

Towards an Asynchronous MIPS Processor

Qianyi Zhang and Georgios Theodoropoulos

School of Computer Science, University of Birmingham
Birmingham B15 2TT, United Kingdom
email: {qyz,gkt}@cs.bham.ac.uk

Abstract. *Synchronous VLSI design is approaching a critical point, with clock distribution becoming an increasingly costly and complicated issue and power consumption rapidly emerging as a major concern. Hence, the last decade has witnessed a resurgence of interest in asynchronous digital design techniques as they promise to liberate VLSI systems from clock skew problems, offer the potential for low power and high performance and encourage a modular design philosophy which makes incremental technological migration a much easier task. This paper discusses an asynchronous version of the MIPS microprocessor, presenting the techniques that have been devised to address data and control hazards.*

1 Introduction

Conventional synchronous architectures use design techniques based on global clocking whereby all the functional units operate in lockstep under the control of a central clock [16]. As VLSI technology advances and systems become larger, faster and more complex, timing problems become increasingly severe and account for more and more of the design and debugging expense. Increased clock speeds make on-chip clock skew significant and inter-chip skew a major problem. One solution to clock-related timing problems is to use asynchronous design techniques without any global synchronization signals to control the rate at which different elements operate. Other potential advantages of asynchronous logic, are low power consumption, high performance and support for a modular design philosophy which makes incremental technological migration a much easier task. As a result, the last decade has witnessed a resurgence of interest in asynchronous systems.

An asynchronous system may be designed as a set of functional modules (subsystems), which communicate only when it is necessary to exchange information. The operation of the system does not proceed in lockstep, but rather is *asynchronous*; each sub-system operates at its own rate synchronising with its peers only when it needs to exchange information. This synchronisation is not achieved by means of a global clock but rather, by the communication protocol employed. This protocol is typically in the form of local request and acknowledge signals which provide information regarding the validity of data signals.

Various asynchronous digital design techniques have been developed, which are typically categorised by the timing model, the signalling protocol and the

data transfer technique they employ. In his influential 1988 Turing award lecture, Ivan Sutherland introduced *Micropipelines*, a new conceptual framework for designing asynchronous systems [25]. The Asynchronous Online Logic Home Page maintained by the AMULET group at the University of Manchester provides continuous, up to date information regarding asynchronous systems research [1].

A number of asynchronous architectures have been developed [28] including one at CalTech [14], NSR [4] and Fred [22] at the University of Utah, STRiP at Stanford University [6], Sun's Counterflow pipeline processor [24], FAM [5] and TITAC [17] at Tokyo University and Institute of Technology respectively, Hades at the University of Hertfordshire [7], Sharp's Data-Driven Media Processor [23] and the series of asynchronous implementations of the ARM RISC processor (AMULET1 [29], AMULET2e [10], AMULET3i [11] and SPA [21]) developed by the AMULET group at the University of Manchester.

Contributing to this effort, we have embarked on work to develop an asynchronous implementation of the MIPS architecture. This work forms part of a larger project which aims to develop an integrated framework for formal verification and distributed simulation of Asynchronous Hardware, utilising Balsa, a CSP-oriented synthesis tool developed at the University of Manchester [2]. The project is jointly undertaken by the Modelling and Analysis of Systems group at the University of Birmingham and the AMULET group at the University of Manchester and is funded by EPSRC¹. This paper discusses the initial findings of our investigation, and presents the techniques that have been devised to address data and control hazards.

2 MIPS Architecture

For our purposes, we are using the base MIPS application architecture as described in [13, 18] and as exemplified by the R3000 processor.

MIPS R3000 is a 32 bit microprocessor consisting of two tightly-coupled processors, namely a full 32-bit RISC CPU, and a system control co-processor, referred to as CP0 as shown in figure 1. The processors are implemented on a single chip and can be extended with three off chip co-processors. The CPU has thirty two 32-bit general-purpose registers, two 32-bit registers for multiplication and division results, one program counter (PC), and a control logic unit. The datapath includes an ALU, a Shifter, a Multiplier/Divider, an Address Adder, and a PC incrementer.

The CP0 co-processor includes exception and control units and memory management hardware for address translation in the form of an on-chip 64 entry Translation Lookaside Buffer.

MIPS datapath is built around a five stage pipeline consisting of (figure 2): Instruction Fetch (IF), Decode/Register File Read (ID), Execution or Address Calculation (EX), Memory Access (MEM), Register Write-back (WB). It is a Harvard architecture utilizing two memory ports one for instruction fetches and one for data accesses.

