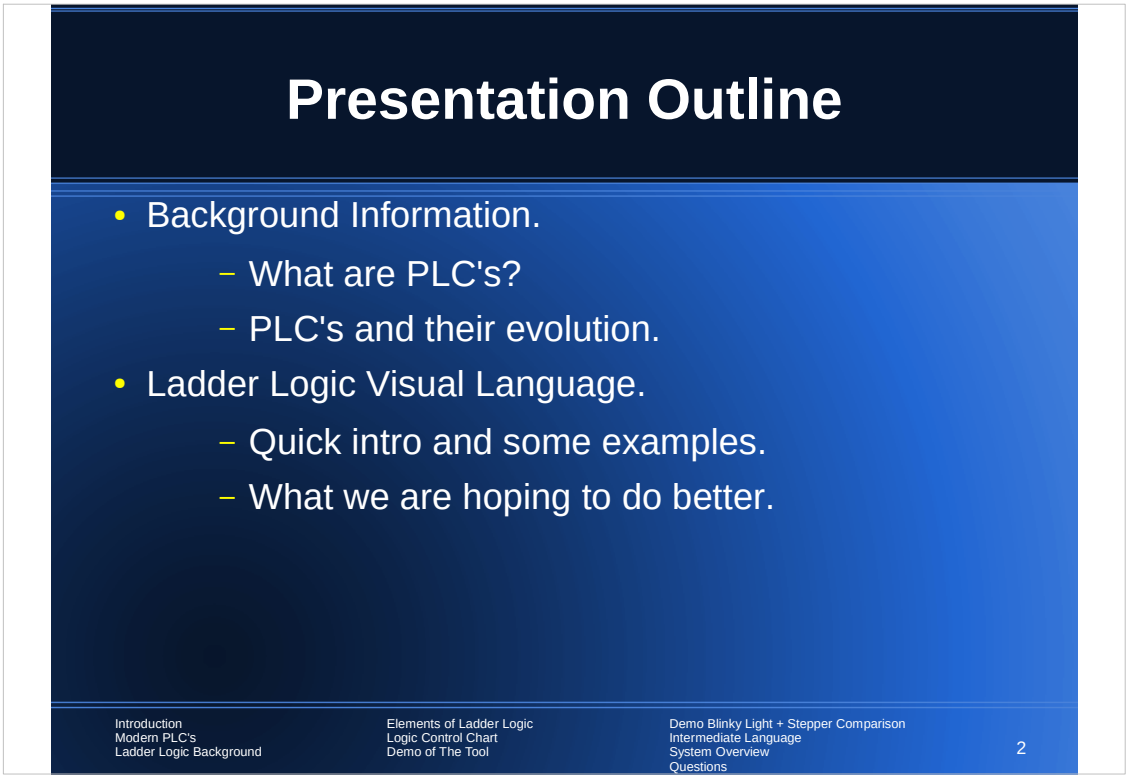
Good Afternoon everyone and thank you all for coming

In this seminar we will look at Programmable Logic controllers.

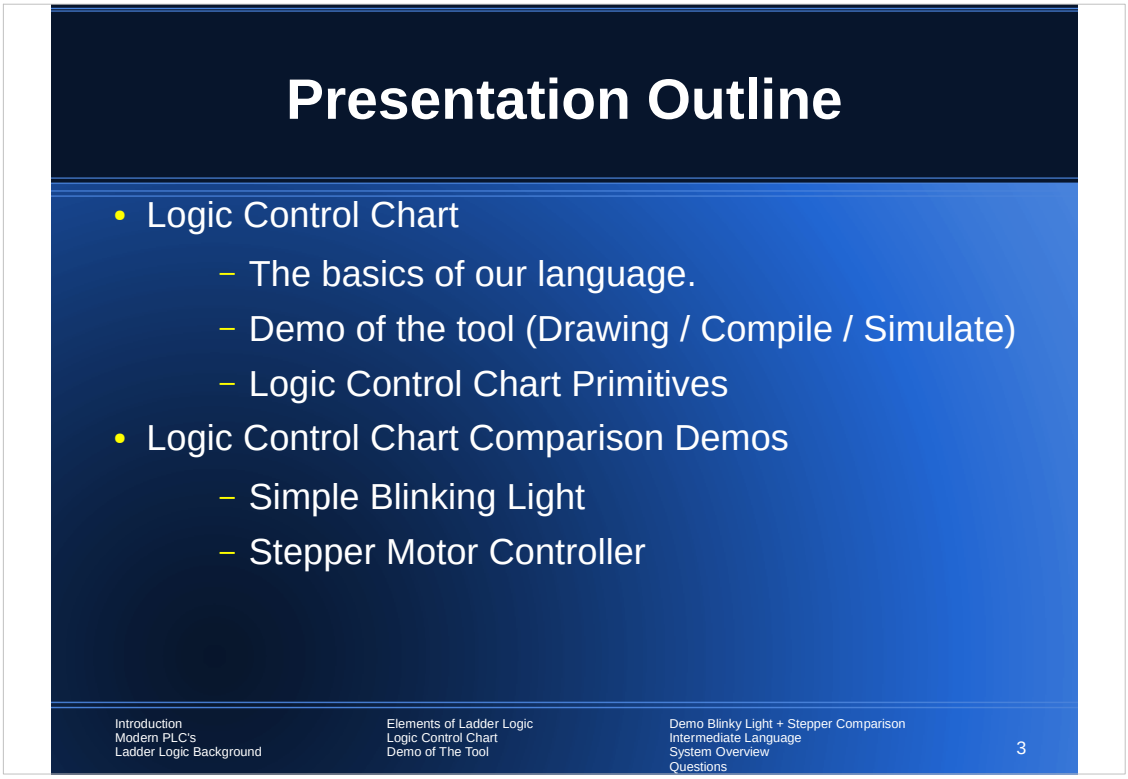
Programmable logic controllers predate micro-controllers and are widely used in heavy industrial sectors.



So first we'll look at what a PLC is, how they began their life, and how they are still used today.

We will then take a look at Ladder Logic the current visual metaphor which PLC's are programmed in.

After understanding the basics of ladder logic we'll look at our goals for Logic Control Chart to improve upon ladder logic.



Finally we'll take a look on Logic Control Chart the visual language developed in this thesis. Due to the time restrictions we will highlight some features only rather than take an exhaustive look at it.

Which will include a demo of the IDE and it's features.

Presentation Outline

- Hardware Abstraction
  - Hardware Support Framework
  - Intermediate Language
- Summary
  - Entire system and project overview.
- Questions

Finally if there is time we will go behind the scenes a little and show how we actually go from the visual metaphor to a pseudo assembly language. We refer to as the Intermediate Language.

Introduction to PLC's

- Programmable Logic Controllers have existed for over 30 years.
- Used widely in the manufacturing industry.

As I said before programmable logic controllers have existed for over 30 years. Today we can view then as a hardened micro-controller. But PLC's predate micro controllers.

Their usage over the years have been widely unchanged they are generally used in harsh environments in manufacturing industry. But you can also find them in your car.

A key distinction from micro-controllers is PLC's are usually designed to drive higher current loads directly where micro-controllers have to interface with another external motor driver.

Introduction to PLC's

- PLC's started life as a relay cabinet
- Noisy
- Failed often
- Hard to program
- Hard to isolate hardware failure

PLC's started as a relay cabinets. Since relay cabinets predated micro-controllers they also predated modern high current transistors.

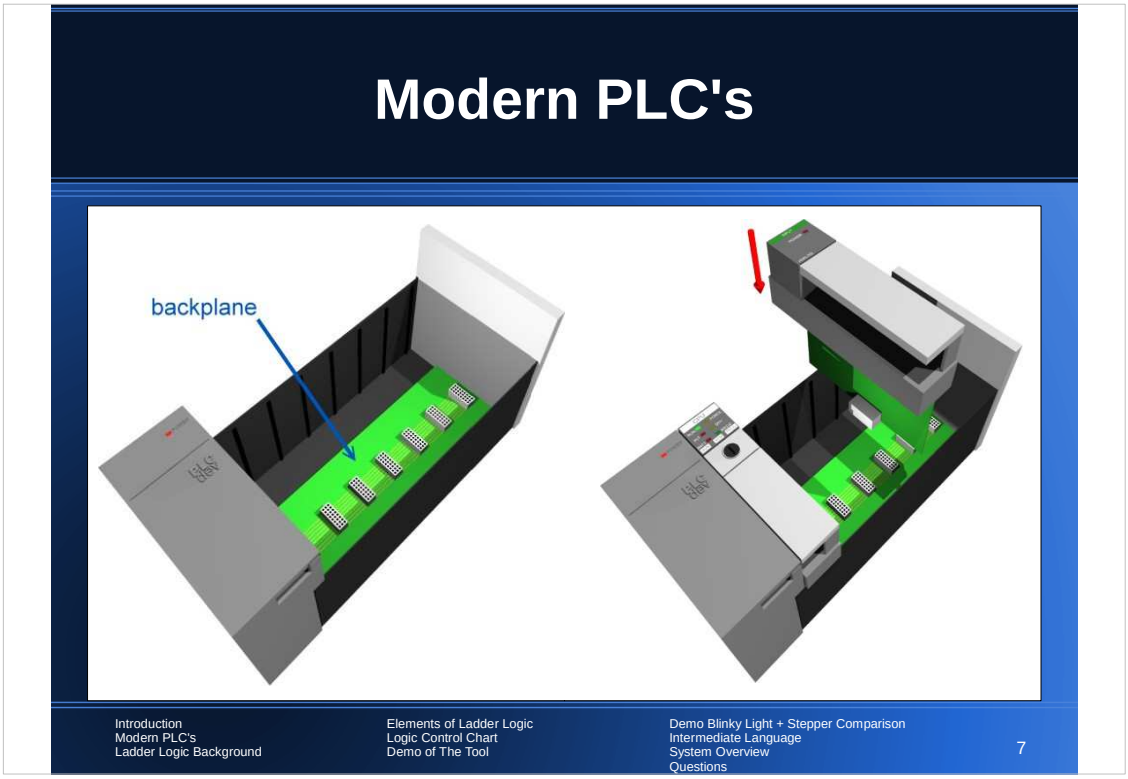
Relays are mechanical devices and because relays were used PLC's the PLC's at the time were known unreliable.

They were noisy.

When relays turned on and off they left an electrical arc which overtime eventually killed the relays.

They were extremely time consuming to program since you had to physically rewire your circuit. And because of failures you had no clue if the relay your re-using is actually still functional.

Very time consuming to figure out which relay actually failed.



When micro-controllers finally became available PLC's quickly migrated from being relay driven devices to micro-controller controlled.

Three different types of PLC quickly appeared on the market they are classified as the “Standard Modular PLC”, “Micro PLC” and the “Nano PLC”

The Standard PLC shown above consists of an outer shell which also houses the power supply. Modules are then slid in depending on the application required.

Modern PLC's

In this example we can see various input and output modules and a CPU added to the shell.

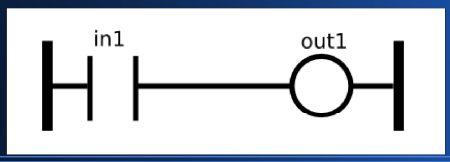
There are a variety of modules most manufacturers have added over time for convenience PLC's are also not high performance devices do modules like an encoder counted are necessary in order for the main CPU to not miss counts.

