accuracy is also affected by truncating and rounding errors, and by conversion of values to and from the display form. Another problem with floating-point arithmetic is that it doesn’t consider special values. These are zero, infinity, and indeterminacy. These are still results and can be very important in scientific calculations. (Kulisch 1999) describes five equations, all of them containing the same numbers with the same signs but arranged differently. Conventional computer with floating-point standard returns different values for each equation, while they all should be the same. Several other examples are provided which prove that even simple mathematical equations can be miscalculated when using floating-point form.

Another type of problem is concerned with programming issues. The programmer has to determine beforehand if fixed-point or floating-point notation should be used and what precision should be used (single or double). Then there is a problem of handling exceptions, and problem of converting between different forms and how this affects accuracy of result. Another concern is the accuracy of the display and how exceptions and special values will be displayed. All this has to be considered by the programmer and wrong decision can cause erroneous results.

Floating-point representation is an improvement over fixed-point form. However, today's computers have become faster and have to perform more accurate and larger calculations. Current technology and requirements make the floating-point form "obsolete" and a new type of computer arithmetic is needed. One of the examples of advanced computer arithmetic is scalar product concept presented by (Kulisch 1999).

1.1.2 Advanced Computer Arithmetic

Scalar product is the fastest way to use the computer (Kulisch 1999). There are no intermediate results that need storing and no intermediate rounding. There are also no overflows or underflows and results are always correct. If desired, final rounding can occur at the end of the calculation. The concept of scalar product is based on two principal solutions. One solution involves long adder and long shift. If a register is built as an accumulator with an adder, then all summands can be added without loss of information. The register would have additional number bit to accommodate possible overflows. This allows adding a sum or scalar product without loss of information. The other solution involves short adder instead of the long adder with additional local memory as part of the arithmetic unit.

Advanced arithmetic expands the arithmetic and mathematical capability of the digital computer. It allows for twelve fundamental data types or mathematical spaces and highest degree of accuracy. The four basic ones are real, complex, interval and complex interval data types. It also allows for matrix and vector operations. Number of complex computations is increasing and more accuracy is needed. Advanced Computer Arithmetic offers improvement of floating-point arithmetic and gives more accurate results. Current research is aimed at incorporating these advanced concepts into the new microprocessors to avoid the bottleneck of Input/Output interface.

1.1.3 Composite Arithmetic

Composite Arithmetic (Neville, 1997a) is based on advanced computer arithmetic concept and combines several different formats. These are stored as a single binary form and formats are distinguished using tags. The storage form is specified by multiples of 16 and four lengths are recommended: short (32 bits), normal (64 bits), long (128 bits) and extended (256 bits). For most commercial calculations on exact numbers and scientific computations on inexact numbers, 32- and 64-bit sizes would be sufficient. The long and extended formats (128- and 256-bit) can be used for financial and number theory calculations, where very long exact results can occur. Also some intermittent technical computations where very precise results are needed would use one of these two formats. Composite arithmetic merges both exact and inexact formats using a tag bit. The bit is set to 0, if the number is exact and it is set to 1, if it is inexact.

For exact values there are two forms. The primary one is an integer form and secondary is rational form. To distinguish between them another bit is added to the tag field. In the integer form all bits, except the tag, can be used to store the value. Negative numbers are stored using 2s complement of the magnitude. This ensures that zero is stored exactly and prevents negative Zero. Rational numbers are typically result of integer division. They can be stored exactly as integers if proper representation is adopted. The secondary exact form allows storing very large numbers, or very small numbers, depending on the size of numerator and denominator. To allow this, the form must provide for sharing of the value bits between numerator and denominator. The solution to that is called a floating slash and it is several bits, depending on word length that separate the numerator from denominator. For example, for a 32-bit number, five bits are needed to be stored to accommodate the floating slash. If the numerator is 1 then all bits are available for the denominator. If the denominator is 1 then the number will be stored in an integer format; therefore the smallest denominator is 2. There are other issues, which have to be addressed when using this form. First, the denominator can be 0, which means that the number is infinity. Infinity needs storing because it can result from division by zero. Also indeterminate result needs storing.

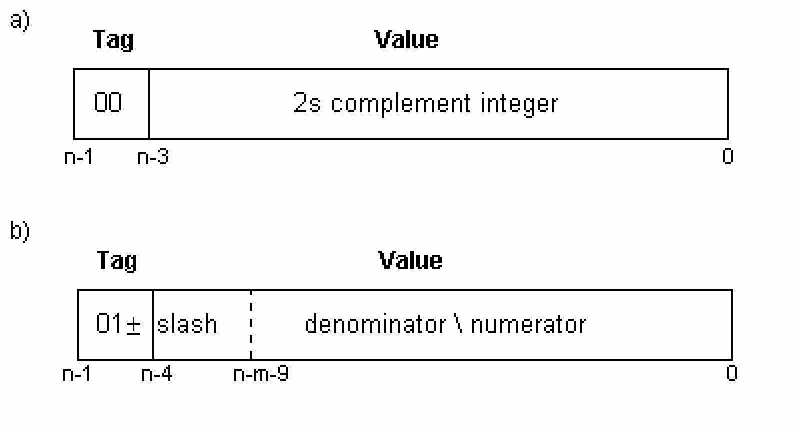


Figure 1. Proposed exact storage forms include (a) primary exact form (integer) and (b) secondary exact form (rational). In the bit numbering as shown, n stands for the number of bits in the form and can be 32, 64, 128, or 256, while m numbers the different form.

If the value cannot be stored exactly then the composite arithmetic will store it in an inexact form. Numbers can be represented in different ways. The first one is called double-number form. An example of it is the floating-point representation. The second one is called single-number form and an example of it is relatively new, signed logarithmic form. This form is subdivided into primary inexact and secondary inexact forms. Primary form uses signed pure logarithmic representation and secondary form uses antitetrational representation. The second form has been adopted by the Composite Arithmetic.

While the above forms are satisfiable for storage purposes, they are not suitable for display. This is because to display, numbers are represented as characters; their values have to be converted to and from an appropriate storage form. Deficiency of the ASCII character set puts several restrictions on available choices. A form for exact values uses decimal point and a fraction point. This would allow a number 456 ¾ to be represented as 456.3.4, where first dot is a decimal point and second dot is a fraction point. A number ½ would be stored as 0.1.2. This approach would also allow displaying infinity as 0.1.0 and indeterminacy 0.0.0. Representation of inexact numbers would be similar to the *e*-notation, where the exponent uses scaling base of 10. The display form would use the scaling base of 1000 and this would be called *k*-notation. This would be supported by *m*-notation (milli notation) to avoid squeezing a negative sign into the exponent. A second *k* or *m* could display secondary inexactness.

The proposal also mentions register form. This would be a fixed-point long accumulator, which would be 512 bytes (4096 bits) long. It would be large enough to store extended primary inexact storage form and some additional data like tags and signs.

Composite arithmetic is more complex than floating-point but many aspects of it have already been implemented in hardware and software. Replacing floating-point arithmetic would benefit electronic calculator arithmetic and would improve capabilities of software packages such as spreadsheets. Another advantage lies in programming using the composite arithmetic. The programmer wouldn't have to make choices between fixed and floating point and all conversions would be done in the register form. Also the accuracy of results, especially for technical computation, would be much greater.

1.2 Field Programmable Gate Arrays

Computer designers struggle to find balance between speed and generality. They can build chips, which are versatile but perform different function relatively slowly, or they can build devices to carry out specific tasks but do them more quickly. Microprocessors such as Intel Pentium or Motorola PowerPC are general-purpose designs. This means that they can carry out basically any logical or mathematical operation that can be programmed using instructions encoded in binary format. On the other hand, custom made hardware devices, known as Application-Specific Integrated Circuits (ASICs), are designed for a specific task. This allows producing chips which are smaller, faster and consume less energy than a general-purpose processor. The problem, which arises here for computer designers, is cost. If an ASIC is designed for a specified task and the task is slightly modified after the design is finished, then the ASIC will probably be incapable of solving the modified problem. If a modified ASIC can be developed to solve the new problem, the original hardware may be too highly customised to be reused in the new design. This will significantly increase the cost of the design and therefore cost of the final product.

New advances in electronics allow for a third solution. This is a use of large, fast, Field Programmable Gate Arrays (FPGAs). These circuits can be modified at almost any point during not only development but during the use. FPGAs consist of arrays of Configurable Logic Blocks (CLBs). They are like switches with multiple inputs and single output and they are used to perform operations such as AND, NAND, OR, NOR, XOR, etc. While in most hardware device logical functions of gates cannot be modified, in FPGAs, however, sending signals to the chip can change logic functions and the connections between the CLBs. This means that FPGAs can be re-programmed continuously and perform wide range of tasks with high speed and lower cost.

The structure of FPGA consists of a number of Configurable Logic Blocks and a programmable grid of connections that can link CLBs in any way depending on the design. The coarse-grained FPGAs have a small number of powerful CLBs; the fine-grained structure has many simple blocks. This means that a single element in a coarse-grained FPGA may be capable of adding or comparing two numbers, while one block in a fine-grained FPGA might be capable of comparing only two binary digits.

Field Programmable Gate Arrays allow for easy programming of the chip. The re-configuration times vary, with slower being able to reset within several seconds and the faster devices can be reset within one millisecond. The fast speed of re-configuration creates potential for use in video communication systems and in the image recognition field.

Other fields, where configurable computing can be successful, are pattern matching and encryption systems. Pattern matching is used in tasks such as handwriting recognition, face identification, database retrieval, and automatic target recognition. In case of target recognition, which is a military application, the greatest issue is the rapid comparison of an input image to thousands of templates. Another application mentioned above is encryption systems. There are ASICs, which are designed for only one kind of encryption algorithm. Configurable Computing would allow changing keys and algorithms and it would therefore make the encryption more secure and more convenient. The DES algorithm implemented on FPGAs is using 13,000-gate FPGA instead of 25,000-gate device previously required (Villasenor and Mangione-Smith 1997). These two examples show enormous flexibility of FPGAs. This, combined with increasing capacity and speed of FPGAs, can be used in digital communications, design automation and digital filtering for radar, while decreasing the cost of design.

Configurable Computing is still under development. Current FPGAs can have up to 100,000 logic elements and this will increase with further advances in technology. It is expected that by the end of the decade, FPGAs with a million logic elements will be developed. These will be used in highly complex communications and signal processing applications. Another possible application of FPGAs is Dynamic Instruction Set Computer (DISC). This would allow storing large number of circuit configurations, which could be activated by a programmer using a function call. Thanks to their flexibility, speed and overall cost, Field Programmable Gate Arrays provide lots of opportunity in many areas of computer science.

1.3 Objectives and Outline of Research

The aim of the composite arithmetic research is to implement the proposed standard and produce an Arithmetic Unit that could become a part of future CPUs. This thesis is the first research project on this topic. During the feasibility studies it was decided that the thesis should concentrate on the exact forms of the composite arithmetic. This would include integers and rational numbers. To further simplify the concept, integers would become a special case of rational numbers where the denominator is 1. The integer and rational separation would occur in further research.

The main emphasis of the thesis is to produce arithmetic circuits that will carry out operations on rational numbers. These can be then used in further research and serve as building blocks for the final Arithmetic Unit. To test the circuit designs, control unit needed to be developed. This was to be a simple, control circuit with limited number of operations that would be driven by a software driver. To store results internally, registers were required. These would store intermediate results of the calculations and the final outcomes that would be transferred to the PC. The transfer would be accomplished using an interface unit that would have circuitry to communicate with parallel port on the computer. Finally a small application program would be developed to produce results of arithmetic operations.

2.1 Composite Arithmetic Unit

The Composite Arithmetic Unit is the main objective of the thesis. The arithmetic operations presented here will form a basis for future versions of the CAU and allow carrying out arithmetic operations on rational numbers. Sections 2.1.1 and 2.1.2 describe Greatest Common Divisor and Casting Circuits respectively. Section 2.1.3 contains Swap Circuit. Multiply and Divide operations are in Section 2.1.4 and 2.1.5. Sections 2.1.6 and 2.1.7 describe Add and Multiply Circuits. The last two sections 2.1.8 and 2.1.9 describe Copy and Move Circuits.

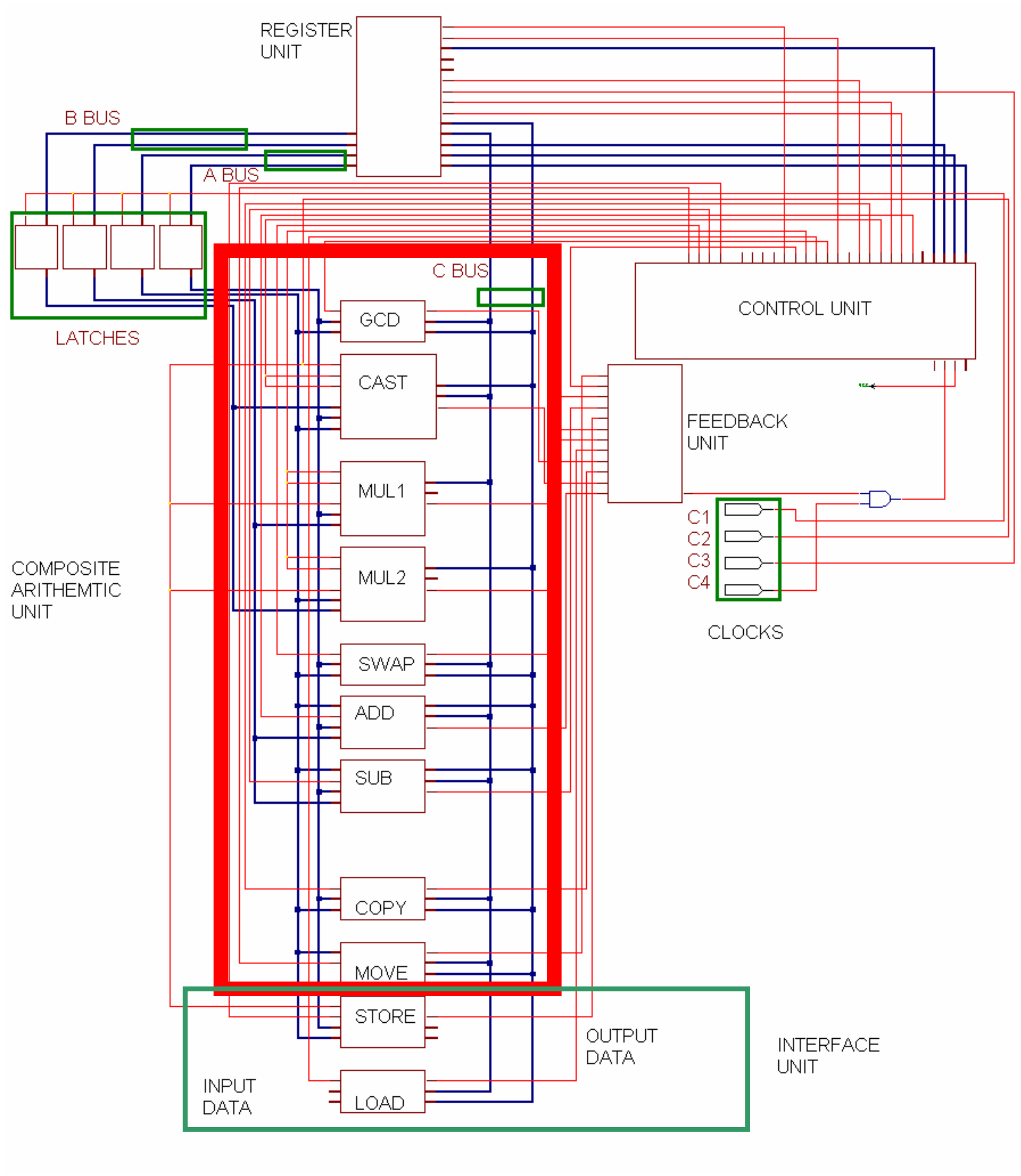


Figure 2. Simple Composite Arithmetic Machine (SCAM). The thick rectangle shows circuits of the CAU.

2.1.1 Greatest Common Divisor Circuit

The function of the circuit is to calculate the Greatest Common Divisor (GCD) of numerator and denominator of a rational number. The outcome of this operation is a number, which can be used with the rational number to cast out the common factors and reduce the size of the number.

The algorithm for finding GCD is based on the Euclid’s Algorithm. There are several variations of the Euclid’s Algorithm (including binary algorithm used in computing) but the following algorithm is a simple recursive function and can be easily implemented using hardware components.

```
int gcd(int m, int n)            // m is numerator, n is denominator
{
if (m < n)                       // when numerator is greater than denominator
    {
    m = gcd (m, n - m);          // make a recursive call with adjusted denominator
    return m;
    }
else    if (m > n)               // when numerator is less than denominator
        {
        m = gcd (m - n, n);      // make a recursive call with adjusted numerator
        return m;
        }
    else
        return m;                // When numerator equals denominator,
                                 // Stop and return the numerator (m),
                                 // Which holds the GCD of (m,n)
}
```

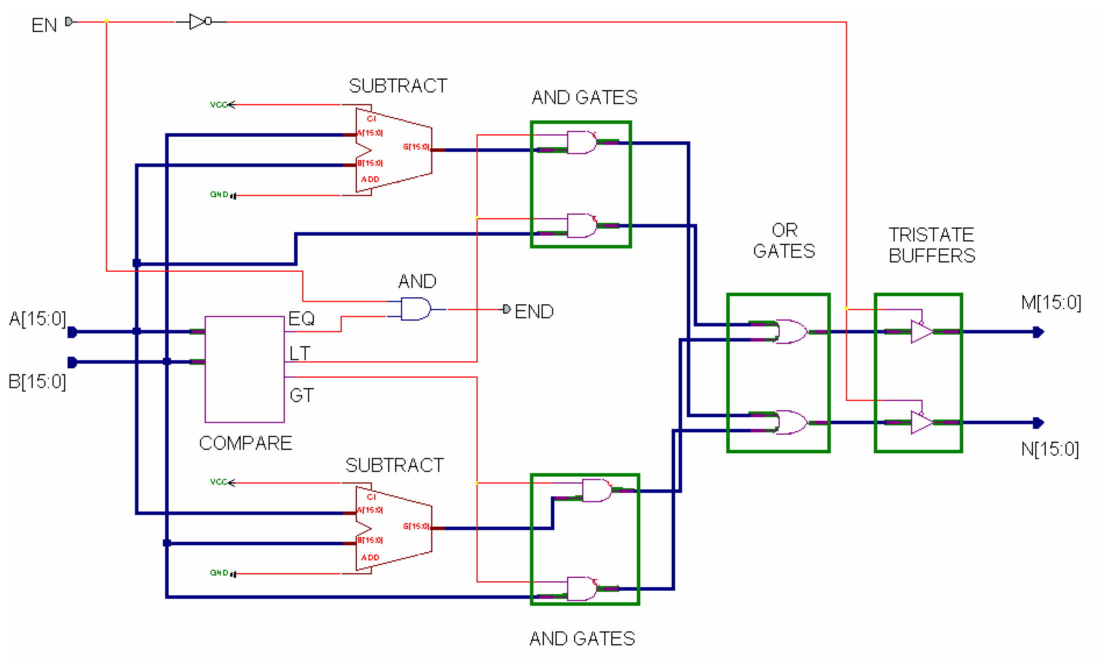


Figure 3. GCD Circuit follows Euclid's Algorithm to find Greatest Common Divisor.

Two input data buses are A [15:0] (denominator) and B [15:0] (numerator). The EN line enables the outputs and is used to generate the END signal. The output data buses are M [15:0] (denominator) and N [15:0] (numerator). The END line signals the end of operation and the result is contained in **M bus**. COMPARE unit checks if the two input buses are equal or **A bus** is less than **B bus**, or **A bus** is greater than **B bus**. If **A bus** equals **B bus** then EQ line goes high. It is then ended with EN line to set the line END to high. If **A bus** is less than **B bus** (LT line is high) then AND gates for the upper SUBTRACT unit are enabled. If **A bus** is greater than **B bus** (GT line is high) then AND gates for the lower SUBTRACT unit are enabled. Both outputs from the AND gates are put through OR gates and the final output is **M** and **N buses**, regulated using TRISTATE BUFFERS.

2.1.2 Casting Circuit

Casting Circuit is used to cast out the common factors. The GCD input is calculated using the circuit in Section 2.1.1. This value is stored in a register and then fetched into the casting circuit. Casting out common factors allows decreasing the size of the numerator and denominator and still maintaining the precision. It simplifies calculation just like in normal paper-and-pencil rational arithmetic.

The algorithm for casting out common factors uses simple subtraction of GCD from numerator and denominator until they reach 0. The count of subtractions for each number produces the new numerator and denominator. This is equivalent to dividing the rational number by the GCD.

```
void cast(int& m, int& n, int gcd) // m is the numerator, n is the denominator
{
int count1=0;              // counts subtractions of the numerator
int count2=0;              // counts subtraction of the denominator
while (m > 0)              // while loop for the numerator
{
      count1 = count1 + 1; // add one to the counter
      m = m - gcd;         // subtract the GCD from the numerator
}
while (n > 0)              // while loop for the denominator
{
      count2 = count2 + 1; // add one to the counter
      n = n - gcd;         // subtract the GCD from the denominator
}
m = count1;                // The new numerator is set to the count of subtractions
n = count2;                // The new denominator is set to the count of subtractions
}
```

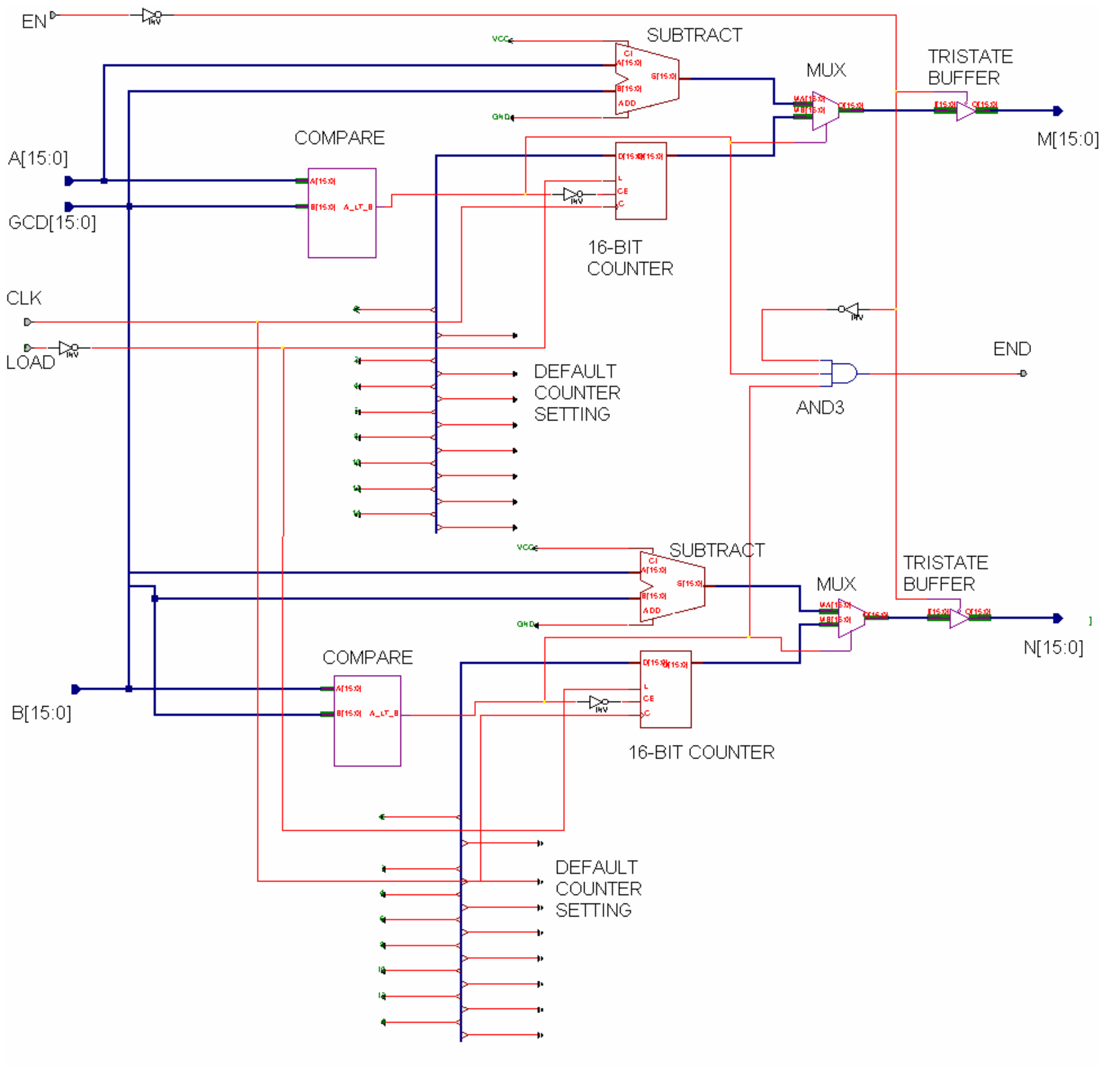


Figure 4. Cast Circuit casts out common factors of two 16-bit numbers.

Three data inputs are A [15:0] (denominator), B [15:0] (numerator) and GCD [15:0] (GCD of A and B calculated earlier). The input lines are EN (enable), CLK (clock), and LOAD (load counter). The data outputs are M [15:0] (denominator) and N [15:0] (numerator). The output line is END, which signifies the end of operation. The description given is for **A bus** only as **B bus** is identical. **A bus** is put through the COMPARE units with **GCD bus**. While the output of the COMPARE is low (meaning GCD is less than **A bus**) the 16-BIT COUNTER is enabled. Otherwise it is disabled. The COMPARE output line also provides feedback to the END line and it manipulates the MUX selector. SUBTRACT units subtract **GCD bus** from **A bus**. The output of the SUBTRACT and 16-BIT COUNTER are put through the MUX selector. If the COMPARE output line is low the SUBTRACT output goes through, and if it is high the 16-BIT COUNTER output goes through. The final output M [15:0] is put through the TRISTATE

BUFFER. The **B bus** operates on the same principle and the output goes to N [15:0]. DEFAULT COUNTER SETTING lines are set of GND and VCC connections to set up the initial counter (using LOAD line).