¹ <http://www.cs.bham.ac.uk/~gkt/Research/par-lard/>

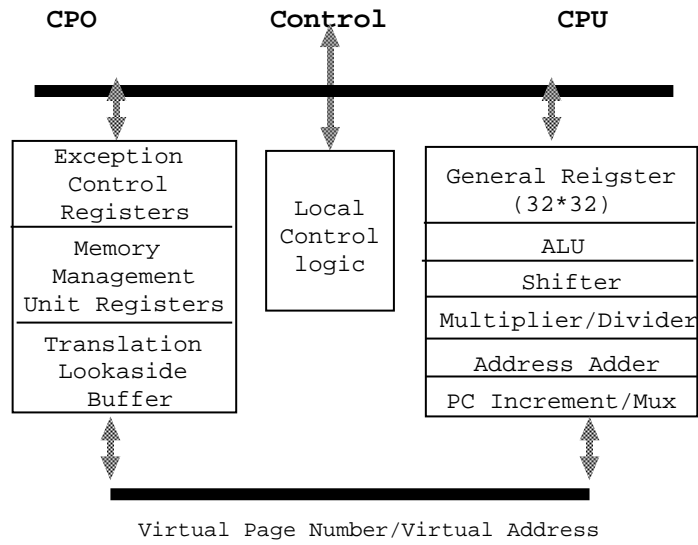


Fig. 1. MIPS R3000 Functional Blocks

MIPS supports three types of instructions: *I-TYPE* (immediate), *J-TYPE* (jump) and *R-TYPE* (register), as illustrated in figure 3. *rs*, *rt*, *rd* are operand register numbers, *immediate* is either an immediate operand or an offset for branch/memory address calculation, *target* is the jump target offset, *funct* is a supplement to Opcode and *sa* is a shift amount for shift operations.

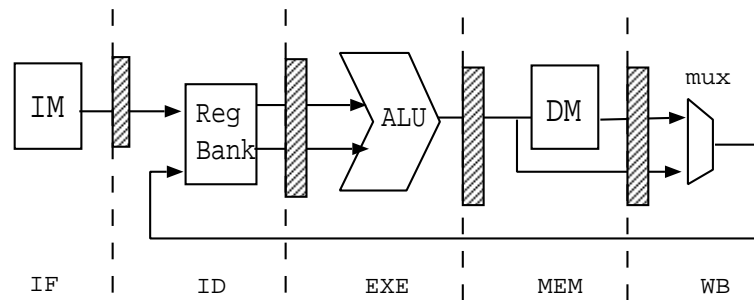


Fig. 2. MIPS Datapath

For more information on the MIPS processor, the reader is referred to e.g. [13, 18].

I-TYPE:	Opcode	rs	rt	immediate		
J-TYPE:	Opcode			target		
R-TYPE:	Opcode	rs	rt	rd	sa	funct

Fig. 3. MIPS Instruction Formats

3 Towards an Asynchronous Design

Our main objective for designing an asynchronous MIPS, is to use it as a test case for our integrated formal verification and distributed simulation environment. Within this environment, designs are specified in terms of Balsa, a CSP-based Hardware Description Language, at the Register Transfer level, as it is at this level that the communication and computation semantics of CSP can capture the concurrent, nondeterministic behaviour of asynchronous hardware [26]. Consequently, our effort to design an asynchronous MIPS targets the Register Transfer Level.

Balsa generates purely asynchronous macromodular circuits similar to those of Philip’s Tangram [20]. Descriptions of RTL Balsa designs are translated into implementations in a syntax directed-fashion with language constructs being mapped into networks of parameterised instances of “handshake components” each of which has a concrete gate level implementation. It is technology independent (e.g. channel connections can be implemented using speed-independent or delay-insensitive schemes) and it targets standard cell and FPGA technologies for producing gate-level netlists. For our MIPS design, we have assumed a 2-phase bundled data signalling protocol.

Another important decision that was taken was to initially adhere to the five stage pipeline of the synchronous MIPS. A five stage pipeline design will provide a basis for comparison with previous attempts to develop an asynchronous MIPS most notably that undertaken at Caltech [15] which chose to adopt a three stage pipeline arguing that this would exploit better the potential advantages of asynchronous logic. Furthermore, the five stage pipeline introduces challenging hazard-related problems that call for innovative asynchronous solutions.

Three main problems with regard to the asynchronous MIPS design have been addressed: distributing the control, dealing with data hazards and tackling control hazards. The next sections describe these problems and the solutions that have been devised.

