# *ECET 4530*

## *Industrial Motor Control*

### *Introduction to Ladder Logic Programming*

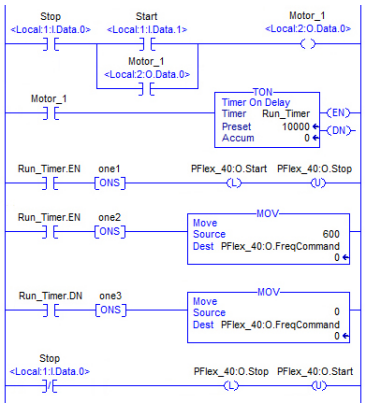
***(in the RSLogix environment)***

# Ladder Logic Programming

**<u>Ladder Logic</u> is one of many different languages* that can be utilized when programming a PLC.**

**Ladder Logic programs take the form of a <u>Ladder Diagram</u>, the contents and structure of which defines the PLC's operation.**

**A Graphical User Interface (GUI) is typically utilized in order to provide a simple drag-and-drop method for creation of the ladder diagram.**



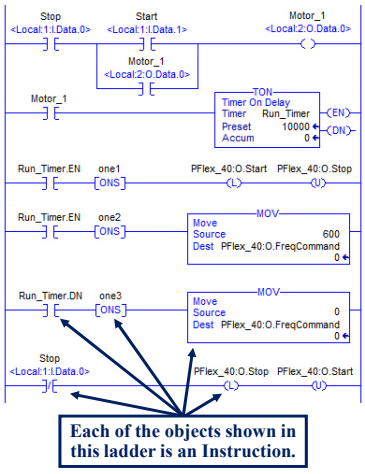

*** – other languages include Function Block Diagram (FBD), Instruction List (IL), Structured Text (ST), Sequential Function Chart (SFC), Continuous Function Chart (CFC), as well as high-level programming languages like Automation Basic and C++.**

# Presentation Format – Part A

**This presentation on Ladder Logic Programming will begin by introducing the overall structure of a Ladder Diagram.**

**Included within the introduction is a brief discussion on the general types of Instructions that will appear within a Ladder Diagram, and the overall flow of the program as it is executed.**

**Several commonly used Instructions will then be discussed in detail, including both the format of the instructions and their operation when utilized within a Ladder Logic Program.**

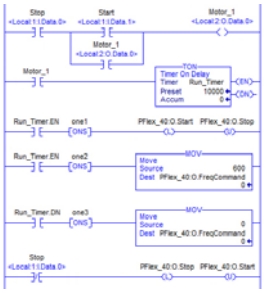


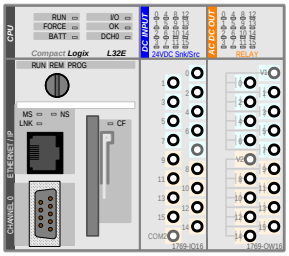

**Each of the objects shown in this ladder is an Instruction.**

# Presentation Format – Part B

**The second part of this presentation will focus on the practical implementation of Ladder Logic Programming in order to control the operation of a PLC.**

**The method for linking the operation of the PLC's Input and Output Ports to the operation of the Instructions within a Ladder Diagram will first be presented, after which several basic control system tasks will be implemented with the use of Ladder Logic Programming.**






**Allen-Bradley CompactLogix PLC**

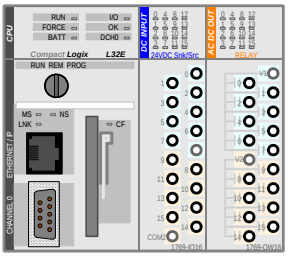
# RSLogix/Studio 5000

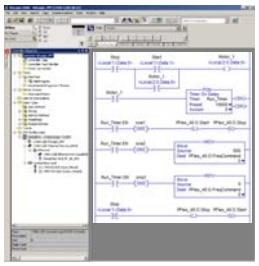
**RSLogix 5000*** **is a platform developed by Rockwell Software for the programming of Allen-Bradley PLCs.**

**RSLogix 5000 (v.21) is shown in this presentation since we will utilize that software to program the PLCs in the lab.**

**Note that Ladder Logic is not exclusive to the RSLogix platform.**

**It is a standardized programming language (IEC 61131-3) that has been adopted by all PLC manufacturers because of its structural similarity to relay-logic-based control circuits.**



RSLogix 5000 Software Platform

* – Studio 5000 is the new version of the RSLogix platform that was developed to support PLCs with multi-core processors. Although this presentation discusses RSLogix 5000, the contents also applies to the Studio 5000 platform.

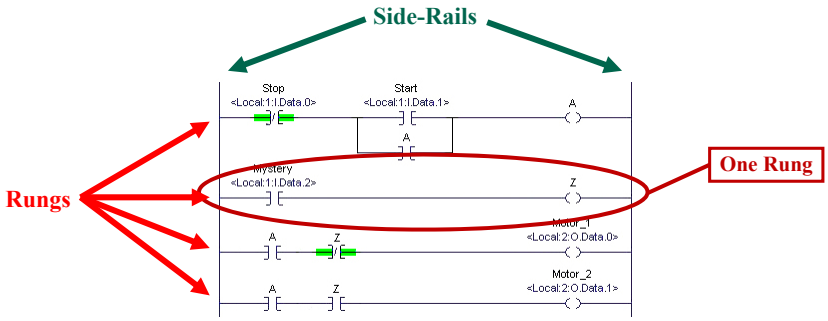
(*Part A*)

# *Introduction to Ladder-Diagrams*

# Ladder Diagrams

**Ladder Diagrams are graphical representations of a ladder logic program.**

**They are named such because of their ladder-like appearance, with two vertical Side-Rails and multiple horizontal Rungs.**



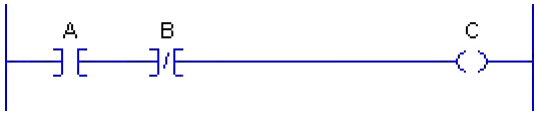

# Ladder Diagrams – Rungs

**The Rungs of the Ladder Diagram contain multiple Instructions that, when combined together, can provide the function of one or more lines of code in a text-based programming language.**

**For example, the function**

**let C=1 if (A=1 and B=0) else let C=0**

**can be implemented using either Ladder-Logic or C++ as:**




Ladder Logic Rung

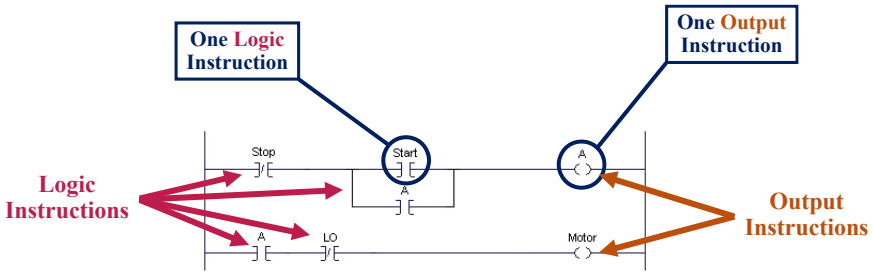
```
If ( A == 1 && B == 0) {
   C = 1
}
else {
   C = 0
}
```

