

# GETTING STARTED WITH PROGRAMMABLE LOGIC DEVICES, THE 16V8 AND 20V8



© Robert G. Brown  
All Rights Reserved  
August 25, 2000

Alta Engineering  
58 Cedar Lane  
New Hartford, CT 06057-2905  
(860) 489-8003  
www.alta-engineering.com  
alta@ieee.org

## Hands on Design with Programmable Logic Devices

The introduction of programmable logic devices (PLDs) was a great boon to the field of digital hardware design. The second generation PLD, the GAL (which stands for Generic Array Logic, a trademark of Lattice Semiconductor) is particularly suited for the small scale hardware designer. GALs offer the following benefits to the hardware designer:

**Flexibility** - GALs are very flexible devices, they can implement both combinatorial logic functions (AND, OR, NAND etc.) and registered logic functions (counters, shift registers etc.) on the same chip.

**PAL replacement**- The GAL16V8 and GAL20V8 each can directly replace over 20 of the common PAL (Programmable Array Logic - the first generation PLD) types each. This means you only need to stock 2 GAL types to handle your PLD needs.

**Space savings** - In my experience each GAL has typically replaced between 2 and 4 standard TTL chips, saving a large amount of board space.

**Speed** - GALs are fast devices with propagation delay down as low as 7 ns. Typical GALs have a propagation delay of only 15 ns - faster than standard 7400 or 74LS series logic.

**Reprogrammability** - Not only are GALs programmable giving the ability to correct design errors and make board layout easier, they can be reprogrammed up to 100 times. Erasing and programming takes only a few seconds.

**Cost** - In addition to the savings in PC board real estate, standard speed GAL16V8s and GAL20V8s (25 and 15 ns) cost only a few dollars even in small quantities.

There are several varieties of GALs but I will limit this article to the GAL16V8 and GAL20V8. They are easy to design with and are the least expensive and most readily available GAL devices. Rather than get bogged down with the internal details of the devices, we'll cover what is needed to use these PLDs in your designs and then look at a real life design example.

### The Device Architecture

The GAL16V8 is commonly packaged in a standard 20 pin DIP and the GAL20V8 is commonly packaged in a 24 pin skinny DIP (a 24 pin skinny DIP is 0.3 inches wide, the same width as a 20 pin DIP and half the width of a standard 24 pin DIP). The pinout for both devices is shown in figure 1. For the GAL16V8 pin 10 is the ground pin and pin 20 the +5 volt pin (VCC). Pins 12 through 19 are each connected to Output Logic Macro Cells (OLMC). The OLMC allows these pins to act as inputs, combinatorial outputs, registered outputs and input/output pins. Pins 2 through 9 are always general purpose input pins. If any of the OLMC are configured as registered outputs then pin 1 is a Clock input and pin 11 is the Output Enable for the registered outputs. If none of the OLMC

are registered then pins 1 and 11 are general purpose inputs. Internal to the chip is an array of and/or logic that is configured with each OLMC when the chip is programmed. The 20V8 has a similar design, the main difference from the 16V8 is the four additional input pins.

## The Design Tools

In addition to your PC you will need only three tools to do design work with PLDs, a text editor, a logic compiler and a device programmer. A logic compiler is a program that translates a high level design file, in which the relationship between inputs and outputs is expressed in the form of equations, to a low level file device specific file for the programmer. The low level file used by the programmer is called a JEDEC file and is sometimes referred to as a 'fuse map'. (Earlier PLDs were programmed by literally blowing up fuses internal to the device leaving only the desired connections - of course they could not be reprogrammed - you threw away your mistakes.) National Semiconductor used to offer a FREE logic compiler before they got out of the PLD business. You can still get a copy of their PLAN logic compiler off of the Alta Engineering web site at [www.alta-engineering.com](http://www.alta-engineering.com). The high level design file for PLAN is called an equation file and uses the extension .EQN, the output JEDEC file uses the extension .JED. Since PLAN is available to everyone I will use it in the examples, the concepts however are universal, not specific to PLAN. The equation file is a standard ASCII text file and can be produced using any text editor.

