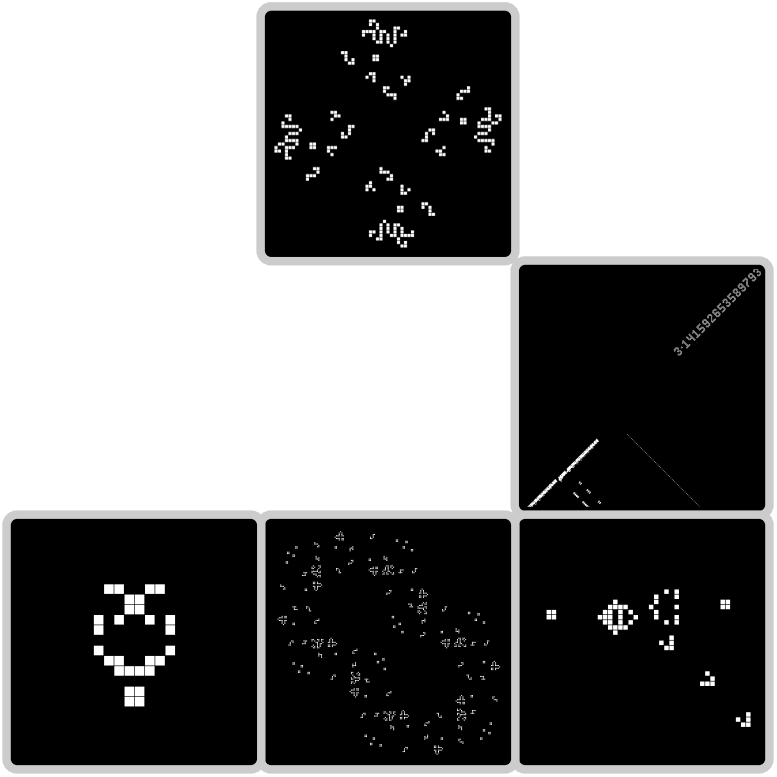
Conway's Game of Life

Mathematics and Construction

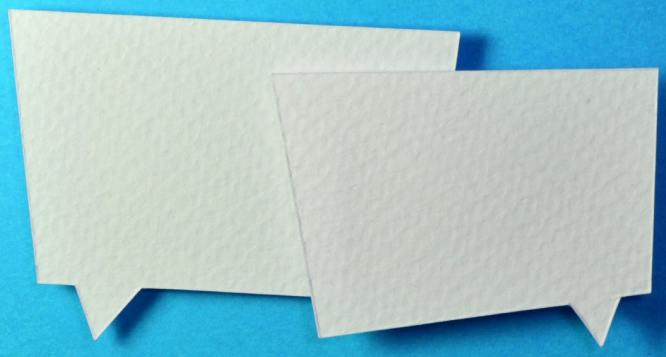


Nathaniel Johnston and **Dave Greene**

CONWAYLIFE.COM

To John Horton Conway
For giving us 50 years of Life.

Contents



Preface

The Goal

This book provides an introduction to Conway's Game of Life, the interesting mathematics behind it, and the methods used to construct many of its most interesting patterns. This book generally tries to avoid presenting patterns in isolation or as historical notes, but rather tries to guide the reader through the thought processes that went into creating them in the first place.

While we will largely follow the history of the Game of Life as we go through the book, we emphasize that this is *not* the primary goal of the book, but rather it is a by-product of the fact that most recently-discovered patterns build upon patterns and techniques that were developed earlier. Instead, the goal of this book is to demystify the Game of Life by breaking down the complex patterns that have been developed in it into bite-size chunks that can be understood individually.

Intended Audience

While this book does not have any formal mathematical or computer science prerequisites, it is written at a level aimed at students of at least a first-year undergraduate university level. Some high-school-level topics like logarithms, the "floor" function $f(x) = \lfloor x \rfloor$ for rounding numbers down, the binary representation of a positive integer, and summation notation like $\Sigma_{k=1}^{n} k^2$ for adding numbers, are used frequently and without much explanation. A basic understanding of how mathematical proofs and computer programming work is also expected. That is, we expect a certain level of mathematical and computer science maturity from the reader, but no specialized knowledge of either topic.

More specifically, we prove some simple theorems about the Game of Life in Chapters 1 through 5. These proofs do not use any specialized proof techniques or expect the reader to have any specific university-level mathematical knowledge, but rather just expect the reader to be able to follow a logical argument. Similarly, we introduce a programming language for building computer programs out of Life circuits in Chapter 2, so that material will be easier to master if the reader has had prior exposure to computer programming.

Somewhat more advanced mathematical topics that we make use of are summarized in Appendix **??**, though they are typically introduced very gently in the main text as well, and we only require a very surface-level understanding of them. We make use of the greatest common divisor, the least common multiple, and Bézout's identity when discussing oscillator periods in Chapter **??**,

so we introduce these tools in Appendix **??**. Infinite series make brief appearances at the end of Chapter **??** and in Section 2.6, though the reader is not expected to really have any familiarity with them or understand convergence issues. Finally, big-O notation is used to discuss the growth rate of patterns in Sections **??** and 2.7.2, so we introduce this concept in Appendix **??**.

How to Use

Conway's Game of Life is an extremely visual game, so this book makes very liberal use of figures throughout, particularly when new patterns or techniques for creating patterns are introduced. However, the Game of Life is best observed in motion, which makes static images in a textbook less than ideal as learning tools. In order to help present the motion of patterns a bit better, we do five things:

- Figures are presented with alive cells in black and dead cells in white, but furthermore we use a gradient from blue to orange to denote cells that were alive in past generations of the pattern. Bright blue cells were just alive, whereas cells that are orange were alive in the more distant past (roughly 75 generations or more—see Figure 1).

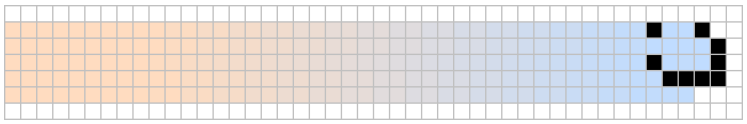


Figure 1: An object moving to the right with a gradient behind it indicating how long ago it was in each location. White cells were never alive, black cells are currently alive, blue cells were alive recently, and orange cells were alive long ago (roughly 75 generations or more).

- If we really wish to emphasize what a pattern looks like in different generations, all generations of interest will be displayed, along with arrows that specify how many generations have passed. For example, if we want to clarify exactly what the pattern in Figure 1 does as it moves from left to right, we might display it as in Figure 2.

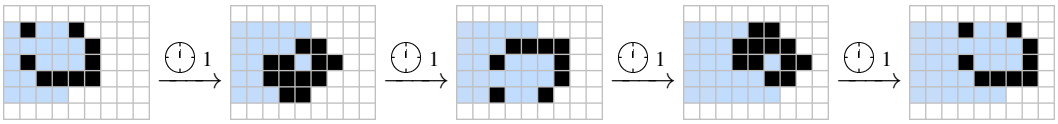


Figure 2: The same object as in Figure 1, but displayed in a more explicit fashion. This image shows how the pattern changes every time 1 generation elapses.

- We use colors to highlight various pieces of patterns, and we maintain a consistent coloring scheme throughout the book. Light pastel colors like aqua, magenta, light green, yellow, and light orange are used to highlight *around* objects—cells in these colors are dead, but highlight a region in the Life plane consisting of cells that all serve some common purpose or logically make up one "object" (see Figure 3). On the other hand, darker colors like dark green, dark orange, and dark red are used to highlight certain live cells. Dark green is typically used to highlight the input to some reaction, dark orange is typically used for the output of the reaction, and dark red is used otherwise (see Figure 4).

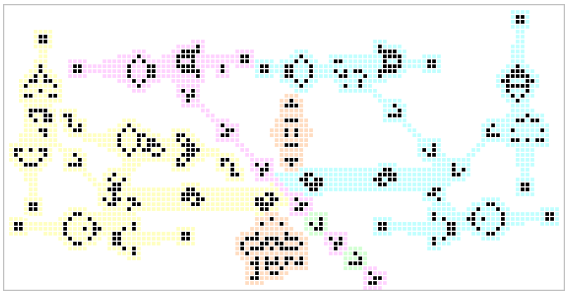


Figure 3: A glider gun made up of several different component reactions that are highlighted in different colors. In particular, gliders are created by a gun highlighted in magenta, lightweight spaceships are created by guns highlighted in aqua and yellow, and those lightweight spaceships are merged into the original glider stream by oscillators highlighted in light orange.

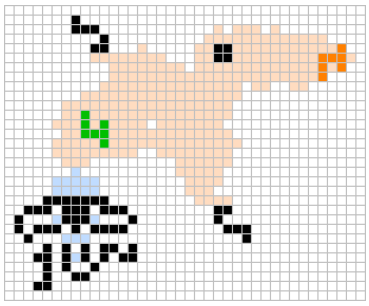


Figure 4: A dark green "Herschel" comes in from the left and creates the dark orange Herschel on the right. Cells highlighted in blue are constantly changing due to being part of an oscillator, whereas cells that are light orange are usually dead, except when the Herschel passes through.

- In the electronic version of this book, almost every figure is actually a clickable link that will open a text file containing RLE or Macrocell code for the displayed pattern (go ahead, click on any of the figures above, or even one of the five images on the cover page). This code can be copy and pasted into Life simulation software like Golly (golly.sourceforge.net) so that it can be explored and manipulated.[1] Note that clicking on figures may not work in certain PDF viewers (such as the viewers built into web browsers), so we recommend using Adobe Acrobat Reader (get.adobe.com/reader) to read this book digitally.
- RLE, LifeHistory, or Macrocell codes for all of the patterns displayed in the book are also available at the book's website (conwaylife.com/book). Furthermore, the patterns can be viewed and manipulated right on that website as well via any modern web browser, without downloading any additional software.

Exercises

We strongly encourage the reader to work through this book's many exercises that can be found at the end of each chapter. For this reason, it is extremely important to either download Life software or use the book's website to view and edit patterns while making your way through the book—Life is meant to be played, not just watched, and many of the exercises simply cannot be solved without the assistance of Life simulation software.

Roughly half of the exercises are marked with an asterisk (∗), which means that they have a solution provided in Appendix **??**.

Acknowledgments

The authors are indebted to dozens of people who opened the world of Conway's Game of Life to them. Rather than acknowledging the people who discovered the reactions and patterns that we discuss here, they are credited in footnotes and figures throughout the book.

We extend thanks to Nicolay Beluchenko, Steven Eker, Mark Niemiec, and Michael Simkin for helpful conversations about content of the book. Thanks to Andrew Trevorrow and Tomas Rokicki for creating the open-source cross-platform CA editor and simulator Golly (golly.sourceforge.net), without which many of the patterns discussed in this book would not have been discovered. Thanks to Chris Rowett for creating LifeViewer (lazyslug.no-ip.biz/lifeview), which is used on this book's website (conwaylife.com/book) to display patterns and make them interactive. Thanks

[1] For an explanation of how RLE or Macrocell code works, see conwaylife.com/wiki/Run_Length_Encoded or conwaylife.com/wiki/Macrocell.

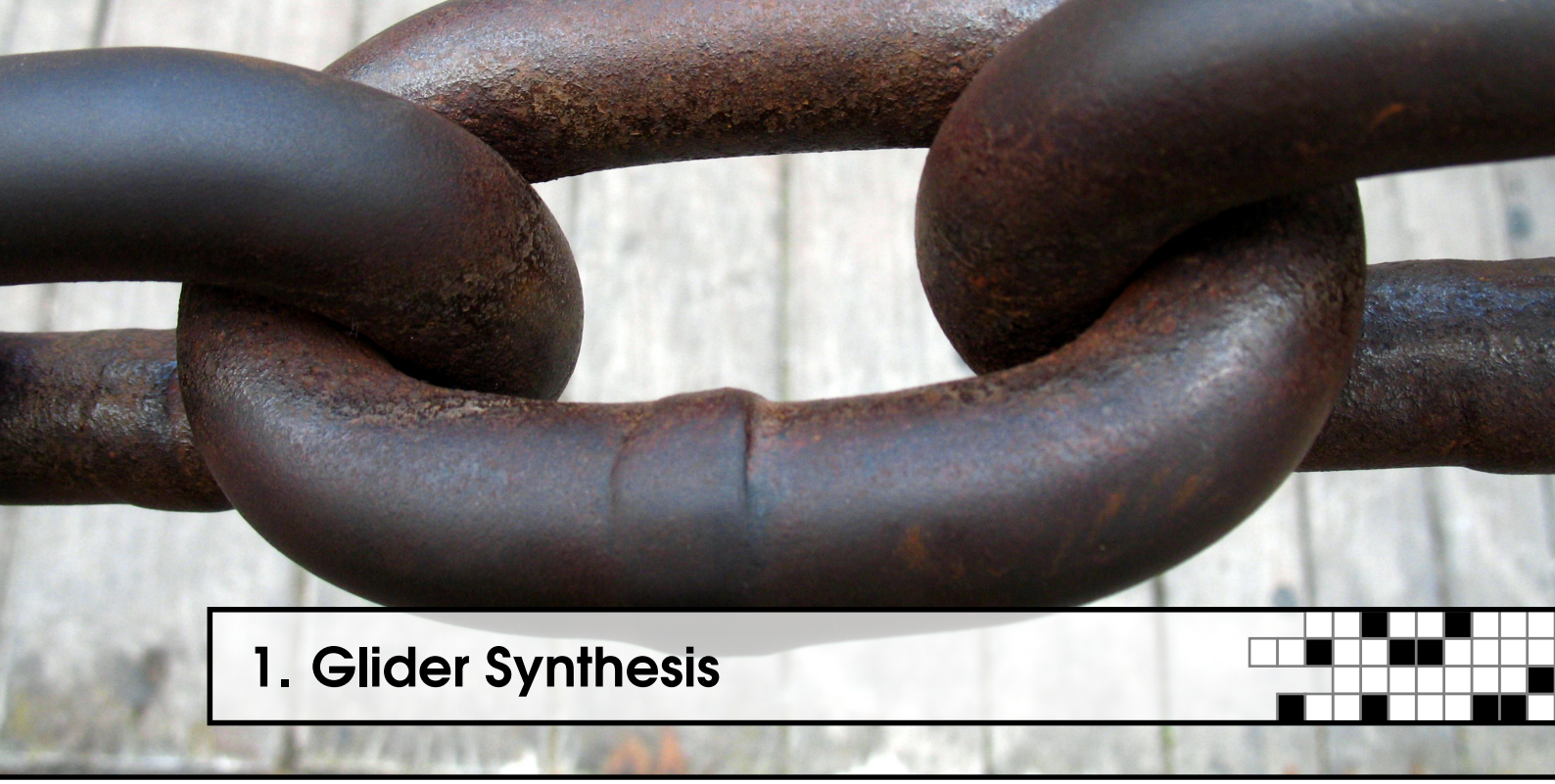
to Velimir Gayevskiy and Mathias Legrand for the *Legrand Orange Book* LaTeX template from LaTeXTemplates.com.

Finally, the authors would like to thank their wives Kathryn and Melanie for tolerating them during their years of mental absence glued to this book, and their parents for encouraging them to care about both learning and teaching. Many thanks also to Mount Allison University for giving the first author the academic freedom to pursue a project like this one.

I

Classical Topics

II Circuitry and Logic



1. Glider Synthesis

Life isn't about finding yourself. Life is about creating yourself.

George Bernard Shaw

In previous chapters, we saw several different patterns that were capable of generating an endless supply of gliders. Glider guns like the Gosper glider gun and the twin bees gun create streams of gliders coming from a fixed location, and rakes like the space rake create waves of gliders that come from a moving source. We have also seen a few patterns (mostly based on the switch engine) capable of creating other objects like blocks or other small still lifes. In this chapter, we develop a systematic method of constructing patterns that turn these simple objects like gliders and blocks into more and more complicated patterns. In particular, we will look at how we can collide gliders together in order to create other objects. For example, it is possible to collide three gliders in such a way as to produce a lightweight spaceship (see Figure 1.1).

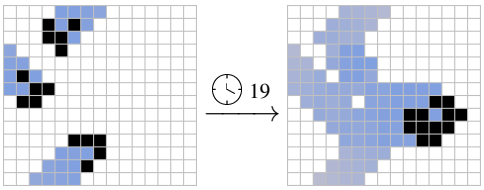


Figure 1.1: Three gliders colliding in such a way as to create a lightweight spaceship.

We call this a *3-glider synthesis* of the lightweight spaceship, and it is useful because we already know how to generate gliders (via glider guns), so we now know how to generate lightweight spaceships: just produce three gliders that will collide with each other in the right way. For example, Figure 1.2 uses three Gosper glider guns to create a period 30 lightweight spaceship gun.

There's nothing particularly special about the lightweight spaceship in this example—we will see throughout this chapter that we can collide gliders from glider guns to create a wide variety of moving objects. We can also use gliders to synthesize stationary objects, but for this the glider source has to be moving in order to prevent subsequent gliders from colliding with the synthesized object. Because our ultimate goal is to make use of these glider collisions via guns and rakes, we require that the gliders in a synthesis could arrive at their positions from arbitrarily far away—an example of a

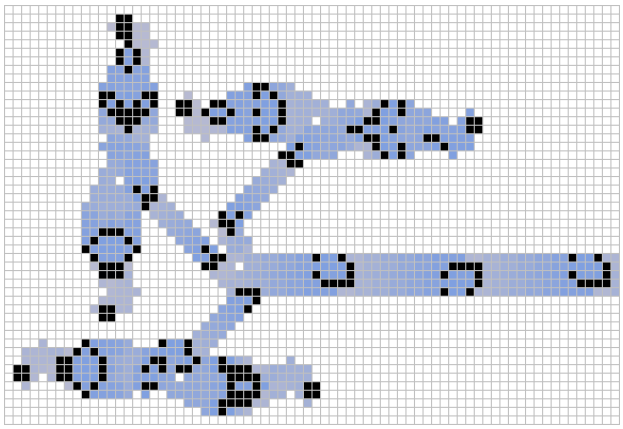


Figure 1.2: A lightweight spaceship gun constructed by using three Gosper glider guns (top-right, top-left, and bottom-left) to shoot three gliders at each other, which collide in the orientation depicted in Figure 1.1 in order to create a lightweight spaceship.

glider collision that is not considered a true glider synthesis, since there is no way to get the gliders in the indicated positions, is provided in Figure 1.3. Once we have a large catalogue of glider syntheses, the world of Life will open up considerably for us—we will be able to create patterns that construct almost any object in almost any location on the Life plane that we like.

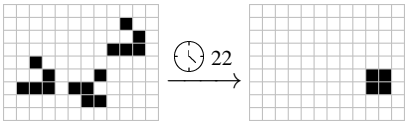


Figure 1.3: A collision of three gliders that produces a block. However, if the two rightmost gliders came from farther away, they would collide with each other before reaching the displayed configuration, and thus this is not a valid 3-glider synthesis.

1.1 Two-Glider Syntheses

Once we have a glider synthesis in hand, it is typically straightforward to make use of it, since glider streams can often be created just by lining up glider guns or rakes that we already know in the right spots. But how can we come up with glider syntheses in the first place? For example, how was the three-glider synthesis of a lightweight spaceship in Figure 1.1 found? The simplest answer, as unsatisfying as it might seem, is to just collide a bunch of gliders together in different ways and catalogue which collisions lead to which objects. Once we have a good selection of "simple" objects that we know how to synthesize, we can then try colliding gliders with those objects to create more complicated patterns, and so on.

To begin down this road, let's look at the simplest type of glider synthesis we can perform: synthesis using a collision of only two gliders. We already saw a few two-glider collisions in Section **??** when we aimed two glider waves at each other so as to destroy some gliders and reflect others, and again in Section **??** when we aimed two glider waves at each other to create a bi-block wick. Fortunately, cataloging all two-glider syntheses is not too difficult; there are only 71 different ways that two gliders can collide,[1] and we can simply write down which objects are created by each collision. A full summary of exactly which two-glider collisions lead to which objects is provided by Table 1.1, but we note that only a handful of different useful objects can actually be created with two gliders, including a blinker, block, boat, hive, loaf, and eater 1.

[1] There does not seem to be a nice "mathematical" way to arrive at the number 71 here: it's just a matter of arranging gliders in all possible different orientations and phases and seeing which ones result in collisions.

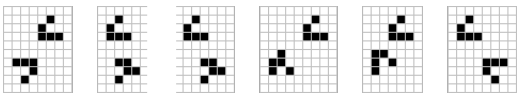
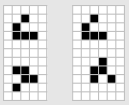
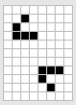
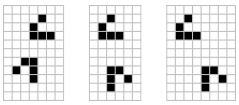
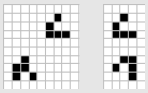
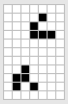
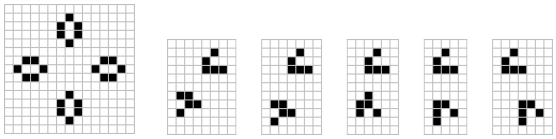
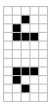
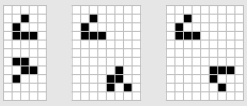
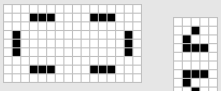
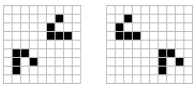
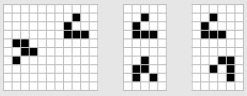
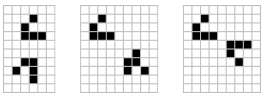
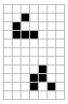
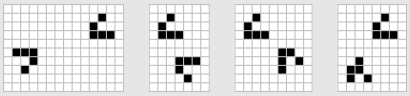
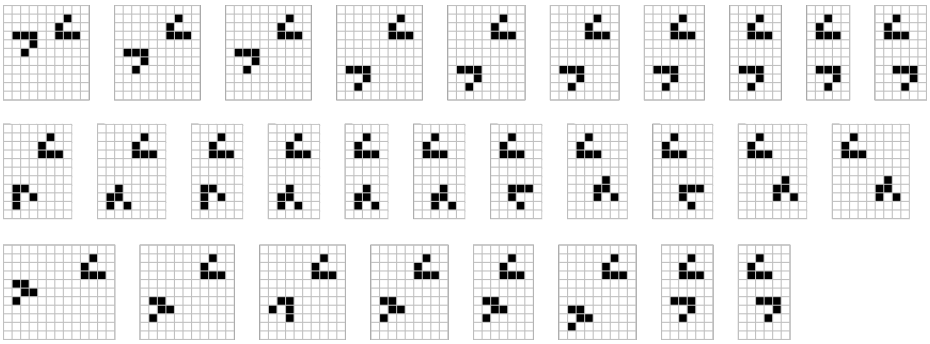
result	2-glider collisions	result	collision
block		boat	
pond		beehive	
blinker		loaf	
glider		eater 1	
honey farm		bi-block	
traffic light		interchange	
loaf + blinker		traffic light + glider	
B-hept.		lumps of muck	
pi-hept.		teardrop	
misc.		octomino	
nothing			

Table 1.1: A summary of the results of all 71 possible 2-glider collisions. The four "misc" collisions yield somewhat messy combinations of common objects like blocks and blinkers. The rightmost of the "misc" collisions is sometimes called the *two-glider mess*, as it takes 530 generations to stabilize—more than any of the other collisions.

While the objects that we can create with just two gliders are not particularly exciting, many of them are essential building blocks of complex patterns, such as eater 1 and the B-heptomino (which we recall evolves into a Herschel). It's also worth pointing out that the two collisions that result in a single glider can in fact sometimes be useful. For example, one of those collisions results in a glider going in a different direction than either of the two original gliders, which can help us navigate gliders around tight spots and simplify many more complicated glider syntheses (see Exercise 1.13).

1.2 Syntheses Involving Three or More Gliders

When moving from two-glider syntheses to three-glider syntheses, things become much more complicated—it is no longer possible to list all collisions and catalog their output, since there are far too many possibilities. To see why this is, recall that the two-glider mess takes 530 generations to stabilize, and we could fire a third glider from dozens of different positions and hundreds of different timings to collide with that chaotic mess.

On the other hand, being unable to catalog all of these three-glider collisions is perhaps not too much of a loss, since we expect that the majority of them would lead to uninteresting combinations of blocks, blinkers, and other common objects that we already know how to synthesize with just two gliders. Some of the more interesting three-glider syntheses that are known are presented in Table 1.2. However, we do stress that this table is not complete: there are known 3-glider syntheses not presented in this table, and it is entirely possible that there are simple objects with three-glider syntheses that have not yet been found.[2]

Although the 3-glider synthesis of a single glider in Table 1.2 might seem somewhat silly at first glance, it has the interesting property that the synthesized glider travels perpendicular to each of the input gliders[3]—a property that is not shared by any of its 2-glider syntheses. For this reason, it is actually quite useful when trying to manipulate glider positions or reduce the number of directions used in glider syntheses of more complicated objects (see Exercise 1.15).

Glider syntheses of spaceships (such as the three-glider syntheses of the lightweight, middleweight, or heavyweight spaceships) can be used to construct guns for these spaceships, as we already saw at the start of this chapter. The 3-glider collision that creates a glider-producing switch engine is also quite exciting, as it is our first example of a synthesis of an infinitely-growing pattern. Being able to synthesize queen bees is similarly exciting, since a Gosper glider gun is made up of nothing more than two blocks and two queen bees, all of which we now know how to synthesize, so we are now able to use gliders to synthesize a pattern that creates additional gliders (see Figure 1.4),[4] and we could conceivably use rakes to create Gosper glider guns that in turn create other patterns.

We have a fairly wide variety of objects that we can now synthesize with gliders, but there are still some rather fundamental objects that are missing from our lists of 2- and 3-glider syntheses. For example, if we wanted to synthesize a twin bees gun, we would first have to know how to synthesize twin bees, which (as far as we know) requires at least 4 gliders. Similarly, even though we can use 3 gliders to synthesize a glider-producing switch engine (plus lots of debris), we haven't yet seen how to synthesize a switch engine by itself. Again, as far as we know this requires at least 4 gliders.

It is also sometimes beneficial to be aware of glider syntheses that make use of more gliders, even when syntheses involving fewer gliders exist. For example, the 3-glider synthesis of the heavyweight spaceship given in Table 1.2 is actually quite difficult to make use of, since the two gliders coming in from the top-right are so close together that they cannot be generated by adjacent Gosper glider guns. Thus we often prefer glider syntheses that use widely-spaced gliders, even if that comes at the

[2] As an example, the pentadecathlon was not known to be synthesizable with only three gliders until Heinrich Koenig found the synthesis given in Table 1.2 in April 1997. Similarly, it was not known how to create *any* infinitely-growing object using just 3 gliders until Michael Simkin found the displayed 3-glider collision in October 2014.

[3] Even more specifically, two of the gliders collide in such a way as to destroy each other, but they create a banana spark in the process that reflects the third glider as in Figure **??**.

[4] Synthesizing the Gosper glider gun "piece-by-piece" like this is the quite straightforward, and requires 10 gliders. However, slightly smaller syntheses of the Gosper glider gun are known, requiring as few as 8 gliders.

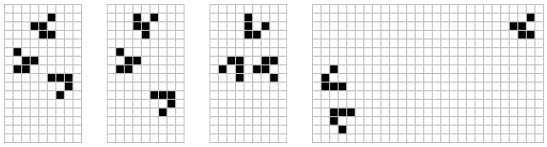
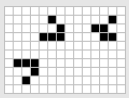
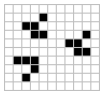
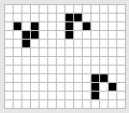
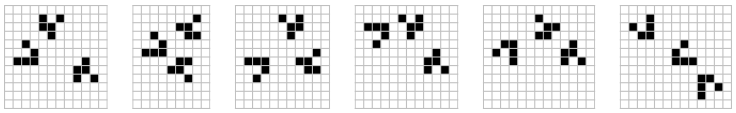
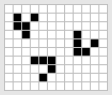
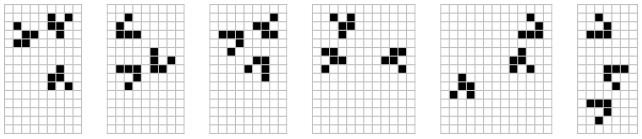
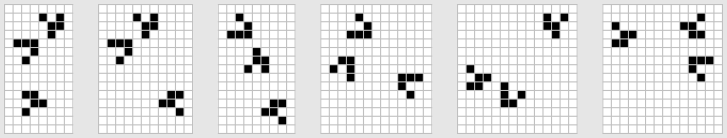
result	3-glider collisions
lightweight spaceship	
middleweight spaceship	
heavyweight spaceship	
glider	
pulsar	
pentadecathlon	
R-pentomino	
queen bee	
glider-producing switch engine (plus junk)	

Table 1.2: A selection of useful 3-glider syntheses. The 3-glider collision that creates a glider-producing switch engine is not a true glider synthesis due to the fact that it also creates a wide assortment of other debris, but it is nonetheless noteworthy for being the only known way of generating infinite growth with just 3 gliders.

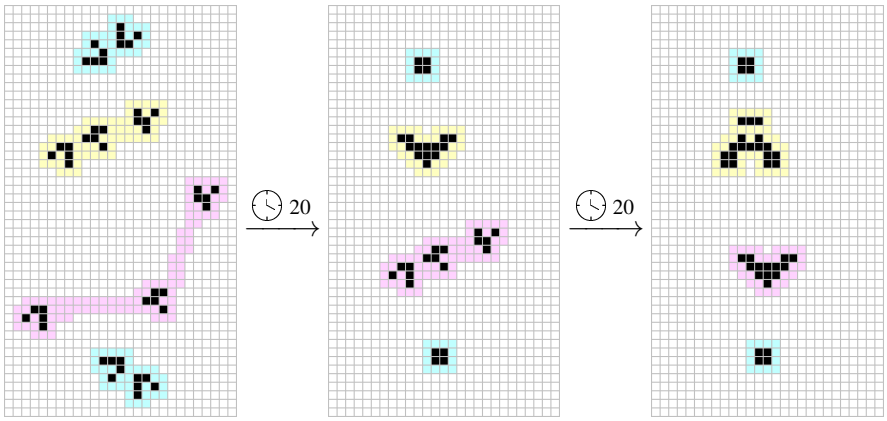


Figure 1.4: A ten-glider synthesis of a Gosper glider gun, which is composed of two syntheses of blocks (highlighted with an aqua background) and two syntheses of queen bees. The queen bee synthesis with the magenta background is the same as the one with the yellow background, but delayed by 20 generations in order to make the queen bees collide in their proper phases.

expense of a larger number of gliders. In particular, there are 4-glider syntheses of a heavyweight spaceship that consist of one glider coming from each direction, and thus are often much easier to make use of. We present a summary of useful syntheses that involve 4 or more gliders in Table 1.3.

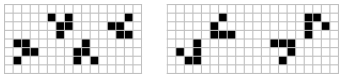
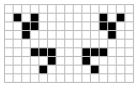
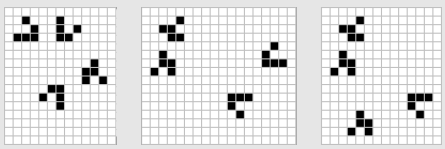
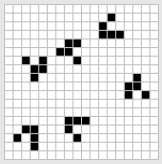
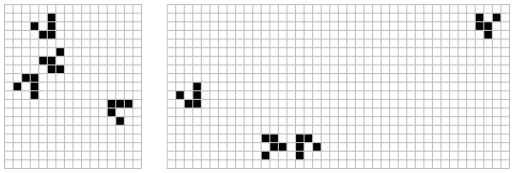
result	glider syntheses	result	synthesis
clock		twin bees	
switch engine		eater 2	
HWSS			

Table 1.3: Some useful glider syntheses involving 4 or more gliders. Most of these syntheses have been known for a long time, but the eater 2 synthesis was found by Tanner Jacobi in November 2014 (previously no synthesis with fewer than 17 gliders was known, though there were syntheses of slight variants of eater 2 with as few as 8 gliders).

1.3 Incremental Syntheses

Glider syntheses can quickly become cumbersome to use as the number of gliders involved increases. While the guns and rakes that we have already developed let us fire two streams of gliders that collide in any of the 71 possible ways illustrated in Table 1.1, as soon as three or more gliders are involved, problems can arise. If the gliders in a synthesis are very close together, it can be very difficult to pack guns or rakes together tightly enough to produce the desired stream of gliders. For example, even the task of creating a heavyweight spaceship gun using the 3-glider synthesis provided in Table 1.2 is quite challenging (see Exercise 1.8). Trying to actually make use of the ten-glider synthesis of a

Gosper glider gun displayed in Figure 1.4 would similarly be challenging, as there are just too many streams of gliders travelling along paths that are close together in the same direction.

To get around this problem, we make use of *incremental synthesis*: we build up complicated patterns by first synthesizing a small stable pattern (like a block or a pond), and then collide only a few gliders with that stable pattern to create a slightly more complicated stable pattern, and so on until we arrive at the pattern we actually want. The main benefit of this type of synthesis is that we do not need to be particularly precise with the timing of the gliders—the different stages of synthesis can take place as many generations apart from each other as we like, so we only need to coordinate a few gliders at a time.

As an example, consider the incremental synthesis of the fumarole that is presented in Figure 1.5. While there is a glider synthesis for the fumarole that involves only 7 gliders (see Exercise 1.1), this 12-glider synthesis is somewhat easier to make use of, since each stage of its synthesis requires no more than 3 synchronized gliders.

