Asynchronous Microprocessors: From High Level Model to FPGA Implementation

L. Lloyd, K. Heron, A. M. Koelmans, A. V. Yakovlev

Department of Computing Science,

University of Newcastle upon Tyne, NE1 7RU, England.

Abstract

In order to determine the applicability of both programmable software tools and programmable hardware for asynchronous logic applications an implementation, employing FPGA devices, of the instruction decode and the instruction execution stages of an asynchronous microprocessor, the ADLX, is presented. The foundation for that microprocessor is based on the employment of event driven logic, specifically 2-phase transition signalling, that functions within the conceptual framework of a Sutherland micropipeline.

The entire design has been constructed from a series of VHDL descriptions that have been compiled and simulated using both the Cypress **WARP** VHDL Development System and the AMD **MACHXL** software packages. A number of the asynchronous specific areas of the ADLX have been synthesized using Petrify, a Petri Net tool designed for the manipulation of concurrent specifications of asynchronous control circuits. The ADLX itself has been constructed from a range of "off-the-shelf" products including HM 65764 high speed CMOS SRAM semiconductors and FPGA logic devices.

Keywords: Asynchronous Logic, Petri Nets, Field Programmable Gate Arrays.

1   Introduction

Almost all digital logic produced today is based upon the synchronous design approach. In such a framework a system is a collection of clocked finite state machines which themselves are controlled by a master clock. These synchronous systems require specific interface protocols whenever a signal crosses from the domain of one particular clock to the domain of another clock. These systems behave in a discrete and deterministic manner.

Asynchronous logic is significantly different. There is no clock that is used to regulate the timings of state changes. All subsystems are allowed to exchange information at mutually negotiated instants whilst not being bounded by any external timing regime. This removal of the clock allows an asynchronous design to be naturally conservative with regards to power

consumption. As the inherent nature of asynchronous logic is to be data driven, only those areas of a circuit that are actually performing useful work will be consuming power. The fewer the number of transistors that are active in a device, the less current will be consumed.

The only asynchronous versions of commercial processors currently available are the Amulet series [2], which are based upon Sutherlands micropipeline architecture [19]. The Amulet designs were produced without the help of formal asynchronous design tools, which required a lot of effort. Often, asynchronous CAD tools tend to be "bolt-ons" to more sophisticated synchronous design packages [18] or tend to be "experimental" in aspect and thus have not been so widely disseminated throughout the design community. Of the currently available asynchronous CAD tools, the Tangram language compiler [21] is best known for being able to produce circuits that are equivalent to or better than their synchronous counterparts [20].

Petri Nets [12] are a well established mechanism for systems modelling. They are mathematically sound and can be subjected to a large variety of analysis methods, for example, to test for the presence of deadlocks. Since Petri Nets are essentially an event driven formalism they are ideal for modelling asynchronous hardware, which is also event driven. Petri Net descriptions must be translated into logic equations by means of an appropriate software tool since the standard techniques for designing synchronous logic are not valid. In order to generate asynchronous hardware from Petri Net descriptions, a state of the art software tool, Petrify [1], can be used.

A methodology for the high level design of processor architectures using Petri Nets was described in [16]. This methodology allows the development of Petri Net models of the main stages of any type of asynchronous processor that functions with a Sutherland micropipeline framework. This allows a designer to refine the model according to the results of the behavioural analysis of the nets. The analysis can be performed using existing methods for state graph and partial order traversal. The properties obtained include freedom from deadlocks, concurrency and conflict relations between individual actions and performance characteristics.

After the design has been developed and validated it becomes possible, through the use of VHDL descriptions, to employ field-programmable gate array logic that will allow a custom device to be rapidly prototyped and tested. As any asynchronous circuit can be represented by a combination of one or more of the above methods, it therefore becomes possible to develop FPGA versions of equivalent synchronous digital devices that can then be used for comparison purposes. An FPGA version of a digital circuit is always likely to be slower than an equivalent ASIC version, due to the fact that the FPGA will very likely be constructed from a series of logic blocks interconnected via regularly structured wiring channels rather than custom-built logic, but this does not mean that meaningful comparisons cannot be made.

We present the design of a microprocessor that utilises the framework of asynchronous logic in the design of a simple micropipelined microprocessing unit. This microprocessor, called ADLX (Asynchronous Deluxe), is essentially the asynchronous version of Henessey and Patterson's well known DLX RISC processor design [6]. Unlike [16], which concentrated on the high level aspects of the design, this paper presents the implementation aspects. This was accomplished using commonly available FPGA logic devices.

The paper is organised as follows. In section 2 the issues relating to synchronous and asynchronous FPGA design will be discussed and in section 3 the modelling approach of the ADLX using Petri Nets will be presented. In section 4 we examine the main datapath pipeline of the ADLX together with a description of the signalling mechanism implemented in the design. Section 5 details the synthesis of the asynchronous specific areas of the datapath as carried out by performing a Petri Net analysis whilst section 6 will be concerned with describing the CAD tools used and simulation results achieved. The implementation of the ADLX is covered in section 7 with testing issues described in section 8. Section 9 concludes the paper.

2   FPGA design issues for Asynchronous logic

There is no denying that FPGA's are an extremely effective means of performing fast development and test of digital circuits. The employment of large amounts of simple logic gates and datapaths that can be rapidly programmed and reprogrammed until a desired solution has been found is a very cost effective method of hardware design.

The disadvantage in using FPGA's for asynchronous logic development is concerned with the physical architecture of such devices. Asynchronous circuits have to be able to deal with synchronisation issues, hazards and arbitration concerns and currently it is still not possible to address these problems efficiently in current FPGA architectures that have been constructed with the implementation of strictly synchronous designs in mind [4].

These restrictions though do not mean that asynchronous FPGA's cannot be developed. Whilst it is true that the software mapping, routing and placement algorithms may reintroduce hazards into a design, for example isochronic forking problems in quasi-delay insensitive circuits [5], more robust asynchronous designs, especially in the case of bounded-delay circuits, can be constructed.

With regards to timing issues it has already been stated that FPGA's are geared towards synchronous clocked implementations of circuit designs. There is a discipline though, "self-timed design", that allows a signal exchange to be carried out in a "handshaking" manner and which functions by allowing actions to be decided upon the edge of a signal transition which may have an indeterminate, but finite, periodicity between such events. This process of signal exchange fits in very well with the bounded-delay domain of asynchronous logic. By using a technology-independent method a designer can model at a high level, for example using a hardware description language, asynchronous circuits that can be directly targeted to a specific hardware device. There are problems associated with this procedure in that there may be a loss of efficiency in such a translation from software to hardware but these issues can and have been overcome [11] [15]. Self-timed implementations of asynchronous circuits tend also to be expensive in terms of logic cells and routing resources required but efficient self-timed micropipelines have been developed [13].

3   Petri Net modelling of ADLX

DLX, pronounced "Deluxe", was a simple microprocessor designed to use a load/store architecture and is described as "the average of a number of recent experimental and commercial machines" [6]. The overall design was based upon observations of the most commonly occurring primitives in programs, thus allowing an efficient pipeline and an easily decoded instruction set to be developed. Over the years, DLX has provided such a good architectural model for both study and design evaluation that recent reworking has resulted in the development of a superscalar version [7].