Micro PLC's contain all the same parts as a Standard PLC but the modules are baked in or integrated into the controller itself. This reduces cost and size to an extend but limits flexibility

Nano PLC's further reduce the size of the controller so they can be used in embedded applications. In my research I haven't found many people use Nano PLC's in this way since these days most prefer to just use micro-controllers to do this since they are cheaper.

Ladder Logic Background

- Ladder logic is an analogy to physical circuits.
- The left most and right most lines are power rails.
- Each circuit is wired on a horizontal “rung”.
- A circuit consists of series loads and exactly one output.
- Loads can be thought of as inputs and can be physical or logical.

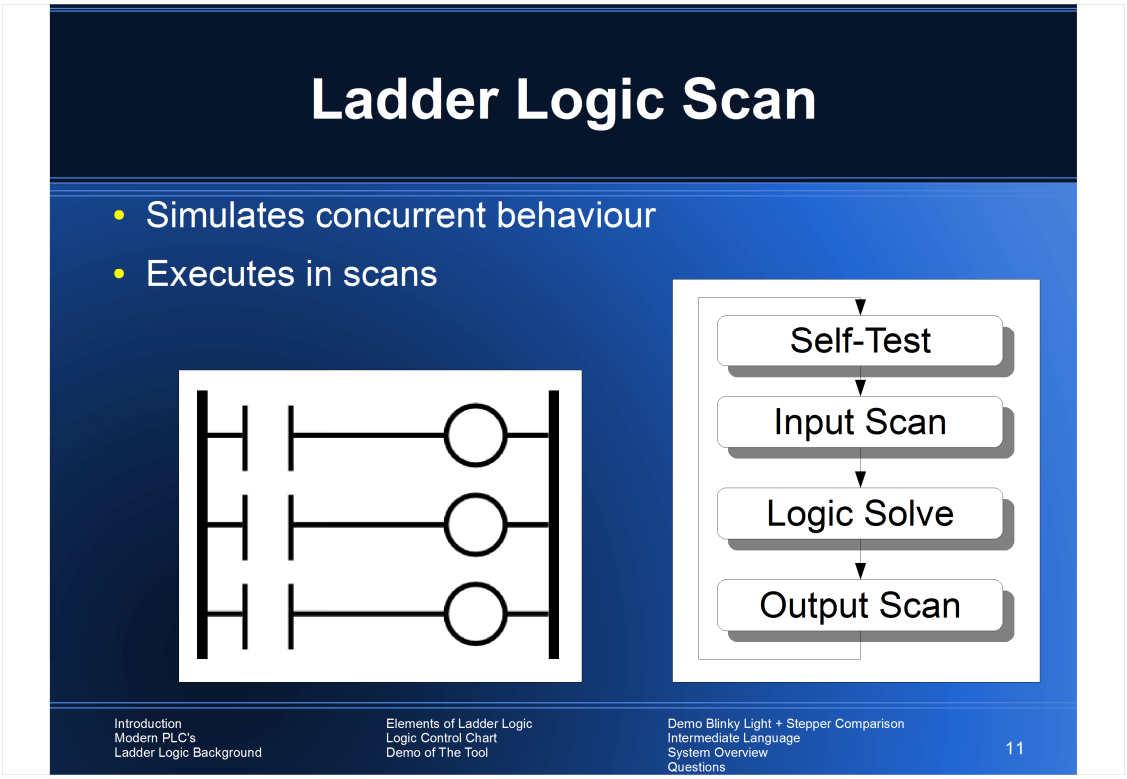


Even with all these different manufacturers PLC's all utilize ladder logic. Ladder Logic is at it's foundation based on the relay cabinets.

Two power rails run from the left and right of each “rung” (point to slide). Any number of “loads” or relays can be placed on the the left region of a rung (point to slide). One and only one output is placed on the right most area of each rung represented by a “circle” symbol.

Loads are either real and coming from the outside world. Or logical and used internal for logic not existing outside of the PLC itself.

Ladder Logic comes from circuits and relays it is read to be energized simultaneously (switch to the next slide).



That is in this example where we have three rungs there is no implicit order of execution rung 1, rung 2, and rung 3 in this example all are energized together.

There is also no delays that are modelled so outputs are understood to all happen at the same time regardless of how many loads are on the left hand side of them.

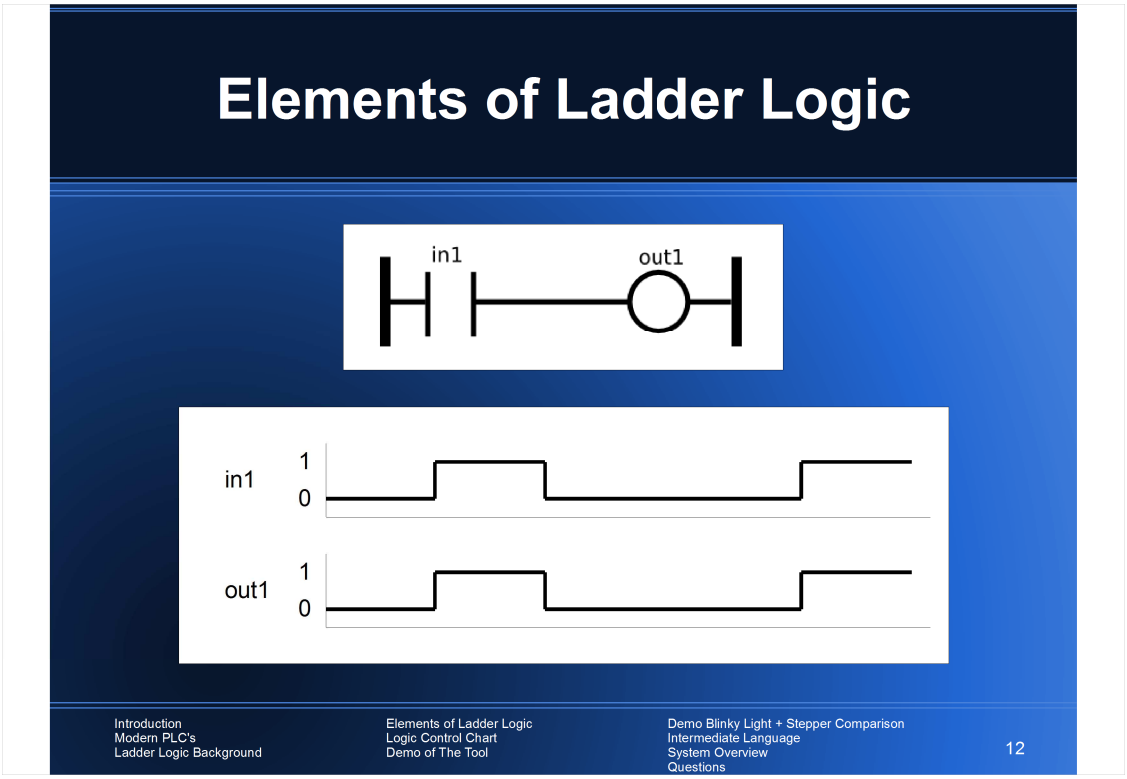
In order to achieve this type of simulated concurrent execution. PLC's executions occur in “SCANS”. A scan represents 4 distinct operations that are done to produce the next output.

The first operation is a communication self-test where the modules ensure they can successfully communicate with the CPU.

Following that an input scan reads all input pins connected to the PLC these inputs are latched and can be later read internally via registers.

In the logic solve stage the previously latched inputs are used in the ladder logic diagram to produce new outputs. These new outputs are not directly sent to the output pins but instead are set to temporary buffer registers until the output scan.

Finally during an output scan the previously buffered outputs are transferred to the pins in one step.



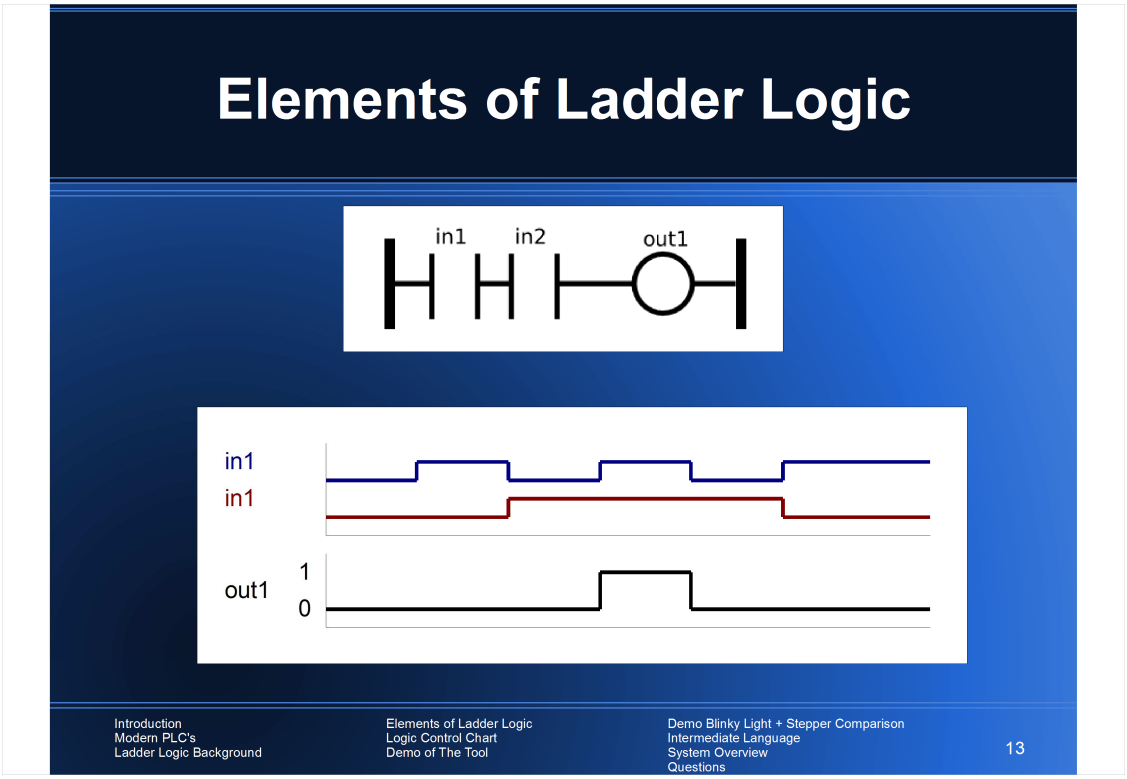
As we said before ladder logic represents circuits so the best way to describe them is with timing diagrams.

For the moment we will ignore the timing delays of the scans.

In this simplest case all we have is a simple relay marked in_1 and a simple output marked out_1. When the relay in_1 is closed the output out_1 immediately goes high, similarly when the input in_1 goes low the output out_1 follows and goes low.