## Designing with Equations

If you normally design with standard TTL devices, shifting to design using GALs might take a slight adjustment. However the underlying concepts are the same. In the equations a + is used to represent OR, a \* to represent AND a / for NOT or inversion. In figure 2a show the equation represent by an AND gate. Each group of signals ANDed together is referred to as a product term. Figure 2b shows the equivalent representation for a two input OR gate. In figure 2c a more complicated piece of logic is represented, it includes the use of the / symbol to show inversion. Notice how the equation is organized. The equations are written in a sum of products format, a useful convention is to list each product term on a separate line. The inversion can also occur on the output as shown is in figure 2d. In the 16V8 and 20V8 up to eight outputs can be defined in this way (each of the eight OLMC). The inputs for the equations can come from any of the input or output pins either normal or inverted. A maximum of seven or eight product terms are allowed for each OLMC, this depends on the exact configuration of the OLMC. Given this, it is obvious that a single GAL can replace several packages of AND, NAND, NOR and OR gates. But this is only the start.

So far all the examples have used combinatorial logic, in addition GALs can also handle sequential logic such as shift registers and counters. To do this the OLMC is configured as a register (D flip flop). If any of OLMC are configured as registered then pin 1 is the clock input to the register. Where as an = symbol is used to show a combinatorial output in an equation, a := symbol is used to show a registered equation output. For example:

$Q := D$

The  $:=$  indicates that the output Q is registered. This means that Q will take on the value of D following the rising edge of the clock on pin 1. Two or more outputs can be combined to form counters and shift registers as shown in figure 3. In this case it shows a two bit counter with a terminal count. The two outputs Q0 and Q1 will count from 0 to 3 continuously and the terminal count indication will be active when the count is at its maximum value of 3.

The registered outputs have a common output enable at pin 11 on a 16V8 and pin 13 on a 20V8. When output enable is low the registered outputs are all enabled. If output enable goes high all the registered outputs will be disabled (tri-stated). Even when the outputs are tri-stated the register outputs are still available internally as feedbacks (so the counter would continue to work even if the outputs were disabled). Combinatorial outputs can each have an output enable defined, this is limited to a single product term. For example:

$X.OE = B * C$

This would indicate that output X should be enabled when B AND C are high.

### A Real Design Example

To illustrate the use of GALs in a real design, I will use the main board from a high speed, low cost 16 channel logic analyzer. The logic analyzer main board uses a total of 17 ICs, of these 2 are static RAMs, 4 are octal latches, 4 are octal buffers and the remaining 7 are programmed GAL16V8s. All standard logic was handled by the 7 Gals, they replace about 20 high speed TTL ICs and make the logic analyzer buildable. (For more information on the logic analyzer design download RGBLOGIC.ZIP from the TCJ BBS, the schematic and all of the EQN files are included in the file, so you can review the complete design.)

Central to the logic analyzer is the clock selection GAL. This allows the analyzer clock to be selected from 5 different internal clock, an external clock or a software controlled clock. The PLAN equation file, CLOCK.EQN is shown in figure 4. The lines that start with a ; are comment lines, they are ignored by the compiler and are used for documentation. The equation file must contain two sections, the declaration section and the equation section. The declaration section should appear first and is indicated by the keyword CHIP. The line:

CHIP clock 16V8

begins the declaration block, it gives the chip a name (clock) and selects the device type (16V8). The next part of the declaration block is optional, but will be needed in most cases. In this section we assign symbolic names to each pin on the chip, as follows:

```
clk nc ext self s2 s1 s0 dir wr gnd /oe sysclk1 wrdat sysclk0 q4 q3 q2 q1 q0 vcc
```

The symbolic names start with pin 1 and are assigned in order through pin 20. In this case pin 1 is assigned the name clk, pin 9 the name wr, pin 12 the name sysclk1 and pin 19 the name q0. If we later need to change the pin assignments, we simply rearranged the names given here.