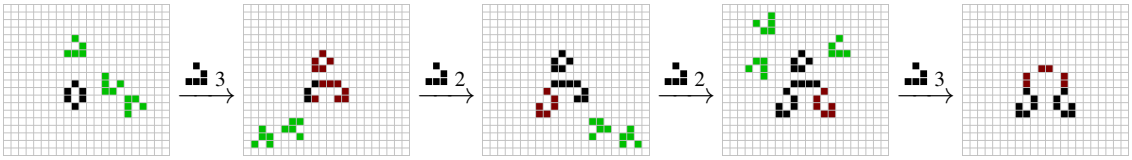


Figure 1.5: A 12-glider incremental synthesis of a fumarole (2 gliders that are not shown are required to synthesize the initial beehive). The cells that were created or modified in the previous step of the synthesis are shown in red, and the gliders that will be involved in the next collision are shown in green.

Incremental syntheses are also advantageous for the fact that we do not have to find ways of synthesizing complicated patterns from scratch, but rather we can keep a catalog of syntheses of various objects, and then to synthesize a new object we can use the most similar still life that we already know how to synthesize as our starting point. For example, the incremental synthesis of the fumarole in Figure 1.5 works by synthesizing a still life that looks similar to the stator of the fumarole (i.e., its stable bottom half), and then the last step of the synthesis creates the fumarole's rotor (i.e., its oscillating top half). This is the most commonly-used technique for synthesizing oscillators, and generally billiard table oscillators are much more difficult to synthesize than other oscillators precisely because their rotors are contained within their stators and thus this synthesis technique does not work for them.

By using the reactions listed in Table 1.4, it is straightforward to come up with an incremental synthesis of more complicated objects as well. For example, we can synthesize a Gosper glider gun in a way that never requires more than two synchronized gliders at a time by using incremental syntheses to turn a pond into a ship into a queen bee, as in Figure 1.6. We note that this synthesis uses a total of 12 gliders instead of the 10 gliders used in Figure 1.4, but we consider this a small price to pay for the benefit of not having to synchronize as many gliders in any of the steps of synthesis.

Because incremental synthesis typically works by first constructing a still life or a collection of still lifes, and then transforming those still lifes into the desired object, the Life community has gone to great lengths to synthesize still lifes. In fact, syntheses are known for all 32,538 strict still lifes with 18 or fewer live cells. Mark Niemiec maintains a large database of glider syntheses that is available at `codercontest.com/mniemiec/lifepage.htm`

1.4 Synthesis of Moving Objects

Incremental synthesis is the most useful when trying to construct objects that are made up of many simpler objects. We already demonstrated this fact when constructing glider syntheses of the Gosper glider gun, and some more perfect examples include many of the spaceships, puffers, and rakes that we introduced in Chapter **??**. These objects are mostly composed of many copies of small objects like lightweight spaceships, B-heptominoes, and switch engines that we already know how to synthesize, so we can just synthesize each component individually. However, it can sometimes be tricky to

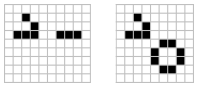
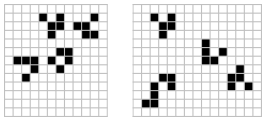
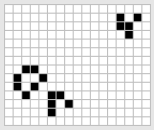
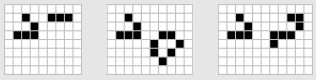
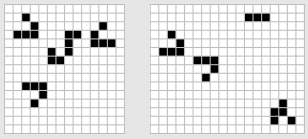
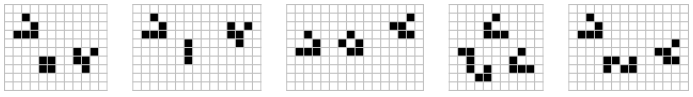
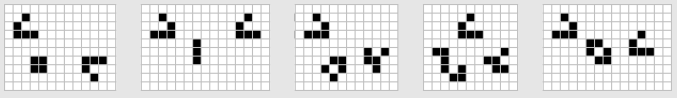
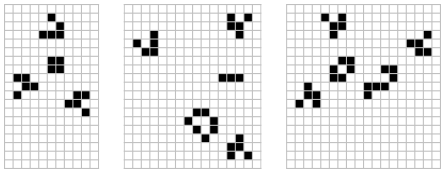
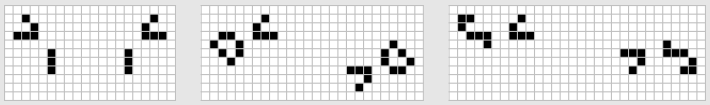
result	last step of incremental synthesis
ship	
tub	
tub with tail	
hat	
queen bee	
T-tetromino / traffic light	
honey farm	
snake	
LWSS	
MWSS	
HWSS	
pentadecathlon	

Table 1.4: A selection of useful incremental syntheses that can be used to construct many of the simple Life objects that we have seen. Note that the arrangement of a loaf and a blinker in the HWSS incremental synthesis is the exact arrangement produced by the 2-glider syntheses of a loaf and blinker in Table 1.1.

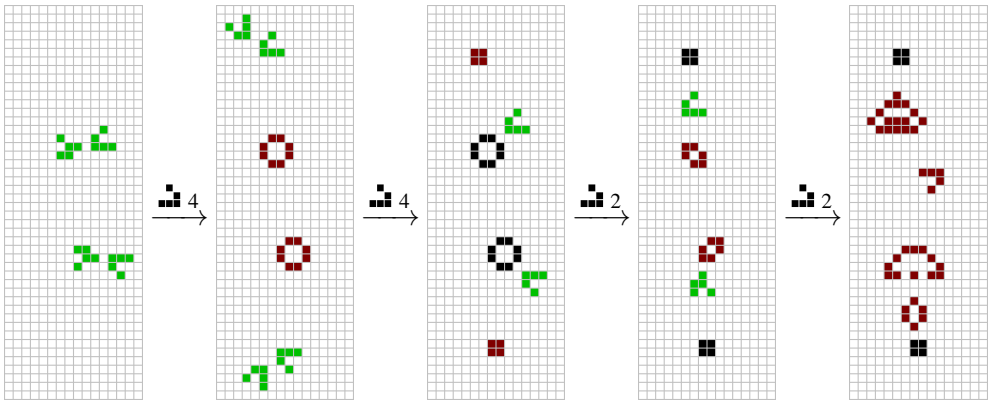


Figure 1.6: An incremental synthesis of the Gosper glider gun. Each stage in the synthesis works by either using two gliders to construct a simple still life or using just one glider to transform one object into another one. The objects that were created or modified in the previous step of the synthesis are shown in red, and the gliders that will be involved in the next collision are shown in green.

construct the various bits and pieces of these spaceships and puffers with timing that works, so we make these syntheses explicit.

1.4.1 The Ecologist and Space Rake

Recall that the ecologist of Figure **??** is made up of three lightweight spaceships and a B-heptomino, so we can synthesize it just by synthesizing those four individual pieces. The only part of the synthesis that is somewhat complicated is the construction of the B-heptomino, since it is not stable without a lightweight spaceship on both sides of it (so we cannot place it first), but there is not enough room for gliders to synthesize it between already-placed lightweight spaceships (so we cannot place it last). One solution is to synthesize the B-heptomino at the same time as the final lightweight spaceship, as in Figure 1.7.

Figure 1.7: An incremental synthesis of the ecologist. The first two steps simply use one of the three-glider syntheses of lightweight spaceships, and the third step uses a two-glider synthesis of a block. The fourth step is slightly more complicated, since we have to construct the B-heptomino at the same time that we construct the final lightweight spaceship below it. The top two gliders form the B-heptomino, while the bottom two gliders and the block form one of the incremental syntheses of a lightweight spaceship that we saw in Table 1.4.

There is another way of synthesizing the ecologist that has the advantages of never using more than two synchronized gliders at a time, and only ever using gliders that come from two of the four possible directions. The way that this synthesis works is to first create four lightweight spaceships (three in the positions that we want and one where the B-heptomino should be), each using a two-glider synthesis of a block followed by the two-glider block-to-LWSS synthesis from Table 1.4. Then we can fire one additional glider at the central lightweight spaceship to transform it into the desired B-heptomino (to actually get this final glider in the proper orientation though, we make use of the 2-glider collision that results in a glider travelling in a different direction). The tricky final stage of this synthesis is made explicit in Figure 1.8.

Once we have constructed the ecologist, it is straightforward to turn it into the space rake or its backward variant just by synthesizing the extra lightweight spaceship in the correct position, as in Figure 1.9 (also see Exercise 1.19).

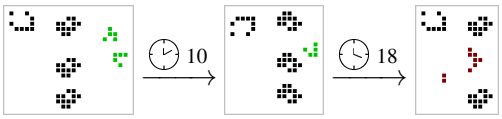


Figure 1.8: Another incremental synthesis of the ecologist. This synthesis has the advantage of never using more than two gliders at a time, and they can all arrive from the lower-left and lower-right (see Exercise 1.17). The top-left glider looks like it might collide with the lightweight spaceships before getting into the indicated position, but because the lightweight spaceships are moving in the same direction as the glider, they never actually cross paths.

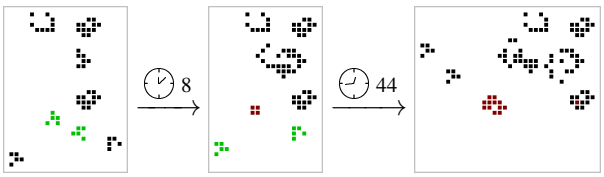


Figure 1.9: An incremental glider synthesis of a (forward) space rake. The same technique can be used to synthesize the backward space rake (see Exercise 1.19).

1.4.2 The Schick Engine and Coe Ship

To create a glider synthesis of the Schick engine from Section **??**, we recall that it is made up of two lightweight spaceships, followed by a spark that is essentially just a T-tetromino (see Exercise **??**). Thus the various syntheses of lightweight spaceships and T-tetrominoes that we have seen can be used together in a straightforward manner to synthesize the Schick engine, as in Figure 1.10—the only difficulty comes from having to carefully choose ways of synthesizing the individual components so that they do not interfere with each other.

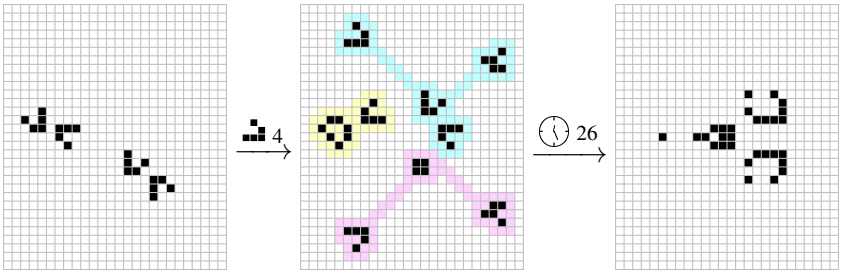


Figure 1.10: A glider synthesis of a Schick engine. Lightweight spaceships are created by the magenta and aqua glider collisions, and a T-tetromino (which becomes the Schick engine's trailing spark) is created by the loaf + glider collision outlined in yellow.

With this synthesis of the Schick engine in hand, we are also now able to synthesize the period 60 variants of the space rake that we saw in Section **??**—we just synthesize the space rake and then synthesize a Schick engine next to it in the appropriate position, or vice-versa (see Exercise 1.18(b)).

Synthesizing the Coe ship is slightly less straightforward, since we have not yet seen how to synthesize the deformed heavyweight spaceship that makes up half of it. Perhaps the simplest way to synthesize this object is to first synthesize a lightweight and middleweight spaceship next to each other, and then use two additional gliders to transform the MWSS into the deformed HWSS, as in Figure 1.11.

1.4.3 Corderships

Corderships form another family of objects that are prime candidates for glider synthesis, since they are also made up of several easily-synthesized components—switch engines. The difficult part of synthesizing a Cordership is that the switch engines have to be synchronized with each other, making an incremental synthesis difficult, and each switch engine requires at least four gliders to synthesize. Thus the 10-engine Cordership from Figure **??** would require at least 40 gliders to synthesize, and

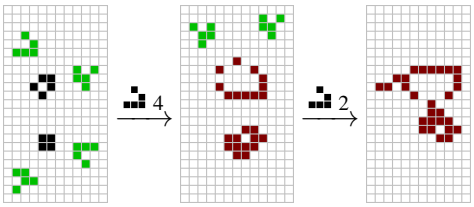


Figure 1.11: An incremental glider synthesis of a Coe ship, which works by synthesizing a lightweight spaceship next to a middleweight spaceship, and then transforming that middleweight spaceship into the necessary deformed heavyweight spaceship.

the 6 front and middle switch engines would have to be constructed at almost the exact same time, as would the 4 rear switch engines.[5]

With these difficulties in mind, it seems desirable to try synthesizing a Cordership that uses as few switch engines as possible—the 3-engine Cordership from Figure **??**. Even just with 3 switch engines to synthesize, it is somewhat challenging to find an arrangement of switch engine syntheses that can be placed close enough together without the gliders that are synthesizing different switch engines interfering with each other. Thus we use the 3-direction syntheses of the switch engine rather than the 4-direction synthesis so as to minimize the glider crossings that we have to worry about. In particular, we use the second synthesis listed in Table 1.3 to construct the central switch engine, and we use the third synthesis to construct the other two switch engines.

Unfortunately, one more small problem arises after we use these 12 gliders. While the Cordership is indeed synthesized correctly, the debris from the switch engines take a few generations to sync up with each other, and thus some unwanted junk is left behind. To prevent this junk from forming, we use three extra gliders—one for each switch engine—to suppress the debris until it matches up properly with the debris from the other switch engines. Altogether, this results in the 15-glider synthesis of the 3-engine Cordership displayed in Figure 1.12.

1.5 Developing New Syntheses

So far the glider syntheses that we have seen have been restricted to two basic types: (a) syntheses that were found just by crashing gliders together and cataloguing what objects were created, and (b) syntheses that can be made simply by piecing together the syntheses of type (a). Let's now start to branch out and construct glider syntheses of more exotic objects that do not decompose in any natural way into simpler objects that we already know how to synthesize.

Perhaps the most common way to develop new glider syntheses is to find a soup that leaves behind the pattern of interest in its ash, and try to reverse-engineer the part of the soup's evolution that led to the object's formation. To illustrate this method, consider Rich's p16 (the period 16 oscillator that we introduced in Figure **??**), which was found by evolving the soup displayed in Figure 1.13.

To develop a glider synthesis for this oscillator, we evolve the soup to just before its formation (in this case, slightly earlier than generation 196) and try to isolate the reaction that creates the oscillator. In this step, it is important to keep in mind that the goal is to go back far enough that all (or at least most) of the reactions are ones that we are familiar with, such as a T-tetromino or a queen bee colliding with some still lifes. In this particular case, if we go to generation 164 of the soup's evolution then the reaction has decomposed enough that we can identify all of the important pieces of the oscillator's creation: Rich's p16 (plus a lot of extra debris) is created when two honeyfarm predecessors and two B-heptominoes collide with two blocks and two beehives (see Figure 1.14).

We now have enough tools at our disposal to synthesize this oscillator, since we know how to synthesize blocks, beehives, honey farms, and B-heptominoes. However, the B-heptominoes in this reaction are quite messy, and create a lot of leftover debris that we would then have to clean up with several additional gliders (see Exercise 1.2). To reduce the number of required gliders somewhat, we

[5] Despite these obstacles, Stephen Silver created a 60-glider synthesis of the 10-engine Cordership in 1998.

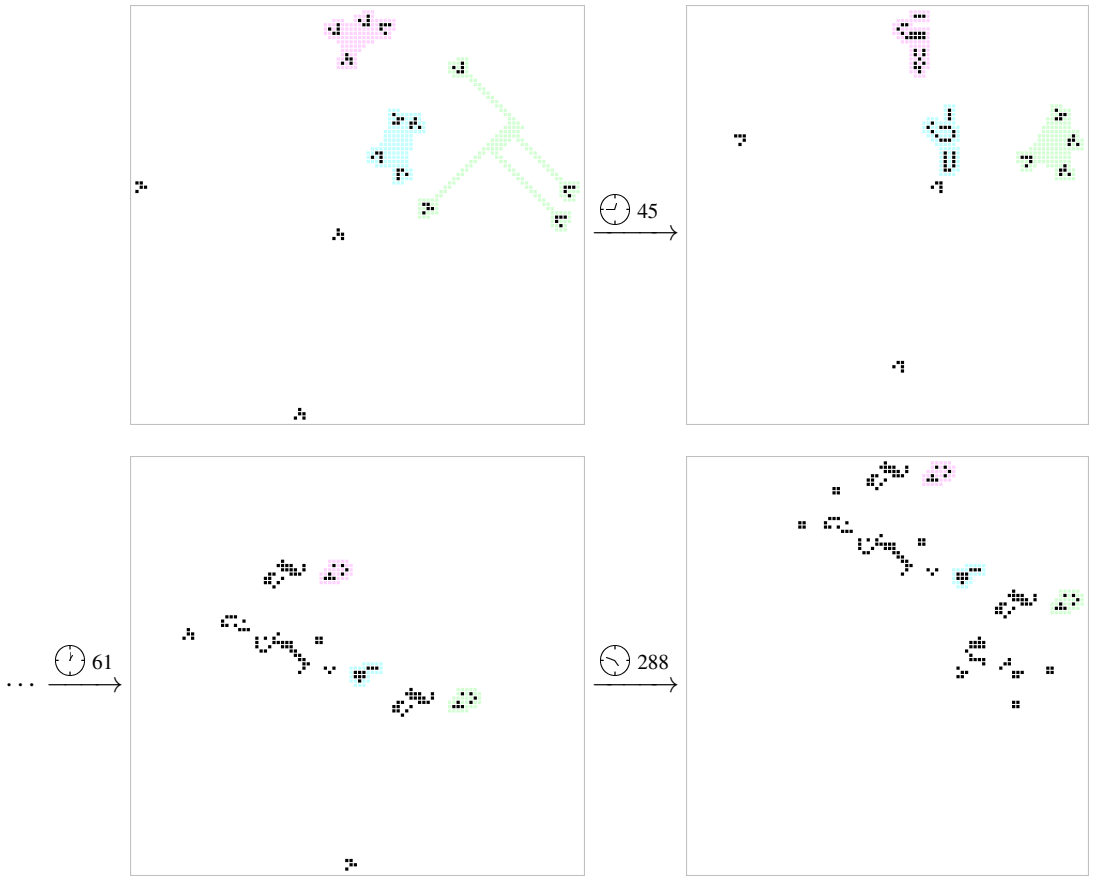


Figure 1.12: A 15-glider synthesis of the 3-engine Cordership, which was found by Jason Summers in 2004. The 3 switch engines are synthesized by the aqua, magenta, and green glider collisions. The remaining 3 gliders clean up some extra debris that is left behind by the switch engines.

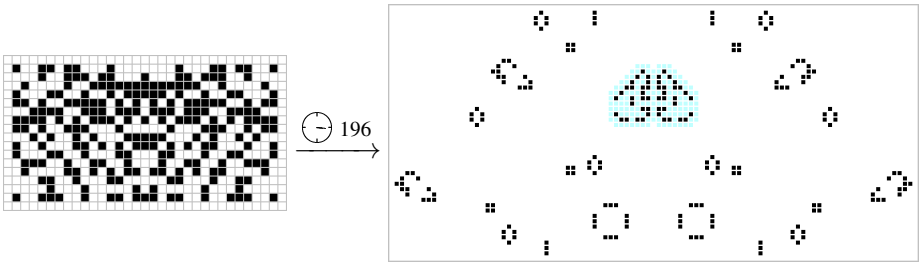
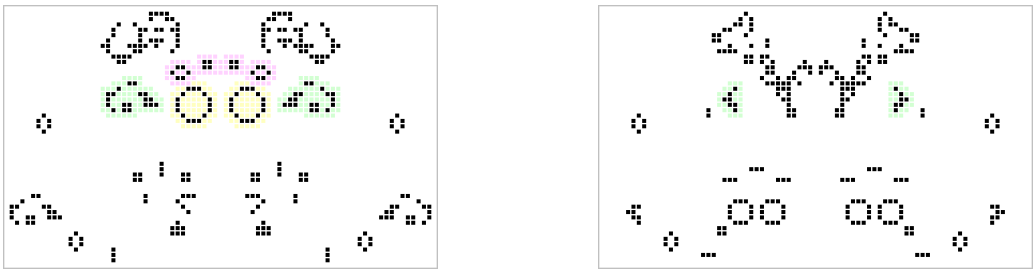


Figure 1.13: The soup that led to the discovery of Rich's p16 (highlighted on the right in aqua).

can notice that only three cells on one side of the B-heptominoes interact with the rest of the reaction, and only for two generations. Another easy-to-synthesize object with the same configuration of three cells is the eater 1, which we can thus use to replace the B-heptominoes as in Figure 1.15. Fortunately, using eater 1s in this way does result in a much smaller amount of debris being created around the p16 oscillator.

The only somewhat tricky part of synthesizing this collection of objects is creating the two honey farm predecessors, since they are unstable and thus we have to create them after creating the still lifes. Thus we opt to use the incremental honey farm synthesis from Table 1.4, so that instead of having to use 4 coordinated gliders in the final stage of synthesis (2 for each honey farm), we only need to use 2 coordinated gliders (plus 2 blocks that we can place much earlier). A complete incremental



(a) Generation 164. **(b)** Generation 155.

Figure 1.14: Rich's p16 is created when (a) two honey farm predecessors (highlighted in yellow) and two B-heptominoes (highlighted in green) collide with two blocks and two beehives (highlighted in magenta). The honey farm predecessors can be identified by comparing them with Figure **??**, and the B-heptomino can be identified by going backward to generation 155 of the soup as in (b).

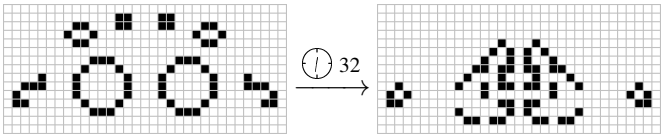


Figure 1.15: A reaction of common objects that results in Rich's p16 (and two unwanted boats).

synthesis of this oscillator is presented in Figure 1.16,[6] and we stress that there is nothing clever about this synthesis; it just makes use of the 2-glider syntheses from Table 1.1, the incremental honey farm synthesis we already mentioned, and one final glider collision (which is easily found by hand) to destroy the left-over boats.

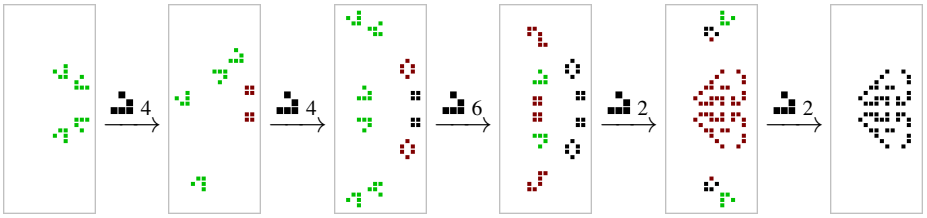


Figure 1.16: An 18-glider incremental synthesis of Rich's p16. The first three stages of synthesis just involve placing still lifes that can each be synthesized by 2 gliders. The next stage synthesizes the honey farm predecessors, and the final stage uses 2 additional gliders to destroy the 2 leftover boats. Objects that were created or modified in the previous step of the synthesis are shown in red, and the gliders that will be involved in the next collision are shown in green.

1.6 A Gosper Glider Gun Breeder

We have now learned quite a few recipes for synthesizing different objects with gliders. To demonstrate how useful these glider syntheses can be, we now switch focus a bit and start *using* these glider syntheses to create new types of patterns. In particular, we will now construct our first *breeder*: a pattern that grows quadratically[7] (as opposed to objects like guns or rakes, which grow linearly). In other words, we will construct a pattern that does not just fill up some *strip* of the Life plane, but instead fills an ever-expanding *region* of the Life plane.

The idea behind our construction is simple enough: we use the rakes we introduced in Section **??** to fire gliders in such a way that those gliders synthesize Gosper glider guns (using the incremental synthesis in Figure 1.6). Since the rakes will create Gosper glider guns at a linear rate, and the Gosper

[6] This synthesis was originally found, using the method we outlined in this section, by user "BlinkerSpawn" on the ConwayLife.com forums less than a day after the oscillator's discovery.

[7] There are actually breeders that grow slower than quadratically, and not all patterns with quadratic growth are breeders. We will return to the subject of breeders in Section **??** and give a more precise definition then.

glider guns will then each create gliders at a linear rate, the overall growth of our pattern will be quadratic.

To make this construction explicit, we first note that we cannot possibly use the period 20 space rakes directly, since they can only be used to construct objects that are 10 cells apart, and Gosper glider guns are wider than that, so they would collide and destroy each other. Thus we make use of the slightly more complicated period 60 variants of the space rake that we introduced in Section **??**, which will produce glider guns that are 30 cells apart from each other.

Now we just go through the Gosper glider gun's incremental synthesis step by step, lining up rakes so that the gliders they produce collide in the desired way. For example, for the first step of the synthesis (i.e., creating the two initial ponds), we will need four rakes, which we can position as in Figure 1.17.

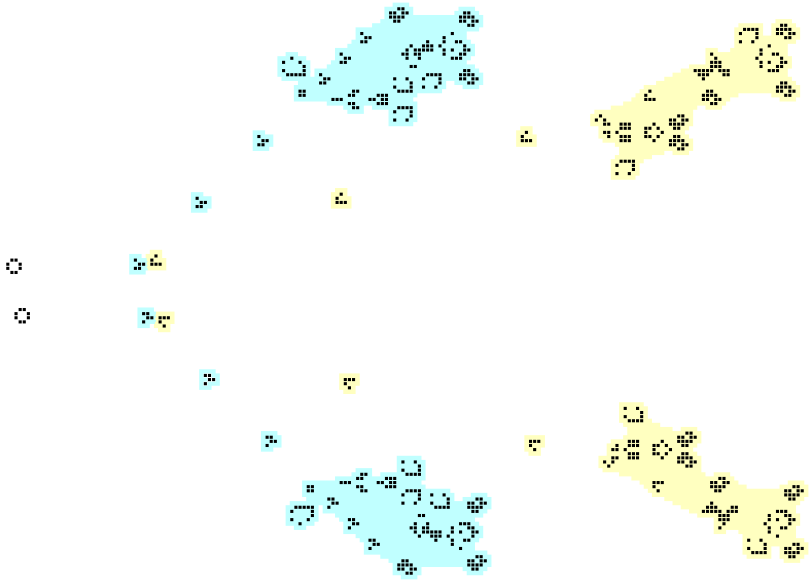


Figure 1.17: The front end of our breeder features two copies of the period 60 backward space rake (outlined in yellow), followed by two copies of the period 60 forward space rake (outlined in aqua), all traveling to the right. The gliders from these rakes collide in such a way as to create ponds, which is the first step of the incremental synthesis of a Gosper glider gun presented in Figure 1.6.

Once we have positioned those four rakes, they create two endless lines of ponds with a distance of 30 cells between each consecutive pair. At this point, we simply move on to the next portion of the glider synthesis: we need to construct two blocks near each pair of ponds, which we can do with another four rakes. Then we need to hit the ponds with a single glider each in order to turn them into ships, which we do with two more rakes. Finally, we use two more rakes to fire gliders at the pair of ships, turning each of them into queen bees and thus completing the Gosper glider gun synthesis. Our completed breeder is depicted in Figure 1.18.[8]

In generation 0 (i.e., its starting phase), this breeder has a population of 2038 cells. The quadratic growth of this breeder can be quantified by noting that every 60 generations, an additional 44-cell Gosper glider gun is created, and each of the guns creates two additional 5-cell gliders, for a total population in generation $60n$ of $5n^2 + 44n + 2038$. This quadratic growth is demonstrated in Figure 1.19, which shows the expanding triangular region of gliders behind generation 1200 of the breeder.

While this breeder is quite large, we have purposely not optimized it all, in order to make the

[8]The first breeder was found by Bill Gosper in the early 1970s, and used the exact same ideas that we used in the construction of ours. However, instead of using space rakes, it used another puffer that emits two blocks and four gliders every 64 generations. The fact that this puffer drops blocks is useful because there is then no need to synthesize them, but the trade-off is that the puffer only shoots gliders backward, not forward, so lining up other collisions is slightly more difficult.

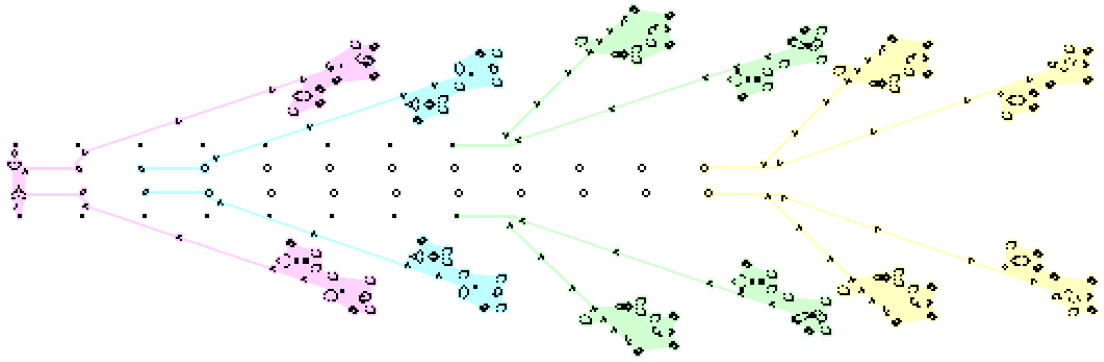


Figure 1.18: A breeder constructed using an incremental glider synthesis of a Gosper glider gun. First, a forward and backward period 60 space rake are used to collide two gliders and create a pond as in Figure 1.17 (outlined on the right in yellow), followed by another forward and backward space rake colliding two gliders to create a block (outlined in green). Next, a backward space rake uses one glider to convert the pond into a ship (outlined in aqua), and finally another backward space rake uses one glider to convert the ship into a queen bee (outlined in magenta), thus completing the synthesis of the Gosper glider gun.

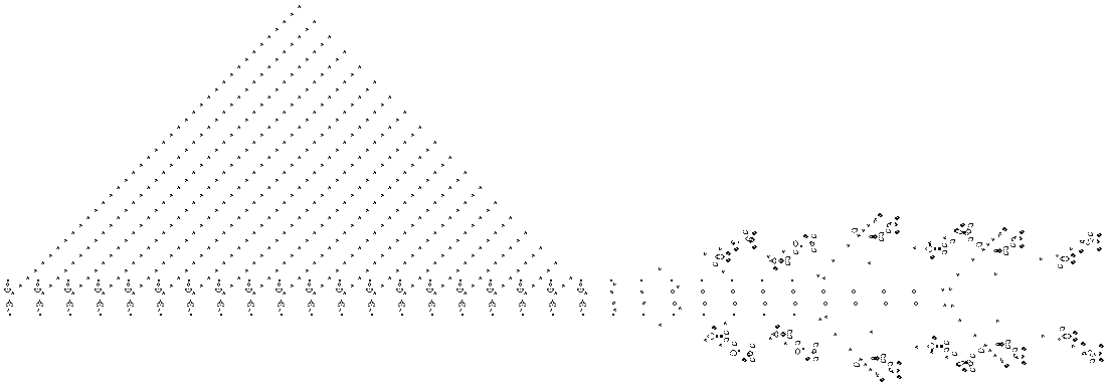


Figure 1.19: After running the breeder for 1200 generations, we see a triangle of gliders form on the left, above the line of Gosper glider guns that has been synthesized. As the breeder moves farther to the right, the triangle continues to expand.

mechanisms that make it work more apparent. The space rakes could be moved much closer to each other without compromising its functionality, and by cleverly manipulating the space rake debris it is actually possible to place more than one of the still lifes at the same time, thus halving the total number of space rakes from 12 to 6. We will present and make use of a much more compact Gosper glider gun breeder that comes from these ideas a bit later, in Section **??**.

1.7 Slow Salvo Synthesis

In Section 1.3, we saw that we can often break glider syntheses down into bits and pieces that are individually easy to understand and make use of. In this section, we take this idea to its most extreme by considering *slow salvos*, which are collections of gliders[9] that (1) are *slow*: only one glider interacts in the synthesis at a time, and (2) form a *salvo*: all of the gliders come from the same direction.

In order for the gliders in a slow salvo to be able to actually synthesize anything, we need another object (called a *seed*) for it to crash into. While we could in principle use any object, it is typical to use a block as a starting point, since it is the simplest and most symmetric stationary object. Furthermore, it is typical to only consider either *p1 slow salvos* or *p2 slow salvos*, which are slow salvos in which the intermediate objects that are created after every glider collision are still lifes or some combination

[9]Technically, a salvo can be made up of any spaceships, but the term almost always refers to gliders.

of still lifes and period 2 oscillators, respectively.[10] This perhaps seems like quite an extensive list of restrictions, but we have in fact already seen a few objects that can be created via p1 slow salvos. For example, we saw in Table 1.4 that we could use a pond as a seed in a p1 slow salvo to produce a ship, and then a queen bee.

Despite how restrictive slow salvos seem at first, it is a remarkable fact that they are exactly as general as regular glider synthesis. That is, if we can synthesize an object with gliders then we can synthesize it with a p2 slow salvo.[11] In order to pin down exactly why this is the case, we will introduce several new reactions that allow us to systematically create slow salvo syntheses from standard glider syntheses.

1.7.1 Creating and Moving Blocks

Our first step toward showing that slow salvos can construct anything that regular glider synthesis can is to demonstrate how we can create almost any arrangement of any number of blocks in the Life plane that we like. The first reaction that we will need is the one displayed in Figure 1.20, which uses a slow salvo of 2 gliders to turn a single block into two blocks.

