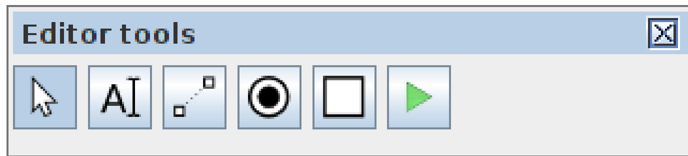
The main area of the window is occupied by the **Editor** panel where the model is displayed for editing and simulation. The frequently used editing and simulation actions are accessible at the right side via **Editor tools**, **Tool controls** and **Property editor** panels, that are detailed below. The **Output** panel at the bottom provides information on the currently executed task.

Editor tools

This panel provide access to frequently used tools for capturing, modification and simulation of the graph models.



Some of the tools are implemented in the core of Workcraft and are available in all the plugins. These include selection, connection and text note tools.

- \- **selection tool** for editing, moving, deleting, and grouping the model elements (can also be activated by pressing `S`). When this tool active you can select the graph elements and modify them as follows.
  - Click a graph element to select it. Outline a rectangular area to select several elements. Outline *from-right-to-left* for adding fully covered elements and *from-left-to-right* for adding any touched elements.
  - Hold `Shift` to include elements into selection and `Ctrl` to exclude elements from selection.
  - Press `Ctrl+A` to select all components (*Edit→Select all*). Press the exclamation mark to inverse the selection (*Edit→Inverse selection*). Press `Esc` to resets the selection (*Edit→Deselect*).
  - Selected components can be removed by pressing `Delete` (*Edit→Delete*), cut by pressing `Ctrl+X` combination of keys (*Edit→Cut*), copied by pressing `Ctrl+C` keys (*Edit→Copy*), and a previously copied part of a model can be inserted by pressing `Ctrl+V` keys (*Edit→Paste*).
  - Double click inside a group to enter it (same action as `Page↑`). Double-click outside the current group to go one level up (same action as `Page↓`).
  - Use left mouse button or `←`, `↑`, `→`, `↓` to move selected components.

- \- **text note generator** for creating textual comments (can also be activated by pressing `N`).

- \- **connection tool** for connecting the model nodes with arcs (can also be activated by pressing `C`). When the connection tool is active, click on the source node to initiate a connection, then click on the destination node to complete the connection. You can hold `Ctrl` key to connect nodes in continues mode, so that each destination node becomes a source node for the next connection.

Most of the tools are model-specific and are implemented by the corresponding plugin. For example, the Directed Graph plugin implements a basic vertex generator tool while the Petri net plugin implements place generator, transition generator and simulation tools.

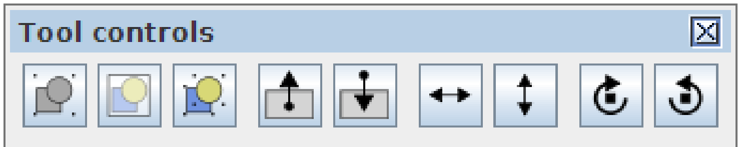
- - create Petri net places (can also be activated by pressing P).
- - create Petri net transitions (can also be activated by pressing T).
- - activate simulation of the model (can also be activated by pressing M).

Tool Controls

This panel provides access to the extended functionality (if present) of a selected tool. Of the generic tools only the **Selection tool** and **Simulation tool** have such extended functionality. Let us consider them in detail.

Selection controls

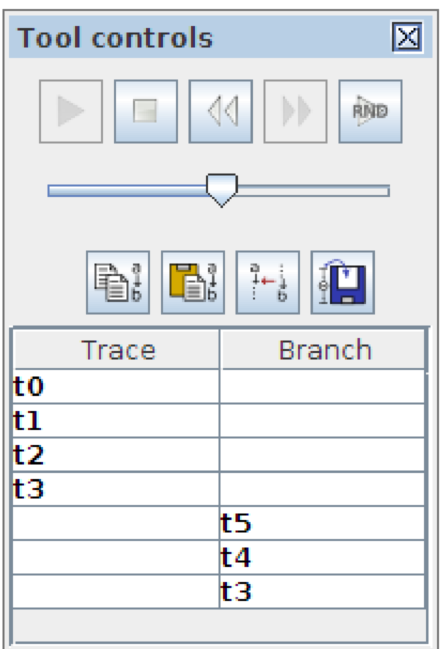
The selection tool controls provide the means to transform the selected nodes and connections of the model.



- - combine the selected elements into a group (or press Ctrl+G).
- - combine the selected elements into a page (or press Alt+G).
- - decomposes the selected group into the comprising elements (or press Ctrl+Shift+G). .
- - level up action: if the focus is currently inside a group, then pressing this button shifts the focus to the container group or the root of the model. The same can be done by pressing Page↑ or *double-clicking* outside the group boundaries.
- - level down action: if a single group is selected, then pressing this button enters this group. The same can be done by pressing Page↓ or *double-clicking* inside the group boundaries.
- - flip selected elements horizontally (or press Ctrl+F).
- - flip selected elements vertically (or press Ctrl+Shift+F).
- - rotate selected elements clockwise (or press Ctrl+R).
- - rotate selected elements counterclockwise (or press Ctrl+Shift+R).