C++ If-Else Statement

# Ladder Logic Instructions

**Ladder Logic Instructions can be separated into two primary categories:**

- **Output Instructions**
- **Logic (Input) Instructions**

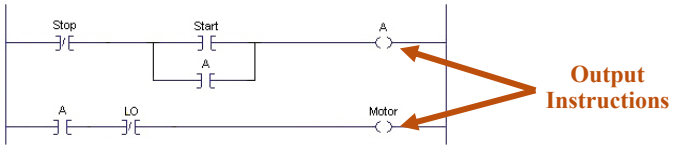



# Output Instructions

**Output Instructions perform a task.**

**The task performed may be simple or complex depending on the specific instruction, such as:**

- **Changing the Value of a Number Stored in Memory**
- **Turning ON or OFF one of the PLC's Output Ports**
- **Operating as a Timer or a Counter**
- **Energizing or Changing the Output Frequency of a VFD**



# Output Instructions

**Output Instructions perform a task.**

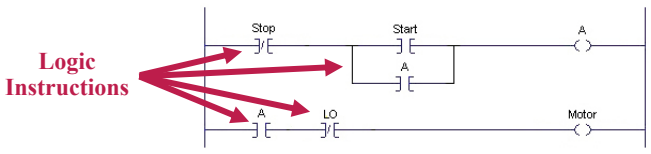
**The task performed may be simple or complex depending on the specific instruction, such as:**

- **Changing the Value of a Number Stored in Memory**
- **Turning ON or OFF one of the PLC's Output Ports**
- **Operating as a Timer or a Counter**
- **Energizing or Changing the Output Frequency of a VFD**

# Logic Instructions

**Logic Instructions provide the functional logic that controls the operation of the Output Instructions.**

**In general, the "associated parameters" are one or more values stored in specific memory locations that are directly linked to a specific logic instruction.**



# Rung Requirements

Each rung must have at least one Output Instruction.

Multiple Output Instruction may be placed in-series and/or in-parallel on the same rung.

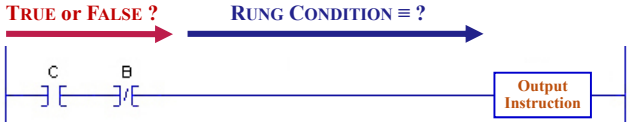
Multiple Logic Instructions may be placed in-series and/or in-parallel on the same rung.

Each rung is not required to have any Logic Instructions.

* – If multiple Output Instructions are placed on the same rung, Logic Instructions may be placed between those instructions provided that the right-most position on the rung is occupied by an Output Instruction. Note that rungs configured in this manner will not be covered in this presentation.

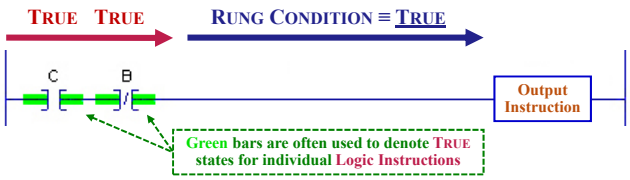
# Rung Condition

Rung Condition is a TRUE/FALSE logic state that is based on the state and placement of a rung's Logic Instructions and whether or not those instructions provide at least one path through Logic Instructions that are all TRUE, beginning from the left side-rail and progressing to the right.



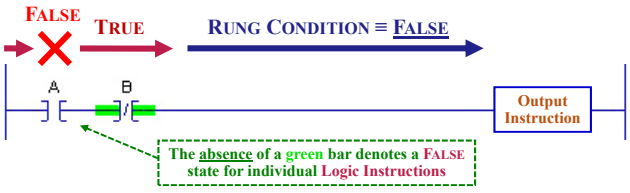
# TRUE Rung Condition

An Output Instruction experiences a TRUE Rung Condition when the rung's Logic Instructions provide a "TRUE path" from the left-hand side-rail of the ladder to the left side of the Output Instruction.



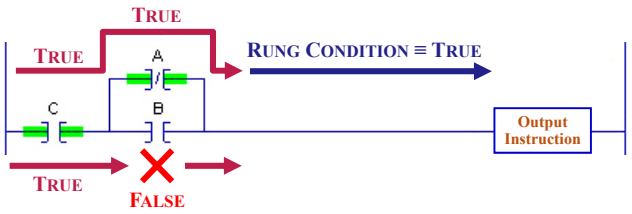

# FALSE Rung Condition

An Output Instruction experiences a FALSE Rung Condition when the rung's Logic Instructions prevent a "TRUE path" from the left-hand side-rail of the ladder to the left side of the Output Instruction.



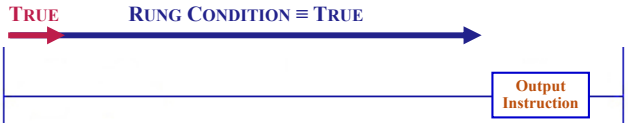
# Rung Condition

**Note that the Logic Instructions placed on a rung may offer more than one potential path from the left-hand rail to an Output Instruction.**




# Empty Rung Condition

**When determining the Rung Condition, a rung is assumed to begin with a TRUE logic state at the left side-rail.**



# Rung Condition Evaluation Order

**Rung Condition is not affected by an Output Instruction when progressing from left-to-right across a specific rung.**

**Thus, if a rung contains multiple Output Instructions, then:**

- **the Rung Condition for the left-most Output Instruction is determined first and the operation that instruction is completed, after which**
- **the process repeats for each additional Output Instruction on the rung until the right side-rail is reached.**

# Rung Condition Evaluation Order

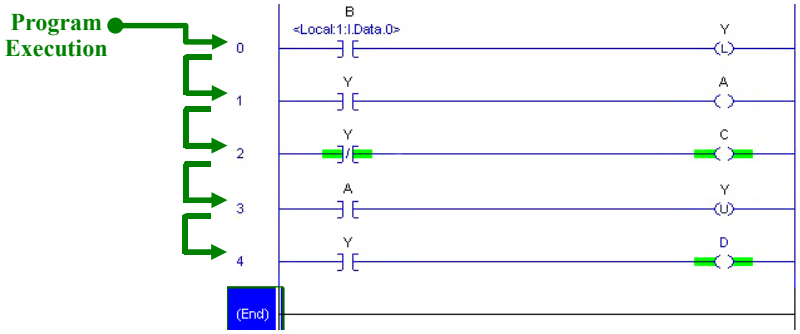
**Even if the operation of an Output Instruction would change the state of the previously-evaluated Logic Instructions, those Logic Instructions will not be re-evaluated and the previously-determined Rung Condition (to the left) will remain unchanged until the next time the rung is executed.**

