BusWire, Symbol, Sheet

HLP Project 2022 Specification

Issie & HLP Project Work

- Buswire, Symbol, and Sheet comprise the Draw Block in the Issie (Intuitive Simulator and Schematic Integrated Editor) application: https://github.com/tomcl/ISSIE
- See the ISSIE Wiki for an overview of how Symbol, BusWire and Sheet work inside Issie.
- The 2022 project work will be to improve and enhance the code in BusWire and Symbol (and possibly Sheet) modules.
- You can play with Issie, checking functionality, by downloading the latest release.
- You can (with Github login) fork & clone (or template copy) the repo, then follow the README instructions for setting up a full dev environment to compile and run it. That takes only 15 min. Changes in the code will (usually) be hot reloaded and appear in the running app.
- If you only want to view or edit (not run) the code in these modules, install Visual Studio 2022 and .Net 6 as specified on the F# tools setup page and then double-clicking the top-level Issie.sln file in a local (clone of fork) copy of the Issie repo. Setup for this takes only 10 minutes.

Buswire Code Partitioning: 1,2,3

Buswire is badly written at the moment at many levels

- Poor code partitioning
- Poor quality code

It has some interdependence of different functionality

- This should be rationalized and turned into module helper functions which are consistently used

The code divides:

- Types, interface and conversion
- Render wires and segments
- Autoroute (initial), move segments, and partial autoroute on movement
- Work out jump positions
- Update

Lines	Function	Who?
16-92	Types	all
94-158	Debug print functions	all
160-308	Interface Conversions (segments, vertices, etc)	1
309-425	Misc segment functions	1
425-672	Render & view functions	1
674-996	Autoroute	2
998-1142	Partial autoroute	2
1144-1222	Work out jump positions	3
1223-1486	Update function	3
1491-1554	Interface functions	3

Buswire - enhancements

Operation	Enhancement	Individual coding	Group coding
Wire type	Change Wire, Segment to allow easier coding, and wires with arbitrary orientation endpoints See also symbol – endpoint orientation comes from symbol ports and need not be part of wire type.	1,2,3	
Render wires	Allow arbitrary orientation of endpoints (with bit width text correct)	1	
	Switch display to radiused corners on wires, or display of circles or display of jumps: slide 13	1	
Autoroute	Allow any orientation when autorouting. Must use symmetry to reduce cases: slides 7-11. (Slide 12: autoroute with sticking to same net wire as low priority extension)	2	YES
	Autoroute round symbols		YES
Partial autoroute	Allow any orientation when partial autorouting from endpoint to nearest manually routed (so fixed) segment		YES
Wire jumps	Switch to calculate jumps or circles: slide 13	3	
Drag segments	Stick to same net wire, stick to no kink (if extra segments allowed). Slide 12.		YES

Symbol Code Partitioning: 1A, 1B, 2

- All the code must be refactored for new types
- Symbol code splits into two parts fairly cleanly
  - View function and render subfunctions
  - Port positions, copying symbols (working out new labels)
- In order to render ports their positions must be known, so there is some dependence
- View function is crucial, and will have two people (1A,1B) independently implementing individual code.

