Project work 2024

Contents

Introduction

This year the HLP project will be improving Issie's Schematic Editor (AKA Draw Block). Context:

- Issie currently is a lot more capable than in your 1st year!
- Last year's HLP students worked on some enhancements to the Draw Block: better auto-routing, several other things. That was integrated last Summer and we now have solid code which autoroutes well in most cases with wires automatically separating themselves. Try it, moving components in the demos, to see how this works.
- This year your work will be to write whole sheet *beautify* functions, and related property-based tests, to improve the current code. Beautify functions will adjust component orientations, scaling, positions, port ordering, etc to simplify wiring and reduce the number of wire crossings and right-angle bends in wires.

The focus on testing this year is partly because high quality beautify deliverables need this, and partly because property-based testing is an interesting topic, and writing the tests using a functional framework is worthwhile. It is expected that students will work mainly in a pairs: deliverable and tests, with some overlap. The testing motivates what the deliverable should do and the two students can help each other debug etc.

Project teams will share a single group repo and implement three separate beautify features. They will need to make sure that changes are compatible – both at the code level: so that git merges do not break functionality, and at the operation level, so that the features work well together. Because automatic tests will be written with the features "work well together" can be determined systematically.

Feedback from last year's project

Students spent a lot of time understanding the complex Draw Block code, and a lot of time working out what to do. This year the deliverables will be more precisely mandated. Also, a set of driver functions into the Draw Block code will be written in the individual part of the coding: guidance will be given on what these could be and how to implement them. That will allow most of the work to be done without understanding all the Draw Block complexity although inevitably some of this will remain.

Prerequisites

Worksheet 3 contains an introduction to the Elmish architecture. Tick3 contains an introduction to the Draw Block code and operation at a high level, and the test framework used. A Project intro lecture on team deliverables will be available from week 2. See the project 2024 intro slides for more explanation of the Team phase work.

Organisation

The work is split into an Individual phase, and a Team phase.

Teams will be 6 members / team.

The standard number is 6 students working in pairs: working on beautify deliverable, and auto-testing of deliverable.

Number of students in team	**Number of distinct deliverables (One student per deliverable)**	**Number of testers (One tester per one or two deliverables)**
4	2	2
5	3	2
6	3	3

During the Team phase it is expected that:

- Students will work closely together in pairs implementing deliverables and testing them. Therefore in some cases testers may do deliverable implementation, implementors may write test code and analysis.
- Each pair of students must communicate closely in a timely fashion with other pairs in a team to ensure that all deliverables are compatible and merged.
- Pairs must communicate in a timely way to allow Teams to support each other or make contingency plans when there are problems.

Individual Phase Work

The individual project phase work will comprise 3 parts weighted as in the table below.

- Write a required set of *build* and *testing* functions (Figure 1) that will be useful in Team phase work. (40%)
- Improve part of the RotateScale code making it comply better with Issie Coding Guidelines, stating as a set of bullet points how you believe it is improved (30%)
- Using the Figure 1 functions as needed, make a coding contribution to your deliverables of the Team phase work. For example: some tests, some useful helper functions, a partly working beautify function. The functionality you provide, and how it addresses a Team deliverable, must be specified and submitted as comments in the code. (30%).

Code Organisation

- DrawBlock/SheetBeautifyHelpers – helper functions, well documented
- DrawBlock/SheetBeautify – D1,D2,D3 deliverables
- TestDrawBlock – tests for deliverables

To avoid merge clashes when integrating you should add extra modules in separate files: e.g. SheetBeautifyD1, SheetBeautifyD2,SheetBeautify D3, TestDrawBlockD1,TestDrawblockD2,TestDrawblockD3. Then, when integrating (D4), they could easily all be folded into the single file, keeping them as submodules. The Helpers module will contain the specified library functions (choose whose) and otehr functions (well documented and approved) can be added.

Individual Phase Draw Block Build and Test Helper Functions

In the Figure 1, which lists the functions that must be written, R = Read, W = write. Where a value is RW two functions are needed one for Read and one for Write. These should be combined in a Lens (if possible).

All required functions must be written with appropriate signatures and name (see Issie guidelines). You are expected to check other Draw Block code to find appropriate types. The Tick3 intro lecture provides a guide to the values required, and the data structures used to store them in the Model.

Your functions should be written in `SheetBeautifyHelpers`.

