AμFLIPS: An Asynchronous Microprocessor With FLexIbly-timed Pipeline Stages

Zaheer Tabassam
Institute for Computer Engineering
TU Wien
Vienna, Austria
zaheer.tabassam@tuwien.ac.at

Syed Rameez Naqvi
Department of Electrical and Computer Engineering
CUI, Wah Campus
Wah Cantonment, Pakistan
rameeznaqvi@ciitwah.edu.pk

Andreas Steininger
Institute for Computer Engineering
TU Wien
Vienna, Austria
steininger@ecs.tuwien.ac.at

***Abstract*—AμFLIPS is an asynchronous microprocessor with novel pipeline register organization to resolve data and control hazards using synchronous hazard resolving schemes. Existing works claim that mechanisms for handling data and control hazards in synchronous systems are not directly applicable to asynchronous pipelined processors, because of distributed control nature of the latter. As a result of that, most asynchronous equivalents of MIPS propose novel hazard resolution methods, adding an overhead in terms of performance and complexity. In this work, we build a counter narrative by proposing a novel pipelined register organization that maintains synchrony with a flexible clock generator instead of the rigid clock, which also allow us to utilize the synchronous hazard resolving methods. Our simulation results – using Balsa – suggest 20.8% improvement in execution time as compared to one of the existing asynchronous processors.**

I. Introduction

Asynchronous logic from the last three decades has been adopted by the research community because of its remarkable properties like high operating speed, low electromagnetic emission, low power consumption, robustness towards process, voltage and temperature variations, no clock skew and distribution [1]. Efficient asynchronous implementation compared to synchronous is one answer to the challenges posed by ”global clock” in VLSI design. In this work our focus is a microprocessor as it is the principal part of a computer system where we only deal with pipelined versions of these. Major asynchronous microprocessor implementations adopt existing synchronous processors as benchmark, so they compare their performance and other parameters in a clear manner or because it’s an easy way to start.

But it’s not as simple as it looks. Replacing the global clock with an asynchronous handshake does not suffice, as all pipeline stages now start working concurrently. In synchronous microprocessors all stages share the common clock, so every stage without any doubt knows the state of the other ones. In a five stage pipelined microprocessor, e.g., when stage 1 is executing the fifth instruction it’s not hazardous by this stage to assume stage 3 is executing the third instruction.

This research was partially supported by the project ENROL (grant I 3485-N31) of the Austrian Science Fund (FWF). and Pakistan Science Foundation under grant number PSF/Res/P-CIIT/Engg (159).

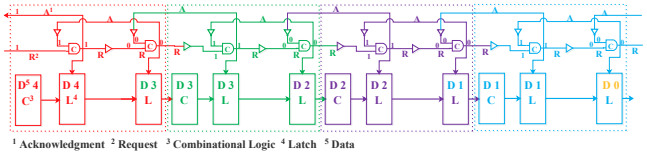

Fig. 1. 4-phase bundled data pipeline

But in an asynchronous pipeline microprocessor, each stage is only aware of its adjacent stage in a sense that it safely hands over the data when it’s ready to accept. There may be a case when stage 1 is executing the fifth instruction and stage 3 is executing the fourth one or may be done with that and containing nothing meaningful. So, this distributed nature of asynchronous logic does not allow to simply port synchronous hazard resolving schemes directly (as synchronous schemes utilize the property of maintaining the sequence of instructions with respect to pipeline stages). As a result, most asynchronous implementations proposed novel methods, adding an overhead in terms of performance and complexity.

In this work we realize an asynchronous microprocessor in a manner that we not only use the basic architecture of the synchronous benchmark, but we also make it compatible with synchronous hazard resolving schemes to check its performance.

So, we propose AμFLIPS, that is based on a novel asynchronous pipelined organization, in which buffers are controlled with a locally generated flexible clock, which helps in resolving hazards in a manner identical to the synchronous MIPS [2]. AμFLIPS is designed and implemented using Balsa (asynchronous circuit synthesis system and language) [3] where a Balsa generated netlist (suitable for Xilinx gate arrays) is verified using Xilinx EDA as well.