The fact that DLX has an architecture that is very simple to understand has made it an excellent model to use as the basis for the equivalent design of an asynchronous microprocessor. This asynchronous processor, called ADLX, was first modelled and verified at a high level in [9] using Petri Nets. Since Petri Nets are an ideal means for the modelling and simulation of concurrent systems, they can be effectively employed in the development of VLSI circuits. As such, Petri Nets have recently been used in the design of a number of synchronous and asynchronous microprocessor projects [16, 23].

A brief description of the semantics of Petri Nets is as follows. Any Petri Net is a tuple (representing a graph) such that $P = (S, T, F, M_0)$ where $S =$ the set of vertices that represents the state components of the graph, $T =$ the set of transitions (or actions) that can be performed, $F =$ the flow relation, defined as $F \subseteq (S \times T) \cup (T \times S)$ (both S and T are finite disjoint sets) and $M_0 =$ the initial state marking of the graph, defined as $M_0 \subseteq S$. Tokens flow round a net, representing events. In circuit terms, they indicate signal changes. For a fuller description of the semantics of Petri Nets see [12]. A thorough discussion of hardware synthesis from Petri nets falls outside the scope of this paper; the interested reader is referred to [1].

The design methodology employed during the development of ADLX was as follows. The main ADLX processor design was first modelled at a high level within a micropipeline framework, with decomposition taking place by creating refined Petri Net models of each processing stage. A basic model was developed for all possible instructions that could be executed in that stage. These individual *instruction execution* Petri Nets could then be combined into one single *processing stage* Petri Net which could then be translated by the Petrify software tool into a logic diagram that could be used for subsequent logical decomposition and synthesis. In this paper we are only concerned, for the sake of brevity, with the implementation of the Instruction Decode and Execution stages of ADLX. We aim to show how the top-level abstract view was of those stages was decomposed. The top-level view of ADLX is shown in Figure 1.

The decomposition of the Instruction Decode unit as relating to the execution of an R-type (ALU operation) instruction is shown in Figure 2. This shows how an instruction can be decoded into a number of address or opcodes that are employed in either the ID stage or further down the ADLX pipeline. The *rs1* and *rs2* addresses are used to forward a 32-bit data value to the A and B multiplexor units respectively. The *rd* address is forwarded into a 3-stage FIFO pipeline which is ultimately used for the write back of a data value to the register file.
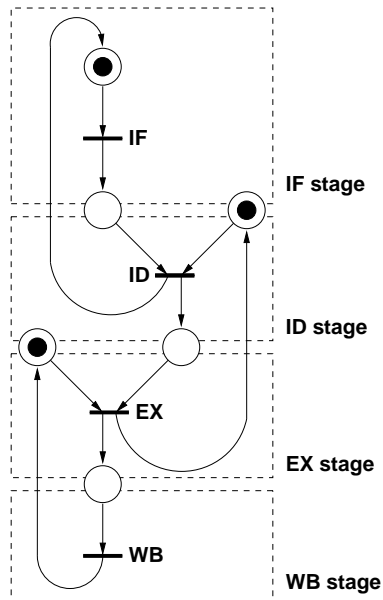
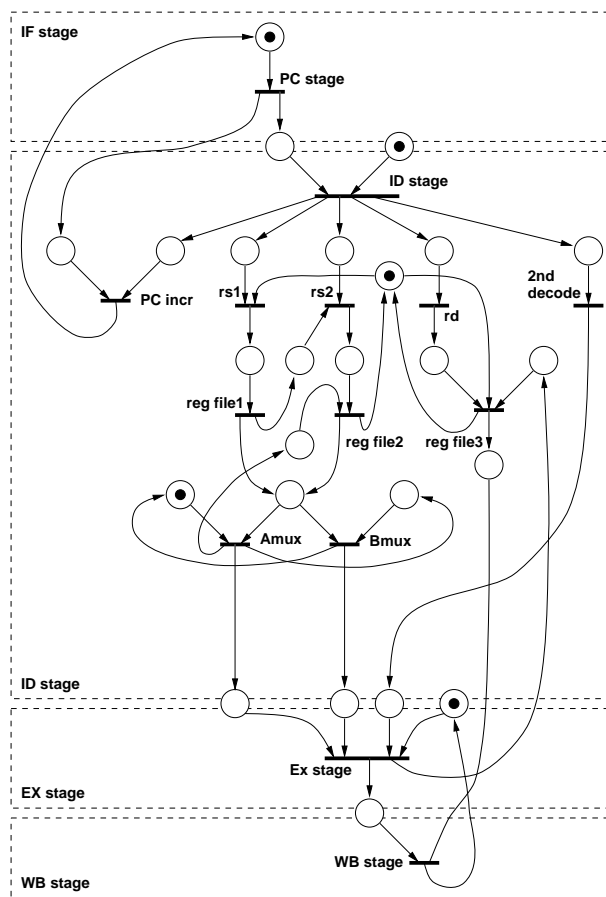Figure 1: Top-level abstract view of the ADLX design.

Figure 2: Petri Net model of decode operation of an R-type instruction.

In [16] it was shown how such a model can be refined to produce more detailed Petri Net descriptions of each stage of the pipeline. The paper presented techniques for converting detailed Petri Net models into parts of the micropipeline control of the processor. A brief summary of these techniques can be described as follows: In order to convert a Petri Net into a circuit description a /it net-level transformation /rm must be performed. There are two possible ways of doing this:

- **Conversion of multiple input arcs:** This is the merging of a number of control operations and so can often be described through the use of an XOR element, for example if we take the Petri Net fragment below (taken from the Petri Net of the Instruction Fetch pipeline stage) and provided that we ensure that all inputs will be mutually exclusive we will realise a hazard-free circuit.
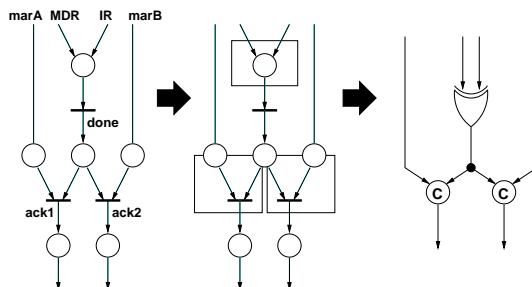

Figure 3: Petri Net to circuit translation of places with multiple input arcs

- **Synchronisation of transitions:** This may be a collection of transitions that, not being capable of synchronisation by themselves, would require a single input event to enable the synchronisation of a request for a particular operation. This type of sequencing is employed extensively throughout the various processing stages of the ADLX.
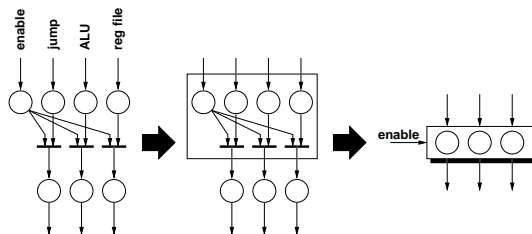

Figure 4: Petri Net to circuit translation for the synchronisation of transitions

4 Logical implementation of ADLX

The development of ADLX was carried out under the following constraints:

- The *stages of execution* or how an instruction is actually pipelined through the synchronous DLX was to be adhered to as closely as possible;

- The *physical* logic devices of the DLX were to be duplicated in the ADLX including, where it was necessary, the employment of strictly asynchronous control circuitry;

- The DLX instruction set was to be implemented without any changes to the composition of that instruction set.