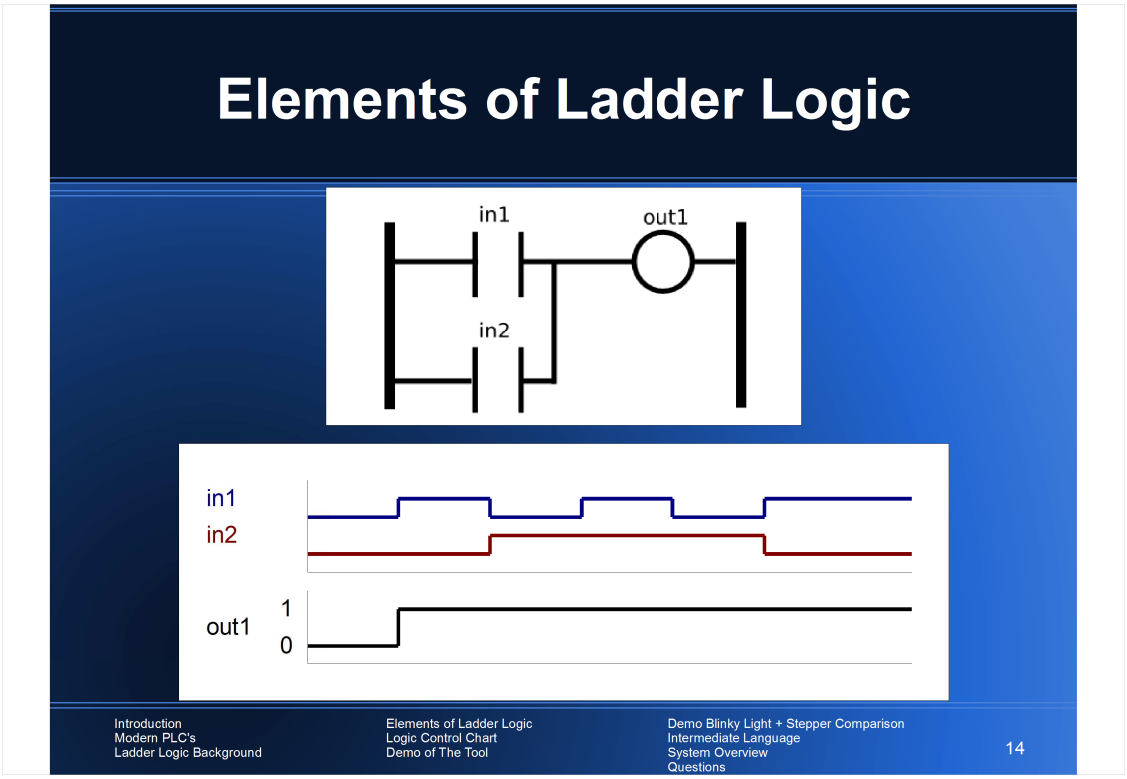
This is how ladder logic is understood to function. The reality is there is always a delay between the input scan and the output scan so the actual diagram for out_1 should be slightly shifted to the right.

However in most cases it is assumed that the PLC operates reasonably fast and a functional diagram as shown here (which ignores timing delays) is good enough to show the behaviour of the system.



So if you know how to make basic logic in circuits you should be right at home in ladder logic.

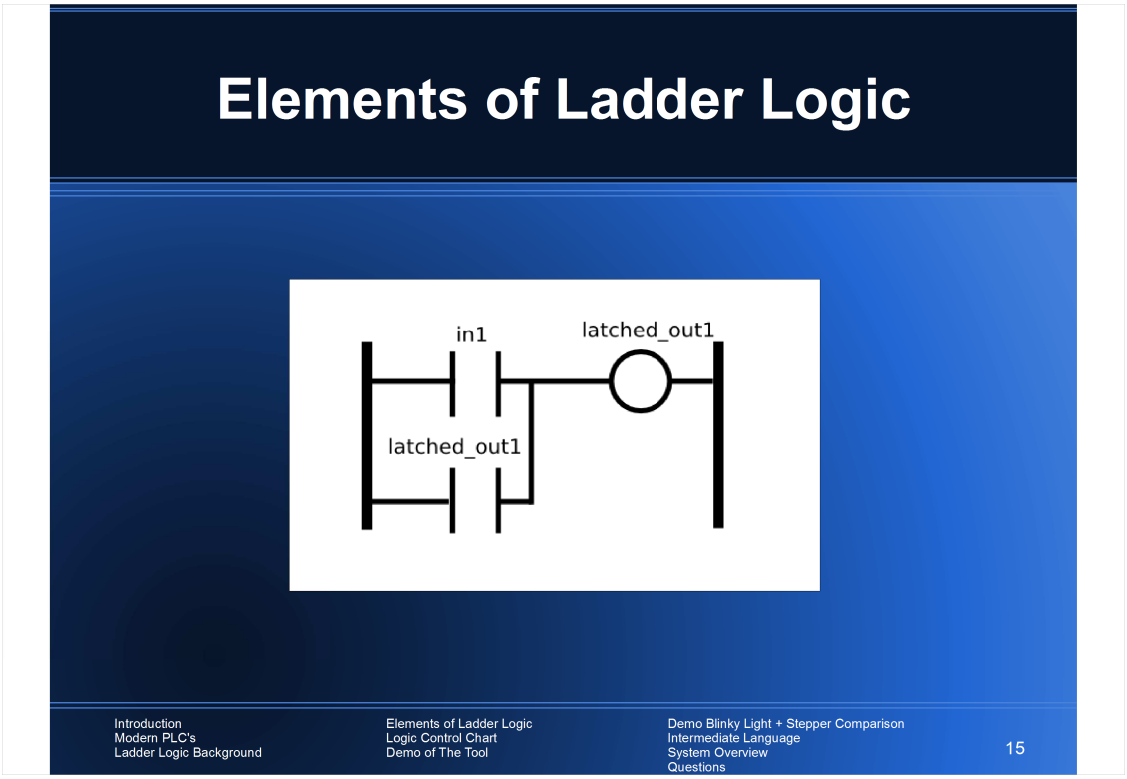
Here we have a basic logical “and” where both relays must be held closed in order to produce an output.



Similarly we can have logical “OR” gates in the same fashion.

In this case ladder logic does allow for many rungs to share a common output.

If either in_1 or in_2 is energized in this case we can see that out_1 is energized.

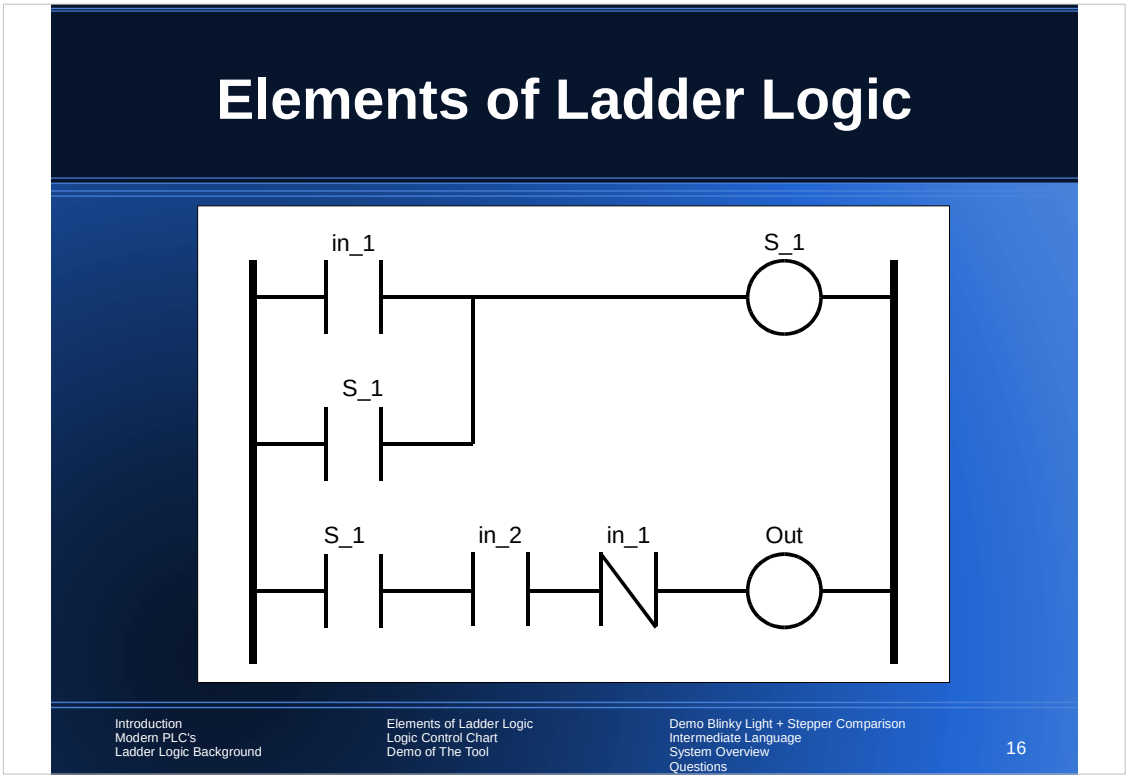


Because ladder logic is based on circuits it does have some elements that may not be immediately intuitive.

For example a common construct is if you want a output to stay on after an initial input first put it on.

Imagine a motor that you want to stay on after you press a button initially.

Here once latched_out1 is initially energized by in_1 it stays permanently on.



Now as programmers we know that it's often the case that we want A to happen before B to happen.

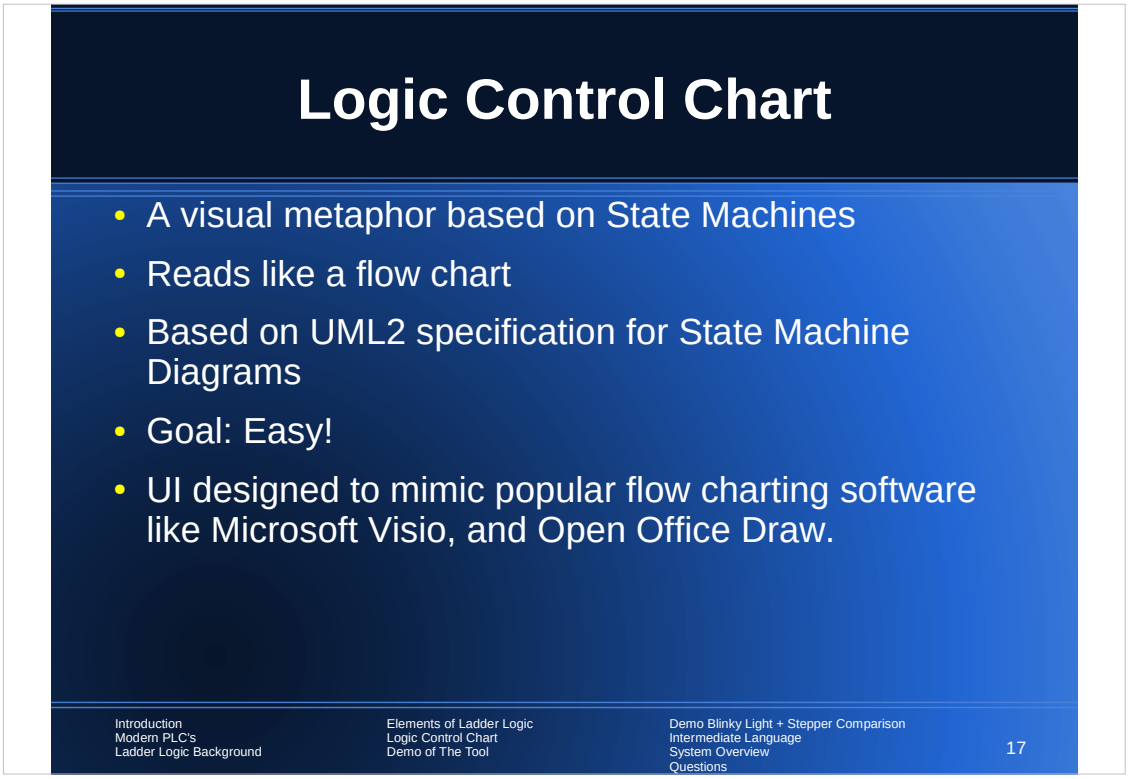
Suppose we wanted to express a simple circuit where our output “Out” only goes high if first in_1 is pressed then in_2.