3.1 Distributing the Control

Asynchronous logic calls for distributed control schemes, which facilitate the concurrent, asynchronous operation of the system.

Assuming a correct implementation of the communication protocol, at the Register Transfer Level, an asynchronous system may be viewed as a network of concurrent modules communicating via synchronous, unbuffered communication. The modules are data-driven; each module will start computation as soon as

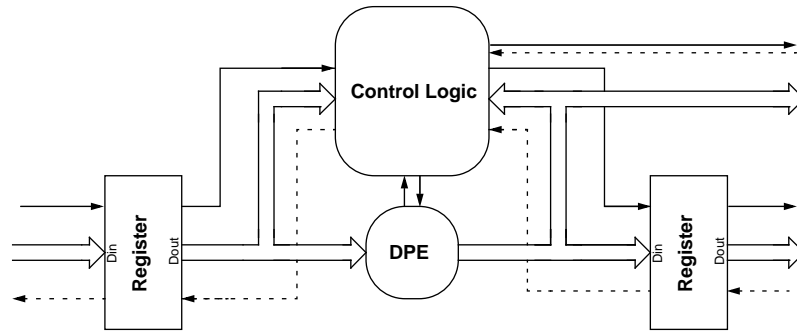


Fig. 4. Asynchronous Pipeline: A High Level View

data is available on its input wires, and will signal when its result has been computed. Within this framework, control signals are bundled together with the corresponding data, accompanying the latter through the pipeline. Thus, at the Register Transfer Level, a general asynchronous pipeline with processing may be viewed as depicted in figure 4. The sending register outputs its contents, consisting of data and control bits, onto the data bus and produces a request event (request wires are indicated in the figure by solid lines, while acknowledge wires are denoted by dotted lines). The control bits are used by the control logic to direct the request event to its correct destination activating, if necessary, the data processing elements (DPEs, e.g. ALUs, multipliers, shifters etc.) of the datapath. Data passes through the DPEs and propagates to the next stage.

Synchronous MIPS utilises a centralised control unit in ID stage as depicted in figure 5a. This unit produces the necessary control signals which propagate through the pipeline together with the data to drive circuits in the different stages of the datapath. This scheme provides a natural basis to generate and distribute the control information in the asynchronous design.

Figure 5b illustrates the asynchronous design. A main Decode unit is placed in the ID stage to perform the instruction decoding and generate the control signals required for the different stages; these signals will thereafter follow the data through the pipeline (figure 6) driving a set of decentralised local control circuits.

4 Dealing with Data Hazards

In pipelines systems, there are situations where the next instruction, although it has been prefetched and has entered the pipeline, it cannot or must not execute in the following cycle. One such situation arises when an instruction depends on the results of a previous instruction still in pipeline and is referred to as data

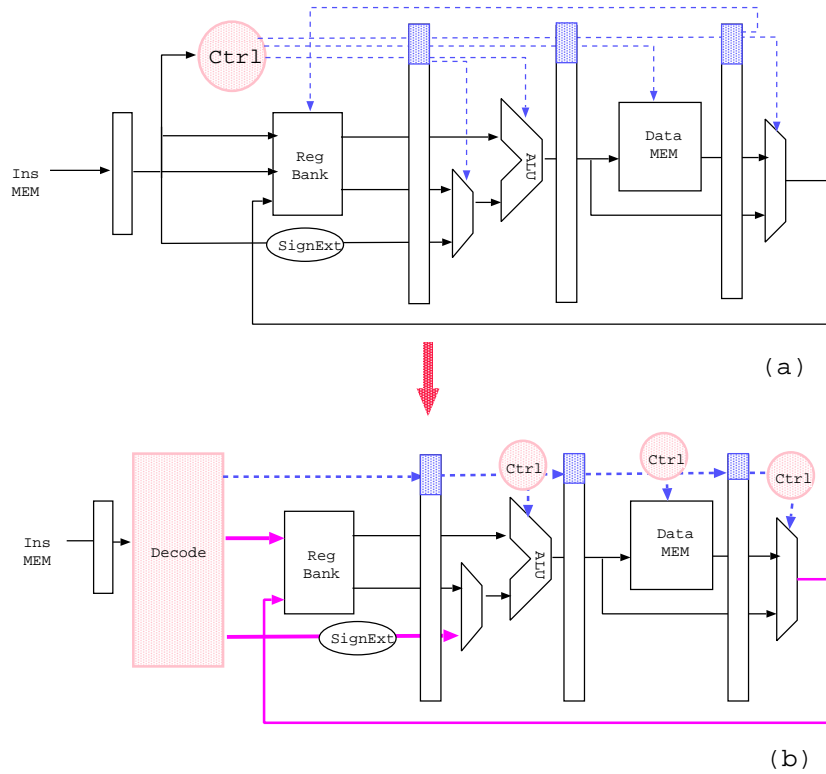


Fig. 5. Distributing the Control

hazard. Figure 7 shows a sample MIPS code² where a data hazard will occur because of the write back delay to register \$2.