2.1.3 Swap Circuit

This simple circuit swaps numerator and denominator to produce the reciprocal of the original number. The result of this operation is applied in Division.

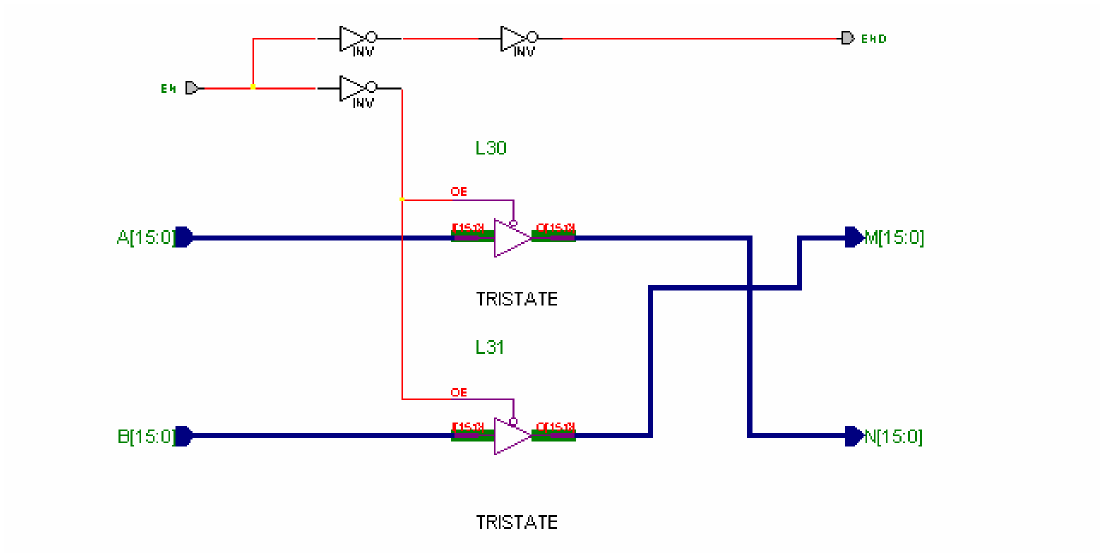


Figure 5. Swap Circuit is used in conjunction with Division operation.

The input data buses are A [15:0](denominator) and B [15:0](numerator). These are put through tristate buffers using EN line to activate the buffers. Then **A bus** is tied to N [15:0] (numerator) and **B bus** is tied to the M [15:0] (denominator). The final output line is END, which has been inverted twice to strengthen the signal.

2.1.4 Multiply Circuit

The multiply circuit allows multiplication of two rational numbers by multiplying the numerators and denominators separately.

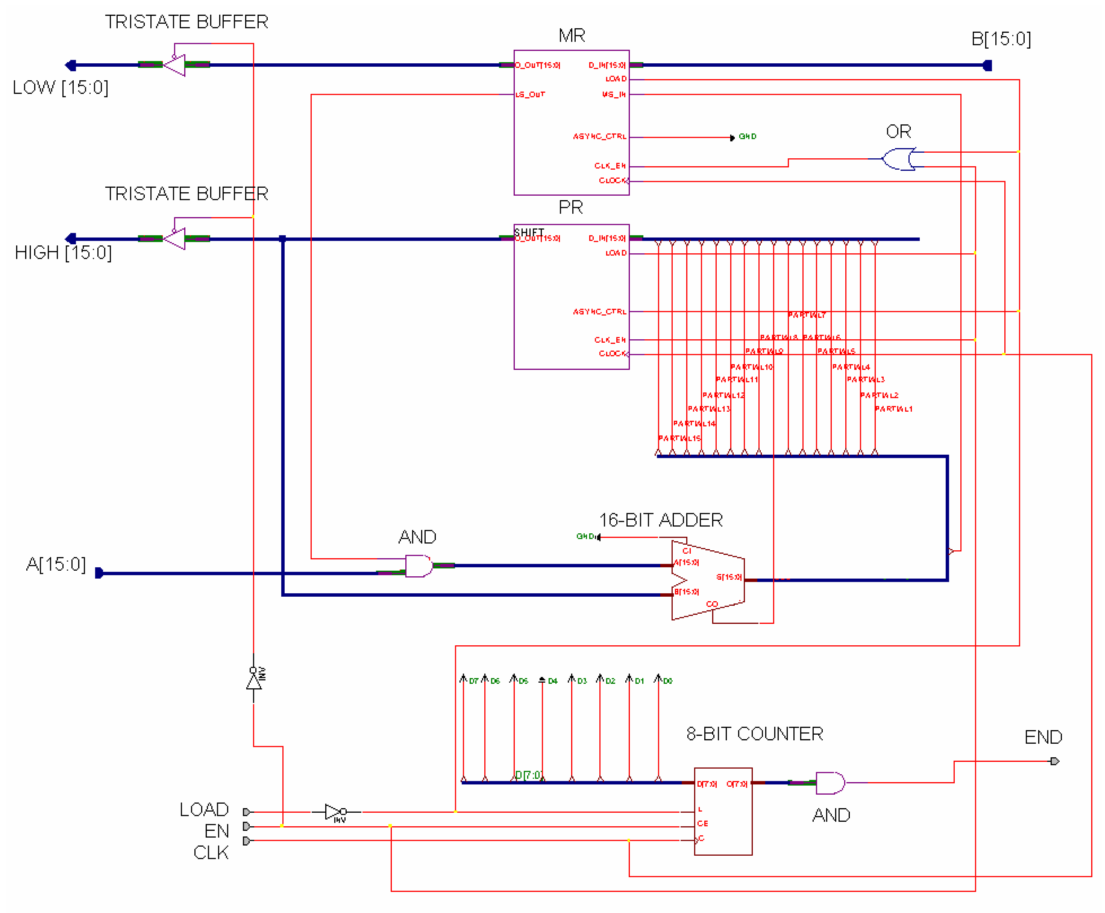


Figure 6. Multiply is the most important circuit because it is used with Division, Addition and Subtraction operations.

The design for this circuit is based on Hwang's algorithm (Hwang 1979). The two input buses are A [15:0] (multiplicand) and B [15:0] (multiplier). The input lines are LOAD (loads the counter), EN (enable line), CLK (clock line). The output buses are LOW [15:0] (lower 16-bits of the result) and HIGH [15:0] (higher 16-bit of the result). There are three registers involved in the circuit. The first two are internal. These are MR (Multiplier Register) and PR (Partial product Register). The third register is a register from the Register Unit containing the Multiplicand (**A bus**). Also 16-BIT ADDER is required and 8-BIT COUNTER. The result of the multiplication is a 32-bit number, which is stored in **LOW** and **HIGH buses**, corresponding to 16 lower bits and 16 higher bits respectively. 8-BIT COUNTER is used to keep track of the number of additions and to signal the completion of operation by setting the END line to high. Each clock cycle there is number of operations, which occur. First Multiplicand A is ANDed with LSB of MR. This result is added to the PR (initially zero). The 15 higher bits of the sum are put into 15 lower bits of the PR. The LSB of the sum is right shifted into MR and becomes the MSB of the MR and Cout from the ADDER is put into the MSB of the PR. The 8-BIT COUNTER controls the number of right shifts of the MR, so at the end the MR contains 16 lower bit of the result. The Multiplier B that was originally in the MR is pushed towards its right end.

2.1.5 Divide Circuit

The divide circuit is in fact a logical operation carried out by the software driver. It is used to multiply a number with a reciprocal of another to give a result of rational division. More detailed explanation is contained in Chapter 3, where the division operation is simulated.

2.1.6 Add Circuit

This circuit performs addition of numerators of two rational numbers. The denominators must be the same in order to produce the correct results. This is achieved using the multiply circuit.

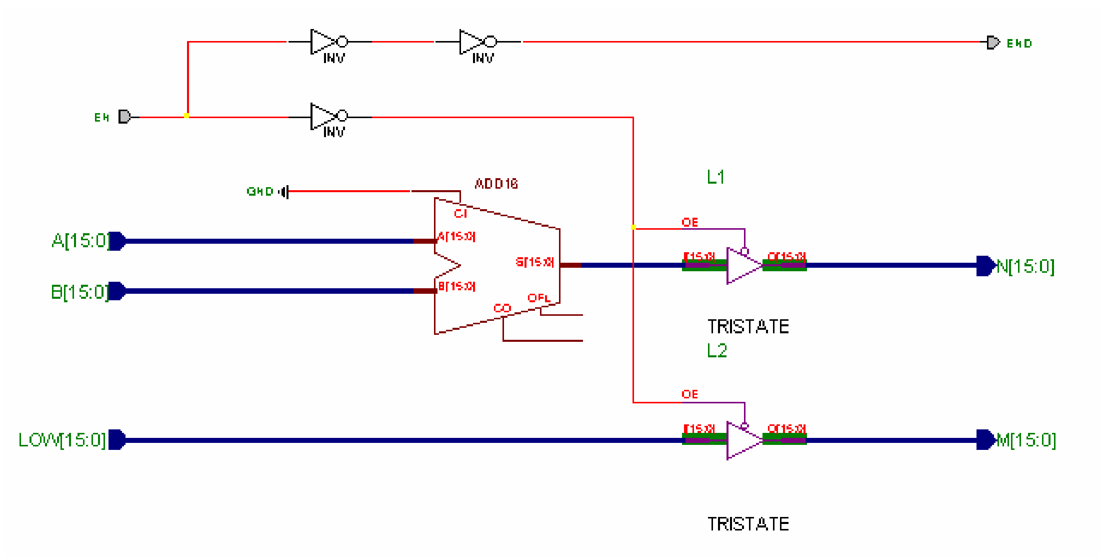


Figure 7. Add Circuit.

The three data inputs are A [15:0], B [15:0] and LOW [15:0]. **A** and **B buses** are numerators of the added numbers, while **LOW bus** is the common denominator.

The input line EN enables the whole circuit. The circuit works by adding A and B and sending the result to N [15:0] via TRISTATE buffer. The **LOW bus** goes straight to the M [15:0] output bus and is enabled using TRISTATE buffer. The output line END has been inverted twice to strengthen the signal.

2.1.7 Subtract Circuit

The subtraction circuit performs a function similar to the addition circuit by subtracting the numerators.

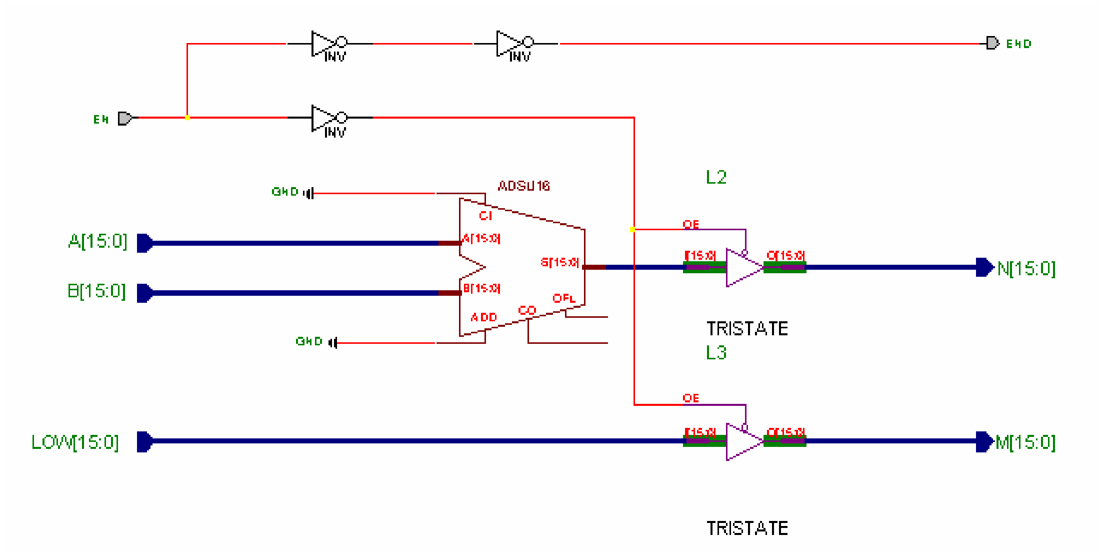


Figure 8. Subtract Circuit.

The three data inputs are A [15:0], B [15:0] and LOW [15:0]. **A** and **B buses** are numerators of the subtracted numbers, while **LOW bus** is the common denominator. The input line EN enables the whole circuit. The circuit works by subtracting B from A and sending the result to N [15:0] via TRISTATE buffer. The **LOW bus** goes straight to the M [15:0] output bus and is enabled using TRISTATE buffer. The output line END has been inverted twice to strengthen the signal.

2.1.8 Copy Circuit

This circuit allows copying a rational number from one register into another register.

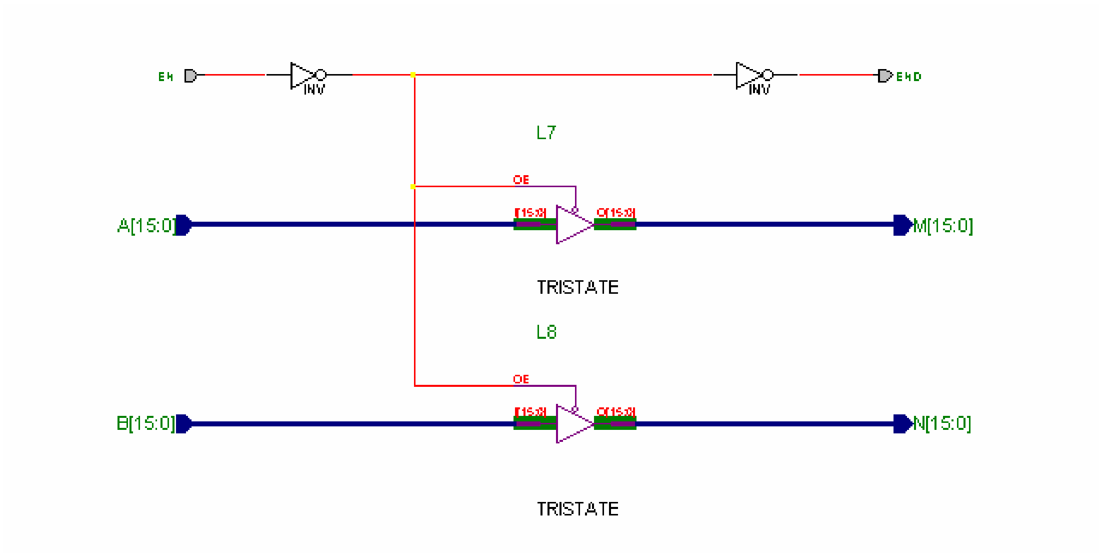


Figure 9. Copy Circuit.

The inputs are A [15:0] (denominator), B [15:0] numerator and EN line (enable). The two input buses are put through a pair of TRISTATE buffers and are fed to the two output buses: M [15:0] and N [15:0]. The TRISTATE buffers are enabled by the EN line and the output line END is inverted twice to strengthen the signal.

2.1.9 Move Circuit

This circuit copies denominator of a rational number and puts it into a new register as numerator and denominator (value of 1). It is then used with addition.

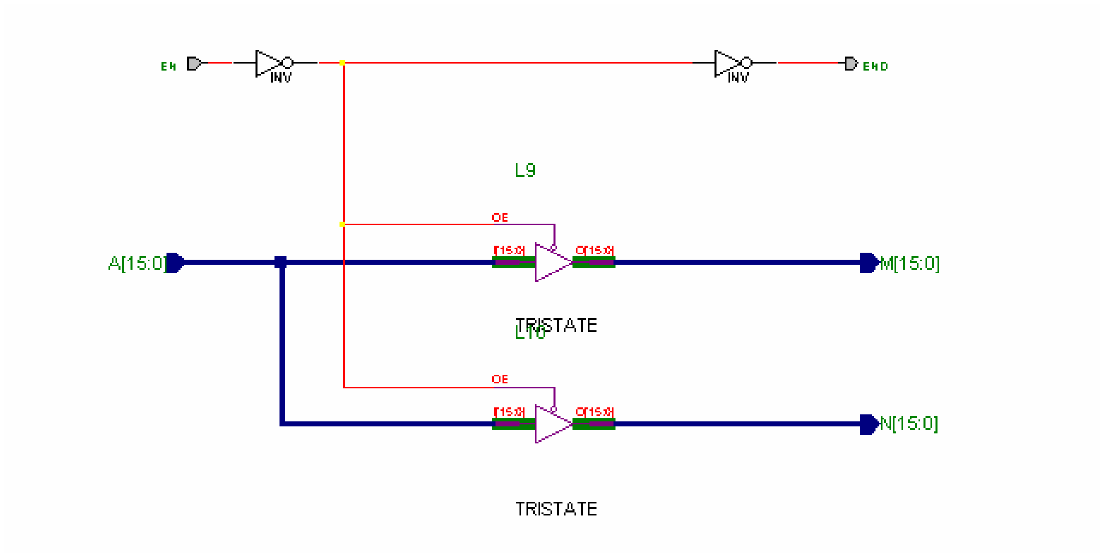


Figure 10. Move Circuit.

The inputs are A [15:0] (denominator) and EN line (enable). The input bus is put through a pair of TRISTATE buffers and is fed to the two output buses: M [15:0] (denominator) and N [15:0] (numerator). The TRISTATE buffers are enabled by the EN line and the output line END is inverted twice to strengthen the signal.

2.2 Register Unit

The Register Unit is based on logical design of Tanenbaum (Tanenbaum 1990). The unit contains 9, 32-bit general-purpose registers (non-shifting). There is two input buses (CL [15:0] and CH [15:0]) and four output buses (AL [15:0], AH [15:0], BL [15:0] and BH [15:0]). The output buses are buffered from the main bus using tristate buffers. The unit can be upgraded to accommodate up to 16 registers. Section 2.2.1 describes the 32-bit register in detail and Section 2.2.2 explains Register Address Decoding.

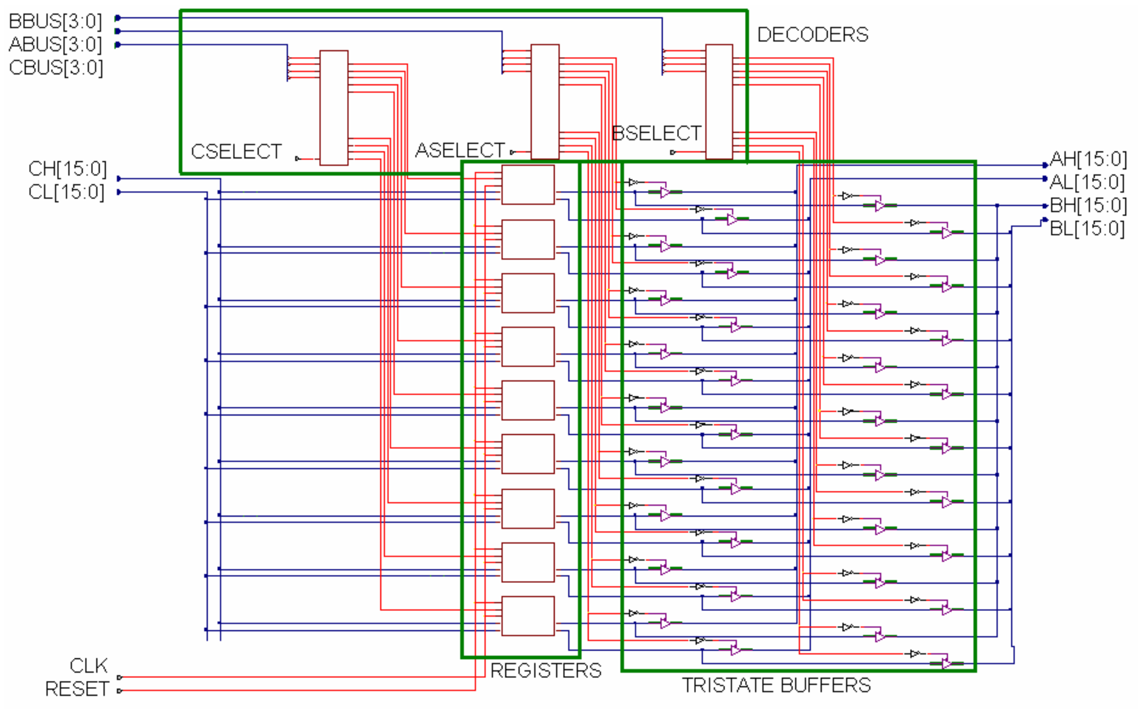


Figure 11. Register Unit.

2.2.1 32-bit Register Circuit

The 32-bit Register is used to store a rational number. Lower 16 bits are allocated for the denominator and higher 16 bits are allocated for the numerator.

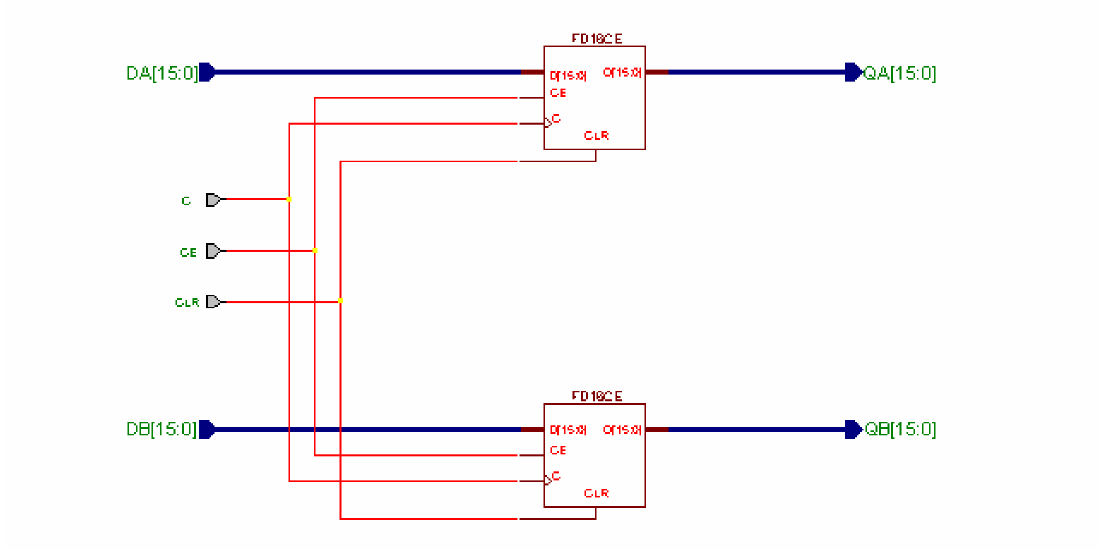


Figure 12. 32-bit Register consists of two 16-bit Registers.

The register has two data inputs. The first one is **DA bus** and represents the denominator (lower 16 bits). The second data input is **DB bus** and it represents the numerator (higher 16 bits). Input line CLR clears the register, CE line enables the clock and C line is a clock, which is edge activated. The outputs consist of two data buses. The first one is QA (denominator) and the second one QB (numerator).

2.2.2 Register Address Decoding Circuit

The Address Decoding Circuit is used to enable and disable I/O operations within the Register Unit according to the data from the Control Unit. These operations allow for the data to be stored in registers or data can be retrieved from registers and put on the main bus. Figure 11 show the decoding circuits as integral part of the Register Unit.

The Address-Decoding Unit is a set of three identical 4-to-16 DECODERS. The three lines that enable each Decoder are ASELECT, BSELECT and CSELECT. These are sent from the Control Unit (Section 2.3.2). The three input buses are ABUS [3:0], BBUS [3:0] and CBUS [3:0]. These buses contain a number of the register to be used. CBUS is used for input purposes. The output of the CBUS Decoder enables the registers and data is stored during the clock cycle. The ABUS and BBUS are used for controlling output from the registers to their respective buses. They enable (or disable) the TRISTATE BUFFERS depending on the value of ABUS and BBUS.

2.3 Control Unit

Control Circuit is composed of two units: Control Unit and Feedback Unit. Feedback Unit is described in detail in Section 2.3.3. The Control Unit is responsible for interpreting the Control Word and enabling specific parts of the system. It can enable arithmetic operations, input to the Register Unit and output from the Register Unit. If the Control Unit is idle the data from the Parallel Port Interface can be accepted. There is one input data bus and it is BUS [31:0]. It contains Control Word. The following Figure shows the breakdown of the Control Word.

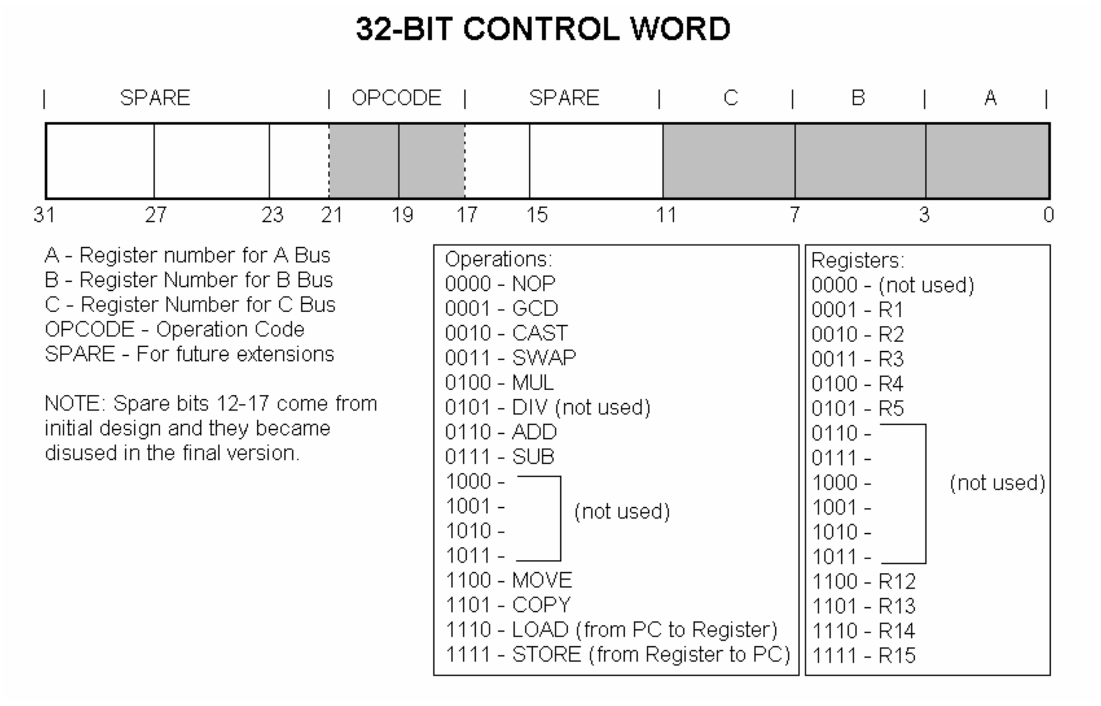


Figure 13. SCAM's Control Word.