**On the other hand, an Output Instruction may affect the state of any Logic Instructions that are placed to its right on a rung, in-turn possibly affecting the Rung Condition that is experienced by any additional Output Instructions that are also placed to its right*.**

*** – This is a complex situation that will not be covered in this presentation.**

# Ladder Logic Program Execution

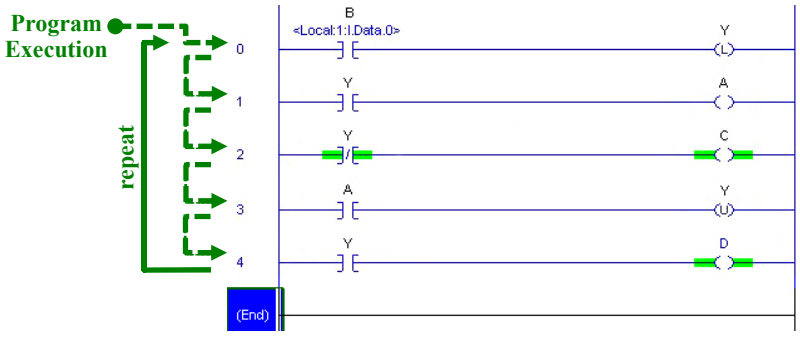
*** – The PLC has two primary modes of operation, PROGRAM and RUN. When switched to PROGRAM mode, the controller stops executing its Ladder Logic program.**




# Ladder Logic Program Execution

**When the controller reaches the End rung, it jumps back to the top of the ladder (Rung 0) and the process begins again.**

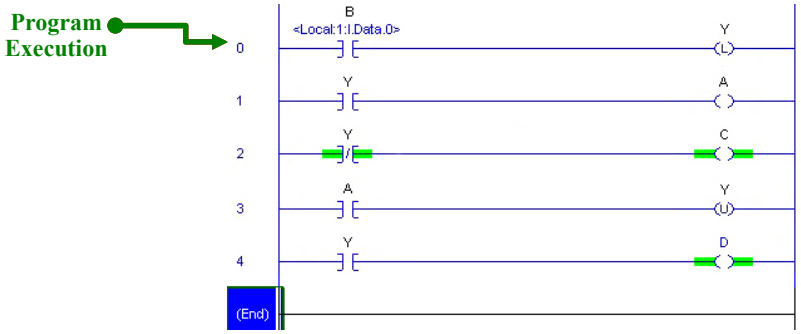
**The controller will repetitively keep stepping through the ladder as long as the PLC remains in RUN mode.**



# Individual Rung Execution

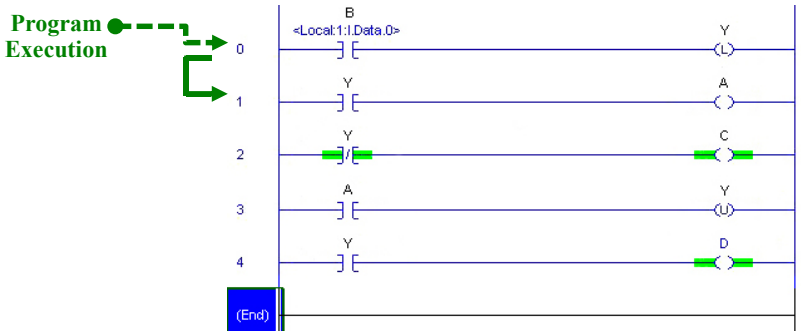
**Thus, beginning with Rung 0, the controller:**

- **Determines the Rung-Condition based on the state of the rung's Logic Instructions, and**
- **Completes any required tasks based on the Rung-Condition and the Output Instruction(s) on that rung.**




# Ladder Diagram Order of Execution

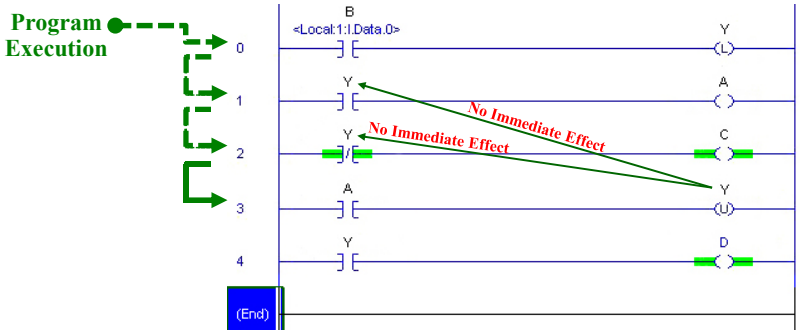
**Once the controller completes the execution of a rung, it then moves to and executes the next rung by determining the Rung Condition on that rung and then applying the results to that rung's Output Instruction(s).**



# Ladder Diagram Order of Execution

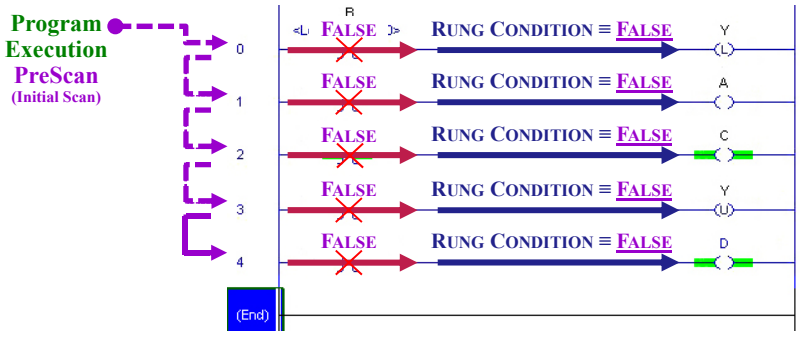
**Although the execution of a rung may affect the state of the Logic Instructions on any previously-executed rungs, those changes are not acted upon until those rungs are re-executed.**

**I.e. – Once executed, a rung will not be re-evaluated until after the entire ladder has been executed and the controller sequentially steps through to that rung again.**