Figure 1. Required functions

Key	Type	Difficulty	Value read or written
B1R, B1W	RW	Low	The dimensions of a custom component symbol
B2W	W	Medium	The position of a symbol on the sheet
B3R, B3W	RW	Medium	Read/write the order of ports on a specified side of a symbol
B4R, B4W	RW	Low	The reverses state of the inputs of a MUX2
B5R	R	Low	The position of a port on the sheet. It cannot directly be written.
B6R	R	Low	The Bounding box of a symbol outline (position is contained in this)
B7R, B7W	RW	Low	The rotation state of a symbol
B8R, B8W	RW	Low	The flip state of a symbol
T1R	R	Low	The number of pairs of symbols that intersect each other. See Tick3 for a related function. Count over all pairs of symbols.
T2R	R	Low	The number of distinct wire visible segments that intersect with one or more symbols. See Tick3.HLPTick3.visibleSegments for a helper. Count over all visible wire segments.
T3R	R	Low	The number of distinct pairs of segments that cross each other at right angles. Does not include 0 length segments or segments on same net intersecting at one end, or segments on same net on top of each other. Count over whole sheet.
T4R	R	Medium	Sum of wiring segment length, counting only one when there are N same-net segments overlapping (this is the *visible* wire length on the sheet). Count over whole sheet.
T5R	R	Low	Number of visible wire right-angles. Count over whole sheet.
T6R	R	High	The zero-length segments in a wire with non-zero segments on either side that have Lengths of opposite signs lead to a wire *retracing* itself. Note that this can also apply at the end of a wire (where the zero-length segment is one from the end). This is a wiring artifact that should never happen but errors in routing or separation can cause it. Count over the whole sheet. Return from one function a list of all the segments that retrace, and also a list of all the end of wire segments that retrace so far that the next segment (index = 3 or Segments.Length – 4) - starts inside a symbol.

One helper function has been written in TestDrawBlock.HLPTick3 for you to use.

n/a	R	High difficulty	`See TestDrawBlock.HLPTick3.visibleSegments` The *visible* segments of a wire, as a list of vectors, from source end to target end. Note that a zero length segment one from either end of a wire is allowed which if present causes the end three segments to coalesce into a single visible segment.

Team Deliverables

The Team code deliverable comprises 2 or 3 (depending on number of students) specified beautify functions: D1, D2, D3, integrated with optional unspecified extensions (X1, X2) into an optimised whole-sheet beautify function D4 with a set of tests. The signature of D4 is fixed as below. The tests can be arranged as you like. The assessment is based on code quality assessed separately for both D4 & the tests, and functionality as described in a short document that states achievements and references the tests.

Code quality relates to quality (Issie guidelines) and overall structure, as well as comments. It should be easy to read and add to the code. Quantity is not rewarded, because conciseness can make code more readable, however code that has more functionality and is still easy to understand will gain higher marks than code that has very little functionality, even if it is equally easy to understand.

`beautifySheet: SheetT.Model -> SheetT.Model`

- (30%) Beautify code. All the code that makes up D4, and code improvements to RotateScale.fs
- (30%) Test code. A set of tests that how show well D4 works and identify corner case problems, with a metric that approximates overall visual quality.
- (20%) Analysis. Written as a single Wiki page in your team repo. D4 and how it is tested. The overall functionality of D4 is assessed here as well as how useful are the tests in evaluating this.
    - What are the achievements (and how do tests prove these)?
    - What are the additional problems (X1, X2) addressed or diagnosed (and which tests illustrate them)?
    - What artifacts or other issues remain (with illustrative tests)?
- (20%) Teamwork and integration. This mark will reflect how well the team as a whole has worked together, communicating well, adapting team phase working to optimise deliverables, completing D4 smoothly. Individuals who make an exceptional very high or low contribution to the team will be rewarded or penalised in this mark.