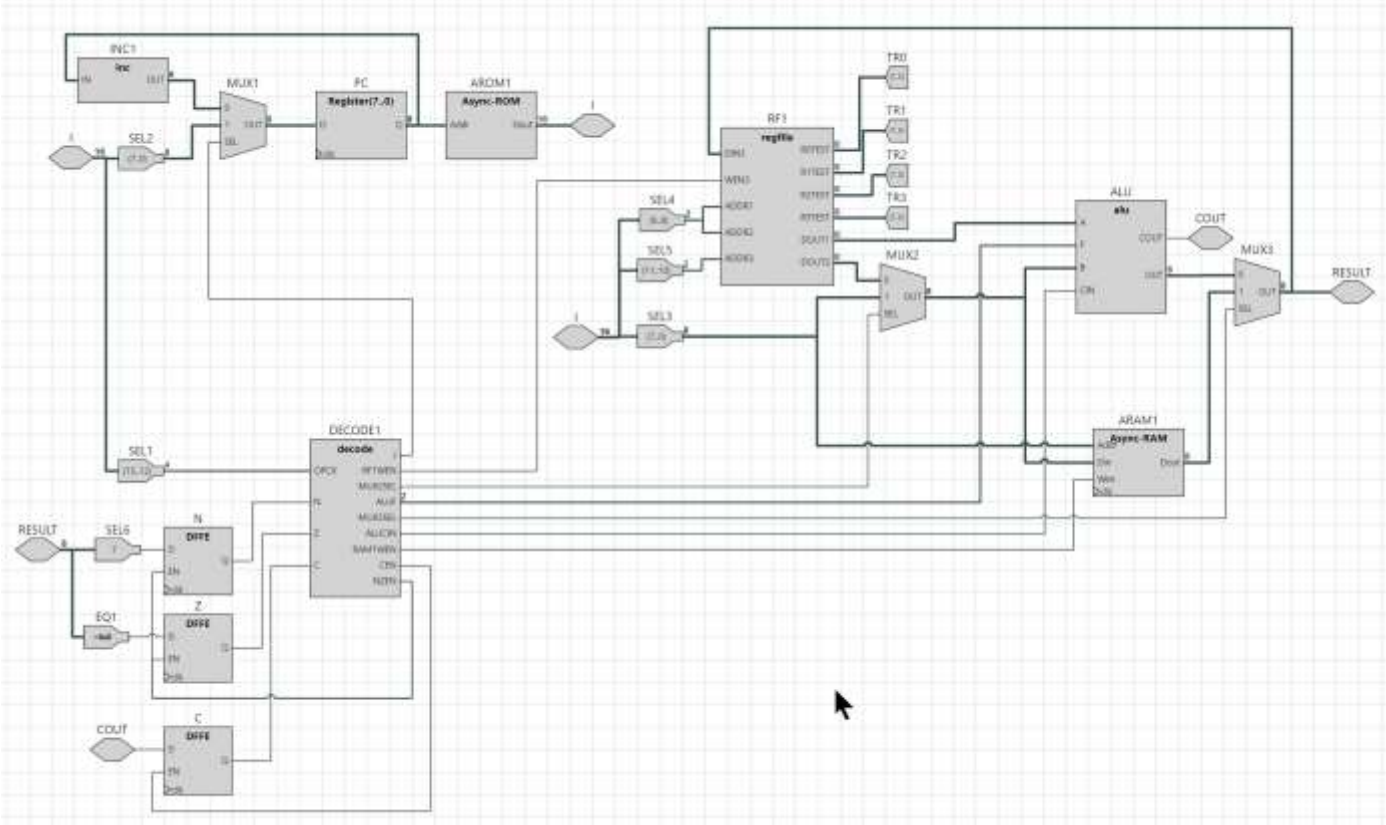
Lines	Function	Who?
1-68	Types	1A,1B, 2
70-500	View	1A,1B
501-602	Port positions	2
603-867	Copy symbols	2
870-1044	Update function	2
1047-1053	Direct interface to Issie	2

Symbol - enhancements

Operation	Enhanced operation	Individual Coding	Group work
Types	Add rotation, adjustable edge for custom comp ports (this also changes port orientation). Adjustable order for ports	1A,1B,2	
Render symbol	Add 90, 180, 270 degree rotation (simple symbols). Slide 14-16	1A,1B	
	Render custom component ports on arbitrary edges		YES
	Better way to specify compactly how each symbol is rendered. Slide 19		YES
	Make MUX + etc symbols better with ports on other edges	1A,1B – MUX, DFFE, REGE, ADDER	
	Make custom components that contain clocked logic have a clock on their symbol (like Reg, RegE). This would not be a port – just a visual indicator.		YES
Locate port position, given symbol position.	Locate port position and orientation, given symbol position and orientation	2	
n/a	UI to rotate symbol, e.g. with right-mouse-click	2	
n/a	Move port to different edge of symbol (custom symbols only). Order ports on symbol. Slide18.		YES
Auto-align to grid	Auto-align to other symbols		YES
Bounding box	Make symbol bounding box work with rotation	2	
Output/Input symbols from Issie to save/restore	Can incorporate rotation by allowing Component W,H fields to be -1 times real values (4 rotations can be encoded like that). Port movements could be recorded in a complete separate data structure with all extra symbol info per sheet		YES

BusWire - routing and wire geometry type

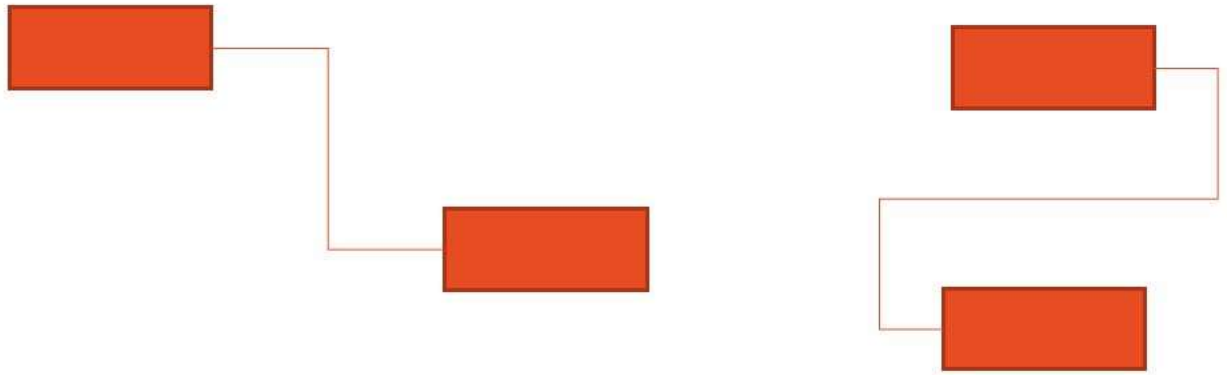
- These slides show how more general wire autoroute could be implemented, and give advice about wire data structures