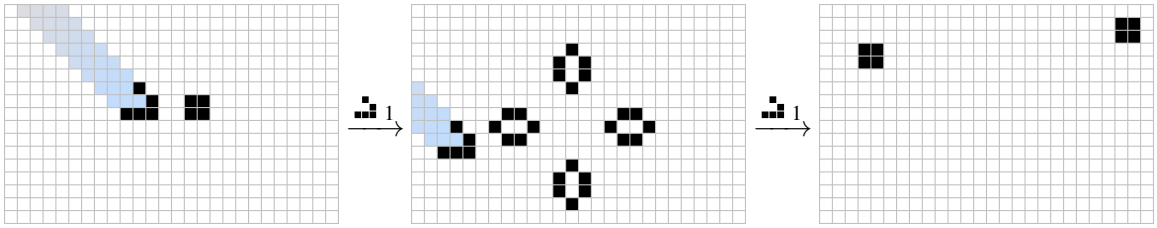
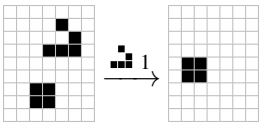
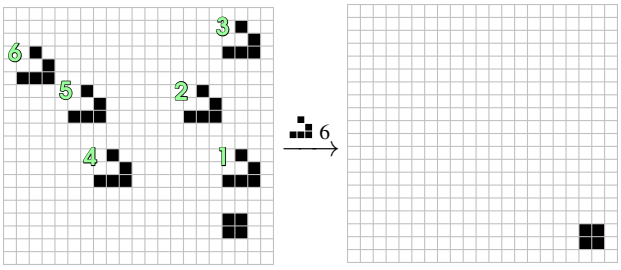


Figure 1.20: A 2-glider slow salvo can turn one block into two blocks.

While the two resulting blocks are in very different positions than the initial block, it turns out that this does not particularly matter, as we are able to use p1 slow salvos to move a block from any position to any other position on the Life plane. To show that such a movement is possible, we will use the two block-moving reactions displayed in Figure 1.21. The first of these reactions uses a single glider to move a block left 1 cell and up 2 cells (in fact, this is the exact same block-moving reaction that we already saw way back in Figure **??**), and the second reaction uses six gliders (in a p1 slow salvo) to move a block down and to the right 1 cell.



(a) A single glider can move a block to the left by 1 cell and up by 2 cells.



(b) A 6-glider slow salvo moving a block down and right by 1 cell.

Figure 1.21: Two methods of moving a block around with p1 slow salvos that can be repeated in order to move a block to any position on the Life plane. The order in which the gliders should come in is indicated by the green numbers—their exact timing does not matter, as long as they are far enough apart that each collision settles down before the next glider arrives.

By combining these two reactions, we can move a block a single cell up, down, left, or right. For

[10] P2 salvos are advantageous because blinkers are so easy to synthesize, and we will see shortly that the clock is also a useful tool when working with slow salvos. We could analogously consider pn slow salvos for some $n \geq 3$, but in practice not much is gained by doing so, since objects with period 3 or higher typically aren't any easier to create in intermediate stages of synthesis than objects period 1 or period 2.

[11] However, the slow salvo might contain considerably more gliders than other glider syntheses.

example, to move a block up a single cell, we could perform the movement in Figure 1.21(a), followed by the movement in Figure 1.21(b) (for a total cost of 7 gliders). Similarly,

- To move a block right by a single cell, perform the movement in Figure 1.21(a), followed by the movement in Figure 1.21(b) twice.[12]
- To move a block left by a single cell, perform the movement in Figure 1.21(a), but reflected along the diagonal from the top-left to the bottom right. Then perform the movement in Figure 1.21(b).
- To move a block down by a single cell, first move it left by a single cell and then perform the movement in Figure 1.21(b).

Now that we know how to use p1 slow salvos to move a block by a single cell in any direction, we can easily repeat these reactions over and over until the block is moved to whatever position we desire.[13]

1.7.2 One-Time Turners

Just like we can use gliders to create and move blocks around the Life plane, we can also use blocks to move gliders around the Life plane. Thus we can use a p1 slow salvo to create arrangements of blocks that then change the direction of other gliders in the salvo, thus creating salvos that come from multiple directions.

To be a bit more explicit, consider the arrangement of blocks presented in Figure 1.22, which can be used to rotate a glider by 90 degrees, but is destroyed in the process (contrast this with reflectors like the Snark, which rotate gliders but remain unchanged). Reflectors like this are called *one-time turners*, and they will be invaluable for us when using slow salvos to emulate arbitrary glider syntheses. There are also numerous other 90-degree one-time turners involving various numbers of blocks (and sometimes other small still lifes), but this turner involving just two blocks is perhaps the simplest, and for now it will be enough for our purposes.

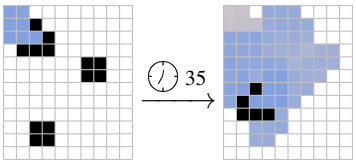


Figure 1.22: A *one-time turner* composed of two blocks that rotates a glider by 90 degrees and destroys the blocks at the same time.

Importantly, because this one-time turner is made up of nothing other than blocks, we can construct it using a p1 slow salvo via the techniques that we developed in the previous section. An explicit 12-glider p1 slow salvo that does the job (and also leaves us with an extra block that we can then use to construct additional turners) is displayed in Figure 1.23. This salvo works by using 2 gliders to duplicate the initial block (as in Figure 1.20), 8 gliders to move one of these blocks to a new position, and then 2 more gliders to duplicate this block again (and due to the clever way we repositioned the second block, this second duplication leaves a pair of blocks in exactly the one-time turner configuration).

Thus we can use a total of 13 gliders in a p1 slow salvo (12 to create the one-time turner and 1 to use and destroy it) to create one glider in a perpendicular direction, while still having a block

[12] This requires a total of 13 gliders, which is far from optimal—another reaction is known that accomplishes the same movement in only 7 total gliders. However, we are just interested in presenting the conceptually simplest method of moving blocks, not necessarily the most efficient method. Paul Chapman and Dave Greene put together an extensive table of the cheapest known block-moving slow salvos, which is available at `b3s23life.blogspot.com/2004/09/glues-slow-salvo-block-move-table.html` (note that the values in their table assume that the gliders come from the bottom-right, rather than the top-left). A slightly more up-to-date table that makes use of p2 salvos rather than just p1 salvos is available at `conwaylife.com/forums/viewtopic.php?f=7&t=1072#p8182`

[13] In linear algebra terms, what we have shown is that every point in $\mathbb{Z}^2$ is a positive linear combination of the vectors $(-1,2)$, $(-2,1)$, and $(1,-1)$.

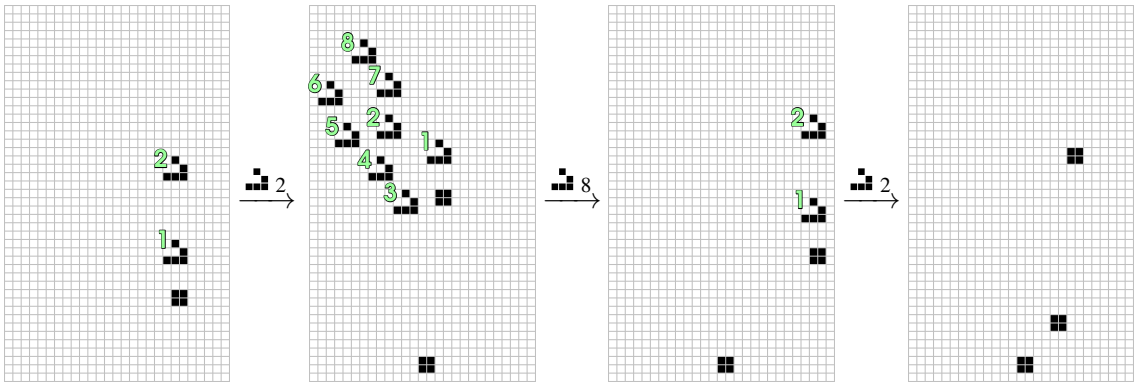


Figure 1.23: A p1 slow salvo of 12 gliders can be used to create the one-time turner from Figure 1.22 and leave us with another block left over (to create additional turners or other objects with). The first two gliders duplicate the initial block, using the reaction from Figure 1.20. The next 8 gliders move one of the blocks down and to the right, and then the final two gliders duplicate that moved block. The order in which the gliders should come in is indicated by the green numbers—their exact timing does not matter, as long as they are far enough apart that each collision settles down before the next glider arrives.

left over to perform additional constructions with. Using this technique, we are now able to emulate multi-directional glider syntheses from unidirectional ones. For example, consider the four-glider syntheses of the clock that we saw in Table 1.3, which use one glider coming from each of the four possible directions. A unidirectional synthesis of the clock can be constructed by using a p1 slow salvo to construct four one-time turners (one of the gliders does not need to be turned at all, two gliders need to be turned once each, and one glider needs to be turned twice) and then firing four gliders at the one-time turners with the proper timing. An explicit configuration of one-time turners that works is displayed in Figure 1.24.[14]

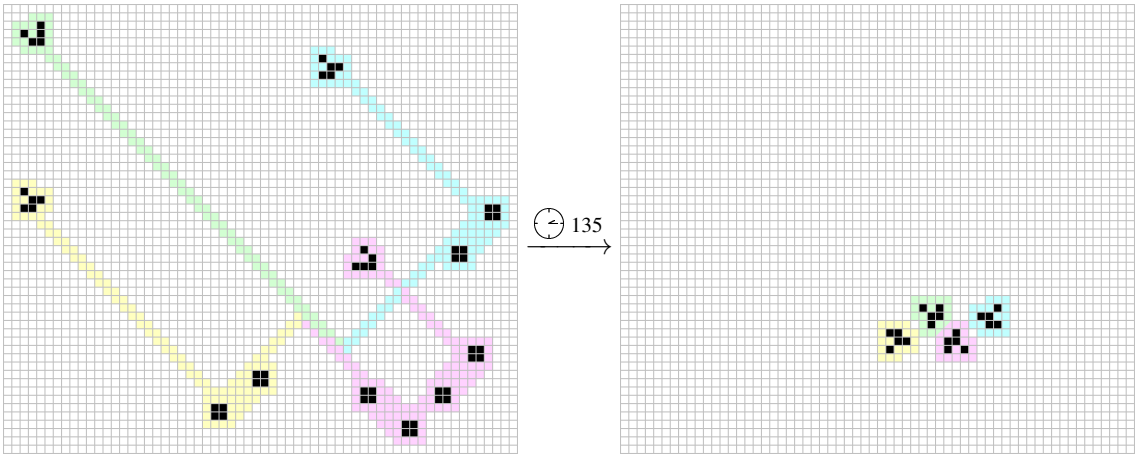


Figure 1.24: One-time turners (which can be synthesized with a p1 slow salvo) can be used to allow unidirectional glider waves (like the one on the left) to emulate multi-directional glider syntheses (like the one on the right). The gliders on the right are in exactly the position of the clock synthesis that we saw in Table 1.3.

Unfortunately, this construction does not give us a true p1 slow salvo for constructing a clock, since the final four gliders in the synthesis (i.e., the ones that hit the one-time turners) must be synchronized with each other—we are not free to space them arbitrarily far from one another. In order to fix this problem, we will need to introduce some methods for using blocks to duplicate gliders and then adjust the timing of those gliders.

[14] We do not explicitly demonstrate how to create this arrangement of blocks using a p1 slow salvo, since it would require dozens of gliders.

1.7.3 Splitters and Timing

The first ingredient that we will need in order avoid having to send synchronized gliders at our one-time turners is a method of turning one glider into many gliders. Just like our one-time turner, we would like this pattern to be composed entirely of blocks, and we would like it to be cleanly destroyed during the glider-duplicating reaction. Patterns with these properties are called *blockic splitters*,[15] and one example is provided in Figure 1.25.

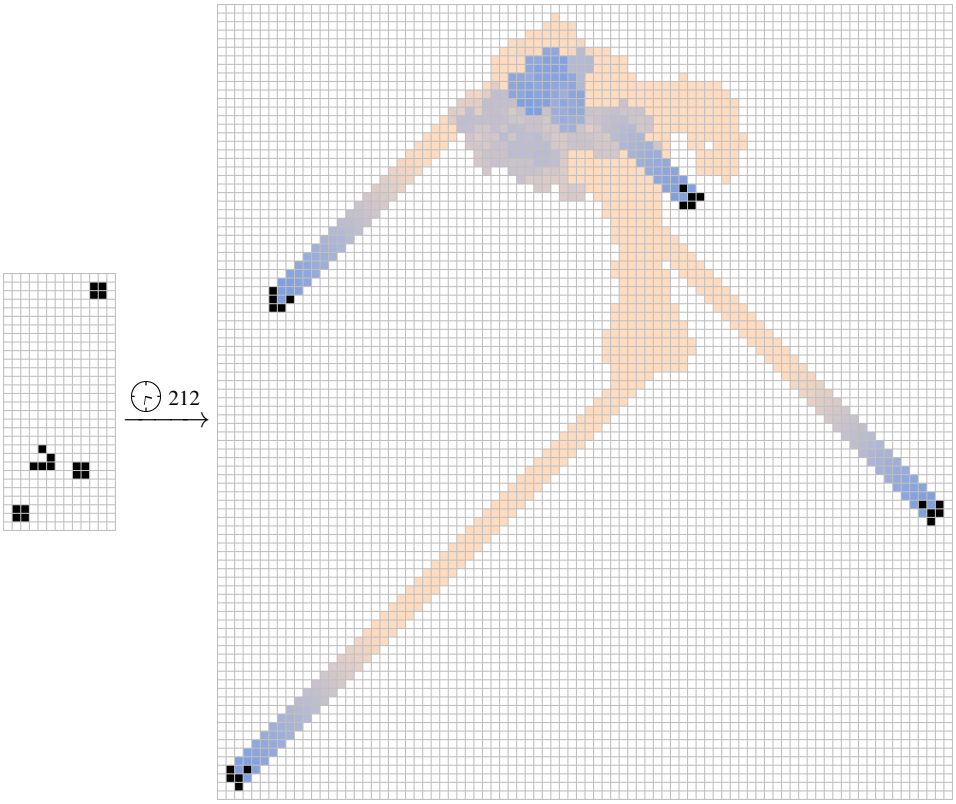


Figure 1.25: A *blockic splitter,* which uses three blocks to turn one glider into four gliders (while destroying the blocks in the process).

This splitter *almost* gets us what we need—it lets us turn one glider into four gliders, and we could chain multiple splitters together to turn one glider into as many gliders as we desire. We can use one-time turners to move these gliders around the Life plane, but unfortunately we are faced with two brand new problems:

- The one-time turner that we introduced in Figure 1.22 preserves the glider's color, so we have no way of obtaining glider color combinations other than those provided to us by the splitter in Figure 1.25 (in which the bottom-left glider has the same color as the input glider, whereas the other 3 gliders have the opposite color). Since the 4-glider synthesis of a clock uses 2 gliders of each color, in order to make a slow salvo synthesis of the clock we would have to use this splitter multiple times and then use blocks to delete the excess gliders, which is quite wasteful.
- Even though the one-time turner in Figure 1.22 is pretty good at moving gliders around the plane (ignoring the color issue that we just discussed), it gives us no control over timing, and we need some way of making sure that all of the desired gliders not only arrive at the right place, but they also arrive there at the same time.

Fortunately, there is a common solution to both of the above problems, and it is simply to introduce several new one-time turners. In particular, the family of 180-degree one-time turners presented in Table 1.5 are capable of either preserving or changing the color of a glider, and also offsetting its

[15] The term *blockic* refers to the fact that it is composed entirely of blocks. There are also splitters (and one-time turners) made up of other still lifes (or even blinkers), but we do not consider them here since blocks are enough for our purposes.

timing by any amount that we desire.[16]

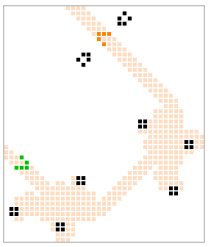
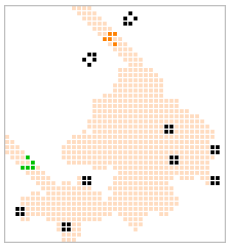
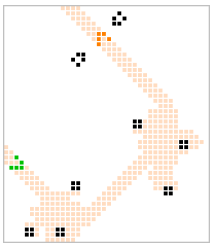
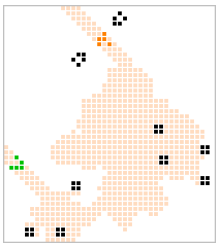
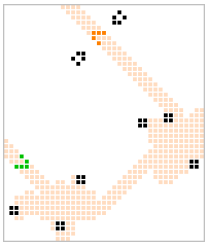
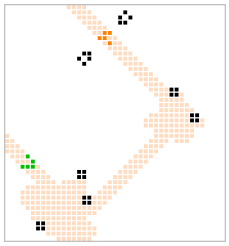
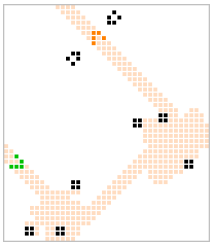
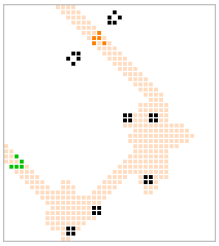
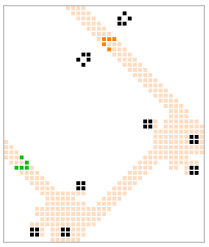
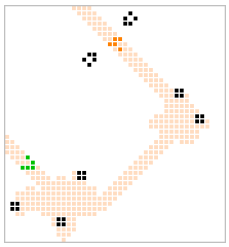
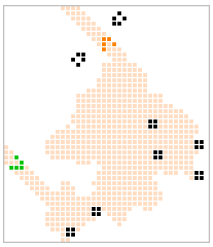
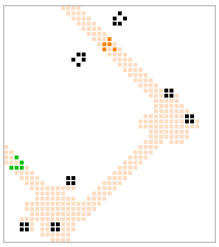
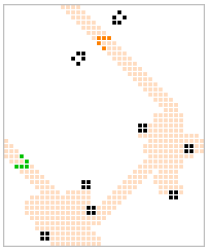
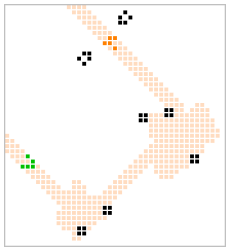
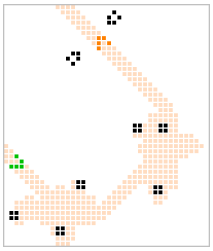
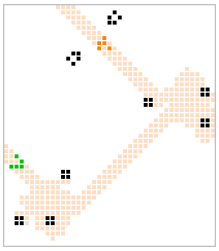
	Offset			
	0	1	2	3
Color-Preserving				
Color-Changing				
	4	5	6	7
Color-Preserving				
Color-Changing				

Table 1.5: By using different 90-degree one-time turners together, we can create this collection of 180-degree one-time turners that can be used to set a glider to any timing and color. In all cases, the input glider is highlighted in green and comes in from the top-left, while the location of the output glider exactly 200 generations later is highlighted in orange. The boats are not required for the turners to function, but just serve to make it easier to line up multiple different turners.

In order to make use of these 180-degree one-time turners to get gliders in (almost) any position and timing that we want, first simply use the 90-degree one-time turner in Figure 1.22 and the 180-degree one-time turners with offset 0 in Table 1.5 to put the gliders into their correct positions. Importantly, be sure to place at least one 180-degree turner in the path of each glider, even if it is not required to get the positioning right (we will need that 180-degree turner to get the timing right momentarily).

Even though the positioning of the gliders is now correct, their timing will likely be horribly wrong. To fix this problem, focus on the glider that gets to its destination last—we will synchronize all

[16] This collection of 180-degree one-time turners was compiled by Dave Greene.

of the other gliders with this one. To slow down the other gliders, we note that moving a 180-degree turner 1 cell father away causes its glider to reach its destination 8 generations later (it now has to travel 1 cell, and thus 4 generations, farther in both directions). Thus to delay a glider by n generations, we can move its 180-degree turner father away by $\lfloor n/8 \rfloor$ generations and then replace it by the turner from Table 1.5 that has the same effect on its color but has offset $n \pmod 8$.

Since we have a complete set of glider-preserving or glider-changing turners capable of delaying a glider by any of $0, 1, 2, \ldots, 7$ generations, we thus now have all the tools we need to position *and* time gliders as we see fit. An example of how we can use the 4 gliders produced by the splitter in Figure 1.25 to synthesize a clock is presented in Figure 1.26. Note that we placed a 180-degree one-time turner in the path of each of the four gliders so that we could use the technique for correcting their timing described in the previous paragraph, and since each of those 180-degree turners is simply made up of two 90-degree one-time turners, we are able to separate their two halves in order to have greater control over the glider's output position.

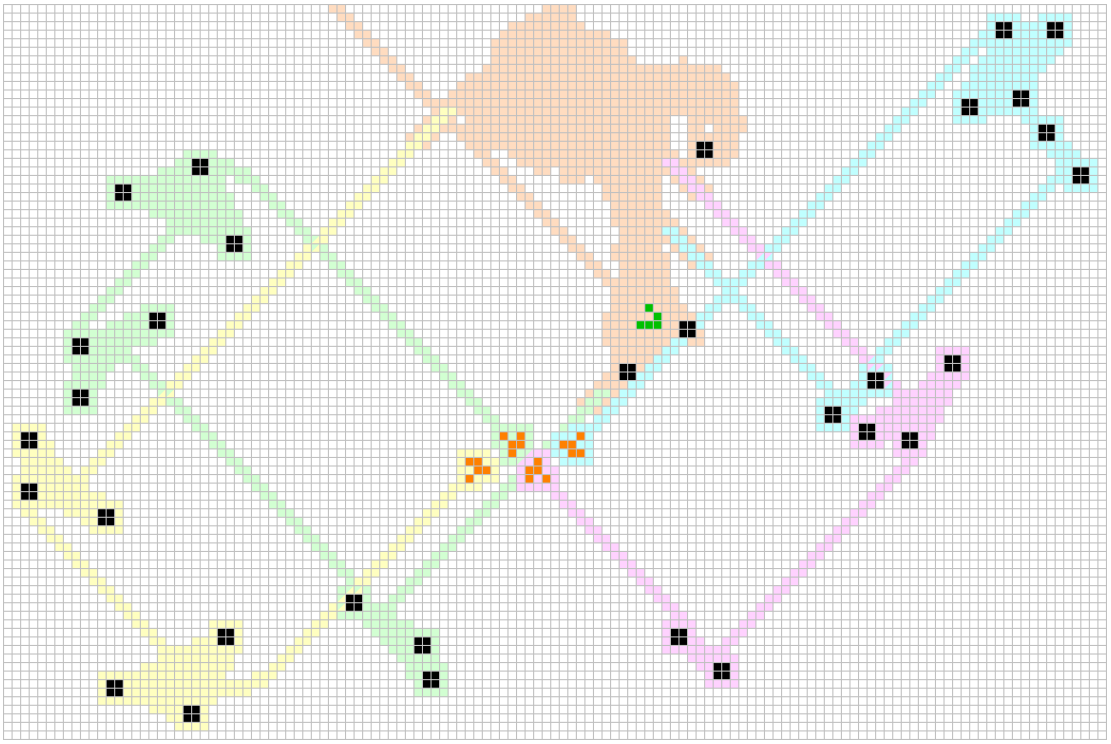


Figure 1.26: A *blockic seed* for a clock: The blockic splitter at the top-center (highlighted in orange) turns the single incoming glider into four gliders whose paths are outlined in yellow, green, aqua, and magenta. Those gliders are repeatedly reflected via one-time turners in such a way that they end up in the indicated positions required to synthesize a clock after 537 generations. Note that we could probably get away with using fewer one-time turners, but by placing a 180-degree turner in the path of every glider it is much easier to get their timing right.

The arrangement of blocks shown in Figure 1.26 immediately gives a p1 slow salvo for constructing a clock—we "just" have to use a p1 slow salvo to construct the blocks (using the block moving and block duplicating techniques we discussed earlier) and then fire one additional glider to trigger the synthesis. However, a salvo for constructing that arrangement of blocks would contain dozens of gliders—recall that we already needed 12 gliders just to create a single 2-block one-time turner in Figure 1.23. Thus we re-emphasize at this point that even though these techniques show how to turn arbitrary glider syntheses into slow salvo syntheses, they are not efficient. Even very small syntheses can require hundreds or thousands of gliders to emulate in a p1 slow salvo.[17]

[17] As an example, Dave Greene constructed an explicit p1 slow salvo consisting of 997 gliders that constructs a configuration of 81 blocks that, when hit by one additional glider, transforms into an 8-glider synthesis of the loafer spaceship. This salvo can be seen at conwaylife.com/forums/viewtopic.php?f=2&t=1006&p=7574#p7574.

1.7.4 Tight Packings of Gliders

There is still one final problem that might arise when trying to emulate an arbitrary glider synthesis with a slow salvo synthesis, and that is the fact that some syntheses involve packings of gliders coming from the same direction that are too close together for us to place with our one-time turners. This was not a problem for the clock synthesis that we have been using since each of the gliders in the final synthesis comes from a different direction. However, if we were to try to use a p1 slow salvo to emulate the 3-glider synthesis of a HWSS displayed in Table 1.2, it is not obvious whether or not we can position the two gliders that are heading southwest close enough to each other (since the various one-time turners that we have access to might interfere with one of the gliders when trying to place the other one). To fix this glider-packing problem, we introduce one final reaction: the *clock inserter*,[18] which uses two opposing gliders to transform a clock into a perpendicular glider, as in Figure 1.27.

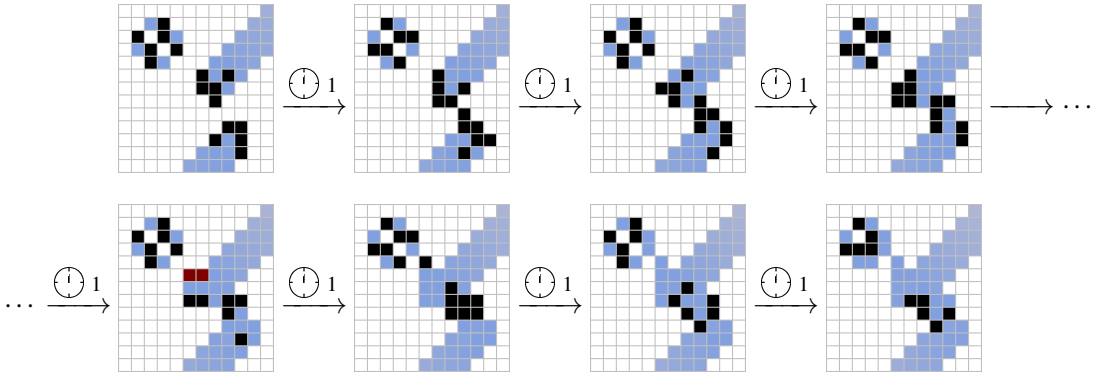


Figure 1.27: The *clock inserter* is a reaction that very cleanly collides two gliders in such a way as to transform a clock into a perpendicular glider. The two-glider collision produces a domino spark in generation 5 (shown in red) and then dies off, while the domino spark initiates the clock-to-glider transformation. The debris at the top-left in generation 8 dies off in 5 more generations.

The key facts that make this reaction so useful for us are (1) that the output glider appears at the back of the reaction, behind the input gliders, and (2) the reaction is almost effortless—it does not disrupt any other cells around the clock and output glider, and thus does not disrupt other nearby gliders either. Thus this reaction can be used to place a glider very close in front of, or to the side of, another already-placed glider (see Figure 1.28 for an example of how closely it can place a new glider to other gliders). Because of this, we can use this reaction to build any arrangement of gliders, no matter how tight, as long as we start by placing the gliders in the back first. Actually *proving* that the clock inserter reaction can be used to build any arrangement of gliders is somewhat technical and messy,[19] since there are many possible ways for gliders to be near each other, so we defer the proof to Appendix **??**.

With this reaction in hand, we can now use a p2 slow salvo to construct any arrangement of gliders in the plane that we desire. First, we "rewind" the desired glider synthesis back as far as we like, so that it consists of four salvos of gliders—one coming from each direction.[20] We then construct the four salvos one glider at a time. If the gliders in a salvo are spaced sufficiently far apart, we can insert gliders into it via one-time turners. If the gliders are spaced close together, we instead first synthesize a clock (via p1 slow salvo synthesis, such as in Figure 1.26) and then use the clock inserter to insert the close gliders one at a time. We have thus finally proved the following theorem:

[18] Found by Martin Grant in December 2014.

[19] Chris Cain wrote a script that automatically uses clock inserters to build glider arrangements in 2014, and then he and Dave Greene used the script to build such a wide array of tight glider arrangements that this could probably be considered the first proof that p2 slow salvos can build any configuration of gliders. Cain's script can be found at `conwaylife.com/forums/viewtopic.php?f=2&t=1512&start=25#p15133`.

[20] Recall that the gliders must be able to reach their positions from arbitrarily far away in order to be considered a valid glider synthesis.

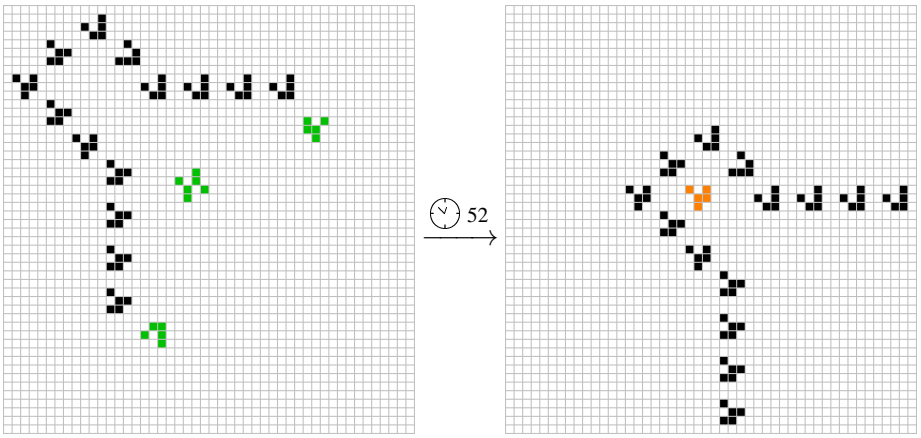


Figure 1.28: A demonstration of how the clock inserter (shown in green on the left) can be used to place a glider (shown in orange on the right) closely beside and in front of other already-placed gliders.

Theorem 1.1 Every pattern that can be constructed via glider synthesis can be constructed by a p2 slow salvo glider synthesis.

The only p2 object that we needed to prove Theorem 1.1 was the clock, which we used to place clusters of tightly-packed gliders. The are other reactions involving only p1 objects that can similarly be used to place gliders close to each other (see Exercise 1.34), but none of them are quite as clean, simple, and universal as the clock inserter. In particular, there is no known p1 inserter that works for all tight glider packings—some packings require one p1 inserter, while other packings require a different p1 inserter. For this reason, it is widely accepted that we can replace "p2" in Theorem 1.1 with "p1", but pinning down the details is messy enough that no one has made the proof explicit.

Notes and Historical Remarks

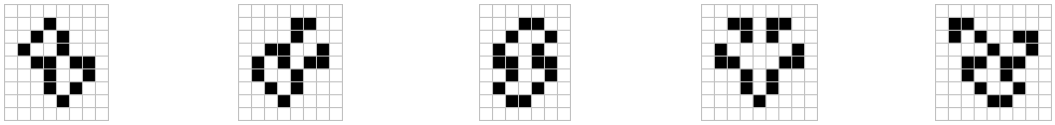
The importance of glider synthesis was known essentially as soon as the glider itself was found in 1970, with common folklore being that we could send gliders as signals throughout the Life plane and collide those gliders in different ways to simulate arbitrary computations. This basic idea has been refined and made more precise repeatedly over the past 45 years, to the point that there are now explicit patterns that do exactly this—they collide gliders so as to perform arbitrary computations and build almost any pattern of our choosing (we will delve deeply into the specifics of how these patterns work in Chapter ??).

The first specific and explicit uses of glider syntheses were demonstrated in 1971, when Bill Gosper constructed the first breeder (essentially the breeder that we built in Section 1.6) as well as the first lightweight and middleweight spaceship guns. These patterns demonstrated the kind of leap in complexity that is possible in patterns when taking advantage of glider synthesis, and it led to a surge in interest in the topic over the following years.

By 1973, the majority of "basic" Life objects were synthesizable, including lightweight, middleweight, and heavyweight spaceships, switch engines (and their block-laying and glider-producing counterparts), commonly-occurring oscillators like the pentadecathlon and pulsar, and all still lifes and oscillators with 7 or fewer cells other than the clock and the long snake. Syntheses of larger composite patterns were even being discovered by this point, with Douglas Petrie constructing the 11-glider synthesis of the Schick engine displayed in Figure 1.10 in 1973. The majority of these early syntheses were developed by David Buckingham, Mark Niemiec, and Douglas Petrie.

Over the following decades, David Buckingham continued to develop syntheses for still lifes and oscillators, and he completed syntheses of all of them with 14 or fewer cells by no later than 1992. His technique was to make heavy use of incremental synthesis, building the objects from the inside

out, only using a couple of gliders at a time to tweak the outermost portion of the object that he was synthesizing. This method works very well when synthesizing objects that have "end pieces" like tails that can be removed or altered without affecting the pattern's stability. However, compact still lifes like the one in Figure 1.29(a) are much more difficult to construct, and this particular still life (which was the last 14-cell still life to be synthesized) was initially synthesized using a massive incremental synthesis involving more than 30 gliders.



(a) 14 cells. **(b)** 15 cells. **(c)** 16 cells. **(d)** 17 cells. **(e)** 18 cells.

Figure 1.29: The final still lifes with 14–18 cells to be synthesized by gliders.