The single lines are CNTL (from the I/O Interface Unit), E (enables DECODER) and FEEDBACK line (from the Feedback Unit). The three output buses are ABUS [3:0], BBUS [3:0] and CBUS [3:0]. These carry the register number (4-bits) that will be used during an arithmetic operation. The output lines are divided into Operation Lines and Register Selection Lines. Operation Lines consist of NOP, MOVE, GCD, CAST, SWAP, MUL, DIV, ADD, SUB, COPY, LOAD and STORE. These correspond to the operations carried out by the SCAM. The Register lines are ASELECT, BSELECT and CSELECT. They indicate which data bus will be enabled for a particular operation. The Section 2.3.1 describes how the Operation Decoding works and Section 2.3.2 describes Register Selection Circuit. Section 2.3.3 explains the Feedback Unit as mentioned above.

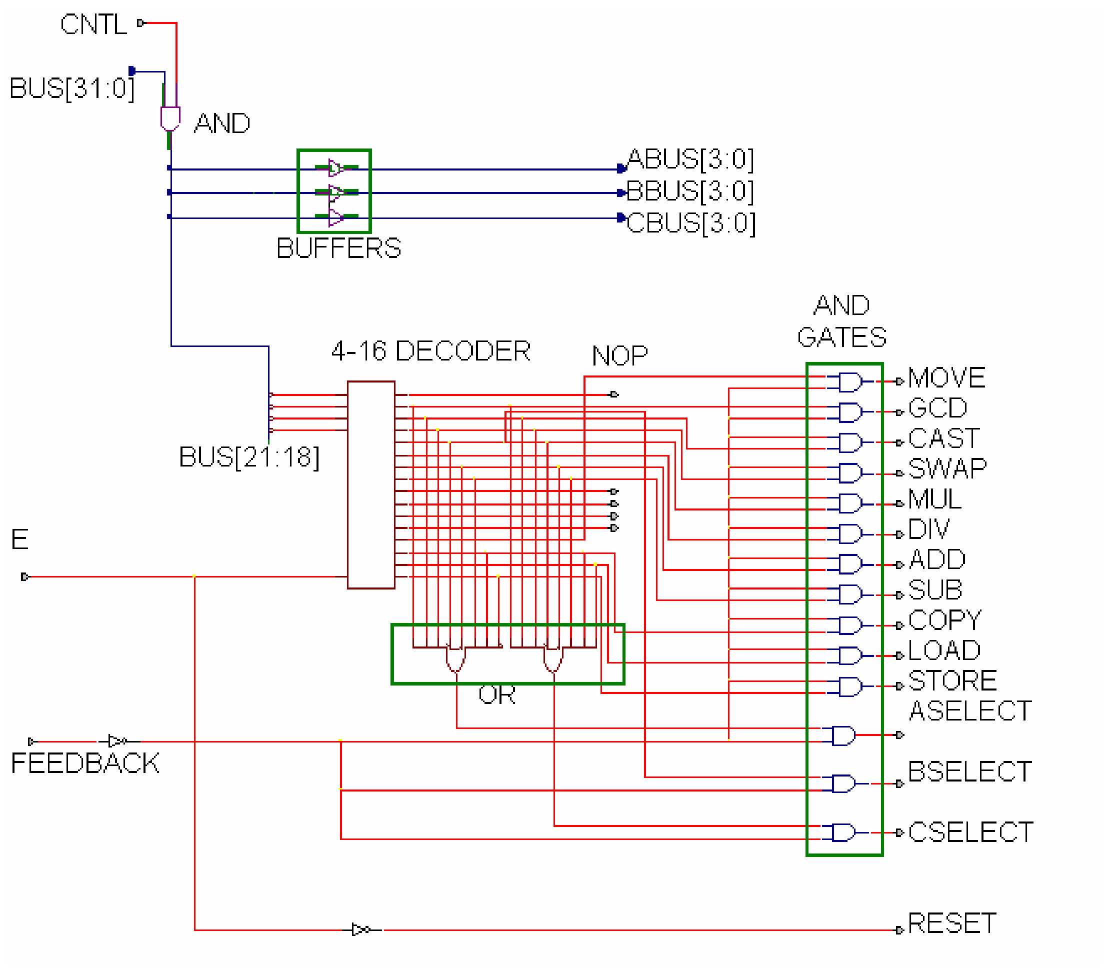


Figure 14. Control Unit.

2.3.1 Operations Decoding Circuit

The Operation Decoding Circuit is used to enable the Operation Lines based on the Control Word. Bits 18 to 21 of the Control Word are reserved for the opcode and are used here.

The BUS [21:18] carries the opcode. It is fed into 4-to-16 DECODER to produce a maximum number of 16 operations. At the moment 11 operation slots are used with 5 spares. The DECODER produced outputs are then ANDed with the FEEDBACK line to produce the right timing. Only NOP operation is not ANDed with FEEDBACK. The DECODER is enabled by E line, which is always high. The E line also provides the RESET line with an inverted signal (always low). This line is made for future extensions where E line will be connected to some feedback unit allowing resetting the registers.

2.3.2 Register Selection Circuit

Register Selection Circuit allows for the appropriate registers to be selected based on the Control Word. The selection of which register to be used is done using ABUS [3:0], BBUS [3:0] and CBUS [3:0]. These correspond to the BUS [3:0], BUS [7:4] and BUS [11:8], which are parts of the Control Word bus. This output is filtered using BUFFERS. To select which data bus will be used ASELECT, BSELECT and CSELECT lines are used. The enable the right data paths in the Register Unit (Section 2.2.2). ASELECT is created by ORing all operations that use **A bus** and ANDing the output with FEEDBACK line. CSELECT is done in the same fashion. BSELECT line contains only one line (Multiply) and it is ANDed with FEEDBACK. (Note: CAST operation requires **B bus** and the new line has been added to the BSELECT in the later version of the circuit. The new version has MUL and CAST lines ORed and the result is ANDed with FEEDBACK line.)

2.3.3 Feedback Unit

Feedback circuit is used to gain information from executing operations (Add, Sub, Mul, Div, etc.) about the state of completion (executing or finished). These inputs are combined and a single feedback line is sent to the Control Unit (Section 2.3.1 and 2.3.2).

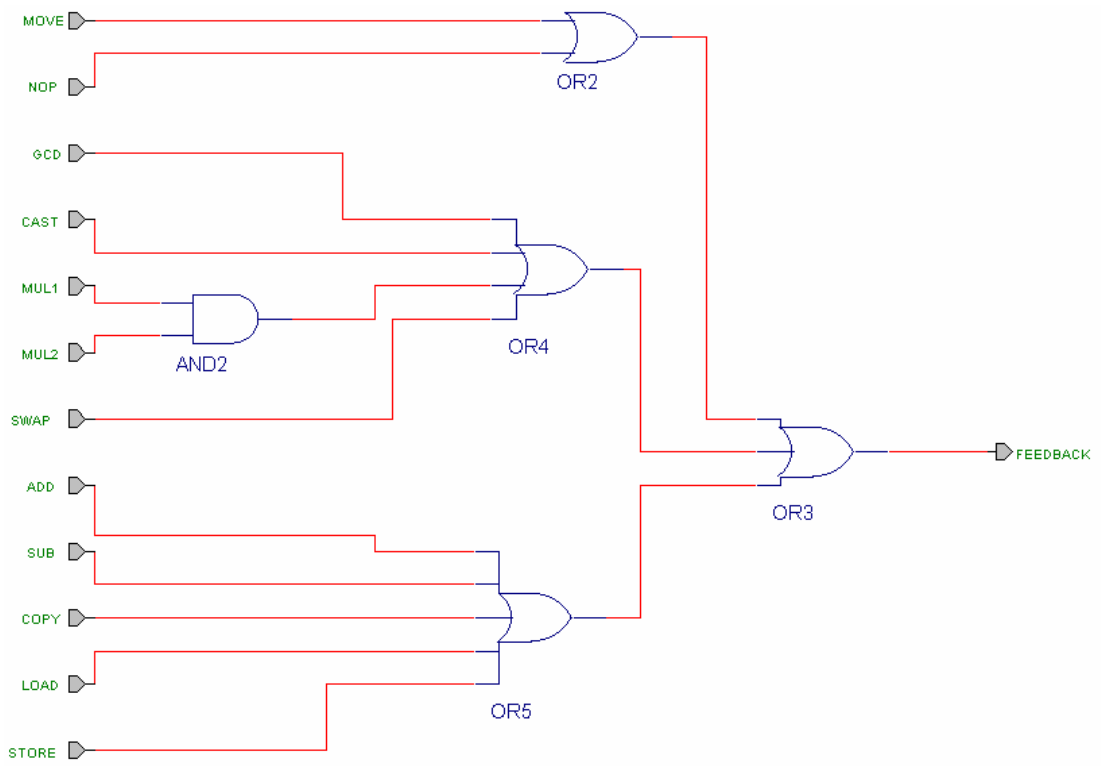


Figure 15. Feedback Unit.

There are 12 input lines in the Feedback Unit. These are MOVE, NOP, GCD, CAST, MUL1, MUL2, SWAP, ADD, SUB, COPY, LOAD and STORE. Most of these lines come directly from the corresponding arithmetic circuits. Line NOP comes from the Control Unit and is used for I/O control. The MUL1 and MUL2 lines come from the two multiply circuits, one for numerator, the other for denominator. They are ANDed together to give only one line. All other inputs are ORed together and the resulting line FEEDBACK is sent to the Control Unit.

2.4 Input/Output Interface Unit

2.4.1 Parallel Port Interface

The Parallel Port Interface allows SCAM to communicate with a PC. It allows for the Control Word to be received, followed by a Data Word. The I/O Unit transforms series of eight 8-bit inputs into a single 32-bit input and two 16-bit inputs. Currently there are some timing problems when integrated with the rest of the system and the design of the output part is postponed, until these problems can be resolved.

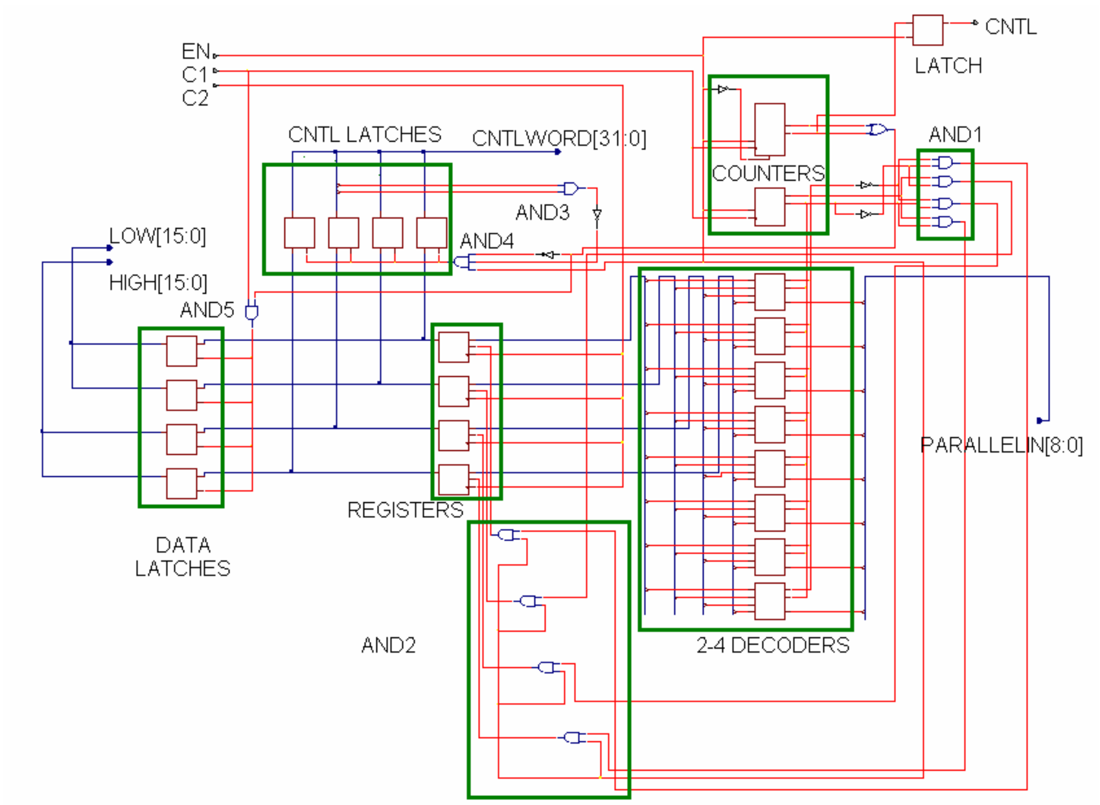


Figure 16. Input/Output Interface Unit.

The data input is PARALLELIN [8:0]. It contains data from the parallel port. Lines EN (enable), C1 and C2 (clocks) are used to control the different part of the circuit. The outputs are CNTLWORD [31:0] (Control Word), LOW [15:0] (denominator), HIGH [15:0] (numerator) and CNTL lines, which goes back to the Control Unit.

2.4.2 Load Circuit

Load Circuit accepts data from the Parallel Port Interface and sends it onto the main bus to be stored in a register.

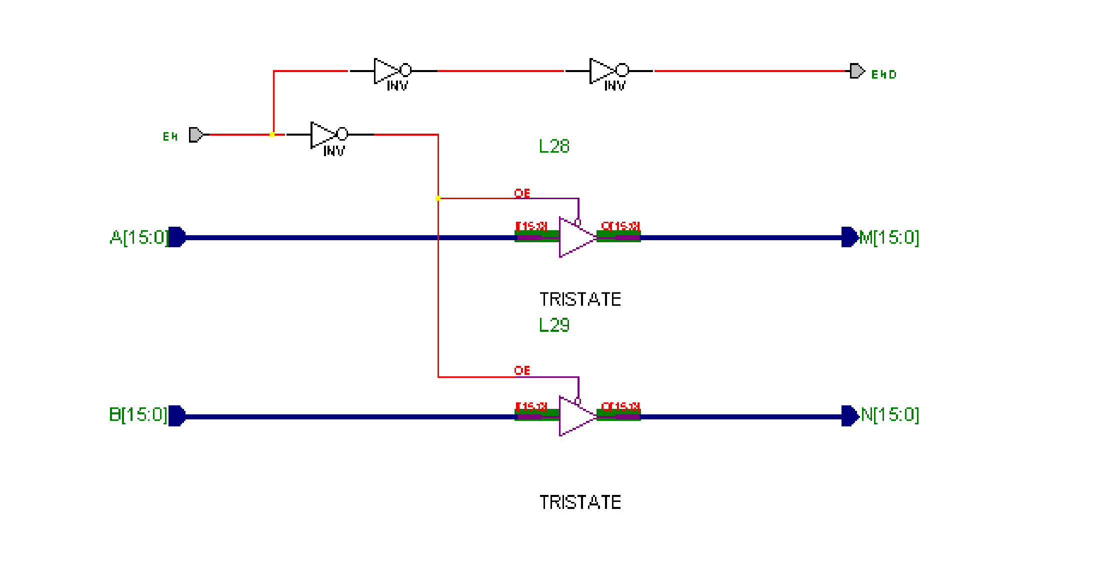


Figure 17. Load Circuit

Three inputs are A [15:0] (denominator), B [15:0] (numerator) and EN (enable line). The outputs are M [15:0](denominator), N [15:0] (numerator) and END line. The input from the Parallel Port Interface is fed through A and B and is put through the TRISTATE buffers. These are enabled using the EN line and the outputs M and N are put onto the main bus. The output line END is inverted twice to strengthen the signal.

2.4.3 Store Circuit

Store Circuit accepts data from the Register Unit and sends it to the Parallel Port Interface.

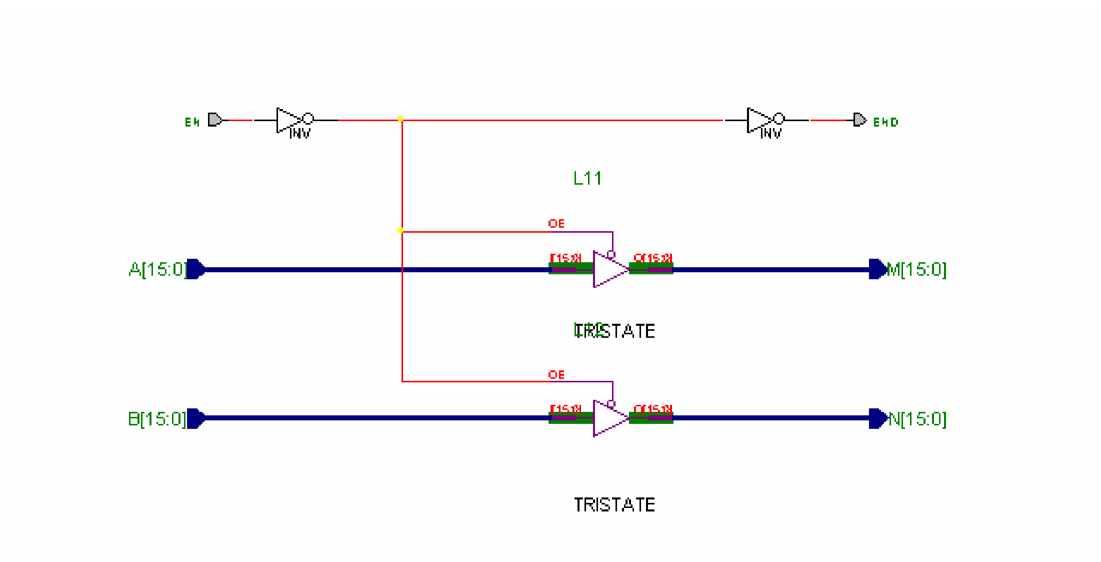


Figure 18. Store Circuit.

Three inputs are A [15:0] (denominator), B [15:0] (numerator) and EN (enable line). The outputs are M [15:0](denominator), N [15:0] (numerator) and END line. The input from the Register Unit is fed through A and B and is put through the TRISTATE buffers. These are enabled using the EN line and the outputs M and N are sent to the Parallel Port Interface. The output line END is inverted twice to strengthen the signal.

3.1 Composite Arithmetic Unit

3.1.1 Greatest Common Divisor Circuit

The simulation of the GCD Circuit is part of the Simulation 1 (Appendix B.3). The GCD Circuit takes contents of a register, processes them and returns data to the same register.

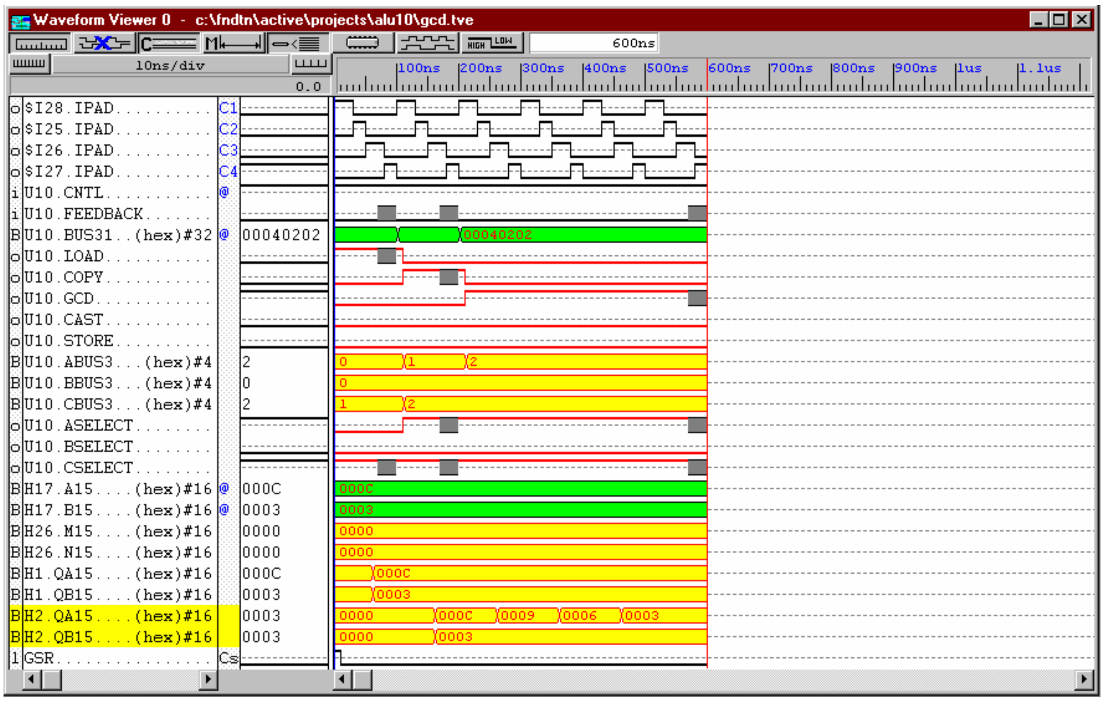


Figure 19. GCD Simulation.

The first two cycles are for the LOAD and COPY commands. The GCD Circuit starts at 200ns. The inputs are H2.QA15 and H2.QB15 and they are 0Ch and 03h respectively. The GCD continues to decrement the larger number (0Ch to 09h, to 06h, to 03h) until it equals with the other number. Then the feedback line goes high and execution is finished. The result is contained in H2.QA15. The whole operation is done in four clock cycles. The number of clock cycles for GCD depends on the size of the numbers and the difference between the two numbers.

3.1.2 Casting Circuit

The Casting Circuit simulation is part of the Simulation 1 (Appendix B.3). The Casting Circuit takes contents of two registers, processes them and returns data to the first of the registers.

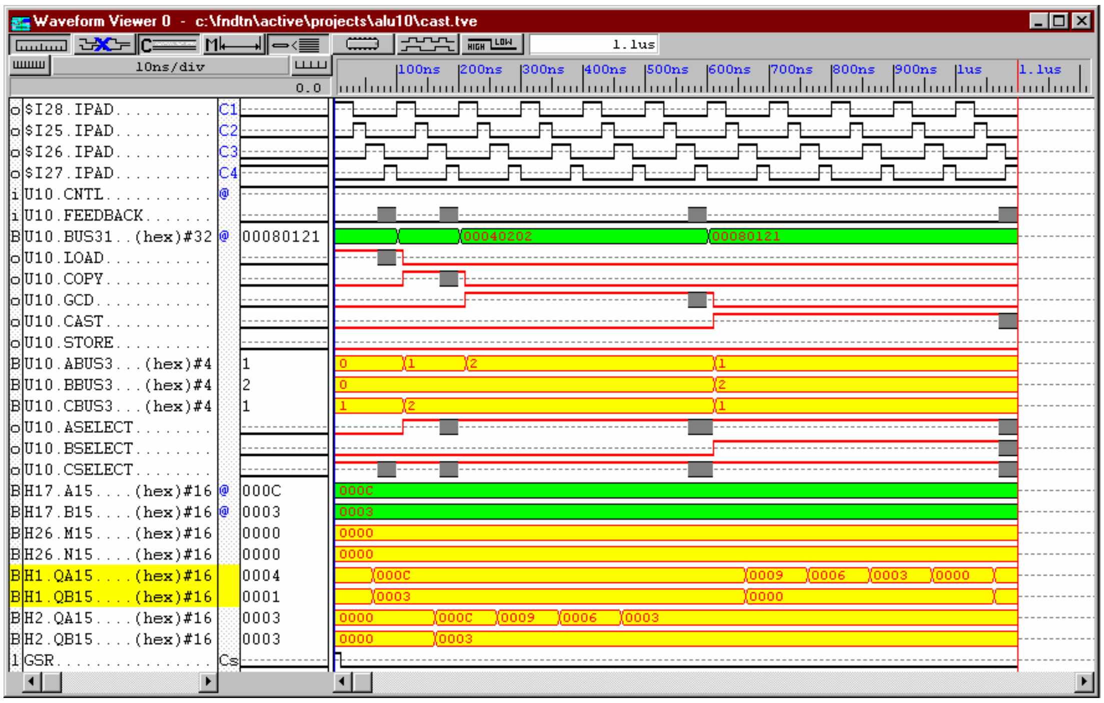


Figure 20. Casting Simulation.

The casting operation starts at 600ns (after GCD). The three inputs are H1.QA15, H1.QB15 AND H2.QA15 (GCD). The Casting Circuit subtracts GCD from both numbers until they reach zero. Once both numbers reach zero, the circuit returns the count of the subtractions for the respective numbers. These are stored back into H1.QA15 and H1.QB15. The operation is done in five cycles but the number of cycles depends on the size of the numbers and the value of the GCD.

3.1.3 Swap Circuit

This is a very simple circuit and is part of Simulation 3 (Appendix B.5). It is used for Division in conjunction with Multiply Circuit. The two inputs are simply swapped and put in the same register.