The equation section of the file is indicated by a line with the word EQUATIONS. Outputs q0, q1, q2, q3 and q4 form a 5 bit counter (a straight forward expansion of the 2 bit counter used before). This counter provides several reference frequencies at the outputs, q1 is half of q0, q2 is half of q1 etc. With a crystal oscillator of 40 Mhz connected to pin 1, q0 provides a 20 Mhz clock, q1 a 10 Mhz clock, q2 a 5 Mhz clock, q3 a 2.5 Mhz clock and q4 a 1.25 Mhz clock. The outputs sysclk0 and sysclk1 use identical equations, so I refer to them together as sysclk. The sysclk outputs are combinatorial, they allow the system clock to be selected from q0, q1, q2, q3, q4 or the inputs self or ext. The inputs s0, s1 and s2 select which clock is output to sysclk. From the equations you can see that if s0, s1 and s2 are all 0 then sysclk is the same as the input self. If s0, s1 and s2 are all 1 then the last product term will apply and sysclk will follow q0. The final output defined in the equations is wrdat, this is obviously just a simple two input OR function. I remember that Bill Kibler said he wanted to see the equivalent circuit in standard logic for all programmable logic used in TCJ. It is not always possible, but in this case I had first looked designing the logic analyzer using standard logic. The equivalent circuit to this GAL is shown in the schematic in figure 5, notice the savings in chip count and cost.

The equation file is compiled it with the command line:

```
EQN2JED -N CLOCK
```

EQN2JED is the PLAN program that produces the JEDEC file, this will take our CLOCK.EQN file check it for errors and if error free produce the JEDEC file CLOCK.JED. The -n in the command line tells the program to produce a new log file, rather than append to the existing log file. The log file produced is CLOCK.LOG. The log file has a lot of interesting information about the programmed device the use of each device pin, the product term usage and the device pinout.

### A Memory Decoder

Let's take a look at a real life example that was suggested to me by Dave Baldwin (Editor of The Computer Journal). Let's say that we are designing a Z80 based system and we would like the memory map to include an 8K EPROM at address 0000h, a 2K EEPROM at address 2000h and have the remainder of the 64K address space filled with static ram (2 32K devices). So a table of our memory map looks like:

Address (hex)	Device 0000-1FFF	EPROM 2000-27FF
	EEPROM 2800-7FFF	SRAM0 8000-FFFF
		SRAM1

This type of decoding is a pain in the butt if we use standard TTL devices. We would need several devices and have to deal with the problems of propagation delays through the several levels of devices. If we use a 16V8 or 20V8 we can easily handle this decoding with one 20 or 24 pin device. Our total propagation delay will be the single propagation delay of the chip. This means we can have a propagation delay as low as 5 ns with a GAL rated at 5 ns. In addition we can include the decoding of the I/O space on the same chip.

In this case defining the equations for each chip select output is very straight forward. We want the EPROM chip select to be active when address lines A15, A14 and A13 are low and the Z80 MREQ signal is low. Since the active state of the EPROM chip select is low we would express the equation as:

$$\text{/EPROM} = \text{/A15} * \text{/A14} * \text{/A13} * \text{/MREQ}$$

The EEPROM chip select is only slightly more involved. The address range 2000-27ff is selected when A15, A14, A12, A11 and MREQ are low, while A13 is high. This gives the equation:

$$\text{/EEPROM} = \text{/A15} * \text{/A14} * \text{A13} * \text{/A12} * \text{/A11} * \text{/MREQ}$$

The first static ram has the most complicated equation (but still well within the capabilities of the 16V8). The 2800-7FFF address space can be thought of as three regions 2800-2FFF, 3000-3FFF and 4000-7FFF. The product terms defining each of these regions is then ORed together to define the complete equation as follows:

$$\text{/SRAM0} = \text{/A15} * \text{/A14} * \text{A13} * \text{/A12} * \text{A11} * \text{/MREQ} + \text{/A15} * \text{/A14} * \text{A13} * \text{A12} * \text{/MREQ} + \text{/A15} * \text{A14} * \text{/MREQ}$$

It is obvious that the equation for the second static RAM's chip select is:

$$\text{/SRAM1} = \text{A15} * \text{/MREQ}$$

The complete equation file with the I/O decoding using a 16V8 is printed above. If we wanted finer granularity on the I/O decode we could use a 20V8, this would give us 4 additional inputs for address lines, that could be included in the equations. If we wanted to have a 10 chip select outputs we could use a GAL22V10. Note that the pin out selected is arbitrary, in this case we could swap any of the input pins or any of the output pins just by redefining the pin list. This is a great aid if you do your own PC board design.