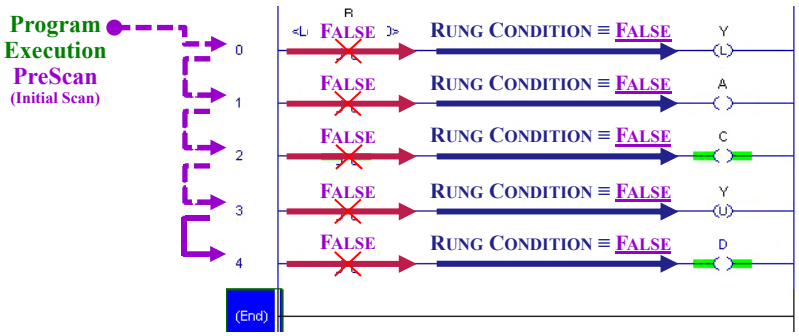



# PRESCAN Mode

**Note that, if the PLC transitions from PROGRAM → RUN mode or if the PLC powers up in RUN mode, the first (initial) scan of the ladder diagram is executed in PRESCAN mode, during which all Logic Instructions return a FALSE state, in-turn resulting in all FALSE Rung Conditions.**



# PRESCAN Mode




# Ladder Logic Instructions

**Thus, the operation of the following three, commonly used, Ladder Logic Instructions will now be presented:**

- **the Examine if Closed (XIC) instruction,**
- **the Examine if Open (XIO) instruction, and**
- **the Output Energize (OTE) instruction.**

**Note – an analogy is often made between the look/operation of these instructions and that of a relay's Normally-Open (NO) contact, Normally-Closed (NC) contact, and Field Coil.**

**Although this analogy may be utilized during the associated lecture for this material, the analogy will NOT be discussed within this presentation because it can lead to several common misconceptions regarding their operation within a Ladder Diagram; instead, they will always be presented and discussed as "Instructions within a Ladder Logic Program".**

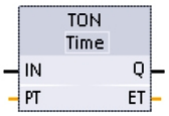
# Icons Used for Ladder Instructions

**Note that the icon shown in this presentation for a various instruction will be the version displayed within the RSLogix (Allen-Bradley / Rockwell Software) environment.**

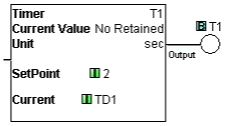
**Although other manufacturers (Siemens, Automation Direct, etc.) may use different icons, the overall operation of the various instructions should be consistent across all of the platforms.**

**Shown below are some of the various Icons utilized for an "On-Delay Timer" in a Ladder Diagram:**


RSLogix (Allen-Bradley)




Siemens




Automation Direct

# Logic Instructions – XIC

**The XIC (Examine If Closed) instruction:**


**is a Logic Instruction that takes on either a TRUE or FALSE state depending on the value stored in a bit of memory.**

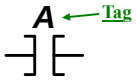
**But, in order to define the specific bit of memory upon which the XIC's state is based, the XIC must be assigned a Tag.**

**An XIC is a Boolean instruction because it can only take on one of two states, TRUE or FALSE.**

# Tags

**Tags** contain information that identifies and characterizes data stored in memory, allowing that data to be linked to the operation of one or more specific instructions.

For example, if the following XIC is assigned tag "A":




The tag is displayed immediately above the icon for the instruction.

then Tag "A" identifies a specific bit in memory upon which this XIC's state is based.

Tag "A" is a Boolean tag because it addresses a single bit that can only take on one of two values, 0 or 1.

# Logic Instructions – XIC

If tag "A" is assigned to the XIC (Examine If Closed) instruction:


then the logic state (TRUE or FALSE) of that instruction is defined as follows:

**When evaluated:** (by the controller)

If bit A=1, then XIC-A ➡ TRUE

If bit A=0, then XIC-A ➡ FALSE

The XIC is TRUE when A=1.

# Logic Instructions – XIO

The XIO (Examine If Open) Logic Instruction:


An XIO is also a Boolean instruction.

takes on the opposite state compared to that of an XIC.

If tag "B" is assigned to the XIO, then the logic state of this instruction (TRUE or FALSE) is defined as follows:

When evaluated: (by the controller)

If bit B = 0, then XIO-B ➡ TRUE

If bit B = 1, then XIO-B ➡ FALSE

The XIO is TRUE when B≠1.

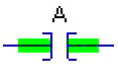
# Instructions – Icons vs. Current State

When a PLC is in RUN mode and its ladder diagram is being displayed (in real time) within the RSLogix environment:

Thus, given an XIC (

), it will be displayed as follows by the RSLogix software depending on its current logic state:

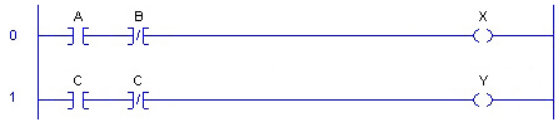

(when the XIC is FALSE)




(when the XIC is TRUE)

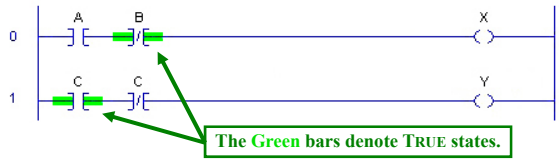
# Instructions – Icons vs. Current State

**For example, given the bit values: A = 0 B = 0 C = 1**
**and the following rungs:**




**If displayed while the PLC is in RUN mode, then:**

**Rung-0: XIC-A ➡ FALSE, XIO-B ➡ TRUE**
**Rung-1: XIC-C ➡ TRUE, XIO-C ➡ FALSE**




# Output Instructions – OTE

**The following Output Instruction is an OTE (Output Energize) instruction and it has been assigned tag "C":**


**An OTE will either set or reset (store a 1 or 0 in) the bit identified by its assigned tag based on the Rung Condition as follows:**

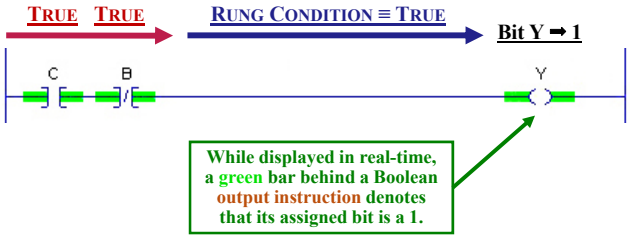
**If the Rung Condition is TRUE, then OTE-C sets bit C ➡ 1**
**(when evaluated by the controller)**
**If the Rung Condition is FALSE, then OTE-C resets bit C ➡ 0**

**If a bit is "set", it is changed to a one (1).**
**If a bit is "reset", it is changed to a zero (0).**

# OTE – TRUE Rung Condition

If the Rung-Condition for an OTE is TRUE, then the OTE will set its assigned bit to 1.