Two main approaches have been developed to deal with this problem in synchronous architectures. The first, simple albeit slow, approach stalls the pipeline by locking the Register Bank until the needed operand is written back. The second, referred to as forwarding, attempts to get the missing item earlier from the internal resources.

Efforts have been made to utilise these techniques in asynchronous designs too however asynchronous forwarding has proved a very challenging problem [10]. The AMULET1 microprocessor used register locking [19]. AMULET2 combined limited forwarding measures by employing a “last result register” at the output of the ALU and a “last loaded value” register at the output of memory [10]. AMULET3 uses a reorder buffer implemented as asynchronous FIFO [12]. The reorder buffer hides the load latency by receiving memory data in an arbitrary order at random, reordering them in the buffer and then forwarding

² The example has been taken from [18]

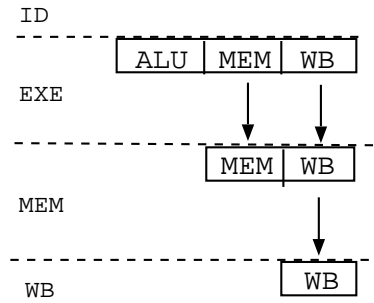


Fig. 6. Passing the Control in Different Stages

them back if necessary. Sun’s Counterflow Processor has a radically different solution whereby the results are sent “backwards” up the pipeline to meet following instructions which propagate in the opposite direction and thus resolve register dependencies rapidly [24]. Another innovative idea has been exploited in SCALP, an asynchronous superscalar machine which attempted to avoid register storage and only forward results by using a purpose designed instruction [8].

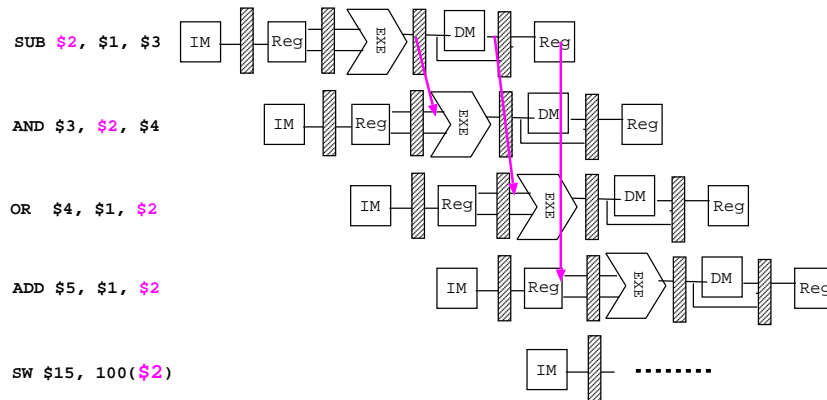


Fig. 7. Data Hazards: An Example

4.1 Forwarding in MIPS

MIPS uses forwarding to handle the data hazard problem as illustrated in figure 8a. The operand of ALU has three sources, namely Register Bank or a forwarded result from EX and MEM stages. The decision as to which source the ALU should

use at any particular moment is taken by a centralised control unit which drives the multiplexers at the ALU input.