I think this example shows you why I use programmable logic wherever I can. We have reduced our decode logic to one 20 pin device. In doing so we have reduced the number of interconnects, saved PC board space, saved money and saved both circuit and PC board design time. In addition we have increased our design flexibility and helped our parts inventory. If we later find that we must have a 16K EPROM instead of the 8K

EPROM. we can change our decode circuit by just reprogramming the GAL16V8. We can also replace our bin of spare TTL chips with few 16V8s. Once you start using programmable logic you can see the advantages continue to pile up.

### Circuit Layout

When using PLDs you must use the same care in circuit layout as you would need to with any high speed logic device. Completely covering the topic of high speed circuit design would fill a book, but here are some things to look for. Be careful with the ground and power layouts to reduce the impedance of these signals to the chip. Use a decoupling capacitor as close as possible to each chip's power and ground pins.

### Device programmers

The device programmer will program the device with logic as defined in the JEDEC file. One example of a device programmer that will program PLDs is the Alta Pro 2000. For more information check out the web site at [www.alta-engineering.com](http://www.alta-engineering.com). I will also post follow on PLD design tutorial on the web site about using the 22V10.

```

; GCLK. EQN Logic Analyzer U15
; This is the declaration section
CHIP GCLK 16V8

; Pin labels here
CLK NC EXT SELF S2 S1 S0 DIR WR GND
OEN SC1 WRDAT SC0 Q4 Q3 Q2 Q1 Q0 VCC

; Next is the equation section
EQUATIONS

; Q0 - Q4 form a 5 bit binary counter

Q0 := /Q0

Q1 := Q0 * /Q1 +
      /Q0 * Q1

Q2 := Q0 * Q1 * /Q2 +
      Q2 * /Q0 +
      Q2 * /Q1

Q3 := Q0 * Q1 * Q2 * /Q3 +
      Q3 * /Q0 +
      Q3 * /Q1 +
      Q3 * /Q2

Q4 := Q0 * Q1 * Q2 * Q3 * /Q4 +
      Q4 * /Q0 +
      Q4 * /Q1 +
      Q4 * /Q2 +
      Q4 * /Q3

; SC0 and SC1 output the system clock as selected by
; S0, S1 and S2

SC0 = /S0 * /S1 * /S2 * SELF +
      S0 * /S1 * /S2 * EXT +
      S0 * S1 * /S2 * Q4 +
      /S0 * /S1 * S2 * Q3 +
      S0 * /S1 * S2 * Q2 +
      /S0 * S1 * S2 * Q1 +
      S0 * S1 * S2 * Q0

SC1 = /S0 * /S1 * /S2 * SELF +
      S0 * /S1 * /S2 * EXT +
      S0 * S1 * /S2 * Q4 +
      /S0 * /S1 * S2 * Q3 +
      S0 * /S1 * S2 * Q2 +
      /S0 * S1 * S2 * Q1 +
      S0 * S1 * S2 * Q0

; WRDAT is a simple OR function of WR and DIR
WRDAT =WR +
      DIR

```

```

; Z80 memory and I/O decoder example - TCJ
; Robert G. Brown - ALTA ENGINEERING (860) 489-8003
; Memory Map
; 0000-1FFF EPROM
; 2000-27FF EEPROM
; 2800-7FFF SRAM0
; 8000-FFFF SRAM1
;
CHIP decode 16V8

```

```

; set pinout - can be altered later if needed
MREQ A15 A14 A13 A12 A11 IOREQ A7 A6 GND
A5 EPROM EEPROM SRAM0 SRAM1 I00 I01 I02 I03 VCC