Figure 2. Team deliverables

Function	Adjust on sheet	Primary Optimisation	Test using random sample inputs
`D1. sheetAlignScale`	Custom component scaling. Positioning of all components	Reduce wire segments without increasing wire crossings. Align sets of componennts	Reduction in wire segments. Visual alignment of same-type components: metric that corresponds to overall visual quality
`D2. sheetOrderFlip`	Port order on custom components, flip components, flip MUX input order	Reduce wire crossings	Reduction in wire crossings, other quality measures
`D3. sheetWireLabelSymbol`	Add or remove wire labels (swapping between long wires and wire labels). Improve symbol rendering.	Reduce wiring complexity (symbol rendering is judged manually on appearance)	Test ability to add labels without overlap in presence of adjacent components with visually helpful placement.
`X1. sheetRouteSeparate*`	Change existing routing or separation code to improve it and reduce artifacts	Reduce special case problems, make operation idempotent	Test for idempotence and corner case problem reduction. Test for over-long wires
`X2. sheetReplace*`	All component placement: topology recreated ignoring existing placement	All of the above	All of the above
`D4.`	Integrate (as much as possible) all deliverables into a single function which works well. Provide tests which demonstrate that it works.		

* These deliverables are ambitious and for extension work only for students who have completed D1, D2 or D3.

Appendix A – sheetAlignScale

Name	Algorithm	Suggestions
ASB	Starter	1. Align all singly-connected components to eliminate wire bends in parallel wires 2. Do not overlap components
	Options	3. Where possible align multiply connected components to eliminate wire bends in nearly straight wires 4. Scale custom symbols to reduce wire bends in parallel wires between two custom components 5. Expand algorithm to include wires which are not nearly straight when there is room to move components 6. Include, where this is worthwhile (heuristic) aligning arrays of components. Note that aligning components will usually mean that connections between the aligned components cannot be straight and vice versa. 7. Ensure wires route ok, symbols do not overlap
AST	Test starter	1. Create “near straight wire” singly connected circuits with random small deviations from straight got by moving components from a selected set of manually generated straight-wire circuits 2. Test metric: number of wires straightened, number of component overlaps
	Test Options	3. Create randomly constructed set of straight wire circuits with some small random deviations from straight wires and multiple connections. Note that some wires may be highly non-straight. 4. Create realistic circuits with random components and random connections 5. Test metrics: components do not overlap, wires are not “squashed” between components too close together, wire routing does not become much longer (due to having to go round components it could previously go through).

Basic wire straightening algorithm

Analyse whole schematic to determine which wires could possibly be straightened by moving components, and which component(s) needs to be moved to straighten a wire.

1. Wires which can be straightened have one visual segment (Fig A1) or 3 visual segments (Fig A2) should be straightened by algorithm (to make them all like A1). They are called **parallel wires**.

2. Wires with right-angles cannot be straightened and are ignored by algorithm. From Fig A1 or Fig A2: S1 -> MUX2.1, S2 -> MUX2.SEL. They are called **right-angle wires**. They have 2 visual segments as in A1,A2, or more than 3 visual segments.

3. **Singly-connected components** connect with only one parallel wire. Examples from Figs A1 or A2: (A,B, S1,C). NB S2 has no parallel wires and its position will therefore not be constrained by straightening operations.

4. **Singly-constrained parallel wires** have a single-connected component at one end. They can always be straightened by moving the single-connected component without affecting any other wire.

5. All the parallel wires in A2 are singly-constrained except for MUX1.OUT -> MUX2.0. It is not obvious it can be adjusted to make it like A1: the simplest algorithm would not be able to do this. We can see that in fact there are enough degrees of freedom in the system to make this possible because MUX1 and MUX2 can be adjusted to make MUX1.OUT -> MUX2.0 straight. Every other wire from MUX1 or MUX2 then goes to a singly-connected component.

6. A1,A2 show 4 horizontal parallel wires, and one vertical parallel wire (S1 -> MUX1.SEL). Note that when a component (e.g. MUX1) has horizontal and vertical parallel wires it can be moved in both X & Y directions and therefore be used to straighten, independently, one horizontal and one parallel wire. Therefore the condition for a wire to be singly-constrained is looser than 4. A wire is singly constrained if one end of it connects to a component single-connected in the relevant direction (of the parallel wire). The movements for each component in X and y directions can be independently determined to straighten such wires.

7. Figure A3 shows a more complex case which cannot simply be straightened. Here, in horizontal direction, S1, MUX3, MUX2 are all multiply-connected. It is possible to straighten most but not all of the horizontal parallel wires.
8. Figure A4 shows parallel non-crossing wires between two custom component edges which can be straightened by scaling one (or the other) of the custom components as in Figure A5. This was accomplished manually using the ctrl-drag user interface in Issie but it was quite difficult to do this exactly (in fact it is not quite exact!).