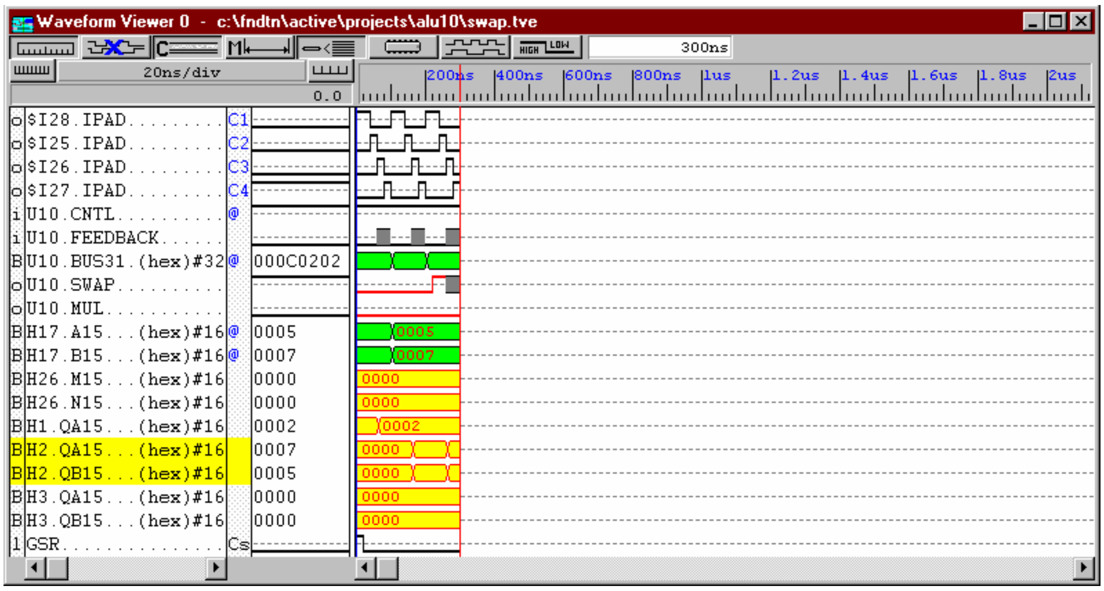
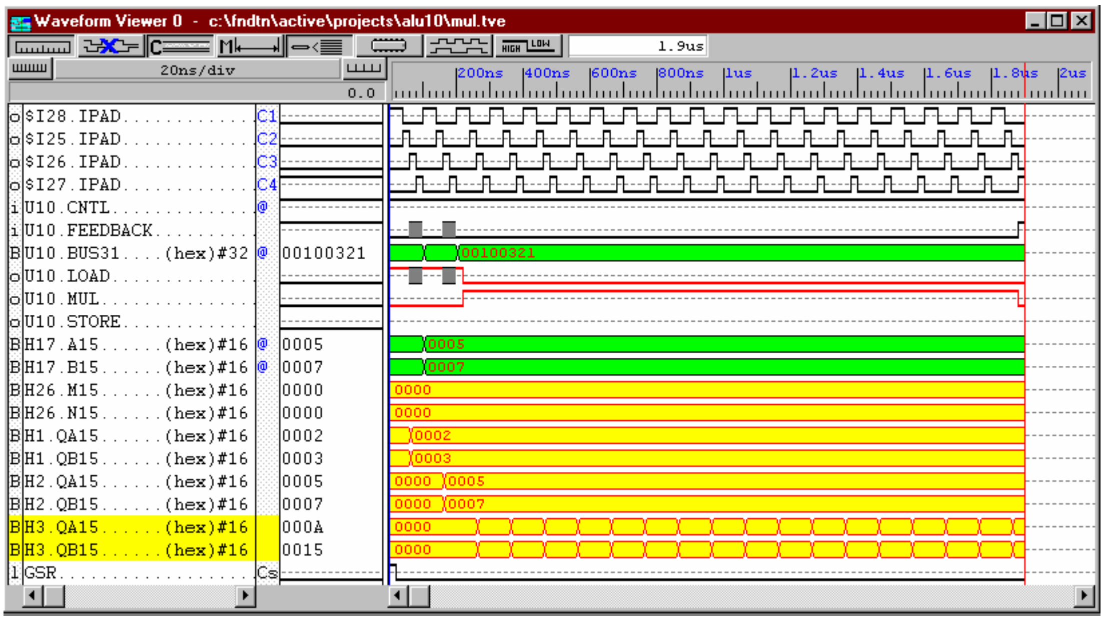

```
Waveform Viewer 0 - c:\fndtn\active\projects\alu10\swap.tve
20ns/div                         300ns
0.0
o $I28.IPAD.........  C1
o $I25.IPAD.........  C2
o $I26.IPAD.........  C3
o $I27.IPAD.........  C4
i U10.CNTL..........  @
i U10.FEEDBACK......
B U10.BUS31.(hex)#32  @  000C0202
o U10.SWAP..........
o U10.MUL...........
B H17.A15...(hex)#16  @  0005
B H17.B15...(hex)#16  @  0007
B H26.M15...(hex)#16     0000
B H26.N15...(hex)#16     0000
B H1.QA15...(hex)#16     0002
B H2.QA15...(hex)#16     0007
B H2.QB15...(hex)#16     0005
B H3.QA15...(hex)#16     0000
B H3.QB15...(hex)#16     0000
1 GSR...............  Cs
```

Figure 21. Swap Simulation.

The initial numbers are loaded into H2.QA15 and H2.QB15. In the third cycle (at 200ns) the SWAP command is carried out. The numbers are swapped and put back into H2.QA15 and H2.QB15. The SWAP command is carried out during one clock cycle.

3.1.4 Multiply Circuit

The Multiply Circuit is part of Simulation 2 (Appendix B.4). It multiplies two registers and puts the output in the third register. Only the lower 16-bits are used within this CAU, which means that multiplication will work for 8-bit numbers only. For example, 8-bit numerator and 8-bit denominator multiplied by 8-bit numerator and 8-bit denominator will produce 16-bit numerator and 16-bit denominator. The circuit design (Section 2.1.4) incorporates the 32-bit extension but it is not used within the SCAM design.



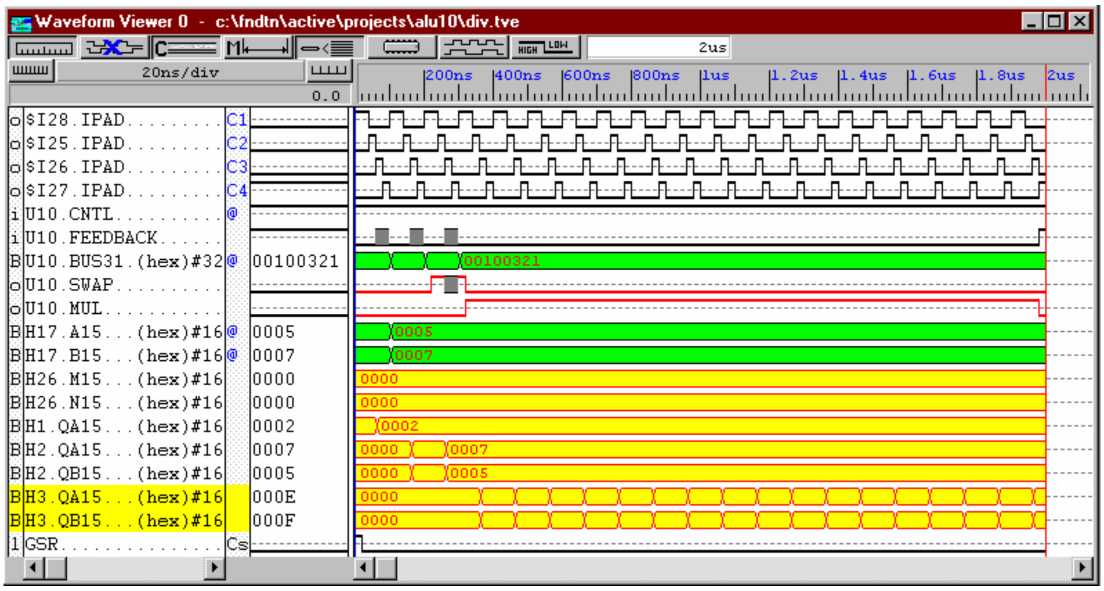
```
Waveform Viewer 0 - c:\fndtn\active\projects\alu10\mul.tve
20ns/div                                  1.9us
0.0
o $I28.IPAD............        C1
o $I25.IPAD............        C2
o $I26.IPAD............        C3
o $I27.IPAD............        C4
i U10.CNTL.............        @
i U10.FEEDBACK.........
B U10.BUS31....(hex)#32        @   00100321
o U10.LOAD.............
o U10.MUL..............
o U10.STORE............
B H17.A15......(hex)#16        @   0005
B H17.B15......(hex)#16        @   0007
B H26.M15......(hex)#16            0000
B H26.N15......(hex)#16            0000
B H1.QA15......(hex)#16            0002
B H1.QB15......(hex)#16            0003
B H2.QA15......(hex)#16            0005
B H2.QB15......(hex)#16            0007
B H3.QA15......(hex)#16            000A
B H3.QB15......(hex)#16            0015
1 GSR..................        Cs
```

Figure 22. Multiply Simulation.

The MUL operation starts at 200ns. The four inputs are H1.QA15, H1.QB15, H2.QA15 and H2.QB15. The outputs H3.QA15 and H3QB15 contain partial sums during the calculation and they contain the final result at the end of the operation. The multiply operation is controlled by a counter. For a 16-bit multiplication the counter counts over 17 clock cycles. The first clock cycle is used to set the internal registers within the Multiply Circuit, while the other 16 are used for calculation of partial sums and for internal shift of registers.

3.1.5 Divide Circuit

The Divide Circuit is very similar to the Multiply Circuit but has SWAP operation carried out before it. More details can be found in the Appendix B.5 (Simulation 3).

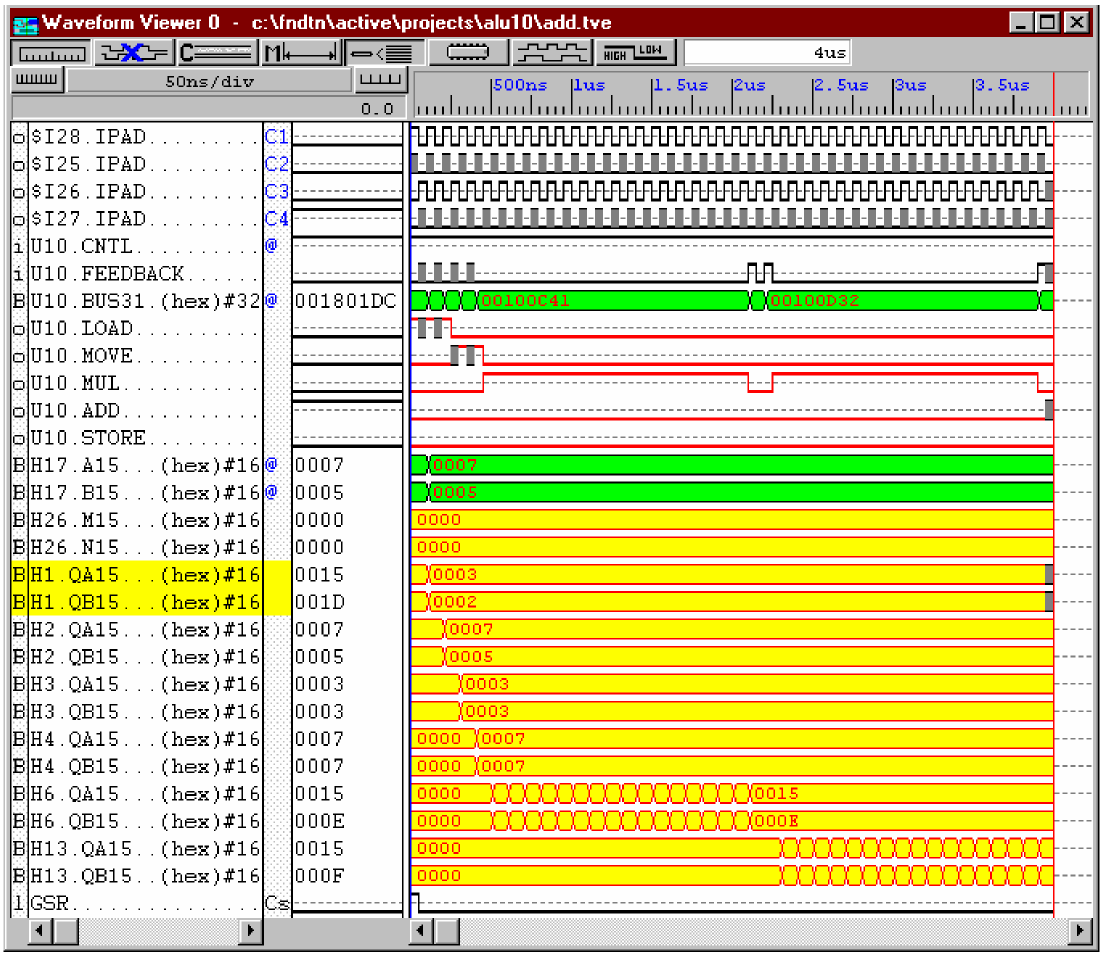


```
o $I28.IPAD.........         C1
o $I25.IPAD.........         C2
o $I26.IPAD.........         C3
o $I27.IPAD.........         C4
i U10.CNTL..........         @
i U10.FEEDBACK......
B U10.BUS31.(hex)#32         @   00100321
o U10.SWAP..........
o U10.MUL...........
B H17.A15...(hex)#16         @   0005
B H17.B15...(hex)#16         @   0007
B H26.M15...(hex)#16             0000
B H26.N15...(hex)#16             0000
B H1.QA15...(hex)#16             0002
B H2.QA15...(hex)#16             0007
B H2.QB15...(hex)#16             0005
B H3.QA15...(hex)#16             000E
B H3.QB15...(hex)#16             000F
1 GSR...............         Cs
```

Figure 23. Divide Simulation.

3.1.6 Add Circuit

The Add Circuit is part of the Simulation 4 (Appendix B.6). It adds numbers from two registers and puts the result in another register. This circuit requires prior use of the Multiply Circuit to make the denominators the same for both numbers.



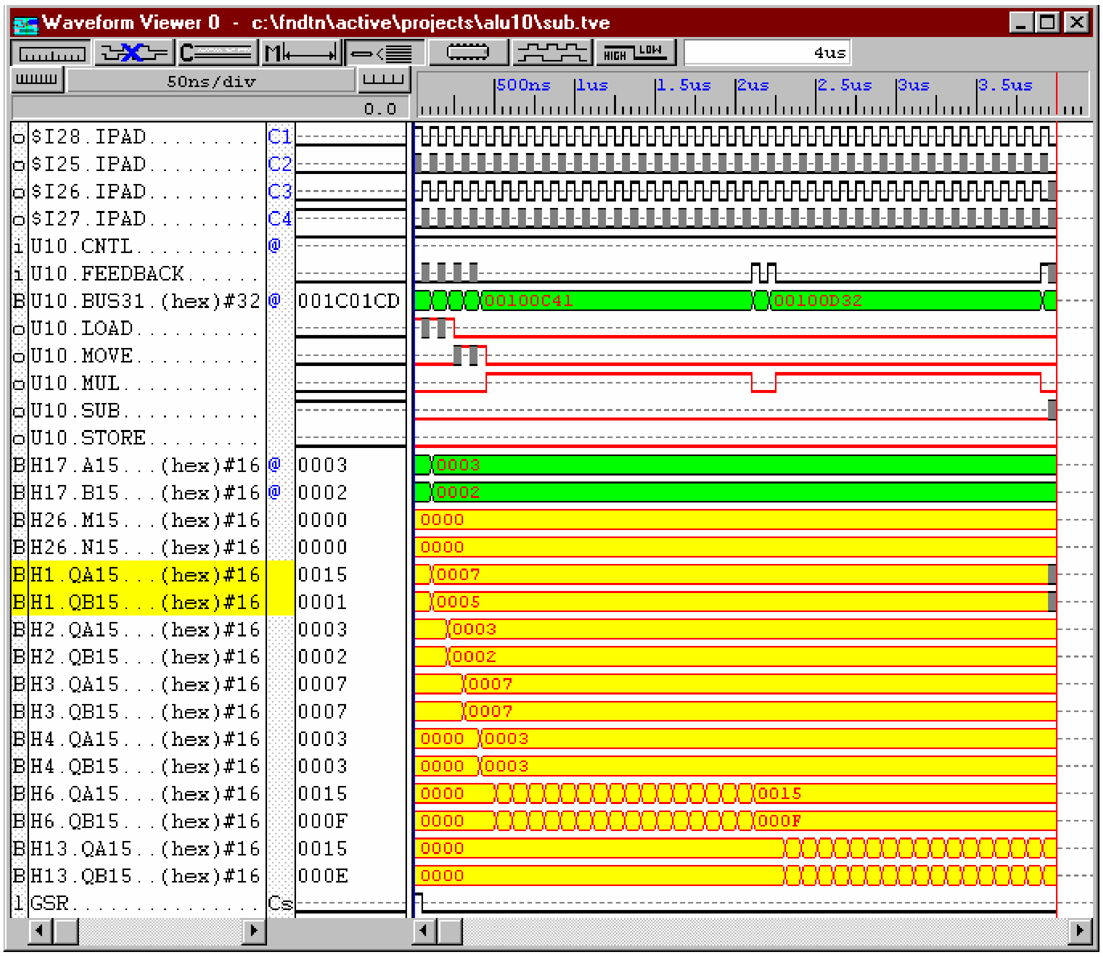
```
o $I28.IPAD........  C1
o $I25.IPAD........  C2
o $I26.IPAD........  C3
o $I27.IPAD........  C4
i U10.CNTL.........  @
i U10.FEEDBACK.....
B U10.BUS31.(hex)#32 @  001801DC
o U10.LOAD.........
o U10.MOVE.........
o U10.MUL..........
o U10.ADD..........
o U10.STORE........
B H17.A15...(hex)#16 @  0007
B H17.B15...(hex)#16 @  0005
B H26.M15...(hex)#16    0000
B H26.N15...(hex)#16    0000
B H1.QA15...(hex)#16    0015
B H1.QB15...(hex)#16    001D
B H2.QA15...(hex)#16    0007
B H2.QB15...(hex)#16    0005
B H3.QA15...(hex)#16    0003
B H3.QB15...(hex)#16    0003
B H4.QA15...(hex)#16    0007
B H4.QB15...(hex)#16    0007
B H6.QA15...(hex)#16    0015
B H6.QB15...(hex)#16    000E
B H13.QA15..(hex)#16    0015
B H13.QB15..(hex)#16    000F
I GSR..............  Cs
```

Figure 24. Add Simulation.

Before ADD operation can be carried out, MUL operations are carried out. The first one starts at 400ns and finishes at 2.1us. Then there is a NOP operation carried out to allow resetting of the multiplication counter. The second MUL operation starts at2.2us and finishes at 3.9us. Then the ADD operation is carried out over a period of one cycle. The input buses are H6.QB15, H13.QB15 (numerators) and H13.QA15 (denominator). The output is directed to H1.QA15 and H1.QB15. The whole addition operation (including the two multiplications) takes 36 clock cycles.

3.1.7 Subtract Circuit

The Subtract Circuit is part of the Simulation 5 (Appendix B.7). It is very similar to the Add Circuit, except subtraction is done not addition. This circuit also requires prior use of the Multiply Circuit to make the denominators the same for both numbers.



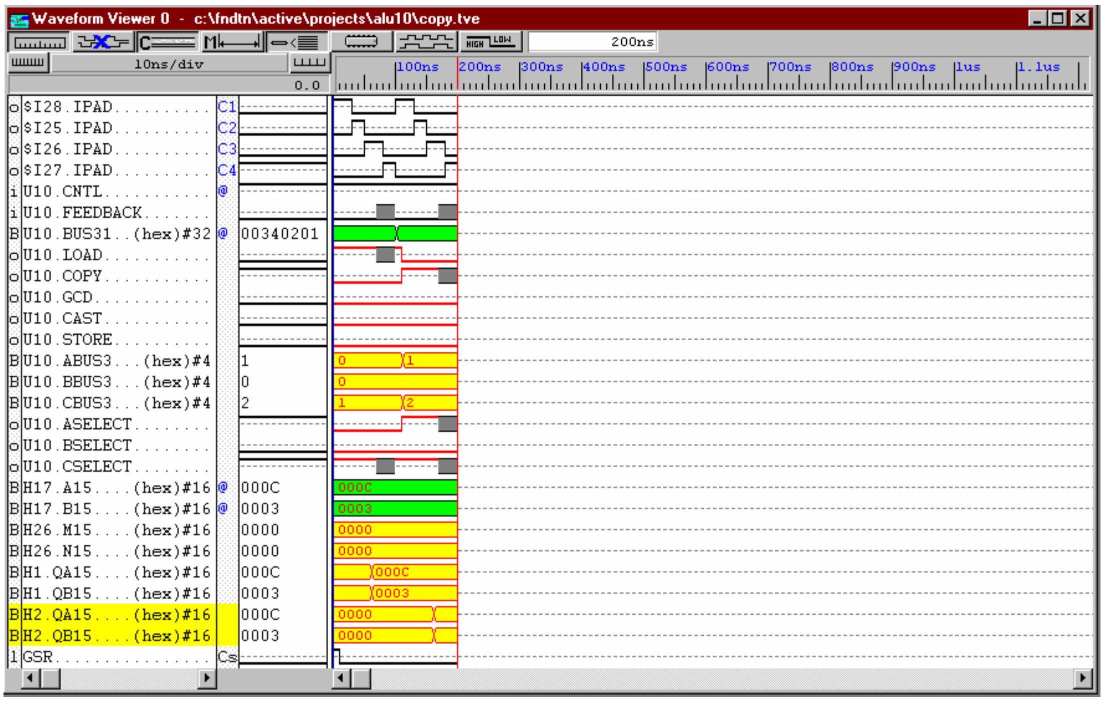
```
o $I28.IPAD........ C1
o $I25.IPAD........ C2
o $I26.IPAD........ C3
o $I27.IPAD........ C4
i U10.CNTL......... @
i U10.FEEDBACK.....
B U10.BUS31.(hex)#32 @ 001C01CD
o U10.LOAD.........
o U10.MOVE.........
o U10.MUL..........
o U10.SUB..........
o U10.STORE........
B H17.A15...(hex)#16 @ 0003
B H17.B15...(hex)#16 @ 0002
B H26.M15...(hex)#16   0000
B H26.N15...(hex)#16   0000
B H1.QA15...(hex)#16   0015
B H1.QB15...(hex)#16   0001
B H2.QA15...(hex)#16   0003
B H2.QB15...(hex)#16   0002
B H3.QA15...(hex)#16   0007
B H3.QB15...(hex)#16   0007
B H4.QA15...(hex)#16   0003
B H4.QB15...(hex)#16   0003
B H6.QA15...(hex)#16   0015
B H6.QB15...(hex)#16   000F
B H13.QA15..(hex)#16   0015
B H13.QB15..(hex)#16   000E
l GSR.............. Cs
```

Figure 25. Subtract Simulation.

This operation works on the same basis as the ADD operation and uses the same set of registers. The whole operation (including two multiplications) takes 36 clock cycles.

3.1.8 Copy Circuit

The Copy circuit is part of the Simulation 1 (Appendix B.3). It copies contents of one register into another.



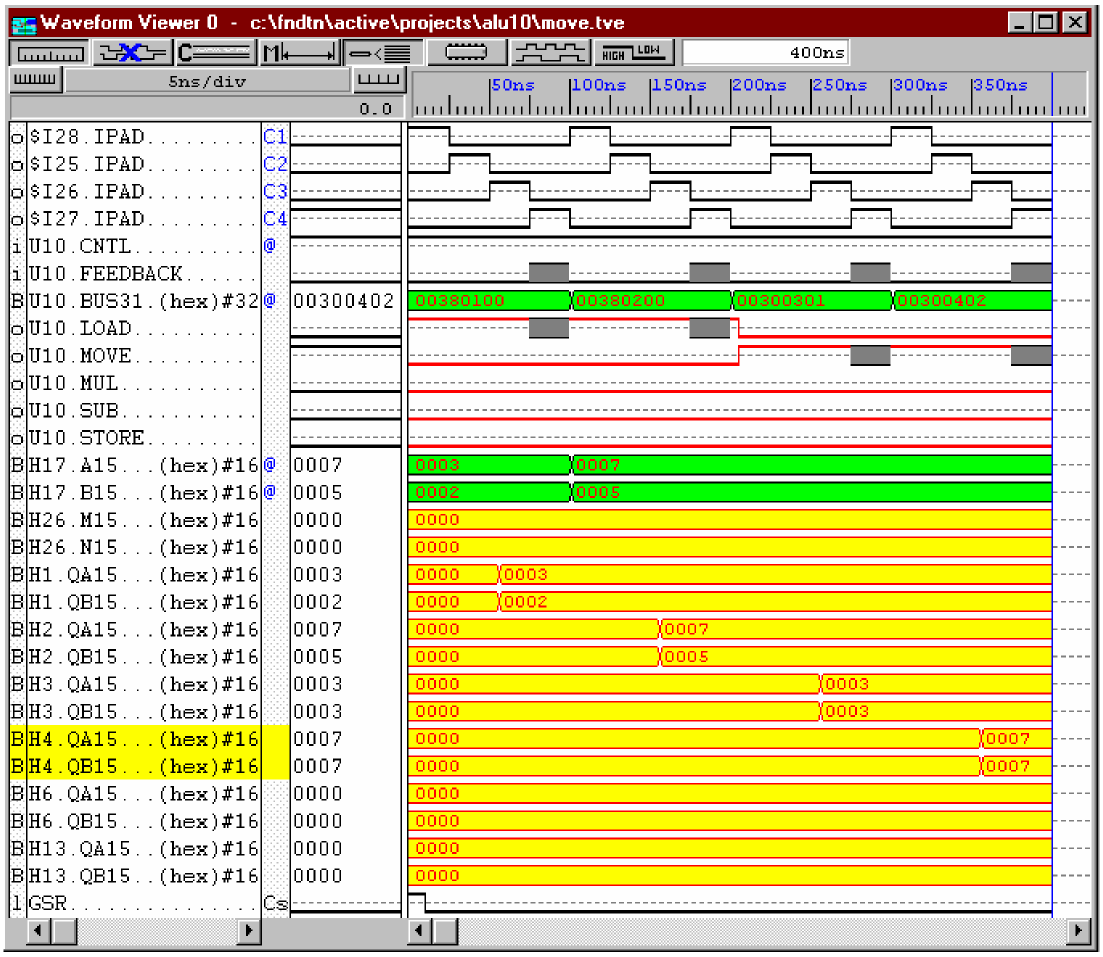
```
Waveform Viewer 0 - c:\fndtn\active\projects\alu10\copy.tve
10ns/div                200ns
0.0

o $I28.IPAD..........  C1
o $I25.IPAD..........  C2
o $I26.IPAD..........  C3
o $I27.IPAD..........  C4
i U10.CNTL...........  @
i U10.FEEDBACK.......
B U10.BUS31..(hex)#32  @  00340201
o U10.LOAD...........
o U10.COPY...........
o U10.GCD............
o U10.CAST...........
o U10.STORE..........
B U10.ABUS3...(hex)#4     1
B U10.BBUS3...(hex)#4     0
B U10.CBUS3...(hex)#4     2
o U10.ASELECT........
o U10.BSELECT........
o U10.CSELECT........
B H17.A15....(hex)#16  @  000C
B H17.B15....(hex)#16  @  0003
B H26.M15....(hex)#16     0000
B H26.N15....(hex)#16     0000
B H1.QA15....(hex)#16     000C
B H1.QB15....(hex)#16     0003
B H2.QA15....(hex)#16     000C
B H2.QB15....(hex)#16     0003
1 GSR................  Cs
```

Figure 26. Copy Simulation.

