

# Modelling of Superscalar Processor Architectures with Design/CPN

Frank Burns, Albert Koelmans and Alexandre Yakovlev  
Department of Computing Science  
University of Newcastle upon Tyne, NE1 7RU, UK

## Abstract

We describe aspects of modelling a generic superscalar processor architecture using Coloured Petri nets, for the purpose of analysis of its real-time properties, such as Worst Case Execution Time for a block of instructions. The model can be simulated within the Design/CPN environment. The results of the simulation are displayed using a custom graphics tool written in Tcl/Tk.

## 1 Introduction

Design of real-time systems, which must respond to external events within strict time bounds, involves a mapping between the logical and physical levels of a system specification. This mapping determines the quality of the timing information available at each level of abstraction. A crucial issue, which introduces a significant element of temporal uncertainty, is the way in which the system design is tied together with the use of specific hardware components. Even though some of these components can be standard, off-the-shelf ones, their timing characteristics can be very imprecise for the purpose of acquiring reliable, yet not too pessimistic data to be used at higher levels (e.g., worst execution times of program code to be used for task scheduling). Moreover, some hardware is designed in a special way (e.g., interface circuits), which is increasingly the case for embedded applications. In such situations, realistic timing information about the hardware becomes available only during software development, which creates an additional adequacy problem.

This problem arises increasingly due to the following three major factors:

- New processor architectures are becoming ever more complex. They include many stages of pipelining, out-of-order execution, parallel execution of several instructions, and multi-level caching mechanisms;
- System architectures often involve non-standard components whose temporal parameters are not known in advance;
- Future systems will tend to become more asynchronous. Even if their core elements are clocked, the overall system will either be multi-clocked or almost entirely self-timed.

We are developing a methodology and an associated set of software tools for the modelling and analysis of timing specifications of hardware platforms, in particular asynchronous RISC processors, based on the use of Coloured Petri Nets (CPNs) [3]. Previous work in this area (e.g. [6]) has mainly focused on the use of P/T nets. The ability to model asynchronous interaction inherent in Petri Nets would enable us to take into account some fine behavioural issues like the effect of potential 'request-acknowledgement' handshakes in the interfaces between units, and dynamic allocation of instructions on different units. In this paper we present a detailed model of a generic superscalar RISC processor developed using the Design/CPN tool.

## 2 Why ordinary Petri nets are insufficient

Petri nets have traditionally been used for modelling and analysis of digital systems. Processor architectures are no exception [2]. In addition to classical "qualitative" properties for verification, like deadlocks and boundedness, the real-time aspect requires ways of obtaining more detailed, "quantitative", analysis, such as worst-case execution time for a block of instructions. The former type of modelling need only capture fairly general properties (possibilities) of the system behaviour (e.g., the processor execution pipe cannot overflow with instructions). The latter is certainly more specific with regards to the actual paths taken by the modelled system under a given set of data. Hence the "quantitative" analysis often lends itself to simulation, performed in addition to the exploration of possibilities.

Use of ordinary Place-Transition nets, which can describe the control flow quite comfortably and provide efficient ways of "qualitative" verification, cannot themselves (without additional annotation) represent the effect of data, associated with the model states, on the execution of actions in the

processor. Note that in modelling processor architectures, the role of data path is played by instruction flow. Each instruction is a complex (again, depending on the level of detail we want to represent) data object. This object consists of attributes reflecting not only its type and operands, but also important (in the superscalar case) information about dependencies, targets, branching etc. Since all these aspects affect the temporal profile of the instruction in the overall instruction flow, the model should be able to represent them as accurately as possible. Coloured Petri nets appear to be capable of providing the processor model with appropriate mechanisms for data typing. These mechanisms can concretise both the notion of a state (token marking) and the data-dependent conditions of action execution.

Before we proceed to a detailed model of a processor, let us consider a simplified control flow model of an asynchronous processor, described with a Place-Transition net as shown in Figure 1. We believe that this figure is self-explanatory. More details about the use of Place-Transition nets in the design of a processor can be found in [4].

At the highest abstraction level, the behaviour of a processor consists of two actions, Instruction Fetching (IF) and Instruction Execution (IE), which alternate and are therefore performed sequentially.

