A Distributed Colouring Algorithm for Control Hazards in Asynchronous Pipelines

Georgios Theodoropoulos and Qianyi Zhang
School of Computer Science, University of Birmingham
Edgbaston , Birmingham B15 2TT , UK
Email: {gkt, qyz}@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, recently, there has been 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. In a pipelined architecture, if a control hazard occurs, the prefetched instructions following a hazard must be discarded and removed from the pipeline before instructions from the new stream are executed. In an asynchronous microprocessor the exact number of the prefetched instructions is nondeterministic and unpredictable. 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. This paper will discuss a distributed, asynchronous technique for dealing with control hazards in asynchronous pipelines where control hazards may potentially occur in more than one stage.

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. 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.

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. Each sub-system operates at its own rate synchronising with its peers only when it needs to exchange information. This synchronisation is achieved by the communication protocol employed, which is typically in the form of local request and acknowledge 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. A number of asynchronous architectures have been developed [8] including two at CalTech, an early design, and a more recent asynchronous version of MIPS, NSR and Fred at the University of Utah, STRiP at Stanford University, Sun's Counterflow pipeline processor, FAM and TITAC at Tokyo University and Institute of Technology respectively, Hades at the University of Hertfordshire, Sharp's Data-Driven Media Processor and the series of asynchronous implementations of the ARM RISC processor (AMULET1, AMULET2e and AMULET3i) developed by the AMULET group at the University of Manchester.

This paper presents a technique for managing control hazards that may occur in asynchronous, pipelined processors. This technique was developed as part of our endeavour to develop an asynchronous implementation of the MIPS architecture (SAMIPS [9][10]), which in turn forms part of a wider collaborative funded research 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].

2 Control Hazards in Asynchronous Systems

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 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 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.

For instance, NSR avoids the problem by having control flow decisions being made by the Instruction Fetch Unit based on conditions set up in advance by the execution Unit (essentially there is no prefetching as such).

FRED (roughly based on NSR) utilises an Instruction Window and a Dispatch Unit to implement a two-phase branch model (address generating and sequence change)and avoid undoing prefetched instructions - exceptions are dealt with by tagging instructions and passing information to the Despatch Unit which is at the top of the pipeline.

Caltech's asynchronous MIPS makes use of a Decoder (immediately after the prefetching unit) which records the order of instructions and deals with branches - a Write Back (WB) unit can cancel instructions in the case of exceptions reconstructing the program order.

The Sun's Counterflow processor takes advantage of the two asynchronous pipelines running in opposite directions - hazard information flows backwards, invalidating prefetched instructions on its way.

A very neat and efficient solution was devised for the AMULET1 processor by the AMULET group at the University of Manchester. Their technique uses a single bit to "colour" of the state of the processor at any particular moment.

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 (figure 1). 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 prefeched instructions following the hazard will be discarded until an instruction from the new valid instruction stream (i.e. the branch target) is encountered.

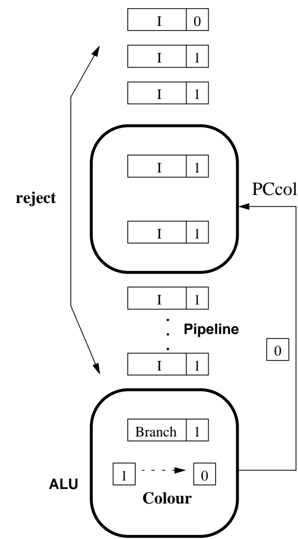

Figure 1. Colouring and Rejecting Instructions in AMULET1

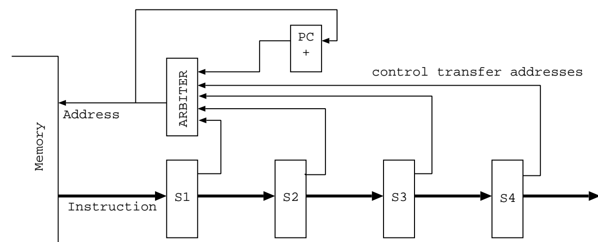

Figure 2. Control Hazards in Multiple Stages

3 The Need for a New Colouring Algorithm

AMULET1's colouring technique is particularly pertinent to our own work, as it does not require any complex hardware or any substantial change in the pipeline structure of the processor. A decision has therefore been made to adopt it for our asynchronous MIPS.