II. Background and Related Work

A. Asynchronous Pipeline

Among the popular asynchronous pipelines asP* [4], GasP [5], mousetrap [6], micropipelines [7], QDI [8], [9], RAMP [10], surfing [11], wave [12], [13] most have a Muller pipeline [14] as their foundation. A simple arrangement of

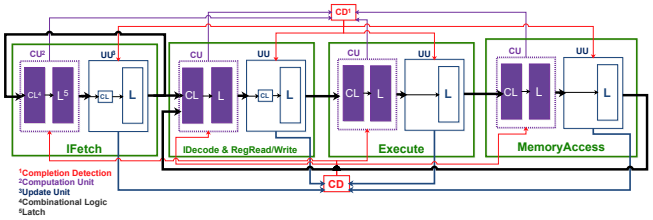

Fig. 2. Abstract View of AμFLIPS

Muller C-elements (MCEs) [15], such that each of them forms a single stage is a Muller pipeline.

Figure 1 illustrates a 4-phase bundled data pipeline [14] with datapath. One can observe that each pair of successive MCEs holds alternating logic. So in a filled pipeline only one latch is enabled per stage, while the other one holds the last value for next stage computation logic. This pipeline configuration is just like a synchronous master-slave design [14].

B. Asynchronous Microprocessors

The global clock in synchronous processors may become problematic when the processing environment is more complex. The crucial issues faced in a synchronous systems are the clock skew and the clock circuit itself [16]. The latter dissipates an enormous amount of power [17]. An alternative considered by the research community is asynchronous logic. In an asynchronous design each function unit communicates with others by using a local clock or more technically by handshaking. Such a design choice delivers simplified interfacing. Asynchronous processors are efficient in power dissipation as only the required part of the circuit is alive. In this section, the design of AsynRISC [18] is presented in detail however other important works, such as Caltech [19], [20], [21] and AMULET [22], [23], [24] series may be found in the survey [25]. The choice of studying AsynRISC [18] in preference to others was made because it is more relevant to our work in two aspects: 1) it is based on the MIPS R2000 architecture as our proposed work. 2) it is designed and implemented using the same EDA tool. It is an asynchronous pipelined processor with instruction set similar to MIPS R2000.

AsynRISC has five pipeline stages, namely instruction fetch (IF), instruction decode with register fetch (ID), instruction execute or memory address calculation (EXE), memory access (MEM) and register write back (WB). Control hazards are solved using two one-bit registers $instcolor$ (in IF) and $syscolor$ (in EXE). As all control transfer takes place in EXE, every new instruction from IF proceeds to the next stage after attaching color bit of the $instcolor$ register. The color bit is later checked by EXE by matching the color bit with the $syscolor$ register. If the color bits match, the instruction is executed normally, otherwise, it is discarded.

Data hazards are resolved by adding two extra fields in every general-purpose register. A pending bit indicates the register is up-to-date or waiting for new contents. Two bits, known as pending instruction index, record the instruction that produces the new contents for a register. At IF, the 2-bit instruction index register provides an index to every new instruction. As at most four instructions reside in the data-path, two bits are enough for this register. It was designed and verified using the Balsa asynchronous hardware description language and Balsa simulation system, respectively. The performance was measured by executing a particular program having 500 dynamic instructions taking 21256799 unit [1] execution time.

III. AμFLIPS

To realize an asynchronous microprocessor with hazard resolution from the synchronous MIPS [2], we utilize the flexible timing property of the asynchronous design style in a novel fashion. We changed the pipeline organization in a manner that makes its operation more similar to synchronous – thus allowing the use of synchronous hazard resolution techniques, while still maintaining the closed loop timing control, which is a core and beneficial property of the asynchronous paradigm, as it guarantees timing flexibility. In particular, local handshake signals are collected by a global completion detection unit which then generates a global handshake signal.

A. Abstract View of AμFLIPS

In fig. 1 each dotted block represents one pipeline stage. Each stage has two latches, whose control signals are at opposite levels when the pipeline is filled. In fig. 2 each stage of fig.1 is replaced with one microprocessor pipeline stage as shown, where the control signals are generated by completion detector (CD) units. All Computation Units (CUs) operate concurrently, and each of them sends a completion signal to the CD. Upon receiving the completion signals from all CUs, the CD activates the Update Units (UUs) where all Update registers (UR) are updated with new contents. Next, the UUs send their completion signals to their CD in the same manner as the CUs. Using this pipeline organization, we can apply synchronous hazard handling schemes with only little modifications, while maintaining timing flexibility.