Note the difference between groups and pages. Groups are just unnamed decorations for several nodes, while pages are named nodes containing other nodes.

Simulation controls



The simulation tool controls provides the means to analyse and navigate the simulation data. There are two sources of simulation data:

- **Trace** - the *base* sequence of events, often from an external tool, e.g. a trace leading to a deadlock.
- **Branch** - the *deviated* sequence of events executed by explicitly clicking the excited nodes of the model.

Usually the event names correspond to the model nodes whose execution changed the state of the model. The sequences of events are recorded in the corresponding columns of the *Trace–Branch* table. You can click the name of the event in either column to restore the model state just before that event has happened.

The navigation through the simulation data can be done with the following buttons:

- - execute the trace and branch events starting from the current position.
- - stop the execution and reset the trace and branch data.
- - undo the last event that lead to the current state.
- - execute the next event in the trace or branch.
- - randomly execute events from a pool of events that are enabled in the current state.
- - copy the trace, the branch and the current simulation state into the clipboard.
- - paste the trace, the branch and the current simulation state from the clipboard.
- - save the current state of the model as its initial state.

The slide bar under the navigation buttons controls the speed of playback for the existing or randomly generated sequence of events.

Property editor

The Property editor panel enables displaying and modifying the attributes of a model and its elements. It has four distinctive modes of operation:

- **Element properties.** When a single element is selected its properties are displayed and are available for editing.
- **Combined properties.** When a group of elements is selected a "combined" list of their properties is displayed. Those properties which have the same name and class are combined under one editor item and its modification will propagate too all selected components of relevant class. If the initial value of the combined property cannot be agreed between the selected components an empty grey box is shown.
- **Model properties.** When no elements is selected the model-specific properties are displayed. This can be seen in the Policy Net plugin where a list of bundles is shown with a possibility to edit their names, colours and the list of bundled transitions.
- **Template properties.** When a node generator or connection tool is activated, their *template* properties are displayed and can be modified. All all subsequently created nodes or connections inherent these template properties. To reset the template properties for a particular tool to their default settings just double-activate the tool (i.e. activate the tool while it is already selected).

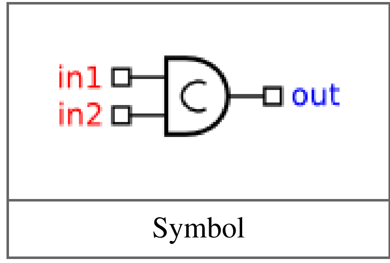
The base list of properties depends on the element type but may be extended by the plugins implementing a particular interpreted graph model. The elements available in all the models are *nodes*, *connections*, *groups* and *text notes*.

Tips and Tricks

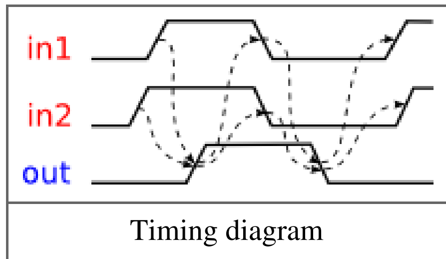
- Selection
  - Hold `Shift` to include objects into a selection and `Ctrl` to exclude objects from a selection.
  - Outline a selection rectangle *from-left-to-right* for adding objects that are inside the selection region, and *from-right-to-left* for adding objects touched by the selection region.
  - Press `Crtl+A` to select everything or `Ctrl+I` to inverse the selection.
  - Use *left mouse button* or `←`, `↑`, `→`, `↓` to move selected components.
  - Selected components can be removed by pressing `Delete`.
  - Press `Ctrl+A` to select all objects or `Esc` to reset selection.
- Clipboard and History
  - Clipboard operations are allowed between the models of the same type: `Ctrl+C` to copy, `Ctrl+X` to cut and `Ctrl+V` to insert.
  - History of modifications can be browsed: `Ctrl+Z` to undo and `Ctrl+Shift+Z` to redo.
- Navigation and Grouping
  - `Ctrl+G` combines selected objects into a group and `Ctrl+Shift+G` splits selected groups into individual objects.
  - Press `Page↓` or *double-click* a group to enter it. Press `Page↑` or *double-click* outside a group to leave it.
  - Scroll the mouse wheel *forward* to zooms in and *backward* to zoom out. Alternatively press `+` to zoom in and `–` to zoom out. Press `Ctrl+0` to restore the default scale.
  - Press `Ctrl+F` to fit the selection into the screen or `Ctrl+T` to centre it.
  - Use the *middle mouse button* or `Ctrl+`*right mouse button* or `Ctrl+←`, `↑`, `→`, `↓` to pan the view.
- Simulation
  - Use `[` and `]` keys to navigate through the simulation trace.
  - In *Signal-State* table the values of excited signals are depicted in bold font.

Synthesis and verification of C-element

C-element [http://en.wikipedia.org/wiki/C-element] is a latch that synchronises the phases of its inputs. A symbol for a 2-input C-element and its timing diagram are shown in the figures below. Initially all the signals are in the low state. When both inputs `in1` and `in2` go high, the output `out` also switches to logical 1. It stays in this state until both inputs go low, at which stage the output switches to logical 0.