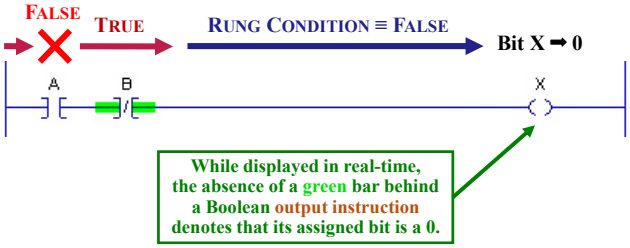
Thus, given the bit values: A = 0 B = 0 C = 1
when the following rung is executed, both XIC-C and XIO-B will be evaluated TRUE, resulting in a TRUE Rung Condition, in-turn causing the OTE to set bit Y ➡ 1.




# OTE – FALSE Rung Condition

If the Rung-Condition for an OTE is FALSE, then the OTE will reset its assigned bit to 0.

Thus, given the bit values: A = 0 B = 0 C = 1
when the following rung is executed, both XIC-A will be evaluated FALSE, resulting in a FALSE Rung Condition, in-turn causing the OTE to reset bit X ➡ 0.



# OTE with Hold-In Example (Part 1)

Part 1 of Hold-In Example

Look closely at the following rung from a ladder diagram:


If bits A, B, and C are all zero (A=B=C=0) when the rung is executed, XIC-B and XIC-C will both be evaluated FALSE, resulting in a FALSE Rung Condition.

Since bit C was already a 0, it isn't changed.

Since the Rung Condition is FALSE, OTE-C resets bit C ➡ 0, and since bit C=0, the results will be the same the next time the rung is executed provided bit B remains zero (B=0).

# OTE with Hold-In Example (Part 2)

Part 2 of Hold-In Example

Given the same rung, what if bit B changes to a one (B ➡ 1) and then the rung is executed again (assuming A=C=0)?

Then, the next time the rung is executed, XIO-A and XIC-B will both be evaluated TRUE, resulting in a TRUE Rung Condition, and…


since the Rung Condition is now TRUE, OTE-C sets bit C ➡ 1.


*(continued on the next slide)*

# OTE with Hold-In Example (Part 2)

Part 2 of Hold-In Example (continued)

Note that, although bit C is now one (C=1), XIC-C is still shown to be FALSE (no green bar).


This is to highlight the fact that, once the state of a specific Logic Instruction is evaluated, it will not be re-evaluated until the rung is executed again.

But, once the rung is executed again, assuming that A and B remain unchanged, the rung will appear as:


# OTE with Hold-In Example (Part 3)

Part 3 of Hold-In Example

Now, given the current state of the rung (A=0, B=C=1), what if bit B resets (B➡0) and then the rung is executed again?

Although XIC-B now evaluates FALSE, XIO-A and XIC-C still provide a TRUE path, maintaining the TRUE Rung Condition, and…


since the Rung Condition stays TRUE, OTE-C keeps bit C set.

# OTE with Hold-In Example (Part 4)

Part 4 of Hold-In Example

But, given the current state of the rung (A = B = 0, C = 1), what if bit A changes to a one (A ➡ 1) and the rung is executed again?

The next time the rung executes, XIO-A will be evaluated FALSE, resulting in a FALSE Rung Condition, and


since the Rung Condition is FALSE, OTE-C resets bit C ➡ 0.


*(continued on the next slide)*

# OTE with Hold-In Example (Part 4)

Part 4 of Hold-In Example (continued)

Note that, although C is now a zero (C = 0), XIC-C was still shown to be TRUE (with a green bar).


This is also to highlight that the state of the Logic Instruction is not re-evaluated until the rung is executed again.

But, if the rung is executed again and XIC-C is re-evaluated, the rung will appear as (assuming A and B remain unchanged):

# OTE with Hold-In Example (Part 5)

Part 5 of Hold-In Example

And finally, what if bit A resets (A ➡ 0) before the rung executes again?

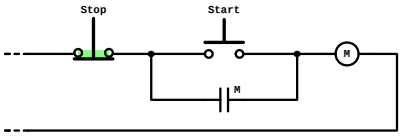
The next time the rung executes, XIO-A will be evaluated TRUE, but the Rung Condition will remain FALSE .


Thus, the rung is back to the same overall state at that existed at the beginning of this example.

# OTE with Hold-In Example (Analysis)

Analysis of Hold-In Example

Does the operation of the rung in this example remind you of the operation of a basic stop-start motor controller?


This rung-structure is often used in a PLC-based control system for which a "hold-in" function is required.

The trick is to somehow associate the value of bit A with the state of a "Stop" button, the value of bit B with the state of a "Start" button, and the value of bit C with the state of a contactor's field coil.

How this is done will be discussed in Part B of this presentation.

# Ladder Logic Instructions

**Although many programming tasks can be completed using just the following three Instructions:**

- ⊣ ⊢ **the Examine if Closed (XIC) instruction,**
- ⊣/⊢ **the Examine if Open (XIO) instruction, and**
- ( ) **the Output Energize (OTE) instruction.**

**a handful of other instructions will also be presented in order to provide a solid foundation of Ladder Logic knowledge upon which the required programming tasks for a wide variety of PLC-based motor control systems can be implemented.**

# Output Latch (OTL)

**OTL – Output Latch**

B
(L)

**An Output Latch (OTL) is an Output Instruction that can set a bit (B ➡ 1), but cannot reset a bit back to zero.**

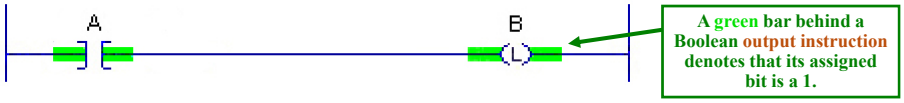
*** – It is often stated that an OTL "latches" a bit. The problem with this statement is that, in electronic circuits, a latch is a device that "sets and holds" a voltage at a specific level. It is true that the OTL can "set" a bit, but it doesn't "hold" the bit at a value of one. Instead, it doesn't have the ability to "reset" a bit, so even if the Rung Condition returns FALSE, the bit will remain set until it is actively reset by another instruction.**

# OTL Example

**Given the following rung that contains on OTL (Output Latch):**


**If bit A changes to a one (A = 1) before the rung executes, then when the rung is executed, XIC-A will be evaluated TRUE, resulting in a TRUE Rung Condition.**


**Since the Rung Condition is TRUE, OTL-B sets bit B ➡ 1.**




# OTL Example

**On the other hand, given the following rung:**


**If bit A resets (A = 0) before the rung executes, then when the rung is executed, XIC-A will be evaluated FALSE, resulting in a FALSE Rung Condition.**


**But, OTL-B does nothing when the Rung Condition is FALSE, so bit B remains set (B = 1).**

# Output Unlatch (OTU)

**OTU – Output Unlatch**


**An Output Unlatch (OTU) is an Output Instruction that can reset a bit (B ➡ 0), but cannot set a bit to a one.**

* – Note that the OTU resets a bit when the Rung Condition becomes TRUE. On the other hand, an OTE resets a bit when the Rung Condition becomes FALSE. This is a critical distinction between the abilities of OTUs and OTEs to reset a bit, and often provides a challenge for beginning Ladder Logic programmers.