The fundamental change that distinguishes ADLX from DLX is in the manner of the pipeline control of these processors. DLX has one central control unit that is used to forward control signals to the relevant logic units at fixed intervals. In ADLX the control signals that are required to ensure correct operation of a particular stage of pipeline execution are generated from within that particular pipeline stage. This effectively allows each pipeline stage to be *actively processing* for as long as required in order to perform a specific function.

The signal control mechanism used within ADLX is known as transition signalling. In using such a transition signalling method, the actual transition itself (i.e. an event) is active on both the rising and falling edges and allows up to twice the normal clocking rate. The operation of such a signalling method is shown in Figure 5.
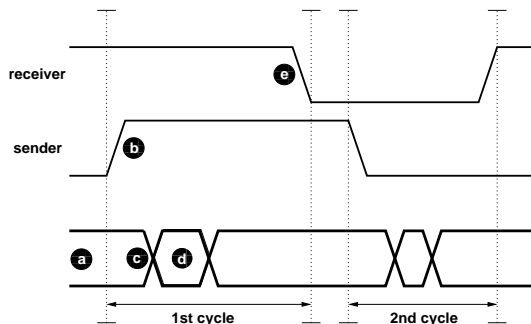

Figure 5: Rising and falling transition events.

- The sender initiates a communication by first placing data onto the data lines at **a** and then placing an event onto the request line at **b**.

- The receiver accepts the request. The data presented to the receiver will be stable at **c**, and so can be processed according to the logic of the receiver at **d**.

- When the receiver has finished processing, an acknowledgement is returned to the sender at **e**.

Transition signalling removes the need to view signals as having either a high or low state and thus of having to return to some neutral state between events (the electrical level of the signal then contains no information). Communication using transition signalling, known as the Two-Phase Bundled Data protocol, is illustrated in Figure 6.
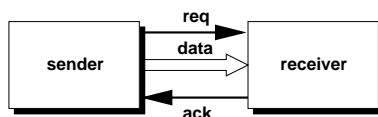

Figure 6: Two-phase bundled data protocol.

A sequence of sender-receiver logic functions is then connected as shown in Figure 7.

Figure 7: Sender-receiver interconnections.

This type of topology is the Sutherland micropipeline [19]. Typically, a micropipeline is composed of a hybrid of bounded-delay and delay-insensitive logic whose topology is to have all the processing actions, the combinatorial logic, forming a bounded-delay datapath that is encapsulated within a delay-insensitive control circuit [3]. The general configuration of a micropipeline employing 2-phase signalling can be seen in Figure 8 and can be translated into a top-level abstract view of the ADLX pipeline, Figure 9.

Figure 8: Sutherland micropipeline.

Figure 9: Top-level view of the ADLX pipeline.

The main drawback of 2-phase signalling is that a individual wire is required for the transfer of the enabling signal relating to a particular instruction. This can be very costly in terms of the required silicon area, especially where large instruction sets are involved, but if suitable translation logic is employed a collection of independent signals can be multiplexed in order to utilise common logic blocks. A top-level schematic of the logic implemented in the FPGA ADLX can be seen in Figure 10.
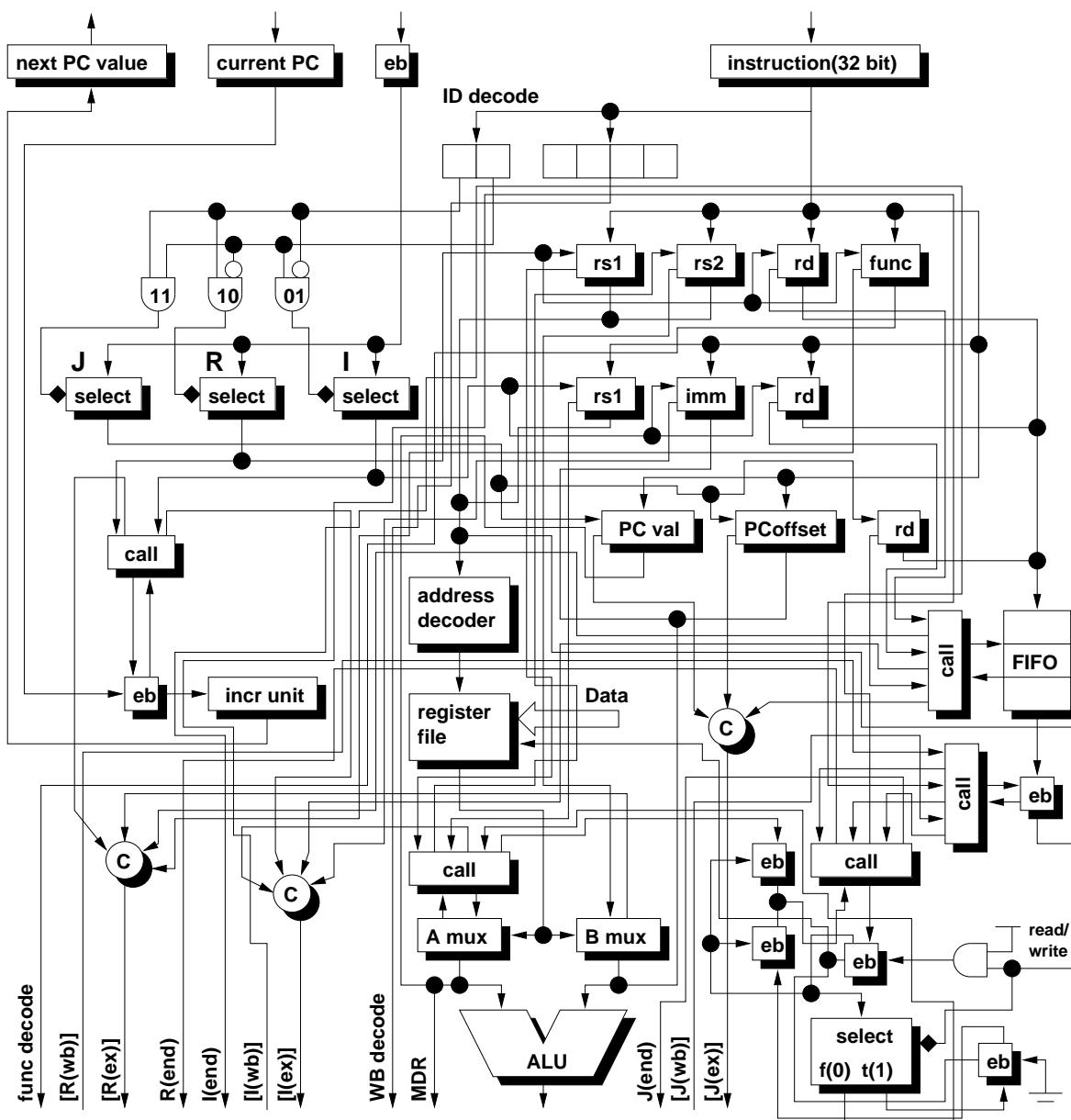

Figure 10: Complete logic schematic of the implemented area of the ADLX.