```

EQUATIONS

$$/EPROM = /A15 * /A14 * /A13 * /MREQ$$

$$/EEPROM = /A15 * /A14 * A13 * /A12 * /A11 * /MREQ$$

$$/SRAM0 = /A15 * /A14 * A13 * /A12 * A11 * /MREQ +$$

$$/A15 * /A14 * A13 * A12 * /MREQ +$$

$$/A15 * A14 * /MREQ$$

$$/SRAM1 = A15 * /MREQ$$

$$/I00 = /A7 * /A6 * /A5 * /IOREQ$$

$$/I01 = /A7 * /A6 * A5 * /IOREQ$$

$$/I02 = /A7 * A6 * /A5 * /IOREQ$$

$$/I03 = /A7 * A6 * A5 * /IOREQ$$

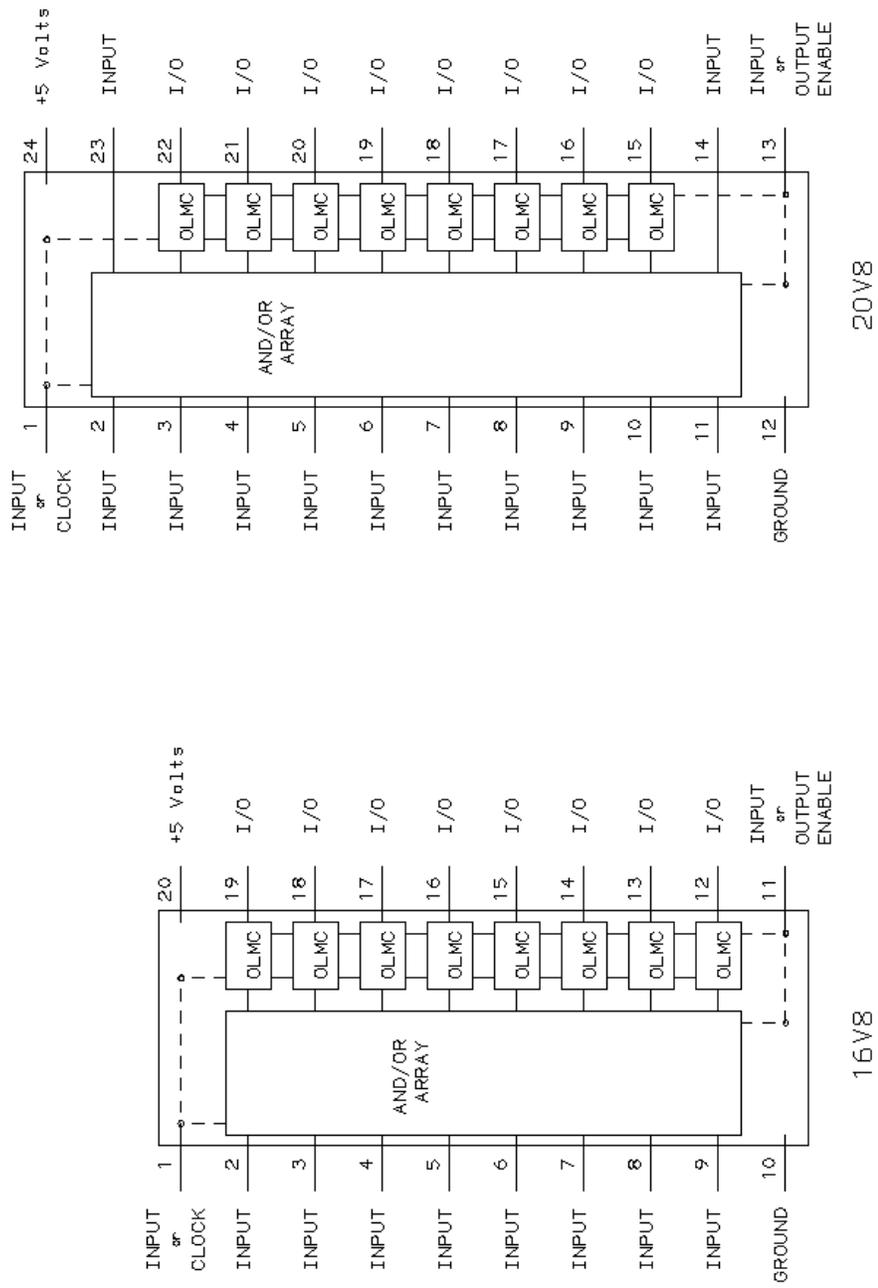
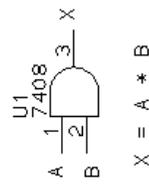
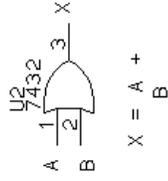