These actions can be refined into subactions according to our knowledge about the processor architecture. Thus, the IF action can be seen as a process, i.e. a Petri net fragment, consisting of the following subactions: incrementing a Program Counter (PC), loading a Memory Address Register with the new address for memory reading (MAR<sub>r</sub>), and reading the new instruction word from Memory (Mem). The IE action can be refined into a process (another Petri net fragment) involving other subactions: loading an Instruction Register (IR), decoding, activating and executing the fetched instruction for two possible instruction formats, a one word instruction (1WdInst and 1WdEx) and a two word instruction (2WdInst and 2WdEx). The part of the process concerned with two word instruction execution requires two memory cycles. As can be observed from the analysis of this Petri net, the initial sequential operation between IF and IE has been refined into a model which allows concurrency between actions with smaller granularity. For example, the PC action can be executed concurrently with instruction reading, decoding and execution. Another paradigm appearing at this level is that of choice between two types of instruction execution. The refined model can be subjected to verification (e.g. for absence of deadlocks or undesirable conflicts between actions) and/or performance analysis (e.g., estimation of the degree of concurrency between transitions, evaluating crit-

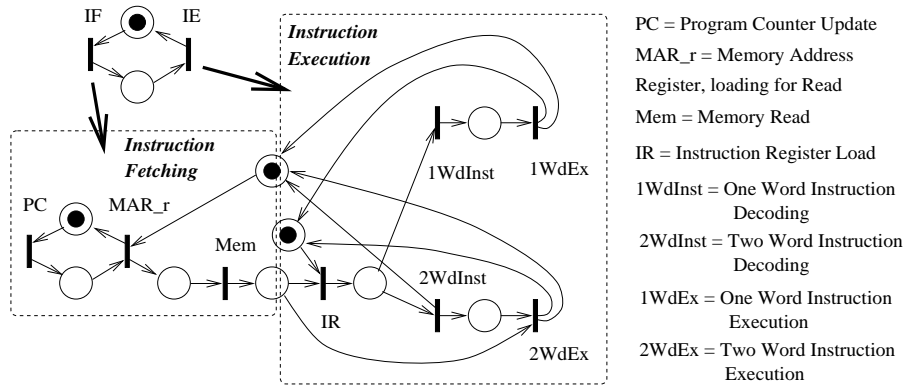


Figure 1: Place-Transition net model of an asynchronous processor.

ical paths, simulation). The process of refining the design can be continued until the designer realises that the abstract behavioural model satisfies the desired functional and quantitative requirements. The result of this design stage is a specification of the control flow in such a form that its actions, i.e. transitions in the labelled Petri net model, can be easily mapped onto the primitive operations of the datapath units. This part of the design process is described in detail in [4].

### 3 Basics of processor modelling

We model instruction types by first defining a set of predefined identifiers using an enumerated colorset as follows:

**Color Instr = with INT | FPADD | MUL | DIV | BRA | NOOP;**

It is then possible to create a record colorset using appropriate fields to model an instruction completely:

```
color Value = record
    no: Line *
    instr: Instr *
    d: Dep *
    d': Dep *
    t: Target timed;
```

assuming that the instruction has dependencies **d**, **d'** and a destination **t**, which links those instructions sharing the same target register. It must be

noted that the model we are trying to develop here is primarily of timing not hardware.

To model processor timing we need to define updates to Program Counter (PC) values at various points in our model. In Design/CPN variables, guards and arc inscriptions can be used in order to do this. CPN variables can be defined for any predefined type or colour. For example, we can define a variable `fetch` of type **Value** to represent a fetched instruction:

```
var fetch : Value;
```

Once variables are defined it is possible to model the occurrence of updates to processor values by referencing the values of tokens arriving at the input places of transitions. Guards provide predicates to check the values of tokens at input places to determine that the correct values are present for enablement. For example, at the **FETCH** stage we can determine the next instruction selection by defining a fetch transition guard which checks the instruction number against the pc count: `[#no fetch = pc]` Here the selector `#no` checks the number of the instruction fetched against the value of the program counter.

An arc inscription can then be defined to update the PC value to the next instruction number ready for the next selection. This is modelled in Design/CPN by using the following multiset arc expression which adds one to the program counter variable.

```
1'(pc+1)
```

The main graphical part of the CPN model of a superscalar processor is shown in Figure 2. In the following sections we discuss individual aspects of this model.