We might also want to make sure that in_1 is not still held down when in_2 is pressed.

So now we have a basic diagram that enforces some form of sequencing. What happens is when in_1 is pressed S_1 goes high remembering that in_1 was pressed. When in_2 is pressed and in_1 is not pressed OUT is energized. If in_1 on the second rung is normally closed so in_1 has to go low to enable OUT.

In this example we can see where ladder logic is particularly weak. It's not good whenever we need to express a sequence of events. But programmers like us always think in sequences.

Sequencing in ladder logic always feels like a “hack” in order to get it to work.



So we felt that it was important for PLC's to maintain their visual programming. But we also felt that Ladder Logic has been vastly outdated.

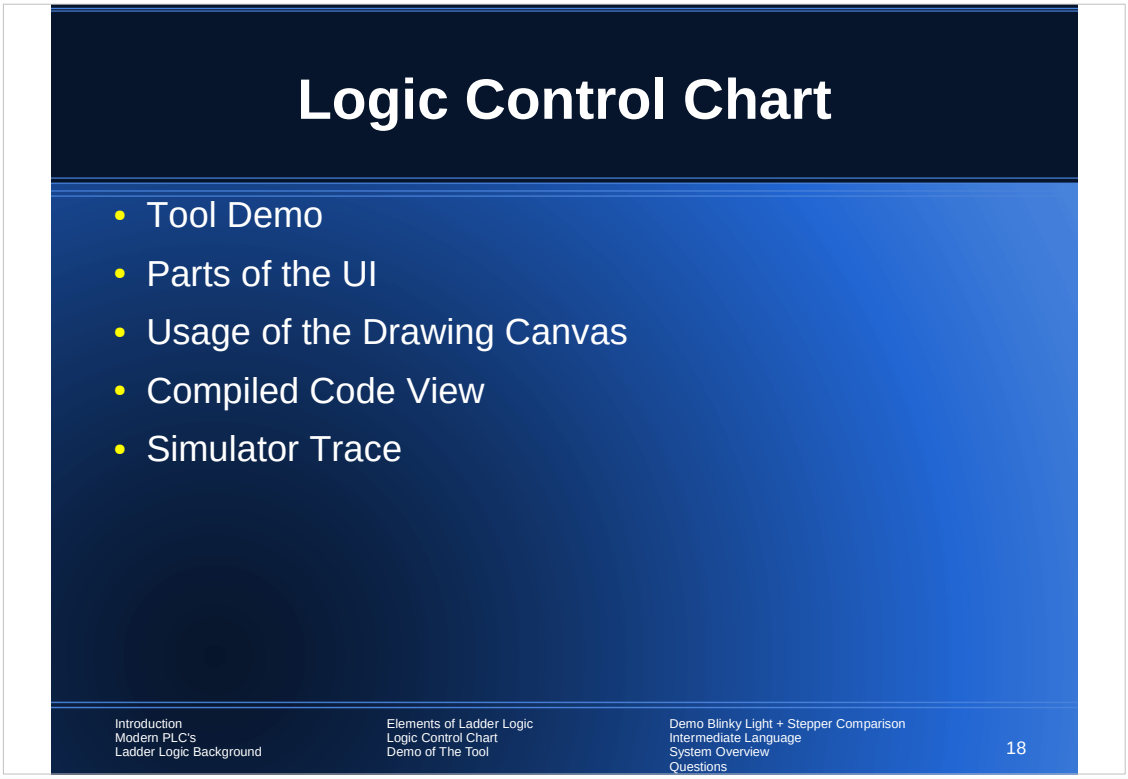
For one we wanted to bring in the ability of sequencing things easily since most programmers trained these days are used to the concept of doing things sequentially. Human beings in general seem to also solve problems in a sequential manner.

Logic control chart was designed to read much like a flow chart, as it was felt this universally understandable by all.

Logic control chart was loosely based off the UML2 specification for State Machine Diagrams it was rigorous enough to form our foundation and contained all the concepts I would need to build my own language. Obviously I had to modify a few things to bring it to the level of concreteness that I can compile actual useful code but overall it's pretty close.

Now I'll have a tool demo right after this slide but a main goal was to make things easier. This also goes for the actual IDE, there's no point of putting all this effort into creating a language that's super easy if the IDE is loaded with tools and foreign concepts.

So for that I looked to Microsoft Visio for inspiration on existing chart drawing tools.



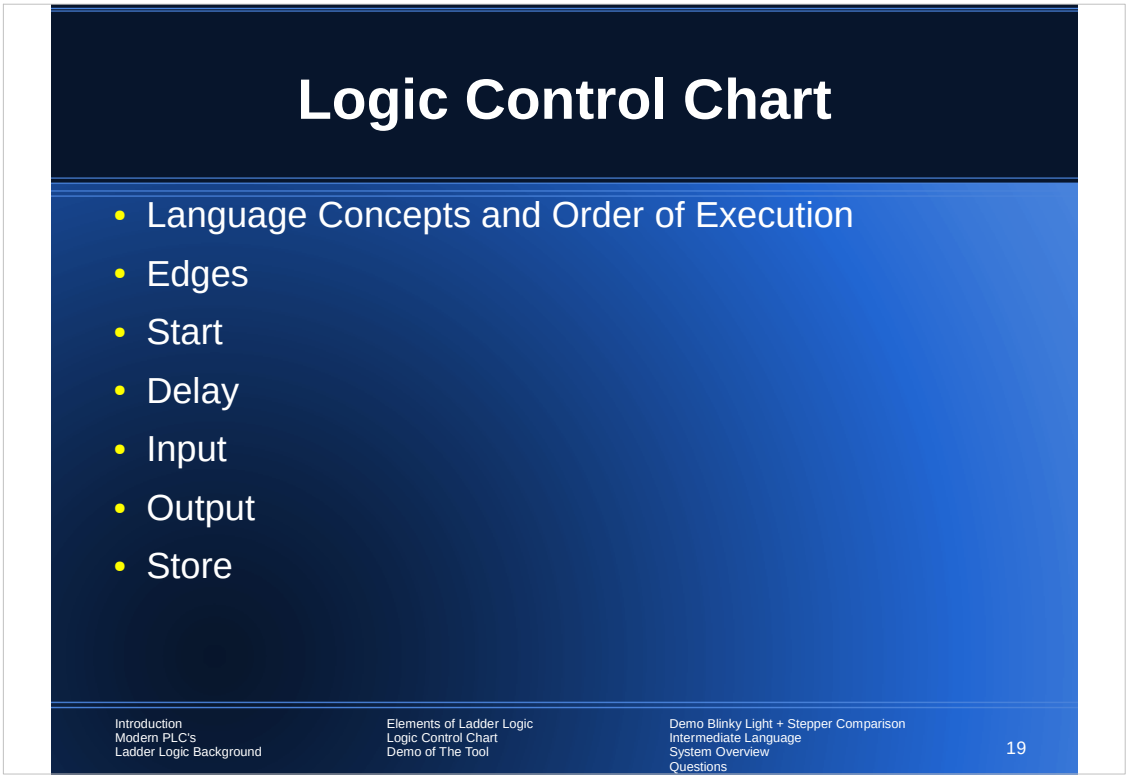
Open tool:

Parts of the UI:
Canvas, Tool pallet, Menu bar, Compile and simulation. Selection, Connectors.

Usage of the drawing canvas:
Draw a simple program, show how variable edits are made. (see blinky.xml)

Hit compile show the compiled code

Hit simulation show the simulated code



In logic control chart if you can understand flow charts you can pretty much read the program.

We only have 5 blocks but are able to do any logic. Again the emphasis here is simple see are sticking to the fundamentals

Start block is an anchoring tool it really does nothing else but tell our program to start in that state.

Edges for us are branch statements it allows you to branch based on conditionals in the guard conditions.

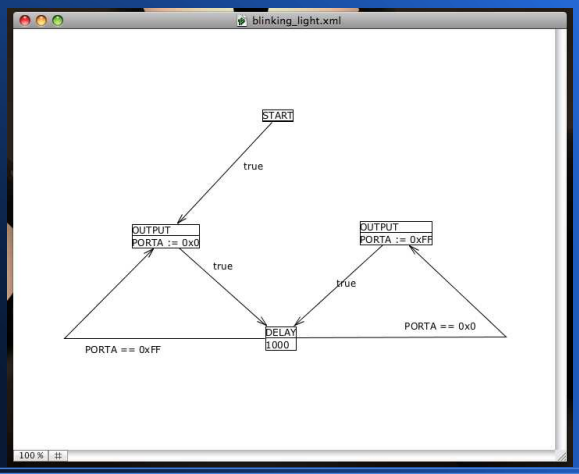
Input and Output allows you to interact with the outside world through your I/O pins.

The store block is very flexible it's used to both store variables and also to perform calculations. It also supports multi-line allowing you to compute things in one block. Multi-line store blocks are done in sequence.

The delay block allows you to wait for a short period of time useful for timing applications, you can do the same with a loop and a counter but it's less accurate.

Demo Blinky Light

- Click to add an outline

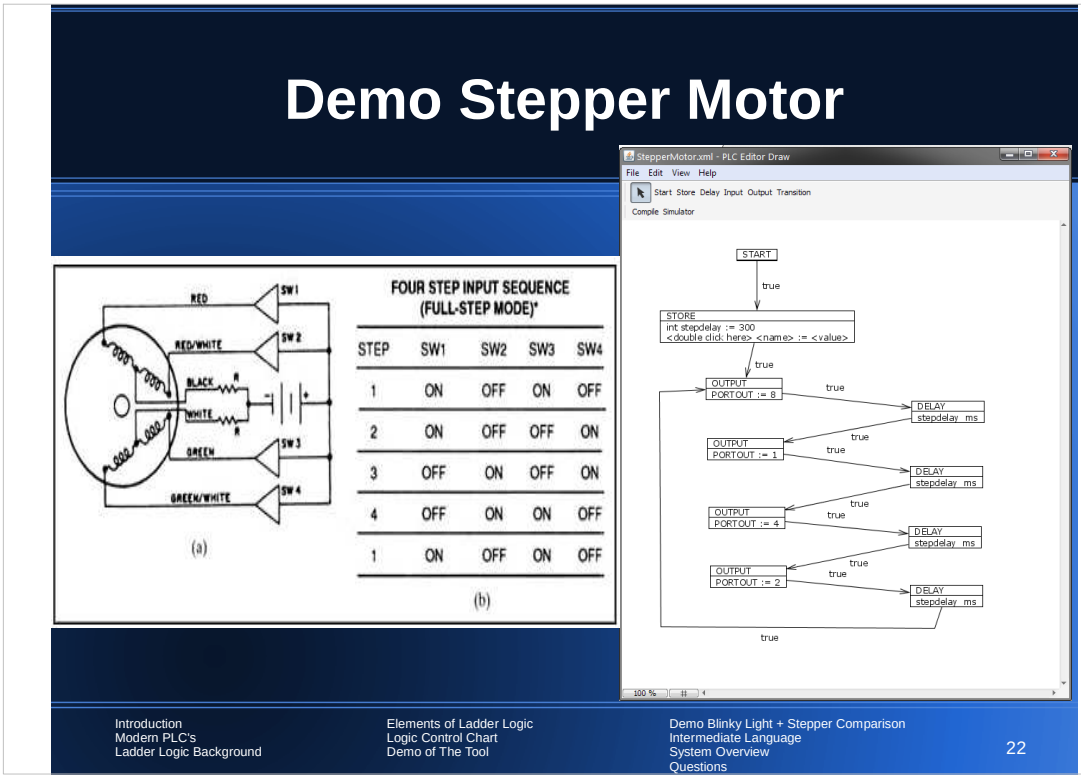


Here we have an example of a light that simply blinks on and off.