Mark Niemiec continued on with this work, creating a large database that he used to automatically generate syntheses of thousands of still lifes by piecing together known reactions [Nie03, Nie10]. This greatly reduced the number of still life syntheses that needed to be found by hand, and in 2013 he completed syntheses of all 15-cell still lifes (as well as syntheses for all except for a few hundred 16-, 17-, and 18-cell still lifes).

Glider syntheses for the remaining 16-cell still lifes were then found via a collaborative effort on the ConwayLife.com forums, mostly led by Martin Grant, Matthias Merzenich, and Mark Niemiec, with the final synthesis (see Figure 1.29(c)) being completed in January 2014. Another five-month collaborative effort, led by the same group of people, completed syntheses of the 17-cell still lifes in May 2014. Finally, a comparatively quick two-month effort finished the remaining syntheses of 18-cell still lifes in November of that year.

To give an idea of the size of this achievement, recall from Table **??** that there are 32,538 different strict still lifes with 18 or fewer cells. Not only is this a huge number of distinct objects to synthesize, but the syntheses themselves are often monstrously large. For example, Figure 1.30 shows how one of the "problematic" 17-cell still lifes was constructed by using a whopping total of 94 gliders and over 20 stages of incremental synthesis.

Rather than trying to push these techniques to synthesize all still lifes with 19 live cells, focus has shifted somewhat to trying to reduce the number of gliders required to synthesize these objects. It has been known for decades how to synthesize all still lifes with 8 or fewer cells via 4 or fewer gliders. Similarly, glider syntheses are known for constructing every still life with $9, 10, 11, 12$, and 13 live cells via $5, 9, 9, 11$, and 12 or fewer gliders, respectively.[21] Recent efforts have attempted to continue this pattern and synthesize all small still lifes in fewer than 1 glider per live cell. This project was completed for 14-, 15-, 16-, and 17-cell still lifes in October 2016, November 2016, May 2017, and September 2019, respectively. For example, we now know how to construct the 17-cell still life from Figure 1.30 (which was first synthesized by 94 gliders) via just 10 gliders—see Figure 1.31.

It remains an open question whether or not every still life (or every oscillator, or even every pattern that has predecessors) can be constructed by glider synthesis. Closely related is the (also open) question of whether or not there exists a still life whose only parent is itself.[22] Finding such a pattern would answer both of these questions, since if a still life's only parent is itself then it can not be constructed via glider synthesis (or any other means).

Exercises

solutions on page **??**

[21] It was not known how to synthesize the 9-cell long^3 snake with fewer than 6 gliders until February 2015, when Matthias Merzenich derived a 5-glider synthesis from an apgsearch soup.

[22] For the purposes of this problem, a still life together with far away non-interacting dying ash is not considered a different parent than just the still life itself. Conway offered a $50 prize for a solution to this problem in October 1972, which has gone unclaimed to this day.

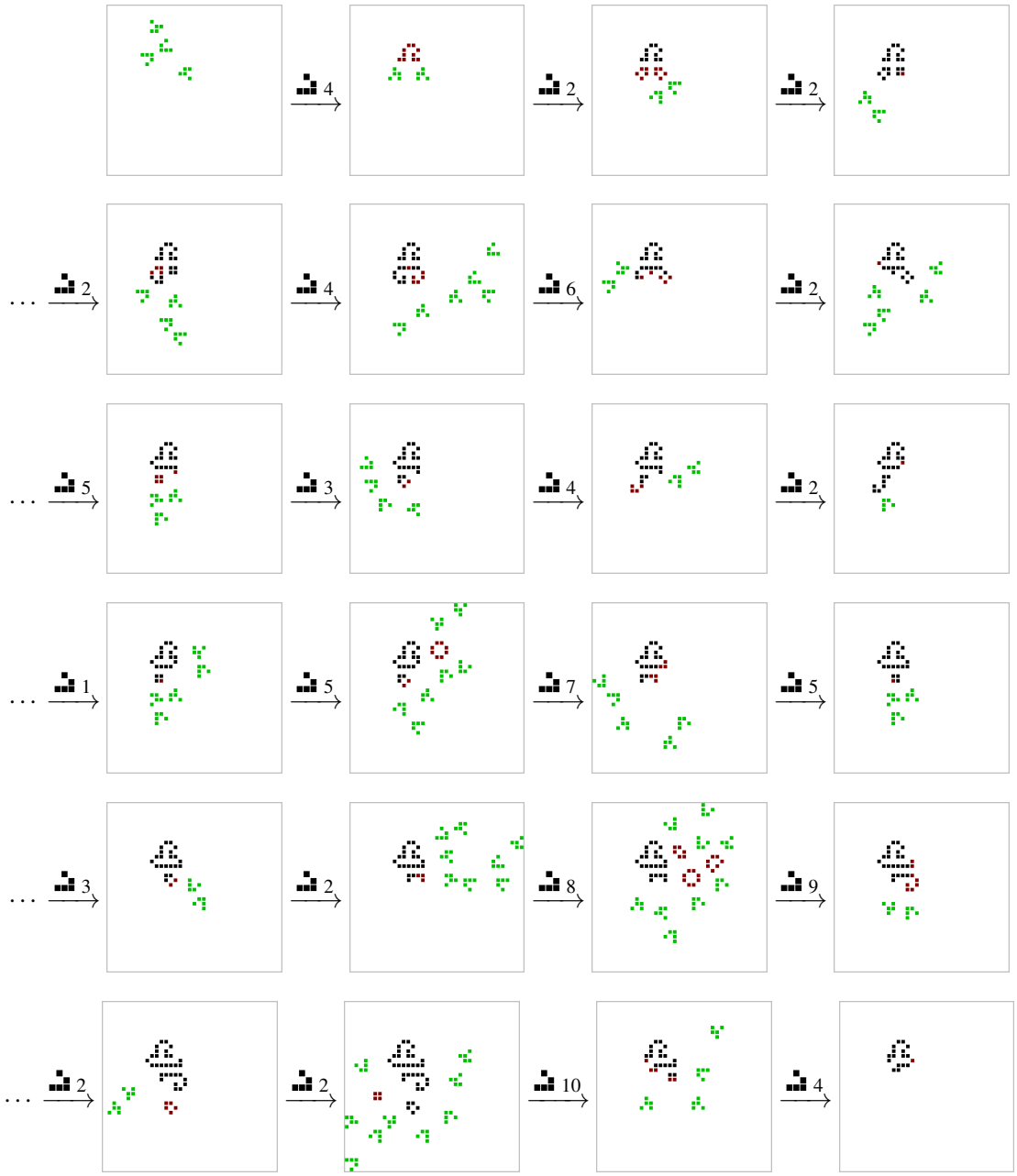


Figure 1.30: A summary of a massive 94-glider incremental synthesis of a hard-to-construct 17-cell still life (shown at the bottom right). Each stage in the synthesis works by using another glider collision to transform one still life into another. The cells that were created or modified in the previous step of the synthesis are shown in red, and the gliders that will be involved in the next collision are shown in green.

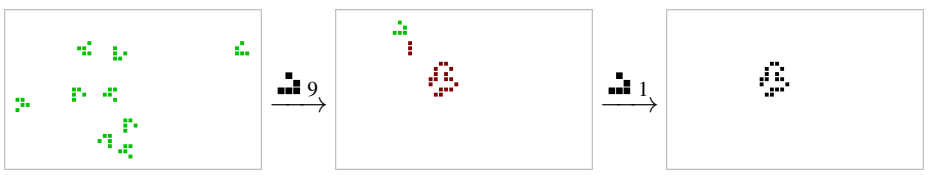
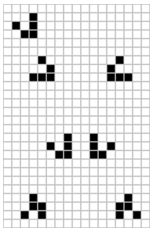


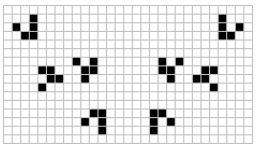
Figure 1.31: A 10-glider synthesis of the 17-cell still life from Figure 1.30 that was constructed with the help of a soup that was found by apgsearch.

1.1 Many glider syntheses work by using a small number of gliders to create the desired object plus some debris, and then additional gliders to clean up the debris. In each of the following syntheses, identify which gliders are used to clean up debris.

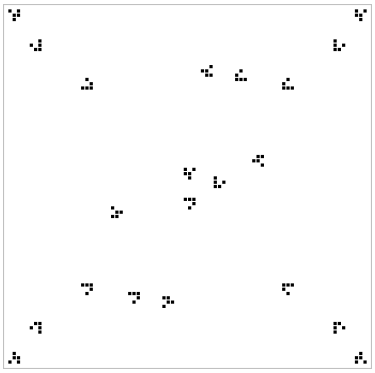
(a)



(b)



(c)



1.2 Gliders can be used to destroy essentially any unwanted debris that is left over after a synthesis. Use a single glider to destroy each of the following objects (and also destroy the glider in the process):

(a) A block.
(b) A beehive.
(c) A blinker.
(d) A ship.
(e) An LWSS.

1.3 Use multiple gliders to completely destroy each of the following objects. [Hint: Use one or two gliders to break the object down into simple ash objects like those from Exercise 1.2 and then use additional gliders to clean up those simpler objects.]

(a) A queen bee shuttle.
(b) A copperhead.
(c) Rich's p16.

1.4 There are exactly 6 distinct ways for a glider to collide with a block. List them all and describe the result of each collision.

1.5 Use the syntheses from this chapter to create a 7-glider synthesis of eater 5.

1.6 Use the syntheses from this chapter to create a backrake that creates a period 120 stream of lightweight spaceships.

1.7 Use three Gosper glider guns to create a middleweight spaceship gun.

1.8 Consider the heavyweight spaceship synthesis in Table 1.2.

(a) Why can't you use three Gosper glider guns and this synthesis to create a heavyweight spaceship gun?
(b) One way to overcome this problem is to use *glider pushers* (see Figure 1.32) to repeatedly push one glider stream closer to another. How many glider pushers would you need to use to fix the problem from part (a)?

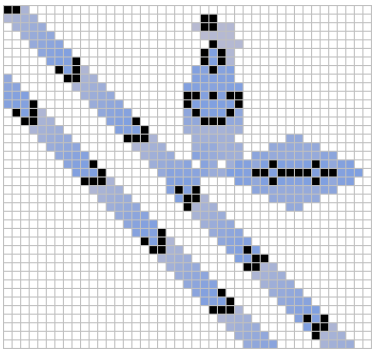


Figure 1.32: A *glider pusher* is a configuration of a queen bee (top center) and a pentadecathlon (middle right) that was found by Dietrich Leithner in December 1993. It pushes a passing glider away by one lane, and thus closes gaps between glider streams with periods that are a multiple of 30.

1.9 Use a two-glider synthesis of a block and a three-glider synthesis of a queen bee to create a 7-glider synthesis of a queen bee shuttle.

1.10 Consider the four-glider synthesis of the twin bees presented in Figure 1.3.

(a) Use this synthesis to synthesize a twin bees shuttle.
(b) Use this synthesis to synthesize the twin bees gun from Figure **??**.

1.11 Create a glider synthesis of the glider pusher from Figure 1.32.

1.12 Construct a glider synthesis of the period 36 oscillator in Figure **??**.

1.13 By using the 2-glider collision in Table 1.1 that changes the direction of a glider, we can decrease the number of directions used in many glider syntheses at the expense of increasing the number of gliders required. Use this technique to create glider syntheses of each of the following patterns, with the property that all gliders come from just two different directions.

(a) A switch engine.
(b) A clock.
(c) A 3-engine Cordership.

1.14 Recall the 2-engine Cordership from Exercise **??**.

(a) Construct a glider synthesis of this Cordership.
(b) Construct a glider synthesis of this Cordership that uses gliders coming only from two different directions. [Hint: Refer back to Exercise 1.13.]

1.15 By using the 3-glider collision in Table 1.2 called a "tee", which produces a glider perpendicular to each of the input gliders, we can modify many glider syntheses so that all input gliders come from two directions that are opposite each other. Use this technique to create glider syntheses of each of the following patterns, with the property that all gliders come from two opposing directions.

(a) A switch engine.
(b) A clock.
(c) Twin bees.

1.16 Construct a glider synthesis for a Gosper glider gun that is stabilized on one end by an eater 1 instead of a block (as in the buckaroo of Figure **??**).

1.17 Complete the incremental glider synthesis of the ecologist presented in Figure 1.8 (i.e., construct it in its entirety from an arrangement of gliders). This synthesis should only use gliders coming from the bottom-left and bottom-right. [Hint: You can use two gliders to create a block and then two more gliders to turn the block into a LWSS.]

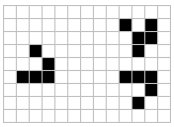
1.18 In Figure 1.9 we showed how to use gliders to turn an ecologist into a forward space rake.

(a) Complete this incremental glider synthesis of the space rake (i.e., construct it in its entirety from an arrangement of gliders).
(b) Show how to synthesize a period 60 space rake. [Hint: Recall the synthesis of the Schick engine from Figure 1.10.]

1.19 In Figure 1.9 we showed how to use gliders to turn an ecologist into a forward space rake. Use similar techniques to turn an ecologist into a backward space rake.

1.20 Create an arrangement of guns that synthesizes the fast forward force field from Figure **??**. Use another gun to fire lightweight spaceships that are teleported through this fast forward force field.

1.21 Use the following 3-glider collision to reduce the 18-glider synthesis of Rich's p16 in Figure 1.16 down to 16 gliders:[23]



1.22 Create a breeder that uses three rakes to synthesize glider-producing switch engines via the 3-glider collision displayed in Table 1.2. [Hint: You will need to use rakes with very high period.]

[23] This 16-glider synthesis was found by Martin Grant.

1.23 Modify the breeder from Figure 1.18 so that the space rakes still move to the east, but the Gosper glider guns shoot gliders to the northwest instead of the northeast.

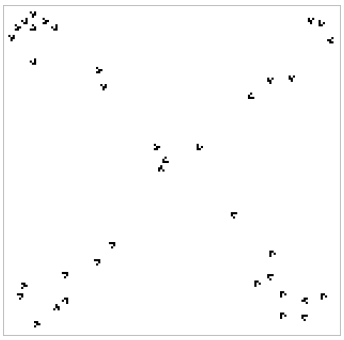
1.24 Construct a breeder that creates Gosper glider guns that are stabilized on at least one side by eater 1s instead of blocks. [Hint: Use the glider synthesis from Exercise 1.16.]

1.25 Create a breeder that uses several rakes to synthesize block-laying switch engines as they move. [Hint: Use a 4-glider synthesis of a switch engine together with the reaction from Figure **??**.]

1.26 Create a breeder that uses several rakes to synthesize twin bees guns as they move, using the glider synthesis that you constructed in Exercise 1.10.

1.27 Use a p1 slow salvo to create the arrangement of 8 blocks in Figure 1.24.

1.28 A 37-glider synthesis of the tubstretcher from Figure **??** is displayed below.



(a) Remove some gliders so as to produce a glider synthesis of the crab.
(b) Add some extra gliders to destroy the tubstretcher (but not the tub itself) after it has been synthesized. Use this method to show that a fixed number of gliders (say 40 or so) can be used to synthesize arbitrarily-large strict still lifes.

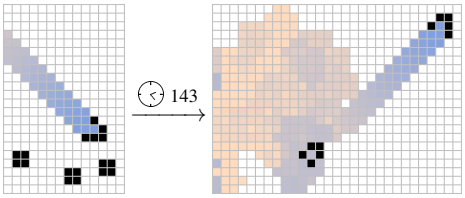
1.29 In this exercise, you will construct some limited-use turners that are different from the blockic one-time turners that we saw in this chapter.

(a) Show that a single boat can be used as a 90-degree one-time turner.
(b) Show that a single eater 1 can be used as a 90-degree one-time turner.
(c) Show that a single long boat can be used as either a 90-degree or 180-degree one-time turner.
(d) Show that the following arrangement of two blinkers can be used as a 90-degree one-time turner.

1.30 In this exercise, we will practice moving gliders around one-time tracks.

(a) Use four boats to move a glider around a square track once, and then leave the track in the same direction that it started in (destroying the track in the process).

(b) The following pattern might be called a "two-time turner", since it can be used to turn two gliders (by firing the second glider at the boat that the first glider produces). Explain why it is *not* possible to use four copies of this pattern to move a single glider around a square track twice (destroying the track in the process).



(c) Use three copies of the two-time turner, together with some additional one-time-turners, to move a glider around a square track twice, and then leave the track in the same direction that it started in (destroying the track in the process).

1.31 One-time turners can be used as tracks for gliders, allowing us to create objects that move at a different speed at the front than at the back (such patterns are called *growing spaceships*).

(a) Use two copies of the blinker puffer from Figure **??** to lay two blinker fuses that a glider bounces back and forth between, using the reaction from Exercise 1.29(d).

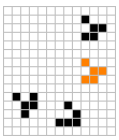
(b) Use rakes of your choosing to synthesize a track of boats that a glider uses as one-time turners and destroys (as in Exercise 1.29(a)).

1.32 Find a way of placing a block near a clock so that a single glider (rather than a pair of gliders, as in Figure 1.27) can trigger the clock inserter reaction.

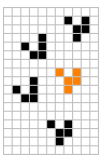
1.33 Use a clock inserter to insert the orange glider into each of these glider salvos, similar to how we inserted the orange glider into the salvo in Figure 1.28. In all cases, the input gliders to the clock inserter must come only from the lower-left and the upper-right.

(a)

(b)



(c)



1.34 Many reactions can be used as inserters other than the clock inserter from Figure 1.27 (however, none are quite as good as the clock inserter itself).

(a) Use the "tee" 3-glider collision from Table 1.2 to insert a glider 15 generations in front of another glider.

(b) Use an eater 1 to insert a glider 15 generations in front of another glider. [Hint: Refer to Exercise 1.29(b).]

(c) Use the clock inserter to insert a glider 14 generations in front of another glider.

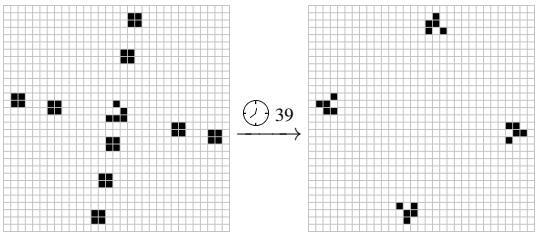
1.35 Create blockic seeds for each of the following objects. [Hint: These can all be created using the same techniques that we used to make the blockic seed in Figure 1.26.]

(a) A lightweight spaceship.

(b) A pulsar.

(c) A switch engine.

1.36 A commonly-used blockic splitter that turns one glider into four gliders is displayed below. This splitter has the advantage of being much faster and cleaner than the one in Figure 1.25, but the disadvantage of requiring 9 blocks instead of just 3. Use this splitter as part of a blockic seed for a clock.



1.37 By chaining together multiple blockic splitters, we can turn one glider into any number of gliders. Create a blockic seed that turns one glider into exactly...

(a) 7 gliders.

(b) 10 gliders.

(c) 9 gliders.

1.38 In the proof of Proposition **??**, we defined a glider's "rank" as its lane number plus its timing. Let w be a real number and suppose that we instead defined a glider's rank as its lane number plus w times its timing.

(a) What parts of the proof of Proposition **??** change if we use $w = 2$? Does the proof still work?

(b) What is the slope of the lines of constant rank in the Life plane when $w = 2$ (recall that the lines of constant rank have slope 3 when $w = 1$)?

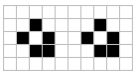
(c) The largest value of w for which the proof of Proposition **??** still works is $w = 7/3$. However, there is one extra technicality in this case—what is it, and how can it be overcome?

(d) What is the smallest value of w for which the proof of Proposition **??** still works? [Hint: There will be a technicality similar to the one from part (b) that has to be overcome if w is minimal.]

(e) What are the possible slopes of lines of constant rank in the Life plane as w ranges from its minimal to maximal values? These are the slopes that we can use to define the "front" of a glider salvo for clock-insertion purposes.

1.39 A *seed* is a constellation (i.e., a collection of simple still lifes and maybe small p2 oscillators) that, when hit by one or more gliders, evolves into a more complicated object. Synthesizing a seed via standard glider collisions and then using that seed to synthesize a more complicated object is a particularly useful technique when working with slow salvo synthesis.

(a) Show how a single glider can be fired at the following pair of boats so as to synthesize (generation 2 of) a switch engine.



(b) Create a glider synthesis that first creates the pair of boats displayed in part (a) and then creates a switch engine from them.

(c) Use your solution to part (a) to construct a seed that, when hit by 2 gliders, synthesizes a 2-engine Cordership.

(d) Create a glider synthesis that first creates your seed from part (c) and then creates a 2-engine Cordership from it.

III Constructions

2. Universal Computation

> Becoming sufficiently familiar with something is a substitute for understanding it.
>
> John H. Conway

At the end of Chapter **??**, we briefly introduced the concept of *computational universality*—the ability of Life (or other data manipulation rules in general) to simulate any computation that a computer can accomplish. It has been known since the early days of Life that it is computationally universal (also sometimes referred to as "Turing complete") [BCG82]. However, the constructions that were used to originally demonstrate this fact were monstrously large and only mostly pieced together. Actually constructing an explicit pattern that works as a universal computer that can be simulated in standard Life software still requires some additional work.

In this chapter, we dive into the details of one particularly flexible computation toolkit[1] that can be used to construct patterns to perform arbitrary computations, and even print the output of those computations in a font made up of blocks. For example, the culmination of this chapter will be a pattern in Section 2.6 that computes and prints the decimal digits of the mathematical constant π.

2.1 A Computer in Life

What exactly does it mean to build a computer in the form of a Conway's Life pattern, anyway? To create a workable computational model in the form of a Life pattern, we need to construct patterns that implement a few different things:

1) We need mechanisms that can store information, preferably in the traditional binary form—zeroes and ones. We *could* perfectly well break with tradition and build, say, a trinary computer based on memory mechanisms with three possible states, or a native decimal computer based on switches with ten different states... but binary two-state switches are much easier to design and to work with.
2) We need to be able to write programs made up simple steps, or "instructions". Each step has a specific effect on data stored in memory. In turn, values stored in memory can affect the flow of

[1] Developed by Adam P. Goucher in 2009 and 2010.

the program. After a conditional instruction, for example, if a given memory location contains "1" instead of "0", a completely different instruction might be the next to be executed.

3) Finally, we also need mechanisms that can represent arbitrarily complex programs. We have to store both the individual instruction steps, as well as the order in which they'll be executed (including the conditional "jump" instructions that depend on the current data stored in memory).

All of this may seem like a big leap in complexity from what we have covered so far, but most of the mechanisms that we need have already appeared in previous chapters. Our job in this chapter is to tie these known mechanisms together in the form of a working Life computer.

2.1.1 A Sliding Block Register

To construct our first usable computational logic component, we observe that we can store a non-negative integer in the position of a block that is moved forward and backward via the block-moving operations that we used to make slide guns (see Section **??**). It is straightforward to use the techniques of Chapter **??** to build a stable conduit that turns a glider into the arrangement of three gliders from Figure **??** that pushes a block diagonally away by 1 cell. We call this an `INC` operation, since it corresponds to increasing the value that the block represents by 1. We can similarly construct a conduit that turns a glider into the arrangement of two gliders from Figure **??** that pulls the block diagonally closer by 1 cell—a "`DEC`" operation that decreases the value of the block by 1.

However, in order for this mechanism to actually be useful in a computer, we need a way of reading its value (i.e., checking where the block is without disturbing it). To this end, we note that it suffices to be able to check whether or not the block is in the "zero" (`Z`) position,[2] since we could repeatedly decrease the value of the block and perform that test after every decrement until we get a positive answer, and then increment the block back to its original position. Once we solve this problem of testing whether or not a block is in the "zero" position, we will have a memory device called a *sliding block register (SBR)*.

The trick that we will use for performing this test is based on the $(2,1)$ block pull reaction that we first saw way back in Figure **??**. Indeed, this reaction is the one that happens when a glider just barely grazes a block, so we can position that glider so that it performs this $(2,1)$ block pull if the sliding block is in the "zero" position, and it simply passes by the block without any interaction at all if it is in any other position. In the latter case, we can simply use the unaffected glider as the "non-zero" (`NZ`) output of our test-if-zero (`TEST`) circuit. However, if the glider was destroyed by the $(2,1)$ block pull reaction (since the sliding block was in the "zero" position), we have some additional work to do: we have to arrange some circuitry so as to create a "zero" (`Z`) output signal and also restore the moved sliding block to the "zero" position.

Restoring the moved sliding block is simple enough—we just send another glider from the opposite direction so as to perform the opposite (2,1) block pull. However, we only want to send that glider if we failed to see an `NZ` output coming out of this `TEST` circuit. We thus need a second signal, split off from the `TEST` input, that is suppressed by a `NZ` output, but otherwise produces a `Z` output as well as a glider that resets the sliding block to the correct position.

This kind of signal-logic thinking takes a while to get used to, but it's less complicated than it sounds. Figure 2.1 displays a schematic that illustrates one way of handling the glider paths needed for this sliding block test-if-zero circuit and its two possible results.

We could then create a complete sliding block register by adding two additional inputs beyond just the `TEST` input:

- a `DEC` input that produces two gliders that are aimed at the sliding block as in Figure **??**, so as to pull the block one cell closer, and
- an `INC` input that produces the three gliders from Figure **??** (two of which are the same as the gliders produced for the `DEC` operation), so as to push the block one cell farther away.

[2] We could equally well call this the "one" position and store positive integers instead of non-negative integers, but it's more traditional to have the block's starting position be zero.

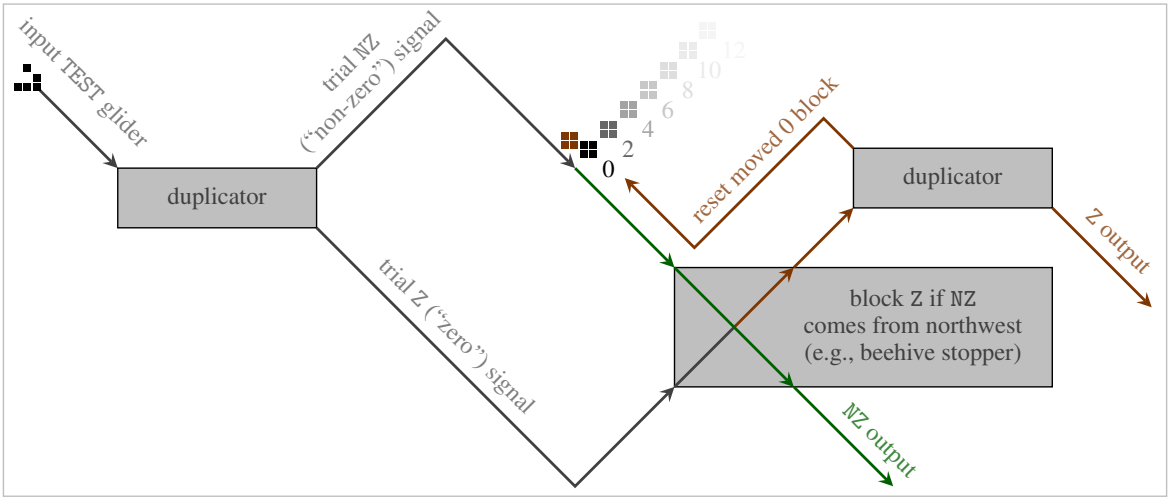


Figure 2.1: A schematic of a TEST circuit that tests whether or not a sliding block is in the "zero" (Z) position. If the block is in a non-zero (NZ) position then the trial NZ glider passes by without affecting it at all. Otherwise, the zero block is shifted via the $(2,1)$ block pull reaction, the trial NZ glider is destroyed, and the Z glider performs another $(2,1)$ block pull to put that zero block back in its original place.

A sliding block register with these three inputs could certainly be used in a Life computer pattern, but we would have to be very careful never to write a program that might accidentally send a DEC signal when the block is already at the "zero" position. After all, if that ever happened then the TEST mechanism would fail catastrophically, because the glider that's supposed to graze the block so as to perform the (2,1) block pull reaction would instead run right into it.[3]

Our program could avoid this danger by always calling TEST first, and then only calling DEC if an NZ output comes back from TEST. However, a better option is to instead bake this idea right into the sliding block register itself, rather than relying on the programs that we write to do so. That is, we adjust the design of the sliding block register so that a TEST always happens just before a DEC. The circuit itself will then ignore the DEC input if the block is already at the Z position.

This slight redesign gives us an equally useful (and much safer!) sliding block register that has only two inputs: INC and TEST-then-DEC, which we abbreviate as TDEC. In this register, which is illustrated in Figure 2.2, there is no longer a way to send in a series of inputs that causes a catastrophic failure. Note that we have made the output Z and NZ lanes of this particular sliding block register transparent so that multiple registers can easily be placed side-by-side.

2.1.2 Assembly Code for a Finite-State Machine

We now have a mechanism that can store integer values, with two simple inputs (INC and TDEC) and two simple outputs (Z and NZ). Our next goal is to figure out how to construct Life "program" patterns that make use of these circuits (sliding block registers) to perform actual calculations. But before delving into the details of how to actually position these registers on the Life plane to carry out computations, we first spend some time thinking about how to perform computations that only involve the operations corresponding to their input lanes—increasing or decreasing the value of some register, and checking whether or not a register is currently equal to zero.

To give a rough idea of how we could carry out a computation of this type, consider the task of just checking which of two registers contains a smaller value. We can solve this problem by repeatedly decrementing the value of the registers, and stopping once either of them is storing a value of 0. Whichever one hits 0 first is determined to be the register that started off storing the smaller value. Pseudocode that implements this algorithm is provided by Pseudocode 2.1.

When we write pseudocode like this, we use labels like U0, U1, U2, ... to denote the various

[3] By contrast, INC operations will never cause problems—it's always possible to move the block out by one more step to store the integer $n+1$ instead of n. So only DEC instructions are potentially dangerous.

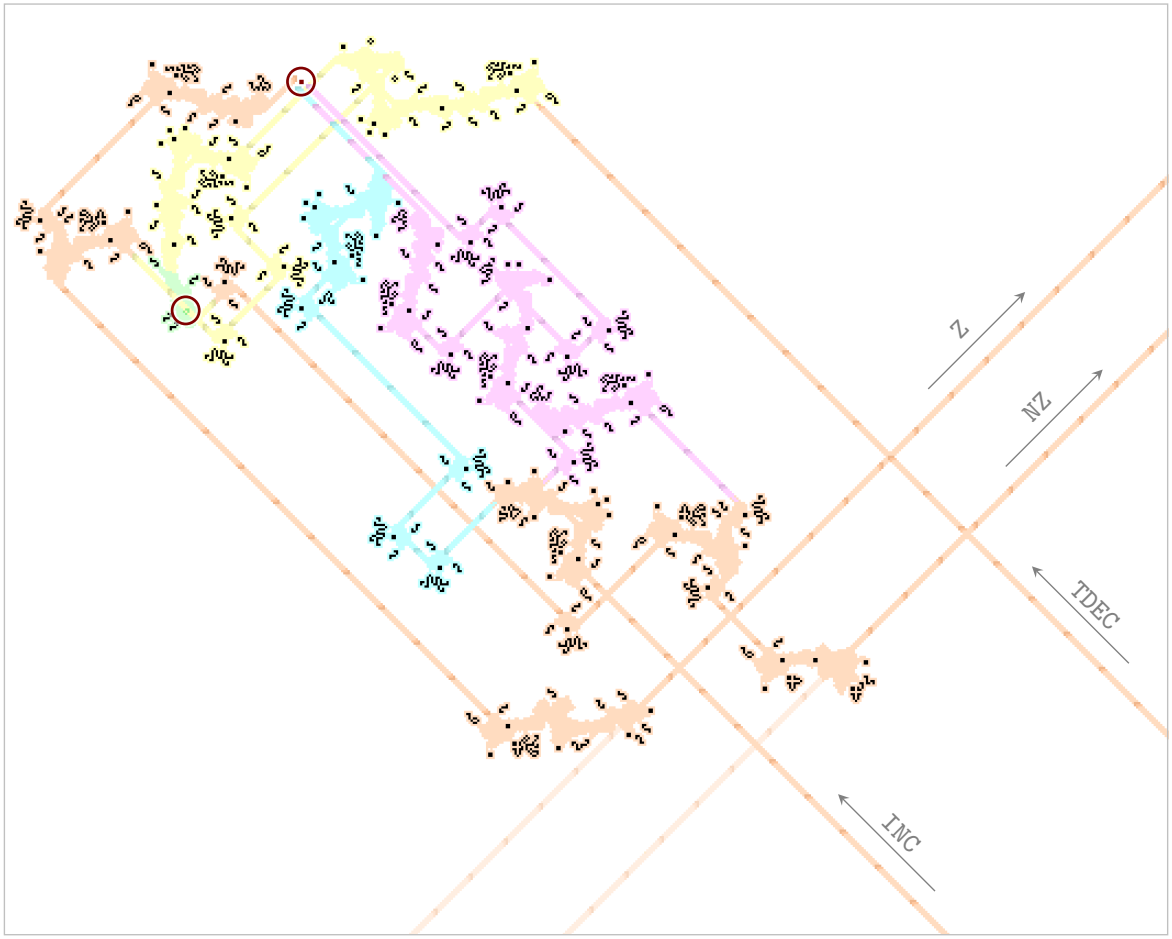


Figure 2.2: A sliding block register. If a glider enters on the INC lane, the aqua and magenta conduits split it into three gliders that push the sliding block (circled in red near the top) northwest by 1 cell. If a glider enters on the TDEC lane, it enters the TEST circuit (the first half of which is highlighted in yellow). That TEST circuit uses a demultiplexer (highlighted in green) to switch the path of the glider—if the block is in the "zero" position then the glider passes through the demultiplexer to the northwest on the Z output path, and otherwise the demultiplexer creates a boat (circled in red near the top-left) that redirects the glider to the northeast on the NZ output path. In the latter case, the NZ output glider is also fed into the magenta conduit that creates two gliders to pull the sliding block 1 cell southeast.

registers used by the computer program.[4] Also, for this pseudocode it is intended that the computer starts by executing the first line of code, then automatically moves to the following line and executes that, and so on—unless it encounters an instruction that tells it to jump to some other line of the program.