Symbol



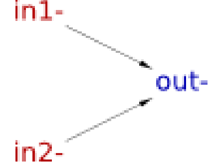
Timing diagram

Modelling

Let us model the C-element behaviour using STG formalism. Create a new STG work called *stg-celement* and *translate* the sequence of events in the timing diagram into a sequence of STG transitions. Basically you need to create a signal transition for each event of the timing diagram and capture the causality between these events by means of directed arcs between signal transitions. Recreate the following STG model in Workcraft.

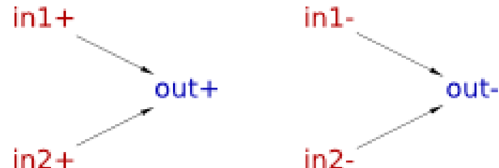
Start with the reset phase of the C-element where `out-` event is caused by `in1-` and `in2-` events:

- Activate the **signal transition generator**. Pay attention to the hint at the bottom of the screen: *Click to create a falling (or rising with Ctrl) transition of an output (or input with Shift) signal.*
- Click the Editor panel in the location where you want the `out-` transition to appear.
- Hold `Shift` and click in the desired location of `in1-` transition. The default name for the input signal is `in`, therefore you need to rename it. Go to the Property editor and change *in name* from `in` to `in1`.
- Create `in2-` transition the same way as `in1-`.
- Activate **connection tool**. Pay attention to the hints at the bottom of the screen. Initially the hint says *Click on the first component.* After the first component is chosen, the hint changes to *Click on the second component or create a node point. Hold Ctrl to connect continuously*.
- Click `in1-`, move the mouse pointer to the `out-` transition and click again. A causality arc from `in1-` to `out-` will be created.
- Repeat the connection procedure to capture causality between `in2-` and `out-` transitions.



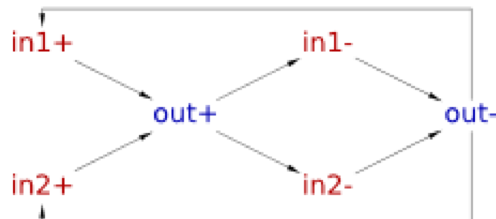
Similarly capture the set phase of the C-element:

- Create transitions `in1+`, `in2+` and `out+`. As before, rename the default `in` signal to `in1` and `in2` respectively.
- Create causality arcs from `in1+` to `out+` and from `in2+` to `out+`.
- Use the **selection tool** to rearrange the STG nodes similar to the following figure.



Now connect the set and reset portions of the specification:

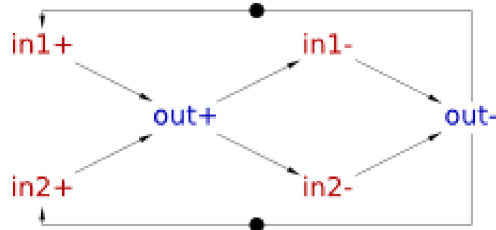
- The same as before, use **connection tool** to introduce causality from `out+` to `in1-` and to `in2-`.
- Create an arc from `out-` to `in1+`, but this time instead of a straight line connection use a polyline. After choosing the source of the connection (`out-` transition) click aside of the existing transitions - this will create a node point of a polyline. Continue forming the shape of the polyline by creating its node points, and finally choose the destination transition `in1+`.
- Repeat the same procedure for creating a polyline connection from `out-` to `in2+`.



Finally, specify the initial state of the C-element in accordance to the starting point of the timing diagram:

- Activate the **selection tool** .
- Select the arc from `out-` to `in1+` and in the Property editor set the *Tokens* value to 1.
- Similarly put a token on the arc from `out-` to `in2+`.

The resulting STG should look as follows.



Validation and verification of specification

Activate the **simulation tool** and exercise the obtained STG model. Click one of the enabled signal transitions (they are highlighted in orange) to *evaluate* the STG into the next state. Note that the sequence

of fired transitions is recorded in the simulation trace that is somewhat similar to the original timing diagram. Check that the simulation traces correspond to the intended behaviour of C-element.

Before proceeding to the synthesis of the C-element it is a good idea to verify that its specification meets essential requirements, e.g. that it is consistent, is free from deadlocks, and output-persistent.

To verify the signal consistency (i.e. that the rising and falling phases of each signal alternate in all possible execution traces) select *Tools→Verification→Consistency [MPSat]* menu item. Similarly, for deadlock checking select *Tools→Verification→Deadlock [MPSat]*, and for output-persistency select *Tools→Verification→Output persistence (without dummied) [MPSat]* menu item.

If the specification violates any of these properties then a trace leading to the problematic state will be reported. This trace can be simulated for better understanding the reported issues and for correcting them in the specification.

Synthesis

The STG specification can now be synthesised into an asynchronous circuit implementation either with Petrify or MPSat backend tools via *Tools→Synthesis* menu.