Issie Wire Routing

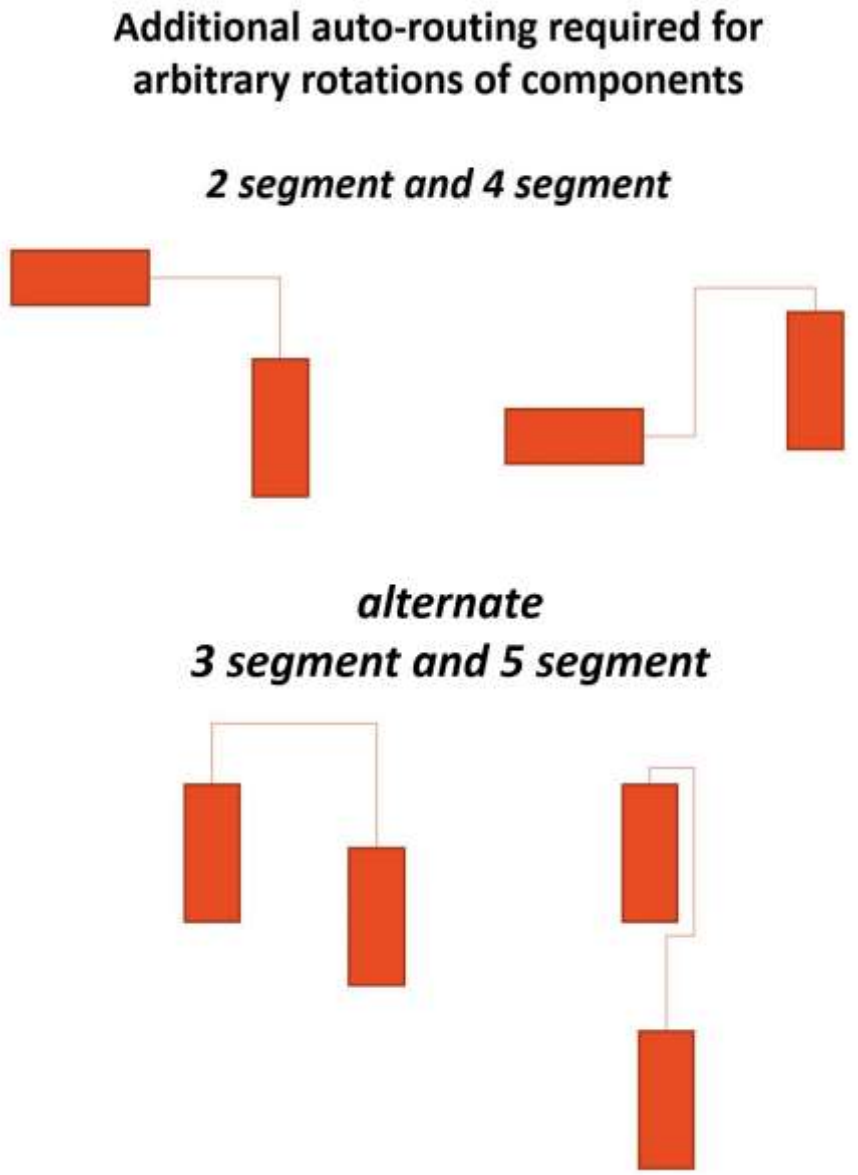
- Wires are auto-routed on creation, or component move
- Issie outputs and inputs are always LHS (RHS) of components
- All auto-routing has the two forms below

Issie *current* auto-routing: 3 or 5 segment



Routing: required

- Rotating components can lead to:
  - 4 new wire geometries as shown.
  - Any wire geometry rotated by any multiple of 90 degrees
  - Any wire geometry reflected.



Wire data structures: notes

- Segments currently each contain position info as 4 floats + direction
- Highly redundant
- Using -c instead of c for a coordinate to indicate that a segment has been manually adjusted and cannot be autorouted is very unpleasant leading to abs everywhere in the code - a separate boolean would be better.
- Not easy to rotate / reflect translate
- Alternative would be 1 float/segment representing segment length and direction (positive or negative in x or y) with initial position and rotation and Y axis reflection for wire. This is rotation-invariant
- Compact rotation-invariant structures are better for algorithms that *change* wires, segment-based structures for algorithms that need to *read* segment positions.
- Could add new rotation-invariant form of wires, with function to derive something like current form from this. That would allow existing code still to work.
- Segments (with absolute cords as now) would be recalculated after any wire change as a *dependent* part of the model.

```
// A straight line segment of a wire
type Segment =
    {
        Id : SegmentId // unique id for segment
        Index: int // index of segment within Wire list of segments
        Start: XYPos
        End: XYPos
        Dir: Orientation
        HostId: ConnectionId // Wire this segment is part of
        // List of x-coordinate values of segment jumps. Only used on horizontal segments.
        JumpCoordinateList: list<float * SegmentId> // SegmentId is id of the crossing segment
        Draggable : bool
    }

// A wire (or bus) connecting a component output to a component input
type Wire =
    {
        Id: ConnectionId // unique Id for wire
        InputPort: InputPortId // unique ID of input (driven) symbol port
        OutputPort: OutputPortId // unique id of output (driving) symbol port
        Color: HighLightColor // color to display wire
        Width: int // width of wire in pixels
        Segments: list<Segment>
    }

    with static member stickLength = 16.0
```