(more to add here! Will be added once demo run-throughs are practised)

Demo Blinky Light

- Demo on proto board (slide not shown)



Oh now we move away from our toy example and show something that has a more industrial application. Here we have a Logic Control Chart diagram from a stepper motor controller.

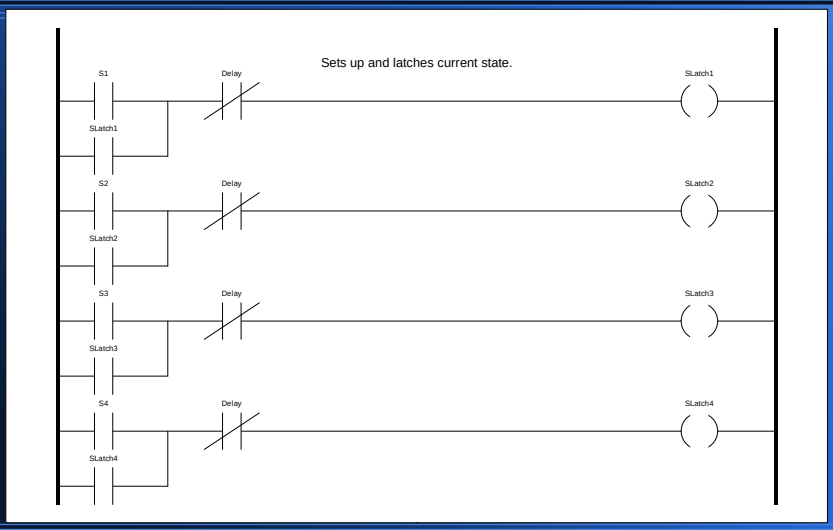
(more to add here!)

Remember how I said sequencing is very hard to do in Ladder Logic? Well here's the same program in ladder logic.

Demo Stepper Motor

- Board demo (slide not shown)

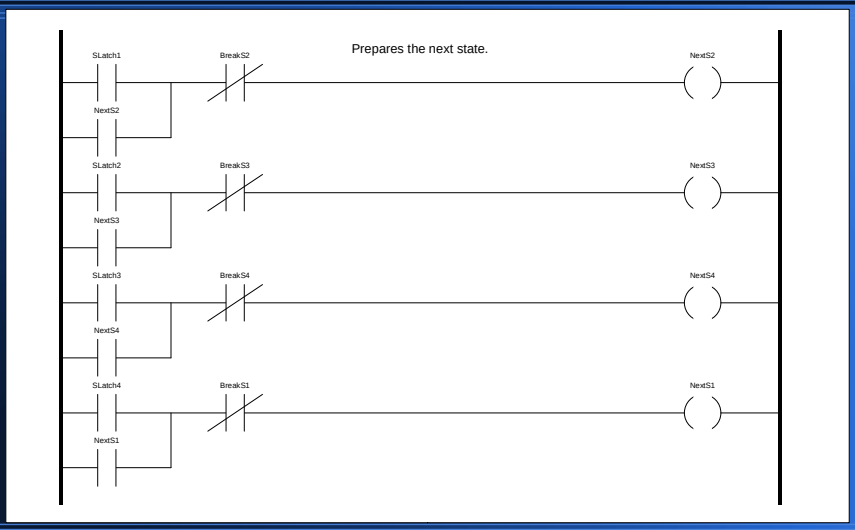
Ladder Logic Equivalent



First in Ladder Logic you need to track the current state, to do this you setup this construct that latches a state until a clock edge is fired (by delay).

This whole construct is book keeping in that we do nothing to the outside world.

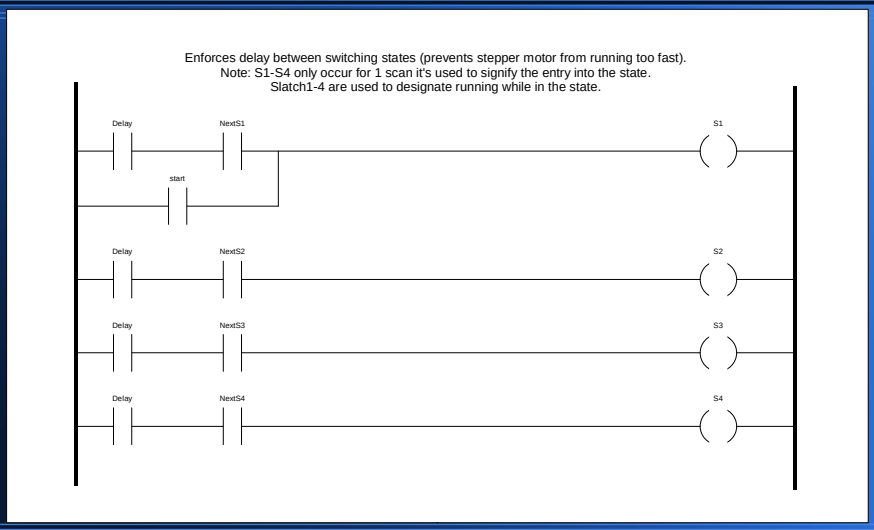
Ladder Logic Equivalent



Next we need to make sure that when we need to get to the next state we go to the correct next state. So this next block sets up the next state function.

So if we trace this, if state_1 is latched then we also latch next_s2 as the next state to be enabled.

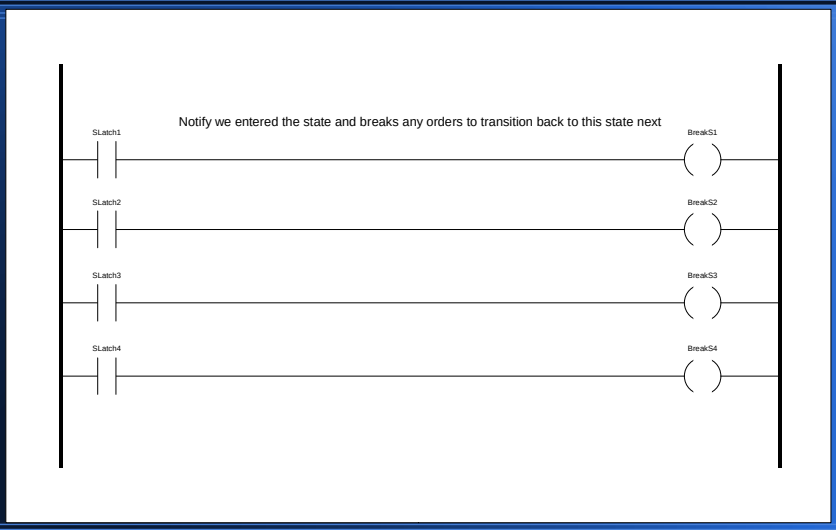
Ladder Logic Equivalent



With the next states in place we can finally enable the states.

Here we say if the next state selected is S1 then when the “delay” which is connected to a clock goes to positive edge we enter S1.

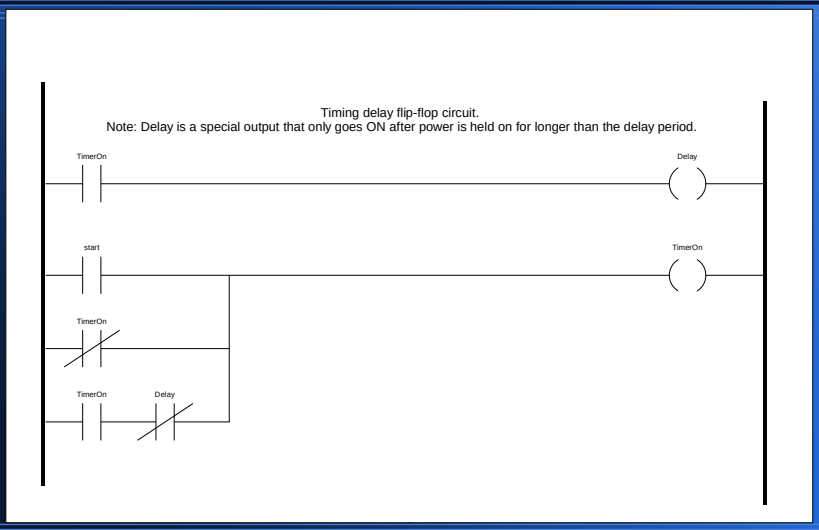
Ladder Logic Equivalent



In doing this diagram I found that after I finished going to a state I ended up being stuck in that state and I could enter multiple states.

So I needed a construct in order to disable the next requested state. In conjunction with <<2 slides back flip>> this helps clear the request to enter the state we just entered as the next state.

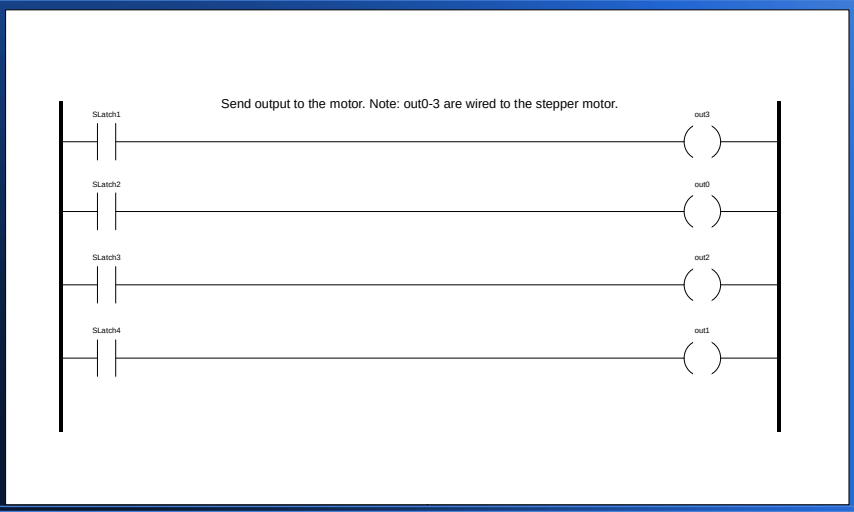
Ladder Logic Equivalent



This is a strange construct Ladder Logic does not have a clock generator so I had to create one simply put this whole block generates a clock that goes on and off with an amount of time specified by how long it takes Delay to go high after its input is held high.