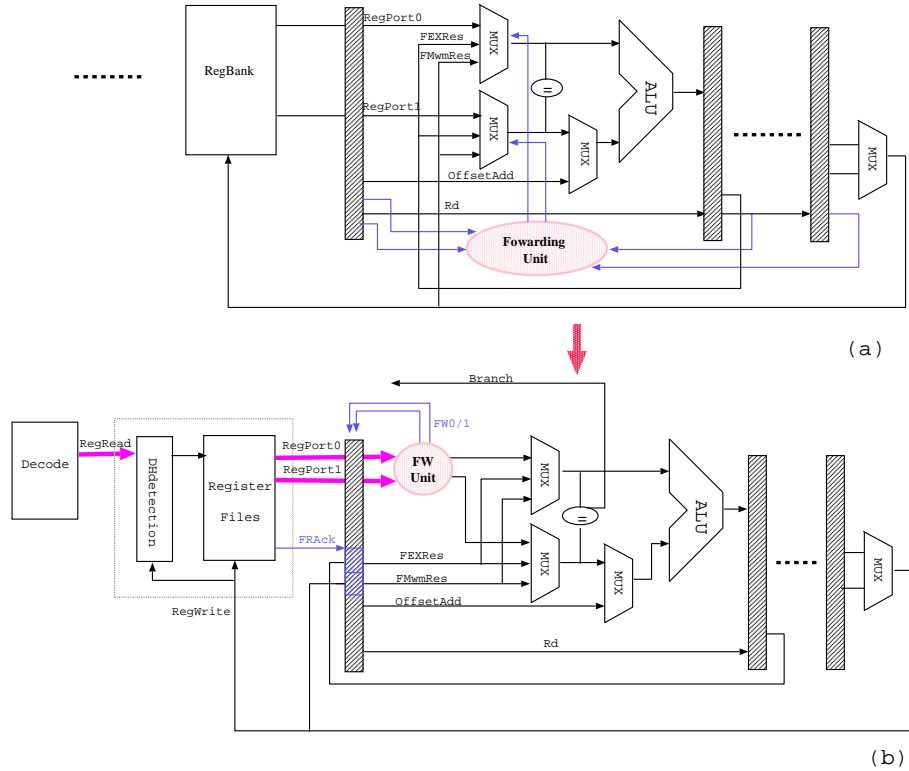


Fig. 8. Register Bank with Result Forwarding

Referring to the example in figure 7, the following behaviour will be exhibited:

- For instruction AND, operand (\$2) is forwarded from EX stage of instruction SUB
- For instruction OR, operand (\$2) is forwarded from MEM stage of instruction SUB
- For instruction ADD, we can separate the ID stage into two substages: first do decode and write back to register, then read from register files
- For instruction SW, a data hazard will never happen, since the instruction SUB has already executed.

4.2 An Asynchronous Forwarding Mechanism for MIPS

Since all the aforementioned asynchronous forwarding mechanisms exploit the particular characteristics of the respective architectures, they cannot be applied

<i>RegNo</i>	<i>Flag</i>	
	<i>Clean</i>	<i>Index</i>
0	1	/
1	0	2
...
31	/	/

Fig. 9. Data Hazard Detection Table

in MIPS and therefore, a new mechanism is required. In Caltech's asynchronous MIPS, the use of a three instead of a five stage pipeline greatly simplifies the problem. In that system, a "bypass" unit is introduced inside the register bank to bypass the required operand when a data hazard occurs.

The objective here is to develop a mechanism that would allow the centralised Forwarding Unit (figure 8a) to be removed and have the control signals that drive the multiplexers at the ALU somehow sent down the pipeline bundled with the corresponding data. The control signals should specify which stage will potentially forward a result (the result may come from two places: ALU output in MEM stage and memory result in WB stage) and whether the result will be forwarded or whether it is needed. The fundamental problem is that forwarding depends on knowledge of global state, while in a distributed non-deterministic system such as an asynchronous architecture, global snapshots of the state are not easily or efficiently obtained. The solution devised does not depend on global current knowledge but rather on knowledge of the past.

Indeed, by observing the sequence of instructions as they pass through, a data hazard can be detected when performing a register read at the ID stage. The output from the register bank is a good candidate to carry this hazard control information to the multiplexers in the EXE stage. The action of the multiplexer will then be to choose the valid data or acknowledge the forwarded result. This does not introduce additional synchronization between the ID and EXE stages. The synchronization between EXE and MEM/WB stages can be removed by employing a buffer for the forwarded results. The pipeline structure with the forwarding is shown in figure 8b.

To achieve data hazard detection, another unit is introduced in the register bank as depicted in figure 8b. The unit utilises a table (called Data Hazard Detection Table - DHDT) which keeps a record of all the passing instructions which will do a register write back and are still in the pipeline. The structure of the table is shown in figure 9. It contains three bits of information for each register referred to as the register *Flag* and consisting of: one bit *Clean*, indicating whether the register is pending to be written; and two bits *Index* essentially indicating which instruction will rewrite the register (or, in other words, which stage will forward the result). At most four instructions can be at any time in the pipeline from ID to WB stage, and therefore two bits are enough for the Index field. This scheme introduces a total of $(2+1)*32=96$ additional bits in

the register bank. The algorithm makes use of another two bits to point to the current instruction in the register bank, called *CurIndex*, which is incremented by one (module 4) every time a new instruction enters the register bank.