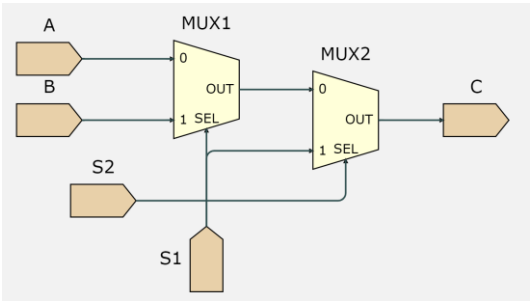


Figure A1

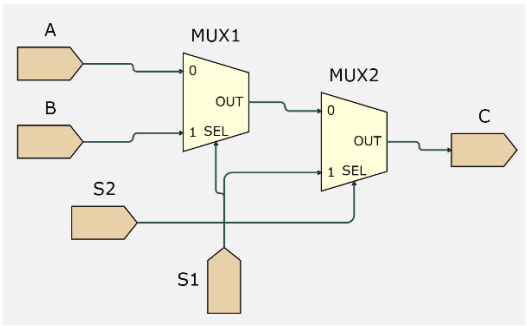


Figure A2

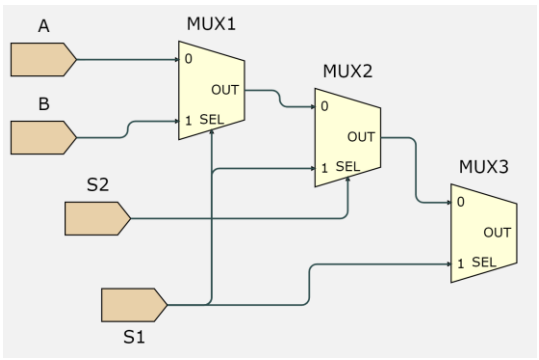


Figure A3

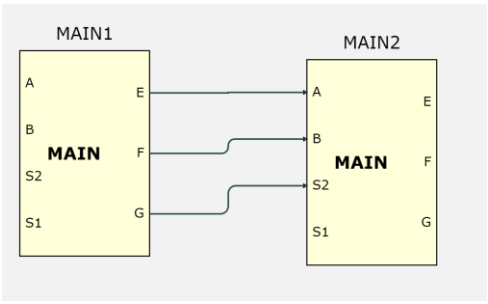


Figure A4

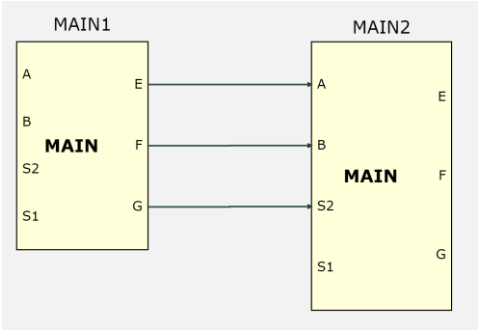


Figure A5

Appendix B – sheetOrderFlip

Name	Algorithm	Suggestions
OFB	Starter	1. Flip all MUX inputs and permute gate inputs to reduce wire crossings Do not increase total number of wire bends
	Options	2. Re-order any custom component ports along any side to reduce crossings 3. Change orientation of gates and MUXes and maybe other components to reduce crossings 4. Heuristic to optimise for minimum crossings + not making wiring more complex + not changing orientation of components that will look weird if that is done.
OFT	Test starter	1. Create manually generated test circuits where gates and MUXes are randomly flipped, or (2-MUX) inputs swapped. 2. Test metrics: number of wires straightened, number of wires crossing
	Test Options	3. Create randomly constructed test circuits with random collection and position of components, random flipping of gates, inputs etc. 4. Constrain these randomly-generated test circuits so they are realistic – e.g. most wires are fairly short. 5. Test metrics: components do not overlap, wires are not "squashed" between components too close together, wire routing does not become much longer (due to having to go round components it could previously go through) as well as those from OFT2

Basic Algorithm to minimise wire crossings

Figure B1 shows an example circuit with wire crossings that can be eliminated (B2) by some combination of vertical flips and 2-MUX input swaps. Note this is an option on the 2-MUX component only that keeps Sel position fixed: so there are 4 possible combinations for a 2-MUX being flipped or having inputs swapped.