Pseudocode 2.1 Test which of the registers U0 or U1 contains a smaller value.

```
1: if U0 = 0, jump to 6
2: if U1 = 0, jump to 8
3: decrement U0
4: decrement U1
5: jump to 1
6: OUTPUT "U0 <= U1"
7: HALT
8: OUTPUT "U1 < U0"
9: HALT
```

[4]The "U" stands for the word *unary*, since these sliding block registers stores integers in unary (i.e., the base-1 numeral system). We will see a different type of register that instead stores integers in binary a bit later, in Section 2.4.

Our next goal is to simplify and standardize our pseudocode so as to make it simpler to implement as a Life pattern. With this in mind, a *finite-state machine* is a model of computation in which the computer can be in one of a finite number of states at any given time, and its state can change based on inputs that it receives. We can implement a finite-state machine with n possible states via a Life pattern that has a single glider traversing a set of n parallel lanes connected by other circuitry. At any given computer clock tick, the glider will be on exactly one of the "state" lanes. It will then run through structures that send gliders to various inputs of various register circuits, and also send a glider to the lane representing the next state in the program.

In order for our pseudocode to more accurately reflect this finite-state machine that our Life program will implement, we require 2 additional things of all pseudocode that we write from this point forward:

1) We include a "jump" instruction at the end of every single line. This corresponds to specifying exactly what state transition happens at each computer clock tick in the finite-state machine that we are implementing.[5]
2) All conditional statements that we include are conditioned on whether the last output value received from a register was `Z` or `NZ`. For this reason, the first line of our pseudocode from now on will always perform some task (like a `TDEC`) that initializes a `Z` or `NZ` value, so that there's always a "return" (i.e., most recent output) value available in the main program loop.

By making the changes outlined in points (1) and (2) above, our original Pseudocode 2.1 is transformed into the Pseudocode 2.2. These pseudocodes implement the same computational task (determining which of the two registers `U0` and `U1` is storing a smaller value), but the new pseudocode will be simpler for us to implement as a Life pattern. In particular, each line in Pseudocode 2.2 now corresponds to a state in the finite-state machine that we will implement. This machine will start in the state of Line 1, and will move from state to state (i.e., line to line) until the algorithm has completed its work. Eventually either one output or the other will be sent out, and the program will reach its `HALT` state.

Pseudocode 2.2 Test which of the registers `U0` or `U1` contains a smaller value—second version.

```
1: TDEC U0; jump to 2
2: if return=Z then jump to 5, otherwise jump to 3
3: TDEC U1; jump to 4
4: if return=Z then jump to 6, otherwise jump to 1
5: OUTPUT "U0 <= U1"; jump to 7
6: OUTPUT "U1 < U0"; jump to 7
7: HALT
```

We now refine this pseudocode even further and standardize it into what we call *APGsembly code*—the programming language that we will use to write and compile programs in Life patterns.[6] In order to make the conditional statements of our pseudocode even easier to implement, we split each state of the finite state machine (i.e., our "computer") into two substates: one corresponding to the most recent return value being `Z`, and one corresponding to the most recent return value being `NZ`. These substates may result in completely different actions being taken and may cause the program to subsequently jump to completely different states (just as in a regular conditional statement).[7]

[5] Our pseudocode, and the programming language that we will develop based on it, is thus one of the few examples of a language where execution does *not* automatically proceed from one line to the next. For a more "real-world" example of such a language, see for example RPC-4000 machine code, as described in The Story of Mel: `catb.org/jargon/html/story-of-mel.html`

[6] The name "APGsembly" is a play on the word "assembly" (low-level code for programming computer instructions) and the initials of its author, Adam P. Goucher. Indeed, APGsembly was used by Goucher to construct the original π calculator in 2010.

[7] In the actual Life implementation of the finite-state machine, this will mean that a glider will appear on one of two different parallel lanes during any given state, depending on whether the last output value received from a register was `Z` or `NZ`.

Once we make the above change, we no longer really need logical instructions in our pseudocode at all: every program can be specified by listing what action should be taken (e.g., `INC` or `TDEC`) when a particular state and substate is encountered, along with which state should then be jumped to next. See APGsembly 2.1 for our first concrete example of APGsembly code—it implements the same "does `U0` contain a small value than `U1`?" test that we saw in Pseudocode 2.2.

APGsembly 2.1 APGsembly code to test which of the registers `U0` or `U1` contains a smaller value. An output value of `0` indicated that `U0 <= U1`, while an output value of `1` indicates that `U1 < U0`.

```
# State    Input    Next state    Actions
# ---------------------------------------
INITIAL;   ZZ;      ID1;          TDEC U0
ID1;       Z;       ID2;          OUTPUT 0, HALT_OUT
ID1;       NZ;      ID2;          TDEC U1
ID2;       Z;       ID1;          OUTPUT 1, HALT_OUT
ID2;       NZ;      ID1;          TDEC U0
```

On its surface, this APGsembly code looks quite different from the pseudocode that we saw earlier, so it is a good idea to trace through the program flow carefully until we are comfortable with the fact that it really does implement the exact same algorithm that we already saw. We now describe how APGsembly code is written, and how it differs from the pseudocode that we saw earlier:

- Line numbers from our pseudocode have been replaced by paired `ID` labels. Each state can have any alphanumeric ID label of our choosing (with the exception of the first, which is always called `INITIAL`),[8] and using IDs like this instead of line numbers makes it much easier to manage our code. Indeed, if we were to continue using line numbers instead of labels then we would have to update every single one of our "jump" instructions if we ever inserted a new line of code.
- There are no longer any explicit "if" statements in the code, since each state (pair of lines sharing an ID label) acts implicitly as an "if" statement. *If* the return value (from the previous action) is `Z`, perform the actions given on the first (`Z`) line; *otherwise* perform the actions given in the second (`NZ`) line.
- Sometimes, a particular state can only ever be reached by a `Z` input, so it does not need a corresponding `NZ` substate. In this case, the `NZ` substate can be omitted from the APGsembly code, and the `Z` substate is instead denoted by `ZZ` so that the compiler knows that the `NZ` omission was intentional. The `INITIAL` state is always assumed to have a `Z` input (just because the computer has to start in *some* substate) and thus the first line of APGsembly code always starts with `INITIAL;ZZ`.[9]
- Each line contains four items, separated by semicolons: state ID, input value, next state ID, and a comma-separated list of actions. The "next state" ID tells the program which state ID to jump to after performing the actions listed on the current line.
- To make code easier to read, we can add comments to our code via lines that start with `#` (so the first two lines of APGsembly 2.1 do not actually affect what the code does, but just help us keep track of the four items on each line of code). For a similar reason, we can include any white space of our choosing between pieces of code, and empty lines between states if desired.
- The `OUTPUT 0` and `OUTPUT 1` actions tell the Life computer to print out either a `0` or a `1` on the Life plane, in a font made up of blocks. We will see how this printing is actually done in Section 2.5, but for now we note that this action can only print the digits `0`–`9` and periods (`.`),

[8] Also, the `INITIAL` state should never be returned to later in a program's execution. It should be the first state, and *only* the first state.

[9] Similarly, if a state performs the exact same actions and jump operation regardless of whether it receives a `Z` or `NZ` return value, we can just list a single line for that state with `*` as its input value, instead of two otherwise identical lines with `Z` and `NZ` as their input values. We will not see an example of this in the main text, but it appears in Appendix A.

not more complicated expressions like `"U0<=U1"`. The `HALT_OUT` action halts the computer[10] (presumably because we are done our desired computation) and sends out a glider (which could be used by another pattern on the Life plane to detect that the computation finished).

There are a few restrictions on what combinations of actions can be performed in a single line of APGsembly code. One of the reasons for this is that all actions on a particular line of APGsembly code are performed simultaneously, rather than sequentially. We should thus avoid single-line lists of actions like "`TDEC U0, INC U0`", for example, since the result of the `TDEC` may depend on whether or not the `INC` has already been completed, which is unpredictable. If the order of actions matters, they should be split into states on multiple different lines.

Furthermore, because of how these actions will be implemented in our Life computer (which we explore in the upcoming Section 2.2), it is not possible to perform the same action more than once in a single substate (i.e., line of APGsembly). For example, if we want to increase the value of the register `U0` by 2, it is tempting to use the list of actions "`INC U0, INC U0`" in a single line of APGsembly. However, this must be avoided—we should instead perform the first `INC U0`, then jump to another state, and then perform the next `INC U0`.

One final restriction comes from the fact that the return value from one line of APGsembly code dictates which substate of the next state is executed, so each line of APGsembly must contain exactly one action that produces a return value. Indeed, if multiple actions produce a return value then it is not clear which substate should be jumped to next, and if no action produces a return value then no substate will be jumped to and the computer will stop running altogether. This is the reason that the `ID1;Z` and `ID2;Z` lines of APGsembly 2.1 contain the `HALT_OUT` lines: the `OUTPUT` action does not produce a return value, so we need to add another action to that line that *does*.[11]

The only actions that produce return values that we have introduced so far are `TDEC` operations (which may return either a `Z` or an `NZ`) and `HALT_OUT` operations (whose return value is irrelevant). Future sections will introduce some additional logic components that include value-returning actions, and a summary of these components and actions can be found by jumping ahead to Table 2.1. But first, let's take a look at what a compiled Life pattern arising from APGsembly code actually looks like.

2.2 A Compiled APGsembly Pattern: Adding Registers

APGsembly code is specifically designed so that it can straightforwardly be compiled into a Life pattern.[12] We now illustrate how this is done by constructing a Life computer that adds the values of two sliding block registers.

Our first step toward building such a computer is to write APGsembly code that implements the desired computational task. The code in APGsembly 2.2 does the job nicely—it works by repeatedly decreasing the value of one register until it reaches a value of `0`, while simultaneously increasing the value of the other register every time. The only pieces of this APGsembly that we have not yet seen are the introductory "`#COMPONENTS`" and "`#REGISTERS`" lines. These lines are used to tell the APGsembly compiler which registers are used in the code,[13] and what the initial values of the registers should be, respectively.[14]

While this code is quite simple (it's only 3 lines long!), it illustrates an important oddity of APGsembly that we must keep in mind. Since `TDEC` is short for "test and *then* decrement", it has the somewhat counterintuitive feature of giving a `Z` or `NZ` return value that is based on the value that was

[10] For this reason, the "next state" provided in lines containing a `HALT_OUT` action does not actually matter, since the computer never actually proceeds past that point.

[11] As a technical note, the `HALT_OUT` action does not actually produce a return value (it does not need to, since we *want* the computer to stop when it encounters `HALT_OUT`). Instead, it is just a special action that bypasses the need for every line to have a return value.

[12] Links to the APGsembly compiler (and an emulator that can be used to help debug APGsembly programs) can be found at `conwaylife.com/wiki/APGsembly`.

[13] "`U0-1`" means that the code uses registers `U0` and `U1`. The "`-`" indicates a range (so "`U0-2`" would refer to registers `U0`, `U1`, and `U2`, for example).

[14] If a register is not initialized via the `#REGISTERS` line, it starts with a value of `0`.

APGsembly 2.2 APGsembly code to add the value of U0 to U1, and zero out U0. The #REGISTERS line pre-loads the registers with the values 7 and 5, respectively (so that after the computation completes, we will have U0 = 0 and U1 = 12).

```
#COMPONENTS U0-1,HALT_OUT
#REGISTERS {"U0":7, "U1":5}
# State    Input    Next state    Actions
# ---------------------------------------
INITIAL;   ZZ;      ID1;          TDEC U0
ID1;       Z;       ID1;          HALT_OUT
ID1;       NZ;      ID1;          TDEC U0, INC U1
```

contained in it *before* it was decremented. In particular, if we use a TDEC to decrement a register from 1 to 0, the return value will be NZ, not Z!

For this reason, APGsembly loops based on the TDEC action often require one more iteration than might be expected at first. If we want to decrement the value of a register from n down to 0, we have to call TDEC n times. However, we then have to call it 1 more time (which does not actually affect the value of the register, since it cannot decrease below 0) in order to get a return value of Z instead of NZ and break out of the loop.

Since APGsembly code requires an action in the INITIAL state anyway before any loops are entered, we often put the extra TDEC command there (we did this in each of APGsembly 2.1 and 2.2). The resulting code has the effect of decreasing the value of the register from n to $n-1$, then looping from $n-1$ to -1, but with the register getting bumped back up to 0 automatically instead of staying at -1.

An actual Life pattern that implements the computation described by APGsembly 2.2 is presented in Figure 2.3. There is a lot going on in this pattern, so we now describe how it is built in some detail. This computer (and Life computers built from APGsembly in general) consists of three main parts: a *computer* (in the southeast), a *component stack* (in the northwest), and a *clock gun* (in the north). We describe these three pieces one at a time.

2.2.1 The Computer

The computer is an implementation of a finite-state machine, so when it is at rest it is always in one of a finite number of different states, represented by a pair of demultiplexers that have a boat. Either a Z or an NZ signal will head southeast from the clock gun as a result of the computation performed in the computer's previous state. That glider will hit either the Z or the NZ demultiplexer and be reflected toward the southwest on a lane corresponding to exactly one specific line of APGsembly code.

If the computer is currently in state ID1, for example, then the two demultiplexers representing the ID1 state both contain boats. If a Z signal comes in, the Z boat turns a glider onto the ID1;Z lane, and the NZ boat is cleaned up by a following glider. Conversely, if an NZ signal comes in, the NZ boat turns a glider onto the ID1;NZ lane, and the Z boat gets cleaned up instead.

The glider heading southwest then passes through one or more *splitters*. These are glider duplicators that create a perpendicular glider while also sending another glider to continue along the exact same lane. As many splitters as we desire can thus be added along these lanes without changing the final destination of the southwest-traveling glider. Each splitter sends a glider on an output lane corresponding to an action from the current line of APGsembly code: TDEC U0, INC U1, and so on.

After going through the sequence of splitters, the southwest-bound glider hits a merge circuit, which is simply a reflector that is transparent to signals passing through it on its output lane. This implements the "jump to" part of each APGsembly line, and the transparency allows multiple lines of APGsembly to jump to the same state. Indeed, multiple merge circuits on the same northwest-to-southeast diagonal all produce gliders on the same lane, which are routed around to trigger the pair of

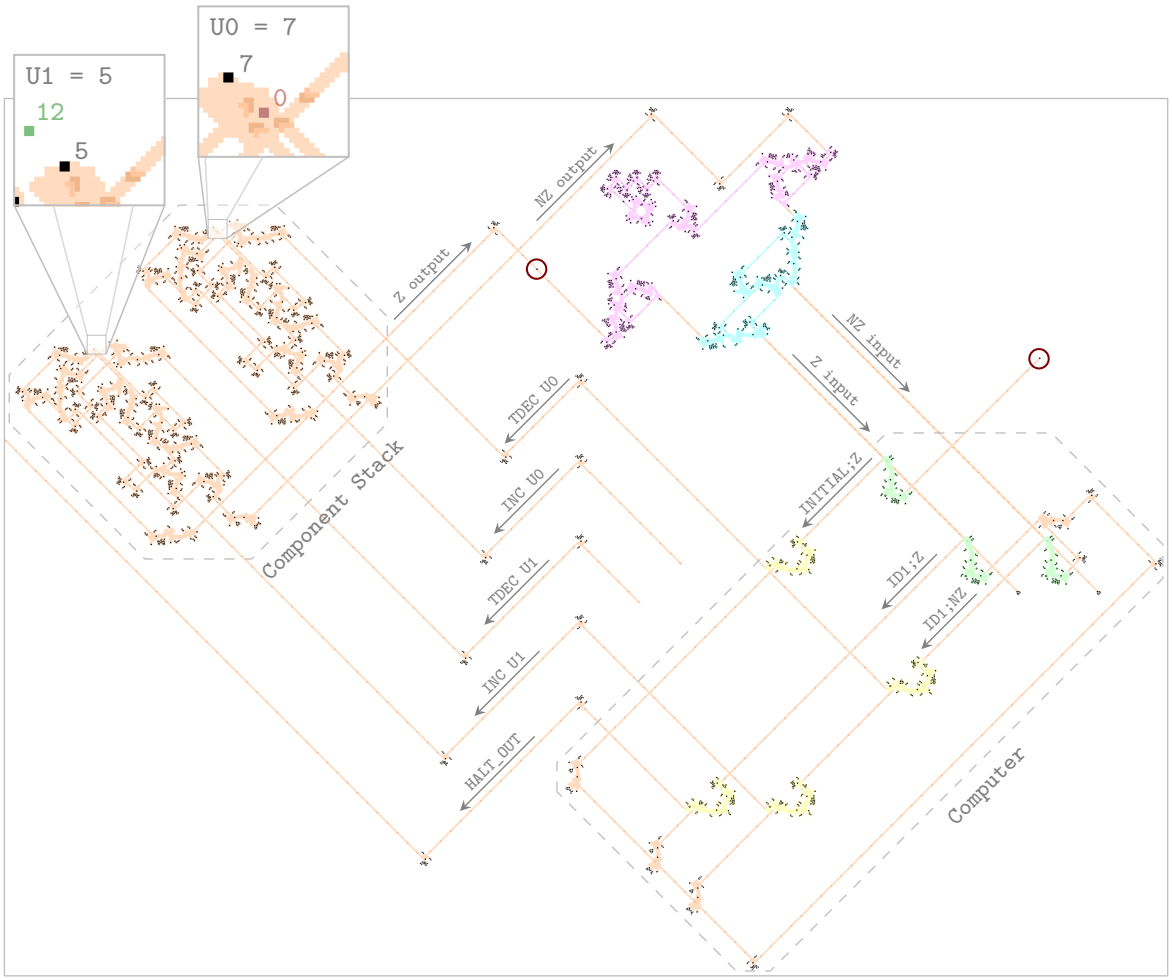


Figure 2.3: A compiled Life pattern that adds the values of the registers `U0` and `U1` (which have been pre-loaded with the values `7` and `5`, respectively), as in APGsembly 2.2. The registers at the west feed their output into the period 2^{20} clock gun (highlighted in magenta) with two separate universal regulators—one for the `Z` path and another for the `NZ` path). The clock gun feeds a glider into a demultiplexer (highlighted in green) corresponding to the next state of the computer (i.e., line of APGsembly). That demultiplexer then feeds into splitters (highlighted in yellow) corresponding to the actions in that line of APGsembly (the `INC U0` and `TDEC U1` lanes are disconnected from the computer because those actions are not called by any line of APGsembly). Finally, the duplicated gliders heading to the northwest feed back into the registers, and the duplicated glider heading to the southwest is reflected around the southeast end of the computer so as to activate the demultiplexer for its next state. The conduit highlighted in aqua puts the `Z` or `NZ` glider coming from the clock gun on the correct input lane, and then produces an extra cleanup glider on each computer input lane so as to reset the demultiplexers. The two gliders circled in red start the computation by activating the `INITIAL;Z` demultiplexer and then feeding a glider into it.

demultiplexers for the next target state.[15] The two boats then wait in the demultiplexer for the next `Z` or `NZ` signal from the clock.

As one technical implementation note, recall that the `INITIAL` state is always assumed to have a `Z` input. That `Z` input is hard-coded into the Life pattern in order to start the computation. The computer displayed in Figure 2.3 does not even have an `INITIAL;NZ` substate, since it would be impossible to reach anyway—recall that this is why the `INITIAL;Z` substate was listed as `INITIAL;ZZ` in APGsembly 2.2.

[15]The computer in Figure 2.3 only has one path around its southeast corner along which gliders can be reflected back into the demultiplexers. This is just because every single line of APGsembly that generated that computer tells it to jump to the same state (`ID1`). We will see other computers shortly that can jump to multiple different states and thus have more glider paths at the southeast.

2.2.2 The Component Stack

In these calculator patterns, information is stored separately from the computer, in an array of "components". These components each have a finite number of inputs (called "actions" in APGsembly code) that can be used to manipulate or retrieve that information, and they potentially have a single Z/NZ output bit. The first example we have seen is the sliding block register, which has two action inputs (`INC Un` and `TDEC Un`) and the two standard outputs (`Z` and `NZ`). Since their output lanes are transparent, any number of sliding block registers `U0`, `U1`, `U2`,... can be stacked side-by-side, and APGsembly code can `INC` or `TDEC` any one of them. If there are a lot of registers, the computer part of the pattern has to be stretched a little wider to accommodate spaces for more splitters. The computer has to be wide enough to hold two splitters for each sliding-block register—one for each `INC` action and one for each `TDEC` action.

Other components may have more or fewer registers. For example, we will see a binary register in Section 2.4 that has four actions: `INC Bn`, `TDEC Bn`, `READ Bn`, and `SET Bn`. New components could be designed to perform any logical function or store any amount of information that we might want; we will see an example of this in Section 2.7.2, with the two-dimensional memory storage "`SQ`" component.

No matter how many components are added to the stack, they're all called in the same way, by a single glider traveling northwest on an "action" lane. Multiple actions can be triggered simultaneously, though each component may be activated at slightly different times depending on where it is placed in the stack. It is perfectly okay for a single line of code in APGsembly to trigger multiple different actions, even if they make use of the same component. It is the responsibility of the programmer to know when this can be done safely, and when this would cause the component's circuitry to fail. Two incompatible actions have to be triggered from separate states (i.e., in separate computer clock ticks).

2.2.3 The Clock Gun

Once a `Z` or `NZ` output signal is generated by the component stack, it is fed back into the computer in order to jump to the next state. However, we have to be slightly careful about when this glider arrives at the computer—we have to be sure that it does not arrive before the next state's demultiplexers are set in the computer (i.e., before the computer's glider has had a chance to go through the merge circuits and find its way all the way around the perimeter of the computer part of the pattern).[16]

To delay the component stack's output glider, we could simply extend the length of the path that it must follow back to the computer. However, we instead make use of a very high-period universal regulator. The advantage of this method is that several pieces of the pattern then repeat predictably at the period of the universal regulator, which lets Life simulation software evolve it much quicker than it could if the glider timing was less regular. In particular, if we choose the universal regulator to align to some high power-of-two period, then Golly's *HashLife* algorithm is able to evolve these patterns extremely quickly.[17]

The universal regulator that we use[18] is conceptually very simple. A stream of gliders coming from a gun of any (sufficiently large) period of our choosing comes in from the northeast and is split into two streams that destroy each other. However, if a glider comes in from the northwest then it is fed into a syringe and then one of the Herschel-to-boat factories from Figure **??**. The resulting boat suppresses a single glider from the gun's duplicated stream, letting a glider that is aligned to the period of that gun escape, as illustrated in Figure 2.4.

The gun that we attach to this universal regulator has period 2^{20}, which we choose simply because it is a power of 2 that is large enough to handle most of the calculators that we will construct in this

[16] This is not much of a concern in the pattern from Figure 2.3 since the path around the computer is so short, but it becomes an issue for patterns with larger computers (i.e., patterns generated by more lines of APGsembly).

[17] HashLife is an algorithm for simulating Life that was developed by Bill Gosper in 1984 [Gos84]. It works by creating a lookup table that keeps track of how the $2^n \times 2^n$ center of certain repetitive $2^{n+1} \times 2^{n+1}$ chunks of a pattern evolve over 2^{n-1} generations. Since nothing outside of that $2^{n+1} \times 2^{n+1}$ square can affect its $2^n \times 2^n$ center within those 2^{n-1} generations, the results of that computation can simply be re-used whenever that same square is encountered in the future.

[18] This universal regulator was constructed by ConwayLife.com forums user "Jormungant" in April 2020.

chapter, but small enough that it can be simulated quickly in software like Golly. To actually construct this gun, we just do exactly what we did in Section **??**; we attach period multipliers to another gun. This particular gun (displayed in Figure 2.5) uses quadri-Snarks (see Exercise **??**) attached to a p256 machine gun, but semi-Snarks could have been used instead at the expense of the gun being slightly larger.

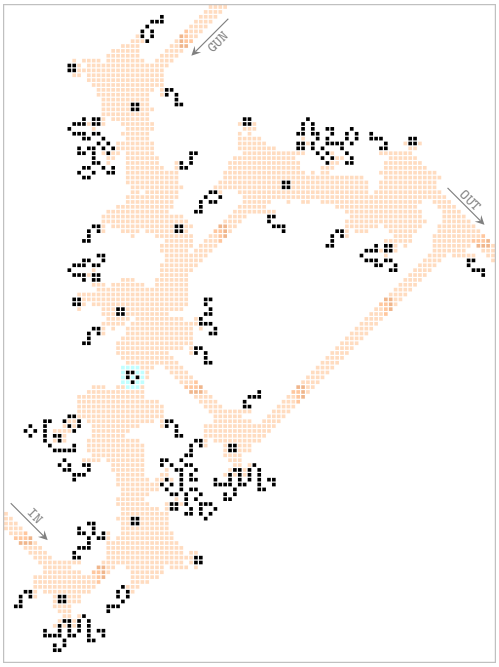


Figure 2.4: A stable universal regulator that works by having the input glider create a boat (highlighted in aqua), which suppresses the glider on the southeast path and allows the northeast glider to escape.

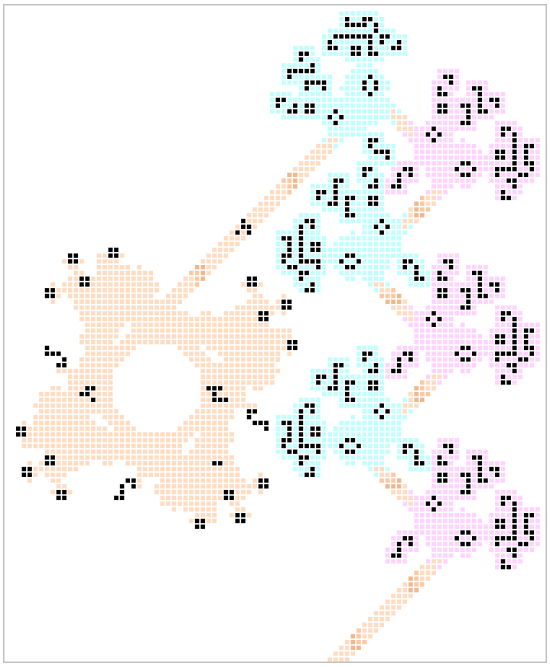


Figure 2.5: A period 2^{20} gun that works by using 6 quadri-Snarks (highlighted in aqua and magenta) to repeatedly quadruple the period of a p256 machine gun.

The high-period universal regulator that results from stitching together Figures 2.4 and 2.5 is called the *clock gun*, and it determines the speed at which the pattern computes whatever it has been programmed to compute. We think of the period of this clock gun as the clock speed of the computer, and one period of the regulator as one clock tick or computational cycle.[19]

In summary, the calculator patterns that we compile from APGsembly work as follows:

- The computer in the southeast corner keeps track of the state of the calculator and feeds into the component stack.
- The component stack keeps track of the calculator's memory and performs actions on that memory. It then feeds into the clock gun.
- The clock gun regulates the glider signal coming from the component stack and feeds it back into the computer.

2.3 Multiplying and Re-Using Registers

Now that we understand how to add the value of two sliding block registers, we ramp up to the problem of multiplying their values. If we wish to multiply `U0` by `U1` and store the result in `U2` then we would like to simply repeatedly add the register `U1` to `U2` a total of `U0` times. However, there is a

[19] Some computations may take longer than one clock tick to complete, but that's okay. For example, if a sliding block register is storing an extremely large value then its sliding block will be extremely far away from the computer, so it will take a long time to `INC` or `TDEC`. In this case, the clock gun halts the computer for one or more clock ticks until an output signal is received from the component stack.

slightly problem with this idea—our method of adding two registers from APGsembly 2.2 zeroes out one of the registers while adding it to the other. Indeed, the only way that we have of looping over one register is based on TDECing it until it hits 0, thus erasing the value that register contained.

To get around this problem, we introduce a temporary register into which we copy the value of one of the registers that we wish to loop over, and then we loop over that temporary register instead. In this particular case of setting U2 = U0 * U1, every time we add U1 to U2, we do so by first copying U1 to a new temporary register U3 (zeroing out U1 in the process), and then looping over U3 so as to add it to each of U1 *and* U2. Pseudocode for carrying out this task, as well as the corresponding APGsembly code, is presented in APGsembly 2.3.[20]

APGsembly 2.3 Pseudocode (left) and APGsembly code (right) to set U2 = U0 * U1 and zero out U0. The register U3 is used just temporarily (it starts and ends at the value of 0) to store the value of U1. After this computation completes, the registers will have the values U0 = 0, U1 = 5, U2 = 35, and U3 = 0.

```
 1: TDEC U0
 2: if return value = Z:
 3:     # Loop U0 times, then halt.
 4:     HALT
 5: else:
 6:     # Add U1 to U2 without erasing U1
 7:     set U3 = U1
 8:     set U1 = U3, U2 = U2 + U3
 9: end if
10: jump to 1
```

```
#COMPONENTS U0-3,HALT_OUT
#REGISTERS {"U0":7, "U1":5}
# State    Input    Next state    Actions
# ---------------------------------------
INITIAL;   ZZ;      ID1;          TDEC U0

# Loop over U0, TDECing it until it hits 0, and then halt.
ID1;       Z;       ID1;          HALT_OUT
ID1;       NZ;      ID2;          TDEC U1

# Copy U1 into U3 while setting U1 = 0.
ID2;       Z;       ID3;          TDEC U3
ID2;       NZ;      ID2;          TDEC U1, INC U3

# Loop over U3, adding its value to U1 (restoring it) and U2.
ID3;       Z;       ID1;          TDEC U0
ID3;       NZ;      ID3;          TDEC U3, INC U1, INC U2
```

A Life pattern that is compiled from APGsembly 2.3 is displayed in Figure 2.6. This multiplication pattern has the same general shape and structure as the addition pattern from Figure 2.3, but with slightly more of everything—4 sliding block registers instead of 2, 7 subtates in the computer (corresponding to the 7 lines of APGsembly code) instead of 3, and so on. Perhaps the most notable change in this calculator pattern is that there are now 3 glider lanes looping around the southeast end of the computer, whereas there was only 1 such lane in Figure 2.3. These extra lanes correspond to the fact that there are lines of APGsembly code in the multiplication program that jump to each of three different states (ID1, ID2, and ID3), whereas every line in the addition program jumped to the same state (ID1).

Just like we can multiply two registers via repeated addition, we can divide two registers via repeated subtraction. In particular, to compute the integer part of U0 / U1 we repeatedly subtract U1 from U0 until U0 = 0. The number of times that we were able to completely subtract U1 is the integer part of U0 / U1, and the value that was contained in U0 when we started our final subtraction is the remainder of that division. Implementing this division-by-subtraction algorithm in APGsembly is very similar to the multiplication-by-addition APGsembly 2.3, so we leave it to Exercise 2.4.

2.4 A Binary Register

One of the unfortunate features of sliding block registers is that the only actions they have available to them are addition and subtraction by 1, so arithmetic operations involving large numbers stored in this registers take a long time to complete. For example, the addition code of APGsembly 2.2 takes U0+1 clock ticks to add the value of U0 to U1, and the multiplication code of APGsembly 2.3 takes roughly 2 * U0 * U1 clock ticks to multiply U0 by U1.[21]

[20] After this code is executed, the value of U1 is preserved, but the value of U0 is not. If we want to preserve the value of U0, we can use *another* temporary register (see Exercise 2.3).

[21] In fact, even the addition and subtraction by 1 operations take longer to complete the larger the value of the register is, since the sliding block takes longer to interact with when it is farther away.