Wire Data Structures and Helpers: Requirements

- Change absolute position wire segment data structure to ASeg with:
  - absolute start XYPos
  - length (+ or – on relevant coordinate)
  - direction (X or Y),
  - manually routed or auto-routed (currently done by negating coordinates)
- Define relative wire data structure RISegs that holds segments in a rotation-invariant (RI) form allowing routing algorithms to exploit rotation symmetry
- There is some flexibility in the type definitions, but must include the above
- You don't have to use my type names

- Redefine Wire in Model so it contains either Aseg list, or RISegs
  1. Model.Wire contains Aseg list
     - create RI type on demand, update model absolute type from RI type
     - Helpers need for the two transformations
  2. Model.Wire contains RISegs
     - Helpers needed for the two transformations
     - High order helper foldOverRISegs, see next slide
- Throughout data structures I take no view whether you should use arrays or lists

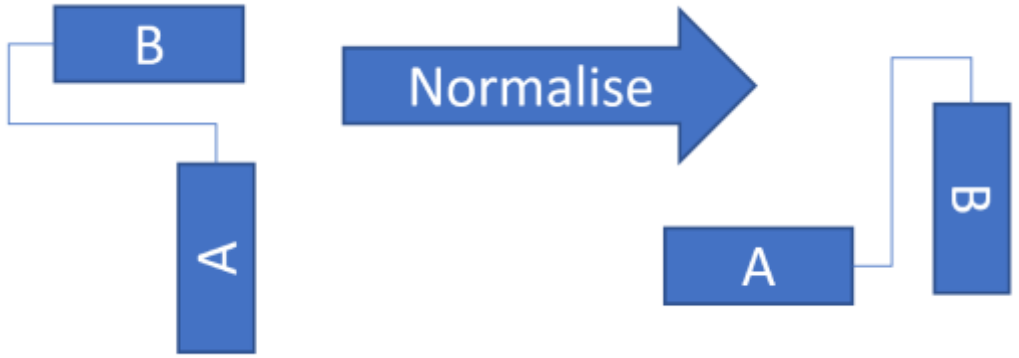
```
ype ASeg = {
    Dummy: Unit


ype RISeg = {
    DummyRI: unit


ype AWire = {
    DummyA1:unit
    DummyA2: unit
    Asegs: ASeg list


ype RIWire = {
    Dummy1: unit
    Dummy2: unit
    RISegs: RISeg list
    }

et foldOverRISegs
    (initState1: 'State1)
    (wire: RIWire)
    (folder: 'State1 -> ASeg -> 'State1) =
    let initState2 = failwithf "to be implemer
    let riFolder = failwithf "to be implemente
    ((initState1, initState2), wire.RISegs)
    ||> List.fold riFolder
    |> fst
```

foldOverRISegs

- Motivation
  - If all wires are held in RISegs form then to check whether a wire has been clicked, check whether wires cross, or generate SVG of a wire an aSeg list form is needed
  - To generate an ASeg list and then process it is inefficient.
    - More flexible and efficient would be to generate the ASeg list and process it at the same time
  - Any processing can be implemented in a fold operation
  - Where wires don't change the more compact RISegs form can be cached (WireRenderProps) and regeneration of the SVG is not needed
- This is an untried technique, so it might seem some modification
- It seems a neat way of implementing BusWire without needing to store an absolute position data structure for Wire, calculating this "on the fly"