B. AμFLIPS Complete Pipeline Operation

Figure 3 is the overall pipeline structure of AμFLIPS. In this section it will become clearer why each stage requires two latches and how we utilize these to resolve hazards. In each pipeline stage, purple color (dotted outline) latches are Computation registers (CR), and dark blue outline latches are UR. When CR are in write mode the UR are in read mode, so the next stage uses UR contents for its computation. In contrast to AμFLIPS, in a normal asynchronous microprocessor each pipeline stage works concurrently. For example, an instruction is fetched from the Instruction Cache (IC) and stored in CR as soon as UR received acknowledgement from ID, the contents are copied from CR to UR and *IF* is ready to fetch a new instruction. In this scenario control is distributed and hazards are resolved by using different techniques, far from synchronous schemes that are much simpler and more convenient. Here we build a counter narrative by proposing a

[1]As unit is not mentioned, the assumed unit is ns.

novel pipelined asynchronous register organization where we enable the registers to use a flexible clock signal.

(i) CR: An instruction is fetched from the IC and stored in *InstregIF*. When all registers complete their read operation in the *IF* Computation Unit (CU), the completion signal from this stage is sent to the CD that reacts after receiving the completion signals from all CUs (other pipeline stages).

(ii) In the next cycle only contents of the CR are updated to UR with some computations for resolving hazards.

The following subsections elaborate those pipeline stages, where during the Computation Cycle (CC) all UR (blue outline) are in read mode while in the Update Cycle (UC) all CR (purple color) are in read mode.

1) Instruction Fetch: In *IF* (fig. 3), CU activity includes: (i) fetching an instruction from the IC, (ii) storing it into *InstregIF*, (iii) incrementing the program counter by adding 1 to the *ADDRESS* register, and (iv) storing it into the *ADDRESS add*. These activities are controlled by a controller named Hazard, Branch and Jump Detection Unit (HDU). For simplification, we place one HDU in *ID* in fig. 3, but both stages have their own HDU. If any type of hazard occurs, this unit blocks a new instruction fetch by maintaining the previous instruction in *InstregIF* and updating *ADDRESS add* with an appropriate address. These decisions are taken by the HDU after reading specific UR and checking the predefined criteria.

As the CD ensures that the Update Unit (UU) are activated after the CU completed their tasks, now a *"Control"* unit in *IF* checks the decoded control signals (branch, jump, and branch on equal etc.) on whose basis the *"Control"* unit generates a signal that serves as select line for two multiplexers and one ALU. The first multiplexer selects the appropriate next program counter address, and the second one decides whether *Instreg read* is updated with the fetched instruction or No Operation (NOP).

2) Instruction Decode: In the *ID* unit the HDU first checks some prerequisite condition using the newly fetched instruction from *Instreg read* and from the previously decoded, executed instructions contents that are stored in UR. If any hazard is detected or branch committed, all computation registers are loaded with a NOP, and if no such condition is met, the normal flow continues. The UU of ID also includes a *Forwarding Unit* that checks whether the *rs (Rs_IDEX_F)* and *rt (Rt_IDEX_F)* register value fetched are updated. If not, it updates the register A and B with updated contents.

3) Instruction Execute, Memory Access and Register Write: As AμFLIPS is realized in a fashion, which, in an abstract view, equals that of the synchronous MIPS [2], abstract operations of these last three stages are like those in the conventional MIPS processor. It's also important that the Register File (RF) operates on read-after-write sequence, so in normal operation no hazard can occur in reading the RF – compatible with MIPS.