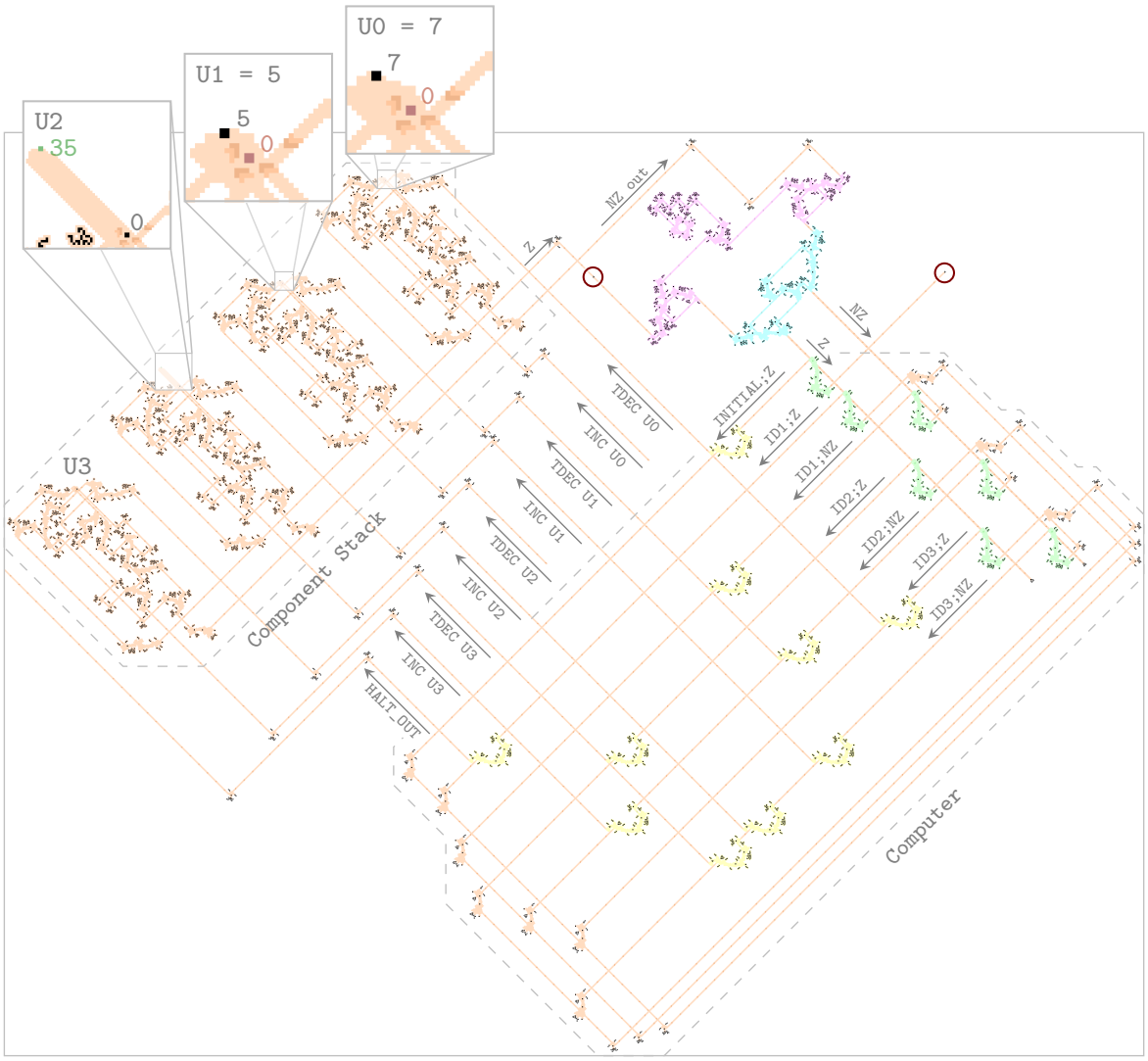


Figure 2.6: A compiled Life pattern that multiplies the values of the registers `U0` and `U1` (which have been pre-loaded with the values `7` and `5`, respectively) and stores the result in `U2`, as in APGsembly 2.3. The color scheme and general structure of this pattern is the same as in Figure 2.3.

While larger inputs leading to longer times to perform arithmetic operations is inevitable, we can save a lot of time by representing non-negative integers in a more efficient way than just as the position of a sliding block. A sliding block register can be thought of as non-negative integers in *unary* (i.e., the base-1 numeral system, in which each number is represented by tally marks or a distance from a fixed point), the register that we now introduce instead encodes non-negative integers in *binary* (i.e., the base-2 numeral system).[22] While performing operations on such a register is somewhat more complicated, it has the advantage of being much quicker because, for example, the non-negative integer n can be stored in $O(\log(n))$ space instead of $O(n)$ space.

We call a register that stores a non-negative integer in this way a *binary register*, and the one that we make use of is displayed in Figure **??**. It uses block-moving shotguns similar to the ones in sliding block registers, except that they move a block 16 cells diagonally at a time instead of just one cell diagonally. This wider spacing leaves room for additional reactions to place objects on one side of the sliding block. Every 16 cells along the sliding block's path is a designated bit location, which can contain either an empty space or a single boat in one of two slightly different positions that correspond to bits `0` and `1`.

[22] Other bases larger than 2 would also be quicker to work with than unary, but they would be even more complicated than binary to implement.

The five actions available to this binary register are a bit more abstract than the actions of the sliding block register, since they work on the individual bits of the register rather than the value that those bits represent. An `INC` action moves the pointer block (called the *read head* of the register) to the next bit location, one step farther away. A `DEC` action moves the pointer block to the previous bit location (or keeps it in the same place if it's already at the zero position). A `READ` action removes the bit from the current pointer location, returning either a `Z` or `NZ` signal as output depending on the bit. A `SET` action places a "`1`" bit at the current pointer location.

Unlike a simple sliding block register where there's no way to damage the mechanism by sending action signals to it, there are definitely ways to program a binary register that will cause it to explode catastrophically. In particular, if you send a `SET` action while there's already a `1` stored at the current pointer location, it will cause irrecoverable damage. There's no built-in safety mechanism to prevent this, but these problems can easily be avoided by always sending a `READ` signal just before any `SET` signal.

2.4.1 A Binary Ruler

To illustrate how to perform some basic arithmetic operations with our binary register, consider the simple problem of increasing the value of the register by `1`. Unlike the sliding block register, there is no single action that performs this operation—we can only perform actions on one bit at a time, but increasing the value of a binary register by `1` may affect several of its bits due to carries.

Fortunately, since we are just increasing the value of the register by `1`, the carry bits are not difficult to take care of: we look for the least significant `0` bit and set it to `1`, and we set all bits that are less significant than it (which are currently equal to `1`) to `0`. Slightly more explicitly, we can increase the value of a binary register by `1` via the following procedure:

1) Start with the read head at the least significant bit and then proceed to step (2) below.
2) Read the value of the current bit (setting it equal to `0` in the process). If it equaled `0`, set it to `1` and then return the read head to the least significant bit. If it equaled `1`, move the read head to the next most significant bit (i.e., increase its position) and then return to step (1).

This method is implemented in APGsembly 2.4, which counts in binary by repeatedly adding `1` to the value of a binary register `B0`. This APGsembly also contains one new action that we have not yet seen: `NOP`. This is an old standard programming abbreviation, short for "No OPeration"; it tells the computer not to change anything right now, but emit a `Z` return value anyway. It is typically used in conjunction with other actions that don't have a return value (`SET B0` and `INC B0` in this case), since exactly one action in every state's list of actions must return a value of either `Z` or `NZ`.[23]

On interesting feature of the pattern that results from compiling this APGsembly code (see Figure **??**) is that it exhibits a new type of slow growth that we have not yet seen. We explored patterns with slowly-growing *population* in Section **??**, but this pattern instead has a slowly-growing *bounding box*. In particular, its *diameter* (i.e., longest bounding box side length) in generation t is $O(\log(t))$, since this is the rate at which new most significant bits are added to the end of the binary register.

This diametric growth rate is much slower than any other unbounded pattern that we have seen so far (all of which have been $O(t)$). We will return to this idea of patterns with slowly-growing bounding boxes, and push it to its ultimate limit, in Section 2.7.2.

2.4.2 Addition, Subtraction, and Multiplication by 10

While addition and subtraction of values that are stored in unary (sliding block) registers is reasonably straightforward (refer back to Section 2.2), these operations are more complicated for binary registers. Even just adding `1` to the value of a binary register makes use of four lines of code (the first four lines

[23] It may be tempting to merge multiple lines of APGsembly 2.4 together so as to get rid of the `NOP` actions. For example, we might want to remove the `LSB1` state altogether and replace the `CHECK1;Z` line with the code `CHECK1; Z; LSB2; SET B0, TDEC B0`. The problem with this is that it is not a good idea to perform two actions on the same register (`B0`) in the same clock tick, as it is not clear which one will be performed first (or even worse, they might interfere with each other and cause the register to self-destruct).

APGsembly 2.4 APGsembly code for a *binary ruler*—a pattern that counts in binary.

```
# State     Input     Next state     Actions
# ---------------------------------------
INITIAL;    ZZ;       CHECK1;        READ B0

## Determine whether the current bit equals 0 or 1.
# If it equals 0, set it to 1 and go to the least significant bit.
# If it equals 1, set it to 0 and go to the next most significant bit.
CHECK1;     Z;        LSB1;          SET B0, NOP
CHECK1;     NZ;       CHECK2;        INC B0, NOP
CHECK2;     ZZ;       CHECK1;        READ B0

# Move B0's read head back to its least significant bit.
LSB1;       ZZ;       LSB2;          TDEC B0
LSB2;       Z;        CHECK1;        READ B0
LSB2;       NZ;       LSB2;          TDEC B0
```

from APGsembly 2.4). The reason for the increase in complexity in this case is that when we add or subtract two binary numbers, we have to keep track of digits that are carried from one bit to the next, possibly many times in a row.

In order to make these operations easier to perform, we introduce additional components that are custom-made for storing these carry digits. For example, the ADD and SUB components displayed in Figures 2.7 and 2.8, respectively, perform bitwise operations to assist in the addition and subtraction of binary registers. The ADD component performs the addition of two bits, outputting the least significant bit of the addition and storing the carry bit in its own internal memory, and the SUB component similarly performs the subtraction of two bits while keeping track of the borrow bit.

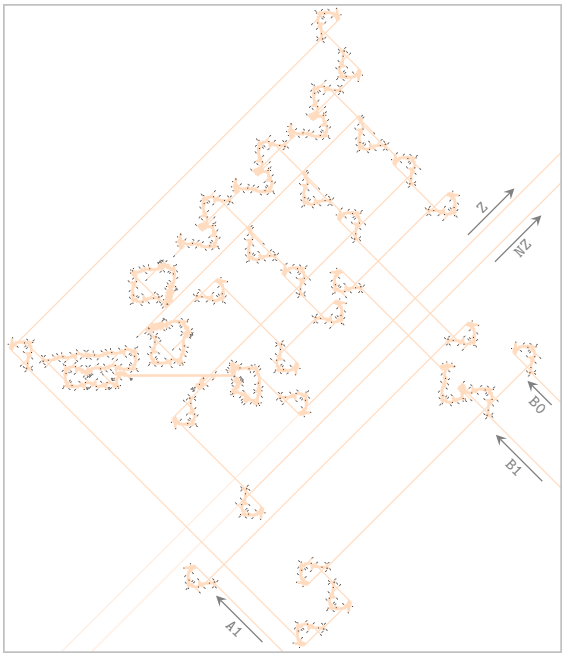


Figure 2.7: The ADD component. To add up two bits x and y while respecting carry bits, input A1 if x = 1 (provide no input if x = 0) and then By.

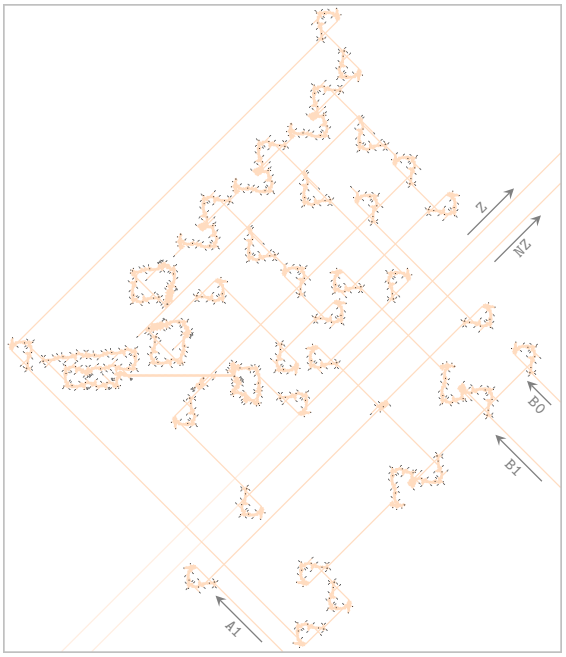


Figure 2.8: The SUB component. To subtract a bit y from x while respecting carry bits, input A1 if x = 1 (provide no input if x = 0) and then By.

More specifically, if we use z to denote the bit that is currently stored in the internal memory of the ADD component, then feeding two bits x and y into it returns a value of (x + y + z) mod 2 and updates its memory to the value (x + y + z)/2 (where we mean integer division here, so that this memory value is either 0 and 1). This ADD component can then be used to add up the values stored in

binary registers by repeatedly calling upon it to add up the least-significant bits of those registers, then their next-least-significant bits, and so on until all bits have been added.

The way that the addition `x + y` is executed via APGsembly is that the pair of commands `ADD Ax` and then `ADD By` must be called (where we replace `x` and `y` by their actual binary values, as in `ADD B1`, for example). The `ADD Ax` command does not return an output (it just adds the input bit to the internal memory) but the `ADD By` command does (so it should always be used second). Furthermore, the `ADD A0` command is not actually implemented, since it does nothing—it does not return a value and also does not alter the internal memory.[24]

For example, to add binary registers containing the numbers $104 = 1101000_2$ and $57 = 111001_2$, we could perform the following sequence of actions in APGsembly (where we perform the "`A`" action in the top row and then the "`B`" action directly below it before moving left-to-right):

```
                                ADD A1                  ADD A1    ADD A1
ADD B1,   ADD B0,   ADD B0,     ADD B1,   ADD B1,       ADD B1,   ADD B0,   ADD B0.
```
[25]

These actions would return the outputs

```
NZ,       Z,        Z,          Z,        Z,          NZ,       Z,        NZ
```

corresponding to the fact that $104 + 57 = 161 = 10100001_2$. Complete APGsembly that implements this addition of two binary registers is provided in APGsembly 2.5.

APGsembly 2.5 APGsembly code for adding (left) or subtracting (right) the binary register `B1` to/from `B0`. In both cases, the new value is stored in `B0` while the value of `B1` is unaffected. The number of bits allocated to the binary registers is stored in `U0`, and the registers `U1` and `U2` are only used temporarily (they start at, and are returned to, a value of `0`).

```
# State    Input    Next state    Actions
# ---------------------------------------
INITIAL;   ZZ;      ADD1;         TDEC U0

# Copy U0 into U2, with the help of U1
ADD1;      Z;       ADD2;         TDEC U1
ADD1;      NZ;      ADD1;         TDEC U0, INC U1
ADD2;      Z;       ADD3;         TDEC U2
ADD2;      NZ;      ADD2;         TDEC U1, INC U0, INC U2

# Loop over U2 to add B1 to B0, one bit at a time.
ADD3;      Z;       ADD7;         TDEC B0
ADD3;      NZ;      ADD4;         READ B0
ADD4;      Z;       ADD5;         READ B1
ADD4;      NZ;      ADD5;         READ B1, ADD A1
ADD5;      Z;       ADD6;         ADD B0
ADD5;      NZ;      ADD6;         ADD B1, SET B1
ADD6;      Z;       ADD3;         TDEC U2, INC B0, INC B1
ADD6;      NZ;      ADD6;         SET B0, NOP

# Move the B0 and B1 read heads back to least significant bit.
ADD7;      Z;       ADD8;         TDEC B1
ADD7;      NZ;      ADD7;         TDEC B0
ADD8;      Z;       ADD8;         HALT_OUT
ADD8;      NZ;      ADD8;         TDEC B1
```

```
# State    Input    Next state    Actions
# ---------------------------------------
INITIAL;   ZZ;      SUB1;         TDEC U0

# Copy U0 into U2, with the help of U1
SUB1;      Z;       SUB2;         TDEC U1
SUB1;      NZ;      SUB1;         TDEC U0, INC U1
SUB2;      Z;       SUB3;         TDEC U2
SUB2;      NZ;      SUB2;         TDEC U1, INC U0, INC U2

# Loop over U2 to subtract B1 from B0, one bit at a time.
SUB3;      Z;       SUB7;         TDEC B0
SUB3;      NZ;      SUB4;         READ B0
SUB4;      Z;       SUB5;         READ B1
SUB4;      NZ;      SUB5;         READ B1, SUB A1
SUB5;      Z;       SUB6;         SUB B0
SUB5;      NZ;      SUB6;         SUB B1, SET B1
SUB6;      Z;       SUB3;         TDEC U2, INC B0, INC B1
SUB6;      NZ;      SUB6;         SET B0, NOP

# Move the B0 and B1 read heads back to least significant bit.
SUB7;      Z;       SUB8;         TDEC B1
SUB7;      NZ;      SUB7;         TDEC B0
SUB8;      Z;       SUB8;         HALT_OUT
SUB8;      NZ;      SUB8;         TDEC B1
```

Since the `ADD` component is somewhat technical and basically only has a single use, not much is lost if we do not actually understand its inner workings and just accept APGsembly 2.5 as a black box that adds up two binary registers. Similarly, we do not go into any great depth in explaining the inner working of the `SUB` component. Instead, we just note that it does for subtraction exactly what the `ADD` component did for addition—it keeps track of the borrow bit (which is analogous to the carry bit for addition) and thus lets us subtract binary registers from each other one bit at a time. The resulting APGsembly code for subtracting one binary register from another one is also provided in APGsembly 2.5.[26]

[24] This contrasts with the command `ADD B0`, which also does not alter the internal memory, but *does* return a value. In particular, `ADD B0` returns the contents of the internal memory and then resets that memory to `0`.

[25] Notice that these `ADD` actions are performed "backwards": they start from the least significant bits of the numbers that we are adding. Also, we need to append an extra `ADD B0` command at the very end to account for the fact that the sum may have one more bit than the summands.

[26] This APGsembly code for subtracting a binary register `B1` from `B0` assumes that `B1 <= B0`. If you are unsure of which binary register is the smaller one, you should first compare them (see Exercise 2.5).

Now that we know how to add and subtract binary components, we can also multiply and divide them via the same tricks that we used for sliding block registers in Section 2.3 and Exercise ??. That is, we simply add or subtract repeatedly. However, doing so can be quite time-consuming when the registers store large values, so we now introduce one additional custom binary register arithmetic component—one that helps us multiply a binary register by 10.[27]

This final new component is called MUL, which stores *four* carry bits so that we can multiply a binary register by 10 one bit at a time, just like the ADD and SUB components made use of a single memory bit to let us add and subtract binary registers one bit at a time. The MUL component is displayed in Figure 2.9, though it is not very enlightening to look at.[28]

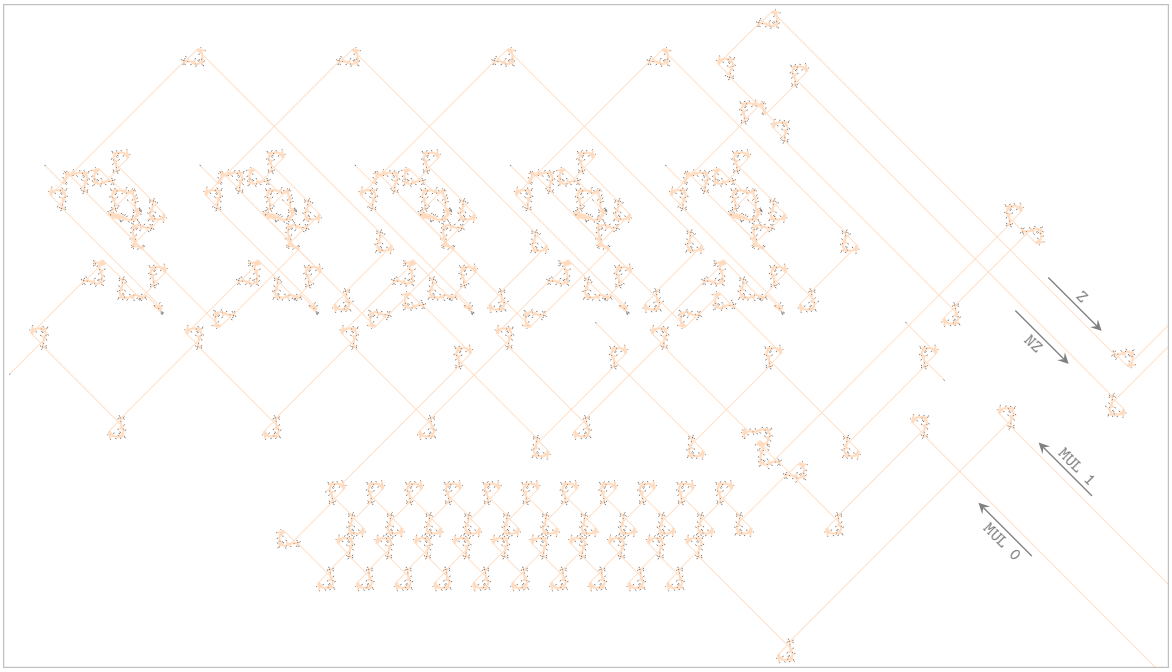


Figure 2.9: The MUL component, which keeps track of multiple carry bits so as to help us easily multiply a binary register by 10.

This component is called in APGsembly via the commands MUL 0 and MUL 1, where the number after the name of the component refers to the current bit of the binary register that we are multiplying by 10. For example, to multiply the number $26 = 11010_2$ by 10, we could perform the following sequence of actions in APGsembly:

MUL 0, MUL 1, MUL 0, MUL 1, MUL 1, MUL 0, MUL 0, MUL 0, MUL 0.[29]

These actions would return the outputs

Z, Z, NZ, Z, Z, Z, Z, Z, NZ,

corresponding to the fact that $26 \times 10 = 260 = 100000100_2$.

We can thus now multiply a binary register by 10 just as quickly and easily as we could add and subtract the values of binary registers. Complete APGsembly code for carrying out this multiplication is provided in APGsembly 2.6.

[27] The reason for us making 10 easier to multiply by (as opposed to any other fixed integer) is that multiplication by 10 is a common operation when working with decimal numbers. In particular, we will make use of this multiplication-by-10 component to help us extract digits in the upcoming π calculator in Section 2.6.

[28] Even moreso than the binary register and ADD and SUB components, the MUL component could be made significantly smaller via modern machinery like Snarks and syringes.

[29] These MUL actions also start from the least significant bit of the number that we are multiplying by 10, just like the ADD and SUB actions. Also, we need to append enough MUL 0 commands in order to retrieve the extra bits that the output will have. Four extra MUL 0 commands always suffices.

APGsembly 2.6 APGsembly code for multiplying the binary register B0 by 10. The number of bits allocated to B0 is stored in U0, and the registers U1 and U2 are only used temporarily (they start at, and are returned to, a value of 0). Compare with APGsembly 2.5 for addition and subtraction.

```
# State    Input    Next state    Actions
# ------------------------------------
INITIAL;   ZZ;      MUL1;         TDEC U0

# Copy U0 into U2, with the help of U1
MUL1;      Z;       MUL2;         TDEC U1
MUL1;      NZ;      MUL1;         TDEC U0, INC U1
MUL2;      Z;       MUL3;         TDEC U2
MUL2;      NZ;      MUL2;         TDEC U1, INC U0, INC U2

# Loop over U2 to multiply B0 by 10, one bit at a time.
MUL3;      Z;       MUL6;         TDEC B0
MUL3;      NZ;      MUL4;         READ B0
MUL4;      Z;       MUL5;         MUL 0
MUL4;      NZ;      MUL5;         MUL 1
MUL5;      Z;       MUL3;         TDEC U2, INC B0
MUL5;      NZ;      MUL5;         SET B0, NOP

# Move the B0 read head back to its least significant bit.
MUL6;      Z;       MUL6;         HALT_OUT
MUL6;      NZ;      MUL6;         TDEC B0
```

2.5 A Decimal Printer

While APGsembly requires us to explicitly specify which digit we want to print (e.g., we cannot print the contents of a sliding block register U0 via the command OUTPUT U0), we can get around this limitation by repeatedly TDECing that register to determine its value, and then printing out that value. This idea is made explicit by APGsembly 2.7. Note that this APGsembly requires that U0 is storing a value between 0 and 9, inclusive, but it could be modified straightforwardly to handle any fixed upper bound just by adding additional TDEC statements. It is even possible to print the contents of a sliding block register regardless of how many decimal digits it has, but this is much more complicated—see Exercise 2.12.

2.6 A Pi Calculator

We now put all of the pieces that we have developed in this chapter together to create one of the most remarkable patterns that has ever been constructed in Life: one that computes and prints the decimal digits of the mathematical constant $\pi = 3.14159\ldots$.[30]

Before we can start writing APGsembly code for our calculator, we have to choose which algorithm we will use to compute π. There are hundreds of known algorithms for this purpose, but because of how APGsembly works, we would like one that has the following two features:

- It makes use entirely of integer arithmetic, rather than floating-point arithmetic (i.e., arithmetic involving numbers that have non-zero digits after the decimal point).[31] This is important because the computational mechanisms that we have developed only work with non-negative integers.

[30] The original π calculator was constructed by Adam P. Goucher in February 2010. The one we construct here uses all of the same ideas and algorithms, but with some slightly newer technology to reduce its size.

[31] We can turn floating-point arithmetic into integer arithmetic by multiplying all numbers involved by large powers of 10, but we then have to be very careful to deal with numerical accuracy concerns, and the integers we would have to use would be monstrously large. It will be better to use an algorithm that is *designed* to only use integer arithmetic.

APGsembly 2.7 APGsembly code for printing the contents of the sliding block register `U0` (and setting `U0 = 0` at the same time).

```
# State     Input     Next state     Actions
# ---------------------------------------
INITIAL;    ZZ;       OUT0;          TDEC U0
OUT0;       Z;        OUT0;          OUTPUT 0, HALT_OUT
OUT0;       NZ;       OUT1;          TDEC U0
OUT1;       Z;        OUT1;          OUTPUT 1, HALT_OUT
OUT1;       NZ;       OUT2;          TDEC U0
OUT2;       Z;        OUT2;          OUTPUT 2, HALT_OUT
OUT2;       NZ;       OUT3;          TDEC U0
OUT3;       Z;        OUT3;          OUTPUT 3, HALT_OUT
OUT3;       NZ;       OUT4;          TDEC U0
OUT4;       Z;        OUT4;          OUTPUT 4, HALT_OUT
OUT4;       NZ;       OUT5;          TDEC U0
OUT5;       Z;        OUT5;          OUTPUT 5, HALT_OUT
OUT5;       NZ;       OUT6;          TDEC U0
OUT6;       Z;        OUT6;          OUTPUT 6, HALT_OUT
OUT6;       NZ;       OUT7;          TDEC U0
OUT7;       Z;        OUT7;          OUTPUT 7, HALT_OUT
OUT7;       NZ;       OUT8;          TDEC U0
OUT8;       Z;        OUT8;          OUTPUT 8, HALT_OUT
OUT8;       NZ;       OUT8;          OUTPUT 9, HALT_OUT
```

- It is "streaming": after producing a particular digit of π, it can carry on to produce its next digit without having to start over from scratch. This is important because we want our calculator to just keep on printing out digits of π forever, without us having to specify how many digits we want ahead of time.

We now describe one algorithm for computing the digits of π (originally developed in [Gib06]) that satisfies all of the criteria outlined above. This algorithm is based on the following series representation for π (see Exercise 2.7 for an explanation of where this series comes from):

$$\pi = 2\left(1+\frac{1}{3}\left(1+\frac{2}{5}\left(1+\frac{3}{7}\left(\cdots\left(1+\frac{k}{2k+1}\left(\cdots\right)\right)\right)\right)\right)\right). \tag{2.1}$$

Indeed, if you truncate the above series after a finite number of additions, you get better and better approximations of π. For example,

$$\begin{aligned} 2 &= 2.0000, & 2\left(1+\frac{1}{3}\right) &\approx 2.6667, \\ 2\left(1+\frac{1}{3}\left(1+\frac{2}{5}\right)\right) &\approx 2.9333, & 2\left(1+\frac{1}{3}\left(1+\frac{2}{5}\left(1+\frac{3}{7}\right)\right)\right) &\approx 3.0476, \end{aligned} \tag{2.2}$$

and so on, with these terms getting closer and closer to $\pi \approx 3.14159\ldots$.

As-written, using these approximations to compute digits of π satisfies neither the streaming property that we wanted—it is not clear how to compute one of the approximations based on previous approximations without starting over from scratch—nor the property that it only uses integer arithmetic. To fix these problems and get an algorithm that it is reasonable to implement in APGsembly, we now introduce a slightly different way of computing these exact same approximations of π.

To this end, define the following 2×2 matrix that depends on a positive integer k:

$$A_0 = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \quad \text{and} \quad A_k = \begin{bmatrix} k & 2k+1 \\ 0 & 2k+1 \end{bmatrix} \quad \text{for all} \quad k \geq 1. \tag{2.3}$$

For example,

$$A_1 = \begin{bmatrix} 1 & 6 \\ 0 & 3 \end{bmatrix}, A_2 = \begin{bmatrix} 2 & 10 \\ 0 & 5 \end{bmatrix}, A_3 = \begin{bmatrix} 3 & 14 \\ 0 & 7 \end{bmatrix}, \quad \text{and} \quad A_4 = \begin{bmatrix} 4 & 18 \\ 0 & 9 \end{bmatrix},$$

and so on. In particular, the entries of each A_k are all non-negative integers.

Next, for each integer $n \geq 1$ define B_n to be the product of the first $n+1$ of the A_ks: $B_n = A_0A_1A_2\cdots A_n$.[32] For example,

$$B_1 = A_0A_1 = \begin{bmatrix} 2 & 6 \\ 0 & 3 \end{bmatrix}, \qquad B_2 = A_0A_1A_2 = \begin{bmatrix} 4 & 40 \\ 0 & 15 \end{bmatrix},$$
$$B_3 = A_0A_1A_2A_3 = \begin{bmatrix} 12 & 308 \\ 0 & 105 \end{bmatrix}, \quad \text{and} \quad B_4 = A_0A_1A_2A_3A_4 = \begin{bmatrix} 48 & 2880 \\ 0 & 945 \end{bmatrix}.$$

Again, each B_n is an integer matrix with a 0 in its bottom-left corner. Importantly, each B_n can be easily obtained from the previous one, since $B_n = B_{n-1}A_n$ for all $n \geq 2$.

If we let q_n and r_n denote the top-right and bottom-right entries of B_n, respectively, then q_n/r_n is exactly the n-th π approximation from Equation (2.2). For example,

$$\frac{q_1}{r_1} = \frac{6}{3} \approx 2.0000, \qquad \frac{q_2}{r_2} = \frac{40}{15} \approx 2.6667,$$
$$\frac{q_3}{r_3} = \frac{308}{105} \approx 2.9333, \quad \text{and} \quad \frac{q_4}{r_4} = \frac{2880}{945} \approx 3.0476.$$

This gives us a way of computing better and better approximations of π using just integer arithmetic (we only introduce decimals at the very final step, which is inevitable since π is not an integer), and which is streaming (since each B_n can be computed straightforwardly from B_{n-1}).

Since each term in the series (2.1) is approximately $1/2$ as large as the term that came before it, the distance between π and our approximation q_n/r_n decreases by roughly a factor of 2 every time we increase n. We thus need to iterate the above algorithm an average of $\log_2(10) \approx 3.3219$ times for each decimal place of accuracy that we would like. For simplicity, we simply round this quantity up to 4 and thus note that if we want n decimals places of π, it suffices to use the approximation based on B_{4n}.[33] For example,

$$B_4 = \begin{bmatrix} 48 & 2880 \\ 0 & 945 \end{bmatrix}, \qquad \frac{q_4}{r_4} = \frac{2880}{945} \approx \mathbf{3}.0476,$$
$$B_8 = \begin{bmatrix} 80640 & 108103680 \\ 0 & 34459425 \end{bmatrix}, \qquad \frac{q_8}{r_8} = \frac{108103680}{34459425} \approx \mathbf{3.1}371,$$
$$B_{12} = \begin{bmatrix} 958003200 & 24835120128000 \\ 0 & 7905853580625 \end{bmatrix}, \qquad \frac{q_{12}}{r_{12}} = \frac{24835120128000}{7905853580625} \approx \mathbf{3.14}14,$$

and so on.

We can now put all of this together into the reasonably APGsembly-friendly Pseudocode 2.3 for computing and printing the decimal digits of π. In this pseudocode, the registers `U0`, `U1`, `U2`, `U3`, `B0`, `B1`, and `B2` store the following quantities:

[32] Here we are using matrix multiplication, which works via the formula $\begin{bmatrix} a & b \\ 0 & c \end{bmatrix} \begin{bmatrix} d & e \\ 0 & f \end{bmatrix} = \begin{bmatrix} ad & ae+bf \\ 0 & cf \end{bmatrix}$.

[33] There is actually an extraordinarily small chance that this algorithm computes an incorrect digit of π at some point. The argument we just gave guarantees that we extract its n-digit from an approximation that is within about 2^{-4n} of the true value of π. Since $2^{-4n} = 16^{-n} < 10^{-n}$, the n-digit of this approximation matches the n-th digit of π as long as π does not have an exceptionally long string of consecutive 0s in its decimal expansion significantly before we would expect to see one (since that could cause a long string of 9s in one approximation that turns into a string of 0s in the next approximation and π itself). However, over 31,000,000,000,000 decimal places of π are known, and our algorithm computes them all correctly.

- `U0`: top-left corner of A_n matrix
- `U1`: top-right and bottom-right corners of A_n matrix
- `U2`: the current digit being computed
- `U3`: the index of the current digit being computed (`U3` $= 0$ for "3", `U3` $= 1$ for "1", `U3` $= 2$ for "4", and so on)
- `B0`: top-left corner of B_n matrix
- `B1`: top-right corner of B_n matrix ($= q_n$)
- `B2`: bottom-right corner of B_n matrix ($= r_n$)

The reason that we use the "`T`" labels for the entries of B_n and "`R`" labels for other quantities is that the entries of B_n grow quite large as n increases, so they should be stored in binary registers. On the other hand, all other quantities used in this algorithm are quite small, so they can simply be stored in sliding block (unary) registers.

Pseudocode 2.3 Pseudocode for computing and printing the decimal digits of π. If any variable is used before it is set, it is assumed to start at a value of 0.

```
 1: # Initialize the A and B matrices.
 2: set U1 = 1, B0 = 2, B2 = 1

 3: # Never stop computing and printing digits.
 4: loop forever:
 5:     # Iterate 4 times before printing a digit.
 6:     loop 4 times:
 7:         # Update the A matrix.
 8:         set U0 = U0 + 1, U1 = U1 + 2

 9:         # Update the B matrix.
10:         set B0 = U0 * B0
11:         set B1 = U1 * (B0 + B1)
12:         set B2 = U1 * B2
13:     end loop

14:     # Compute and print the desired digit of B1 / B2.
15:     set U2 = the U3-th digit after the decimal point of B1 / B2
16:     OUTPUT U2

17:     # Print a decimal point after the 0-th digit (3), and loop.
18:     if U3 = 0 then OUTPUT .
19:     set U3 = U3 + 1
20: end loop
```

We have seen how to implement most of the pieces of Pseudocode 2.3 in APGsembly already. Adding fixed values to sliding block (unary) registers is as straightforward as using their `INC` command the corresponding number of times. Looping over a code section 4 times can be done by setting a register's value to 4 and then using a `TDEC` command to decide whether to restart the loop or exit it. Finally, adding binary registers together and multiplying them by unary registers can be done via the code that we saw in Section 2.4.