ADLX incorporates a 32-bit pipeline and has a basic functionality that reads operands from the register file, manipulates those operands in the ALU, and then stores any resultant value back to the register file or to an external memory. The instruction set embodies a fixed-field method of decoding with three types of instruction being provided: loads and stores, ALU operations (integer arithmetic only - no floating point functionality is implemented) and branches. The

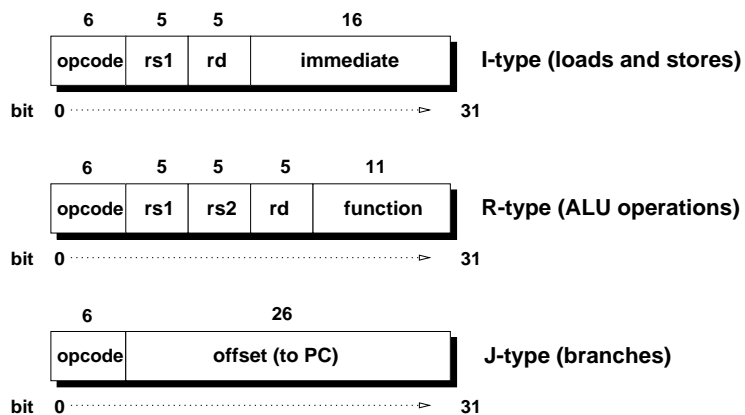bitmap pattern of these instructions is shown in Figure 11.

Figure 11: Instruction layout of the ADLX.

The main features of the ADLX implementation are as follows.

4.1 The Decode Unit

The function of the decode unit is to convert a 2-bit level-based bit pattern into a 1-bit event-based signal which is then used as the enabling signal for the processing actions of an instruction within the Instruction Decode pipeline stage. Initially, bits 0–1 of an instruction opcode are decoded to produce a series of level-based signals which are then forwarded to a number of select elements as boolean enables, see [14]. When used in conjunction with the data value, the enabling signal for the Instruction Decode stage is generated within the Instruction Fetch stage. This Boolean input generates an event-based signal on one of the output *true* lines of one of those select elements (when the boolean has a logical high value) and *no* outputs on any of the other true select output lines. Signals will be generated on the output false lines but these are not required as part of the ADLX specification. This 1-hot of 3 decoding allows for the efficient translation of a level-based signal into an event-based signal.
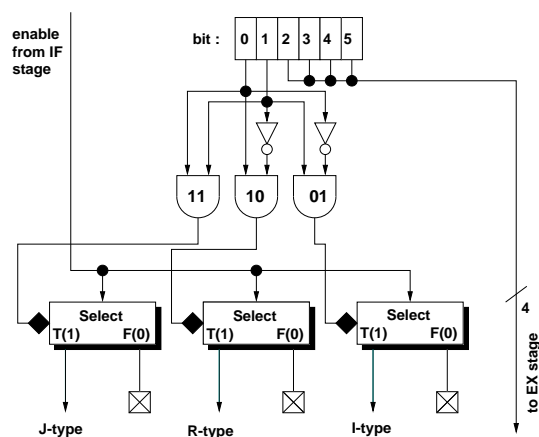

Figure 12: 1-hot of 3 decoding of an instruction opcode.

Bits 2–5 of the opcode are forwarded to the Execution stage to be decoded there. As much

as possible all cases of decode and function determination are carried out only when and where they are needed.

4.2 The Forward FIFO

The Forward FIFO is a simple three stage micropipeline with no internal combinatorial logic that is used to propagate a Register File destination address, for reading or writing operations, to the Register File. Each destination address is a 5-bit opcode that is stored in a series of transparent latches.

4.3 The Register File

The Register File consists of 32 x 32-bit latches that provide 1024 storage elements in total. Register(0) is hard-wired to zero whilst Register(31) is used to preserve the old value of the Program Counter after a branch address calculation has been carried out.

4.4 The Arithmetic Logic Unit

In order to simplify the basic design the implementation of the Arithmetic Logic Unit provides only the integer functions of addition and subtraction. The internal configuration of the ALU is based on an implementation described in [22] and is shown in Figure 13.
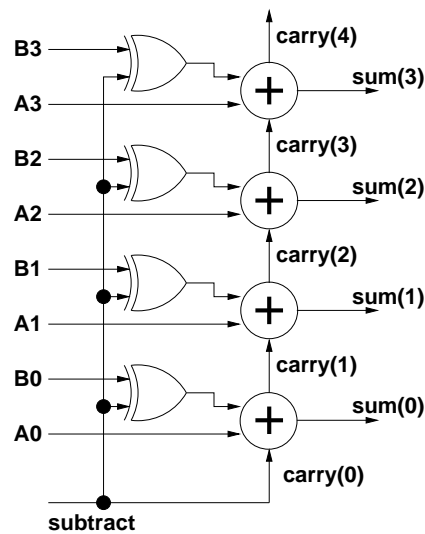

Figure 13: Internal configuration of the Arithmetic Logic Unit.

4.5 Storage elements - the Event Buffer

The logic elements in Figure 10 labelled *eb* are an asynchronous version of synchronous tri-state devices and have been designed to respond to transitional input events. These *event buffers* are a modified form of a capture-pass storage element and have been derived from an original design of a transparent latch as detailed in [14]. The design of the event buffer has been verified by Petrify. The full analysis of this device is described in [10].

11

5   Petri Net based synthesis of ADLX

Petri Nets provide a highly suitable means for the modelling of asynchronous behaviour. *Petrify* has been used to carry out the required synthesis tasks. Petrify produces an optimised net-list in the target gate library that preserves the original input/output signal behaviour. This net-list is guaranteed to be speed-independent, ensuring that the net-list will be *hazard-free* regardless of the distribution of gate delays or multiple input signal changes. For a fuller description of the theory and functionality of Petrify see [1].

The realisation of the Call elements of ADLX was carried out by using Petrify to perform the generation of the synthesis equations. The 2-input Call element, of the type that is used to control the functional operation of the PC increment unit, has a Petri Net whose behavioural analysis leads to the production of those synthesis equations.