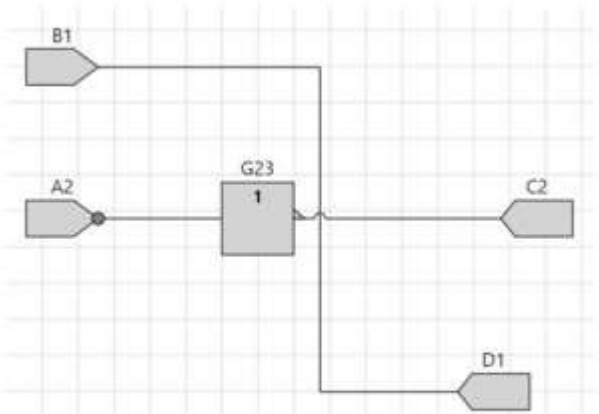
Using symmetries

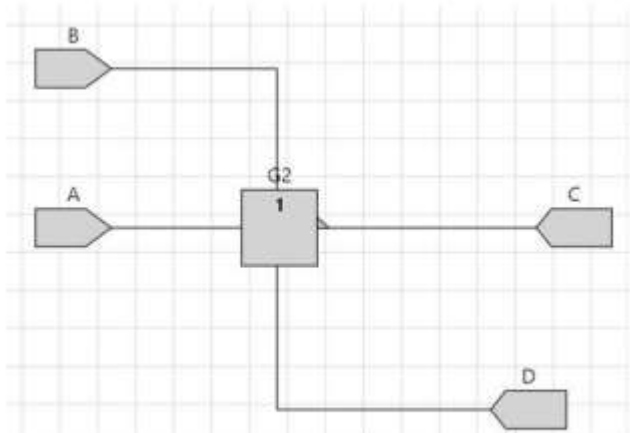
- For simpler autorouting wires can first be normalized by choosing a rotation around wire starting point and possible reflection so that e.g.:
  - First segment is always in positive X direction
  - Second segment is always in positive Y direction
  - The positions of each segment end-point can then be calculated from this info
  - The ith segment is always horizontal for even i, vertical for odd i.
- Solve general autoroute problem by reducing to equivalent normalized problem and solving that.
- Algorithms can be implemented with many fewer cases!

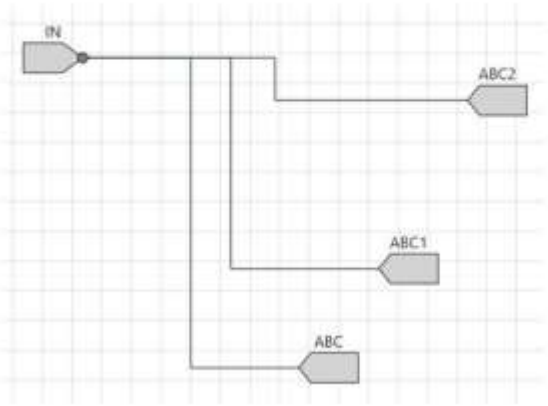


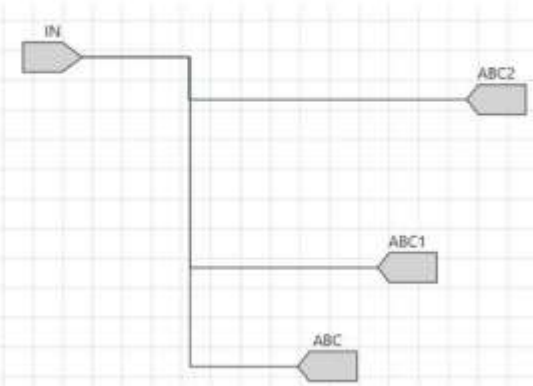
Wires: smart autoroute

- Avoid going through components
- Autoroute a new wire with existing wires so it joins neatly onto the nearest part of common nets
  - Manual routing could snap same nets together
- .