The 1-bit colouring technique works in AMULET1, as 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 dis-

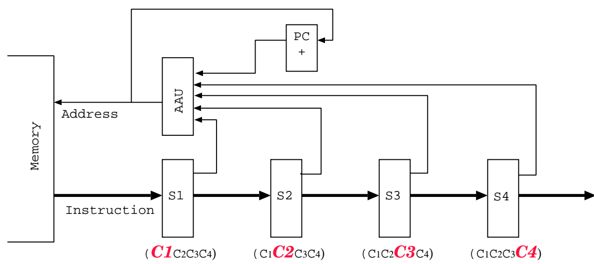

Figure 3. Pipeline Stages Colour State Vectors

carded (comparison of the respective colour bits) all take place in the same pipeline stage (the ALU).

If, however, the asynchronous architecture is such that control hazards can occur at different stages in the pipeline, the above mechanism which uses a single colour bit to define the state of the system is insufficient, as the operating colour of the system may be modified by more than one stage in a distributed, non-deterministic fashion. An example is MIPS, where control hazards may potentially occur in more than one stage (e.g. where conditional branches maybe taken in the EXE stage while unconditional jumps are executed in ID stage) [6].

This model is illustrated in figure 2 where control transfer addresses may be generated by any stage in the pipeline and they may arrive at the prefetching unit at any order.

The fundamental problem is that in an asynchronous, distributed system, global snapshots of the state of the system at any particular moment are not easily obtainable. If a single colour bit is used to hold the state of the processor, it is not clear where this bit should be maintained, or how a pipeline stage can know that the colour has been changed by a different stage, or how to adhere to causality and associate a change in colour with a particular hazard event. In this case, an improved technique is required to deal with the distributed, non-deterministic nature of the system. The rest of the paper describes such a technique.

4 A Generic Distributed Solution

The proposed solution is based on two fundamental observations:

- The state of the system is distributed.
- Stages that are deeper in the pipeline have higher priority than stages before them. In other words, a control transfer event that occurs at a pipeline stage renders other events that may occur in pipeline stages earlier in the pipeline irrelevant and invalid, event if the latter precede the former in time.

Based on the above two observations, in the proposed scheme the colour state of the processor at any particular

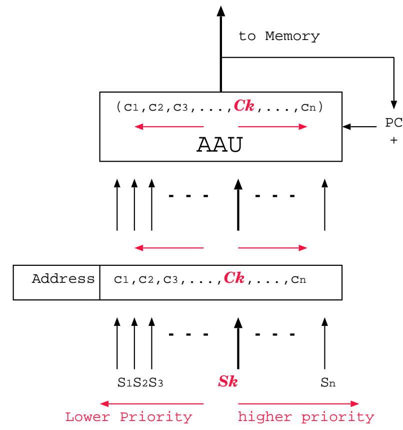

Figure 4. The Address Arbitration Unit

moment is defined as a vector $c = (c_1, c_2..., c_n)$ in the set C^n, where C is the set of colours $C = \{0, 1\}$, n is the number of stages in the pipeline and c_i is the colour of the stage i. Priority of $c_i >$ Priority of $c_j, i > j$.

Each pipeline stage S_k where control hazards may occur maintains a copy of the vector of the colour state $c = (c_1, c_2, c_k, ...c_n)$ but is in charge of managing only the element that corresponds to it (figure 3). Since target addresses may be generated at any time by any pipeline stage, this scheme assumes the existence of an arbitration unit, referred to as the Address Arbitration Unit - AAU) in the figure which is described in the following section.

4.1 The Address Arbitration Unit

The Address Arbitration Unit issues all instruction address information to memory, namely, sequential instruction addresses as they arrive from the Program Counter (normal operation) or from the pipeline stages (in the case when a control hazard occurs) as illustrated in figure 4.

In the case of control hazards, the role of the AAU is to let through to memory instruction addresses that are the result of high priority control hazards, while blocking any subsequent lower priority target addresses from reaching memory and thus interrupting the high priority instruction stream.

To achieve this, the AAU keeps a record of the colour state of the processor (vector c), which it updates based on the colour vectors of the instruction addresses arriving to it.