The 2-input Call element has the symbolic representation shown in Figure 14(a) with a corresponding internal logic configuration shown in 14(b).
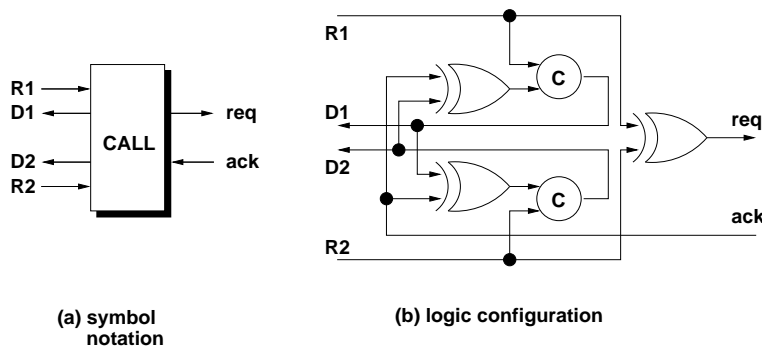

Figure 14: Call element specification.

The Call has a behaviour that is illustrated in the Petri Net shown in Figure 15.
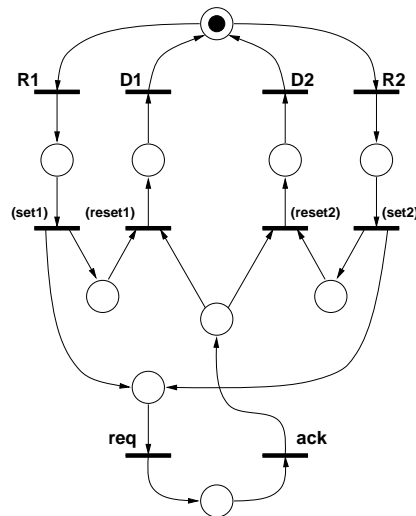

Figure 15: Petri Net specification of a Call element.

The synthesis equations for this Call element as produced by Petrify are as follows:

$$req \quad = \quad (\overline{R1} \cdot R2) + (R1 \cdot \overline{R2})$$

$$D1 \quad = \quad (ack \cdot \overline{D2} \cdot D1) + (\overline{ack} \cdot D2 \cdot D1) + (R1 \cdot ack \cdot \overline{D2}) +$$
$$(R1 \cdot \overline{ack} \cdot D2) + (R1 \cdot D1)$$

$$D2 \quad = \quad (\overline{ack} \cdot D2 \cdot D1) + (ack \cdot D2 \cdot \overline{D1}) + (R2 \cdot ack \cdot \overline{D1}) +$$
$$(R2 \cdot \overline{ack} \cdot D1) + (R2 \cdot D2)$$

These equations are exactly the same as those generated from a VHDL structural description of the Call element as synthesized by the WARP tool [17]. If we take the *D1* synthesis equation we can minimise the sum-of-products as follows.

$$D1 \quad = \quad ((\overbrace{ack \cdot \overline{D2} + \overline{ack} \cdot D2}^{ack \oplus D2}) \cdot D1) +$$
$$((\overbrace{ack \cdot \overline{D2} + \overline{ack} \cdot D2}^{ack \oplus D2}) \cdot R1) + (R1 \cdot D1)$$

Let

$$x \quad = \quad (ack \oplus D2)$$

then

$$D1 \quad = \quad (R1 \cdot D1) + (x \cdot D1) + (x \cdot R1)$$

and so rearranging to a more familiar format for a C-element

$$D1 \quad = \quad (x \cdot R1) + (D1 \cdot (x + R1))$$

we can show that this is equivalent to one half of the Call element as we would expect.
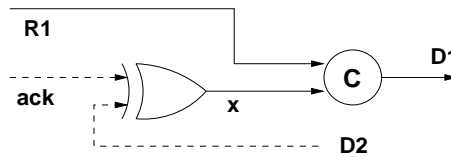

Figure 16: Logical implementation of synthesis equation D1.

The technology mapping aspects of Petrify allow for the targeting of specific cell libraries for implementation, but at the moment this facility does not extend towards FPGA architectures. Petrify essentially processes an original specification as a single unit and generates a *complex gate* description as an end result. Care must therefore be taken when employing Petrify for

FPGA synthesis in that the initial specification has been sufficiently decomposed into modules that are of a size that can be mapped to the FPGA architecture under question. The case may be that inefficient logic utilization by design tools can significantly reduce the physical amount of gates available for an implementation. If we consider the complex gate in 17(a) we can see how the decomposition of this gate, 17(b), may result in a circuit into which hazards may be introduced purely because the complexity of inter-chip routing will cause greater delays.

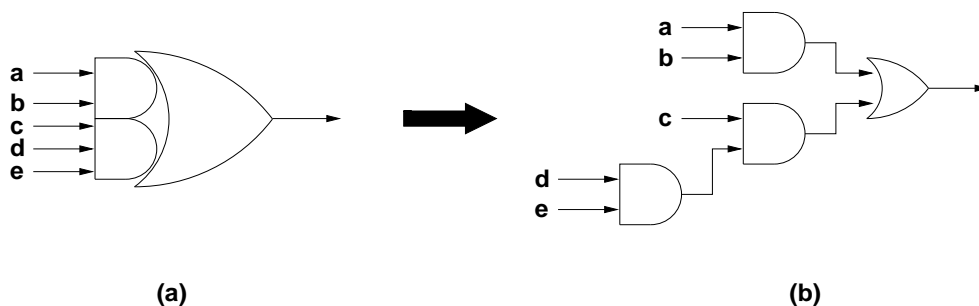(a)   (b)

Figure 17: Decomposition of complex gates.

Alternatively, a large logic block may be implemented as a single complex gate that cannot be mapped to an individual FPGA macrocell or the complex gate may require a greater number of product terms than that macrocell supports. In either case this may result in a partitioning that could violate a timing constraint giving a circuit realization that is no longer speed-independant. These concerns are especially relevant when developing bounded-delay or quasi-insensitive delay, QDI, applications. In the case of QDI logic decomposition by a fitter program across several macrocells may introduce extra levels of logic that disrupts the timing conditions necessary for correct operation.

Petrify does address some of these issues and will perform a resynthesis of a logic design in order to ensure that no gate transition is left unsensed. In this respect Petrify answers the application of decomposition and synthesis concerns raised in [5].

6   Compilation and simulation

ADLX has been implemented using a number of structural representations that have been developed as a series of VHDL behavioural descriptions. A top-down, divide and conquer approach was taken in which logic diagrams were partitioned into a collection of smaller and smaller modules until a point was reached at which it could be determined that a VHDL description could be implemented with the aim of target fitment in a particular device. An abstract view of the logic to be realised in ADLX, as seen in Figure 10, can be seen as a collection of the modules that were constructed.