The first clock cycle is taken by the LOAD operation. The second cycle is the COPY operation. The inputs are H1.QA15 and H1.QB15. These are simply copied into another register H2. This operation takes one clock cycle. It is usually used in combination with GCD and CAST operations.

3.1.9 Move Circuit

The Move Circuit is part of Simulations 4 and 5 (Appendix B.6 and B.7). It moves the denominator from one register and puts it into the numerator and denominator of another register. This is then used in conjunction with Add Circuit to make the denominators the same for both numbers.



```
o $I28.IPAD.........  C1
o $I25.IPAD.........  C2
o $I26.IPAD.........  C3
o $I27.IPAD.........  C4
i U10.CNTL..........  @
i U10.FEEDBACK......
B U10.BUS31.(hex)#32  @  00300402   00380100  00380200  00300301  00300402
o U10.LOAD..........
o U10.MOVE..........
o U10.MUL...........
o U10.SUB...........
o U10.STORE.........
B H17.A15...(hex)#16  @  0007       0003  0007
B H17.B15...(hex)#16  @  0005       0002  0005
B H26.M15...(hex)#16     0000       0000
B H26.N15...(hex)#16     0000       0000
B H1.QA15...(hex)#16     0003       0000  0003
B H1.QB15...(hex)#16     0002       0000  0002
B H2.QA15...(hex)#16     0007       0000  0007
B H2.QB15...(hex)#16     0005       0000  0005
B H3.QA15...(hex)#16     0003       0000  0003
B H3.QB15...(hex)#16     0003       0000  0003
B H4.QA15...(hex)#16     0007       0000  0007
B H4.QB15...(hex)#16     0007       0000  0007
B H6.QA15...(hex)#16     0000       0000
B H6.QB15...(hex)#16     0000       0000
B H13.QA15..(hex)#16     0000       0000
B H13.QB15..(hex)#16     0000       0000
1 GSR...............  Cs
```

Figure 27. Move Simulation

After two LOAD operations are carried out the two MOVE operation take place. Contents of H1.QA15 are transferred to H3.QA15 and H3.QB15. In the second MOVE operation contents of H2.QA15 are transferred to H4.QA15 and H4.QB15. Each MOVE operation takes one clock cycle. The MOVE operation is used in combination with Add and Sub operations.

3.2 Register Unit

3.2.1 32-bit Register Circuit

The 32-bit Registers store data from the CAU. The registers are enabled using Control Unit (Section 3.3) and the third clock sub-cycle. All operations of the CAU use registers, plus LOAD and STORE operations.

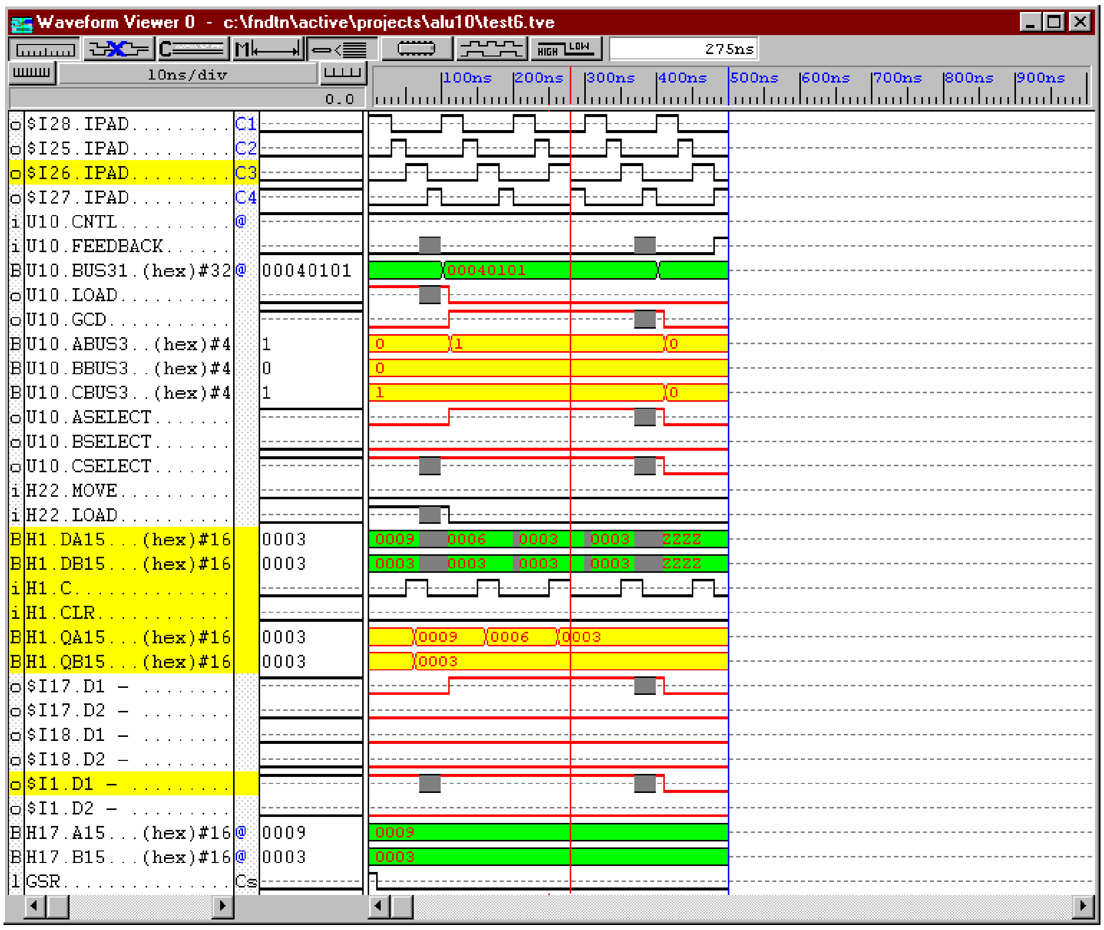


Figure 28. 32-bit Register Simulation.

This simulation represents GCD operation. The third cycle starts at 200ns. Data from the GCD Circuit is fetched to H1.DA15 and H1.DB15. The clock for the register is connected to the third sub-cycle clock and is enabled at 250ns. During that time line $I1.D1 from the Register Decoder for the **C bus** is enabled to accept any input. As the clock changes from low to high (edge trigger), the input data is stored in the register. The result of that can be seen on H1.QA15 and H1.QB15 buses. This principle applies to all other registers.

3.2.2 Register Address Decoding Circuit

Register Address Decoding Circuit allows selecting right registers for input and output. For **A** and **B Bus** the decoding enables or disables Tristate Buffers and for **C bus** the decoding enables or disables registers.

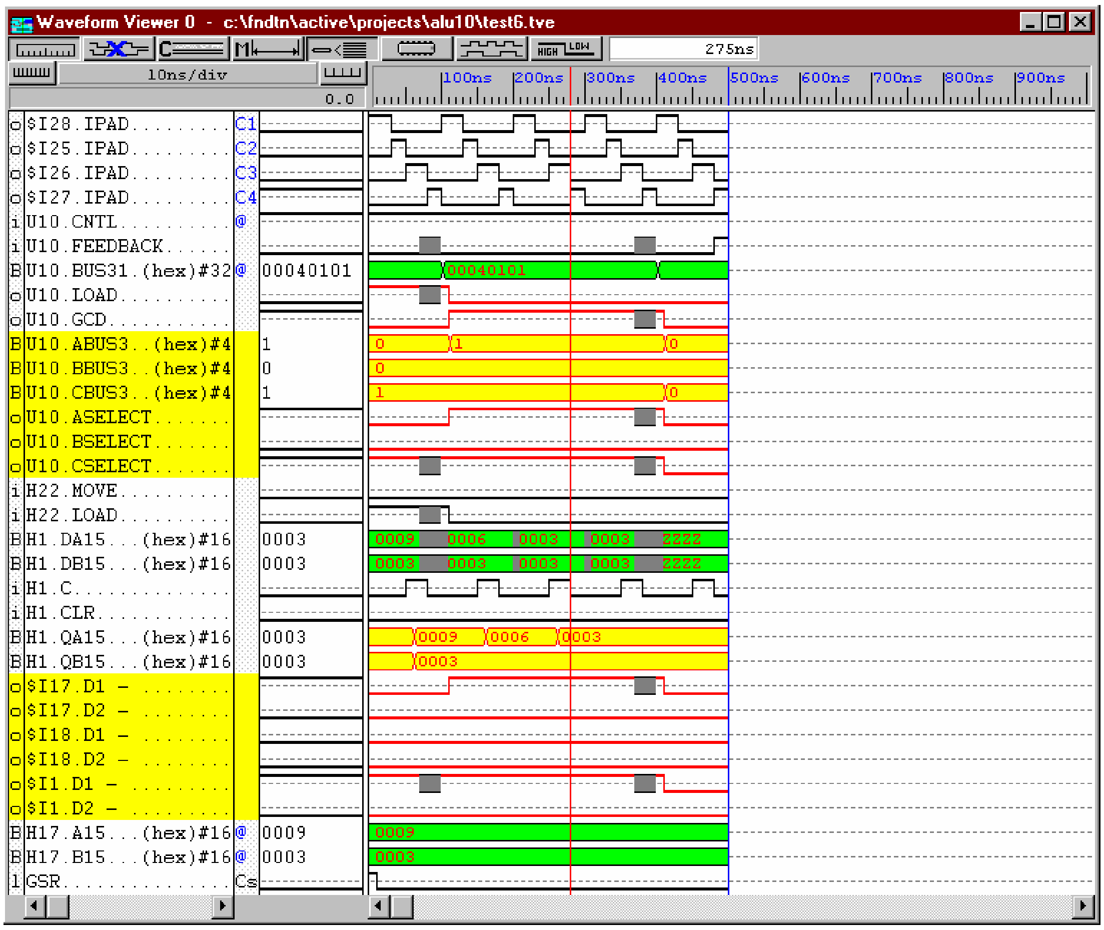


Figure 29. Register Address Decoding Simulation.

This simulation is during GCD operation. Register numbers are selected using U10.ABUS3, U10.BBUS3 and U10.CBUS3 signals. In this case ABUS and CBUS are set to 1, which means GCD will receive data from Register R1 and will store results into R1. $I17.D1 is High indicating that Tristate Buffer for **A bus** Register R1 is to be enabled. For contrast $I17.D2 denoting R2 on **A bus** is set to Low. $I18.D1 and $I18.D2 are Low because **B bus** is not used in this operation. $I1.D1 is set to High meaning that the data will be stored into R1 on **C bus**. $I1.D2 is set to Low to disable R2 for input from **C bus**.

3.3 Control Unit

3.3.1 Operations Decoding Circuit

Operations Decoding Circuit allows the Control Unit to interpret the Control Word and enable a specific operation.

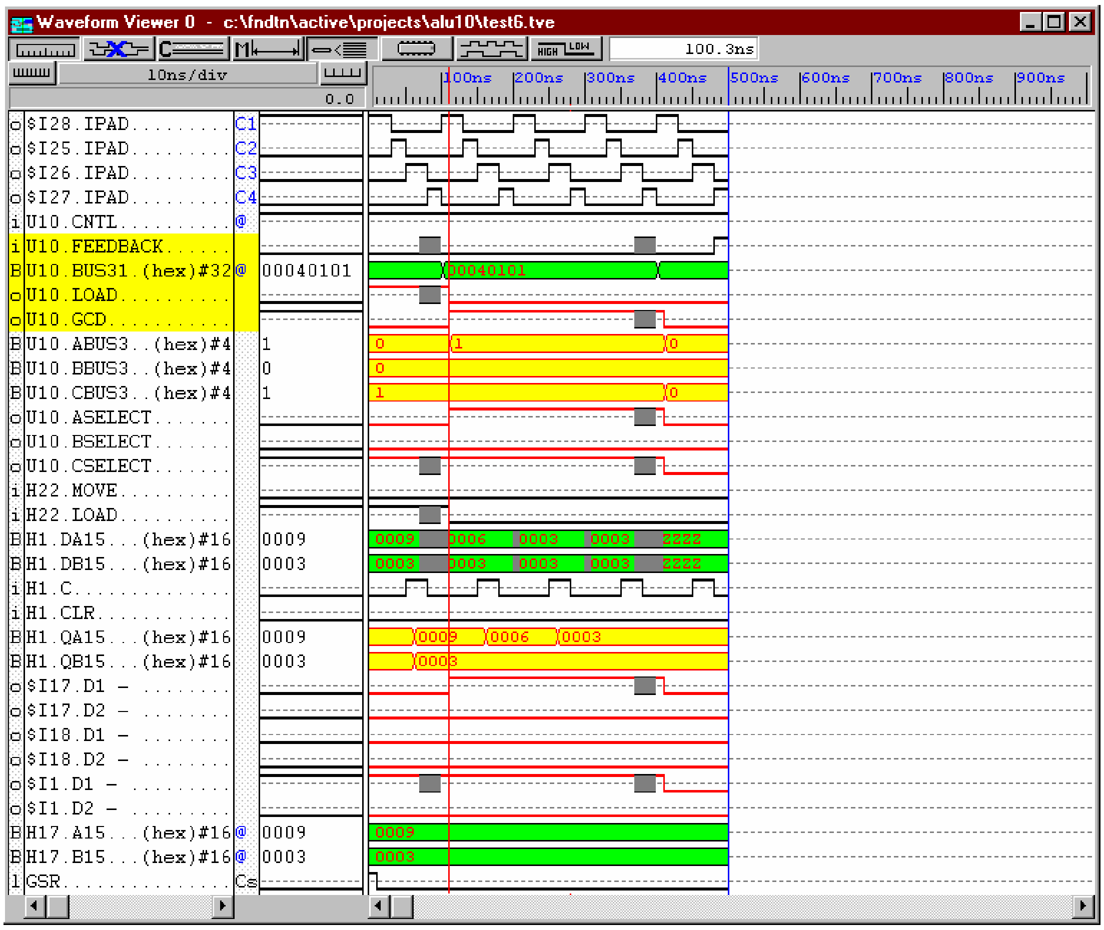
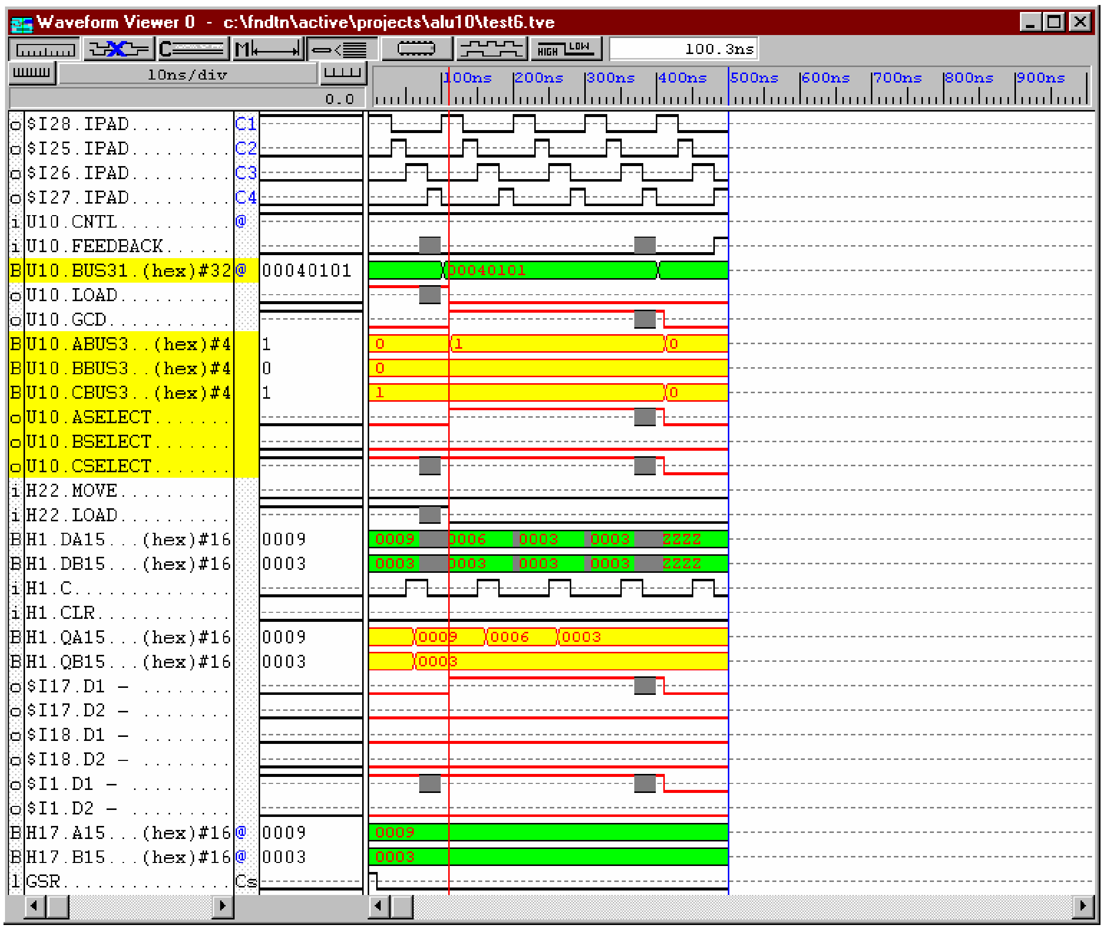


Figure 30. Operations Decoding Simulation.

This simulation is of GCD operation. The GCD operation starts at 100ns. The Control Word is 00040101. The byte 04h specifies the operation 00[0001]00, where the middle 4 bits correspond to the actual operation. When the 4 bits 0001 are put through the Operations Decoder, GCD line is enabled. Because the feedback line is Low, its inverse is ANDed with GCD line to provide the final output enabling the GCD Circuit. At that time LOAD line goes Low.

3.3.2 Register Selection Circuit

Register Selection Circuit interprets the Control Word and enables the right set of registers for the given operation.



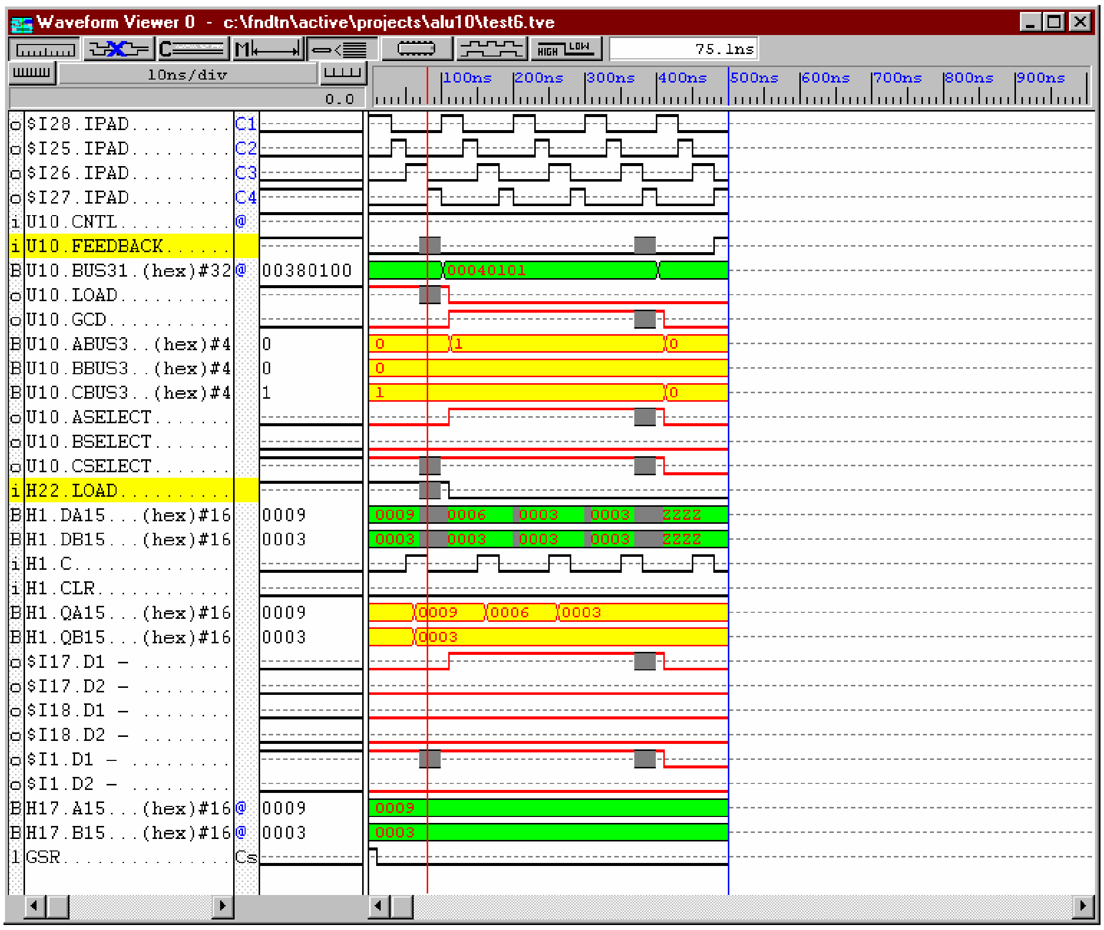
```
o $I28.IPAD.........  C1
o $I25.IPAD.........  C2
o $I26.IPAD.........  C3
o $I27.IPAD.........  C4
i U10.CNTL..........  @
i U10.FEEDBACK......
B U10.BUS31.(hex)#32  @   00040101
o U10.LOAD..........
o U10.GCD...........
B U10.ABUS3..(hex)#4      1
B U10.BBUS3..(hex)#4      0
B U10.CBUS3..(hex)#4      1
o U10.ASELECT.......
o U10.BSELECT.......
o U10.CSELECT.......
i H22.MOVE..........
i H22.LOAD..........
B H1.DA15...(hex)#16      0009
B H1.DB15...(hex)#16      0003
i H1.C..............
i H1.CLR............
B H1.QA15...(hex)#16      0009
B H1.QB15...(hex)#16      0003
o $I17.D1 - ........
o $I17.D2 - ........
o $I18.D1 - ........
o $I18.D2 - ........
o $I1.D1 - .........
o $I1.D2 - .........
B H17.A15...(hex)#16  @   0009
B H17.B15...(hex)#16  @   0003
1 GSR...............  Cs
```

Figure 31. Register Selection Simulation.

This is a simulation of GCD operation. The cycle starts at 100ns. The Control Word is 00040101. This means that opcode is 0001, the output register is R1 and input register is also R1. To set the right register number U10.ABUS and U10.CBUS are set to 1. CBUS is for register input and ABUS is for register output. U10.ASELECT and U10.CSELECT are affected by the opcode 0001. For the GCD operation they are set to High (see Section 2.3.2). The U10.BSELECT line is set to Low. These lines and buses are set to these values until the U10.FEEDBACK line goes High. Then they are reset for a new operation.

3.3.3 Feedback Unit

The Feedback Unit informs the Control Unit about completion of a specific operation. The END line from all circuits goes to the Feedback Unit, where they are processed and a single line is sent back to the Control Unit.



```
o $I28.IPAD......... C1
o $I25.IPAD......... C2
o $I26.IPAD......... C3
o $I27.IPAD......... C4
i U10.CNTL.......... @
i U10.FEEDBACK......
B U10.BUS31.(hex)#32 @  00380100
o U10.LOAD..........
o U10.GCD...........
B U10.ABUS3..(hex)#4    0
B U10.BBUS3..(hex)#4    0
B U10.CBUS3..(hex)#4    1
o U10.ASELECT.......
o U10.BSELECT.......
o U10.CSELECT.......
i H22.LOAD..........
B H1.DA15...(hex)#16    0009
B H1.DB15...(hex)#16    0003
i H1.C..............
i H1.CLR............
B H1.QA15...(hex)#16    0009
B H1.QB15...(hex)#16    0003
o $I17.D1 - ........
o $I17.D2 - ........
o $I18.D1 - ........
o $I18.D2 - ........
o $I1.D1 - .........
o $I1.D2 - .........
B H17.A15...(hex)#16 @  0009
B H17.B15...(hex)#16 @  0003
1 GSR............... Cs
```

Figure 32. Feedback Simulation.

This is a simulation of LOAD operation. During the first clock cycle (0-100ns) the Load Circuit sets its END line to High. This signal goes to H22.LOAD line. Other lines that input the Feedback Unit are set to Low but the H22.LOAD sets the Feedback line to high. The U10.FEEDBACK line, which goes into the Control Unit is a result of ANDing the Feedback line from the Feedback Unit and 4th clock sub-cycle.