# OTU Example

**Given a rung that contains on OTU (Output Unlatch) (A = 0, B = 1):**


**If bit A changes to a one (A = 1), then the next time that the rung is executed, XIC-A will be evaluated TRUE, resulting in a TRUE Rung Condition, and**


**since the Rung Condition is TRUE, OTU-B resets bit B ➡ 0.**

# OTU Example

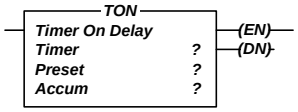
Given the same rung when A=1 and B=0:


If bit A resets (A=0), then the next time that the rung is executed, XIC-A will be evaluated FALSE, resulting in a FALSE Rung Condition.


But, OTU-B does nothing when the Rung Condition is FALSE, so bit B remains reset (B=0).

# On-Delay Timer (TON)

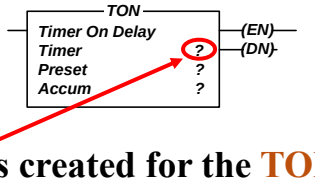
TON – On-Delay Timer




The On-Delay Timer (TON) is an Output Instruction that functions as a "non-retentive" timer.

# TON – Base Tag & Sub-Tags

## TON – On-Delay Timer



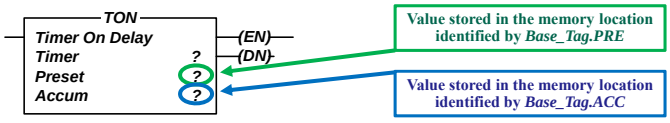

**When a Base Tag is created for the TON, the name of which appears in the *Timer* field, several sub-Tags are automatically created by the RSLogix software:**

**The sub-Tags include:**

| | | |
|---|---|---|
| ***Base_Tag.PRE*** | ***Base_Tag.EN*** | ***Base_Tag.TT*** |
| ***Base_Tag.ACC*** | ***Base_Tag.DN*** | |

# TON – Preset & Accumulator

## TON – On-Delay Timer



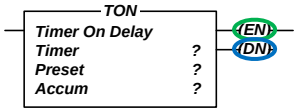

**In addition to the *Timer* field that contains the Base Tag name, there are two user-defined fields shown in the icon:**

- ***Preset*** **– Contains the time-delay value (specified in msec) up to which the TON will count.**
  (*.PRE*)
- ***Accum*** **– Contains the initial time value (also in msec).**
  (*.ACC*)

**Note that the *Accum* field displays the current value stored in the accumulator when viewed "online" (in real time).**

# TON – Operational State Bits

## TON – On-Delay Timer



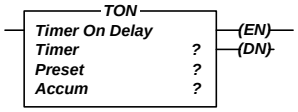

The .*TT* bit is not shown on the TON icon

**Three sub-Tags characterize the TON's operational state:**

- ***Base_Tag.EN*** – The ***.EN*** (***Enable***) bit is set to 1 when the TON is enabled, and reset to 0 when it's disabled.
- ***Base_Tag.DN*** – The ***.DN*** (***Done***) bit is set to 1 when the *Accum* equals the *Preset*, otherwise it's reset to 0.
- ***Base_Tag.TT*** – The ***.TT*** (***Time Transitioning***) bit is set to 1 when the TON is actively counting, otherwise it's reset to 0.

# TON – Documentation

## TON – On-Delay Timer




**The following information can be found by using the Help tab within the RSLogix 5000 software:**

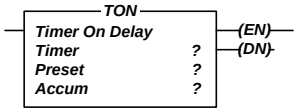
| Instruction: | Relay Ladder: | Function Block: | Structured Text: | Description: |
|---|---|---|---|---|
| TON<br>Timer On Delay |  | see TONR | see TONR | The TON instruction is a non-retentive timer that accumulates time when the instruction is enabled (rung-condition-in is true). |

| Operand: | Type: | Format: | Description: |
|---|---|---|---|
| Timer | TIMER | tag | timer structure |
| Preset | DINT | immediate | how long to delay (accumulate time) |
| Accum | DINT | immediate | number of msec the timer has counted; initial value is typically 0 |

| Arithmetic Status Flags: | Major Faults: | | |
|---|---|---|---|
| not affected | Type 4 | Code 34 | • .PRE < 0<br>• .ACC < 0 |

# TON – Detailed Operation

## TON – On-Delay Timer

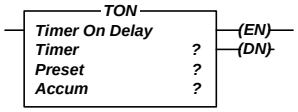



**The TON is "enabled" when its Rung Condition is TRUE.**

**Once enabled, its *Enable* bit is set (*.EN* ➡ 1) and the TON begins "actively" counting (accumulating time).**

# TON – Detailed Operation

## TON – On-Delay Timer



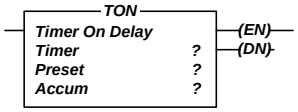

**As long as the TON remains enabled, its accumulator will continue incrementing until it reaches the *Preset* value.**

**Note that, while the TON is "actively" accumulating time, the *Time Transitioning* bit will be set (*.TT* ➡ 1).**

**But, if the TON is not accumulating time, either because it's disabled or its accumulator has reached the *Preset* value, then the *.TT* bit will be reset (*.TT* ➡ 0).**

# TON – Detailed Operation

## TON – On-Delay Timer



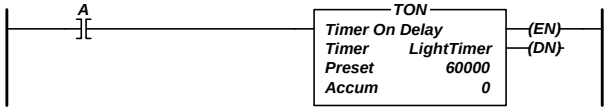

**Once the TON's accumulator reaches the *Preset* value, it will stop incrementing and the *Accum* value will remain equal to the *Preset* value as long as the timer is enabled.**

**Additionally, as long as *.ACC* = *.PRE*, the timer's *Done* bit will be set (*.DN* ➡ 1).**

**But, if for any reason *.ACC* ≠ *.PRE*, such as the timer being disabled or reset, the *Done* bit will be reset (*.DN* ➡ 0).**

# TON Example

**Given the TON that has been placed on a rung with XIC-A:**




**If the timer is configured as follows:**

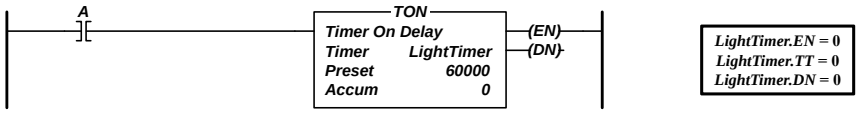
- **the name of the Base Tag is "LightTimer",**
- **the *Preset* value is 60,000 (60,000 msec = 60 sec), and**
- **the initial *Accum* value is 0.**