Algorithm 1. Exhaustive search algorithm. Try every possible combination of flips and swaps: measure wire crossings each time, choose minimum. This is optimal but slow and unscalable.

Algorithm 2. Clock face algorithm. Determine the likely correct orientation of a component by matching the circular order of its ports with the circular order of the connecting ports. See Figure B3, and MUX2.

- MUX2 has 4 wires connection to ports in order going clockwise: S2, S1, MUX1.OUT, G1.
- The blue arrows show the relevant angles which could be calculated from SVG coordinates and some trigonometry.
- The circular order of the MUX2 ports in Figure B3, starting with SEL and going clockwise, is SEL, 1, 0, OUT.
- Matching these two orders you can see the wiring is compatible with the two orders: SEL -> S2, 1 -> S2, 0 -> MUX1.OUT, OUT -> G1.
- This allows the wires to be routed without crossing, as is the case in B3. In B2 the wires cross, and the two orders are not the same.

Clock face is not always optimal because wires can be routed round a component two ways, but usually if the original wires route to a close side of the component it will be optimal. It allows every component to be considered independently. It is much faster than algorithm 1 which requires the circuit to be routed and then wire crossings to be counted.

Algorithm 3. Heuristic. Use a heuristic to partition components into independent connected groups and then use algorithm 1 or algorithm 2 on each group.

Probably algorithm 1 & 3 is the way to go, but 2 may have some use for example when trying to re-order many ports on a custom component.

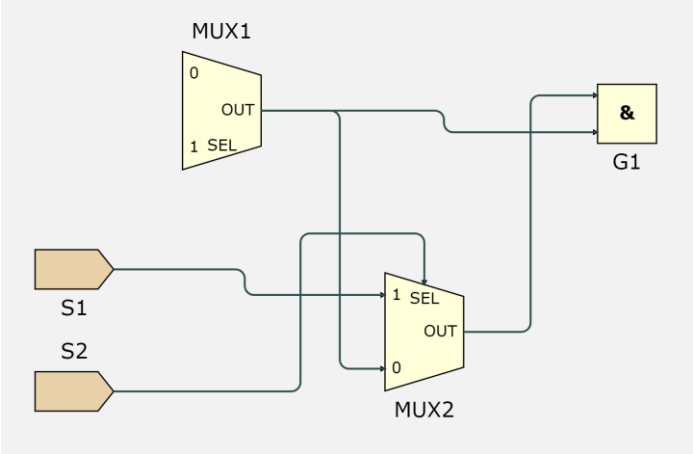


Figure B1

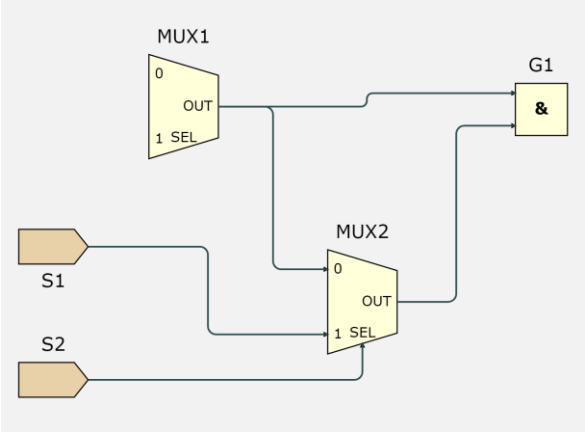


Figure B2

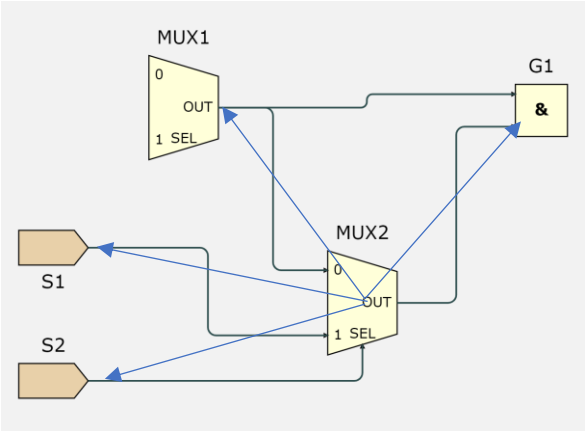


Figure B3

Appendix C – sheetWireLabelSymbol

Name	Algorithm	Suggestions
WLB	Starter	1. Identify long wires. Automatically replace with wire labels where there is room
	Options	2. Choose position of wire label so there is enough room for it 3. Choose good heuristic for when to use labels with user-variable threshold (user can choose). Turn wire labels back into wires when they score below threshold. 4. Refactor symbol rendering. Make bit legends in brackets on wires work properly for all orientations – intelligently move legends where they cross wires or symbols
WLT	Test starter	6. Create manually generated circuits with random component positions 7. Test metrics: symbols overlapping, total number of wire bends
	Test Options	8. Test metric: count when Bit legends on wires overlap wires or symbols