(Trace the flip flop)

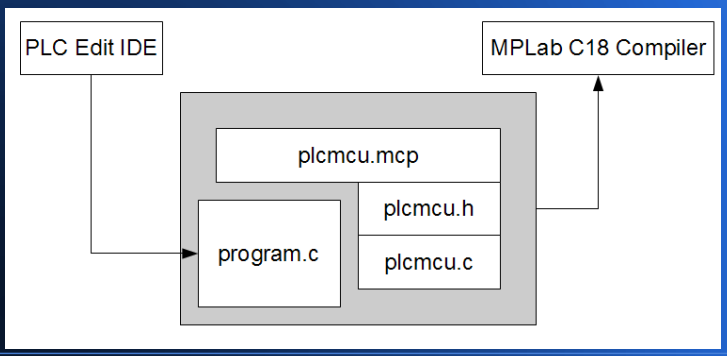
Ladder Logic Equivalent



Finally the work is done here!

Intermediate Language

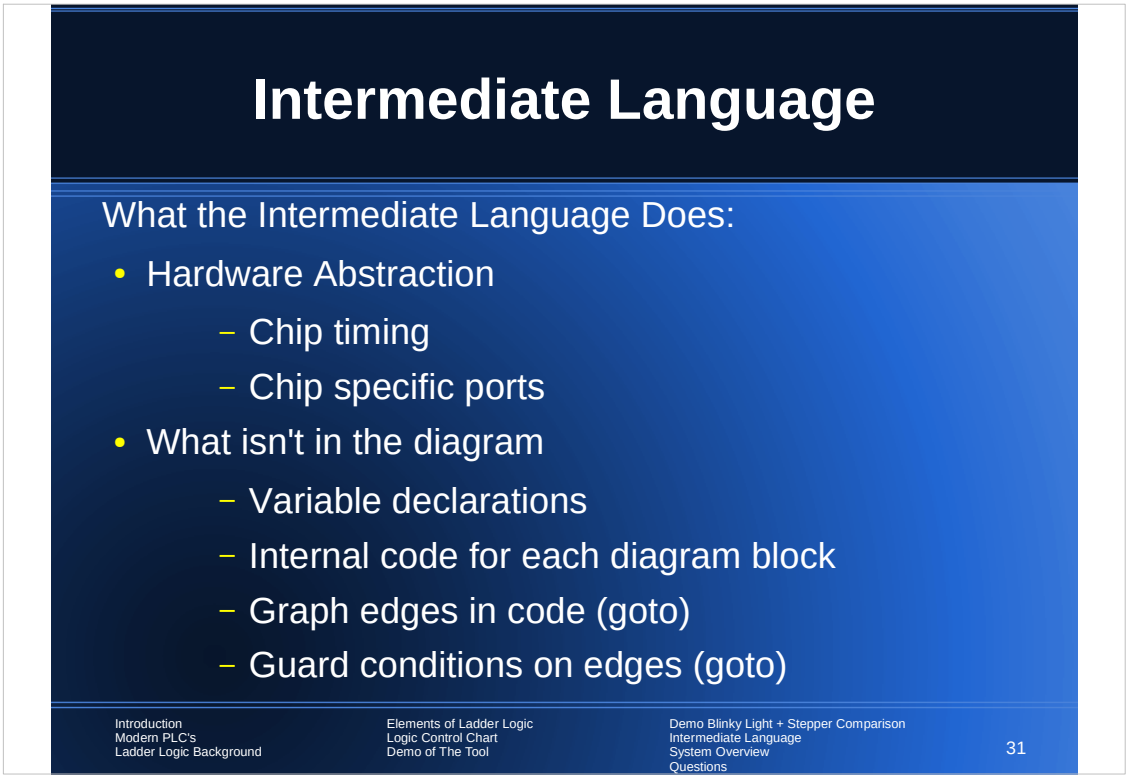
- It's just C with extra functions.
  - C18 compiles C for PIC chips
  - AVR has an equivalent C compiler



Ok so when I showed you the tool and hit compile I said it was compiled to “C” but required our framework. Well our framework is what fills in the gaps of our intermediate language.

In order to make this tool hardware portable the tool itself as much as possible refers to macros. For example I the tool doesn't actually generate the code for “delay” it just calls a “delay function” that it expects to be implemented by that hardware port. This allows for multiple targets to adjust how many cycles they need to wait to get the same delay time. Likewise I/O pins are configured the same way through header files.

In our current MPLab environment our tool generates a Program.c file. This file is referenced by our MPLab project (provided to the end user). plcmcu.h has headers that define ports and timings. The main OS is actually in plcmcu.c. Program.c is actually a subroutine that is called in plcmcu.c, we did it this way so that plcmcu.c can do any number of setup tasks and cleanup tasks when it completes.



AVR's have built in timers, PIC's do not to do timing delays you have to work out the frequency of the crystal and multiply that by 4 to get the clock frequency. Then all instructions in PIC's take 4 clock cycles to finish so you work out your timings from that. You end up looping a bunch of no-ops in assembly. To get timing delays.

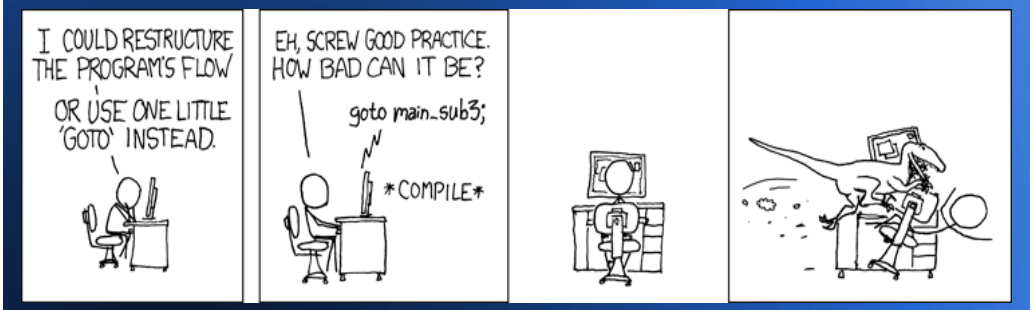
I didn't want to build this into the tool so this can be left for the specific hardware vendors.

Likewise what the port names actually are are left out to be mapped by the hardware manufacturer.

Intermediate language also houses many concepts that are abstracted away to make the diagram simpler to read.

Intermediate Language

- Gotos?

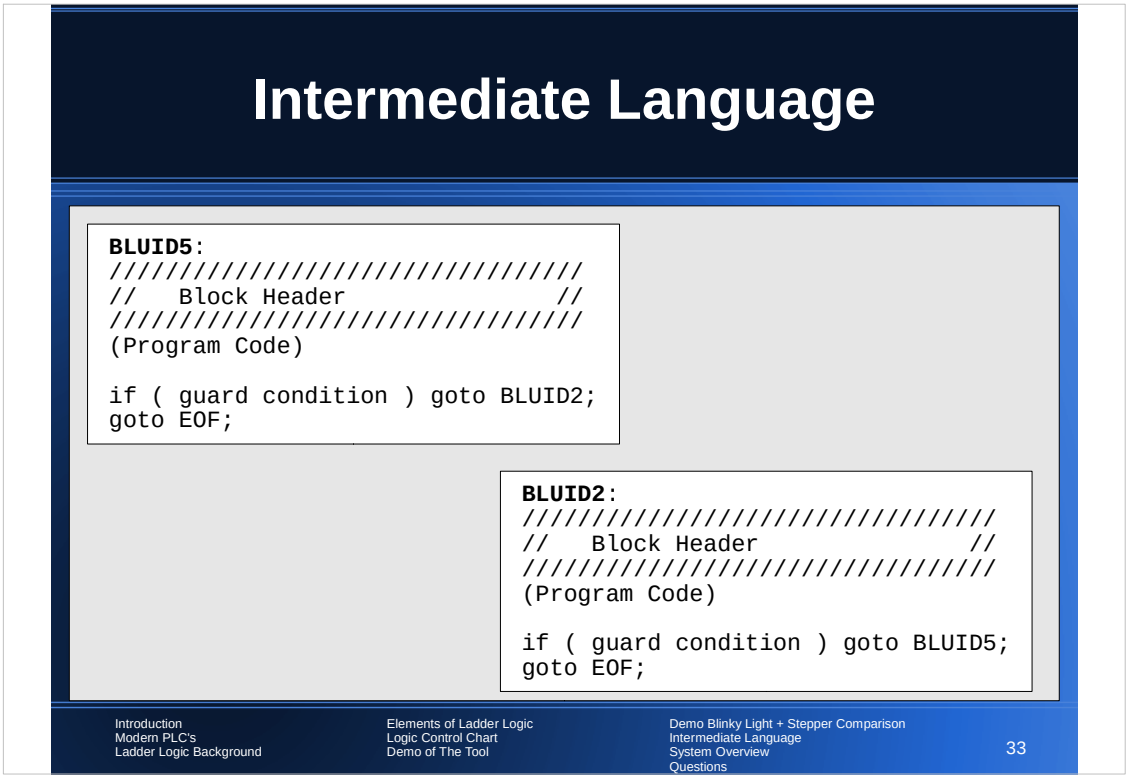


Source: http://xkcd.com/292/

Yes we are using Goto's in C. But before the panic sets in let me just briefly remind you that the intermediate language should be treated as an assembly language. No human is really suppose to be editing the compiled code directly.

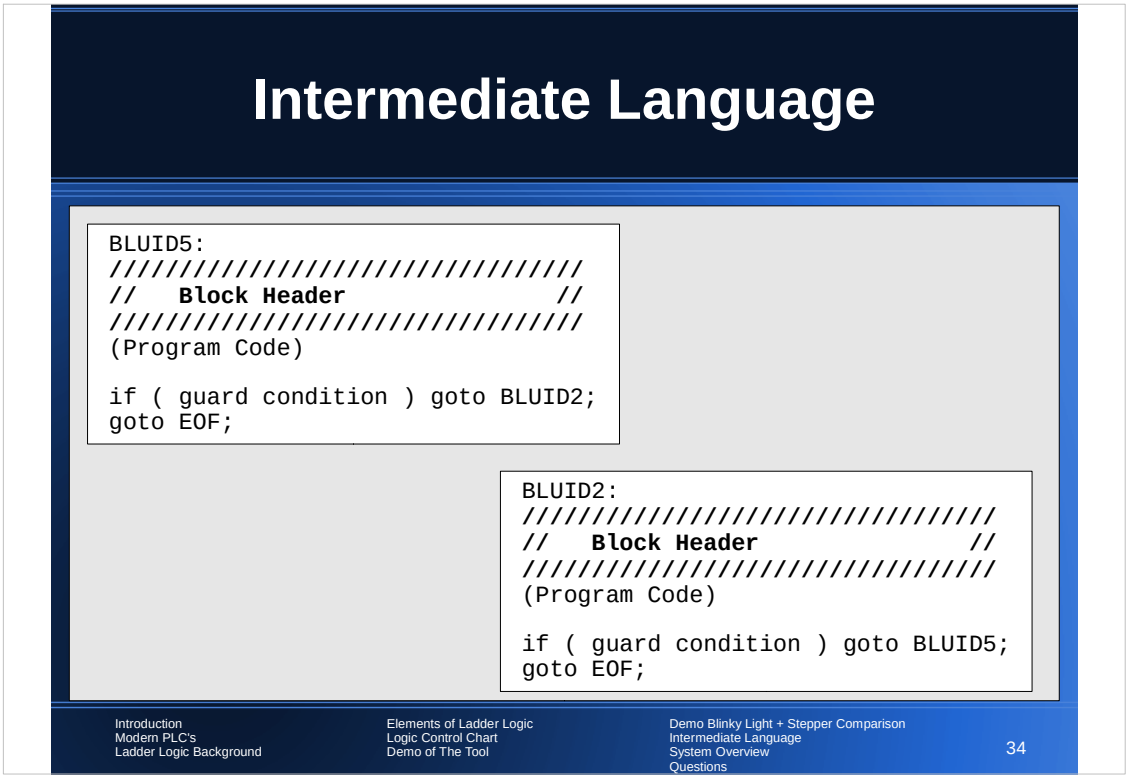
The intermediate language even though it's in C should be treated as a non platform specific assembly language.