When a new transfer address arrives from stage S_k (see figure 4), the AAU checks the state vector carried by that address.

If all the high priority bits c_j (where $j > k$) in the transfer address vector are the same as the corresponding colour bits of the AAU, then this instruction belongs to the current stream (i.e. no other higher priority control hazard has taken place) and therefore the address is allowed through to memory. The colour state vector of the instruction becomes

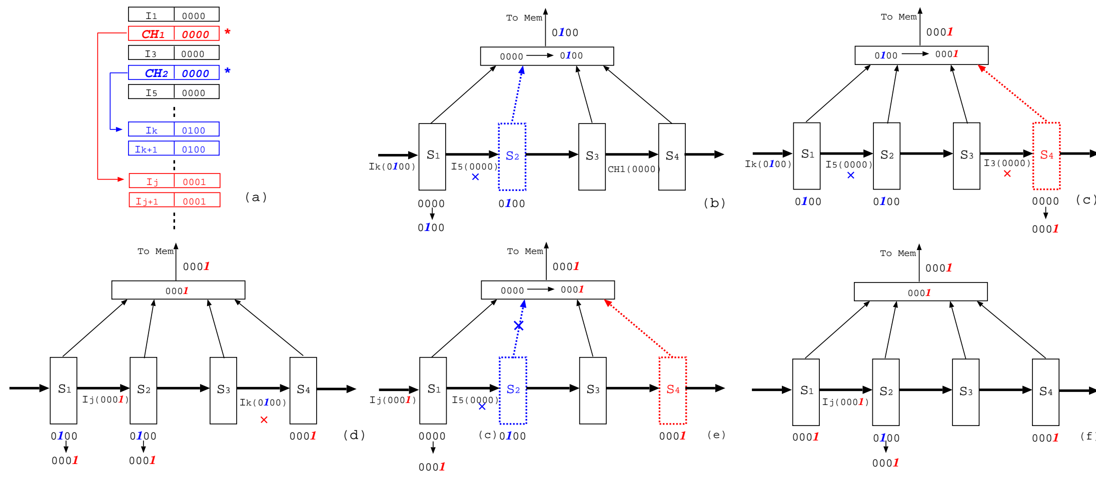

Figure 5. A Constructive Proof

now the state vector of the AAU too.

If any higher priority colour bit c_j (where $j > k$) in the transfer address vector is different than the corresponding colour bit of the AAU, that means that a higher priority control hazard has already taken place, and its target address has gone through to memory, and therefore the transfer address is rejected.

4.2 Functionality of a Pipeline Stage

For each new instruction that arrives for execution at a stage S_k(figure 3), its colour state vector is compared against the state vector of the stage. If any higher priority colour bit (c_j where $j > k$) in the instruction is different than the corresponding colour bit of the stage, that means that the instruction is the first of a transfer address as a result of a control hazard that has taken place deeper in the pipeline. Thus, the stage lets the instruction through (performing the required processing) and now the state vector of the instruction becomes its own.

If the stage's own colour bit (c_k) is different than the corresponding bit in the instruction vector then, this instruction is one of the instructions following an instruction that has already caused a control hazard in the stage, and therefore the instruction is rejected. Otherwise, the instruction is executed and the state vector of the instruction becomes its own.

5 A Constructive Proof

As a constructive proof for the proposed mechanism, figure 5 presents a possible scenario to illustrate how the proposed mechanism works.

In the example scenario, a program (figure 5a) is executed in a four stage asynchronous pipeline (figure 5b). The colour state vector consists of four values (bits) and is initially (0,0,0,0); this is the vector piggybacked on the instructions as they initially enter the pipeline. Two instructions may cause control hazards, namely CH1 and CH2 in stages S_4 and S_2 respectively. The control hazards may take place in any order, non-deterministically, depending on the order that CH1 and CH2 reach the respective stages (S_4 and S_2). Irrespectively of the order they occur, CH1 has higher priority than CH2 since it is deeper in the pipeline. Depending on the relative order of the occurrence of the two control hazards, two scenarios are distinguished.

In the first scenario, CH2 causes a control hazard before CH1 reaches S_4 (figure 5b). Then the operation proceeds based on the algorithm: the state vector of S_2 and AAU changes to (0,1,0,0), following instructions (e.g. I_5) are rejected at S_3 up to the point when I_k enters the pipeline whereupon the vector of S_1 changes too.