4) Register File Reading and Writing: The RF read/write function in AμFLIPS is described in fig. 4. Whenever *case fetch* receives a request from the output register, it fetches

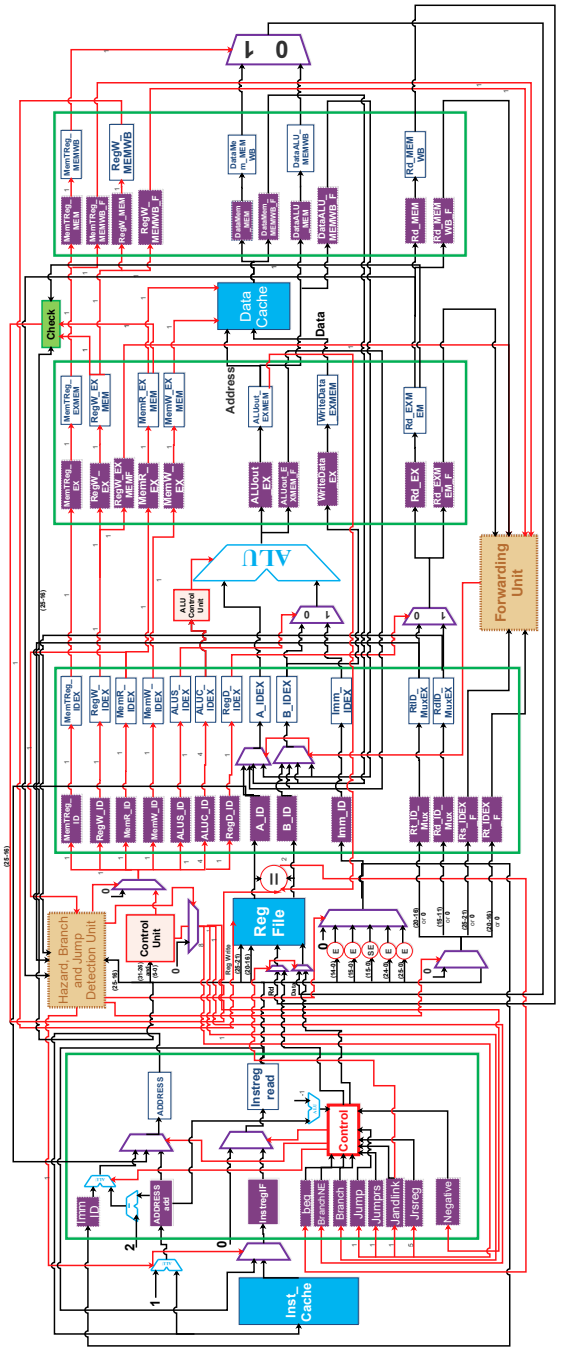

Fig. 3. AμFLIPS Complete Pipeline Structure

data from the respective register(x), where the "x" is the index provided by the index register, and grants data to the *output register*. The write function is initiated by the *activate* signal to the *address* module. The *address* module determines to which register the *request* signal for writing is issued, where the *F* block is a connector. To reduce complexity, the *sequence* block is not included in fig. 4. Its purpose is to guarantee that read is performed after write.

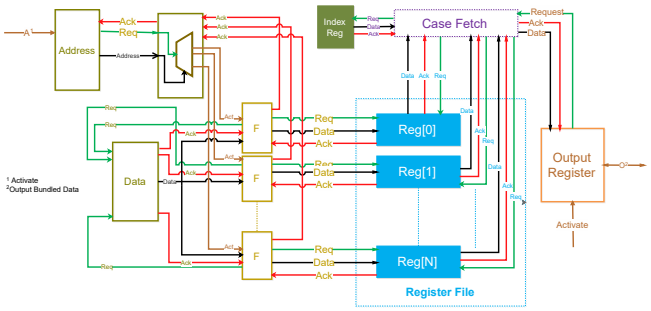

Fig. 4. Register File Writing and Reading

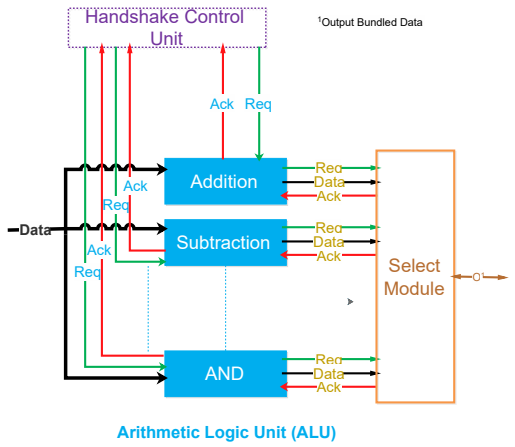

Fig. 5. Arithmetic Logic Unit

5) Arithmetic Logic Unit (ALU): ALU operation in AμFLIPS is illustrated in fig. 5, where the computational unit requested computes the input. The output is only passed if the same unit is requested by the Handshake Control Unit. If so, it passes data and request to the Select Module .