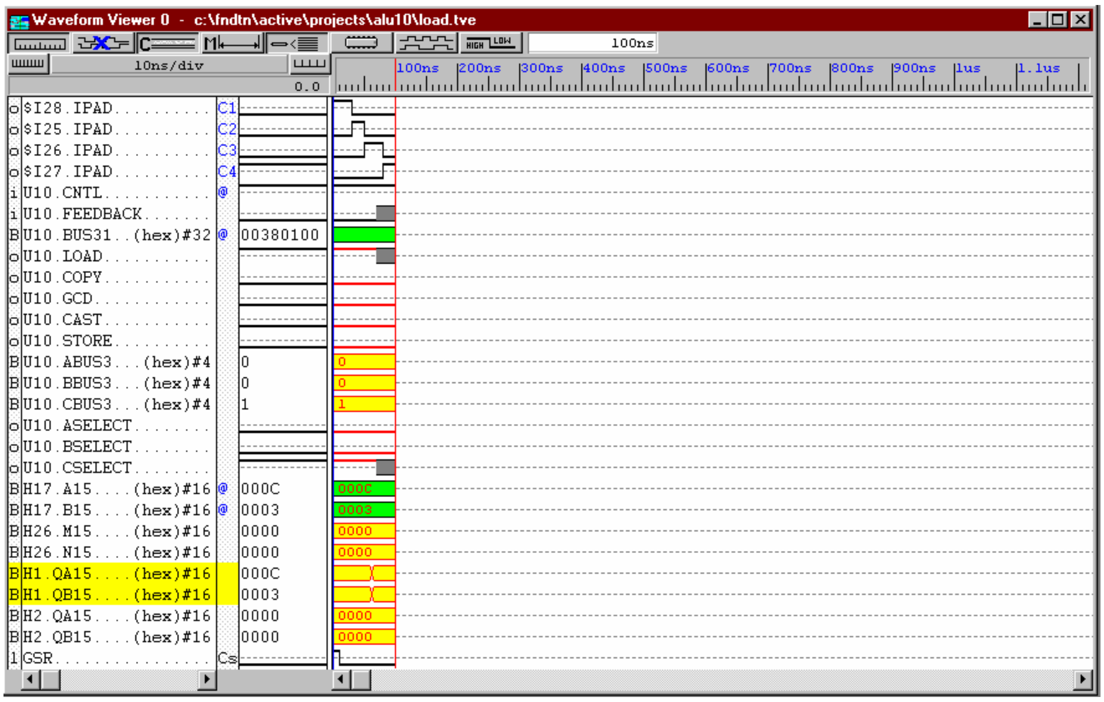
3.4 Input/Output Interface Unit

3.4.1 Parallel Port Interface

No simulations have been performed for this unit. When it became apparent that the FPGA device will not be obtained on time, main effort was put on fine tuning the arithmetic circuits. While this circuit works when tested independently, it has several timing problems when integrated with the rest of the device. Also the design of output to the parallel port wasn't accomplished. However this circuit can give some idea to future researchers on how to do the interfacing with PC. Help on this topic was very hard to obtain and this circuit may be of help.

3.4.2 Load Circuit

The Load Circuit is part of all Simulations in Appendix B but the most detail is provided in Simulation 1 (Appendix B.3). The circuit allows receiving data from the parallel port and putting it in one of the registers.



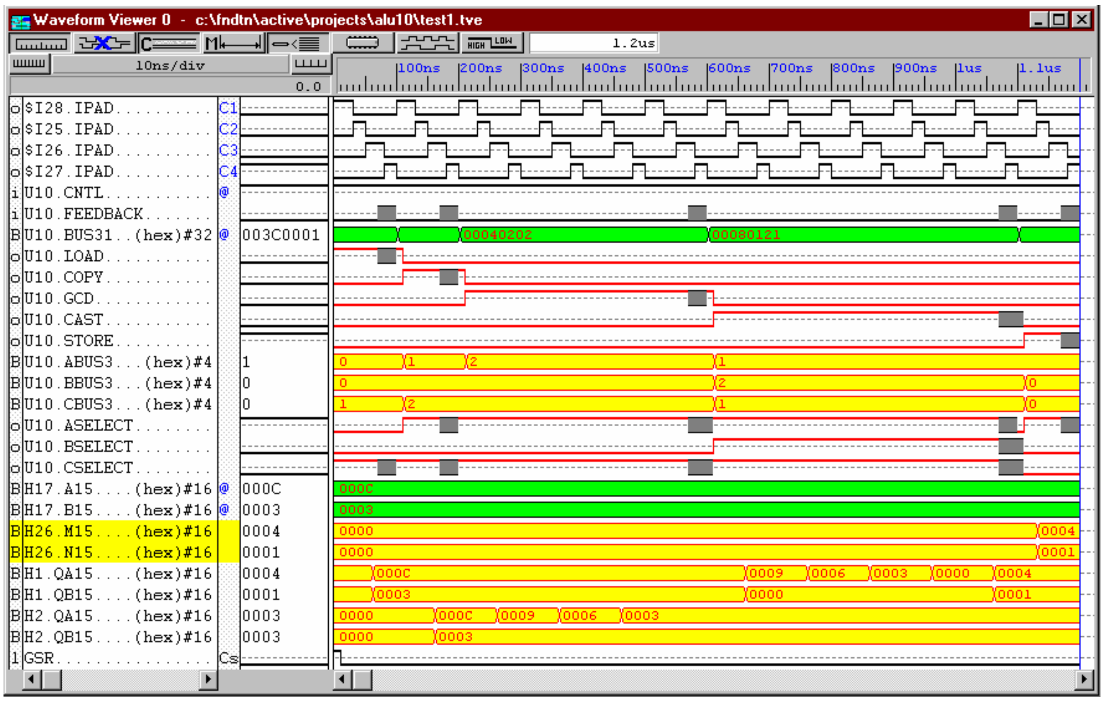
```
Waveform Viewer 0 - c:\fndtn\active\projects\alu10\load.tve
10ns/div                                    100ns
0.0          100ns 200ns 300ns 400ns 500ns 600ns 700ns 800ns 900ns 1us 1.1us
o $I28.IPAD..........      C1
o $I25.IPAD..........      C2
o $I26.IPAD..........      C3
o $I27.IPAD..........      C4
i U10.CNTL...........      @
i U10.FEEDBACK.......
B U10.BUS31..(hex)#32      @  00380100
o U10.LOAD...........
o U10.COPY...........
o U10.GCD............
o U10.CAST...........
o U10.STORE..........
B U10.ABUS3...(hex)#4         0           0
B U10.BBUS3...(hex)#4         0           0
B U10.CBUS3...(hex)#4         1           1
o U10.ASELECT........
o U10.BSELECT........
o U10.CSELECT........
B H17.A15....(hex)#16      @  000C        000C
B H17.B15....(hex)#16      @  0003        0003
B H26.M15....(hex)#16         0000        0000
B H26.N15....(hex)#16         0000        0000
B H1.QA15....(hex)#16         000C
B H1.QB15....(hex)#16         0003
B H2.QA15....(hex)#16         0000        0000
B H2.QB15....(hex)#16         0000        0000
1 GSR................      Cs
```

Figure 33. Load Simulation.

The two inputs for this operation are H17.A15 and H17.B15. Contents of the H17 inputs are put into the register H1 (H1.QA15 and H1.QB15). This operation takes one clock cycle.

3.4.3 Store Circuit

The Store Circuit is part of all Simulations in Appendix B but the most detail is provided in Simulation 1 (Appendix B.3). The circuit allows receiving data from a register and sending it to the parallel port.



```
Waveform Viewer 0 - c:\fndtn\active\projects\alu10\test1.tve
10ns/div                0.0                1.2us
o $I28.IPAD.........    C1
o $I25.IPAD.........    C2
o $I26.IPAD.........    C3
o $I27.IPAD.........    C4
i U10.CNTL..........    @
i U10.FEEDBACK......
B U10.BUS31..(hex)#32   @  003C0001
o U10.LOAD..........
o U10.COPY..........
o U10.GCD...........
o U10.CAST..........
o U10.STORE.........
B U10.ABUS3...(hex)#4      1
B U10.BBUS3...(hex)#4      0
B U10.CBUS3...(hex)#4      0
o U10.ASELECT.......
o U10.BSELECT.......
o U10.CSELECT.......
B H17.A15....(hex)#16   @  000C
B H17.B15....(hex)#16   @  0003
B H26.M15....(hex)#16      0004
B H26.N15....(hex)#16      0001
B H1.QA15....(hex)#16      0004
B H1.QB15....(hex)#16      0001
B H2.QA15....(hex)#16      0003
B H2.QB15....(hex)#16      0003
1 GSR...............    Cs
```

Figure 34. Store Simulation.

The two inputs for this operation are H1.QA15 and H1.QB15. Contents of the H1 bus are stored in a local register used by the Store Circuit and put on the **H26 bus** (H26.M15 and H26.N15). Data then becomes available to the parallel port. This operation takes one clock cycle.

4.1 Results

The results obtained suggest that the design of the arithmetic circuits is correct. The most important thing however is controlling the end of operation, especially for circuits like MUL, GCD and CAST. Biggest difficulties were with combining the Control Unit with the Interface Unit and handling of feedback loops. Also timing for the whole circuit had to be changed frequently to achieve the best sequence for the events. The difficulty with obtaining the FPGA device forced to delay the implementation significantly and in the end it could not be carried out. It also delayed the design of the Interface Unit, which works independently, but it has several timing faults that prevented it from final integration with the rest of the device. The implementation of the device could be part of another project or results of this thesis could be incorporated in a further study on exact forms.

4.2 Relevance of the Thesis

The thesis is the first type of research in this field. While the proposal for Composite Arithmetic has been published (Holmes, 1997a), work is required to implement and test the concept. The project dealt with exact forms of the Composite Arithmetic. Integer numbers are special case of rational number with denominator 1. This was done to fit integers into the scope of the project. The thesis will form a base for investigating the exact forms. The circuits completed can be easily incorporated into other designs and control process will serve as an example of how to control the CAU. A paper has also been submitted to IEEE conference (2000 IEEE SoutheastCon) in Nashville, Tennessee, USA, in hope of introducing the concept to other researchers, and getting some feedback and help on this topic. Successful submission could result in creating new projects in the field of computer arithmetic and could benefit to the University of Tasmania.

4.3 Further Research

Further Research in this field will involve exact forms, as mentioned above. This will involve separating integers from the rational numbers by addition of tags. Also development of the Long Accumulator will be necessary, as it is an important part of the overall concept. The next step would be development of the inexact forms. These require different circuits from the exact forms and different notation. Once both forms are done they have to be integrated into the CAU. Significant changes may need to be made to both forms to put them together. Also a device to automatically select the right form will be needed and this will be getting feedback from the exact part of the CAU. Once the CAU is designed memory storage and display issues will need to be solved. This will require memory management device and software driver that will transform the data from register form into appropriate display form. Only then the system will be fully capable of performing Composite Arithmetic operations.

References

Holmes, N. “Composite Arithmetic: Proposal for a New Standard.” IEEE Computer, 1997a, 65-73

Holmes, N. “Floating Point and Composite Arithmetic.” Proceedings, 8th Biennial Computational Techniques and Applications Conference, 1997b

Hwang, K., 1979, Computer Arithmetic: Principles, architecture and design, John Wiley & Sons

Kulisch, U.W. Advanced Arithmetic for the Digital Computer – Design of Arithmetic Units, Version 2, 1999

Tanenbaum, A.S., 1990, Structured Computer Organisation, 3rd Edition, Prentice Hall, New Jersey

Villasenor, J., Mangione-Smith, W.H. “Configurable Computing.” Scientific American, 1997, URL – http://www.sciam.com:80/0697issue/0697villasenor.html

Appendix A: Simple Composite Arithmetic Machine

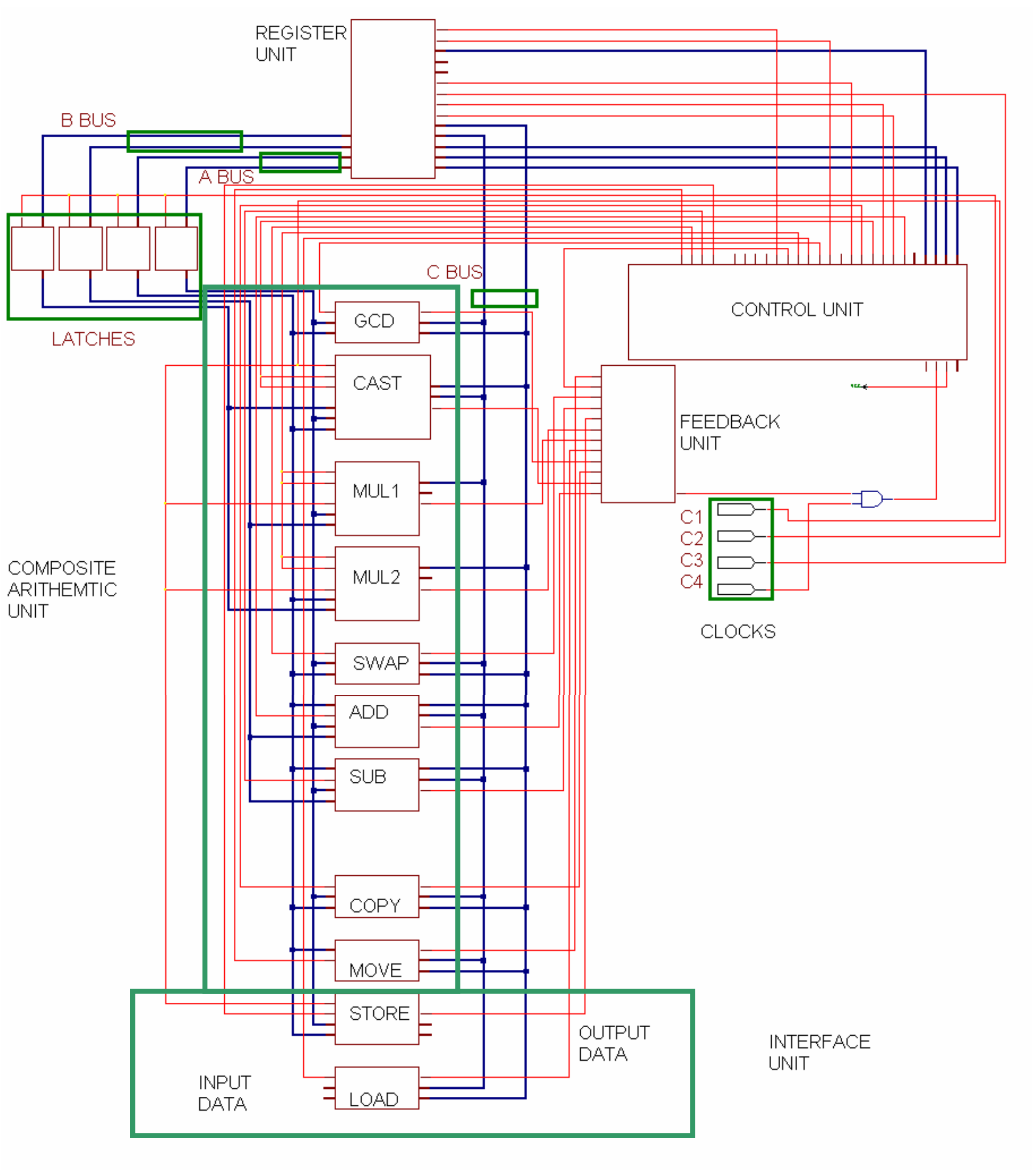


Figure 35. SCAM Diagram showing main components.

Main Components:
Composite Arithmetic Unit
Control Unit
Register Unit
Feedback Unit
Interface Unit
Other Components:
Bus Latches
System Clocks

Appendix B: Simulations

1. CONTROL WORD

Control Word provides 32-bit data to the SCAM device about the next instruction to be executed. The higher 16 bits represent the opcode of the operation to be carried out. In fact higher 8 bits are always 0 and only lower 8 bits are used. The lower 16 bits of the control word tell which register is to be used. Starting from the highest nibble down, the first nibble is always zero, the second nibble denotes the C Bus, then third and fourth is for B Bus and A Bus respectively.

Operation	**OPCODE**	**Registers**	**Comment**
NOP	0000	0000	
GCD	0004	0X0X	Get contents of X and put it back into X
CAST	0008	0XYX	Get contents of X and Y and put it back into X
SWAP	000C	0X0X	Get contents of X and put it back into X
MUL	0010	0XYZ	Multiply Z by Y and put result into X
DIV			Done using SWAP and MUL operations
ADD	0018	0XYZ	Add Z to Y and put result into X
SUB	001C	0XYZ	Subtract Z from Y and put result into X
MOVE	0030	0X0Y	Get contents of Y and put it back into X
COPY	0034	0X0Y	Get contents of Y and put it back into X
LOAD	0038	0X00	Put data into X
STORE	003C	000X	Get contents of X

2. CLOCK SIMULATION

SEQUENCE: Nil
INPUTS: Nil
OUTPUTS:
$I28.IPAD – Clock 1
$I25.IPAD – Clock 2
$I26.IPAD – Clock 3
$I27.IPAD – Clock 4
TEST DATA IN: Nil
TEST DATA OUT: Nil

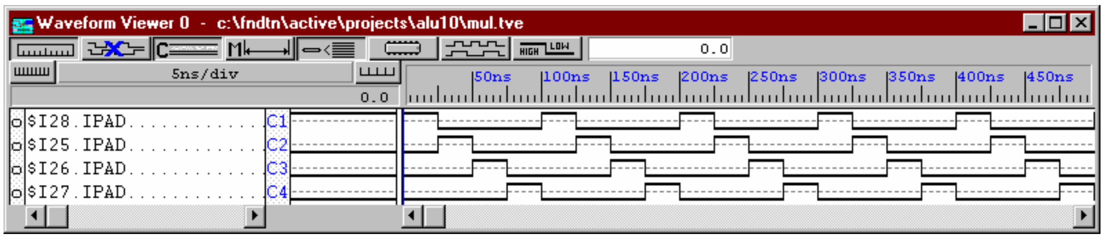


Figure 36. Clock Simulation.

Clock Timings:
Clock 1 – 25ns High, 75ns Low
Clock 2 – 25ns Low, 25ns High, 50ns Low
Clock 3 – 50ns Low, 25ns High, 25ns Low
Clock 4 – 75ns Low, 25ns High
Clock Functions:
Clock 1 – Operates the latches that latch outputs from the Register Unit.
Clock 2 – Provides clock cycle for the Composite Arithmetic Unit.
Clock 3 – Provides clock cycle for the Register Unit.
Clock 4 – Operates the Feedback Unit.

3. SIMULATION ONE – GCD AND CAST

SEQUENCE: Load R1, Copy R1 to R2, GCD R2, Cast R1 and R2 to R1, Store R1

INPUTS:

U10.CNTL – Control Signal from I/O Interface
U10.FEEDBACK – Feedback signal from the Feedback Unit
U10.BUS31 – Control Word
H17.A15 – Denominator of the input number
H17.B15 – Numerator if the input number

OUTPUTS:

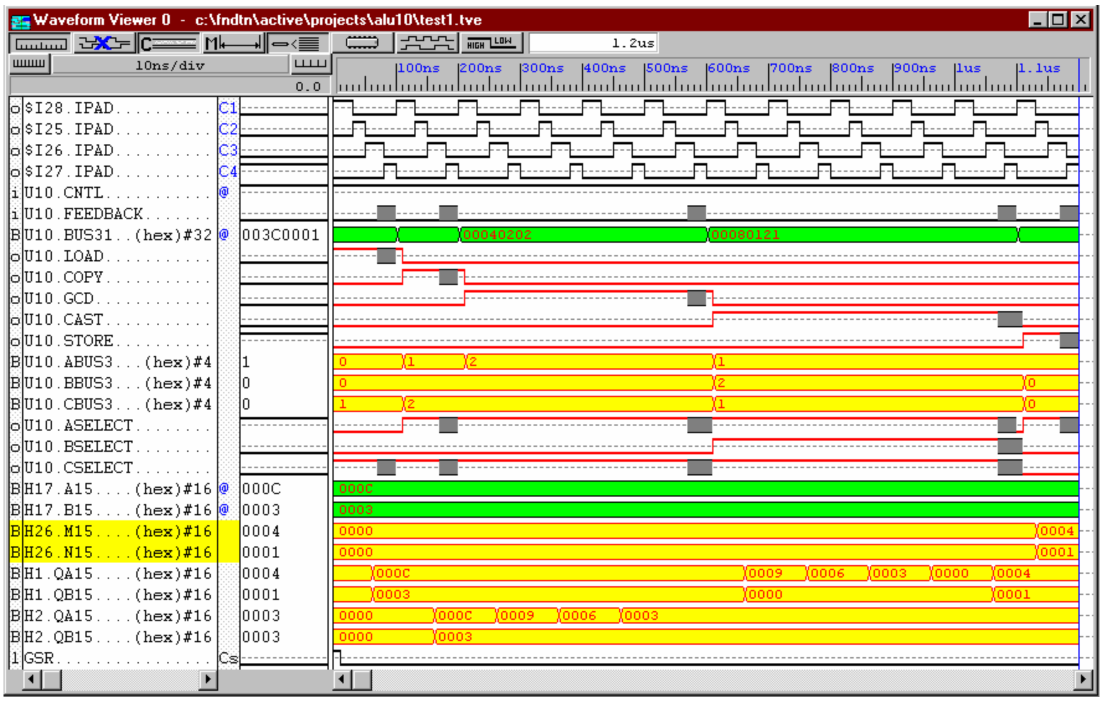
$I28.IPAD – Clock 1
$I25.IPAD – Clock 2
$I26.IPAD – Clock 3
$I27.IPAD – Clock 4
U10.LOAD – Line enabling Load operation
U10.COPY – Line enabling Copy operation
U10.GCD – Line enabling GCD operation
U10.CAST – Line enabling Cast operation
U10.STORE – Line enabling Store operation
U10.ABUS3 – Register number on A bus
U10.BBUS3 – Register number on B bus
U10.CBUS3 – Register number on C bus
U10.ASELECT – Select line for A bus
U10.BESLECT – Select line for B bus
U10.CSELECT – Select line for C bus
H26.M15 – Denominator of the output number
H26.N15 – Numerator of the output number
H1.QA15 – Denominator output of the Register R1
H1.QB15 –Numerator output of the Register R1
H2.QA15 – Denominator output of the Register R2
H2.QB15 – Numerator output of the Register R2

TEST DATA IN:

Rational Number 3/12

TEST DATA OUT:

Rational Number: 1/4

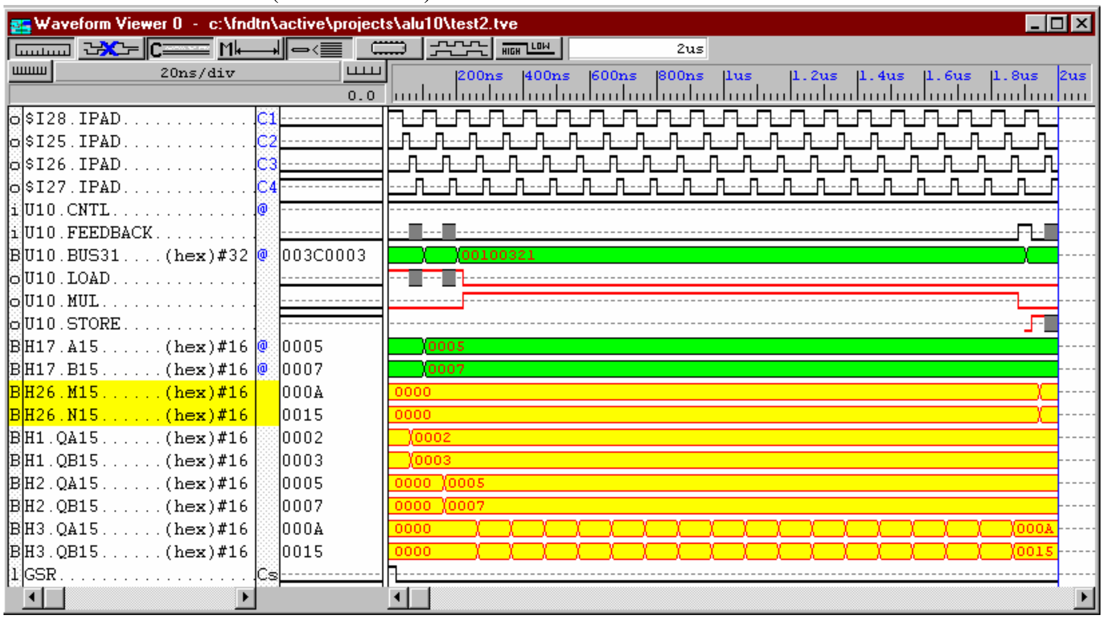


```
Waveform Viewer 0 - c:\fndtn\active\projects\alu10\test1.tve
10ns/div                 1.2us
0.0

o $I28.IPAD..........  C1
o $I25.IPAD..........  C2
o $I26.IPAD..........  C3
o $I27.IPAD..........  C4
i U10.CNTL...........  @
i U10.FEEDBACK.......
B U10.BUS31..(hex)#32  @  003C0001
o U10.LOAD...........
o U10.COPY...........
o U10.GCD............
o U10.CAST...........
o U10.STORE..........
B U10.ABUS3...(hex)#4     1
B U10.BBUS3...(hex)#4     0
B U10.CBUS3...(hex)#4     0
o U10.ASELECT........
o U10.BSELECT........
o U10.CSELECT........
B H17.A15....(hex)#16  @  000C
B H17.B15....(hex)#16  @  0003
B H26.M15....(hex)#16     0004
B H26.N15....(hex)#16     0001
B H1.QA15....(hex)#16     0004
B H1.QB15....(hex)#16     0001
B H2.QA15....(hex)#16     0003
B H2.QB15....(hex)#16     0003
1 GSR................  Cs
```

Figure 37. Simulation One.