In the meantime, CH1 reaches S_4 and causes a control hazard there (figure 5c), changing the state vector of S_4 from (0,0,0,0) to (0,0,0,1). When the transfer address from S_4 reaches AAU, AAU will change its vector from (0,1,0,0) to (0,0,0,1), as a higher priority bit (4th) is different in the vector of the transfer address. Henceforth, any instruction that reaches S_4 with a lower priority vector than that of S_4 (namely, (0,0,0,0) from the original stream, or(0,1,0,0) from the CH2 target address, e.g. I_k) will be rejected. Finally, I_j, the first instruction from the CH1 transfer address, will enter the pipeline, changing the all state vectors to that of S_4 (0,0,0,1) (figure 5d).

In the second scenario CH1 reaches S_4 and causes a hazard before CH2 reaches S_2, thus changing AAU vector to (0,0,0,1) (figure 5e). When CH2 reaches S_2 and causes a control hazard, the transfer address that the S_2 will issue, will be rejected by AAU as the vector of the former has lower priority than that of the latter.

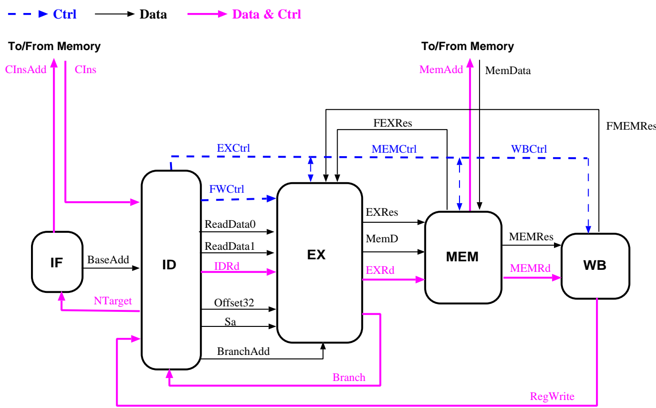

Figure 6. BAL-SAMIPS Top level Process Graph

```
import[type]

procedure S1(input InsIn: Ins;
             output InsOut:Ins;
             output NTAdd: NTAdd) --if a control hazard happens
is local variable Ins_R: Ins        NTAdd is output
         variable Col_S1: Col     --if the colour is matched or high
begin                               priority colour bit is different,
 loop                               the current Ins should be executed
  Ins -> Ins_R;                     in S1

  if ((Ins_R.Col = Col_S1) or (Ins_R.Col.S2 /= Col_S1.S2) or
      (Ins_R.Col.S3 /= Col_S1.S3) or (Ins_R.Col.S4 /= Col_S1.S4)) then
     Col_S1:= Ins_R.Col;          --if the current Ins cuase CH1,change
     if Ins_R.Code = CH1 then       first colour bit of S1 and sent out
        Col_S1.S1:= not Col_S1.S1;  NTAdd(assumed to be 0x08 here
        NTAdd <- {S1, {8, Col_S1}}
     else InsOut <- Ins_R   end   --otherwise pass it to S2
  end
 end                                                              (S1)
end

if ((Ins_R.Col = Col_S2) or (Ins_R.Col.S3 /= Col_S2.S3) or
    (Ins_R.Col.S4 /= Col_S2.S4) or
    ((Ins_R.Col.S1 /= Col_S2.S1) and (Ins_R.Col.S2 = Col_S2.S2))) then
                                                                  (S2)

if ((Ins_R.Col = Col_S4) or (Ins_R.Col.S4 = Col_S4.S4)) then
                                                                  (S4)

begin
Arb(NTAdd1, NTAdd2, NTAdda) ||         --a binary arbiter tree is used
Arb(NTAdd3, NTAdd4, NTAddb) ||
Arb(NTAdda, NTAddb, NTAdd)  ||
...
if (((NTAdd_R.Stage = S1) and (NTAdd_R.InsAdd.Col.S2 = Col_Arb.S2) and
     (NTAdd_R.InsAdd.Col.S3 = Col_Arb.S3) and
     (NTAdd_R.InsAdd.Col.S4 = Col_Arb.S4))
 or ((NTAdd_R.Stage = S2) and (NTAdd_R.InsAdd.Col.S3 = Col_Arb.S3) and
     (NTAdd_R.InsAdd.Col.S4 = Col_Arb.S4))
 or ((NTAdd_R.Stage = S3) and (NTAdd_R.InsAdd.Col.S4 = Col_Arb.S4))
 or (NTAdd_R.Stage = S4)) then         -- check the validity of NTAdd
    Col_Arb := NTAdd_R.InsAdd.Col ||   -- change the colour
    InsAdd <- NTAdd_R.InsAdd ||        -- send out new address
    Npc <- NTAdd_R.InsAdd
else                                   -- check the validity of pc
    if pc_R.Col=Col_Arb then InsAdd <- pc_R ||
                             Npc <- pc_R end
end
...                                                              (AAU)
end
```