A complex gate solution obtained with Petrify (*Tools→Synthesis→Complex gate [Petrify]* menu item) is as follows: (Note that solution is not unique and you may get a slightly different equation.)

```
[out] = in2 (in1 + out) + in1 out;
```

By opening the parenthesis one can obtained the following equation:

```
[out] = in1 in2 + in2 out + in1 out;
```

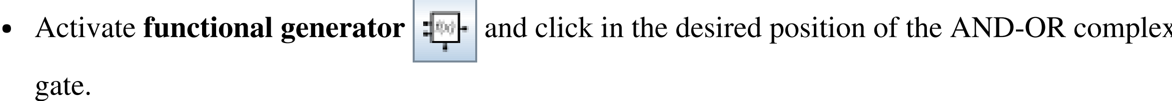
This equation can be directly mapped into an AND-OR complex gate whose function is `Z = A*B + C*D + E*F` ; let us call it *AO222* gate.

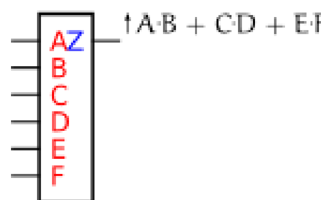
Circuit designers use hardware description languages, such as Verilog [http://en.wikipedia.org/wiki/Verilog] or VHDL [http://en.wikipedia.org/wiki/VHDL], to precisely describe the circuit. For example, the association of the C-element ports to the *AO222* gate pins can be described by the following Verilog module (if you are not familiar with Verilog you can safely skip this part as it is not required by the rest of the tutorial):

```
module celement (in1, in2, out);
  input in1;
  input in2;
  output out;
  AO222 inst_c (.A(in1), .B(in2), .C(in2), .D(out), .E(in1), .F(out), .Z(out));
endmodule
```

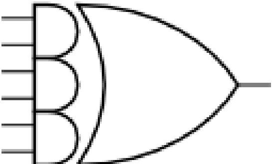
Circuit capturing

Create a new Digital Circuit work called *circuit-celement-cg* and capture the implementation suggested by Petrify in form of a gate-level netlist. In the future versions of Workcraft the derivation of a circuit from the synthesis output will be automated, but for now please do it manually.

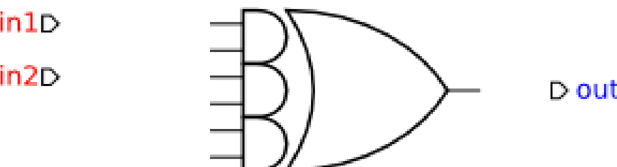
- Activate **functional generator**  and click in the desired position of the AND-OR complex gate.
- Activate **selection tool** .
- Select the only pin of the newly created function component.
- In the Property editor change the *Name* of the pin to `Z` and modify its *Set function* to `A*B+C*D+E*F`.



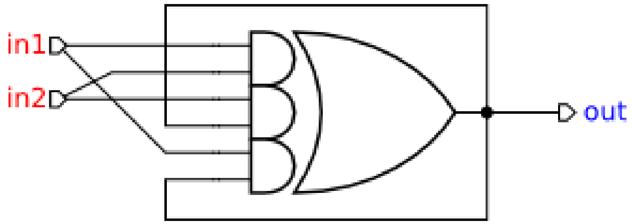
- Select the function component and in the Property editor change its rendering type from *Box* to *Gate*.



- Activate **port generator** . Pay attention to the hint at the bottom of the screen: *Click to create an output port (hold Shift for input port).*
- Click in intended location of the output port. Note that by default the port will be named `out1` - you can change this name to `out` later.
- Hold `Shift` and click in the desired locations of the input ports – they will be automatically assigned `in0` and `in1` names.
- Switch to the selection tool. Choose the output port, go to the property editor and change port *Name* to `out`. Similarly change the name of input port `in0` to `in2`.

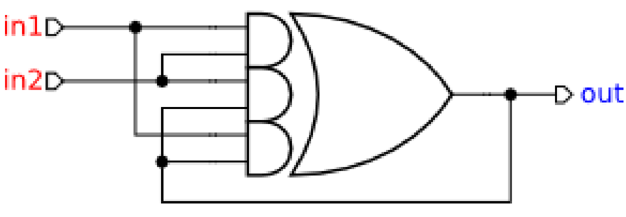


- Activate **connection tool** .
- Connect input ports `in1` to the 1st and 5th pins of the complex gate (`A` and `E` respectively).
- Connect input ports `in2` to the 2nd and 3rd pins of the complex gate (`B` and `C` respectively).
- Connect the output pin of the gate to the output port `out` and to the 4th and 6th inputs of the gate (`D` and `F` respectively).



Optional simplification

You may want to tidy up the circuit schematic and make it more readable by adding forks into the wires. This can be done using the **joint generator** to create the wire split points and then connect wires to them. If you create a connection to/from a wire then a joint will be automatically inserted. This resultant circuit should look similar to the following diagram; this implementation can be downloaded circuit-celement-cg.work (3.09 KiB, 5d ago).