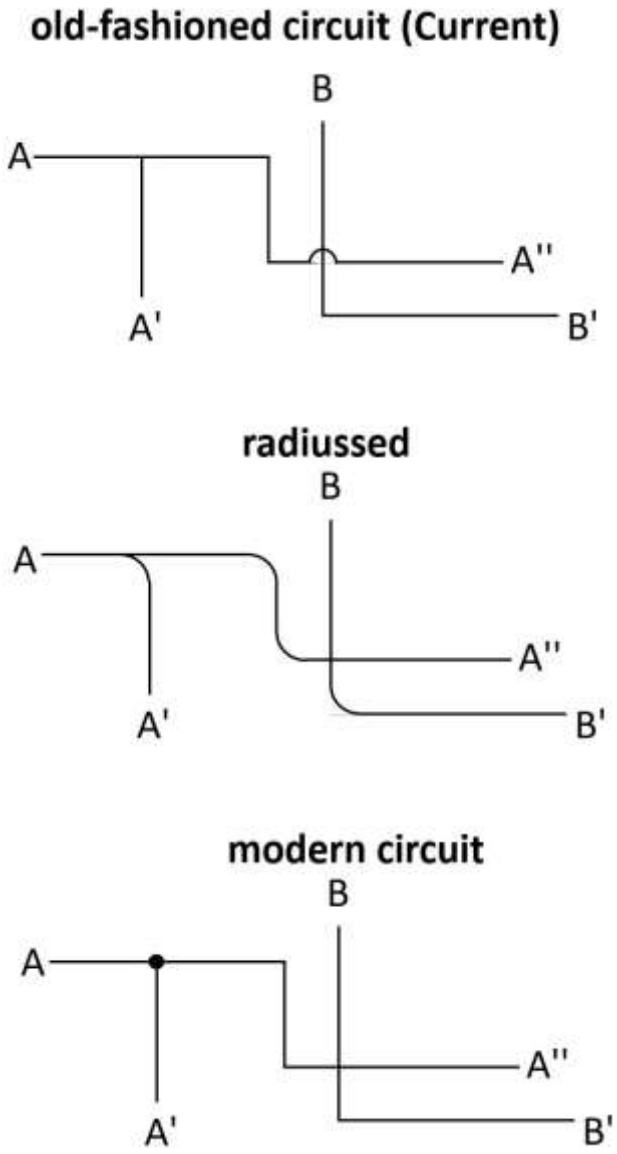






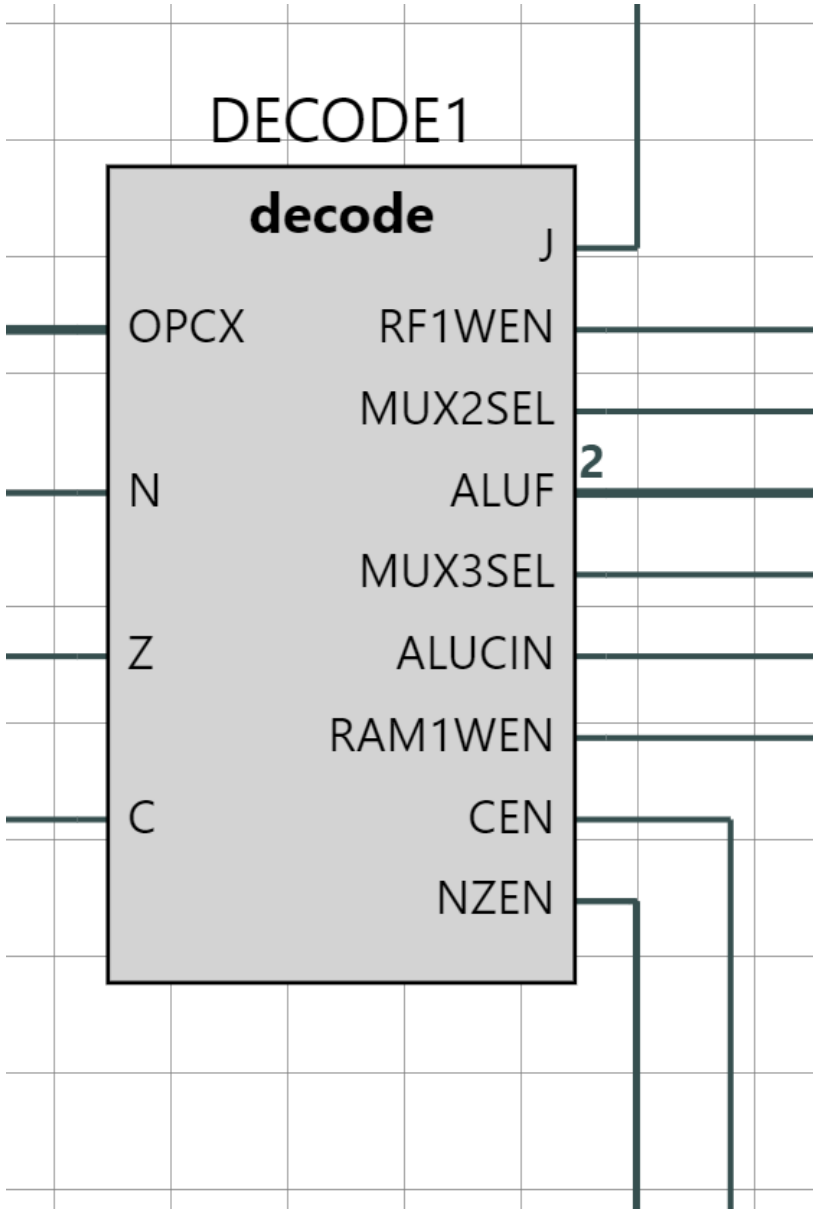
Wire notation

- Radiussed wires are very easy to implement using SVG Paths and drawing one wire as a single multi-vertex path
- Old-fashioned circuit style required explicit display of jumps
- Modern circuit style requires explicit display of dot joins and
- Switchable style via UI menu with all three implemented would be ideal
    - Wire Style could be stored on disk per project in the .dprj file, which currently is empty.



Symbol

- These slides show how a more general Symbol module could be implemented, and give advice on the Symbol data structure.