**describe, in detail:**

**1) the timer's initial conditions while bit A = 0,**
**2) the operation of the timer when bit A is set (A ➡ 1), and**
**3) the operation of the timer if bit A is reset (A ➡ 0).**

# TON Example – Initial Conditions

**Given the TON that has been placed on a rung with XIC-A:**




**If bit A is initially zero (A=0), then:**

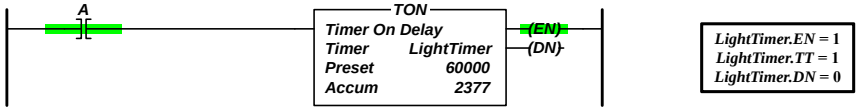
- **both XIC-A and the Rung Condition will be FALSE, and**
- **the TON will be disabled.**

**Since the timer is disabled:**

- **the status bits, *Enable*, *Time Transitioning*, and *Done,* will all be zero (.*EN*=0, .*TT*=0, .*DN*=0), and**
- **the accumulator value will remain at zero (.*ACC*=0).**

# TON Example – Timer Enabled

**Given the TON that has been placed on a rung with XIC-A:**



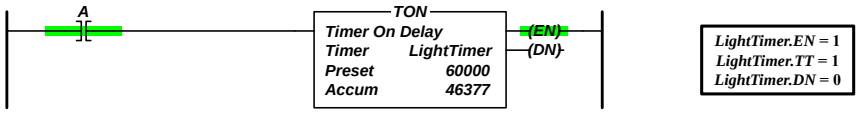

**If bit A ➡ 1, then the next time the rung is executed, XIC-A will be evaluated TRUE, making the Rung Condition TRUE, and the TON will be enabled.**

**When the timer is enabled:**

- **the *Enable* bit will be set (.*EN* ➡ 1),**
- **the accumulator will begin incrementing, and**
- **the *Time Transitioning* bit will be set (.*TT* ➡ 1).**

# TON Example – Time Transitioning

**Given the TON that has been placed on a rung with XIC-A:**



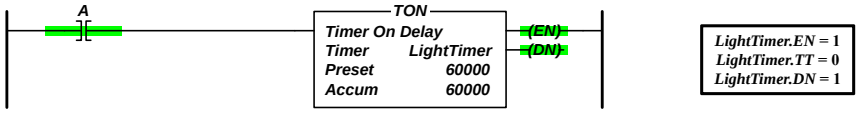

**As long as the TON remains enabled and the *Accum* value is less than the Preset value (*.ACC* < 60,000), then accumulator will keep incrementing until it reaches the *Preset* value.**

**Thus, while the accumulator is incrementing:**

- **the *Enable* bit will remain set (*.EN* = 1), and**
- **the *Time Transitioning* bit will remain set (*.TT* = 1).**

# TON Example – Timer Done

**Given the TON that has been placed on a rung with XIC-A:**

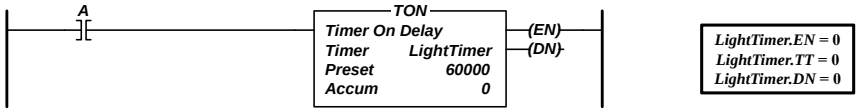



**When the *Accum* value reaches the Preset value (*.ACC* = 60,000), which will occur 60 seconds after the timer is enabled:**

- **the accumulator stops incrementing and remains at 60,000,**
- **the *Enable* bit will remain set (*.EN* = 1),**
- **the *Time Transitioning* bit will reset (*.TT* ➡ 0), and**
- **the *Done* bit will be set (*.DN* ➡ 1).**

# TON Example – Timer Done

**Given the TON that has been placed on a rung with XIC-A:**




**At any point in time, if bit A ➡ 0, then the next time the rung is executed, XIC-A will be evaluated FALSE, making the Rung Condition FALSE, and the TON will be disabled.**

**If the timer is disabled:**

- **the *Enable* bit will be reset (*.EN* ➡ 0),**
- **the accumulator will reset back to its initial value,**
- **the *Done* bit will be reset (*.DN* ➡ 0), and**
- **the *Time Transitioning* bit will be reset (*.TT* ➡ 0) if it was set.**

# Reset (RES)

**RES – Reset**

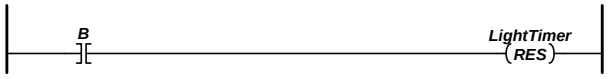

**The Reset instruction is an Output Instruction that can be used to reset the accumulator (.ACC) and the status bits (.EN, .DN, etc.) of a timer or counter.**

**The Base Tag (name) associated with either the timer or counter that the RES instruction is being used to reset must be placed in the Reset's tag field.**

**For example, a RES instruction could be used to reset LightTimer's accumulator if assigned the tag LightTimer.**

# RES – Operation

**RES – Reset**



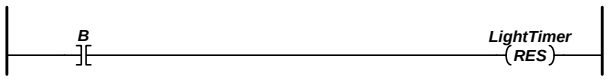

**The operation of the Reset instruction is based upon the Rung Condition on the rung.**

**When the rung is executed:**

**If the Rung Condition is TRUE, the RES will reset the accumulator of its associated timer (counter) back to its initial value, but**

# RES – Operation

**RES – Reset**

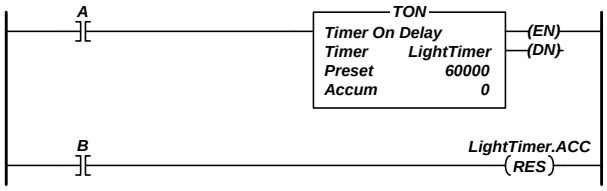



**Note that:**

**Although a RES instruction does not technically disable a timer (counter) since the timer's rung-condition is not affected when the timer is reset, it causes roughly the same effect as the timer being instantaneously disabled and then re-enabled during its next scan.**

# RES Example (with TON)

Given the following rungs that contain a TON and a RES:

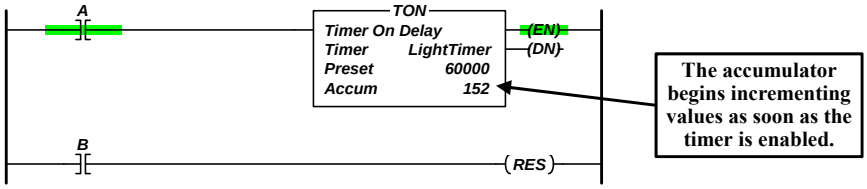



both of which are in the initial state shown in the figure.