## 4 Pipelining and In-Order Issues

The basic processor pipeline can be modelled in Design/CPN by using a set of transitions for each pipestage **FETCH**, **DECODE**, **EXECUTE** etc. The **EXECUTE** stage is further subdivided into transitions to represent execution units **INT**, **FPADD**, **MUL**, **DIV**. Individual execution units are further pipelined by using a hierarchy of places over transitions. Places are used between transitions for storage, queues etc.

Instruction flow through functional units needs to be controlled and we need to cope with hazards in processors when they arise. In order to reference processor values at various pipestages in Design/CPN we define instruction variables for the inputs and outputs leading to and from each pipestage

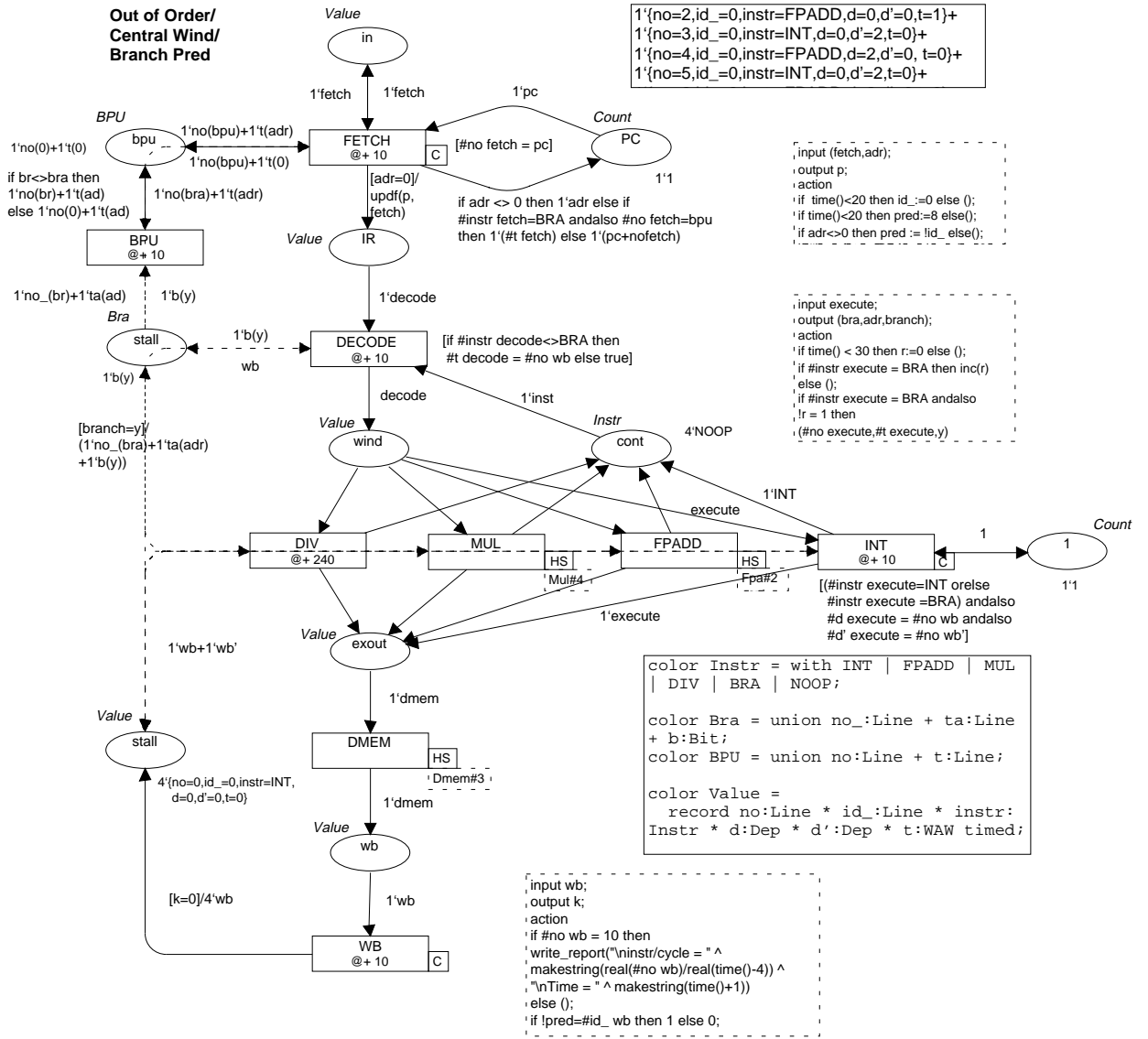


Figure 2: Design/CPN model of a superscalar processor.