As depicted in figure 8b, the forwarding control information that is required at the EX stage, can now be bundled together with the data from the register bank. *RegRead* is a 18-bits channel with the first 3 bits indicating whether this instruction needs to do a register read/write while the following 15 bits give the corresponding register address. *RegWrite* contains two fields, 5 bits for register address and 32 bits for the write back data. *RegPort0* and *RegPort1* contain two fields, a 32-bit register data and a 2-bit control field with data forwarding information passed to the EX stage.

The forwarding algorithm includes three parts:

1. *Initialization*: set *CurIndex* to zero.
2. *Write Register*: two actions are performed when a *RegWrite* signal is sent;
 - (a) write new value to the corresponding *register[i]*
 - (b) set *Flag.clean[i]* to True
3. *Read Register*: three actions are performed when a *RegRead* signal is sent:
 - change the register’s *Flag*. Check whether *register[i]* is pending to be written back. If so set *Flag.Clean[i]* to False and *Flag.Index[i]* to *CurIndex*
 - read register: data hazard is checked simultaneously with register reading if *Flag.Clean[i]* is True then read from *register[i]*
 - else
 - if *Flag.Index[i]* equals to *CurIndex-1* then the result is forwarded from EX
 - if *Flag.Index[i]* equals to *CurIndex-2* then the result is forwarded from MEM
 - else wait for a *RegWrite* signal, then output the new value directly and simultaneously write back into register.
 - increase *CurIndex* by 1.

4.3 A Constructive Proof

The proposed mechanism has been simulated in Balsa and proved correct. As a constructive proof, figure 10 presents a series of snapshots of the DHDT and forwarding logic inside the Register Bank for the instruction sequence shown in figure 7.

5 Dealing with Control Hazards

In conventional, von Neumann machines, instructions are executed sequentially, from consecutive memory locations unless a control hazard, namely the execution of an instruction such as a branch or a jump, or the occurrence of an unpredictable event, such as an exception, changes the flow of control. In a pipelined architecture, if a control hazard occurs, the prefetched instructions following a

<i>Instruction</i>	<i>CurIndex</i>	<i>DHDT</i>	<i>Operation</i>
SUB \$2, \$1, \$3	0	... (1, 1, /) (2, 0, 0) (3, 1, /)	Flag[2].clean=1,Flag[3].clean=1, Using data from RegBank as operands
AND \$3, \$2, \$4	1	(2, 0, 0) (3, 0, 1) (4, 1, /)	\$2 not clean, Index[2]=CurIndex-1, using the forwarded result from MEM stage
OR \$4, \$1, \$2	2	(2, 0, 0) (3, 0, 1) (4, 0, 2)	\$2 not clean, Index[2]=CurIndex-2, using the forwarded result from MEM stage
ADD \$5, \$1, \$2	3	(2, 0, 0) (3, 0, 1) (4, 0, 2) (5, 0, 3)	\$2 not clean, Index[2]=CurIndex-3, waiting for a result coming back
SW \$5, 100(\$2)	0	(2, 1, /) (3, 0, 1) (4, 0, 2) (5, 0, 3)	\$2 should be valid again, otherwise waiting for a result coming back

Fig. 10. Data Hazard Example

hazard must be discarded and removed from the pipeline before instructions from the new stream (e.g. the branch target address or the exception vector address) are executed. Pipeline stall, branch prediction and delayed branches are techniques that have been devised to deal with this problem.

In MIPS, there are two types of instructions that can cause transfer of control, conditional BRANCH and unconditional JUMP. MIPS uses delayed branches, inserting NOPs or instructions not dependent on the branch to avoid flushing the pipeline. In the context of this paper, we do not examine exceptions.

In synchronous pipelined systems, the depth of prefetching, namely, the number of instructions that have entered the processor and thus must be discarded in the case of a control hazard, is defined by the clock cycles and is therefore deterministic. In an asynchronous microprocessor however, where the prefetch unit is completely autonomous and decoupled from the rest of the processor, the exact number of the prefetched instructions is nondeterministic and therefore unpredictable. In this case, the depth of the prefetching depends on the precise point that the interruption of the prefetching by the branch target or the exception vector address takes place. The processor must be able to distinguish between instructions originating from the branch or the exception target, which may thus be executed, and instructions already prefetched when the hazard took place, which must therefore be thrown away.