C. Novel Hazard Resolving Method

Algorithm 1 is a simplified description of AμFLIPS's HDU function. The novelty is not in the algorithm but in its implementation using asynchronous logic. According to the algorithm, an instruction is fetched and decoded only when some predefined criterion is met.

1) Control hazard: Fig. 6 demonstrates in a simplified view how AμFLIPS deals with control hazards:

(i) In CC 2 (Computation 2) instruction 22 enters the pipeline .

(ii) In UC 2 (Update 2) the HDU detects that instruction 22 depends on the previous instruction 21 that is not completed yet.

(iii) In CC 3 the HDU retains instruction 22 at IF, while inserting one *NOP* at ID, and while instruction 21 is in EXE. For conditional branches, a comparator in ID compares contents of the selected registers.

Algorithm 1: Hazard Resolving Method

Comment: $\wedge$ is logical AND, and $\vee$ is logical OR

if *(MemRIDEX = 1 $\wedge$ (Instregread[25:21] = RtIDMuxEX $\vee$ Instregread[20:16] = RtIDMuxEX)) $\vee$ (RegWIDEX = 1 $\wedge$ (Instregread[31:26] = 4 $\vee$ Instregread[31:26] = 5) $\wedge$ (Instregread[25:21] = RdIDMuxEX $\vee$ Instregread[20:16] = RdIDMuxEX)) $\vee$ (MemREXMEM = 1 $\wedge$ (Instregread[31:26] = 4 $\vee$ Instregread[31:26] = 5) $\wedge$ (Instregread[25:21] = RdEXMEM $\vee$ Instregread[20:16] = RdEXMEM))*

then

ADDRESSadd=ADDRESS;
InstregIF=Instregread;
NOP generation;

else

ADDRESSadd=ADDRESS+1;
InstregIF=Instructioncache(ADDRESS);
Decode Instruction;

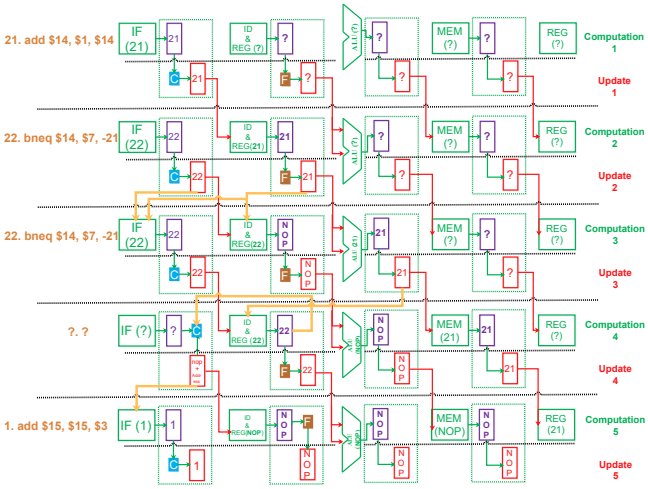

Fig. 6. Resolving Control Hazard in AμFLIPS

(iv) Here the scenario is different: instruction 22 is decoded during CC 4, while in the previous cycle CC 3 contents of instruction 21 are finalized by EXE. These contents are ready to use but not written back to the destination register, so they need to be forwarded from UR for the compare operation.

(v) After checking conditions, ID raises the branch flag in CC 4. So in UC 4 the newly fetched instruction is discarded, and the target instruction address is updated.

(vi) In CC 5 the target instruction is fetched, as the overall scenario is clear now.

This explanation clarifies that by introducing the global handshake we are able to utilize the update cycle to resolve control hazards in the same manner as the synchronous MIPS does.

2) Data hazard: Fig. 7 shows how AμFLIPS deals with data hazards.