4. SIMULATION TWO – MUL

SEQUENCE: Load R1, Load R2, Mul R1 and R2 to R3, Store R3
INPUTS:
U10.CNTL – Control Signal from I/O Interface
U10.FEEDBACK – Feedback signal from the Feedback Unit
U10.BUS31 – Control Word
H17.A15 – Denominator of the input number
H17.B15 – Numerator if the input number
OUTPUTS:
$I28.IPAD – Clock 1
$I25.IPAD – Clock 2
$I26.IPAD – Clock 3
$I27.IPAD – Clock 4
U10.LOAD – Line enabling Load operation
U10.MUL – Line enabling Multiply operation
U10.STORE – Line enabling Store operation
H26.M15 – Denominator of the output number
H26.N15 – Numerator of the output number
H1.QA15 – Denominator output of the Register R1
H1.QB15 –Numerator output of the Register R1
H2.QA15 – Denominator output of the Register R2
H2.QB15 – Numerator output of the Register R2
H3.QA15 – Denominator output of the Register R3
H3.QB15 – Numerator output of the Register R3
TEST DATA IN:
Rational Number: 3/2
Rational Number: 7/5
TEST DATA OUT:
Rational Number: 21/10 (15/0A in Hex)



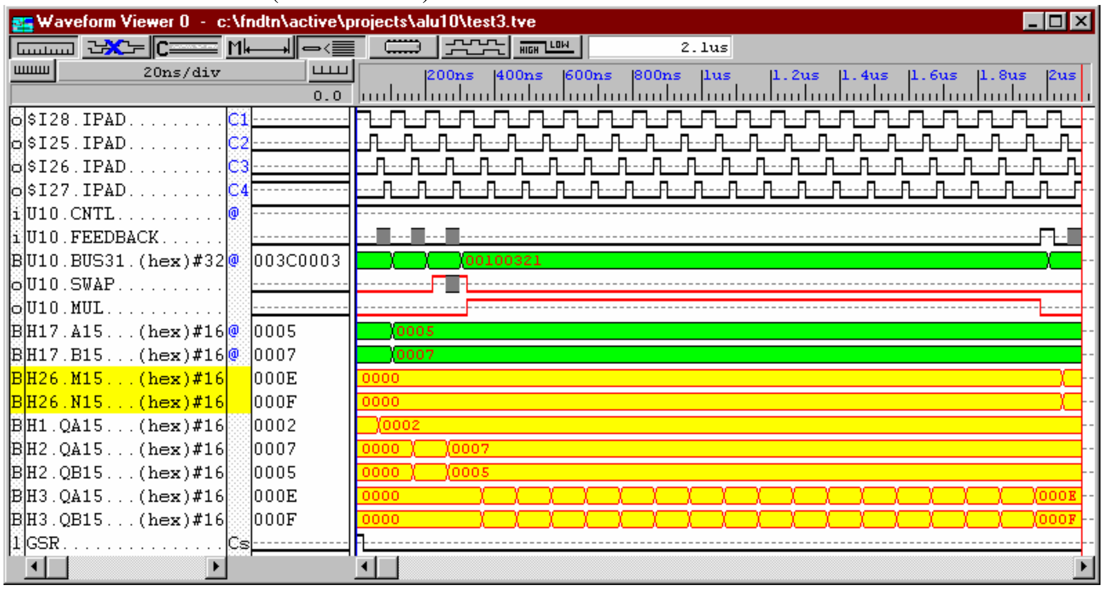
Waveform Viewer 0 - c:\fndtn\active\projects\alu10\test2.tve

```
20ns/div                         0.0        2us
o $I28.IPAD............ C1
o $I25.IPAD............ C2
o $I26.IPAD............ C3
o $I27.IPAD............ C4
i U10.CNTL............. @
i U10.FEEDBACK.........
B U10.BUS31....(hex)#32 @  003C0003
o U10.LOAD.............
o U10.MUL..............
o U10.STORE............
B H17.A15......(hex)#16 @  0005
B H17.B15......(hex)#16 @  0007
B H26.M15......(hex)#16    000A
B H26.N15......(hex)#16    0015
B H1.QA15......(hex)#16    0002
B H1.QB15......(hex)#16    0003
B H2.QA15......(hex)#16    0005
B H2.QB15......(hex)#16    0007
B H3.QA15......(hex)#16    000A
B H3.QB15......(hex)#16    0015
1 GSR.................. Cs
```

Figure 38. Simulation Two.

5. SIMULATION THREE – DIV

SEQUENCE: Load R1, Load R2, Swap R2, Mul R1 and R2 to R3, Store R3
INPUTS:
U10.CNTL – Control Signal from I/O Interface
U10.FEEDBACK – Feedback signal from the Feedback Unit
U10.BUS31 – Control Word
H17.A15 – Denominator of the input number
H17.B15 – Numerator if the input number
OUTPUTS:
$I28.IPAD – Clock 1
$I25.IPAD – Clock 2
$I26.IPAD – Clock 3
$I27.IPAD – Clock 4
U10.SWAP – Line enabling Swap operation
U10.MUL – Line enabling Multiply operation
H26.M15 – Denominator of the output number
H26.N15 – Numerator of the output number
H1.QA15 – Denominator output of the Register R1
H1.QB15 –Numerator output of the Register R1
H2.QA15 – Denominator output of the Register R2
H2.QB15 – Numerator output of the Register R2
H3.QA15 – Denominator output of the Register R3
H3.QB15 – Numerator output of the Register R3
TEST DATA IN:
Rational Number: 3/2
Rational Number: 7/5
TEST DATA OUT:
Rational Number: 15/14 (0F/0E in Hex)



```
o $I28.IPAD........     C1
o $I25.IPAD........     C2
o $I26.IPAD........     C3
o $I27.IPAD........     C4
i U10.CNTL.........     @
i U10.FEEDBACK.....
B U10.BUS31.(hex)#32 @  003C0003   00100321
o U10.SWAP.........
o U10.MUL..........
B H17.A15...(hex)#16 @  0005       0005
B H17.B15...(hex)#16 @  0007       0007
B H26.M15...(hex)#16    000E       0000
B H26.N15...(hex)#16    000F       0000
B H1.QA15...(hex)#16    0002       0002
B H2.QA15...(hex)#16    0007       0000  0007
B H2.QB15...(hex)#16    0005       0000  0005
B H3.QA15...(hex)#16    000E       0000  000E
B H3.QB15...(hex)#16    000F       0000  000F
1 GSR..............     Cs
```

Figure 39. Simulation Three.

6. SIMULATION FOUR – ADD

SEQUENCE: Load R1, Load R2, Move R1 to R3, Move R2 to R4, Mul R1 and R4 to R12, Mul R2 and R3 to R13, Add R12 and R13 to R1, Store R1
INPUTS:
U10.CNTL – Control Signal from I/O Interface
U10.FEEDBACK – Feedback signal from the Feedback Unit
U10.BUS31 – Control Word
H17.A15 – Denominator of the input number
H17.B15 – Numerator if the input number
OUTPUTS:
$I28.IPAD – Clock 1
$I25.IPAD – Clock 2
$I26.IPAD – Clock 3
$I27.IPAD – Clock 4
U10.LOAD – Line enabling Load operation
U10.MOVE – Line enabling Move operation
U10.MUL – Line enabling Multiply operation
U10.ADD – Line enabling Add operation
U10.STORE – Line enabling Store operation
H26.M15 – Denominator of the output number
H26.N15 – Numerator of the output number
H1.QA15 – Denominator output of the Register R1
H1.QB15 –Numerator output of the Register R1
H2.QA15 – Denominator output of the Register R2
H2.QB15 – Numerator output of the Register R2
H3.QA15 – Denominator output of the Register R3
H3.QB15 – Numerator output of the Register R3
H4.QA15 – Denominator output of the Register R4
H4.QB15 – Numerator output of the Register R4
H6.QA15 – Denominator output of the Register R12
H6.QB15 – Numerator output of the Register R12
H13.QA15 – Denominator output of the Register R13
H13.QB15 – Numerator output of the Register R13
TEST DATA IN:
Rational Number: 2/3
Rational Number: 5/7
TEST DATA OUT:
Rational Number: 29/21 (1D/15 in Hex)

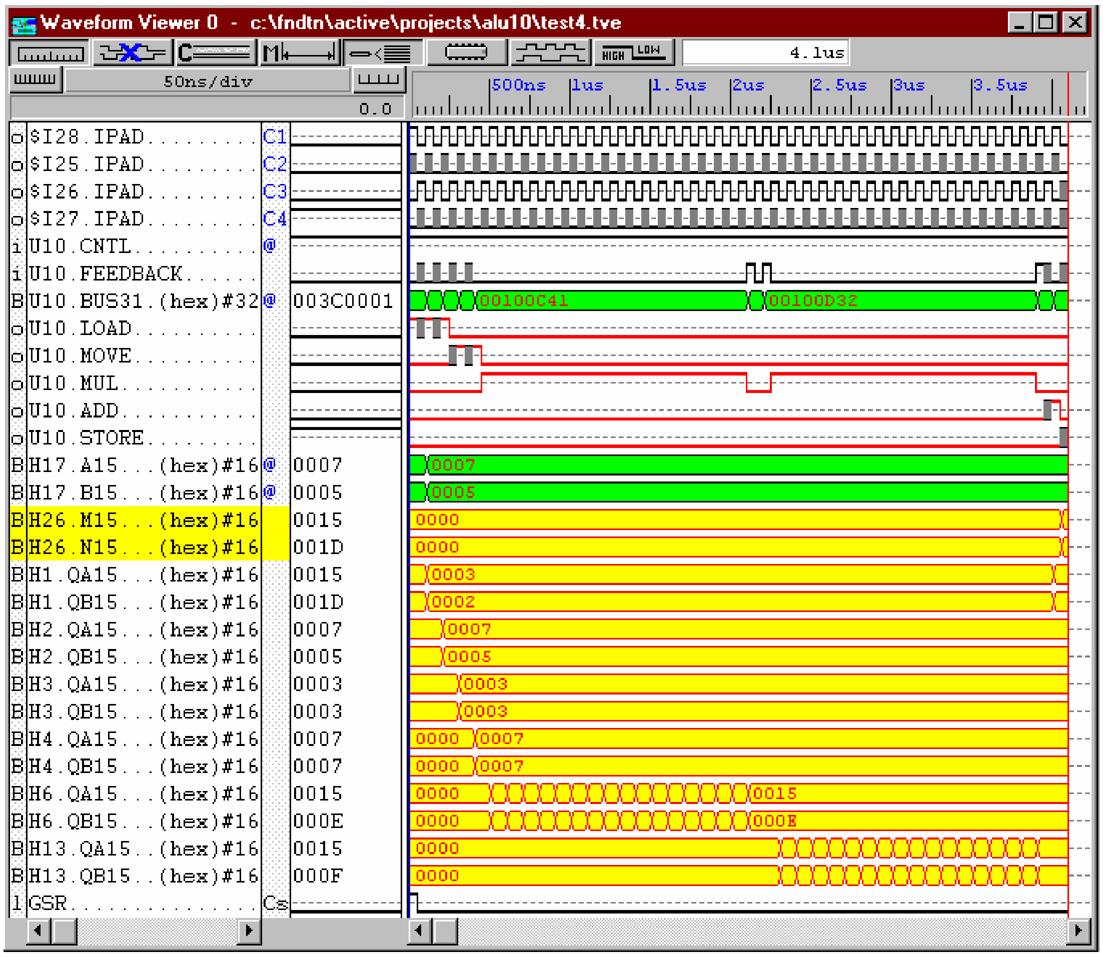

```
Waveform Viewer 0  -  c:\fndtn\active\projects\alu10\test4.tve
50ns/div                                  4.1us
                                    0.0
o $I28.IPAD.........C1
o $I25.IPAD.........C2
o $I26.IPAD.........C3
o $I27.IPAD.........C4
i U10.CNTL..........@
i U10.FEEDBACK......
B U10.BUS31.(hex)#32@   003C0001
o U10.LOAD..........
o U10.MOVE..........
o U10.MUL...........
o U10.ADD...........
o U10.STORE.........
B H17.A15...(hex)#16@   0007
B H17.B15...(hex)#16@   0005
B H26.M15...(hex)#16    0015
B H26.N15...(hex)#16    001D
B H1.QA15...(hex)#16    0015
B H1.QB15...(hex)#16    001D
B H2.QA15...(hex)#16    0007
B H2.QB15...(hex)#16    0005
B H3.QA15...(hex)#16    0003
B H3.QB15...(hex)#16    0003
B H4.QA15...(hex)#16    0007
B H4.QB15...(hex)#16    0007
B H6.QA15...(hex)#16    0015
B H6.QB15...(hex)#16    000E
B H13.QA15..(hex)#16    0015
B H13.QB15..(hex)#16    000F
1 GSR...............Cs
```

Figure 40. Simulation Four.

7. SIMULATION FIVE – SUB

SEQUENCE: Load R1, Load R2, Move R1 to R3, Move R2 to R4, Mul R1 and R4 to R12, Mul R2 and R3 to R13, Sub R13 from R12 to R1, Store R1

INPUTS:
U10.CNTL – Control Signal from I/O Interface
U10.FEEDBACK – Feedback signal from the Feedback Unit
U10.BUS31 – Control Word
H17.A15 – Denominator of the input number
H17.B15 – Numerator if the input number

OUTPUTS:
$I28.IPAD – Clock 1
$I25.IPAD – Clock 2
$I26.IPAD – Clock 3
$I27.IPAD – Clock 4
U10.LOAD – Line enabling Load operation
U10.MOVE – Line enabling Move operation
U10.MUL – Line enabling Multiply operation
U10.SUB – Line enabling Sub operation
U10.STORE – Line enabling Store operation
H26.M15 – Denominator of the output number
H26.N15 – Numerator of the output number
H1.QA15 – Denominator output of the Register R1
H1.QB15 –Numerator output of the Register R1
H2.QA15 – Denominator output of the Register R2
H2.QB15 – Numerator output of the Register R2
H3.QA15 – Denominator output of the Register R3
H3.QB15 – Numerator output of the Register R3
H4.QA15 – Denominator output of the Register R4
H4.QB15 – Numerator output of the Register R4
H6.QA15 – Denominator output of the Register R12
H6.QB15 – Numerator output of the Register R12
H13.QA15 – Denominator output of the Register R13
H13.QB15 – Numerator output of the Register R13

TEST DATA IN:
Rational Number: 7/5
Rational Number: 3/2

TEST DATA OUT:
Rational Number: 1/21 (1/15 in Hex)

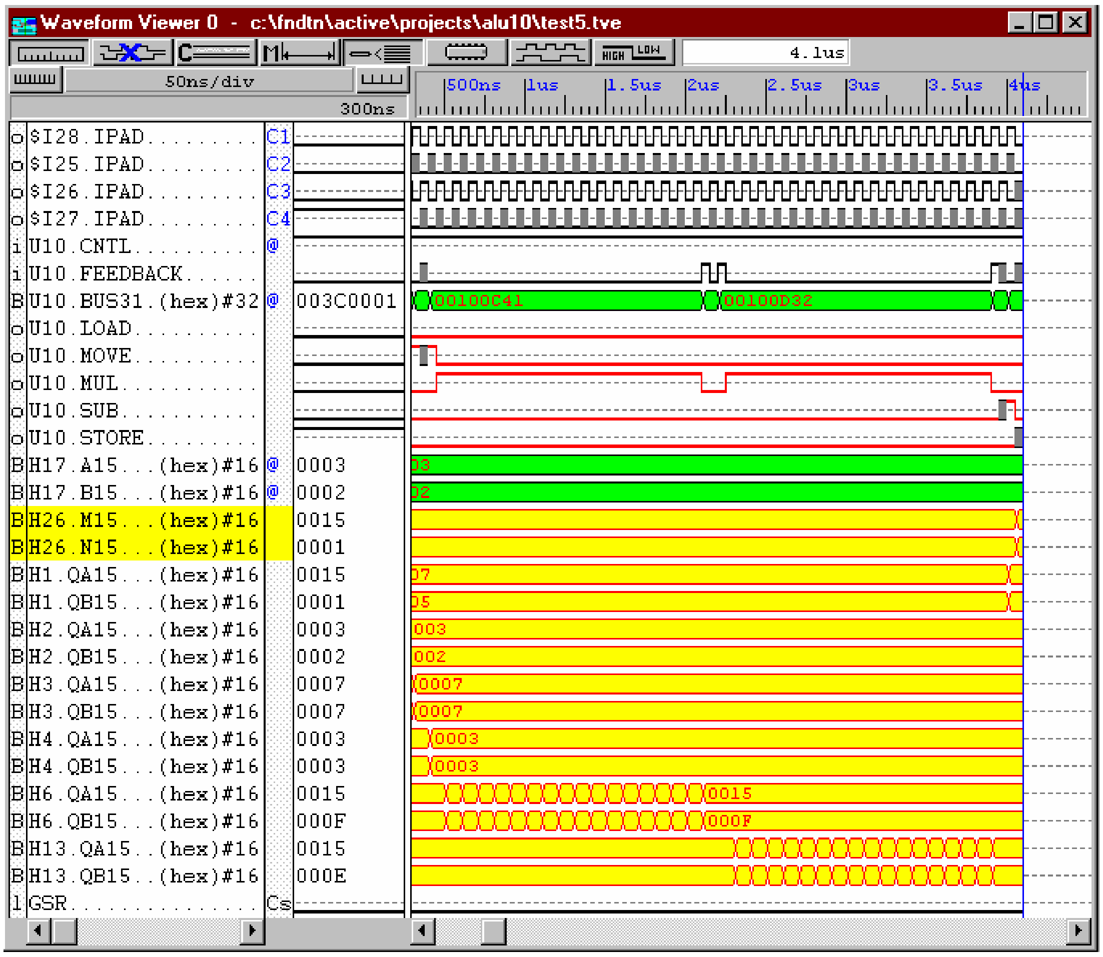
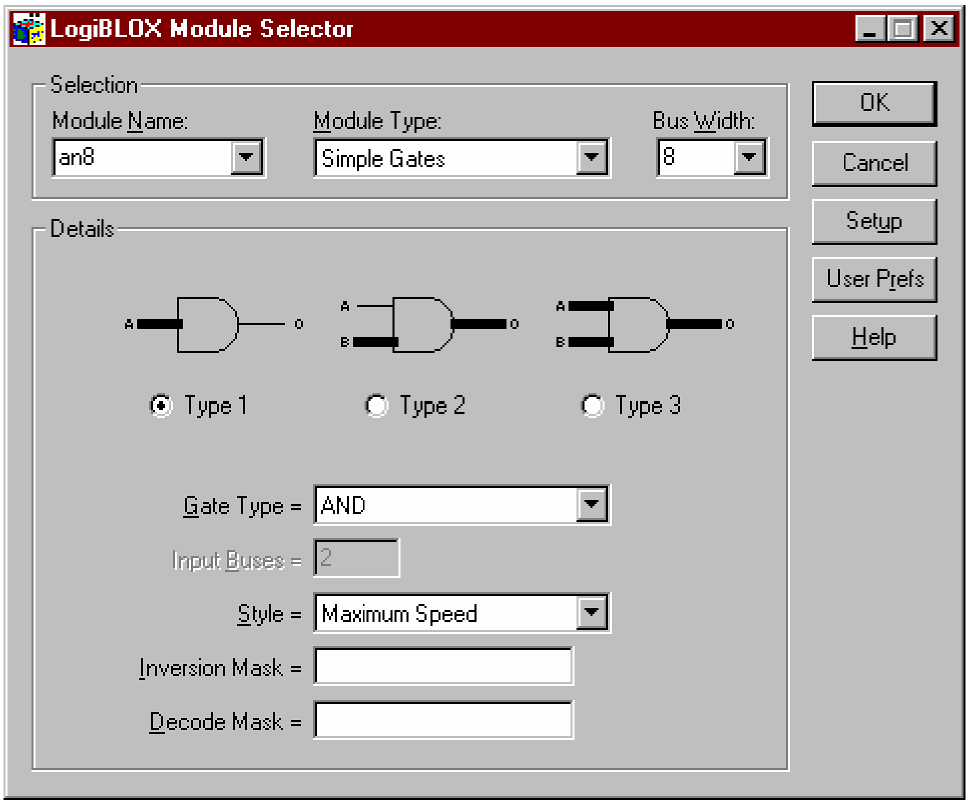

```
Waveform Viewer 0  -  c:\fndtn\active\projects\alu10\test5.tve
                                        4.1us
50ns/div
                                        300ns

o $I28.IPAD.........  C1
o $I25.IPAD.........  C2
o $I26.IPAD.........  C3
o $I27.IPAD.........  C4
i U10.CNTL..........  @
i U10.FEEDBACK......
B U10.BUS31.(hex)#32  @  003C0001
o U10.LOAD..........
o U10.MOVE..........
o U10.MUL...........
o U10.SUB...........
o U10.STORE.........
B H17.A15...(hex)#16  @  0003
B H17.B15...(hex)#16  @  0002
B H26.M15...(hex)#16     0015
B H26.N15...(hex)#16     0001
B H1.QA15...(hex)#16     0015
B H1.QB15...(hex)#16     0001
B H2.QA15...(hex)#16     0003
B H2.QB15...(hex)#16     0002
B H3.QA15...(hex)#16     0007
B H3.QB15...(hex)#16     0007
B H4.QA15...(hex)#16     0003
B H4.QB15...(hex)#16     0003
B H6.QA15...(hex)#16     0015
B H6.QB15...(hex)#16     000F
B H13.QA15..(hex)#16     0015
B H13.QB15..(hex)#16     000E
1 GSR...............  Cs
```

Figure 41. Simulation Five.

Appendix C: LogiBLOX Modules

1. AN8

Function: Provides Logical AND operation of 8-bit input.
Type: Simple Gates (Type 1)
Gate Type: AND
Bus width: 8 bits


LogiBLOX Module Selector

Selection
Module Name: an8
Module Type: Simple Gates
Bus Width: 8

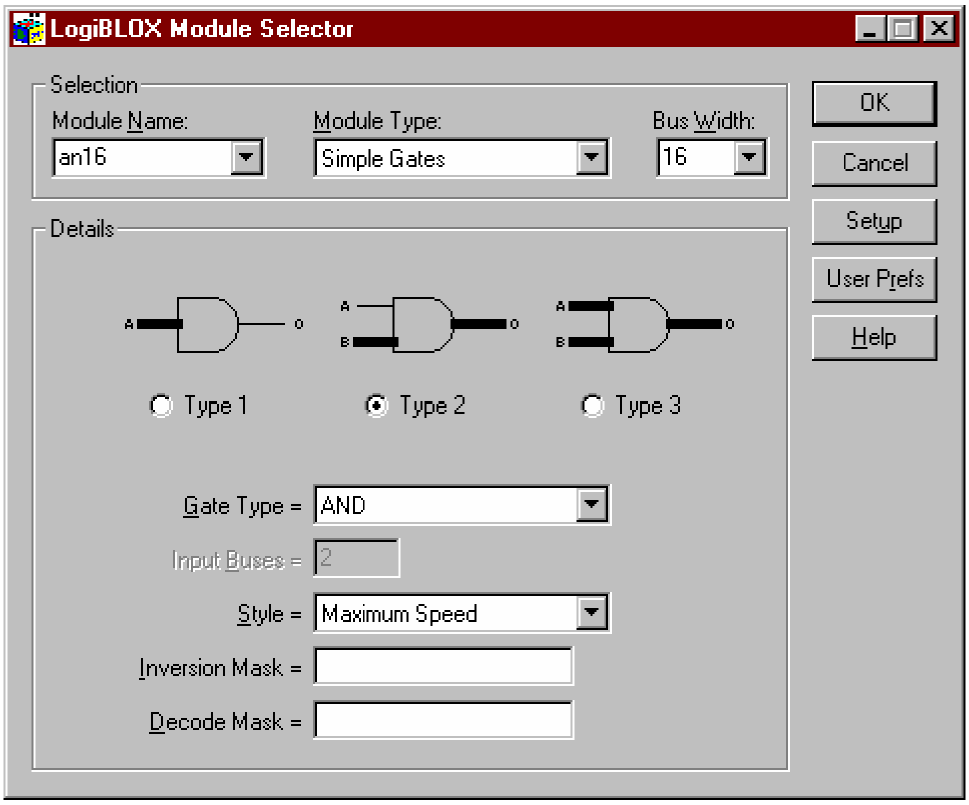
Details
[✓ Type 1] Type 2 Type 3

Gate Type = AND
Input Buses = 2
Style = Maximum Speed
Inversion Mask =
Decode Mask =

Figure 42. AN8 LogiBLOX Module.