A number of both software and hardware tools were used for the VHDL compilation and subsequent programming of the FPGA devices. A flowchart illustrating how these packages were connected together is shown in Figure 19.
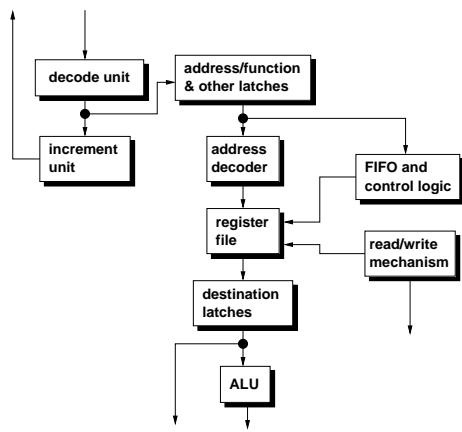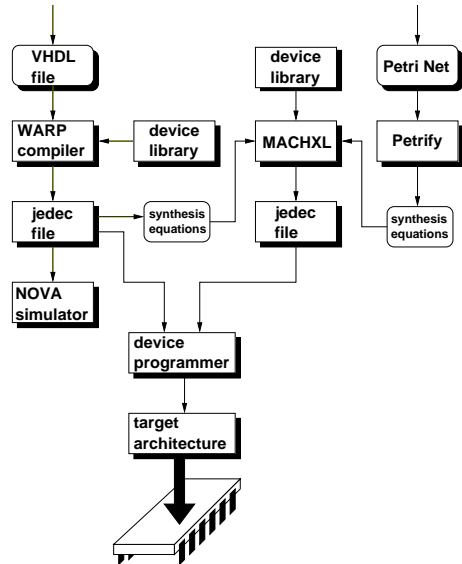
14

Figure 18: FPGA design modules.

Figure 19: FPGA design flowchart.

Depending upon the current FPGA device to be programmed a typical sequence of operations of compilation, simulation and device targeting would be as follows.

- A VHDL file describing a logic module, either structural or behavioural in aspect, would be written and compiled using the VHDL compiler in the CYPRESS WARP Development System [17]. By incorporating a specific target device in the compilation process two main output files would be generated. The first would be a report file that would list the minimised logic equations describing that logic along with other such information as pin-out placement details. The second file would be an industry standard *jedec* file that could be used to program the target device in question.

- Once a successful compilation had taken place the jedec file could be imported into the CYPRESS NOVA Simulation package. NOVA is a jedec functional simulator tool that can be used to read or write stimulus files and simulate the behaviour of a design using a waveform editor.

15

- After the simulation of the compiled design the jedec file could then be used to program a device by employing a Micromaster 1000 programming as supplied by ICE Technology.

- If the device to be targeted for implementation was not of a type supported by the CYPRESS FPGA development tools an alternative software package as produced by Advanced Micro Devices, MACHXL [8], was employed instead. MACHXL is a software package that is designed for the generation of jedec files that are used to program a number of devices in the AMD Mach1XX - Mach4XX range of FPGA products.

- In order to program a Mach device a VHDL file would be compiled and simulated in exactly the same manner as described above. If the design was satisfactory a skeleton MACHXL *pds* file would be created into which would be incorporated the synthesis equations as generated by the respective WARP report file. This pds file could then be compiled to produce a MACHXL jedec to program a device in the manner described above.

- If Petrify was to be employed in order to synthesise an asynchronous logic block a Petri Net that represented the behaviour of that logic would be created. This Petri Net would be used to generate the speed-independent synthesis equations that represented the logic under investigation. These synthesis equations would be imported into a MACHXL pds file for compilation using exactly the same procedure as above.

With regards to the process of performing simulation using the WARP NOVA tool two examples illustrating the usage of the waveform editor can be seen in the following diagrams. In the first case the functionality of the 1-hot of 3 decode mechanism can be seen followed by the operation of a 2-input Call element.
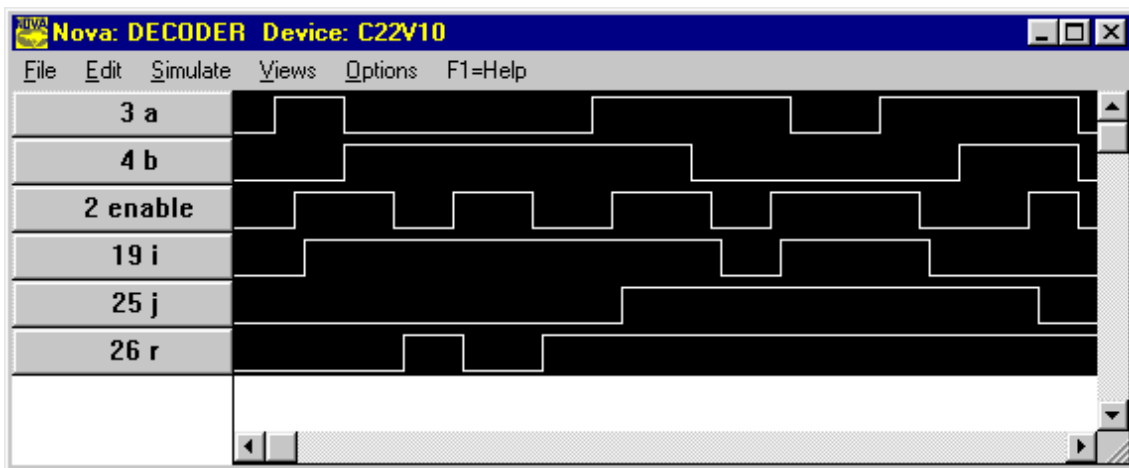

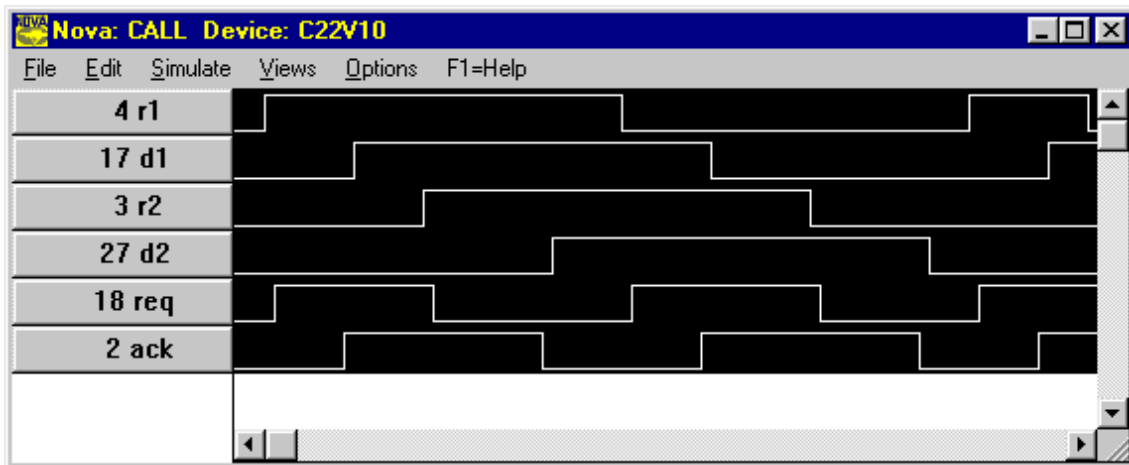Figure 20: NOVA simulation of the decode unit.

Figure 21: NOVA simulation of a 2-input Call element.

From the point of view of the complete synthesis of any design both WARP and MACHXL will follow a common route;

- *Boolean equations* which may be generated from a schematic diagram, a truth table or an HDL.

- *Simplification* in which Boolean equations may be reduced often with the aim of removing redundant terms.

- *Minimisation* in which further equation manipulation may be carried out depending upon the target technology at which they are aimed. Such manipulation may be mandatory in order to comply with device resource requirements or may be purely optional e.g. compiling for area or speed concerns.