Different approaches have been followed in different asynchronous processors to deal with this problem [27]. A very neat and efficient solution was devised for the AMULET1 processor. This technique uses a single bit to “colour” the state

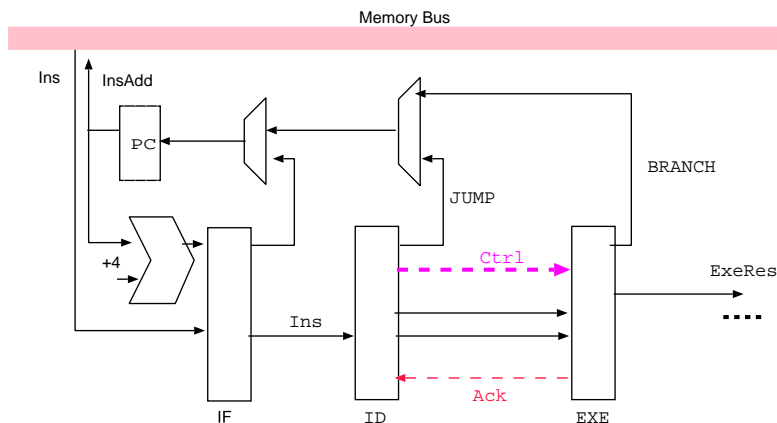


Fig. 11. Dealing with Control Hazards

of the processor at any particular moment. The colour bit changes every time a transfer of control takes place in the processor. Each instruction address issued to memory, carries the current operating colour of the processor, which will be used to mark the corresponding fetched instruction. When a control hazard occurs (branch or exception), the colour of the processor changes, causing a change in the colour of instructions subsequently fetched from the new target address. The colour bit of an instruction which arrives at the datapath for execution, is compared with the current colour of the processor. If a match is found, the instruction belongs to the current valid instruction stream and is thus executed, otherwise it is discarded. Thus, all the prefetched instructions following the hazard will be discarded until an instruction from the new valid instruction stream (i.e. the branch target) is encountered.

In AMULET1, the change of the processor colour, the occurrence of a control hazard with the generation of the new transfer address and the decision as to whether an instruction should be discarded (comparison of the respective colour bits) all take place in the same pipeline stage (the ALU). In MIPS however, control hazards may potentially occur in more than one stage. In particular, we consider the case where conditional branches are taken in the EXE stage while unconditional jumps are executed in ID stage. In this case, an improved technique is required, as due to the distributed nature of the system, it is not clear which stage should maintain the state colour bit, or how can a stage know that the colour has been changed by a different stage.

The solution that has been initially adopted is to have the state “colour” bit exchanged between the ID and EXE stages, piggybacking it onto the request bundle (ID to EXE, to inform EXE that a Jump has taken place) and the acknowledgement signal (EXE to ID, to inform ID that a branch has been taken) as illustrated in figure 11. The basic operation of the algorithm is as follows:

- If a Branch is taken: the decision is made at EXE, which changes its copy of the “colour” bit, issues the new Branch target address with the new “colour” (which via an arbiter will be sent to memory and to the PC unit), and sends an Acknowledgement back to ID, piggybacking the new colour.
- In the case of a Jump: the Jump is executed at the ID stage, a new target address is sent to memory, and the new “colour” will be sent to EXE with the next data bundle.

If a jump follows immediately after a branch, the algorithm would not work. However, this will never happen: Since the MIPS compiler will either insert a NOP instruction or some other instruction after the Branch.

An alternative, more generic, distributed colouring scheme has also been devised as part of our work and is currently being evaluated [27].

6 Summary

This paper has presented a Register Transfer Level design for an asynchronous implementation of the MIPS processor. It has described the distribution of control logic and has presented the techniques that have been devised to address data and control hazards. Future work will focus on developing these techniques further, model the system in Balsa, evaluate, improve and finally synthesise it. Ultimately, our goal is to use it as a test case to evaluate the integrated framework for formal verification and distributed simulation of asynchronous hardware, currently under development jointly at the Universities of Birmingham and Manchester.