So let's take a look at how each block looks like once it's converted into code.



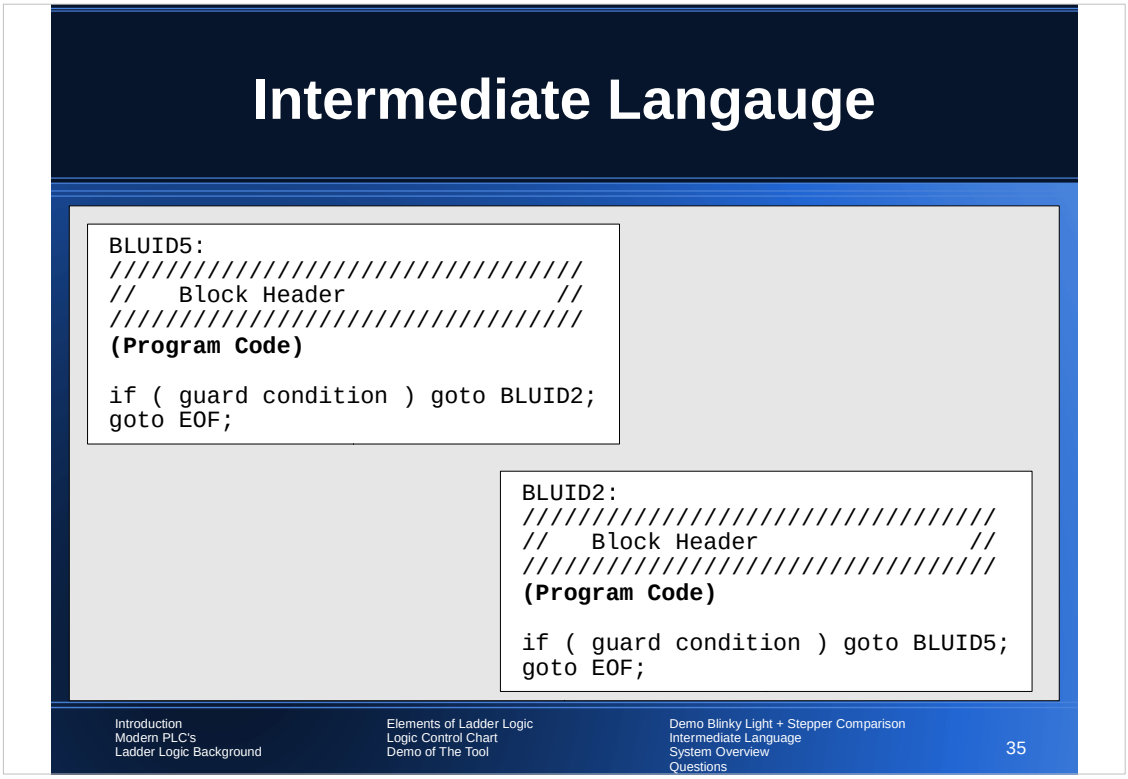
Structurally in the Intermediate language we start each visual block with a BLUID. This stands for Block Unique Identifier. As such each of these are unique and generated at compile time.

You will note that the BLUID's are written as a line label. We do this so that each block can have an edge drawn to it and thus can be entered by a simple goto BLUID#.



The next area is the block header. Generally we only use this to visually identify the blocks. Here we put which block the following code generated is for. Example: “Store Block”.

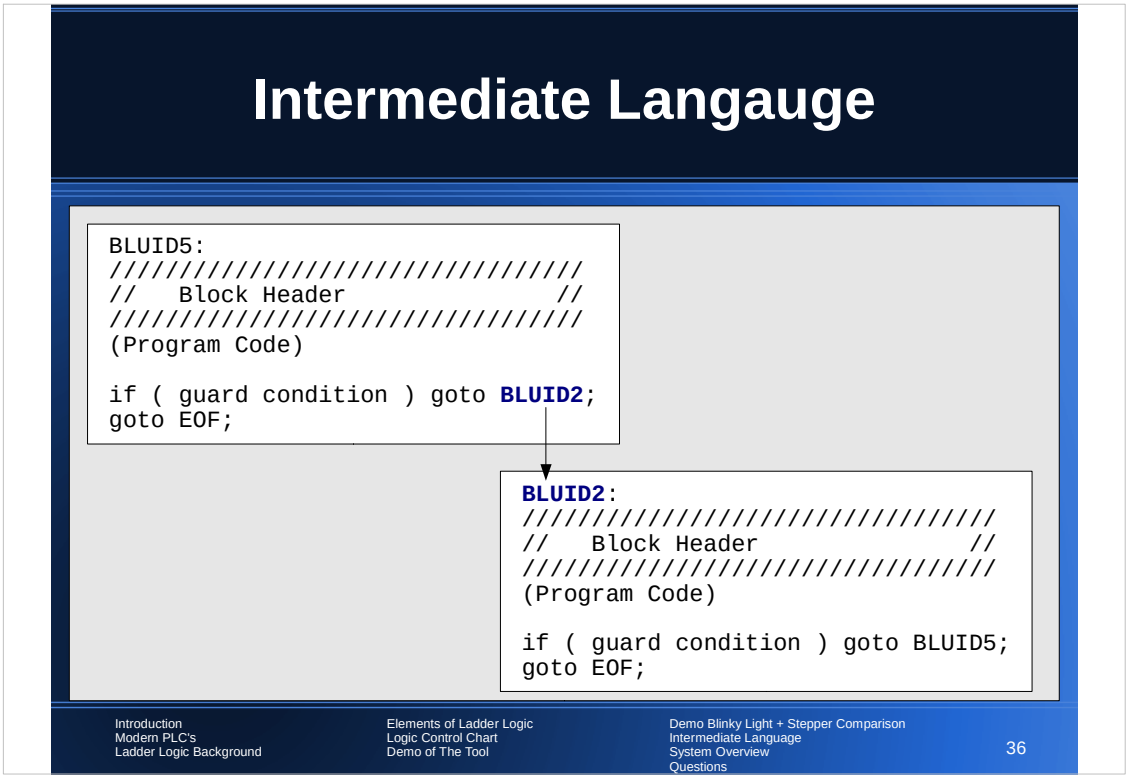
It doesn't serve a purpose other than debugging the compiler. Or tracing a code block back to it's diagram coutnerpart.



Right after the “Block Header” we have the actual block specific program code. This is different depending on which block we are generating code for and can be arbitrarily long.

For example there is no limit on how many assigns can happen in a store block and each store block statement will then appear here in this section.

Most blocks in our code however are simple enough and I expect a high number of future blocks to just be macro calls.

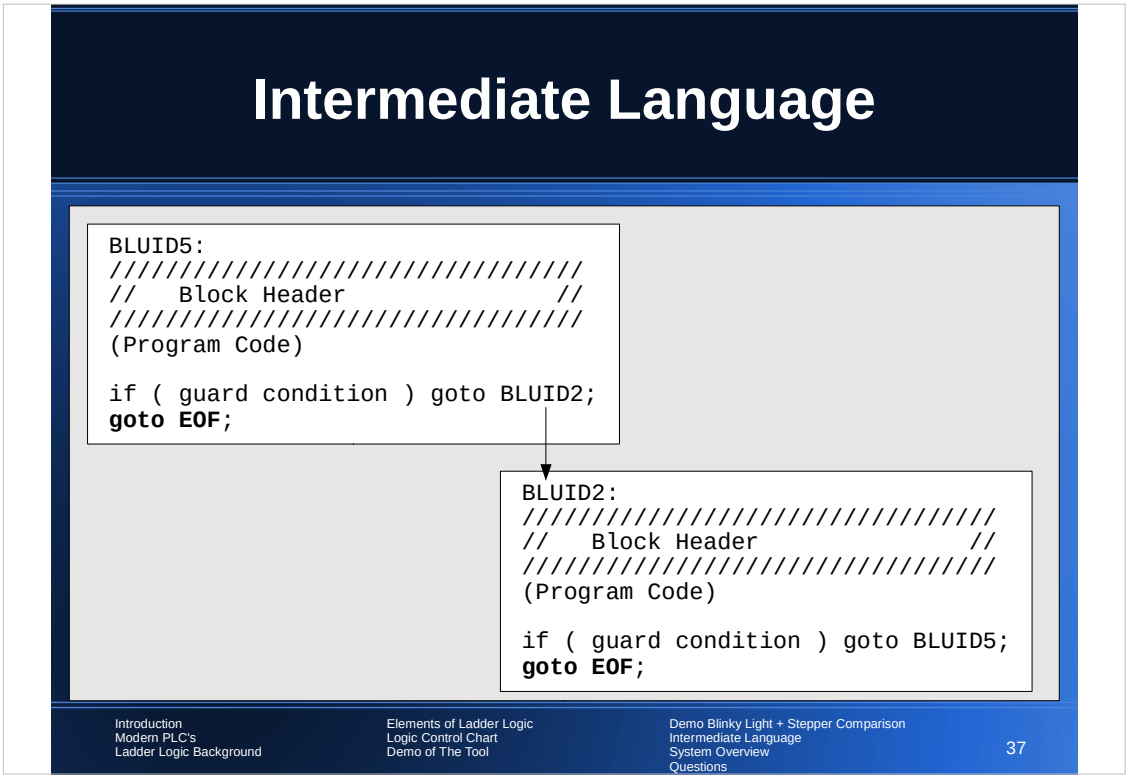


If an edge is drawn in our diagram where the source is from BLUID5 to BLUID2. We will get a guarded GOTO.

The goto fills the transition from one block to another. The guard condition can be any valid C expression.

This does present a problem in that if the user does not provide guard conditions that are mutually exclusive the outcome is undefined. This is well noted in the thesis in that our language does not permit this but our compilation tools cannot detect this.

I've been informed that checking for mutual exclusion is a very difficult problem and can be turned into a thesis itself.

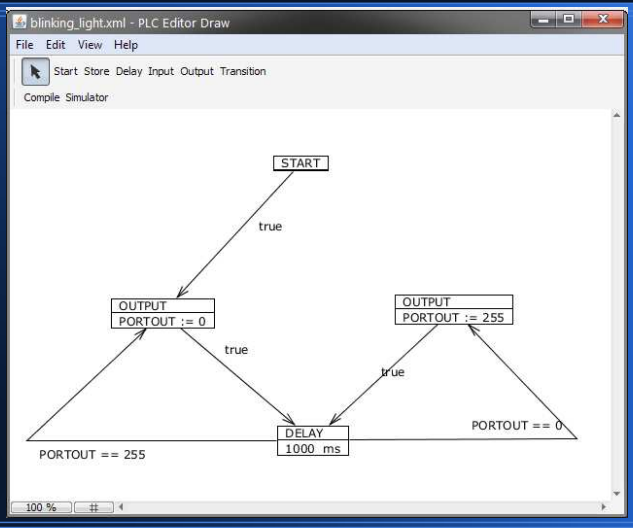


Finally in state machines and flow charts it is understood that if no edges leave a state than we must stay in that state indefinitely.