If bits A and B are both zero (A = B = 0), in-turn making the Rung Conditions for both rungs FALSE, then the timer is disabled and the Reset instruction does nothing.

# RES Example – TON Enabled

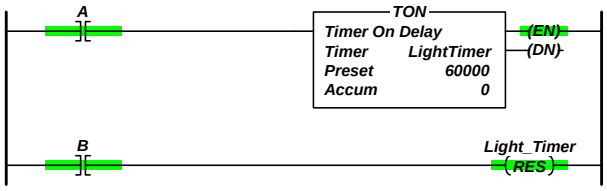
Given the following rungs that contain a TON and a RES:




If bit A ➡ 1
- ⇨ XIC-A ≡ TRUE
- ⇨ the Rung Condition for the 1st rung is TRUE
- ⇨ TON-LightTimer is enabled
- ⇨ the *Enable* bit is set (*.EN* ➡ 1)
- ⇨ the timer's accumulator begins incrementing
- ⇨ the *Time Transitioning* bit is set (*.TT* ➡ 1)

# RES Example – TON Reset

Given the following rungs that contain a TON and a RES:




If bit B ➡ 1 ⇨ XIC-B ≡ TRUE

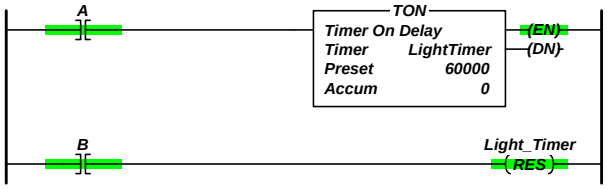
⇨ the Rung Condition for the 2nd rung is TRUE

⇨ TON-LightTimer resets LightTimer's accumulator to its initial value

⇨ If the *Accum* was 60,000 (and *.DN* = 1) when the timer was reset, the *Done* bit is reset (*.DN* ➡ 0)

# RES Example – TON Reset

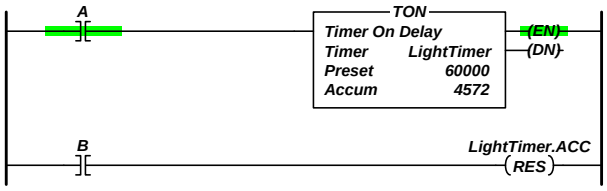
Given the following rungs that contain a TON and a RES:




If bit B remains set (B = 1), then the RES will keep resetting the accumulator back to its initial value and it will appear as if the timer is disabled despite its TRUE Rung Condition.

# RES Example – TON Reset

**Given the following rungs that contain a TON and a RES:**




**But, if bit B is reset (B ➡ 0) while LightTimer is still enabled, then the timer's accumulator will begin incrementing again until either it reaches the *Preset* value, the timer is reset again, or the timer is disabled.**

# Numerical Comparison Instructions

**There are many Logic Instructions available the return either TRUE or FALSE states based on a comparison of two values.**

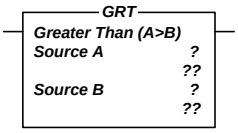
**These instructions include:**

- **GRT – Greater Than**
- **GEQ – Greater Than or Equal To**
- **LES – Less Than**
- **LEQ – Less Than or Equal To**
- **EQU – Equal To**
- **NEQ – Not Equal To**

**Only the Greater Than (GRT) instruction is covered in this presentation since the other comparison-type instructions function similarly.**

# Greater Than (GRT)

**GRT – Greater Than**

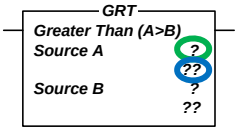



**The GRT (Greater Than) instruction is used to compare the values of two numbers, A and B.**

- **If A > B, then the GRT returns a TRUE state.**
- **If A ≤ B, then the GRT returns a FALSE state.**

# GRT – Configuration

**GRT – Greater Than**

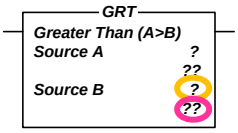



**The GRT (Greater Than) instruction is used to compare the values of two numbers, A and B.**

- **Source A – This field contains either the numerical value of A or the tag name relating to the data that contains the value of A.**
- **If the Source A field contains a tag name, the second field shows the value currently stored in that Tag.**

# GRT – Configuration

## GRT – Greater Than

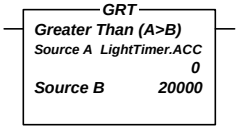



**The GRT (Greater Than) instruction is used to compare the values of two numbers, A and B.**

- **Source B – This field contains either the numerical value of A or the tag name relating to the data that contains the value of B.**
- **If the Source B field contains a tag name, the second field shows the value currently stored in that Tag.**

# GRT Example

**The GRT shown above is configured to compare the current value stored in LightTimer's accumulator to 20,000.**



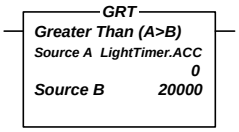

**When evaluated:**

**The GRT return a TRUE state whenever *LightTimer.ACC* is greater than 20000, and**

**The GRT return a FALSE state whenever *LightTimer.ACC* is less than 20000.**

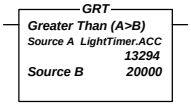
# GRT Example

**The GRT shown above is configured to compare the current value stored in LightTimer's accumulator to 20,000.**

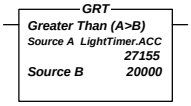



**Thus:**

- **While *LightTimer.ACC* is < 20000, the GRT returns a FALSE LOGIC state.**

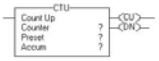



- **While *LightTimer.ACC* is > 20000, the GRT returns a TRUE logic state.**




# Other Instructions – Counter Up (CTU)

**The CTU (Counter Up) is an Output Instruction whose function is similar to that of the On-Delay Timer:**

| Instruction: | Relay Ladder: | Function Block: | Structured Text: | Description: |
|---|---|---|---|---|
| CTU Counter Up |  | see CTUD | see CTUD | The CTU instruction counts upward. |



| Operand: | Type: | Format: | Description: |
|---|---|---|---|
| Counter | COUNTER | tag | counter structure |
| Preset | DINT | immediate | how high to count |
| Accum | DINT | immediate | number of times the counter has counted; initial value is typically 0 |

| Arithmetic Status Flags: | Major Faults: |
|---|---|
| not affected | none |

**The Counter Up (CTU) is useful for keeping track of the number of specific events that occur, such as the number of items passing by an optical detector.**

**When the CTU's rung-condition becomes TRUE, the CTU's accumulator is incremented by one (1).**

**Note that, for the CTU to count up again, the CTU's rung-condition must first become FALSE and then TRUE again.**