and execution unit transitions:

```
var fetch, decode, execute, wb, commit : Value;
```

Guards are used at execution unit transitions to select correct instructions for execution. For example, the following guard filters an instruction of type **MUL** at the multiplier unit transition:

```
[#instr execute = MUL]
```

This models the correct instruction flow through execution units, but hazards also need to be modelled. Different types of hazards can be split into dependency and structural hazards. Dependency stalls are modelled by guards which check dependencies against instructions that have been written back. For example an RAW (read after write) or true dependency can be checked at a decode transition using the following guard:

```
[#d decode = #no wb]
```

This checks that the number of an instruction at writeback satisfies the dependency **d** of a decoded instruction waiting to issue.

Similarly we can check for WAW (write after write) or output dependencies by using the following guard at the **DECODE** stage where **t** is defined as having the same target register as a previous instruction which has already or which is expected to be written back to the same register:

```
[#t decode = #no wb]
```

Note that we use **t** here as an instruction number and not as a register number.

Additionally, we need to be able to cope with structural hazards. Structural hazards occur when some combination of instructions cannot be accommodated because of resource conflicts. These need to be detected and issuing instructions need to be stalled. Structural stalls can be modelled in Petri-nets by using feedback from functional units to indicate that a functional unit is free.

## 5 Superscalar issues

### 5.1 Multiple issue

Superscalar machines depend on the ability to execute multiple instructions in parallel. This is known as *Instruction Level Parallelism* [1]. Multiple issue exploits instruction level parallelism by fetching and decoding more than one instruction at a time.

The use of a fixed length instruction set enhances parallelism. For our multiple issue model we have decided to fetch four fixed length instructions

at a time. In order to do this the select Guard at **FETCH** needs to be changed to accommodate this. Now we have to fetch instructions as a block (see Subsection 8) To do this we fetch a block of four instructions defined as a record:

```

1{blockno=1,
    b1 = {no=1,instr=INT,d=0,d'=0,t=0},
    b2 = {no=2,instr=INT,d=1,d'=0,t=0},
    b3 = {no=3,instr=FPADD,d=0,d'=2,t=0},
    b4 = {no=4,instr=FPADD,d=0,d'=0,t=0}
}

```

The PC count must also be updated accordingly:

```

1(pc+nofetch)

```

Instructions are fetched into a queue (dispatch stack). We assume a queue of length eight split into upper and lower queues each being capable of holding a block of four instructions.

```

var bl : block;

```

Instructions from the lower queue are considered for decode and are replaced by the top half when all instructions from the lower queue have been decoded. In Petri nets two places separated by a transition can be used to model the two queue halves:

```

var upper, lower: block;

```

All instructions must be decoded and issued before another set of instructions is fetched. This is achieved by having a corresponding buffer size control node with four tokens representing the lower queue size. An up/down counter must be used to service incoming requests in order to determine the number of instructions that can be issued.

We can implement the instruction type checking and the dependencies and also the issue order provided the instructions are modelled at decode using an addition multiset:

```

1'I1 + 1'I2 + ..;

```

When the instructions are modelled like this we can relate the instructions to one another and compare them:

```

[#t I1 <> #t I2]

```

This is not simply modelled in Design/CPN because the number of instructions issued may vary. In Design/CPN it is difficult to recognise dynamically a variable number of buffer control size tokens and control a specific number of instructions waiting to be issued. Because of this an up/down counter must be used to determine how many instructions are left in the

issue window before a fixed number of requests can be serviced. This aspect is now being investigated.

## 5.2 Out of order issue and execution

So far we have assumed a model of in order issue. This section covers out of order issue and out of order execution. To issue out of order implies using a buffer(s) or window(s) in which to store instructions waiting to execute. The buffer, called an instruction window, is placed between the instruction decoder and the functional units.