Current Symbol types

```
type Symbol =
    {
        Pos: XYPos // position of top-left corner of symbol
        InWidth0: int option // bus width of input - used by MergeWires and SplitWire
        InWidth1: int option // bus width of input - used by MergeWires
        Id : ComponentId  // unique ID for symbol 9same as for underlying component)
        Compo : Component // Issie component data structure behind symbol
        Colour: string // color of interior of symbol (used to hoighlight selection or errors)
        ShowInputPorts: bool // true to highlight inpout ports positions on symbol
        ShowOutputPorts: bool // true to highlight output port poistions on symbol
        Opacity: float // 1.0 => normal opaque. < 1 => translucent (when moving)
        Moving: bool // true => symbol is being moved
    }

type Model = {
    Symbols: Map<ComponentId, Symbol>
    CopiedSymbols: Map<ComponentId, Symbol>
    Ports: Map<string, Port>                          // string since it's for both input and output ports

    InputPortsConnected:  Set<InputPortId>            // we can use a set since we only care if an input port
                                                      // is connected or not (if so it is included) in the set
    OutputPortsConnected: Map<OutputPortId, int>      // map of output port id to number of wires connected to that port
    }
```

Issie Ports and Components

```
// Specify the position and type of a port on a Component
type PortType = Input | Output
/// A component I/O.
/// Id (like any other Id) is a string generated with 32 random hex characters,
/// so it is (practically) globally unique. These Ids are used
/// to uniquely refer to ports and components. They are generated via uuid().
/// PortNumber is used to identify which port on a component, contiguous from 0
/// separately for inputs and outputs.
/// HostId is the unique Id of the component where the port is. For example,
/// all three ports on the same And component will have the same HostId.
type Port = {
    Id : string
    // For example, an And would have input ports 0 and 1, and output port 0.
    // If the port is used in a Connection record as Source or Target, the Number is None.
    PortNumber : int option
    PortType : PortType
    HostId : string
}
```

```
/// Id uniquely identifies the component within a sheet.
/// Label is optional descriptor displayed on schematic.
type Component = {
    Id : string // unique ID
    Type : ComponentType
    Label : string // All components have a label that may be empty.
    InputPorts : Port list
    OutputPorts : Port list
    X : int // left X coord
    Y : int // top Y coord
    H : int // height (Y)
    W : int // width (X)
}
```

Enhancing Symbol - Rotation and port placing

Currently all inputs are on LHS, all outputs on RHS, of symbols

- Some symbols would be easier to read and wire to with some ports fixed to top or bottom
  - Examples: MUX (Select), ADDER (Cin and Cout)

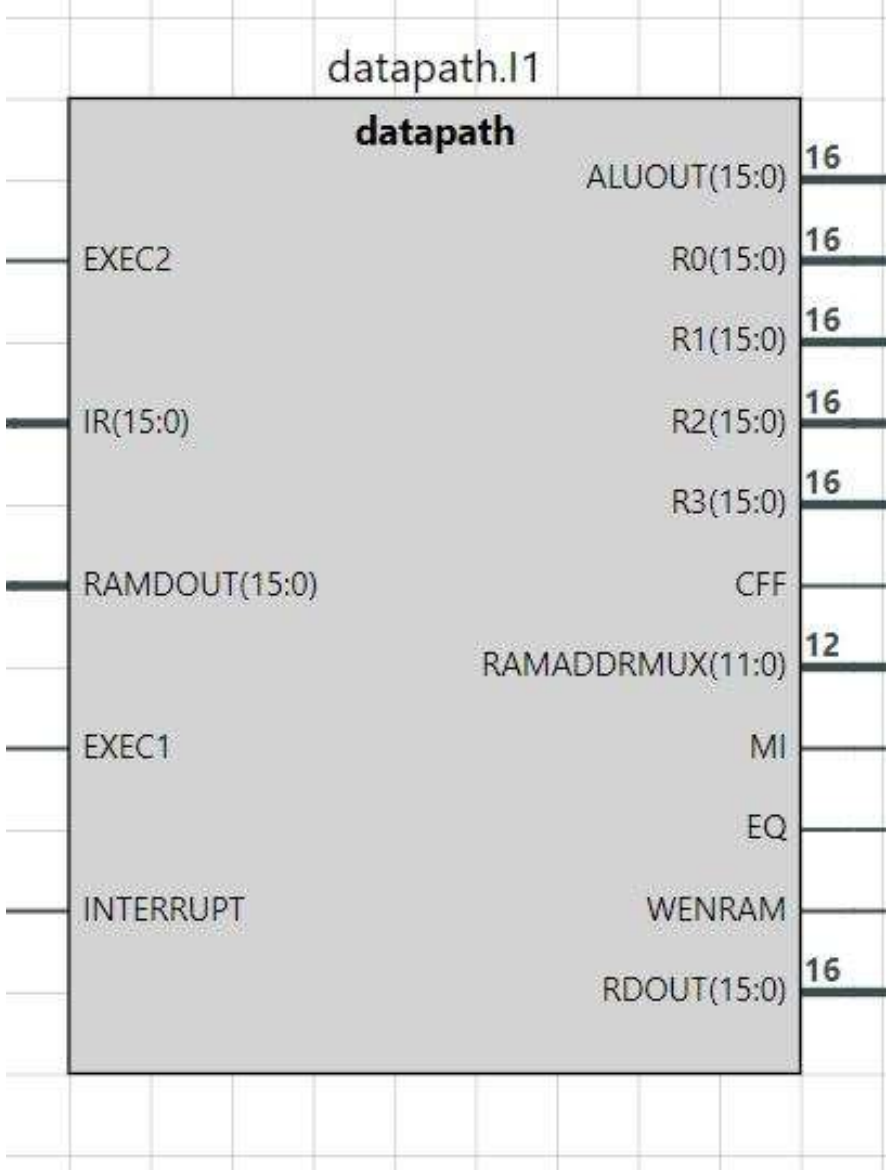
Currently symbols can only be placed on schematic in one orientation

- Allowing some symbols (possibly not the complex ones) to be rotated would make make it easier to compose good schematics. Port name placement (which must remain horizontal) can be a problem when rotating.