2. AN16

Function: Provides Logical AND operation between 16-bit input and 1-bit input.
Type: Simple Gates (Type 2)
Gate Type: AND
Bus width: 16 bits


LogiBLOX Module Selector

Selection
Module Name: an16
Module Type: Simple Gates
Bus Width: 16

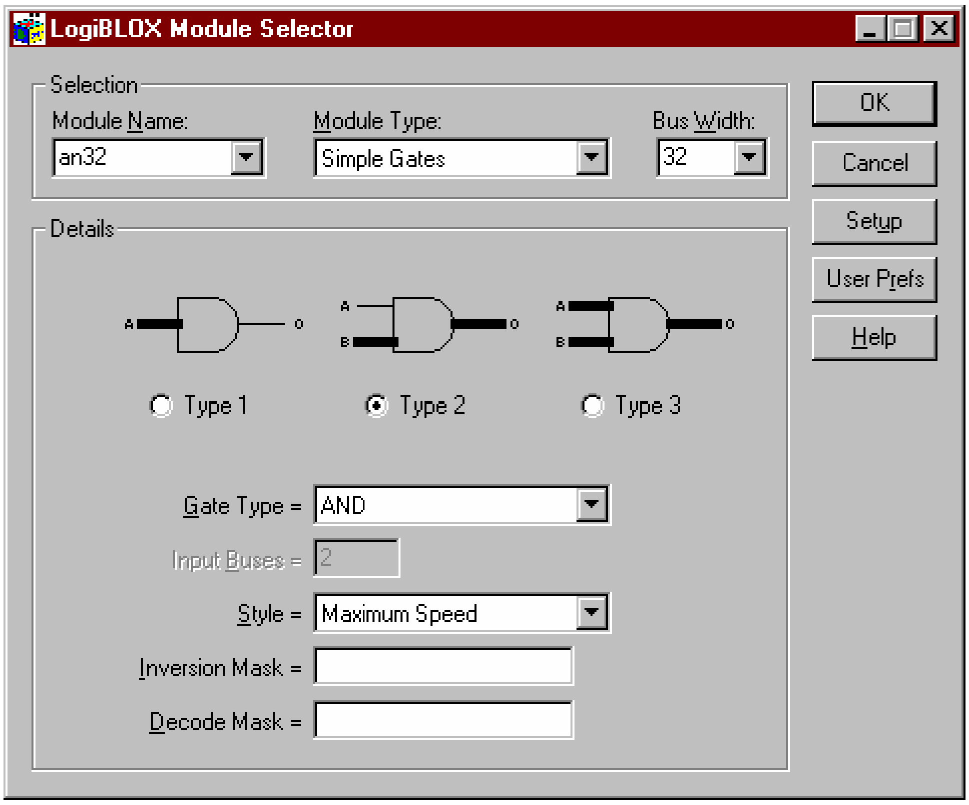
Details
Type 1 [✓ Type 2] Type 3

Gate Type = AND
Input Buses = 2
Style = Maximum Speed
Inversion Mask =
Decode Mask =

Figure 43. AN16 LogiBLOX Module.

3. AN32

Function: Provides Logical AND operation between 32-bit input and 1-bit input.
Type: Simple Gates (Type 2)
Gate Type: AND
Bus width: 32 bits



Module Name: an32
Module Type: Simple Gates
Bus Width: 32

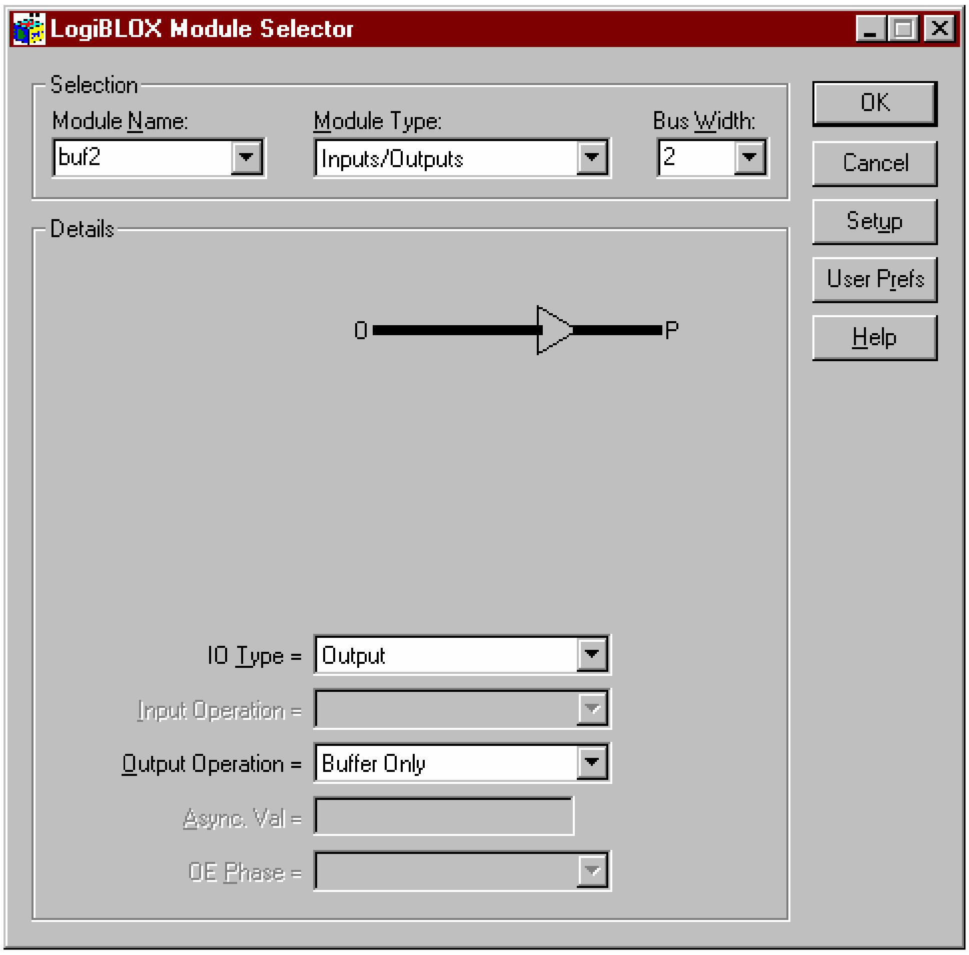
( ) Type 1 [✓ Type 2] ( ) Type 3

Gate Type = AND
Input Buses = 2
Style = Maximum Speed
Inversion Mask =
Decode Mask =

Figure 44. AN32 LogiBLOX Module.

4. BUF2

Function: Provides buffering for 2-bit input.
Type: Inputs/Outputs
IO Type = Output
Output operation = Buffer Only
Bus width: 2 bits


LogiBLOX Module Selector

Selection

Module Name: buf2

Module Type: Inputs/Outputs

Bus Width: 2

Details

IO Type = Output

Input Operation =

Output Operation = Buffer Only

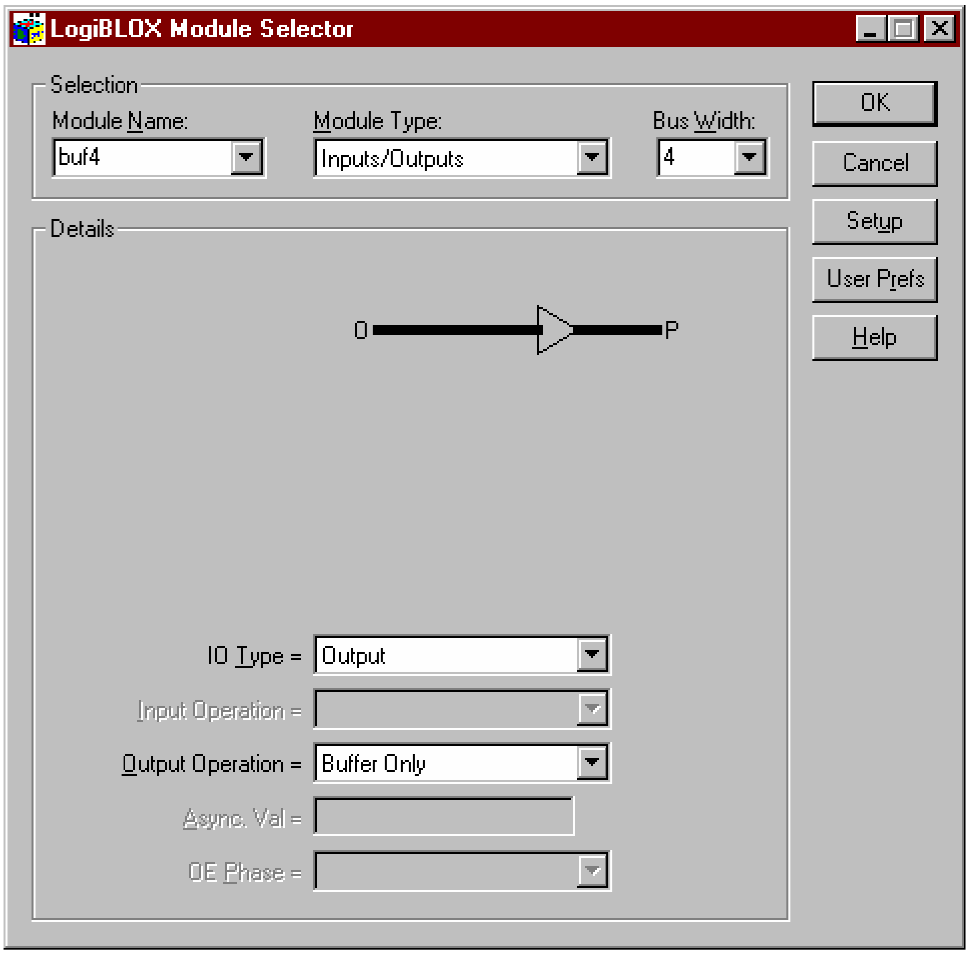
Async. Val =

OE Phase =

Figure 45. BUF2 LogiBLOX Module.

5. BUF4

Function: Provides buffering for 4-bit input.
Type: Inputs/Outputs
IO Type = Output
Output operation = Buffer Only
Bus width: 4 bits


LogiBLOX Module Selector

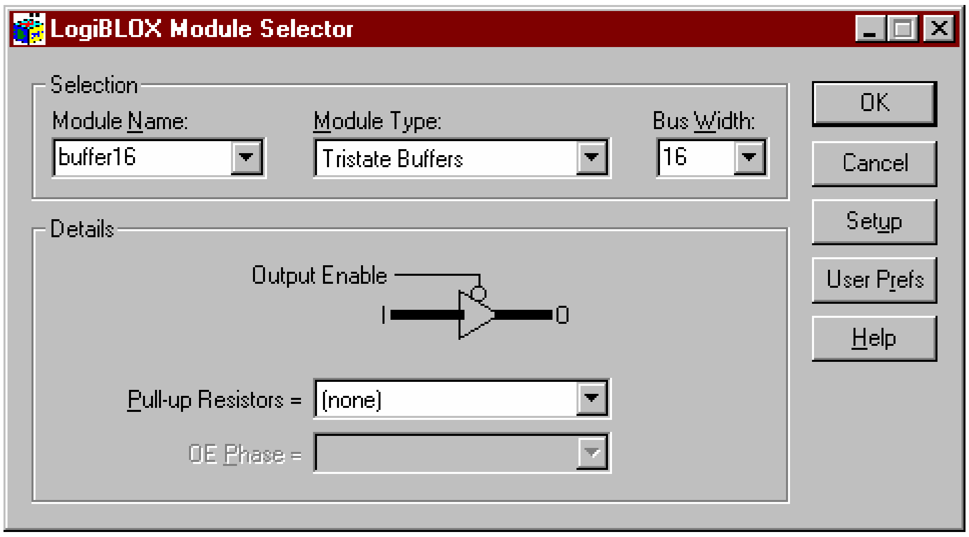
Selection
Module Name: buf4
Module Type: Inputs/Outputs
Bus Width: 4

Details
IO Type = Output
Input Operation =
Output Operation = Buffer Only
Async. Val =
OE Phase =

Figure 46. BUF4 LogiBLOX Module.

6. BUFFER16

Function: Provides tristate buffering for 16-bit input.
Type: Tristate Buffers
Bus width: 16 bits



LogiBLOX Module Selector

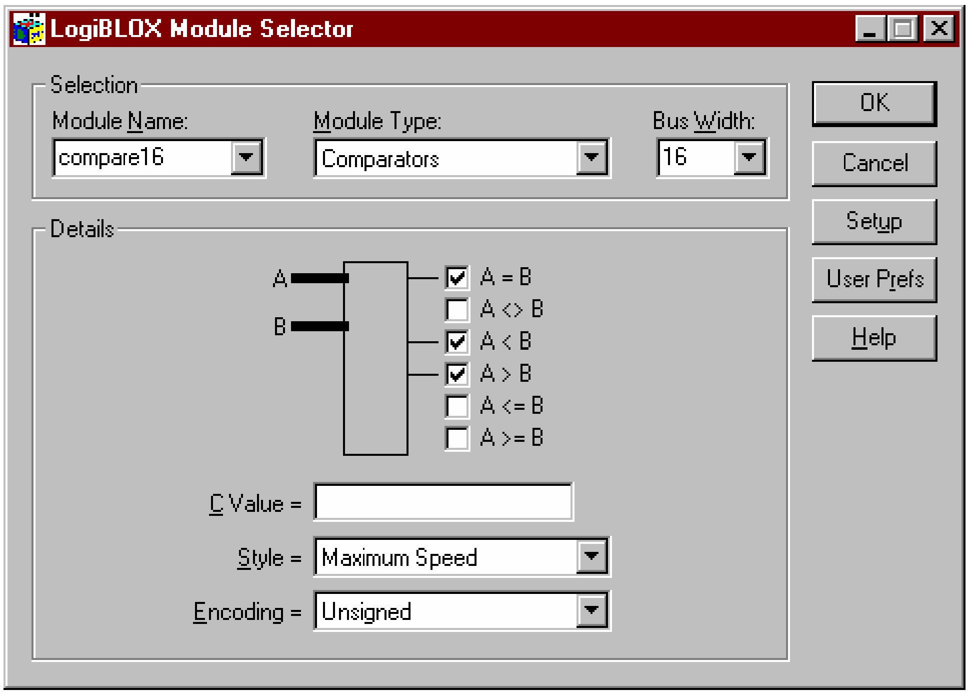
Selection
Module Name: buffer16
Module Type: Tristate Buffers
Bus Width: 16

Details
Pull-up Resistors = (none)
OE Phase =

Figure 47. BUFFER16 LogiBLOX Module.

7. COMPARE16

Function: Compares two 16-bit inputs A and B and returns three lines: A equals B, A is less than B and A is greater than B. These are set according to the result of comparison to High or Low.
Type: Comparators
Operations: A = B, A < B, A > B
Bus width: 16 bits



LogiBLOX Module Selector

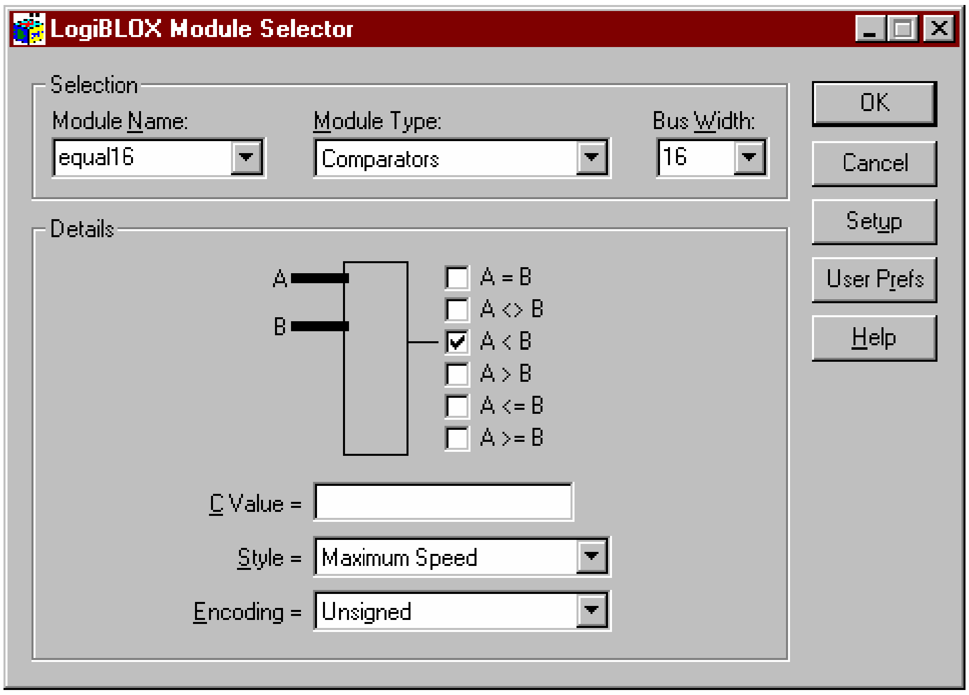
Selection
Module Name: compare16
Module Type: Comparators
Bus Width: 16

Details
[✓] A = B
[ ] A <> B
[✓] A < B
[✓] A > B
[ ] A <= B
[ ] A >= B
C Value =
Style = Maximum Speed
Encoding = Unsigned

Figure 48. COMPARE16 LogiBLOX Module.

8. EQUAL16

Function: Compares two 16-bit inputs A and B and returns one line: A equals B. It is set according to the result of comparison to High or Low.
Type: Comparators
Operations: A = B
Bus width: 16 bits


LogiBLOX Module Selector

Selection
Module Name: equal16
Module Type: Comparators
Bus Width: 16

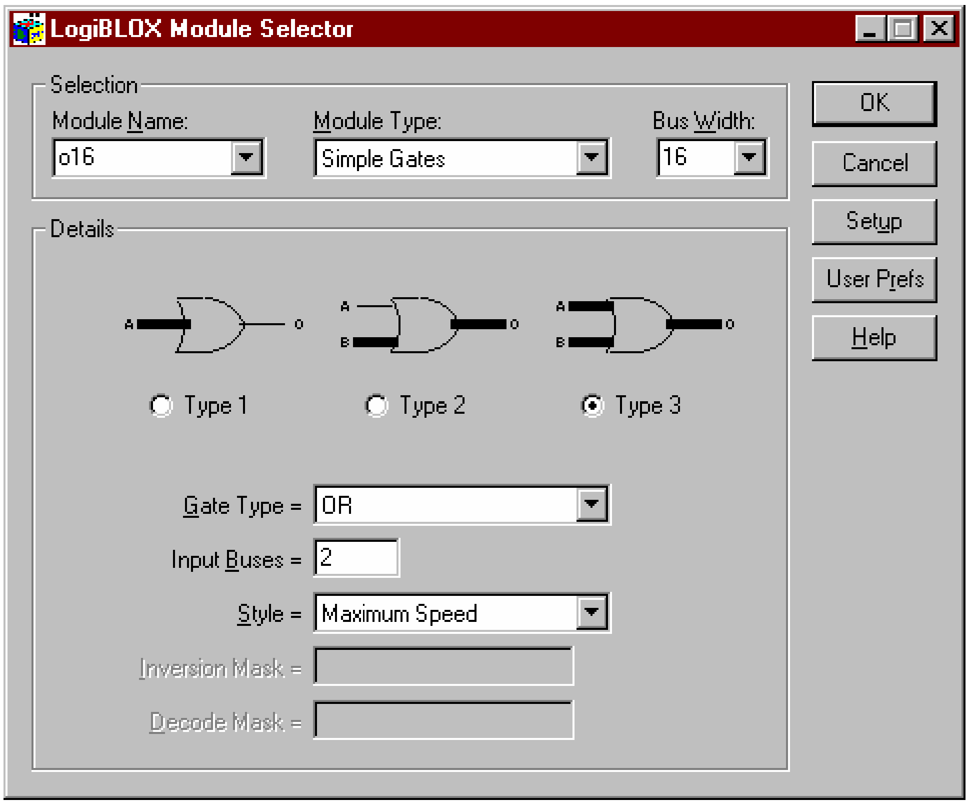
Details
A = B [ ]
A <> B [ ]
A < B [✓]
A > B [ ]
A <= B [ ]
A >= B [ ]

C Value =
Style = Maximum Speed
Encoding = Unsigned

Figure 49. EQUAL16 LogiBLOX Module.

9. O16

Function: Provides Logical OR operation of two 16-bit inputs.
Type: Simple Gates (Type 3)
Gate Type: OR
Bus width: 16 bits


LogiBLOX Module Selector

Selection
Module Name: o16
Module Type: Simple Gates
Bus Width: 16

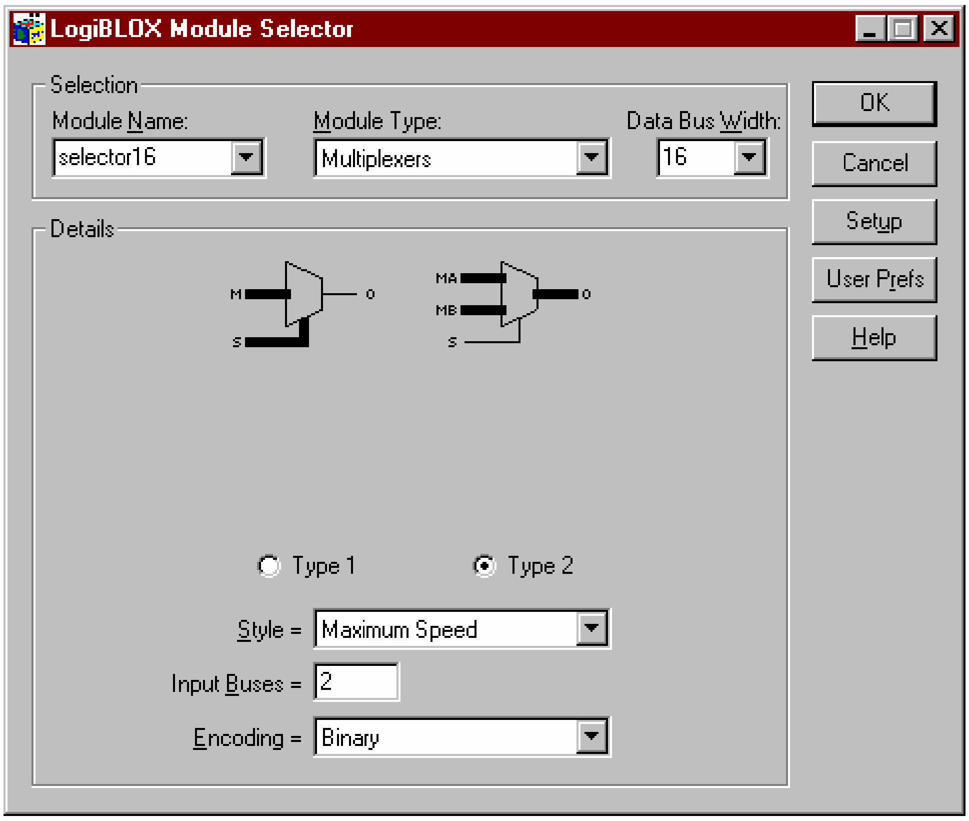
Details
( ) Type 1 ( ) Type 2 [✓ Type 3]

Gate Type = OR
Input Buses = 2
Style = Maximum Speed
Inversion Mask =
Decode Mask =

Figure 50. O16 LogiBLOX Module.

10. SELECTOR16

Function: Multiplexes two 16-bit inputs depending on the value of the control line.
Type: Multiplexers (Type 2)
Input Buses: 2
Bus width: 16 bits



Module Name: selector16
Module Type: Multiplexers
Data Bus Width: 16

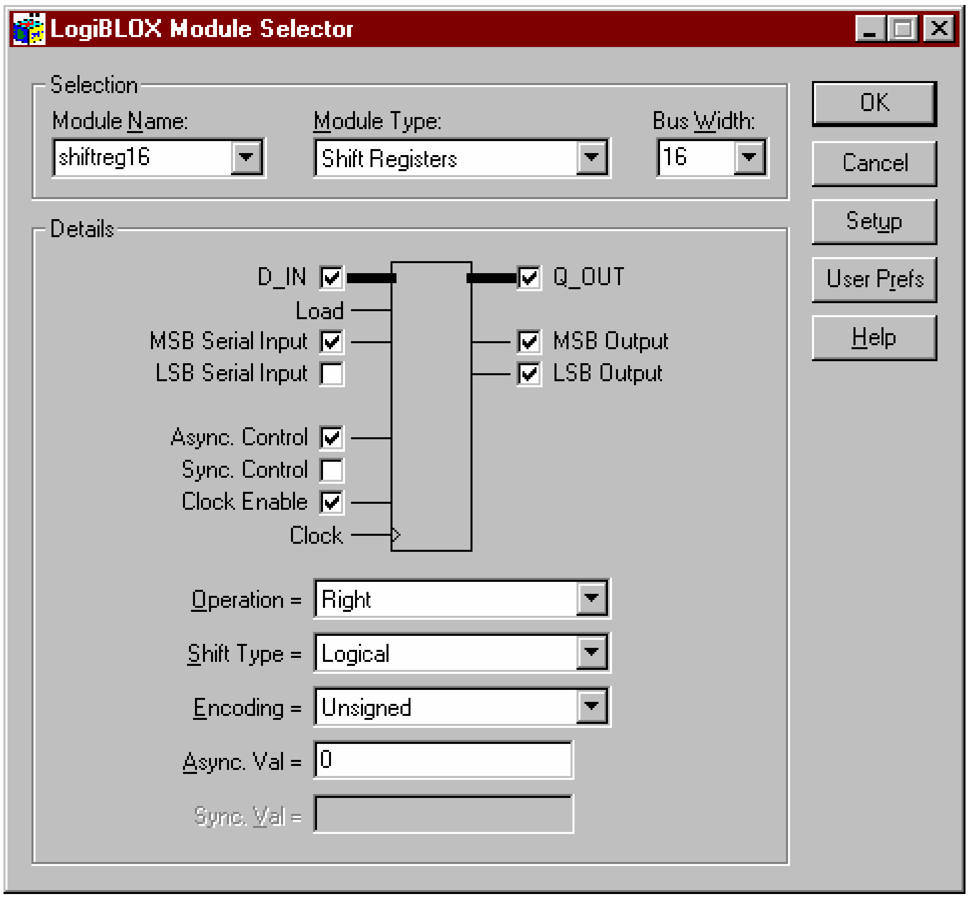
( ) Type 1 (●) Type 2

Style = Maximum Speed
Input Buses = 2
Encoding = Binary

Figure 51. SELECTOR16 LogiBLOX Module.

11. SHIFTREG16

Function: Provides a Logical 16-bit Right Shift Register, with MSB Input and MSB, LSB output
Type: Shift Registers
Shift Operation: Right
Shift Type: Logical
Inputs: MSB Serial
Outputs MSB Serial, LSB Serial
Control: Asynchronous with Clock Enable
Bus width: 16 bits



LogiBLOX Module Selector

Selection
Module Name: shiftreg16
Module Type: Shift Registers
Bus Width: 16

Details
D_IN [✓] — Q_OUT [✓]
Load
MSB Serial Input [✓] — MSB Output [✓]
LSB Serial Input [ ] — LSB Output [✓]
Async. Control [✓]
Sync. Control [ ]
Clock Enable [✓]
Clock

Operation = Right
Shift Type = Logical
Encoding = Unsigned
Async. Val = 0
Sync. Val =

Figure 52. SHIFTREG16 LogiBLOX Module.