There are two ways to implement the instruction window. The first is to centralize the window. We can model this in Petri-nets by using a place after **DECODE** to mimic a central window where collected tokens in the central window represent unissued instructions. To control its size we create a corresponding buffer control size node of colour **Instr** and initialise it with size tokens or instructions.

**Color Instr = with INT | FPADD | MUL | DIV | NOOP;**  
**size'NOOP;**

The buffer control size node is connected to the decode transition by an arc. Each time an instruction token is issued from decode to the central window a corresponding token is removed from the buffer size control node. When size tokens are removed from the buffer size control node the central window reaches its full capacity and no more instructions can be decoded.

Another way of implementing the instruction window is to distribute individual buffers called reservation stations to each of the functional units, buffering instructions destined for a particular functional unit at the input of that functional unit. Multiple windows or reservation stations can be modelled in Petri nets by using a number of buffer places, one for each reservation station. In this case we need to control the numbers of each instruction type entering each buffer place.

To control the size of the reservation stations we make use of the corresponding size control node of colour **Instr** as for the central window but this time initialise it with a multiset which corresponds to the individual sizes of each reservation station:

(1) **2'INT + 2'FPADD + 2'MUL + 2'DIV**

Every time an **INT** instruction is issued from the decode stage a Guard is used to remove the corresponding type of instruction from the buffer size control node.

An arc inscription selector is used from decode to each reservation station

place to control the instruction selected. For example, for the arc leading from the decode transition to the reservation station for the **INT** unit we have the following arc inscription:

**[#instr decode = INT]/1'decode**

If the instruction of type **INT** is taken from 1) the corresponding multiset will result in the buffer size control node:

(2) **1'INT + 2'FPADD + 2'MUL + 2'DIV**

When two **INT** tokens are removed from the buffer size control node this is equivalent to the reservation station at the **INT** execution unit being full.

## 6 Branch prediction

Branch prediction can reduce the average branch delay by predicting the outcomes of branches during instruction fetching. Branch prediction makes use of a Branch Prediction Unit (BPU) for building up a database of speculative branch information.

Branch instructions are represented in colour using the following format:

**color Value = record**

**no: Line \***  
**BRA: Instr \***  
**d: Dep \***  
**l: loop \***  
**t: Target timed;**

where **d** represents a dependency, **l** indicates whether the instruction will loop and the number of times for the loop is predetermined and **t** represents the correct target of the branch instruction which is also predetermined by the user.

For Branch Prediction we assume a branch prediction unit using a target buffer which dynamically collects information about the most recently executed branches. The data structure for the BPU is defined as follows:

**color BPU = union no:Line + t:Line;**

**FETCH** checks the instruction against the target buffer to see if their is a predetermined branch for that branch **no:Line**. If the fetched instruction is a branch the branch target buffer indicates the predicted outcome using the target address **t:Line**.

In the case of a misprediction two values are sent back to the BPU, a branch instruction number defined as the first part of the union: **no\_(bra)**, and the correct target defined as the second part of the union: **ta(adr)**. This is achieved with the following arc inscription flowing from the corresponding execution unit to the BPU :

```
[branch=y]/(1'no_(bra)+1'ta(adr)+1'b(y))
```

This transfers a number for the branch instruction **no\_(bra)** and a corresponding BPU update prediction target address **ta(adr)**. The target for a particular address is checked for by the **FETCH** with the following arc inscription:

```
if adr <> 0
then 1'adr
else if #instr fetch = BRA andalso #no fetch = bpu
    then 1'(#t fetch)
    else 1'(pc+nofetch)
```

## 7 Recovery

Recovery is a process which cancels the effects of instructions that were issued under false assumptions using speculative execution. In asynchronous circuits, recovery from an incorrect branch is accomplished by assigning a tag or colour to each instruction (see e.g., [5]). When a branch is encountered the tag or colour is changed and instructions which do not have the current tag are considered invalid and are terminated further down the pipeline. For speculation it is assumed that a tag needs to cover a range of values between branches and an integer tag is proposed so that each instruction between a branch is represented by a unique **id\_**. An additional record value is provided for this in the instruction definition:

```
color Value = record
    no: Line *
    id_: Count *
    instr: Instr *
    d: Dep *
    d': Dep *
    t: Target timed;
```

The value of the `id_` is changed for each instruction by incrementing it modulo some limit over which the range of tags are defined. Each time a branch instruction is fetched the value of `id_` must be changed and a list created of all the invalid `id_` values built up from an incorrect speculation.