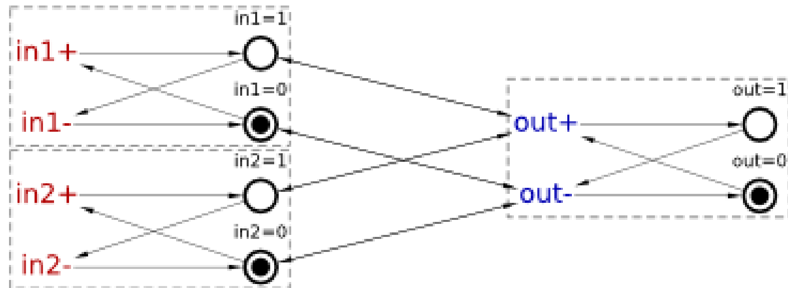


Verification of implementation

Activate the **simulation tool** and simulate the captured complex gate implementation of the C-element. Ports, pins and wires are colour-coded: blue means low level and red means high level of the signal. Excited pins and ports are highlighted in orange.

Click one of the excited pins to toggle its logical value. Similar to the STG simulation, the sequence of signal events is recorded in the simulation trace and can be subsequently replayed for analysing the circuit's behaviour.

To conduct formal verification the circuit has to be converted into an STG. Normally this is done silently by the tool, but it is possible to view the intermediate circuit-STG via *Tools→Conversion→Signal Transition Graph*. A result of such conversion for the C-element circuit is shown below.



Note that if the C-element's inputs are not restricted in any way then they can change in an unexpected manner and cause malfunction of the circuit. This can be confirmed by checking the circuit for hazards *Tools→Verification→Hazards [MPSat]* – the circuit has a hazard after the following trace: `0: in2+, in1+`.

Play this trace to discover that indeed, by the end of the trace the output of the C-element is excited and ready to switch to logical 1, but an unexpected `in1-` or `in2-` transition would disable it, that does not fit the original STG specification.

The STG specification contains some information that is not available in the circuit, namely the behaviour of the environment. The circuit can work correctly only in an environment that respects its contract

specified by the original STG. In other environments the circuit may exhibit hazards, deadlocks, etc.

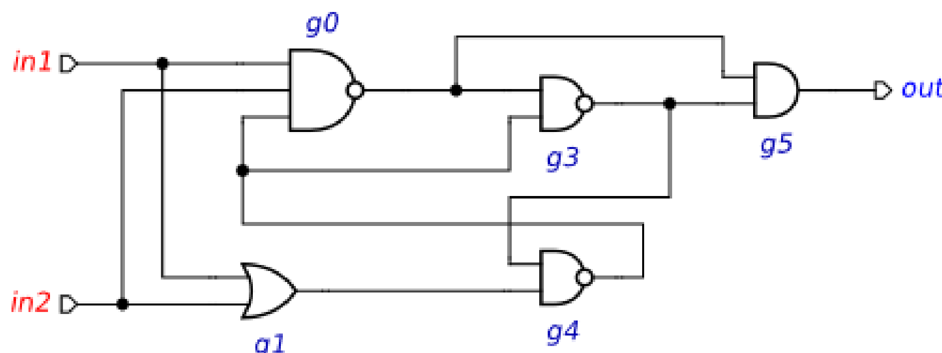
Therefore, to conduct formal verification of the circuit one has to restrict the environment behaviour in such a way that the circuit only receives those inputs that are allowed by the STG specification. This can be achieved as follows:

- In the circuit editor make sure that no components are selected (click on the editor canvas).
- In the property editor choose the *Environment URI* property and select the work file with the original STG specification.

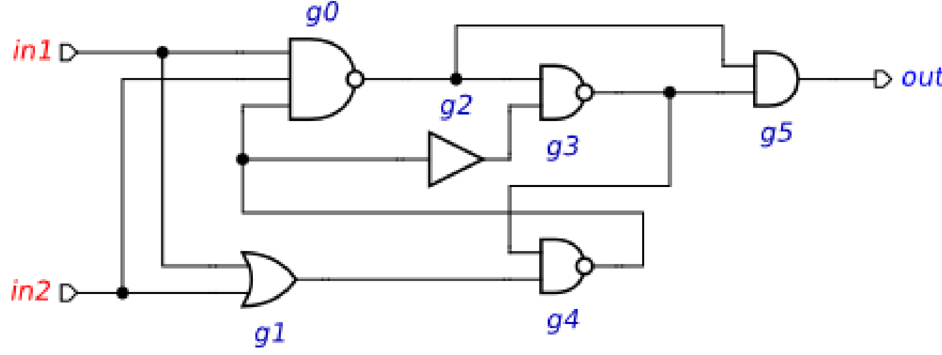
Repeat the verification procedure to check if the circuit is free of hazards under the well behaved environment. Also check the circuit for deadlocks and verify if it conforms to the environment specification. All these verification steps can be run via *Tools→Verification→Conformation, deadlock and hazard (reuse unfolding) [MPSat]* menu.

Large gates like `AO222` may not be available in the technology library and thus other implementations of C-element using smaller gates are of interest. However, it is very easy to make a mistake when designing asynchronous circuits, and so any such implementation has to be formally verified against the original STG specification.

Download the following candidate C-element implementation circuit-celement-decomposed.work (3.78 KiB, 5d ago) and verify that it conforms to the STG specification and is free from deadlocks and hazards.