Figure 7. The Balsa Model

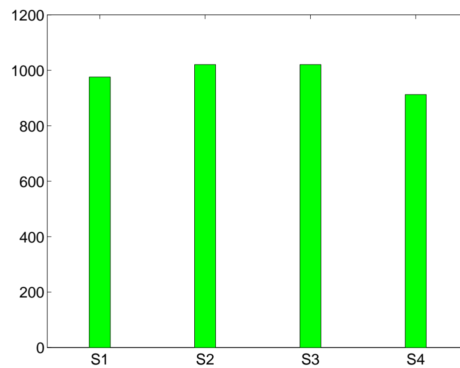

Figure 8. Cost Estimation of Stages

6 Balsa and SAMIPS

This colouring algorithm has been implemented and is currently being evaluated using Balsa system [3], an asynchronous synthesis toolkit developed by AMULET group.

Balsa is based on CSP. It uses CSP-based constructs to express Register Transfer Level design descriptions in terms of channel communications and fine grain concurrent and sequential process decomposition. Descriptions of designs (*.balsa* file) are then translated (***balsa-c***) into implementations in a syntax directed-fashion with language constructs being mapped into networks of parameterised instances of "handshake components" (*.breeze* file) each of which has a concrete gate level implementation [4]. ***balsa-netlist*** automatically generates CAD native netlist files (Avant, Xilinx Alliance FPGA or Cadence), which can then be fed into the commercial CAD tools that further synthesize the netlist to the fabricable layout. Three levels of simulation are supported. Balsa has a simple behavioural simulator while memory related behavioural simulation relies on the LARD [5] toolkit. ***breeze2lard*** tool helps to translate the *.breeze* file to a LARD simulation model. The native simulators of those commercial CAD tools carry out the other two low levels simulation.

As mentioned in the Introduction, the proposed distributed colouring algorithm was developed as part of our effort to design SAMIPS, an asynchronous implementation of the MIPS processor [9]. A Balsa model of SAMPIS as been developed (BAL-SAMIPS [10]) and is depicted in figure 6. SAMIPS adheres to the five-stage pipeline Datapath of the synchronous MIPS processor which comprises the following stages: Instruction Fetch (IF), Decode/Register File Read (ID), Execution or Address Calculation (EX), Memory Access (MEM) and Register Write-back (WB). Control Hazards may occur in ID, EX or WB stages.

7 Evaluation and Results

To evaluate the proposed algorithm we have developed a Balsa model of 4-stage pipeline as depicted in figure 7. The

model consists of 5 parallel processes, one for each of the 4 stages and one for the AAU module.

Figure 7 (S_1) shows the Balsa description of S_1. The modelling of the rest of the stages is similar, the only difference is in their respective implementation of their instruction validity checking statement (IF statement) since as we move from stage to stage, different number of bits have to be checked. Interestingly, all the "middle" stages make use of exactly the same IF statement while the two "ends" of the pipeline are different. Figure 7 (AAU) shows the modelling of the AAU. A set of arbiters is used to make an arbitrated nondeterministic choice amongst the five input channels. As illustrated in previous section, AAU also has a validity checking statement for the incoming address and the new addresses with matched colour are accepted.

Figures 8 and 9 illustrate the area costs obtained from the Balsa (***breeze-cost***) utility. The costs provided by breeze-cost are only guideline figures and depend on particular back-end implementation. The units presented in the figure are linear microns of standard cells based on an old 1um library with a cell pitch of 37.5um and a typical density of about 2/3 cells, 1/3 routing.