Wire labels: transform Figure C1 to Figure C2.

Legends: (a:b) on wires must be positioned correctly automatically for all orientations (Figure C3).

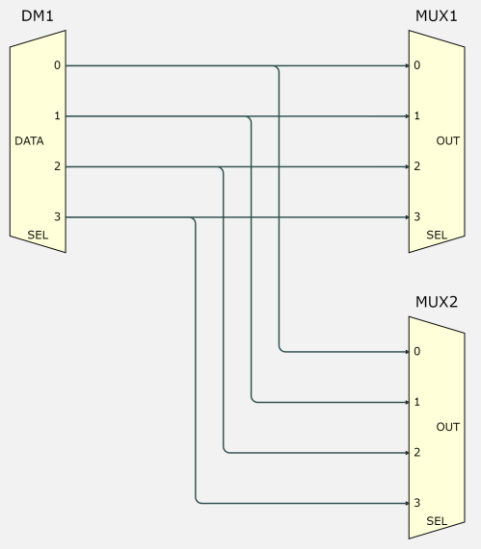


Figure C1

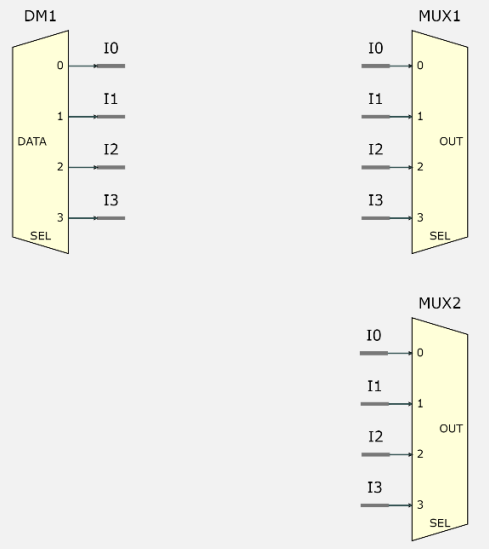


Figure C2

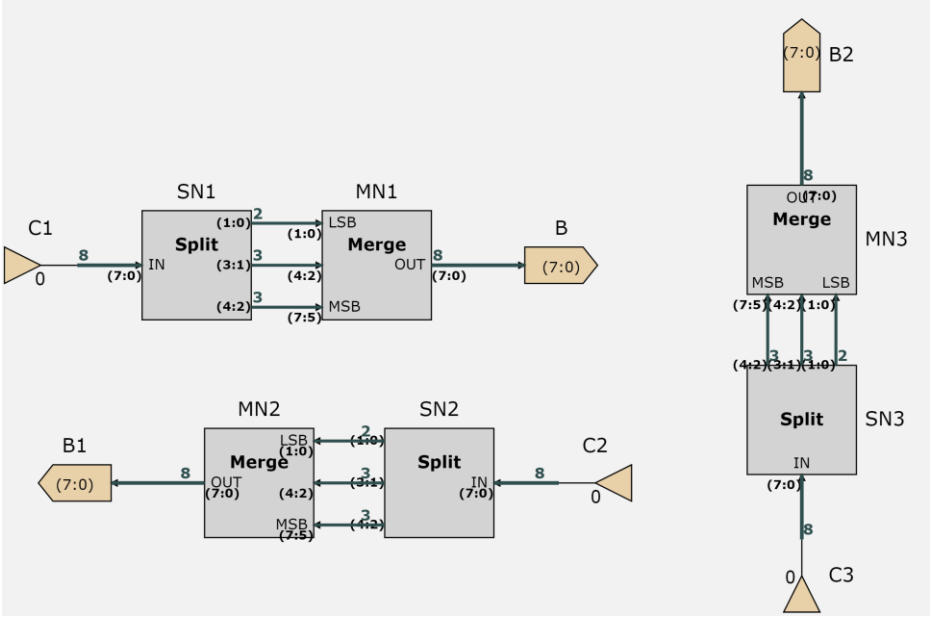
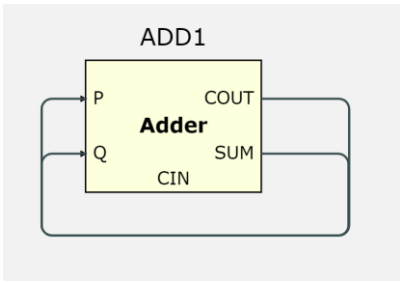