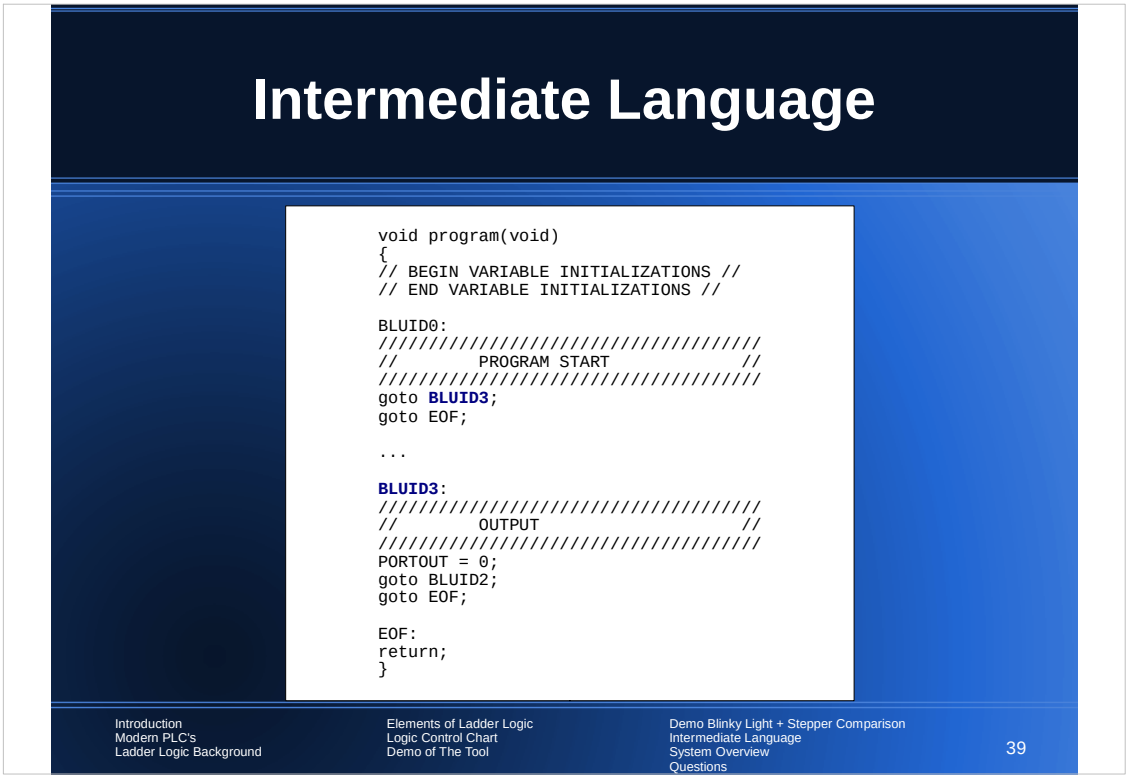
In our Intermediate Language this is accomplished by always inserting a “goto End Of File” at the end after all edge “goto's” have been generated.

At the end of the program we insert an appropriate EOF label so all blocks can “stop” if there are no further takeable transitions.

Intermediate Language



So bringing up the blinking light example we can look at how by design it's relatively easy to generate the code for this diagram.

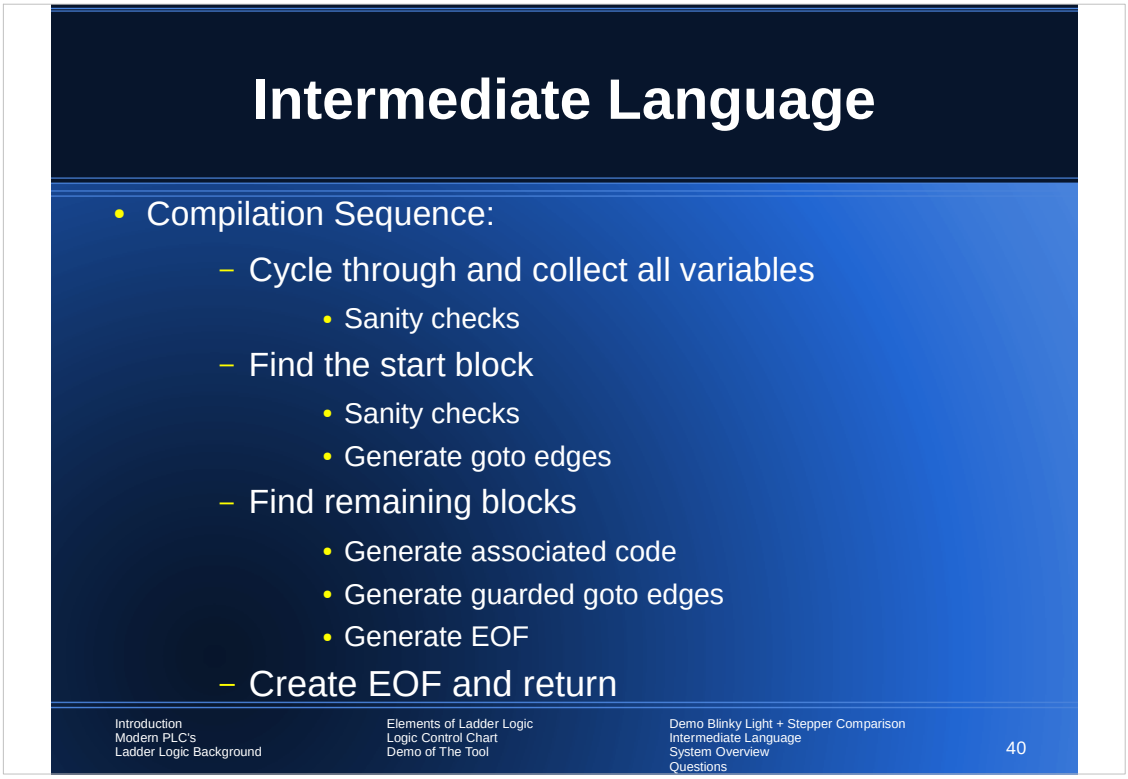


In the accompanying code section for the blinky light example you can see the output block's code.

The start block as you can see contains no code serves only as an entry point and points directly at the output block.

As a convenience any edges that are marked with true are generated without the “if” guard condition.

Finally in this example the only things in the EOF section of the code is the return statement. EOF's in our current architecture are no different from returns. However it is still nice to have the flexibility to change our minds about this later.



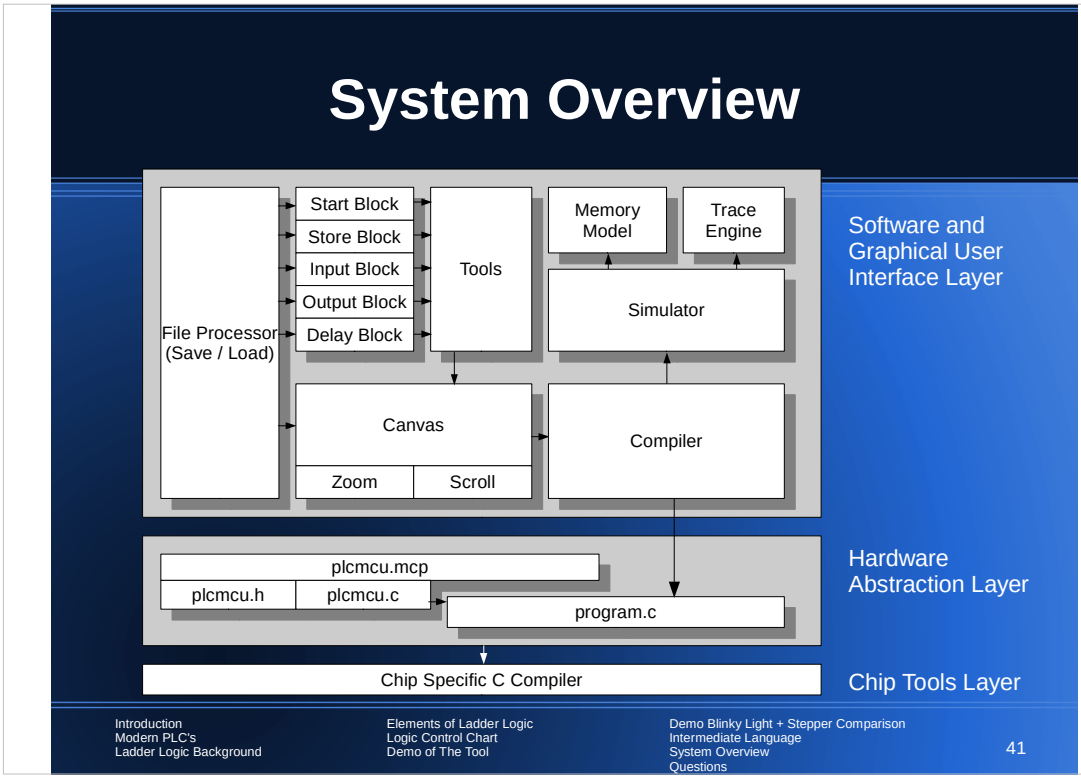
So what's involved in transforming the diagram into compiled IL code?

Because for convenience we don't require the user declare variables in the diagram, it's then up to our compiler to go through every block and collect all the variables and their types used. It then declares each of these at the beginning of the program before the start block. Since it is really cheap we can do one additional sanity check and make sure the user doesn't use one variable name as 2 or more different types in their diagram.

The next step is to search through our collection of blocks and find the start blocks. Only one start block can exist which is easy to check so check for that. The start block is always generated before any other block since we need a point of entry. After the start block is generate all the BLUID's and assign the goto edge leaving the start block.

Finally for all remaining blocks we generate their associated code, and edges and insert a EOF.

Lastly an EOF section of the program is created, in our case we insert a simple return.



So this is the system that was developed as a whole hardware and software.

I tried to keep things simple where possible but the end result is still pretty massive.

Arrows in this diagram represent data flow throughout the program. This isn't exact as there are some details I choose to omit to fit it on the slide.

Questions?

References

- J. Arlow and I. Neustadt, UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd Edition) (Addison-Wesley Object Technology Series). Addison-Wesley Professional, 2005.
- A. B. R. Automation, Rockwell Automation: I/O Adapter Specifications. Rock-well Automation, 1201 South Second St., Milwaukee, WI 53204-2496, USA.
- W. Bolton, Programmable Logic Controllers, Fourth Edition. Newnes, 2006.
- T. L. Booth, Sequential Machines and Automata Theory. John Wiley and Sons, Inc., 1967.
- M. Corp., “Mitsubishi electric automation: Catalog.” Used image from product catalog only, 01 2009.
- A. J. Crispin, Programmable Logic Control lers and Their Engineering Applications. McGraw-Hill Companies, 1996.
- “How plc’s work.” http://www.plcdev.com/how_plcs_work, 01 2009.
- R. Filer, Programmable Controllers Using Allen-Bradley Slc 500 and Control logix. Englewood Cliffs: Prentice Hall, 2002.