Note that the correctness of this implementation depends on the *isochronic forks assumption*: the difference in arrival times of a signal to the ends of a wire fork is negligible compared to any gate delay. If this assumption is violated, the above implementation can exhibit hazards. Download the following C-element implementation circuit-celement-decomposed-hazard.work (3.9 KiB, 5d ago) where a delay in a wire fork is made explicit (modelled by a buffer). Formally verify this model, inspect the violation trace, and explain what can go wrong in it.



Optimising Asynchronous Pipelines Using Wagging

In this practical we consider asynchronous pipelines and use formal techniques to model and analyse them, and to improve their performance.

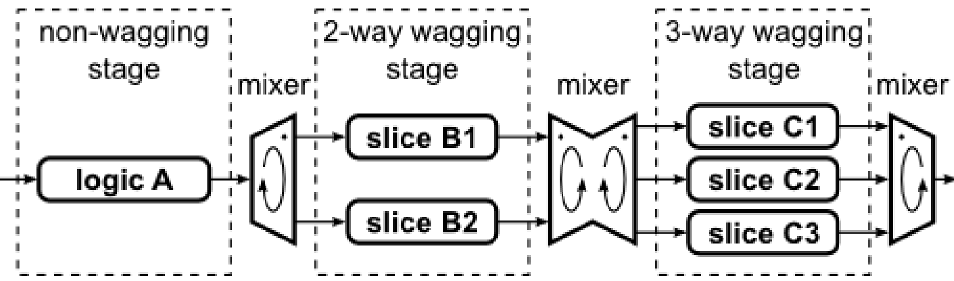
Static Data-Flow Structures

Many systems can be abstractly represented by *Static Data-Flow Structures* [1], which are closely related to pipelines. This abstraction separates the structure and the function of the system from the implementation details of its components.

The possibility of formally modelling and reasoning about the system at this high-level architectural level is crucial, as the design decisions made at this level will affect all the subsequent stages of the design. Moreover, optimisations performed at this level are likely to have a much stronger impact than micro-optimisations applied towards the end of the design process.

Wagging

Wagging [2] is a technique for improving the throughput of a pipeline by replicating the operational unit of a stage and cycling which copy (*slice*) the tokens should go though - see the picture below. (The analogy is with a dog wagging its tail: 2-way wagging has been applied to the 2nd stage in the picture, which means that the 1st token entering the stage goes into the upper slice, the 2nd token goes into the lower slice, the 3rd token goes into the upper slice, etc.) The tokens exit the stage in the same order they have entered it. It is assumed that the pipeline is implemented as an electronic circuit (hence the term `logic' is used instead of `operational unit'), though wagging can be applied to almost any kind of pipelines.



Each stage has a *wagging level* - the number of copies of logic it contains. The inputs and outputs of each slice are connected to *mixers* collecting the data from the outputs of one wagging logic stage and then delivering it to the current slice of the next stage. The mixers also latch the data to allow the connected slices to work independently. The above picture shows an example pipeline with a combination of non-wagging and wagging logic stages. These are connected using a selection of mixers.

Advantages of wagging:

- can significantly increase the throughput of the pipeline;

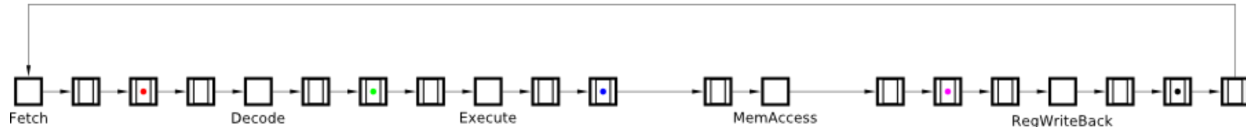
- requires almost no effort from the designer (the technique can be automated to a large extent);
- the energy consumption of wagging circuits is nearly identical to the original non-wagging circuit which forms each of the slices [2].

Disadvantages of wagging:

- logic has to be replicated, which incurs circuit area overhead;
- latency is increased due to mixers.

Conceptual RISC pipeline

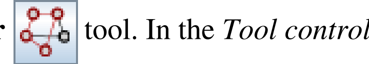
Consider the following conceptual RISC pipeline. Recall that in asynchronous pipelines tokens must be separated by empty registers, and so a cycle with *N* tokens must contain at least *2N+1* registers to avoid deadlocks, and the best performance is achieved with *3N* registers, like in the pipeline below.