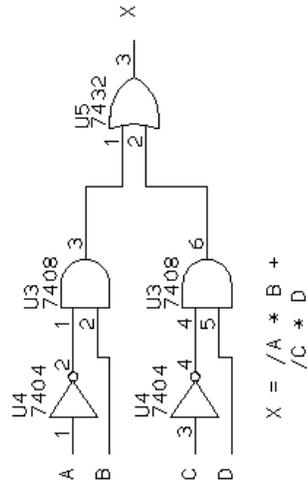
FIGURE 1 - GAL16V8 and GAL20V8 ARCHITECTURE



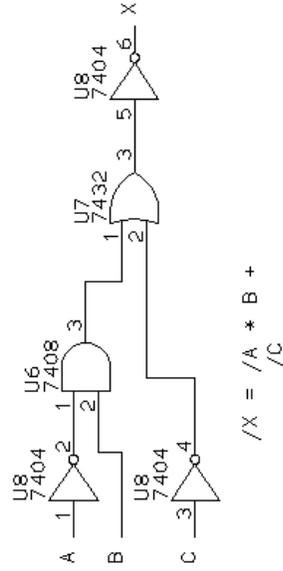
(a) - Two Input AND Gate



(b) - Two Input OR Gate



(c) - AND, OR and NOT (inversion)



(d) Inversion on the Output (Active Low Output)

