Workcraft

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.

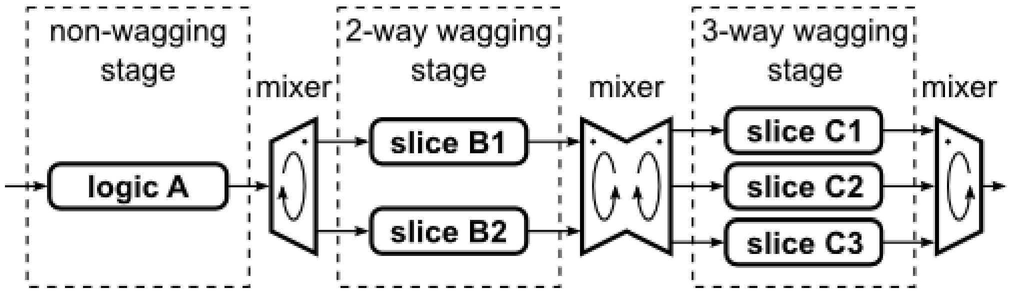
Static Data-Flow Structures

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:

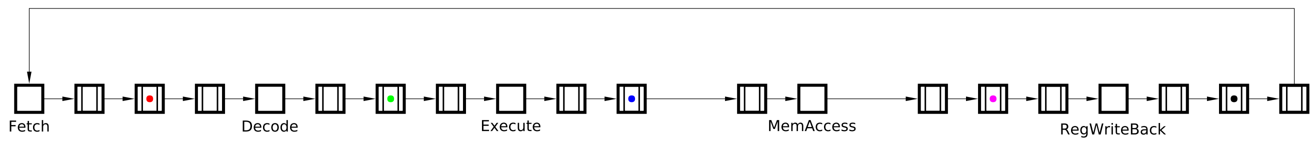
- logic has to be replicated, which incurs circuit area overhead;
- latency is increased due to mixers.

Each slice of a wagging stage should contain at least one register to accept an incoming token. For efficiency reasons it is advantageous to have a register as the first element of the slice – this enables the token to propagate into the active slice with the minimum delay and free the space for the next token.

.

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.