Figure 8 shows the costs estimation for the 4 pipeline stages. The difference in costs are due to the different IF statements, as explained above. Figure 9 is meant to show the overhead in terms of space, that would be introduced in SAMIPS if the proposed distributed colouring algorithm was to be incorporated in it. It compares the costs of the extra functionality required for each stage (average figure of costs in figure 8) and the AAU with the rest of the main stages of SAMIPS. Clearly the overheads introduced are insignificant in comparison with the rest of the processor.

8 Summary and Further Work

This paper has presented a distributed colouring algorithm for dealing with control hazards in asynchronous pipelines. The main advantage of this technique is that it provides flexibility in designing the pipeline of the processor, enabling prefetching at any depth. The overhead it introduces is the AAU unit, comparison circuitry in the stages and extra control bits (for the colour). However, as the results obtained from Balsa indicate, this overhead, in terms of space is not significant. Future work will evaluate the performance of this approach and the overhead it imposes in terms of time and power. We are currently incorporating this technique in the asynchronous version of MIPS which is under development.

Acknowledgements

We would like to thank the members of the AMULET group, and in particular Andrew Bardsley and Doug Edwards for their invaluable advice and help. The research is funded by EPSRC and the School of Computer Science, University of Birmingham (ORS)[2].

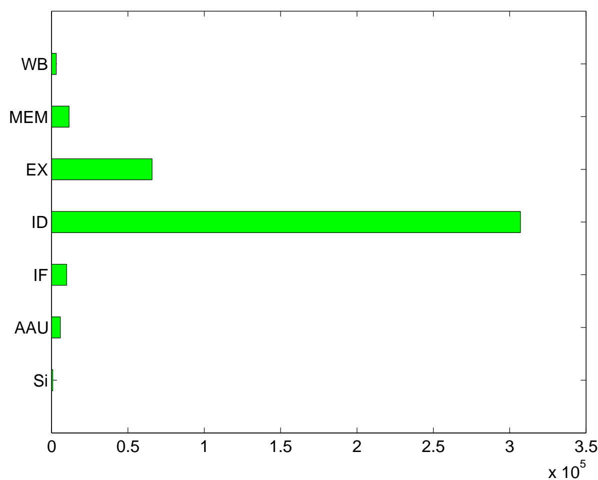

Figure 9. Comparison with SAMIPS Costs

References

[1] The AMULET Group, `http://www.cs.man.ac.uk/amulet/index.html`

[2] An Integrated Framework for Distributed Simulation and Formal Verification of Asynchronous Hardware, EPSRC Project No. GR/S1109/01 & GRS11084/01 `http://www.cs.bham.ac.uk/research/parlard/`

[3] The Balsa Asynchronous Synthesis System, `http://www.cs.man.ac.uk/apt/projects/Balsa/index.html`

[4] Berkel, K. van, *Handshake circuits - an Asynchronous Architecture for VLSI Programming*, Cambridge International Series on Parallel Computers, Cambridge University Press, Cambridge, 1993

[5] LARD Documentation Home Page, `http://www.cs.man.ac.uk/apt/projects/lard/index.html`

[6] Patterson, D.A., Hennessy, J.L., *Computer Organization & Design*, second edition,Morgan Kaufmam, 1997

[7] Sutherland, I. E., "Micropipelines", Communications of the ACM, 32 (1)(1989), pp. 720-738.

[8] Werner, T., Venkatesh, A., *Asynchronous Processor Survey*, IEEE Computer, 30(11)(1997), pp. 67-76.

[9] Zhang, Q., Theodoropoulos, G., "Towards an Asynchronous MIPS Processor", The Eighth Asia-Pacific Computer Systems Architecture Conference, (ACSAC'2003), Aizu-Wakamatsu City, Japan, September 23 - 26, 2003.

[10] Zhang, Q., Theodoropoulos, G., "Modelling SAMIPS: A Synthesisable Asynchronous MIPS Processor", submitted to the 37th Annual Simulation Symposium (IEEE/ACM/SCS), Part of the Advanced Simulation Technologies Conference (ASTC 2004), Hyatt Regency Crystal City Arlington, VA, April 18 - 22, 2004