- *Technology mapping* in which the Boolean equations are implemented in a specific architecture.

The last two points described above are often referred to as *place-and-fitting* methods and they raise important questions for asynchronous logic design especially with regards to signal delay requirements. Currently most FPGA architectures have little or no support for building delay elements and this can have serious implications for example when attempting to implement bounded-delay circuits. There is the possibility that changes in the ordering of bounded-delay signals, as carried out by a routing algorithm, may reorder those signals to the extent that bounded delay data constraints are no longer valid.

A converse argument to the above concerns speed-independent circuits. As modules of this type of logic will be beholden to local time constraints, partitioning and placement issues will be less likely to cause timing problems within a design. This allows greater flexibility in how these circuits may be mapped to an architecture.

7 Implementation

The physical implementation of ADLX has used a number semiconductor devices, e.g. 4 x HM 65764 8K x 8 High Speed CMOS SRAM memory elements that are used to implement the register file, and two types of programmable logic devices, iCT 22V10's and AMD Mach110's. The iCT logic has been used to implement event buffers and control circuitry, for example, C-elements, Call blocks, Select elements, and Multiplexors.

The Mach110 devices, which are capable of supporting approximately three times the logic of a 22V10, have been used to construct the larger and more regular array type structures, such as the Register file, the Increment unit, the ALU, and the Forward FIFO mechanism.

When examining implementation issues we can see how the asynchronous aspects of a logical design have to be embedded within the strictly synchronous framework of a programmable device. If we consider the event buffer, which has been described as an asynchronous version of a synchronous tri-state element, and implemented in a 22V10 we can see how the output from that event buffer, which may be an asynchronous event based control signal, is still *regulated* by the controlling actions of a synchronous tri-state.
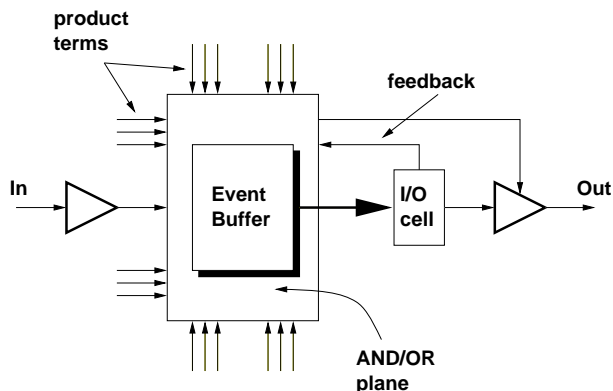

Figure 22: Asynchronous logic embedded in a synchronous framework.

With regards to ensuring that timing conditions were met throughout the ADLX pipeline a mixture of strategies had to be used. When a Petrify complex gate representation of a logic block was being developed it was necessary to ensure that the logic specification had been sufficiently decomposed down to a module size that, even if partitioning was to occur within the target device, the delay constraints would still be met. The fact that Petrify generated a speed-independent circuit description did not remove the small but finite delay requirements that would still be needed to guarantee the correct functionality of the design. A certain amount of partitioning could take place of a complex gate, but it must be ensured that changes to the delay propagation of any internal signal did not conflict with the timing requirements of the surrounding environment. If these constraints were exceeded due to partitioning it would be likely that hazards would be introduced into the circuit.

To ensure bounded-delay timing conditions were met a more ad-hoc approach had to be taken. To overcome the lack of delay element support by the FPGA, hardware invertor chains

were built into the VHDL descriptions. As tools were not available to perform sufficient timing analysis this often meant that a VHDL logic description had to undergo several revisions, essentially by altering invertor chain length, in order to achieve the necessary timing requirements. This was especially the case when designing the access cycle time of the register file.

8 Testing

In order to test the implementation of ADLX a simple program was designed that could be used to exercise all the paths of the pipeline as required by the three instruction types. The general purpose of this program was to load data from an external memory into the register file, manipulate that data through addition and subtraction functions and then return any resultant values back to that external memory. Table 1 shows a typical sequence of execution of instructions.

Instruction	Decode	Execution	Write Back
1	load1	-	-
2	load2	load1	-
3	load3	load2	load1
4	alu1	load3	load2
5	jump1	alu1	load3
6	alu2	jump1	alu1
7	store1	alu2	jump1
8	store2	store1	alu2
9	-	store2	store1
10	-	-	store2

Table 1: Sequence of instruction execution.

There are instances in the execution of a number of instructions in ADLX that require the concurrent access of the register file or of other such logic. As ADLX has not been designed to function in a concurrent pipelined manner, e.g. only sequential processing of instructions can occur in any pipeline stage, there are times when a particular processing action must take precedence over some other such action. With regards to the Instruction Fetch stage there are three cases of when such precedence of action must be allowed to take place. These can be described as follows;

- *Priority 1:* Read/write access of the register file as dictated by a Write Back stage processing action.

- *Priority 2:* Read/write access of the register file as dictated by an Execution stage processing action.

19

- *Priority 3:* Read access of the register file as dictated by an Instruction Decode stage processing action.

The obvious conclusions from the above statements are that there are likely to be a large number of *stalls* in the ADLX pipeline and this has in fact been determined to be the case. Changes to the design of the ADLX as implemented and described in this paper for the correction of these problems have already been suggested. The initial proposal for an ADLX2 is to employ a 4-way superscalar rotary pipeline that also incorporates a binary translation mechanism to allow for the concurrent execution of functionally independent scalar instruction sets.

The ADLX test program, along with a number of other numeric constants, e.g. values representing data that would be actually found in the external memory, have been implemented in a number of EEPROM devices which have been connected to the ADLX pipeline. The general configuration of this external logic with the associated connectivity can be seen in Figure 23.
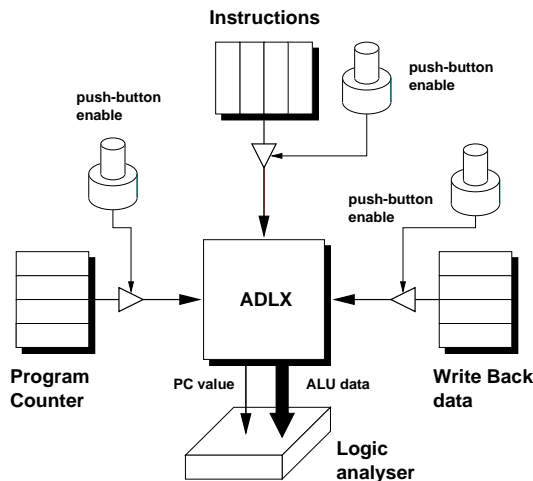

Figure 23: EEPROM input vector test units as connected to the ADLX.