Enhancing Symbol - Custom Components

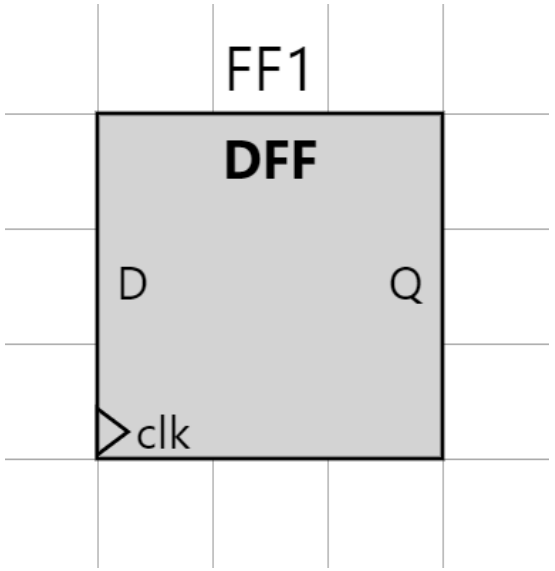
- Custom components represent design subsheets
- Currently all inputs/outputs are on LHS/RHS of symbol
- To make subsheets easier to use, in order of priority:
  - 1. (highest) Move ports to different edge of block
  - 2. Choose in what order a port is placed on its edge
  - 3. (lowest) Add gaps between ports for grouping purposes
- Extra information is needed in Symbol type to do this
  - It can be saved in CustomComponentType extra fields (for load/save to disk)

Minor change (see next slide)
Add clock symbol to top edge of a custom component if it contains clocked logic



Annotating the Custom Component Symbol

- Custom components can be all combinational, or contain clocked logic
- They should have a clock indication if they contain any clocked logic, just like DFF components.
  - This is not a port: all clocks are assumed connected and no connections are made on the sheet.
- To display this (or not) create a `SimulationGraph` from the custom component `CanvasState`, check it
  - `Builder.runCanvasStateChecksAndBuildGraph`
  - `SynchronousUtils.hasSynchronousComponents`
- Some care is needed to do this only when it is needed, on symbol load, since it takes some time.
- You can choose where this symbol is put – this may affect port positioning



Enhancing Symbol: refactoring symbol generation code

- Currently the code that renders the symbol of a component, and calculates corresponding port positions for Buswire, is a mess
- Refactor this code working out a better way to determine the per-symbol shape and text data, so new symbols could easily be added
  - This is an open problem – with many possible approaches
  - A good solution would make adding a new symbol fast, with default sizing automatic based on port names and positions
- The new implementation will use new type for symbol with added information specifying port placement on symbol, and symbol rotation
  - Agreed by all Symbol team members at start of coding

Symbol Types and helper functions: Minimum Requirements

- Add the following
  - **STransform**: field in Symbol type to say how symbol is rotated/flipped
  - **PortOrientation**: type which says which side of symbol port is on
- All type and helper function names can be changed
- Additional fields can be added: this is needed if custom components have more flexible port positions
  - Fields can be added to types after individual phase

- Store port positions in Symbol as one of:
  - 1. Transformed offsets and orientations of ports relative to symbol
    - **APortOffsetsMap**: Map to determine offset and orientation of port relative to symbol
    - **genAPortOffsets**: Helper function to determine APortOffsetsMap
    - **getPortPos**: helper to generate port position and orientation from symbol and port
  - 2. Transform-invariant offsets of ports relative from symbol
    - **TIPortPosMap**: Map to determine transform-invariant offset and orientation of port relative to symbol
    - **genTIPortOffsets**: Helper to determine RIPortOffsetMap from symbol
    - **getPortOffset**: Helper to generate the absolute offset and orientation of a port from the Symbol

Weeks 5/6

- Suggested timescale. It may be slightly changed by Teams, interviews with advisors can be scheduled at different times, etc
- Teams need time to look at code, make initial decisions, check with advisors, make changes, in that order.
- Wednesday PM is deadline for all to be decided since time is needed for individual coding
- Team members less able to participate in this process must accept what others decide

Week	Date	Action
5	Thursday PM	Teams announced
5	Friday PM	Teams decide who does what, agree timescale for week 6
6	Monday lecture slot	Initial types, helpers decided
6	Monday lecture slot	Team advisor interview
6	Wednesday PM	Final changes in types and who does what are made