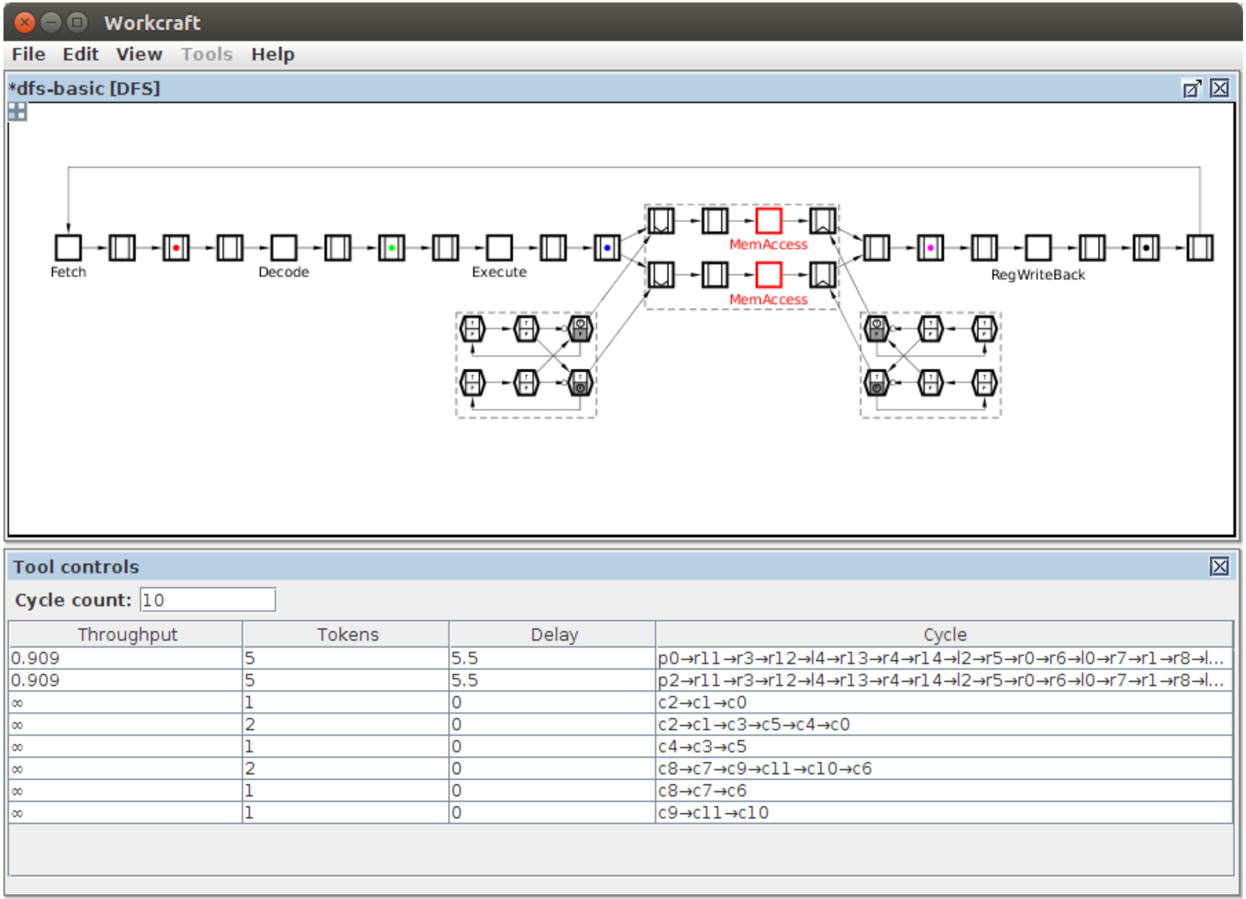
Re-create this *Dataflow Structure* model in Workcraft. For simplicity, set all the register delays to 0, and the delays of combinational logic blocks to 1, except for `MemAccess` whose delay should be set to 3, making it the bottleneck. Make sure that token colours are all different.

Simulate this model to get the feel of the *spread token semantics* [3]. Note that:

- the enabled elements are highlighted in orange - these are:
  - empty registers which can accept a token (only possible when the next register is empty);
  - registers which can lose a token (this is only possible after the token has `spread' to the next register);
  - combinational logic that is ready to be evaluated or reset (the corresponding box turns grey when the logic is evaluated, and turns back to white when the logic is reset);
- the delays are ignored during simulation;
- with spread token semantics, a token can be spread across several adjacent registers, and the colours of the tokens are chosen to help distinguishing which registers contain copies of the same token.

Now analyse the performance of this pipeline with the **cycle analiser**  tool. In the *Tool controls* window one can see the result of the analysis: there is a single cycle with the estimated throughput 0.714. Indeed, the total delay of the pipeline is 1+1+1+3+1=7, and there are 5 tokens in it, so the throughput is 5/7≈0.714. Moreover, the bottleneck `MemAccess` is highlighted in red.

The performance of this pipeline can be improved using wagging. Select the bottleneck `MemAccess` and apply 2-way wagging to it: *Tools→Wagging→2-way wagging*. This transformation replicates the `MemAccess` block and adds some control structures (for the purposes of this practical, the gory details are ~~swept under the carpet~~ abstracted away for the sake of presentation). Now the tokens entering this stage of the pipeline alternate between choosing the upper and lower branch, and exit the stage in the same order they have entered it. This has an effect of halving the stage delay: Indeed, the *Cycle analiser* shows that the throughput has increased from 0.714 to 0.909:



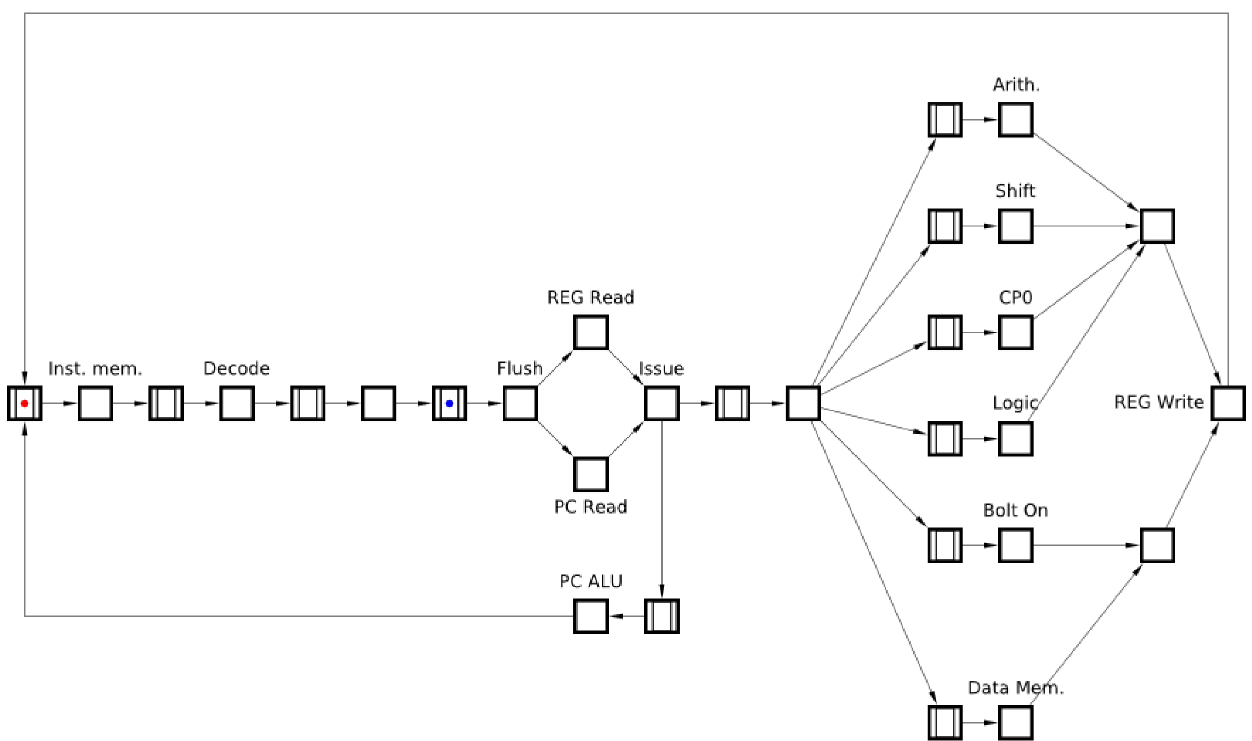
To further improve the performance of the pipeline, 3-way wagging can be applied instead of 2-way wagging: Indeed, the delay of `MemAccess` is three times the delay of the other stages. Undo the 2-way wagging by pressing `Ctrl+Z`, select `MemAccess`, and apply 3-way wagging to it: *Tools→Wagging→3-way wagging*. This increases the throughput to 1.

> Note that there are several cycles with infinite throughput - these are formed by the control structures which have 0 delays and can be ignored. Only the two cycles shown at the top are relevant - they correspond to the two branches formed due to wagging.

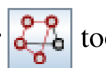
> By clicking on a cycle in the *Tool control* window one can highlight it in the model, with the bottleneck(s) of this cycle shown in red (if any). If no cycle is selected, the bottleneck(s) of the whole model are shown.

More realistic RISC pipeline

The following ARISC pipeline was adapted from [4]. All register delays were set to 0, and delays of most of the logic blocks were set to 1. The only exceptions are the `Arith.` and `Data Mem.` blocks with the delays 2 and 4, respectively.



Download this model: dfs-arisc.work (3.91 KiB, 4h ago).

Analyse this model using **cycle analiser**  tool.

By default the *Tool control* window shows 10 cycles with the worst throughput – this is controlled by the *Cycle count* field in this window. You can increase this number to see all the cycles in the model.

The throughput of the whole pipeline is determined by the throughput of its slowest cycle, e.g. the throughput of the above pipeline is 0.105.

Examine the cycles with the suboptimal throughput, and intelligently apply wagging to improve the performance of this pipeline. Try to increase the throughput from 0.105 to 0.125. Solution:

---

[1] J. Sparsø, S. Furber: *Principles of asynchronous design: a system perspective*, Kluwer Academic Publishers, 2001.
[2] C. Brej: *Wagging logic: implicit parallelism extraction using asynchronous methodologies.* Proc. Application of Concurrency to System Design (ACSD), pp. 35–44, 2010.
[3] D. Sokolov, I. Poliakov, A. Yakovlev: *"Analysis of static data flow structures"*, Fundamenta Informaticae, vol. 88(4), pp. 581–610, 2008.
[4] K. T. Christensen, P. Jensen, P. Korger, J. Sparsø: *The design of an asynchronous TinyRISC TR4101 microprocessor core*, Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, IEEE Computer Society Press, pp. 108–119, 1998.