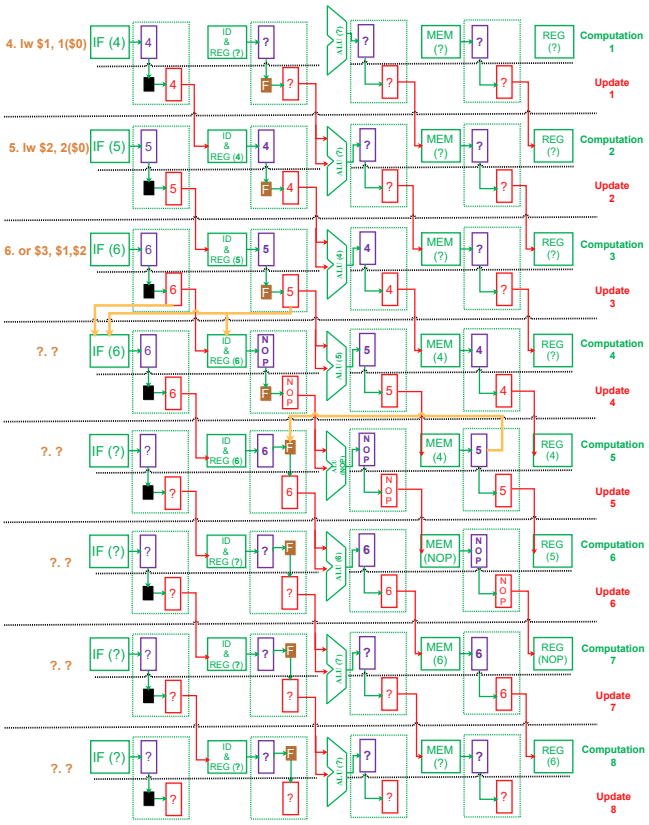

Fig. 7. Resolving Data Hazard in AμFLIPS

(i) As visible in fig. 7 instruction 6 depends on instructions 4 and 5 which are load word instructions and completed after reading the Data Cache (DC). So, here a data hazard occurs when we decode instruction 6.

(ii) Just consider instruction 4 enters pipeline in CC 1 and instruction 5 in CC 2.

(iii) In CC 3, instruction 6 gets fetched and in the very next UC 3 the HDU detects its dependence on contents that are not ready yet.

(iv) Instructions 5 and 4 are at ID and EXE respectively, so we must retain instruction 6 and generate a NOP at the respective locations, as we are not able to forward or directly access the values updated by instructions 4 and 5.

(v) In CC 5, instruction 5 reads the respective DC location where during this cycle instruction 4 writes its contents to the RF.

(vi) In UC 5, we have all values available: instruction 4's contents are already in the RF and instruction 5's contents can be obtained via forwarding.

(vii) In CC 6 we can execute instruction 6 that is then completed in CC 8 as shown.

These examples illustrate the architectural simplicity and the efficient, straightforward handling of data hazards.

Algorithm 2: Fibonacci series

Input : Index
Output: Return
fibo(Index)
if *Index* $\leq$ *1* **then**
  Return = Index;
else
  Return = (fibo(Index-1)+fibo(Index-2));

Algorithm 3: Executing "n" dynamic Instructions

Input: n
while *Instruction_count* $<$ *n* **do**
  mixed logical, arithmetic load-store instructions (each has some type of dependency, so hazards occur randomly)
  Instruction_count = executed_instructions

IV. Evaluation

In our evaluation process we use two different applications as benchmark and for functionality test, namely a Fibonacci series and a code executing one million instructions. To make performance comparison clear with set benchmark microprocessor we also note the execution time of 500 instructions by restricting second one to five hundred instructions.

We generate the Fibonacci series using the recursive method presented by Algorithm. 2 where Table I presents the execution time of generating a series with index 9, that is [0,1,1,2,3,5,8,13,21,34]. In Algorithm. 2 the "fibo" function is called recursively to complete the result. By doing so, all types of conditional and unconditional branches, jump and link instructions, arithmetic and load/store instructions are used, which gives evidence of the proper function of the microprocessor.

While in general, the performance of a microprocessor is measured in million instructions per second, in our assessment, we complete 1 million instructions by using a simple "While loop" with different instructions as presented in Algorithm.3. The loop includes all instructions from the instruction set to make fair distribution. Table I lists the execution time for n = 1M and n = 500.