Figure C3

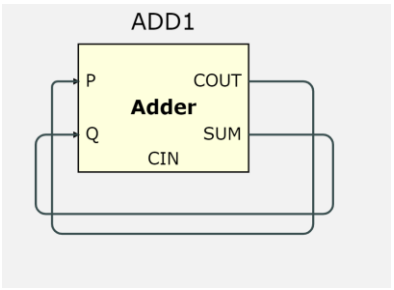
Appendix D – Separation Code (for extension work)

This Appendix need only be read by ambitious individuals when their main deliverable is complete and integrated. I will allow them to add some complex work to successful completion of the main deliverables.

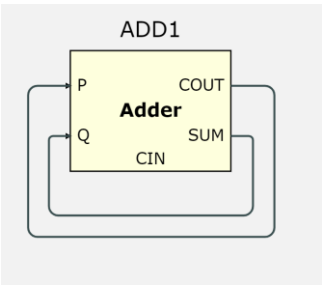
Figure 1 – separating and ordering segments in wires



(a) Routed wires with problems



(b) Wires separated, order non-optimal and separation uneven



(c) Wires separated with equal spacing, and ordered so that there are no wire crossings

The code in `BusWireSeparate` (approx. 1000 lines) is normally run on the whole sheet after `smartAutoroute`. Its purpose is:

- To separate wire segments that would otherwise be autorouted on top of each other, unless they are from the same net (connected to the same output) – in which case they should if possible run on top of each other
- To even up separation so that wires running parallel to each other look uniform.
- To reduce *unnecessary* wire crossings

The top-level pipeline

The separation code is run from **`separateAndOrdermodelSegments`** which has an 8 stage pipeline as shown in Fig 2. The separate function does most of the work. It works by extracting just one orientation (Vertical or Horizontal) of wire segments and processing the segments to get optimal even separation and correct order to reduce unnecessary wire crossings. After this function there are two "cleanup" functions **`separateFixedSegments`** and **`removeModelcorners`** that attempt to deal with special cases and artifacts introduced by the previous functions.

Separating the whole sheet into **`Horizontal`** and **`Vertical`** segments, and processing each independently, greatly simplifies the necessary logic. However running the separation process once for each orientation is not enough: changes in vertical position of horizontal segments caused by the **`Horizontal`** segment separation can make a previous **`Vertical`** segment separation non-optimal. Running the two types of separation several times in this pipeline allows (nearly always) a good stable answer.

The two cleanup functions are only run on newly changed wires (**`wiresToRoute`**) in the case that only part of the sheet has been changed – e.g. by adding a wire or moving a component.

Fig 2: separateAndOrderModelSegments top-level pipeline: BusWireSeparate

```
953             /// convenience abbreviation
954             let separate = separateModelSegmentsOneOrientation wiresToRoute
955
956             // In theory one run Vertical and Horizontal of separate should be enough. However multiple runs work better
957             // chunking togetherclusters that should be connected etc.
958             // TODO: revisit this and see how necessary it is.
959
960             separate Horizontal model // separate all horizontal wire segments
961             |> separate Vertical // separate all vertical wire segments
962             |> separate Horizontal // three more runs allows ordering and "chunking" to work nicely in almost all cases
963             |> separate Vertical
964             |> separate Horizontal
965
966             // after normal separation there may be "fixed" segments which should be separated because they overlap
967             // one run for Vert and then Horiz segments is enough for this
968             // TODO - include a comprehensive check for any remaining overlapping wires after this - and fix them
969             |> separateFixedSegments wiresToRoute Horizontal
970             |> separateFixedSegments wiresToRoute Vertical
971
972             // after the previous two phases there may be artifacts where wires have an unnecessary number of corners.
973             // this code attempts to remove such corners if it can be done while keeping routing ok
974
975             |> removeModelCorners wiresToRoute // code to clean up some non-optimal routing
```

The separate function

The separate function (from Fig 2) has top-level structure as in Fig 3.