FIGURE 2 - COMBINATORIAL LOGIC EQUATIONS

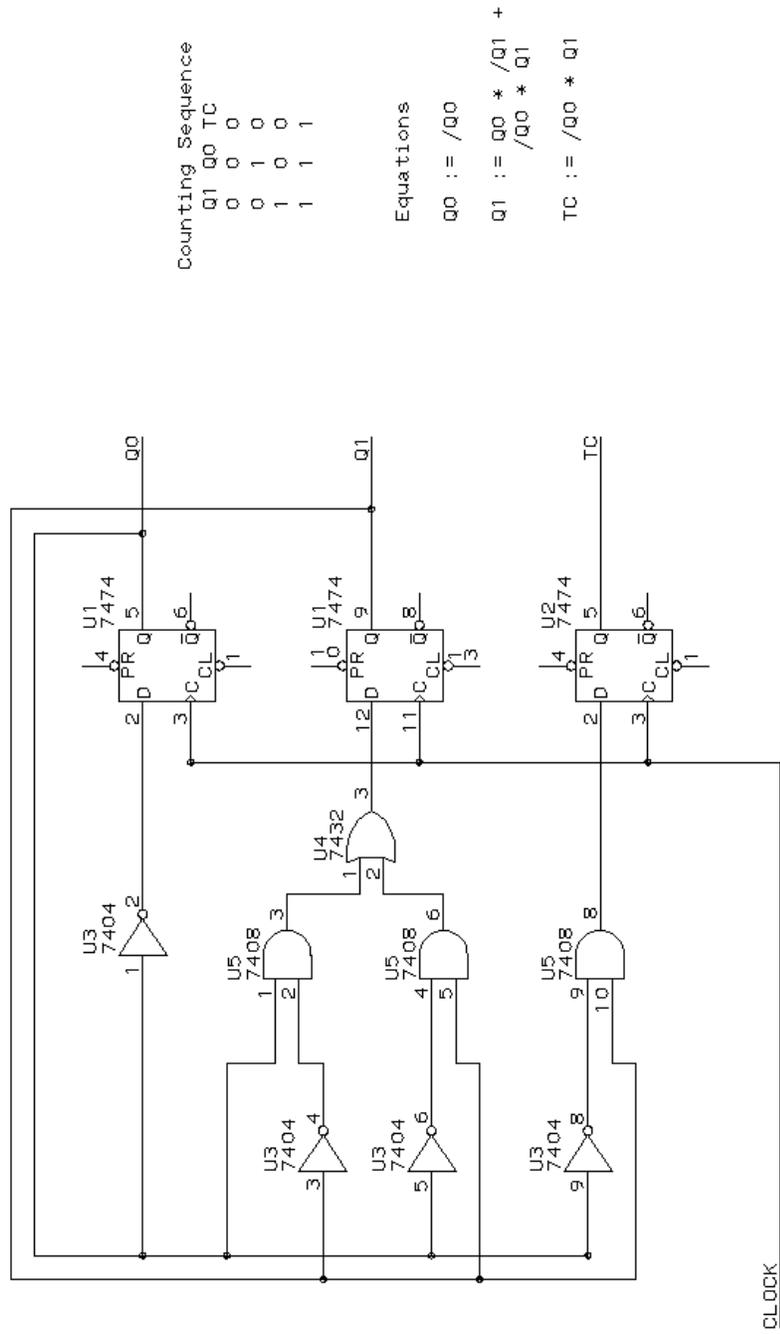
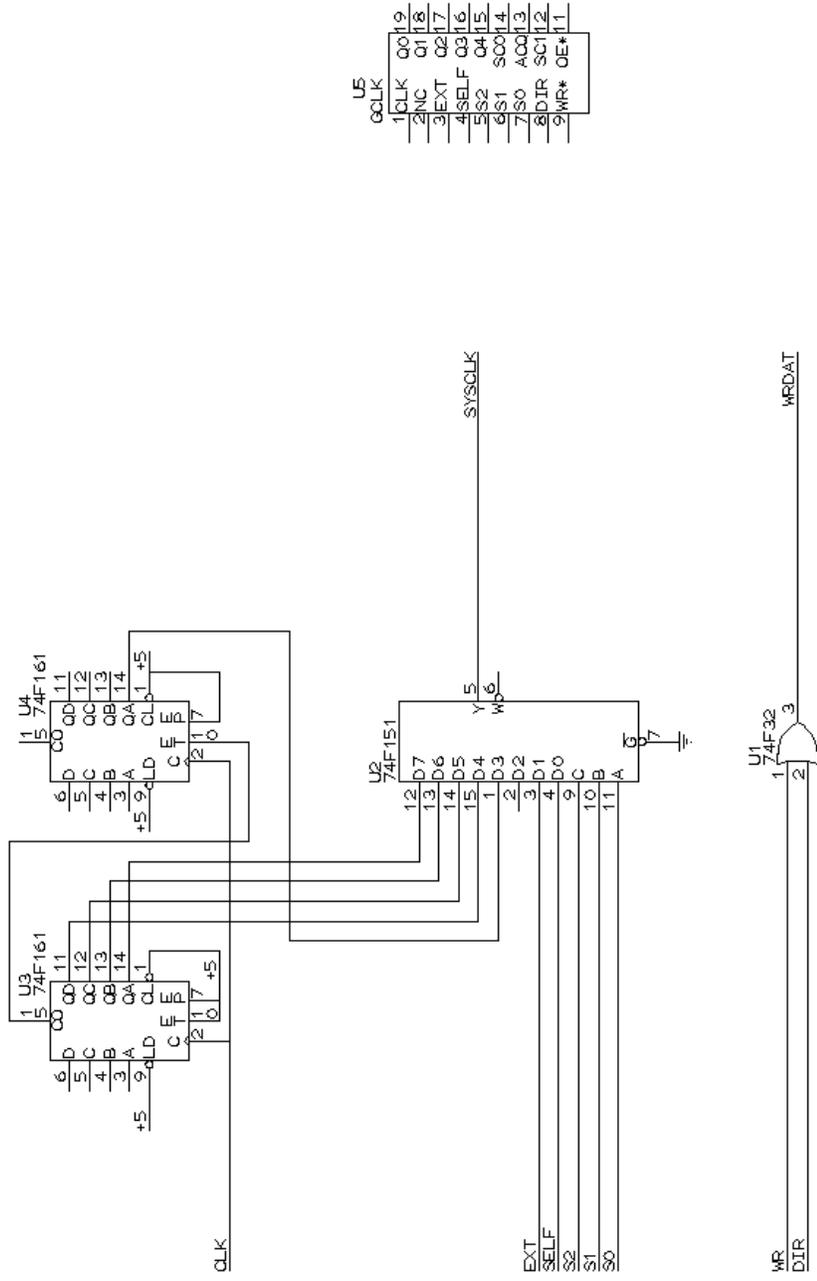


FIGURE 3 - REGISTERED LOGIC - COUNTER



GAL EQUIVALENT

TTL EQUIVALENT TO CLOCK

FIGURE 5 - TTL and GAL CIRCUIT COMPARISON