The presented time scale is in nano seconds and the numbers originate from a behavioral simulation of Breeze Components [3]. That is the reason why at this stage we are only comparing our proposed work with AsyncRISC [18] whose results are presented in the same way. In our future work, we are engaged to port AμFLIPS to a commercially available FPGA (as Balsa allows to implement design in different technologies such as Xilinx) to facilitate direct comparison with other state of the art microprocessors.

V. Comparison and Area Cost

When comparing with the AsyncRISC [18], the results presented in Table II suggest that our architecture exhibits a

TABLE I
EXECUTING TIME OF DIFFERENT CODES RUNNING USING AμFLIPS

Code	Execution Time
Fibonacci series with Index 9	40469000ns
>1M dynamic Instruction	33709952800ns
>500 dynamic Instruction	16854976ns

TABLE II
COMPARISON BETWEEN PROPOSED AμFLIPS AND ASYNRISC [18]

Processor	Execution Time	Breeze cost
AsyncRISC	21256799ns	404920.25
AμFLIPS	16854976ns	689916 Including RF, DC and IC

20.8% reduced execution time (when executing 500 dynamic instructions). This can be mainly attributed to our decision concerning the control transfer. In AsyncRISC all control transfer takes place in EXE, while AμFLIPS is flexible in these decisions. After checking possible conditions, it changes control flow in IF, while remaining transactions are cleared in ID, as presented in section III. Another important contributing factor is the fast forwarding of data in case of data hazards, as discussed in section III-C2.

As we designed and verifyied AμFLIPS using Balsa, the simulation results presented throughout this article are carried out using Breeze components [3]. The Breeze cost comparison is presented in Table II.

VI. CONCLUSION

After surveying asynchronous microprocessors [25] in a comprehensive manner, we came to the conclusion that many asynchronous microprocessors are asynchronous implementations of a synchronous benchmark microprocessor, while some others realized a novel architecture for some special purpose. Common to all implementations is that they target low power and other known benefits of asynchronous logic. While high-throughput pipelining is essential in modern microprocessors, it creates hazards. In asynchronous logic, these hazards are not addressable in the same simple manner as in synchronous logic, because of its higher concurrency, so major existing asynchronous microprocessors are realized with novel hazard resolving methods. In this work we realized an asynchronous microprocessor with a novel pipeline register organization that facilitated resolving data and control hazards in a simple and efficient way. We have given evidence for the correctness of our approach as well as for its beneficial properties through a detailed analysis of its operation, as well as through extensive simulation of benchmark applications.

REFERENCES

[1] J. Sparsø, *Introduction to Asynchronous Circuit Design.* DTU Compute, Technical University of Denmark, 2020, paperback edition available here: https://www.amazon.com/dp/B08BF2PFLN.

[2] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Newnes, 2013.

[3] D. Edwards, A. Bardsley, L. Janin, L. Plana, and W. Toms, "Balsa: A tutorial guide." *The University of Manchester*, 2006.

[4] C. E. Molnar, I. W. Jones, W. S. Coates, J. K. Lexau, S. M. Fairbanks, and I. E. Sutherland, "Two fifo ring performance experiments," *Proceedings of the IEEE*, vol. 87, no. 2, pp. 297–307, 1999.

[5] I. Sutherland and S. Fairbanks, "Gasp: A minimal fifo control," in *async*. IEEE, 2001, p. 46.

[6] M. Singh and S. M. Nowick, "Mousetrap: Ultra-high-speed transition-signaling asynchronous pipelines," in *iccd*. IEEE, 2001, p. 0009.

[7] I. E. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, no. 6, pp. 720–738, 1989.

[8] A. M. Lines, *Pipelined asynchronous circuits*, California Institute of Technology, 1998.

[9] R. O. Ozdag and P. A. Beerel, "High-speed qdi asynchronous pipelines," in *Asynchronous Circuits and Systems, 2002. Proceedings. Eighth International Symposium on*. IEEE, 2002, pp. 13–22.