However, one piece of Pseudocode 2.3 that is tricky to implement is the digit extraction at line 15. In order to carry out this step using just integer arithmetic, we do the following variant of integer division:

1) Subtract `B2` from `B1` until we cannot do so anymore. The number of subtractions that we performed is the k-th digit of `B1 / B2` after its decimal point, where k is the number of times that we have already performed step (1) ($k = 0$ the first time we perform this step). If this is the digit that we are currently interested in, we can stop. Otherwise, we proceed to step (2) below.

2) Multiply `B1` by 10 and then return to step (1) above.

In order to implement step (1) above in APGsembly, we need to know how to subtract the value of one binary register from another, and how to compare the values of two binary registers (so that we know whether or not we can do the subtraction in the first place). Fortunately, we saw how to do the former of these tasks in APGsembly ??, and the latter task is somewhat simpler and left to Exercise 2.5. Similarly, we saw how to do the multiplication by 10 that step (2) requires us to do back in APGsembly ??.

We have thus seen all of the pieces that we need to finally construct our π-calculating Life pattern. The APGsembly code that implements Pseudocode 2.3 is quite long, so we leave it to Appendix A. The resulting calculator that is compiled from that APGsembly code is presented in Figure 2.10.

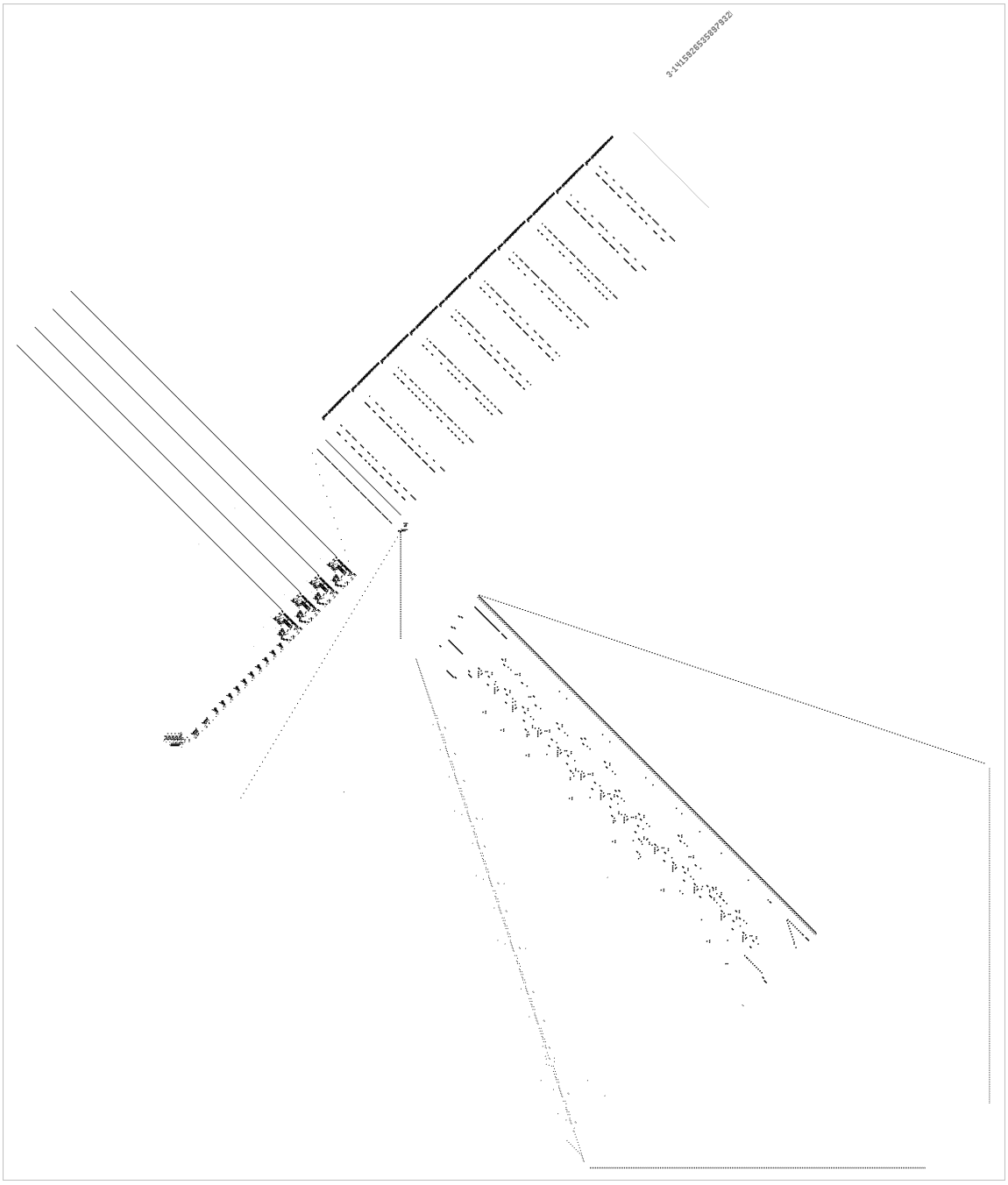


Figure 2.10: After about 168 trillion generations, the π calculator has computed and displayed the first 16 digits of π: 3.1415926535897932. Placeholder for now. Will add annotations and highlighting later.

2.6.1 Computing Other Constants

This same algorithm that we used in our π calculator can also be used to compute many other mathematical constants of interest. In particular, we can use that same method to construct patterns that print the digits of any number that can be represented by a series of the following form, where a, b, p, q, r, and s are non-negative integers:[34]

$$\frac{a}{b}\sum_{k=0}^{\infty}\frac{(p+q)(2p+q)\cdots(kp+q)}{(r+s)(2r+s)\cdots(kr+s)} = \frac{a}{b}\left(1+\frac{p+q}{r+s}\left(1+\frac{2p+q}{2r+s}\left(1+\frac{3p+q}{3r+s}\left(\cdots\left(1+\frac{kp+q}{kr+s}\left(\cdots\right)\right)\right)\right)\right)\right). \tag{2.4}$$

For example, the π series (2.1) arises in the case when $a=2$, $b=1$, $p=1$, $q=0$, $r=2$, and $s=1$.

All that changes in the algorithm and APGsembly code for computing a constant of this more general form, rather than π itself, is that the A_k matrices from Equation (2.3) should instead be defined by

$$A_0 = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix} \quad \text{and} \quad A_k = \begin{bmatrix} kp+q & kr+s \\ 0 & kr+s \end{bmatrix} \quad \text{for all} \quad k \geq 1.$$

The computation of the B_n matrices from the A_ks, as well as the remainder of the algorithm and pseudocode, stays the exact same.

To illustrate how to make this change to construct a calculator for a number other than π, consider the mathematical constant $e = 2.71828\ldots$, which makes frequent appearances in calculus and statistics. It has the well-known series representation

$$e = \sum_{k=0}^{\infty}\frac{1}{k!} = 1+\left(1+\frac{1}{2}\left(1+\frac{1}{3}\left(1+\frac{1}{4}\left(\cdots\right)\right)\right)\right), \tag{2.5}$$

corresponding to the values $a=1$, $b=1$, $p=0$, $q=1$, $r=1$, and $s=0$ in the general series (2.4). We can thus make the following two extremely minor changes to Pseudocode 2.3 for computing π to turn it into an algorithm for computing e:

- Replace line 2 with "`set U0 = 1, B0 = 1, B2 = 1`".
- Replace line 8 with "`set U1 = U1 + 1`".

More generally, to turn Pseudocode 2.3 into pseudocode for computing a number via the series (2.4), we make the following changes based on the values of a, b, p, q, r, and s:

- Replace line 2 with "`set U0 = q, U1 = s, B0 = a, B2 = b`".
- Replace line 8 with "`set U0 = U0 + p, U1 = U1 + r`".

Indeed, modifying the APGsembly code for the π calculator from Appendix A is similarly straightforward. It can be turned into APGsembly for an e calculator by just making minor tweaks to its first 12 (non-comment) lines of code (see Exercise 2.8). Other constants like $\sqrt{2}$ can similarly be computed by making analogous minimal changes (see Exercise 2.9).

2.7 A 2D Printer

While the decimal printer is quite flexible in that we can re-tool it to print any finite set of digits of our choosing, it is still limited by the fact that it can only print linearly. The final component that we introduce, which we denote by `B2D`, solves this problem by being able to print an arbitrary pattern in an infinite 2D quadrant of the Life plane. Alternatively, this component can be thought of as a 2D generalization of the binary register from Section 2.4: while that component kept track of an infinite 1D list of bits, this one keeps track of an infinite 2D array of bits.[35]

[34] If $k=0$ then the term $\frac{(p+q)(2p+q)\cdots(kp+q)}{(r+s)(2r+s)\cdots(kr+s)}$ is an "empty product", which equals 1.

[35] This interpretation explains the abbreviation `B2D` for this component; it stands for "binary 2-dimensional".

The way that this 2D printer works is almost the exact same as the binary register, but with two perpendicular sliding blocks to read and write the desired bits instead of just one. In particular, those sliding blocks are used to fire gliders at each other so as to synthesize or destroy boats that are laid out in a diagonal grid with a separation of 16 full diagonals. These boats can either be interpreted as bits in some sort of computation (with a boat representing a `1` and an empty space representing a `0`), or they can be thought of as pixels in an image that can be viewed by zooming far out in Life simulation software.

In order to actually print those pixels (i.e., boats), the 2D printer can be called via APGsembly actions that are very similar to those of the binary register, but with the minor extra complexity that we can increment and decrement the location of the 2D printer's read head in two different perpendicular directions. We say that the "x" direction of this grid is the diagonal running from southwest to northeast, and the "y" direction is the one running from southeast to northwest (so that the coordinate system for this printer is just the usual Cartesian coordinate system, but rotated counter-clockwise by 45 degrees). Only points with non-negative x and y coordinates are accessible to the printer, so it can print on the top-central quadrant of the Life plane.

To move the sliding blocks of the read head one step farther away from the printer itself (i.e., northeast or northwest), we can use `INC B2DX` and `INC B2DY` actions. Similarly, `TDEC B2DX` and `TDEC B2DY` actions move the pointer block to the previous location in the x or y direction, respectively, or keeps it in the same place if it's already at the zero position.[36] Finally, a `SET` action places a boat (i.e., a pixel) at the current pointer location.[37]

The 2D printer is extremely large, so we do not display it here. However, we will see it in each of the upcoming Figures **??** and 2.12.

2.7.1 Printing a Fractal

To illustrate the flexibility of this 2D printer, we now demonstrate how it can be used to print a fractal. Slightly more accurately, we use it to print better and better approximations of a fractal, which of course is the best we could hope for—our printing capabilities are limited by the fact that we are working on the Life plane and thus must make our images out of pixels.

The way that we construct the fractal is to imagine walking along some path in the plane, printing a pixel after every step that we take. We always step a distance of 1 in one of the eight orthogonal or diagonal directions (in the Moore neighborhood sense, so increasing both the x- and y-coordinates by 1, for example, is a valid move). The path that we walk along is illustrated in Figure 2.11, and is determined as follows:

1) Start at the coordinate $(x,y) = (0,0)$, facing right (toward the coordinate $(1,0)$).
2) Color in the pixel at the current location and then walk forward one step.
3) Let s be the total number of steps that we have taken so far.
   - If s, when represented in base 4, has more "2" digits than $s-1$, turn clockwise by 90 degrees.
   - Otherwise, turn counter-clockwise by 45 degrees.
4) Return to Step (2).

The above procedure results in a path that, on average, moves straight to the right. After all, every time we increase the total number of steps s by one, exactly one digit rolls over to either `1`, `2`, or `3`. Two of those three possibilities result in our orientation rotating 45 degrees counter-clockwise, and the other one of those three possibilities results in our orientation rotating 90 degrees counter-clockwise, for a total average of no rotation at all.

[36] As with the binary register's `TDEC` action, these ones return either a `Z` or `NZ` depending on whether or not the read head was already at the `0` location.

[37] Also just like the binary register's `SET` action, this mechanism will self-destruct if a pixel is `SET` when a boat is already present at the current pointer location.

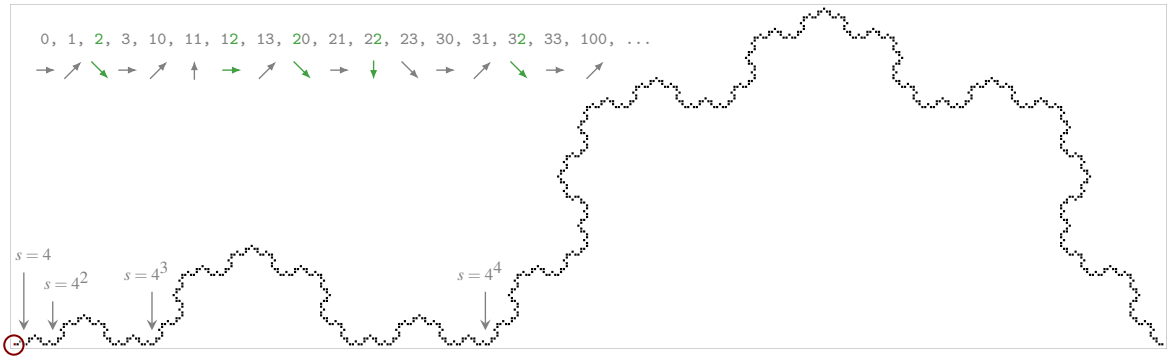


Figure 2.11: The (leftmost portion of the) image that results from starting at the bottom-left corner (circled in red) and printing pixels according to a path-following procedure based on counting in base 4 (in the variable s). These bulbous shapes are better and better approximations of one half of an 8-pointed version of the Koch snowflake fractal.

The fractal-like behavior of this path comes from the repetitive nature of digit strings that are encountered when counting. Indeed, if we ever encounter a particular (base 4) string of digits in s when counting, we will encounter that exact same string of digits infinitely many times later on too, as additional leading digits are added to s. Indeed, this fractal is one half of an 8-pointed version of a well-known fractal called the *Koch snowflake* (which is usually 6-pointed).

APGsembly code that implements this algorithm is displayed in APGsembly 2.8. Although it is somewhat long, it is reasonably straightforward and makes use of 5 sliding block (unary) registers and one binary register to keep track of the following quantities:

APGsembly 2.8 APGsembly code for the 8-pointed Koch snowflake printer.

```
#COMPONENTS U0-5,B0,B2D,NOP
#REGISTERS {"U4":6, "U5":8}
# State    Input    Next state    Actions
# ----------------------------------------
INITIAL;   ZZ;      DIR0;         TDEC U2

# Update U0 and U1, based on U2, so that we walk in the correct
# direction. Also set U3 = U2 and then U2 = 0.
DIR0;      Z;       RESETU2;      TDEC U3, INC U0
DIR0;      NZ;      DIR1;         TDEC U2, INC U3
DIR1;      Z;       RESETU2;      TDEC U3, INC U0, INC U1
DIR1;      NZ;      DIR2;         TDEC U2, INC U3
DIR2;      Z;       RESETU2;      TDEC U3, INC U1
DIR2;      NZ;      DIR3;         TDEC U2, INC U3
DIR3;      Z;       DIRX;         INC U0, INC U1, NOP
DIR3;      NZ;      DIR4;         TDEC U2, INC U3
DIR4;      Z;       DIRX;         INC U0, NOP
DIR4;      NZ;      DIR5;         TDEC U2, INC U3
DIR5;      Z;       DIRXY;        INC U0, INC U1, NOP
DIR5;      NZ;      DIR6;         TDEC U2, INC U3
DIR6;      Z;       DIRY;         INC U1, NOP
DIR6;      NZ;      DIRY;         INC U0, INC U1, INC U3, NOP
DIRX;      ZZ;      RESETU2;      TDEC U3, INC U0
DIRY;      ZZ;      RESETU2;      TDEC U3, INC U1
DIRXY;     ZZ;      RESETU2;      TDEC U3, INC U0, INC U1

# Restore U2 from value in temporary register U3, and then draw
# the pixel (boat) at the current bit.
RESETU2;   Z;       DRAWX1;       TDEC U0, SET B2D
RESETU2;   NZ;      RESETU2;      TDEC U3, INC U2

# Update the B2D read head location based on U0 and U1.
DRAWX1;    Z;       DRAWY1;       TDEC U1
DRAWX1;    NZ;      DRAWX2;       TDEC U0
DRAWX2;    Z;       DRAWY1;       TDEC U1, INC B2DX
DRAWX2;    NZ;      DRAWX3;       TDEC B2DX
DRAWX3;    *;       DRAWY1;       TDEC U1
DRAWY1;    Z;       RESETB0;      TDEC B0
DRAWY1;    NZ;      DRAWY2;       TDEC U1
DRAWY2;    Z;       RESETB0;      TDEC B0, INC B2DY
DRAWY2;    NZ;      DRAWY3;       TDEC B2DY
DRAWY3;    *;       RESETB0;      TDEC B0

# Move B0 read head to least significant bit.
RESETB0;   Z;       INCB0A;       READ B0
RESETB0;   NZ;      RESETB0;      TDEC B0

# Add 1 to B0. If we introduce a new "2" in its base-4
# representation, go to INCU2A, otherwise add 1 to U2.
INCB0A;    Z;       INCB0B;       SET B0, NOP
INCB0A;    NZ;      INCB0E;       INC B0, NOP
INCB0B;    ZZ;      INCB0C;       INC B0, NOP
INCB0C;    ZZ;      INCB0D;       READ B0
INCB0D;    Z;       MOD8A;        TDEC U5, INC U2
INCB0D;    NZ;      MOD8A;        TDEC U5, INC U2, SET B0
INCB0E;    ZZ;      INCB0F;       READ B0
INCB0F;    Z;       INCU2A;       TDEC U4, SET B0
INCB0F;    NZ;      RESETB0;      INC B0, NOP

# Add U4 (= 6) to U2 with the help of U0, without clearing U4.
INCU2A;    Z;       INCU2B;       TDEC U0
INCU2A;    NZ;      INCU2A;       TDEC U4, INC U0, INC U2
INCU2B;    Z;       MOD8A;        TDEC U5
INCU2B;    NZ;      INCU2B;       TDEC U0, INC U4

# Set U2 = U2 mod (U5 = 8), with the help of U0 and U1.
MOD8A;     Z;       RESET3;       TDEC U1
MOD8A;     NZ;      MOD8B;        TDEC U2, INC U0
MOD8B;     Z;       RESET1;       TDEC U0
MOD8B;     NZ;      MOD8A;        TDEC U5, INC U1

# Reset registers and restart.
RESET1;    *;       RESET2;       TDEC U0
RESET2;    Z;       RESET3;       TDEC U1, INC U5
RESET2;    NZ;      RESET2;       TDEC U0, INC U2
RESET3;    Z;       RESET4;       TDEC U0
RESET3;    NZ;      RESET3;       TDEC U1, INC U5
RESET4;    Z;       DIR0;         TDEC U2
RESET4;    NZ;      RESET4;       TDEC U0
```

`U0`: stores the x-direction to move (`0` = none, `1` = right, `2` = left)

`U1`: stores the y-direction to move (`0` = none, `1` = up, `2` = down)

`U2`: a value that determines which of the 8 possible directions to move in at the next step (`0` = right, `1` = up-right, `2` = up, ..., `7` = down-right)—`U0` and `U1` are computed based on this

`U3`: a temporary helper register
`U4`: the amount to add to `U2` after we encounter an extra "2" when counting in base 4 (stays constant at 6, corresponding to a clockwise turn by 90 degrees)
`U5`: how much to mod `U2` by after adding to it (stays constant at 8)

`B0`: keeps track of the number of steps that we have taken in base 4 ($= s$)

The Life pattern[38] that results from compiling this APGsembly is displayed in Figure ??. Since the 2D printer component prints on a diagonal grid, the image that is printed by this pattern is rotated counter-clockwise by 45 degrees from the one in Figure 2.11.

2.7.2 An Extremely Slowly-Growing Pattern

We now return to the problem of creating a pattern with a very slowly-growing bounding box, which we introduced via a binary ruler in Section 2.4.1. In an $n \times n$ bounding box, there are n^2 different cells that can each be in one of 2 states, for a total of 2^{n^2} possible different patterns. It follows that if a pattern exhibits infinite growth then it cannot stay within an $n \times n$ bounding box past generation $t = 2^{n^2}$, since after that point in time its phases would necessarily start repeating.

If we flip this argument around and solve for n in terms of t, we see that in generation t, the bounding box of an infinitely-growing pattern can be no smaller than $\sqrt{\log_2(t)} \times \sqrt{\log_2(t)}$.[39] We now construct a pattern that attains this minimum possible bounding box growth rate, at least asymptotically—its bounding box side lengths in generation t are both $\Theta(\sqrt{\log(t)})$.

The basic idea behind this slowly-growing pattern is the same as it was for the $O(\log(t))$ binary ruler from Section 2.4.1. We count in binary, except instead of placing the bits of the resulting number in a single 1D row via a (`B`) binary register, we arrange them in a 2D triangular array via a (`B2D`) 2D printer. In particular, we place the least significant bit of the number in one row, the next 2 least significant bits in the next row, the next 3 least significant bits in the next row, and so on. The APGsembly code that performs this task is presented in APGsembly 2.9.

This APGsembly code is quite a bit longer, but not much more complicated, than APGsembly 2.4 for the binary ruler. The only extra complexity in this code is that we need some unary registers to help us keep track of how many bits in total we should print in the current row, and how many bits are left before we reach the end of the current row. In particular, it makes use of three sliding block registers as follows:

`U0`: the number of bits left to print before reaching the end of the current row
`U1`: the total number of bits to print in the current row
`U2`: a helper register used to copy `U1` into `U0` when we start a new row

The pattern that results from compiling this APGsembly code is displayed in Figure 2.12. To give an idea of how slowly the diameter of this pattern grows, note in the 7.5×10^{11} generations after printing it first bit (i.e., boat), its computer goes through roughly $44,700$ clock ticks and the pattern's bounding box increases in size by only 64 cells in one direction and 96 cells in the other.[40] We could slow this growth down even more by decreasing the speed of this pattern's clock gun, but this will not change the growth rate's asymptotics.

We have now seen all of the components and actions that we will make use of in this computing toolkit, so this is a natural time to summarize them all. This summary is provided in Table 2.1.

Notes and Historical Remarks

Conway's and Gosper's groups showed in the early 1970s that it was theoretically possible to assemble a Life pattern to complete any computational task that a Turing machine could accomplish [BCG82].

[38] Originally constructed by Michael Simkin in 2019.

[39] This contrasts with our results of Section **??**: even though there is no limit on how slowly the population of a pattern can grow, there is a limit on how slowly its bounding box can grow.

[40] The boats that the 2d printer prints are offset from each other by 16 full diagonals.

APGsembly 2.9 APGsembly code for a pattern whose diameter (i.e., longest bounding box side length) in generation t is $\Theta(\sqrt{\log(t)})$—the smallest unbounded diametric growth rate possible.

```
#COMPONENTS B2D,NOP,U0-2
#REGISTERS {}
# State    Input    Next state    Actions
# ---------------------------------------
INITIAL;   ZZ;      CHECK;        READ B2D

## Determine whether the current bit equals 0 or 1.
# If it equals 0, set it to 1 and go to the least significant bit.
# If it equals 1, set it to 0 and go to the next most significant bit.
CHECK;     Z;       LSB1;         SET B2D, NOP
CHECK;     NZ;      NSB1;         TDEC U0

## The LSB states move the B2D read head to (0,0) and set U0 = U1 = 0.
LSB1;      ZZ;      LSB2;         TDEC B2DX
LSB2;      Z;       LSB3;         TDEC B2DY
LSB2;      NZ;      LSB2;         TDEC B2DX
LSB3;      Z;       LSB4;         TDEC U0
LSB3;      NZ;      LSB3;         TDEC B2DY
LSB4;      Z;       LSB5;         TDEC U1
LSB4;      NZ;      LSB4;         TDEC U0
LSB5;      Z;       CHECK;        READ B2D
LSB5;      NZ;      LSB5;         TDEC U1

## The NSB states move the B2D read head to the next most significant bit.
# If U0 = 0 then we are at the end of the current X row, so start the next one.
# If U0 > 0 then go to the next position in this X row and read the bit.
NSB1;      Z;       NSB3;         TDEC B2DX
NSB1;      NZ;      NSB2;         INC B2DX, NOP
NSB2;      ZZ;      CHECK;        READ B2D

# When going to the next X row, increase U1 (the length of the current X row).
NSB3;      Z;       NSB4;         INC B2DY, INC U1, NOP
NSB3;      NZ;      NSB3;         TDEC B2DX

# Copy U1 into U0, with the help of U2. Then read the current bit.
NSB4;      ZZ;      NSB5;         TDEC U1
NSB5;      Z;       NSB6;         TDEC U2
NSB5;      NZ;      NSB5;         TDEC U1, INC U2
NSB6;      Z;       CHECK;        READ B2D
NSB6;      NZ;      NSB6;         TDEC U2, INC U0, INC U1
```

But it wasn't until almost three decades later that Life technology advanced far enough to allow for universal-computer patterns that were small enough to be simulated on a desktop computer: Paul Rendell's Turing machines in 2000–2011, Paul Chapman's Minsky Register Machines in 2002, Adam P. Goucher's Spartan universal computer-constructor in 2009, and Nicolas Loizeau's 8-bit programmable computer in 2016.

Each of these computational devices used different Life mechanisms to implement logic gates, memory, program storage, and so on.

Exercises

solutions to starred exercises on page ??

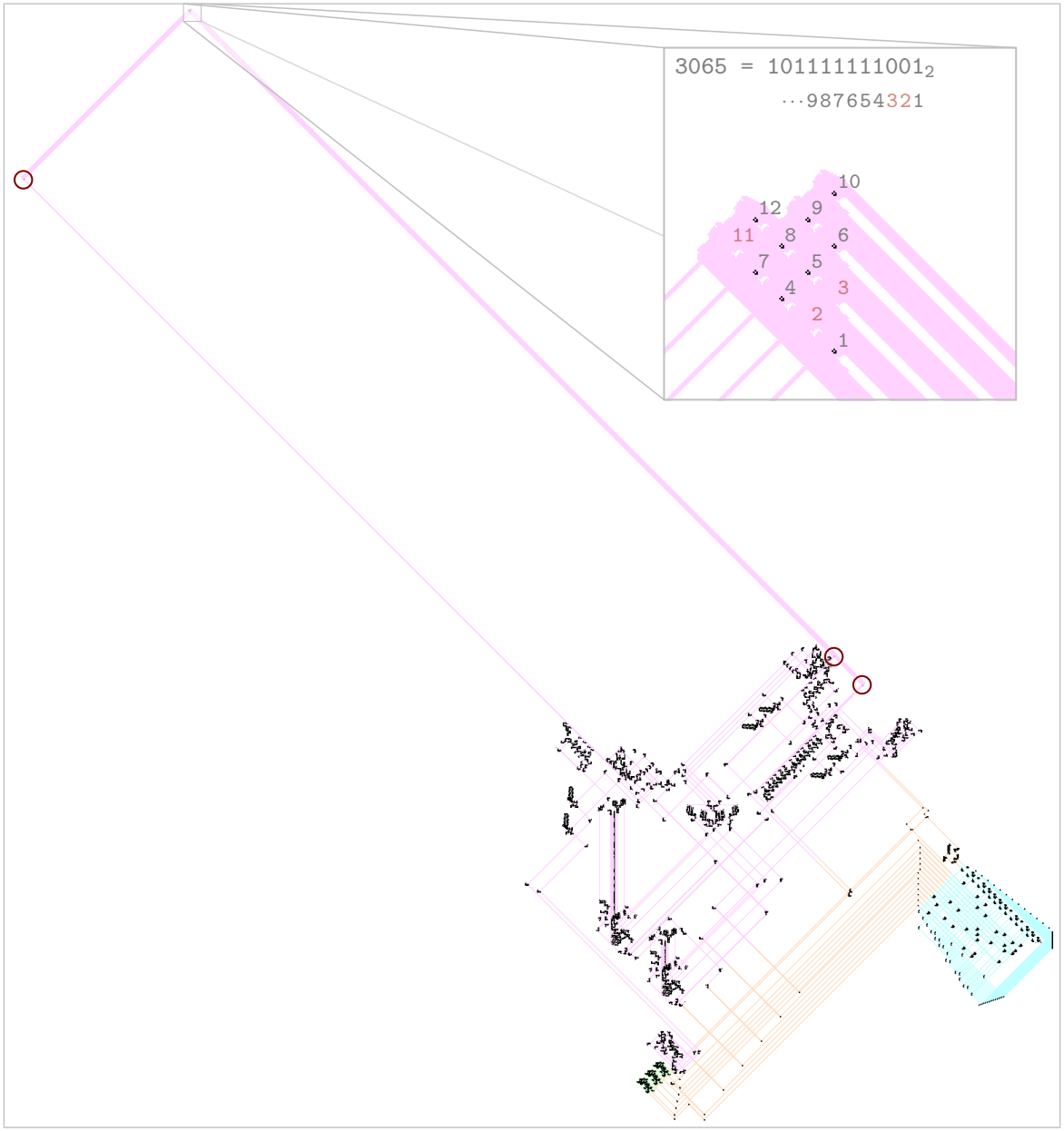


Figure 2.12: A pattern with minimal asymptotic diametric growth rate $\Theta(\sqrt{\log(t)})$, compiled from APGsembly 2.9. The computer at the bottom-right, highlighted in aqua, feeds into three sliding block registers (highlighted in green) and a 2D printer (highlighted in magenta). The 2D printer makes use of three sliding blocks (circled in red) to create and read a 2D array of boats at the top that can in principle extend arbitrarily far to the northwest and northeast. This particular computer counts in binary and instructs the 2D printer to arrange the bits of the current number as boats in a triangle (which gets larger as the number of bits increases). After 7.5×10^{11} generations, it has counted to $3065 = 101111111001_2$.

2.1 Add some additional lines of APGsembly code to APGsembly 2.1 so as to make it have 3 possible outputs instead of just 2: one corresponding to `U0 < U1`, one to `U1 < U0`, and one to `U0 = U1`.

2.2 Modify APGsembly 2.1, with the help of additional registers, so that the `U0` and `U1` registers return to their original values at the end of the program. How many additional registers do you need to accomplish this task?

2.3 Modify APGsembly 2.3 for multiplying two sliding block registers so as to complete the same computation without erasing the value of `U0`.
[Hint: Add another temporary register.]

Component	Actions	Return value and notes
`Un`: sliding block register stores a non-negative integer in unary	`INC Un`	increases the value of the register by `1`, no return value
	`TDEC Un`	returns `Z` if `Un = 0` and `NZ` otherwise, and *then* decreases the value of the register by `1` if `NZ`
`Bn`: binary register stores a non-negative integer in binary	`INC Bn`	increases the position of the read head, no return value
	`TDEC Bn`	returns `Z` if read head is at least significant bit and `NZ` otherwise, and *then* decreases position by `1` if `NZ`
	`READ Bn`	returns the bit (`Z = 0` or `NZ = 1`) at the read head, and then sets it equal to `0`
	`SET Bn`	set the bit at the read head to `1`, no return value, breaks if that bit already equals `1`
`B2D`: 2D binary register stores a 2D array of bits	`INC B2DX`	increases the X position of the read head, no return value
	`INC B2DY`	increases the Y position of the read head, no return value
	`TDEC B2DX`	returns `Z` if X read head is at least significant bit and `NZ` otherwise, and *then* decreases X position by `1` if `NZ`
	`TDEC B2DY`	returns `Z` if Y read head is at least significant bit and `NZ` otherwise, and *then* decreases Y position by `1` if `NZ`
	`READ B2D`	returns the bit (`Z = 0` or `NZ = 1`) at the read head, and then sets it equal to `0`
	`SET B2D`	set the bit at the read head to `1`, no return value, breaks if that bit already equals `1`

`ADD`: binary adder	`ADD A1`
	`ADD B0`
	`ADD B1`
		helps us add one binary register to another one, as in APGsembly 2.5

`SUB`: binary subtractor	`SUB A1`
	`SUB B0`
	`SUB B1`
		helps us subtract one binary register from another one, as in APGsembly 2.5

`MUL`: binary multiplier	`MUL 0` `MUL 1`	helps us quickly multiply a binary register by `10`, as in APGsembly 2.6
`OUTPUT`: digit printer	`OUTPUT x`	prints `x` in a font made up of blocks, no return value, `x` must be one of `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, or `.`
`NOP`: no operation	`NOP`	returns `Z` and does nothing else
`HALT_OUT`	`HALT_OUT`	halts the entire computation and emits a glider

Table 2.1: A summary of the standard components that APGsembly can make use of and the actions that they can perform.

2.4 In this exercise, we implement the division-by-subtraction algorithm that was described at the end of Section 2.3.

(a) Write pseudocode (similar in style to the pseudocode on the left of APGsembly 2.3) that computes and stores the integer part of `U0 / U1` in `U2` and the remainder of that division in `U3`. You may use as many temporary registers as you like.

(b) Write APGsembly code that implements your pseudocode from part (a).