1. After some subfunction definitions line 924 creates from the Model wires a list of lines (type `Line list`) where a `Line` corresponds to a specific wire segment or a symbol boundary.
2. There are two pipelines that perform the processing.The first pipeline 931-933 is procedural code which for performance reasons operates on mutable data (see `Line` type definition for the mutable fields) by partitioning the segments into clusters where each cluster is a set of adjacent parallel segments that should be separated. Clusters are defined so that adjacent symbol edges (which segments cannot be moved across) always form a cluster boundary.
3. After this has run the final pipeline 931-933 (immutable data) writes the segments whose positions have changed back into the `Model` by changing the corresponding wires. The output is thus a `Model` with wires that have had segments separated where this is desirable.

Fig 3. `BusWireSeparate.separateModelSegmentsOneOrientation` **(NB – this is the same as** `separate` **in Fig 2).**

```
897   /// Perform complete segment ordering and separation for segments of given orientation.
898   /// wiresToRoute: set of wires allowed to be moved.
899   /// ori: orientation
900   let separateModelSegmentsOneOrientation (wiresToRoute: ConnectionId list) (ori: Orientation) (model: Model) =
901
902       (*
903           TODO: would it be better, overall, to separate all wires?
904           This was done before excludeclustersWithoutWiresToRoute
905           To go back to this - remove excludeClustersWithoutWiresToRoute.
906       *)
907       /// Add linked line changes before movement changes. Movement changes will override
908       /// linked line chnages if need be.
909
910       let allWires = model.Wires |> Map.keys |> Seq.toList
911
912       /// We do the line generation for ALL wires
913       /// Then, after all  segments are clustered, we actually change segments only in clusters
914       /// that contain wiresToRoute. The other clusters should not need to be reseparated.
915       let excludeClustersWithoutWiresToRoute (lines: Line array) =...
923
924       let lines = makeLines allWires ori model
925
926       // Procedural pipeline that divides line segments into clusters and then calculates new positions
927       // for segments within each cluster.
928       // Mutable data is used here as fields on each line record for performance reasons, because
929       // finding sets of same net lines which overlap can muhc more efficiently be done with mutable links.
930       // In spite of mutable data the code is functional in spirit.
931       makeClusters lines
932       |> excludeClustersWithoutWiresToRoute lines
933       |> List.iter (calcSegPositions model lines)
934
935       // The pipeline here (no mutable data) takes the final line segment data and writes it back into Model
936       // changing wires according to the new segment positions.
937       lines
938       |> Array.toList
939       |> adjustSegmentsInModel ori model
```

Known Problems

Function	Problem	Possible fixes
`BusWireSeparate: makeClusters`	If clusters are too small related wire segments can be put in different clusters and therefore not be properly separated. If clusters are too large unrelated segments can be pulled together	This does not seem much of a problem now. If needed in an analysis could be done of wire segments that might be pulled far way from the rest of the wire and this used to modify cluster generation.
`BusWireSeparate: calcSegPositions`	Ordering segments for minimum overlap sometimes gives different results according to whether Vertical or Horizontal segments are so ordered. This can lead to overall separation which changes segment order when repeated (not idempotent).	Maybe having another parameter to separate that controls which segments are re-ordered and going from wire ends through to wire middle as segmentation is repeated within one call to `separateAndOrderModelSegments` would lead to consistent results? Not sure whether `separateFixedSegments` or `removeModelCorners` can also cause problems? Needs investigation
`BusWireRoute: smartAutoroute`	Sometimes wires end up routed on top of symbols (as noted in Tick 3) because none of the routes checked by smartAutoroute is viable.	Can probably quite easily be fixed by changing the different routes tried by `smartAutoroute` before it gives up: however maybe this fix will cause non-optimal wire routing in other ways?
`BusWireSeparate: separateFixedSegments`	The special segmentation done here to prevent wires overlapping can sometimes cause overly complex wires that can't be simplified by `removeCorners`.	Not sure. Maybe better segment ordering in `calcSegPositions` will reduce this problem?