The type of EEPROM unit employed was a 32K 27256 device that was 8-bit addressed. In order to present a 32-bit instruction or data value to an ADLX bus four of these devices would be enabled in parallel in order to construct that 32-bit word. The structural organisation of these EEPROM units can be seen in Figure 24.
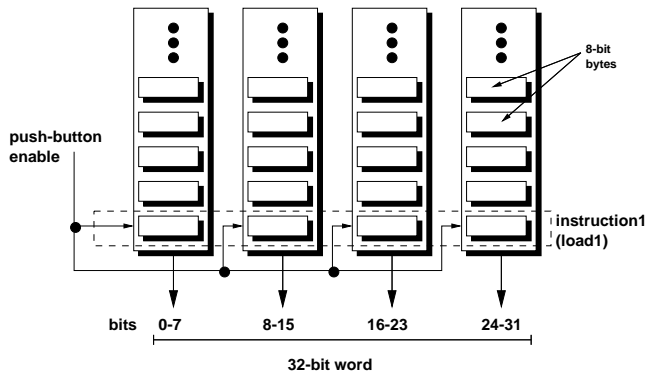

Figure 24: EEPROM 32-bit word configuration.

20

Three such EEPROM configurations were required to provide the simulation of;

- The program instructions as received from the external memory;

- The program counter;

- The data values/memory addresses as calculated and returned from the Write Back stage to the Instruction Decode stage.

These were activated using a simple push-button mechanism to supply the relevant 2-phase enabling signal. All calculations of addresses or the products from arithmetic functions as performed in the ALU together with the incrementation of the program counter value were captured by a logic analyser for examination and evaluation.

9 Conclusions

The design and implementation of ADLX has shown that a combination of both academic and commercially available tools can be used to construct relatively complex asynchronous circuits using programmable logic. ADLX was not intended to be a fully custom designed processor such as the AMULET processors, but rather as a prototype device to investigate how existing tools could cope with such a task.

One aspect of asynchronous design that was not addressed is the problem of arbitration. The metastable nature of an arbiter means that such logic cannot be implemented in the current programmable logic devices that exist today. Metastability cannot be eliminated from synchronous systems subject to asynchronous inputs but these effects can be reduced by employing low frequency signal rates, by synchronising asynchronous inputs or by the addition of extra *clock* cycles to improve signal resolution. A combination of these solutions may be implemented in the future.

If programmable hardware tools are to be used with the aim of the manufacture of commercial asynchronous programmable logic devices, then the major software tool vendors must begin to incorporate into their software the facility for the correct verification and synthesis of those designs.

A more realistic approach to the development of asynchronous logic for programmable hardware may be to use a *sea-of-gates* approach. In such an architecture all macrocells touch each other, thus allowing for fast interconnections between those cells. This means that compact functions can be implemented that do not suffer from having to be decomposed over a number of logic blocks, and therefore ensure that all timing constraints can be adhered to. A sea-of-gates architecture would be an ideal target for Petrify because there would be no restrictions to the size of complex gates that Petrify could synthesise.

Acknowledgements

References

[1] J. Cortadella, A. Kondratyev, M. Kishinevsky, L. Lavagno, and A. V. Yakovlev. **Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous circuits**. In *Proc. 11th Conference on Design of Integrated Circuits and Systems (DCIS'96), Barcelona*, pages 205–210, November 1996.

[2] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, S. Temple, and J. V. Woods. **The Design and Evaluation of an Asynchronous Microprocessor**. In *Proc. International Conf. Computer Design (ICCD)*. IEEE Computer Society Press, October 1994.

[3] S. Hauck. **Asynchronous Design Methodologies: An Overview**. *Proc. of the IEEE*, 83(1):69–93, January 1995.

[4] S. Hauck, G. Borriello, S. Burns, and C. Ebeling. **MONTAGE: An FPGA for Synchronous and Asynchronous Circuits** . In *2nd International Workshop on Field-Programmable Gate Arrays, Vienna*, August 1992.

[5] S. Hauck, S. Burns, G. Borriello, and C. Ebeling. **An FPGA for Implementing Asynchronous Circuits**. *IEEE Design & Test of Computers*, 11(3):60–69, 1994.

[6] J. L. Henessey and D. A. Patterson. **Computer Architecture: A Quantative Approach**. Morgan Kaufman Publishers, Inc., P. O. Box 50490, Palo Alto, California 94303, 1990.

[7] J. L. Henessey and D. A. Patterson. **Computer Architecture: A Quantative Approach (2nd. ed.)**. Morgan Kaufman Publishers, Inc., P. O. Box 50490, Palo Alto, California 94303, 1996.

[8] Advanced Micro Devices Inc. **MACHXL Software User's Guide**. Advanced Micro Devices Inc., P.O. Box 3453, Sunnydale, CA 94088, 1993.

[9] L. Lloyd. **The ADLX: An Asynchronous RISC Implementation of the DLX Microprocessor Architecture**. Master's thesis, Department of Computing Science, University of Newcastle upon Tyne, September 1995.

[10] L. Lloyd, A. V. Yakovlev, and A. M. Koelmans. **A 2-Phase Asynchronous Event Driven Buffer with Completition Detection Signalling**. Technical Report 573, Department of Computing Science, University of Newcastle, February 1997.

[11] K. Maheswaran and J. B. Lipsher. **A Cell Set for Self-Timed Design Using Xilinx XC4000 Series FPGA**. Technical report, U.C.Davis, 1994.

[12] T. Murata. **Petri Nets: Properties, Analysis and Applications**. *Proc. of the IEEE*, 77(4):541–580, 1989.

[13] J. V. Oldfield and R. C. Dorf. **Field Programmable Gate Arrays**. John Wiley & Sons Inc., 1995.

[14] N. C. Paver. **The Design and Implementation of an Asynchronous Microprocessor**. PhD thesis, Department of Computer Science, University of Manchester, England, June 1994.

[15] R. Payne. **Self-Timed FPGA Systems**. In *5th International Workshop on Field Programmable Logic and Applications*, 1995.

[16] A. Semenov, A. M. Koelmans, L. Lloyd, and A. V. Yakovlev. **Designing an Asynchronous Processor using Petri Nets**. *IEEE Micro*, 17(2):54–64, 1997.

[17] Cypress Semiconductor. **WARP^{TM} VHDL Development System: WARP Synthesis Compiler Manual**. Cypress Semiconductor, 3901 North First Street, San Jose, CA 95134, January 1995.

[18] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. **SIS: A System for Sequential Circuit Systhesis**. Technical Report UCB/ERL M92/41, Department of Computer Science, University of California, Berkeley, May 1992. Electronics Research Laboratory Memorandum.

[19] I. E. Sutherland. **Micropipelines**. *Communications of the ACM*, 32(6):720–738, June 1989.

[20] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij. **Asynchronous Circuits for Low Power: A DCC Error Corrector**. *IEEE Design & Test of Computers*, 11(2):22–32, Summer 1994.

[21] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij. **The VLSI-Programming Language Tangram and its Translation into Handshake Circuits**. In *Proc. European Conference on Design Automation (EDAC)*, pages 384–389, 1991.

[22] N. H. E. Weste and K. Eshraghain. **Principles of CMOS VLSI Design**. Addison-Wesley Publishing company, 1992.

[23] Q. Zhang and H. Grünbacher. **Petri Nets Modeling in Pipelined Microprocessor Design**. *Lecture Notes in Computer Science*, 691:582–591, 1997.