To make the recovery process more efficient instructions could be checked and eliminated earlier than the writeback stage to speed up recovery. This is done by using a selector at certain stages to determine if the instruction is valid or invalid:

```
[instructionvalid=0]/1'execute
```

## 8 Caches

While modelling caches we assume that direct cache modelling is being used. In Design/CPN an instruction cache can be modelled by having an extra **CACHE** place between the instruction memory and the processor. An extra **READ** transition is also inserted after the cache place which models a read access made by the processor to the cache for an instruction. The original **FETCH** now becomes a fetch access to memory to load a block into the **CACHE** place.

When modelling caches we need to be able to reference values in the cache. In order to do this tags need to be added to the instructions to model their relative cache positions. We have done this by defining a new record which defines a block of instructions and adding extra fields called `cachetag` and `cacheblock`:

```
color Block = record ct:Line * cb:Line * b1:Value * b2:Value
b3:Value * b4:Value;
```

```
{ct=0,cb=0,
  b1={no=0,id_=0,instr=NOOP,d=0,d'=0,t=0},
  b2={no=0,id_=0,instr=NOOP,d=0,d'=0,t=0},
  b3={no=0,id_=0,instr=NOOP,d=0,d'=0,t=0},
  b4={no=0,id_=0,instr=NOOP,d=0,d'=0,t=0}
}
```

The `cachetag` is used to specify the set of memory blocks which can be mapped to a specific `cacheblock`. The `cachetag` value is calculated as the upper part of the pc address. The PC address is formatted as follows:

```
pcaddr = cachetagbits;cacheaddrbits;offsetbits
```

Here we assume instructions have a fixed length (32 bits). The cachetag for an instruction block is calculated by dividing its instruction address by  $2^{\text{cacheaddrbits}+\text{offsetbits}}$  where **cacheaddrbits** are used to specify the cacheblock address, and **offsetbits** are used to specify the offset of an instruction within a cacheblock.

Initially all the instructions in the cache are set to **NOOP** and their cachetags set to zero, **cachetag=0**.

To access a cache address at READ we select the cacheblock address by using a transition guard:

$$[\#\text{cacheblockread} = ((\text{pcaddrnoDIV}2^{\text{offsetbits}})\text{mod}2^{\text{cacheaddrbits}+\text{offsetbits}})]$$

## 9 Preliminary simulation results

Design/CPN simulation output consists of a large text file. In order to display the timing results in a more suitable format, a graphical tool, written in Tcl/Tk, was written to display the timing results of the simulation in a more suitable format. A software filter processes the output from Design/CPN and drives the Tcl/Tk display.

Figure 3 shows the schedule obtained for a set of instructions which have been simulated using the model shown in Figure 2. In the schedule, the second instruction has been delayed from executing by a Write-After-Write dependency on the first instruction. The second instruction then proceeds to execute, but the subsequent instructions which have a Read-After-Write dependency on the second instruction are delayed, and accumulate in the central window before the dependency on the second instruction has been resolved. Subsequently, instructions are shifted out of the central window and execute in parallel. This allows instructions 6,4 and 8 to enter the FPADD pipestage (4 stages) and execute in parallel. It can be seen from the schedule that instructions execute out of order and are written back out of order.

## 10 Discussion

The use of Design/CPN on a complex processor model shows that the tool has many obvious strengths but also weaknesses.