[10] S. R. Naqvi, J. Lechner, and A. Steininger, "Protection of muller-pipelines from transient faults," in *Quality Electronic Design (ISQED), 2014 15th International Symposium on*. IEEE, 2014, pp. 123–131.

[11] B. D. Winters and M. R. Greenstreet, "A negative-overhead, self-timed pipeline," in *Asynchronous Circuits and Systems, 2002. Proceedings. Eighth International Symposium on*. IEEE, 2002, pp. 37–46.

[12] W. P. Burleson, M. Ciesielski, F. Klass, and W. Liu, "Wave-pipelining: a tutorial and research survey," *IEEE Transactions on very large scale integration (vlsi) systems*, vol. 6, no. 3, pp. 464–474, 1998.

[13] O. Hauck and S. Huss, "Asynchronous wave pipelines for high throughput datapaths," in *Electronics, Circuits and Systems, 1998 IEEE International Conference on*, vol. 1. IEEE, 1998, pp. 283–286.

[14] J. Sparso and S. Furber, "Principles of asynchronous circuit design-a systems perspective." Kluwer Academic Publishers, 2001.

[15] D. E. Muller, *Theory of asynchronous circuits*. Urbana, Illinois : University of Illinois, Graduate College, Digital Computer Laboratory, 1955.

[16] D. L. Oliveira, G. C. Duarte, G. C. Batista, and N. N. M. Cardoso, "Converting synchronous digital systems to asynchronous systems using local-clock," in *2020 IEEE XXVII International Conference on Electronics, Electrical Engineering and Computing (INTERCON)*, 2020, pp. 1–4.

[17] A. Jain, W. Anderson, T. Benninghoff, D. Berucci, M. Braganza, J. Burnetie, T. Chang, J. Eble, R. Faber, O. Gowda, J. Grodstein, G. Hess, J. Kowaleski, A. Kumar, B. Miller, R. Mueller, P. Paul, J. Pickholtz, S. Russell, M. Shen, T. Truex, A. Vardharajan, D. Xanthopoulos, and T. Zou, "A 1.2 ghz alpha microprocessor with 44.8 gb/s chip pin bandwidth," in *2001 IEEE International Solid-State Circuits Conference. Digest of Technical Papers. ISSCC (Cat. No.01CH37177)*, 2001, pp. 240–241.

[18] M.-C. Chang and D.-S. Shiau, "Design of an asynchronous pipelined processor," in *Communications, Circuits and Systems, 2008. ICCCAS 2008. International Conference on*. IEEE, 2008, pp. 1093–1096.

[19] A. J. Martin, S. M. Burns, T. Lee, D. Borkovic, and P. J. Hazewindus, *The design of an asynchronous microprocessor*, California Institute of Technology, 1989.

[20] A. J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and T. K. Lee, "The design of an asynchronous mips r3000 microprocessor," in *Advanced Research in VLSI, 1997. Proceedings., Seventeenth Conference on*. IEEE, 1997, pp. 164–181.

[21] A. J. Martin, M. Nystrom, K. Papadantonakis, P. I. Penzes, P. Prakash, C. G. Wong, J. Chang, K. S. Ko, B. Lee, E. Ou *et al.*, "The lutonium: A sub-nanojoule asynchronous 8051 microcontroller," in *Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium on*. IEEE, 2003, pp. 14–23.

[22] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods, "Amulet1: a micropipelined arm," in *Compcon Spring'94, Digest of Papers.* IEEE, 1994, pp. 476–485.

[23] S. B. Furber, J. D. Garside, P. Riocreux, S. Temple, P. Day, J. Liu, and N. C. Paver, "Amulet2e: An asynchronous embedded controller," *Proceedings of the IEEE*, vol. 87, no. 2, pp. 243–256, 1999.

[24] S. B. Furber, D. A. Edwards, and J. D. Garside, "Amulet3: a 100 mips asynchronous embedded processor," in *Computer Design, 2000. Proceedings. 2000 International Conference on*. IEEE, 2000, pp. 329–334.

[25] Z. Tabassam, S. R. Naqvi, T. Akram, M. Alhussein, K. Aurangzeb, and S. A. Haider, "Towards designing asynchronous microprocessors: From specification to tape-out," *IEEE Access*, vol. 7, pp. 33 978–34 003, March 2019.