References

1. *The AMULET Group*, URL: <http://www.cs.man.ac.uk/amulet/index.html>
2. *The Balsa Asynchronous Synthesis System*, URL: <http://www.cs.man.ac.uk/amulet/projects/balsa/>
3. G. Birtwistle, A. Davis, eds., *synchronous Digital Circuit Design*, Springer Verlag, 1995.
4. E. Brunvand, *The NSR Processor*, Proceedings of the 26th Annual Hawaii International Conference on System Sciences, Maui, Hawaii (1993), pp. 428-435.
5. K. R. Cho, K. Okura, K. Asada, *Design of a 32-bit Fully Asynchronous Microprocessor (FAM)*, Proceedings of the 35th Midwest Symposium on Circuits and Systems, Washington D.C. (1992), pp. 1500-1503.
6. M. E. Dean, *STRiP: A Self-Timed RISC Processor*, Technical Report CSL-TR-92-543, Computer Systems Laboratory, Stanford University, July 1992.
7. C. J. Elston, et al., *Hades - Towards the Design of an Asynchronous Superscalar Processor*, Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies, London(1995), pp. 200-209.
8. P.B. Endecott, *SCALP: A Superscalar Asynchronous Low-Power Processor*, PhD thesis, Dept. of Computer Science, Univ. of Manchester, 1995
9. S. B. Furber, *Computing Without Clocks*, In [3], pp. 211-262.

10. S. B. Furber, et. al., *AMULET2e: An Asynchronous Embedded Controller*, Proceedings of Async '97 Conference, IEEE Computer Society Press(1997), pp. 290-299.
11. J. D. Garside, et. al., *AMULET3 Revealed*, Proceedings of Async'99 Conference, IEEE Computer Society Press(1997), pp. 51-59.
12. D.A. Gilbert, J.D. Garside, *A result forwarding mechanism for asynchronous pipelined systems*, IEEE Proc.Int. Symp. Advanced Research in Asynchronous Circuits & Syst.,1997, pp 2-11
13. G. Kane, J. Heinrich, *MIPS RISC Architecture*, Prentice-Hall, 1992
14. A. J. Martin, et al., *Design of an Asynchronous Microprocessor*, Proceedings of the Decennial Caltech Conference on VLSI, Advanced Research in VLSI 1989, pp. 351-373.
15. A.J. Martin, A. lines, R. Manohar, M. Nystroem, et. al. *The Design of an Asynchronous MIPS R3000 Processor*, IEEE, IEEE Computer Society Press, 17th Conference on Advanced Research in VLSI, 1997, pp. 164-181
16. C. A. Mead, L. A. Conway, *Introduction to VLSI Systems* (Addison Wesley, 1980).
17. T. Nanya, et al., *TITAC: Design of a Quasi-delay-Insensitive Microprocessor*, IEEE Design and Test of Computers, 11(2)(1994), pp. 50-63.
18. D.A. Patterson, J.L. Hennessy, *Computer Organization & Design*, second edition, Morgan Kaufmman, 1998
19. N. C. Paver et al., *Register Locking in an Asynchronous Microprocessor*, Proceedings of ICCD 1992, October 1992, pp. 351-355.
20. A. M. G. Peeters. *Tangram99 talk*. In ACiD WG Workshop - University of Newcastle upon Tyne, UK. Edited by: M. B. Josephs and A. V. Yakovlev. Philips Research, 18-19 January 1999.
21. L.A. Plana, P.A. Riocreux, et. al. *SPA - A Synthesizable Amulet Core for Smartcard Applications*, Proceedings of Async'2002, pp. 201-210
22. W. F. Richardson, E. Brunvand, *Fred: An Architecture for a Self-Timed Decoupled Computer*, Technical Report UUCS-95-008, University of Utah, May 1995. Available at: <ftp://ftp.cs.utah.edu/techreports/1995/UUCS-95-008.ps.Z>
23. *Sharp's Data-Driven Media Processor*, URL: <http://www.sharpsdi.com/DDMPhtmlpages/DDMPmain.html>
24. R. F. Sproull, I. E. Sutherland, C. E. Molnar, *The Counterflow Pipeline Processor Architecture*, IEEE Design and Test of Computers, 11(3)(1994), pp. 48-59.
25. I. E. Sutherland *Micropipelines*, Communications of the ACM, 32 (1)(1989), pp. 720-738.
26. G. Theodoropoulos, *Modelling and Distributed Simulation of Asynchronous Hardware*, Simulation Practice and Theory Journal, Elsevier. (7) (2000) 741-767
27. G. Theodoropoulos, Q. Zhang *A Distributed Colouring Algorithm for Control Hazards in Asynchronous Pipelines*, submitted to the 36th International Symposium on Microarchitecture (MICRO-36), December 3-5, 2003, San Diego, CA, USA.
28. T. Werner, A. Venkatesh, *Asynchronous Processor Survey*, IEEE Computer, 30(11)(1997), pp. 67-76.
29. J.V. Woods, P. Day, S.B. Furber, J.D. Garside, N.C. Paver, and S. Temple, *AMULET1: An Asynchronous ARM Microprocessor*, IEEE Transactions on Computers 46 (4)(1997) pp.385-398.