The strength of Design/CPN modelling lies in its use of colours and datatypes which can be used for modelling basic processor instructions and also more complex memory types such as caches. Because of its typing,

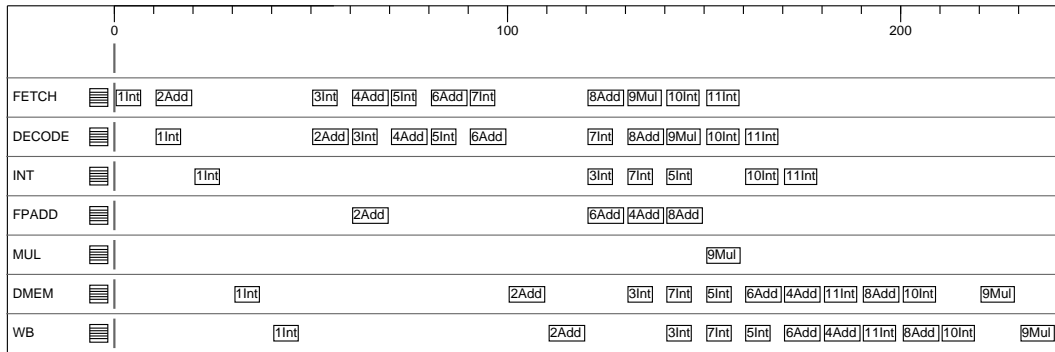


Figure 3: Instruction timing diagram.

Design/CPN can be used to capture dataflow adequately using arc and guard expressions.

Pipelines are easily captured using a sequence of transitions and places. Hierarchy is expressed easily by using pages over transitions.

Dependency hazards can be easily modelled by placing guards at execution units. Structural hazards are easily modelled using feedback. Register locking can also be modelled.

In order and out of order issue modelling is easily captured. The natural flow in Design/CPN is out of order or arbitrary but this can be changed to in order by using Guards. Design/CPN is also good at modelling buffers, and therefore reservation stations can be easily modelled.

Multiple issue can be modelled but not without difficulty. This is because it is difficult to express dynamically the relationship between token flow expressions on different arcs. Compensatory techniques have to be found to overcome this such as counters to recognise how many tokens can be allowed to pass over arcs.

Branch prediction can be modelled. However, there is a possibility of arbitration occurring between the fetch unit and the branch prediction unit. Arbitration is not easily simulated, as it has no priority of choice and cannot make a decision about the correct flow of data. Although the simulator tends to correct this, it is not an adequate solution at the moment. This requires further investigation.

Recovery is difficult to model, but it is naturally difficult to model in

asynchronous designs. The techniques to overcome this are based on assigning tags to instructions, and terminating instructions when they become invalid. The problem with this is that more than one tag needs to be used for out of order issue.

## 11 Conclusions

We have described the main aspects of our approach to the modelling of a superscalar processor with Coloured Petri nets, and some preliminary results. The model is successfully run using the Design/CPN software. Typical speed of simulation is about 3000 instructions per minute under Linux on a 166Mz PC. Design/CPN provides the designer of a real-time system with both qualitative analysis of reachable states and analysis of timing characteristics, such as worst case execution for a block of instructions. A number of modelling issues has been revealed that require further investigation of the descriptive power of Design/CPN. Further tool development should allow extraction of timing parameters from Design/CPN output files, and their display using graphical tools not available within the Design/CPN environment.

## 12 Acknowledgements

This work is supported by EPSRC grant GR/L28098 (project TIMBRE) and Esprit LTR Project 20072(DeVa). The authors also wish to thank the referees for helpful comments on the paper.

## References

- [1] D.K. Arvind and V.E.F. Rebello. Instruction-level parallelism in asynchronous processor architectures. Proceedings of the Third Int. Workshop on Algorithms and Parallel VLSI Architectures, M. Moonen and F. Catthoor (Eds), Leuven, Belgium, August 1994, Elsevier Science Publishers, pp. 203–215.
- [2] J.B. Dennis. Modular, Asynchronous Control Structures for a High Performance Processor. Proceedings of Project MAC Conference on Concurrent Systems and Parallel Computation, June 1970, pp. 55-92.

- [3] K. Jensen. Coloured Petri Nets. Basic concepts, analysis methods and practical use. EATCS Monographs on Theoretical Computer Science, Springer-Verlag 1992.
- [4] A. Semenov, A.M. Koelmans, L. Lloyd, and A. Yakovlev. Designing an asynchronous processor using Petri nets. *IEEE Micro*, 17(2):54–64, March 1997.
- [5] N.C. Paver. The design and implementation of an asynchronous micro-processor. Ph.D. Thesis, University of Manchester, 1994.
- [6] R.R. Razouk. The Use of Petri Nets for Modeling Pipelined Processors. Technical Report 87–29, University of California, Department of Information and Computer Science, 1987.