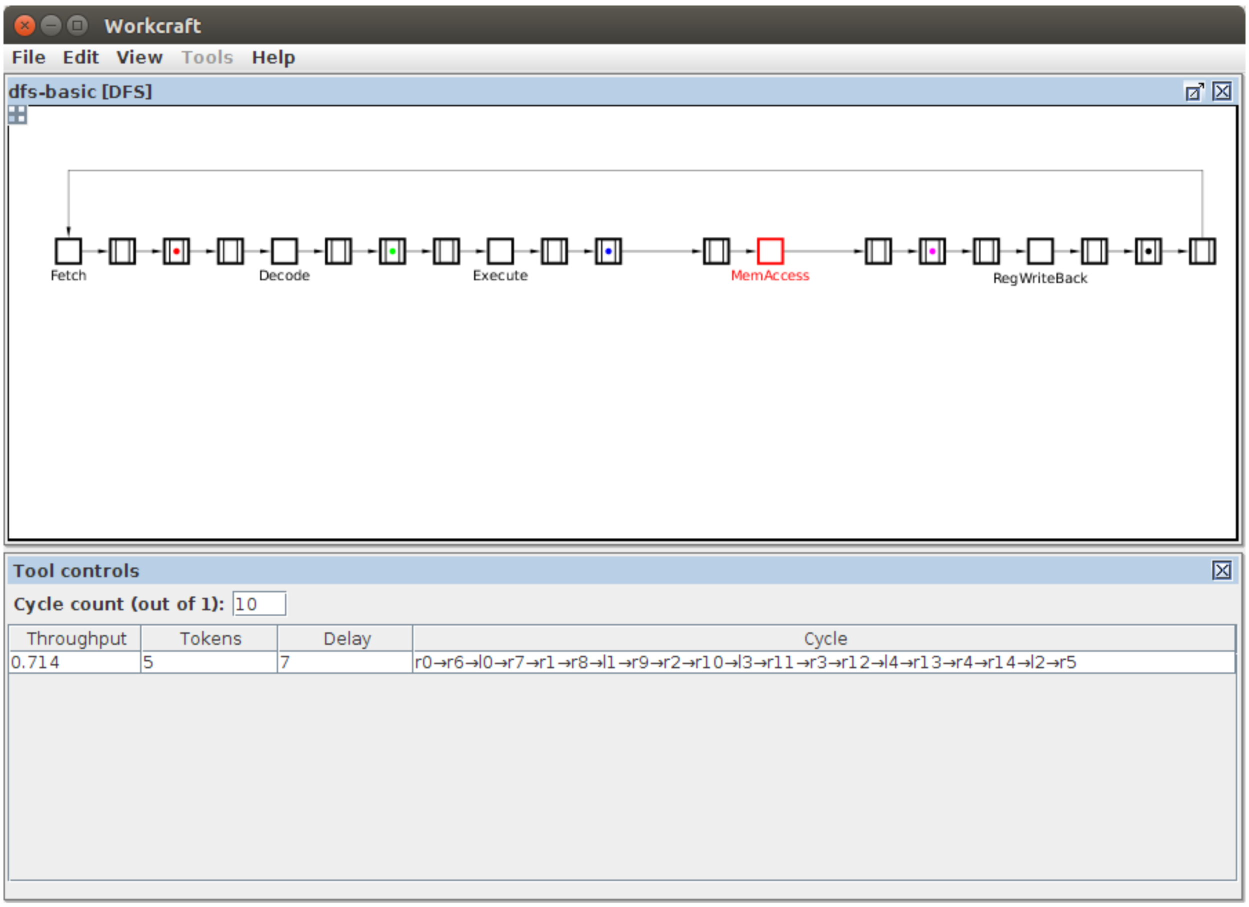


Re-create this *Dataflow Structure* model in Workcraft. For simplicity, set all the register delays to 0, and the delays of combinational logic blocks to 1, except for `MemAccess` whose delay should be set to 3, making it the bottleneck. Make sure that token colours are all different - see the *Token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

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 analyser** 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:

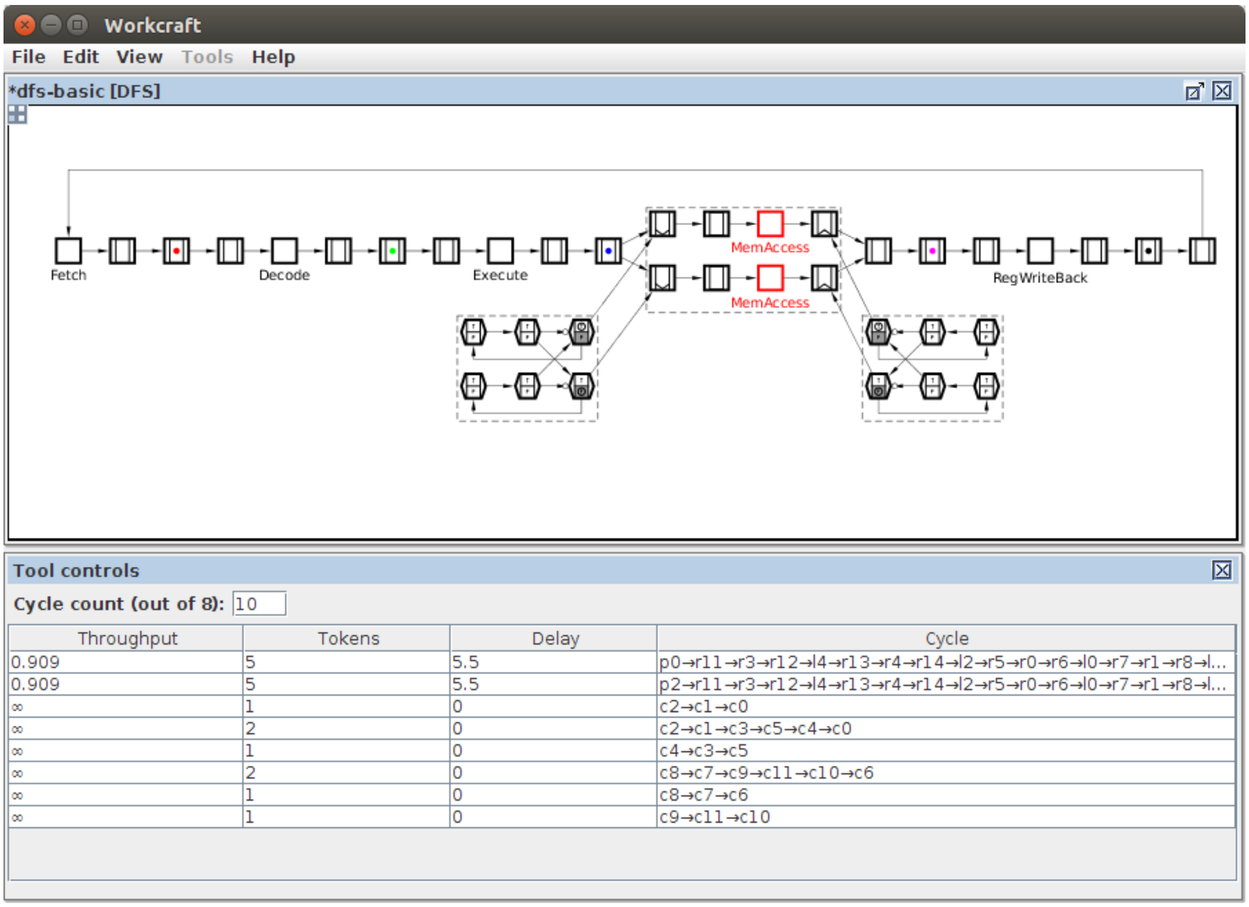


Cycle count (out of 1): 10

Throughput	Tokens	Delay	Cycle
0.714	5	7	r0→r6→l0→r7→r1→r8→l1→r9→r2→r10→l3→r11→r3→r12→l4→r13→r4→r14→l2→r5

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.

The performance of this pipeline can be improved using wagging. Select the bottleneck `MemAccess` together with preceding register (see the note about wagging efficiency) and apply 2-way wagging to it: *Tools→Wagging→2-way wagging*. This transformation replicates the selected section of pipeline 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 analyser* shows that the throughput has increased from 0.714 to 0.909:



Cycle count (out of 8): 10

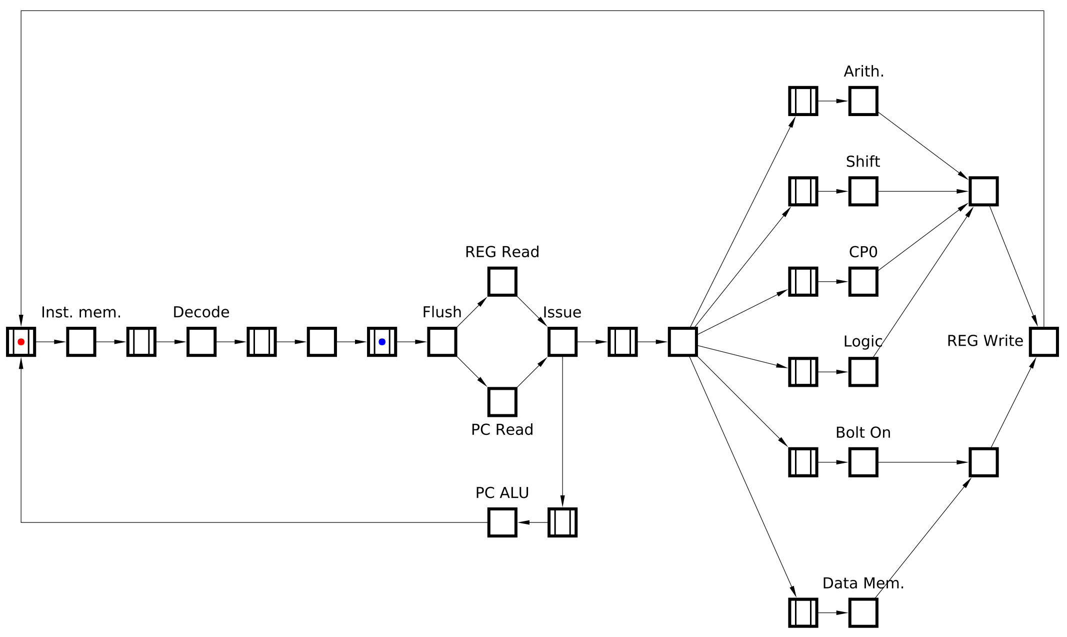
Throughput	Tokens	Delay	Cycle
0.909	5	5.5	p0→r11→r3→r12→l4→r13→r4→r14→l2→r5→r0→r6→l0→r7→r1→r8→l...
0.909	5	5.5	p2→r11→r3→r12→l4→r13→r4→r14→l2→r5→r0→r6→l0→r7→r1→r8→l...
∞	1	0	c2→c1→c0
∞	2	0	c2→c1→c3→c5→c4→c0
∞	1	0	c4→c3→c5
∞	2	0	c8→c7→c9→c11→c10→c6
∞	1	0	c8→c7→c6
∞	1	0	c9→c11→c10

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.

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 preceding register, and apply 3-way wagging to it: *Tools→Wagging→3-way wagging*. This increases the throughput to 1.

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 (3.95 KiB).

Analyse this model using **cycle analyser** 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 0.154.

Examine the cycles with the suboptimal throughput, and intelligently apply wagging to improve the performance of this pipeline. Try to increase the throughput to 0.2. 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.