(c) What happens in your code if `U1 = 0` and you try to divide by it? Modify your code, if necessary, so that it halts and provides some sort of error code in this case (e.g., maybe it could set some designated "error register" `U4` to `1`).

(d) Use the APGsembly compiler that is linked at `conwaylife.com/wiki/APGsembly` to compile a Life pattern that implements your APGsembly code from part (b).

2.5 Write APGsembly code that determines which of two binary registers `B0` and `B1` is storing a larger value (i.e., code that outputs different results depending on whether `B0 <= B1` or `B1 < B0`, analogous to APGsembly 2.1 for unary registers).

2.6 Explain why the 2d printer (displayed in Figure 2.12) prints its bits so far away from the computer and other logical circuitry.
[Hint: You can move the sliding block at the top-left corner a bit closer to the logical circuitry without breaking anything, but something goes wrong if you move it, for example, 5,000 cells closer.]

∗**2.7** In this exercise, we explore where the series (2.1) for π comes from.

(a) Explain why $\pi = 4 - 4/3 + 4/5 - 4/7 + 4/9 - \cdots$.
[Hint: We did this already in the "Notes and Historical Remarks" section of Chapter **??**.]

(b) Manipulate the series from part (a) to show that

$$\pi = 2 + \left(\frac{4}{1\cdot 3} - \frac{4}{3\cdot 5} + \frac{4}{5\cdot 7} - \frac{4}{7\cdot 9} + \frac{4}{9\cdot 11} - \cdots \right).$$

[Hint: Write each term $4/(2k+1)$ as $2/(2k+1) + 2/(2k+1)$ and regroup parentheses.]

(c) Use the same method on the parenthesized terms from part (b) to rewrite that series as

$$\pi = 2 + \frac{2}{3} + \left(\frac{8}{1\cdot 3\cdot 5} - \frac{8}{3\cdot 5\cdot 7} + \frac{8}{5\cdot 7\cdot 9} - \cdots \right).$$

(d) Repeat this method over and over again to rewrite this series as

$$\pi = 2 \left(1 + \frac{1!}{3} + \frac{2!}{3\cdot 5} + \frac{3!}{3\cdot 5\cdot 7} + \frac{4!}{3\cdot 5\cdot 7\cdot 9} + \cdots \right).$$

(e) Finally, repeatedly factor the series from part (d) to obtain the series (2.1).
[Side note: This method of converting an alternating series into one with non-negative terms is called the *Euler transform*.]

∗**2.8** Modify the first 12 lines of the APGsembly code from Appendix A to turn the π calculator into an e calculator.
[Hint: We described the changes that need to be made in Section 2.6.1.]

2.9 In this exercise, we use the following series to tweak the π calculator to instead compute

$$\sqrt{2} = \sum_{k=0}^{\infty} \frac{(2k+1)!}{2^{3k+1}(k!)^2} = \frac{1}{2} + \frac{3}{8} + \frac{15}{64} + \frac{35}{256} + \frac{315}{4096} + \cdots.$$

(a) The above series representation of $\sqrt{2}$ can be put in the general form (2.4). What are the values of a, b, p, q, r, and s?

(b) Change the first 12 lines of the APGsembly for the π calculator (from Appendix A) to account for these different values of a, b, p, q, r, and s.

(c) If you compile the APGsembly code that you made in part (b), the resulting pattern will not compute the digits of $\sqrt{2}$ correctly. Why not? What additional change must you make to the introductory lines of that APGsembly code to make it function properly?

2.10 The e calculator from Exercise 2.8 can be sped us so as to compute digits considerably quicker, since the series (2.5) for e is so much simpler and converges so much quicker than the series (2.1) for π.

(a) In the π series (2.1), each term is roughly half as large as the term that came before it, so we need to iterate on average $\log_2(10) \approx 3.3219$ times per decimal place of π. How many times do we need to iterate per decimal place of e?

(b) In the e calculator, you do not actually need the `B0` register that was used in the π calculator. Why not?

(c) In the e calculator, the binary registers `B1` and `B2` do not require as many bits of memory as in the π calculator (i.e., `U6` does not need to be as large). Why? Roughly how many bits of memory are needed after n iterations?

(d) The clock delay of the e calculator can be decreased from 2^{20} generations to 2^{18} generations without introducing any errors. Create a period 2^{18} universal regulator that can replace the e calculator's period 2^{20} universal regulator.[41]

(e) Use the simplifications and speed-ups suggested by parts (a–d) to rewrite the APGsembly code for the e calculator so that the resulting compiled pattern prints `2.718` before generation 2×10^{10} (instead of around generation 10^{12}, like the e calculator from Exercise 2.8).

2.11 Write APGsembly code that determines how many decimal digits the integer stored in a sliding block register has.

2.12 We saw APGsembly code for printing the contents of a sliding block register, as long as that register contains a value no larger than 9, in APGsembly 2.7. Now write APGsembly code that prints the contents of a sliding block register, regardless of how many decimal digits it has.
[Hint: This is much more difficult. Make use of the code from Exercise 2.11]

[41] Even though decreasing the clock delay decreases the number of Life generations needed to perform a calculation, it generally does not decrease the real-world time that it takes Life software like Golly to evolve the pattern to the point of having performed that calculation. The reason for this is that the HashLife algorithm used by Golly is able to extremely quickly skip over "wasted" generations where the clock is just waiting, since so little of the Life plane changes during those generations.

2.13 Write APGsembly code for a Life pattern that prints all positive integers, one after another, separated by dots. That is, its output should begin

```
1.2.3.4.5.6.7.8.9.10.11.12.13.14.15.
```

[Hint: Be careful with integers that have 2 or more digits. Make use of the code from Exercise 2.12.]

2.14 Write APGsembly code for a Life pattern that prints the prime numbers, separated by dots. That is, its output should begin

```
2.3.5.7.11.13.17.19.23.29.31.37.41.
```

[Hint: Modify the code from Exercise 2.13 so that an integer n is not printed if the integer division n/k has a remainder of 0 for some $2 \leq k \leq n-1$.]

2.15 Write APGsembly code for a Life pattern that prints the Fibonacci numbers, separated by dots. That is, its output should begin

```
0.1.1.2.3.5.8.13.21.34.55.89.144.
```

with each integer being the sum of the previous two integers.

Appendices and Supplements

A. Extra APGsembly Code

In Section 2.6, we developed an algorithm and pseudocode for computing the decimal digits of $\pi = 3.14159\ldots$. We now present the full APGsembly code that implements that algorithm and pseudocode, and was used to compile the π calculator that we saw in Figure 2.10. In particular, APGsembly A.1 and A.2 implements that pseudocode via sliding block (unary) registers `U0`, `U1`, …, `U9` and binary registers `B0`, `B1`, `B2`, `B3` that store the following quantities (the A_n and B_n matrices and quantities q_n and r_n are as defined in Section 2.6):

- `U0`: top-left corner of A_n matrix
- `U1`: top-right and bottom-right corners of A_n matrix
- `U2`: the current digit being computed
- `U3`: the index of the current digit being computed (`U3` $= 0$ for "3", `U3` $= 1$ for "1", `U3` $= 2$ for "4", and so on)
- `U4`: the number of times that the iteration has run (`U3` increases by 1 every time `U4` increases by 4), typically denoted here by n
- `U5`: counter that forces the program to iterate 4 times before printing a digit
- `U6`: number of bits of memory allocated for the binary registers (`U6` $= 6$ to start, and add n more bits after we run the iteration for the n-th time)
- `U7`–`U9`: temporary registers used to help perform arithmetic operations and miscellaneous loops

- `B0`: top-left corner of B_n matrix
- `B1`: top-right corner of B_n matrix ($= q_n$)
- `B2`: bottom-right corner of B_n matrix ($= r_n$)
- `B3`: temporary register used to help perform arithmetic operations

Recall that lines listed with a substate (input value) $*$ just do the same listed actions and jump operation regardless of whether they receive a `Z` or `NZ` input value.

APGsembly A.1 Page 1 of APGsembly code for a π calculator that implements Pseudocode 2.3.

```
#COMPONENTS NOP,DIGITPRINTER_SE,B0-3,U0-9,ADD,SUB,MUL
#REGISTERS {"U1":1, "U6":6, "B0":2, "B2":1}
# State    Input    Next state    Actions
# ---------------------------------------
INITIAL;   ZZ;      ITER1;        NOP

# Iterate 4 times per digit.
ITER1;     ZZ;      ITER2;        INC U5, NOP
ITER2;     ZZ;      ITER3;        INC U5, NOP
ITER3;     ZZ;      ITER4;        INC U5, NOP
ITER4;     ZZ;      ITER5;        INC U5, NOP
ITER5;     ZZ;      ITER6;        TDEC U5

# Each iteration, set U0 = U0 + 1, U1 = U1 + 2.
ITER6;     Z;       ITER11;       TDEC U3
ITER6;     NZ;      ITER7;        INC U0, INC U1, NOP
ITER7;     ZZ;      MULA1;        INC U1, NOP

## The MULA states set B3 = B1, B1 = U1 * B1.
# Copy B1 into B3, without erasing B1.
MULA1;     ZZ;      MULA2;        TDEC U6
MULA2;     Z;       MULA3;        TDEC U9
MULA2;     NZ;      MULA2;        TDEC U6, INC U9
MULA3;     Z;       MULA4;        TDEC U7
MULA3;     NZ;      MULA3;        TDEC U9, INC U6, INC U7
MULA4;     Z;       MULA7;        TDEC B1
MULA4;     NZ;      MULA5;        READ B1
MULA5;     Z;       MULA6;        READ B3
MULA5;     NZ;      MULA6;        SET B1, SET B3, NOP
MULA6;     *;       MULA4;        INC B1, INC B3, TDEC U7
MULA7;     Z;       MULA8;        TDEC B3
MULA7;     NZ;      MULA7;        TDEC B1
MULA8;     Z;       MULA9;        TDEC U1
MULA8;     NZ;      MULA8;        TDEC B3

# Copy U1 to temporary register U8.
MULA9;     Z;       MULA10;       TDEC U7
MULA9;     NZ;      MULA9;        TDEC U1, INC U7
MULA10;    Z;       MULA11;       TDEC U8
MULA10;    NZ;      MULA10;       TDEC U7, INC U1, INC U8

# Set B1 = U1 * B3 and U8 = 0.
MULA11;    *;       MULA12;       TDEC U8
MULA12;    Z;       MULB1;        TDEC U6
MULA12;    NZ;      MULA13;       TDEC U6
MULA13;    Z;       MULA14;       TDEC U9
MULA13;    NZ;      MULA13;       TDEC U6, INC U9
MULA14;    Z;       MULA15;       TDEC U7
MULA14;    NZ;      MULA14;       TDEC U9, INC U6, INC U7
MULA15;    Z;       MULA19;       TDEC B3
MULA15;    NZ;      MULA16;       READ B3
MULA16;    Z;       MULA17;       READ B1
MULA16;    NZ;      MULA17;       READ B1, SET B3, ADD A1
MULA17;    Z;       MULA18;       ADD B0
MULA17;    NZ;      MULA18;       ADD B1
MULA18;    Z;       MULA15;       TDEC U7, INC B1, INC B3
MULA18;    NZ;      MULA18;       SET B1, NOP
MULA19;    Z;       MULA20;       TDEC B1
MULA19;    NZ;      MULA19;       TDEC B3
MULA20;    Z;       MULA12;       TDEC U8
MULA20;    NZ;      MULA20;       TDEC B1

## The MULB states set B3 = B0, B0 = U0 * B0.
# Copy B0 into B3, without erasing B0.
MULB1;     Z;       MULB2;        TDEC U9
MULB1;     NZ;      MULB1;        TDEC U6, INC U9
MULB2;     Z;       MULB3;        TDEC U7
MULB2;     NZ;      MULB2;        TDEC U9, INC U6, INC U7
MULB3;     Z;       MULB6;        TDEC B0
MULB3;     NZ;      MULB4;        READ B3
MULB4;     *;       MULB5;        READ B0
MULB5;     Z;       MULB3;        INC B0, INC B3, TDEC U7
MULB5;     NZ;      MULB5;        SET B0, SET B3, NOP
MULB6;     Z;       MULB7;        TDEC B3
MULB6;     NZ;      MULB6;        TDEC B0
MULB7;     Z;       MULB8;        TDEC U0
MULB7;     NZ;      MULB7;        TDEC B3

# Copy U0 to temporary register U8.
MULB8;     Z;       MULB9;        TDEC U7
MULB8;     NZ;      MULB8;        TDEC U0, INC U7
MULB9;     Z;       MULB10;       TDEC U8
MULB9;     NZ;      MULB9;        TDEC U7, INC U0, INC U8

# Set B0 = U0 * B3 and U8 = 0.
MULB10;    *;       MULB11;       TDEC U8
MULB11;    Z;       MULC1;        TDEC U1
MULB11;    NZ;      MULB12;       TDEC U6
MULB12;    Z;       MULB13;       TDEC U9
MULB12;    NZ;      MULB12;       TDEC U6, INC U9
MULB13;    Z;       MULB14;       TDEC U7
MULB13;    NZ;      MULB13;       TDEC U9, INC U6, INC U7
MULB14;    Z;       MULB18;       TDEC B3
MULB14;    NZ;      MULB15;       READ B3
MULB15;    Z;       MULB16;       READ B0
MULB15;    NZ;      MULB16;       READ B0, SET B3, ADD A1
MULB16;    Z;       MULB17;       ADD B0
MULB16;    NZ;      MULB17;       ADD B1
MULB17;    Z;       MULB14;       TDEC U7, INC B0, INC B3
MULB17;    NZ;      MULB17;       SET B0, NOP
MULB18;    Z;       MULB19;       TDEC B0
MULB18;    NZ;      MULB18;       TDEC B3
MULB19;    Z;       MULB11;       TDEC U8
MULB19;    NZ;      MULB19;       TDEC B0

## The MULC states set B1 = B1 + (U1 * B0).
# Copy U1 to temporary register U8.
MULC1;     Z;       MULC2;        TDEC U7
MULC1;     NZ;      MULC1;        TDEC U1, INC U7
MULC2;     Z;       MULC3;        TDEC U8
MULC2;     NZ;      MULC2;        TDEC U7, INC U1, INC U8

# Set B1 = B1 + (U1 * B3) and U8 = 0.
MULC3;     Z;       MULD1;        TDEC U6
MULC3;     NZ;      MULC4;        TDEC U6
MULC4;     Z;       MULC5;        TDEC U9
MULC4;     NZ;      MULC4;        TDEC U6, INC U9
MULC5;     Z;       MULC6;        TDEC U7
MULC5;     NZ;      MULC5;        TDEC U9, INC U6, INC U7
MULC6;     Z;       MULC10;       TDEC B3
MULC6;     NZ;      MULC7;        READ B3
MULC7;     Z;       MULC8;        READ B1
MULC7;     NZ;      MULC8;        READ B1, SET B3, ADD A1
MULC8;     Z;       MULC9;        ADD B0
MULC8;     NZ;      MULC9;        ADD B1
MULC9;     Z;       MULC6;        TDEC U7, INC B1, INC B3
MULC9;     NZ;      MULC9;        SET B1, NOP
MULC10;    Z;       MULC11;       TDEC B1
MULC10;    NZ;      MULC10;       TDEC B3
MULC11;    Z;       MULC3;        TDEC U8
MULC11;    NZ;      MULC11;       TDEC B1

## The MULD states set B2 = U1 * B2.
# Copy B2 into B3, without erasing B2.
MULD1;     Z;       MULD2;        TDEC U9
MULD1;     NZ;      MULD1;        TDEC U6, INC U9
MULD2;     Z;       MULD3;        TDEC U7
MULD2;     NZ;      MULD2;        TDEC U9, INC U6, INC U7
MULD3;     Z;       MULD6;        TDEC B2
MULD3;     NZ;      MULD4;        READ B3
MULD4;     *;       MULD5;        READ B2
MULD5;     Z;       MULD3;        INC B2, INC B3, TDEC U7
MULD5;     NZ;      MULD5;        SET B2, SET B3, NOP
MULD6;     Z;       MULD7;        TDEC B3
MULD6;     NZ;      MULD6;        TDEC B2
MULD7;     Z;       MULD8;        TDEC U1
MULD7;     NZ;      MULD7;        TDEC B3

# Copy U1 to temporary register U8.
MULD8;     Z;       MULD9;        TDEC U7
MULD8;     NZ;      MULD8;        TDEC U1, INC U7
MULD9;     Z;       MULD10;       TDEC U8
MULD9;     NZ;      MULD9;        TDEC U7, INC U1, INC U8

# Set B2 = U1 * B3 and U8 = 0.
MULD10;    *;       MULD11;       TDEC U8
MULD11;    Z;       ITER8;        INC U4, NOP
MULD11;    NZ;      MULD12;       TDEC U6
MULD12;    Z;       MULD13;       TDEC U9
MULD12;    NZ;      MULD12;       TDEC U6, INC U9
MULD13;    Z;       MULD14;       TDEC U7
MULD13;    NZ;      MULD13;       TDEC U9, INC U6, INC U7
MULD14;    Z;       MULD18;       TDEC B3
MULD14;    NZ;      MULD15;       READ B3
MULD15;    Z;       MULD16;       READ B2
MULD15;    NZ;      MULD16;       READ B2, SET B3, ADD A1
MULD16;    Z;       MULD17;       ADD B0
MULD16;    NZ;      MULD17;       ADD B1
MULD17;    Z;       MULD14;       INC B2, INC B3, TDEC U7
MULD17;    NZ;      MULD17;       SET B2, NOP
MULD18;    Z;       MULD19;       TDEC B2
MULD18;    NZ;      MULD18;       TDEC B3
MULD19;    Z;       MULD11;       TDEC U8
MULD19;    NZ;      MULD19;       TDEC B2
```

APGsembly A.2 Page 2 of APGsembly code for a π calculator that implements Pseudocode 2.3.

```
# Increase the amount of memory that we are allocating to the
# binary registers, by adding U4 to U6.
ITER8;     ZZ;      ITER9;       TDEC U4
ITER9;     Z;       ITER10;      TDEC U7
ITER9;     NZ;      ITER9;       TDEC U4, INC U7
ITER10;    Z;       ITER6;       TDEC U5
ITER10;    NZ;      ITER10;      TDEC U7, INC U4, INC U6

## Extract the units digit from (10^U3) * B1 / B2, as that is
## the digit of pi that we want to print.
# Copy U3 to temporary register U8.
ITER11;    Z;       ITER12;      TDEC U7
ITER11;    NZ;      ITER11;      TDEC U3, INC U7
ITER12;    Z;       ITER13;      TDEC U6
ITER12;    NZ;      ITER12;      TDEC U7, INC U3, INC U8

# Copy B1 into B3, without erasing B1.
ITER13;    Z;       ITER14;      TDEC U7
ITER13;    NZ;      ITER13;      TDEC U6, INC U7
ITER14;    Z;       ITER15;      TDEC U9
ITER14;    NZ;      ITER14;      TDEC U7, INC U6, INC U9
ITER15;    Z;       ITER18;      TDEC B3
ITER15;    NZ;      ITER16;      READ B3
ITER16;    *;       ITER17;      READ B1
ITER17;    Z;       ITER15;      INC B1, INC B3, TDEC U9
ITER17;    NZ;      ITER17;      SET B1, SET B3, NOP
ITER18;    Z;       ITER19;      TDEC B1
ITER18;    NZ;      ITER18;      TDEC B3
ITER19;    Z;       CMP1;        TDEC U6
ITER19;    NZ;      ITER19;      TDEC B1

# Now compare B2 with B3 to see which is bigger. This
# determines which of the two upcoming code blocks to go to.
CMP1;      Z;       CMP2;        TDEC U7
CMP1;      NZ;      CMP1;        TDEC U6, INC U7
CMP2;      Z;       CMP3;        TDEC U9
CMP2;      NZ;      CMP2;        TDEC U7, INC U6, INC U9
CMP3;      Z;       CMP4;        READ B3
CMP3;      NZ;      CMP3;        TDEC U9, INC B2, INC B3
CMP4;      Z;       CMP5;        READ B2
CMP4;      NZ;      CMP8;        READ B2, SET B3
CMP5;      Z;       CMP6;        TDEC B2
CMP5;      NZ;      CMP9;        TDEC B3, SET B2
CMP6;      *;       CMP7;        TDEC B3
CMP7;      Z;       CMP12;       TDEC B2
CMP7;      NZ;      CMP4;        READ B3
CMP8;      Z;       CMP11;       TDEC B3
CMP8;      NZ;      CMP6;        TDEC B2, SET B2
CMP9;      Z;       CMP10;       TDEC B2
CMP9;      NZ;      CMP9;        TDEC B3
CMP10;     Z;       DIG1;        TDEC U8
CMP10;     NZ;      CMP10;       TDEC B2
CMP11;     Z;       CMP12;       TDEC B2
CMP11;     NZ;      CMP11;       TDEC B3
CMP12;     Z;       SUB1;        TDEC U6
CMP12;     NZ;      CMP12;       TDEC B2

# If B2 <= B3 then subtract B2 from B3.
# That is, start or carry on with the integer division B3 / B2.
SUB1;      Z;       SUB2;        TDEC U7
SUB1;      NZ;      SUB1;        TDEC U6, INC U7
SUB2;      Z;       SUB3;        TDEC U9
SUB2;      NZ;      SUB2;        TDEC U7, INC U6, INC U9
SUB3;      Z;       SUB7;        TDEC B3
SUB3;      NZ;      SUB4;        READ B3
SUB4;      Z;       SUB5;        READ B2
SUB4;      NZ;      SUB5;        READ B2, SUB A1
SUB5;      Z;       SUB6;        SUB B0
SUB5;      NZ;      SUB6;        SUB B1, SET B2
SUB6;      Z;       SUB3;        INC B2, INC B3, TDEC U9
SUB6;      NZ;      SUB6;        SET B3, NOP
SUB7;      Z;       SUB8;        TDEC B2
SUB7;      NZ;      SUB7;        TDEC B3
SUB8;      Z;       CMP1;        TDEC U6, INC U2
SUB8;      NZ;      SUB8;        TDEC B2

# If B2 > B3 we cannot subtract anymore.
# Multiply B3 by 10 and reset U2, or jump ahead and print
# the digit that we have now computed.
DIG1;      Z;       OUT0;        TDEC U2
DIG1;      NZ;      DIG2;        TDEC U2
DIG2;      Z;       DIG3;        TDEC U6
DIG2;      NZ;      DIG2;        TDEC U2
DIG3;      Z;       DIG4;        TDEC U7
DIG3;      NZ;      DIG3;        TDEC U6, INC U7
DIG4;      Z;       DIG5;        TDEC U9
DIG4;      NZ;      DIG4;        TDEC U7, INC U6, INC U9
DIG5;      Z;       DIG8;        TDEC B3
DIG5;      NZ;      DIG6;        READ B3
DIG6;      Z;       DIG7;        MUL 0
DIG6;      NZ;      DIG7;        MUL 1
DIG7;      Z;       DIG5;        INC B3, TDEC U9
DIG7;      NZ;      DIG7;        SET B3, NOP
DIG8;      Z;       CMP1;        TDEC U6
DIG8;      NZ;      DIG8;        TDEC B3

# Print the current digit, which is stored in U2.
OUT0;      Z;       OUTD1;       NOP, OUTPUT 0
OUT0;      NZ;      OUT1;        TDEC U2
OUT1;      Z;       OUTD1;       NOP, OUTPUT 1
OUT1;      NZ;      OUT2;        TDEC U2
OUT2;      Z;       OUTD1;       NOP, OUTPUT 2
OUT2;      NZ;      OUT3;        TDEC U2
OUT3;      Z;       OUTD1;       NOP, OUTPUT 3
OUT3;      NZ;      OUT4;        TDEC U2
OUT4;      Z;       OUTD1;       NOP, OUTPUT 4
OUT4;      NZ;      OUT5;        TDEC U2
OUT5;      Z;       OUTD1;       NOP, OUTPUT 5
OUT5;      NZ;      OUT6;        TDEC U2
OUT6;      Z;       OUTD1;       NOP, OUTPUT 6
OUT6;      NZ;      OUT7;        TDEC U2
OUT7;      Z;       OUTD1;       NOP, OUTPUT 7
OUT7;      NZ;      OUT8;        TDEC U2
OUT8;      Z;       OUTD1;       NOP, OUTPUT 8
OUT8;      NZ;      OUTD1;       NOP, OUTPUT 9

# Check whether or not we just printed the very first digit (3).
# If so, print a decimal point. Either way, increase U3, which
# counts which decimal place we are currently at, and loop back
# to start the next digit calculation.
OUTD1;     ZZ;      OUTD2;       TDEC U3
OUTD2;     Z;       ITER1;       INC U3, NOP, OUTPUT .
OUTD2;     NZ;      OUTD3;       INC U3, NOP
OUTD3;     ZZ;      ITER1;       INC U3, NOP
```

Bibliography

[BC98] David J. Buckingham and Paul B. Callahan. Tight bounds on periodic cell configurations in Life. *Experimental Mathematics*, 7(3):221–241, 1998.

[BCG82] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways for Your Mathematical Plays, volume 2: Games in Particular*, chapter 25. Academic Press, 1982.

[Bos99] Robert A. Bosch. Integer programming and Conway's game of Life. *SIAM Review*, 41(3):596–604, 1999.

[BT04] Robert Bosch and Michael Trick. Constraint programming and hybrid formulations for three life designs. *Annals of Operations Research*, 130:41–56, 2004.

[Coo03] Matthew Cook. *Still life theory*, pages 93–118. Santa Fe Institute Studies in the Sciences of Complexity. Oxford University Press, 2003.

[CS12] Geoffrey Chu and Peter J. Stuckey. A complete solution to the Maximum Density Still Life Problem. *Artificial Intelligence*, 184:1–16, 2012.

[CSdlB09] Geoffrey Chu, Peter J. Stuckey, and Maria Garcia de la Banda. Using relaxations in maximum density still life. In *Proceedings of the Fifteenth International Conference on Principles and Practice of Constraint Programming*, pages 258–273, 2009.

[Elk98] Noam D. Elkies. The still-life density problem and its generalizations. In *Voronoï's Impact on Modern Science, Book I*, volume 21 of *Proceedings of the Institute of Mathematics of the National Academy of Sciences of Ukraine*, pages 228–253. Institute of Mathematics, 1998.

[Epp02] David Eppstein. Searching for spaceships. In *More Games of No Chance*, volume 42 of *MSRI Publications*, pages 433–453. Cambridge University Press, 2002.

[Fla94] Achim Flammenkamp. Natural grown oscillators out of random soup. `http://wwwhomes.uni-bielefeld.de/achim/oscill.html`, 1994. [Online; accessed May 6, 2016].

[Fla04] Achim Flammenkamp. Frequency of Game of Life objects in settled random areas. http://wwwhomes.uni-bielefeld.de/achim/freq_top_life.html, 2004. [Online; accessed May 6, 2016].

[Gar70] Martin Gardner. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123, 1970.

[Gar71] Martin Gardner. On cellular automata, self-reproduction, the Garden of Eden and the game of "life". *Scientific American*, 224:112–117, 1971.

[Gib06] J. Gibbons. Unbounded spigot algorithms for the digits of pi. *The American Mathematical Monthly*, 113(4):318–328, 2006.

[Gos84] R. Wm. Gosper. Exploiting regularities in large cellular spaces. *Physica*, 10D:75–80, 1984.

[HD74] Jean Hardouin-Duparc. Paradis terrestre dans l'automate cellulaire de Conway. *Revue française d'automatique, informatique, recherche opérationnelle*, 8:63–71, 1974.

[LMN05] Javier Larrosa, Enric Morancho, and David Niso. On the practical use of variable elimination in constraint optimization problems: 'still-life' as a case study. *Journal of Artificial Intelligence Research*, 23:421–440, 2005.

[Moo62] Edward F. Moore. Machine models of self-reproduction. *Proceedings of Symposia in Applied Mathematics*, 14:17–33, 1962.

[Myh63] John Myhill. The converse of Moore's Garden-of-Eden theorem. *Proceedings of the American Mathematical Society*, 14:685–686, 1963.

[Nie03] Mark D. Niemiec. Synthesis of complex Life objects from gliders. In *New Constructions in Cellular Automata*, pages 55–78. Oxford University Press, London, 2003.

[Nie10] Mark D. Niemiec. Object synthesis in Conway's Game of Life and other cellular automata. In *Game of Life Cellular Automata*, chapter 8, pages 115–134. Springer London, 2010.

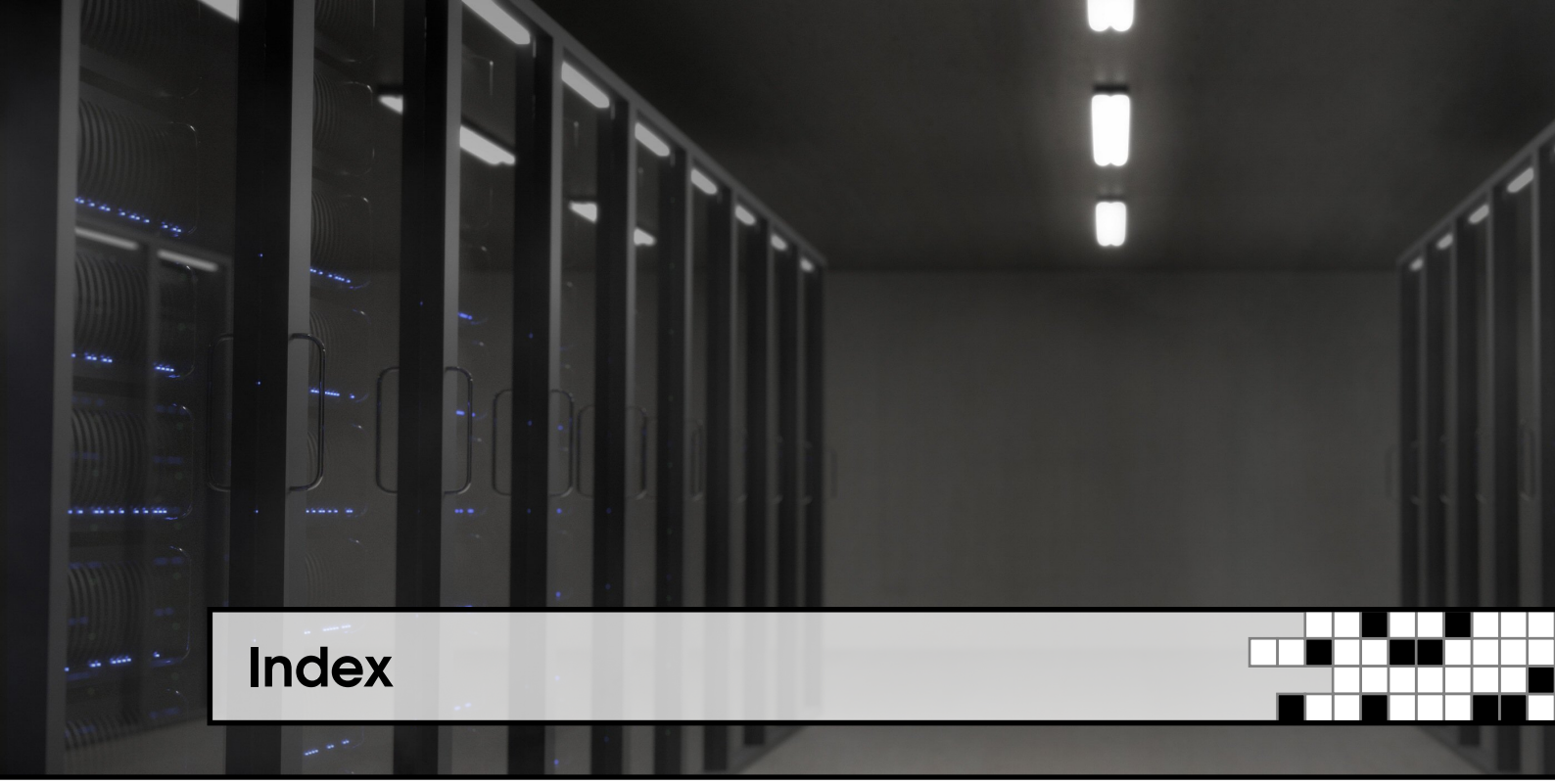
[Okr03] Andrzej Okrasinski. Andrzej Okrasinski's website dedicated to the Game of Life. geocities.ws/conwaylife/, 2003. [Online; accessed May 6, 2016].

[Slo96] N. J. A. Sloane. Sequence A019473 in *The On-Line Encyclopedia of Integer Sequences*. oeis.org/A019473, 1996. Number of stable n-celled patterns ("still lifes") in Conway's Game of Life, up to rotation and reflection. [Online; accessed Jan. 30, 2020].

[Slo99] N. J. A. Sloane. Sequence A046932 in *The On-Line Encyclopedia of Integer Sequences*. oeis.org/A046932, 1999. $a(n)$ = period of $x^n + x + 1$ over $GF(2)$, i.e., the smallest integer $m > 0$ such that $x^n + x + 1$ divides $x^m + 1$ over $GF(2)$. [Online; accessed July 28, 2018].

[Slo00] N. J. A. Sloane. Sequence A056613 in *The On-Line Encyclopedia of Integer Sequences*. oeis.org/A056613, 2000. Number of n-celled pseudo still lifes in Conway's Game of Life, up to rotation and reflection. [Online; accessed Jan. 30, 2020].

[Slo11] N. J. A. Sloane. Sequence A196447 in *The On-Line Encyclopedia of Integer Sequences*. oeis.org/A196447, 2011. The number of parents of successive approximations used in a greedy approach to creating a Garden of Eden in Conway's Game of Life. [Online; accessed May 14, 2016].



Index

Z